Impact of Software Engineering Research on the Practice of Software Configuration Management

JACKY ESTUBLIER
Grenoble University
DAVID LEBLANG
Massachusetts Institute of Technology
ANDRÉ VAN DER HOEK
University of California, Irvine
REIDAR CONRADI
NTNU
GEOFFREY CLEMM
Rational Software
WALTER TICHY
Universität Karlsruhe
and
DARCY WIBORG-WEBER
Telelogic

Software Configuration Management (SCM) is an important discipline in professional software development and maintenance. The importance of SCM has increased as programs have become larger, more long-lasting, and more mission and life critical. This article discusses the evolution of

This article has been developed under the auspices of the Impact Project. The aim of the project is to provide a scholarly study of the impact that software engineering research—both academic and industrial—has had upon the practice. The principal output of the project is a series of individual papers covering the impact upon practice of research in selected major areas of software engineering. Each of these papers is being published in ACM TOSEM. Additional information about the project can be found at http://www.acm.org/sigsoft/impact.
This article is based upon work supported by the US National Science Foundation (NSF) under award number CCF-0137766, the Association of Computing Machinery Special Interest Group on Software Engineering (ACM SIGSOFT), the Institution of Electrical Engineers (IEE), and the Japan Electronics and Information Technology Industries Association (JEITA).
Any opinions, findings and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF, ACM SIGSOFT, the IEE, or JEITA.
Authors' addresses: J. Estublier, Grenoble University, 220 rue de la Chimie, BP53 38041, Grenoble, France; email: Jacky.Estublier@imag.fr; D. Leblang, 24 Oxbow Road, Wayland, MA 01778; email: leblang@alum.mit.edu; A. van der Hoek, Department of Informatics, University of California, Irvine, Irvine, CA 92697-3425; email: andre@ics.uci.edu; R. Conradi, Department of Computer and Information Science, NTNU, NO-7491, Trondheim, Norway; email: Reidar.Conradi@idi.ntnu.no; G. Clemm, Rational Software, 20 Maguire Road, Lexington, MA 02421; email: Geoffrey.Clemm@us.ibm.com; W. Tichy, Department of Informatics, Universität Karlsruhe, 76128 Karlsruhe, Germany; email: tichy@ira.uka.de; D. Wiborg-Weber, Telelogic 9401 Geronimo Road, Irvine, CA 92618; email: darcy@telelogic.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2005 ACM 1049-331X/05/1000-0001 $5.00

P1: JOL
ACMJ091-01 ACM-TRANSACTION October 29, 2005 3:11

SCM technology from the early days of software development to present, with a particular emphasis on the impact that university and industrial research has had along the way. Based on an analysis of the publication history and evolution in functionality of the available SCM systems, we trace the critical ideas in the field from their early inception to their eventual maturation in commercially and freely available SCM systems. In doing so, this article creates a detailed record of the critical value of SCM research and illustrates how research results have shaped the functionality of today's SCM systems.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Management—*Software configuration management*

General Terms: Algorithms, Experimentation, Management

Additional Key Words and Phrases: Versioning, data model, process support, workspace management, software configuration management, software engineering, research impact

1. INTRODUCTION

In 2000, the authors were invited to mount an effort to document the impact of software engineering research on the practice of software configuration management (SCM). What followed was a lengthy and in-depth debate among the authors to determine what conceptually should be considered impact, how it should be measured, what specific SCM research we believed had impact, and how it should be presented. The cumulative result of this debate is documented in this article, which we believe makes an honest and balanced attempt at describing the impact that three decades of SCM research has had on shaping the functionality of today's commercial and free SCM tools.

SCM concerns itself with controlling change in large and complex software systems. The discipline has been in existence for multiple decades, but has seen a sharp rise in popularity during the last one. Early SCM tools had limited functionality and applicability, but modern SCM systems provide advanced functionality through which it is possible to effectively manage the evolution of many different artifacts as they arise in the development of complex, multimillion line of code systems. This change from small, simple tools to entire SCM environments can be largely attributed to a steady flow of research, undertaken in both academic and industrial settings, that identified and incrementally improved many ideas, approaches, tools, features, and so on. Some of these ideas were simply so compelling that they quickly made the transition to widespread practice. Other ideas, however, never made this transition, or did so only after much additional research and (re)-engineering.

To distinguish the research that did have impact from the research that did not, we extensively discussed what defined "impact." In doing so, we reached the following consensus: for research to have had impact, it must have been: (1) published in the literature (including scholarly papers, patents, user manuals, and technical reports that are publicly available), and (2) incorporated in actual SCM products that are (or were) on the market, commercially or freely. From this definition, we went back and evaluated the functionality of SCM products and compared that functionality against published results. Additionally, we traced these results to their early beginnings, going back through

references and the functionality of older versions of SCM tools. In this process, we also interviewed some of the lead designers of early SCM products, which gave us interesting insights into the importance of events such as conferences, workshops, and other venues at which research is typically presented. Slowly but surely, a picture emerged of a vibrant field in which both academic and industrial research had a significant impact on the practice of SCM.

In this article, we report on this study. We first, in Section 2, discuss in more detail the ground rules underlying our approach to documenting the impact of SCM research. Then, in Section 3, we provide background material to introduce the field of SCM and briefly sketch its historical evolution and importance. In Sections 4 to 8, we describe, per area of SCM functionality, the research that was historically performed and summarize its impact on actual commercial and free tools. Section 9 brings together the impact from all areas of SCM functionality, presenting an overall picture of the volume and critical role of SCM research impact. To further illustrate why impact occurs, the section also takes a closer look at a few successful and failed transitions of research into practice. Sections 10 and 11 finish the report with a word of caution and our high-level conclusions.

2. APPROACH

During the preparation of this report, there was a lively debate among the authors about what defines impact and which research to include. We first looked at the state of the field: a successful, billion dollar industry in software configuration management tools has arisen. SCM tools have become so pervasive that virtually all major projects use them. SCM provides tangible and recognized benefits for software professionals and managers; software development standards even prescribe their use. There are now over a dozen textbooks dealing exclusively with SCM, and most textbooks on software engineering include a chapter on that topic. Software engineering conferences and workshops regularly seek contributions in this area. Finally, there is a lively international community of SCM researchers and an international SCM workshop series.

To measure impact, we decided to focus on the relationship between SCM tools and the research that predated the existence of the tools. Our decision to focus on tools is easily explained: it is difficult to imagine performing any kind of configuration management without using an appropriate SCM tool. Moreover, other contributions such as methods or best practices usually make their way into the tools anyways, sometimes as predefined processes, other times as new features in support of the method or best practice. We believe impact can therefore be gauged by the extent to which SCM tools and their features are used. This includes both commercial and free tools. While a significant fraction of projects adopts a commercial tool, a massive number of projects rely on CVS [2000] and now Subversion [Wheeler 2004], in particular in the open source community.

Identifying whether research had a role in the development of the tools was more difficult. We (the authors) came quickly to the conclusion that citing only academic research would be far too narrow, because corporate research had contributed a great deal of results and ideas. In fact, the software configuration

management community owes much of its liveliness to a healthy and competitive mix of researchers and developers from both academia and industry. Our first ground rule was therefore to take an inclusive view of research.

We discovered quickly that it was futile to determine who contributed "more", academia or industry. Our opinions about what was more important diverged widely, depending on our personal points of view. We will therefore leave the evaluation of the relative merits of research contributions to our readers and to historians. Our goal is to provide an honest and accurate picture of the major research ideas in SCM and show how and where they had an impact.

Given the long lead time for research results to show up in practice, some yet-unused results may have their impact still ahead of them. Thus, we decided to discuss current research even if it has not yet had any discernible impact on practice.

These three ground rules, though important, do not help in identifying relevant research. After some debate we decided to concentrate on publications in the open literature (scholarly papers, patents, user manuals, and technical reports that are widely available, etc.). This criterion is not perfect, since it leaves out unpublished work, in particular results that go directly into products. However, fortunately for the community, research results were vigorously published even by industry, notably Bell Labs, Apollo, Atria, and others. While we cannot be a hundred percent certain, we believe that the research literature contains a publication record of most major research ideas and results in the field of SCM. Of course, there is a chance of some unpublished invention being heavily used in some real product. However, this is unlikely, because competition among the SCM vendors forces the major players to offer comparable functionality and feature sets. By comparing what is known about the functionality of current SCM tools with scholarly published material, it does appear that the major research results incorporated in products were actually published. Conversely, consider what happens to research results that are not published. The probability is high that these results will be forgotten once the people who were engaged in the work move on. The impact of such work is severely limited; after a while, it appears as if it never happened. Others cannot extend the work without actually redoing it. We concluded that basing this report on research published in the open literature would be adequate.

We thus reached the following definition for impact in the field of SCM: for research to have had impact, it must have been: (1) published in the literature (including scholarly papers, patents, and technical reports that are publicly available), and (2) incorporated in actual SCM products that are (or were) on the market, commercially or freely.

Feature creep may invalidate this viewpoint: for impact to occur, features of tools should be used in practice. We observe, however, that SCM tools are complicated, and that vendors do not gain in making them more complicated. On the contrary, simplicity and elegance drive the incorporation of new features. SCM vendors have therefore been vigorous in user testing potentially new features, and have no qualms about not incorporating less useful features or even taking out existing features that are no longer used. Section 7.3 contains one example of functionality that was "taken out", but many other examples exist.

We therefore believe that our position of defining impact based on a comparison of the literature with the features in SCM tools remains valid.

The emphasis on publications and tools leaves out other activities that have had impact. A major impact is caused by people, in particular university graduates moving to industry. Graduates with Ph.D. topics in SCM directly benefit the SCM industry, but even at the B.S. and M.S. levels many computer science students have been exposed to SCM in software engineering courses and projects. Nevertheless, it has been observed that SCM remains an *"under-taught"* topic and that improvements can be achieved in promoting SCM in computer science education [Lethbridge 2000]. Recent graduates may lack the necessary experience, but it is generally acknowledged that young people bring with them fresh and new ideas.

Industry also impacts academia via people. Career moves from industry to academia are relatively rare, but many of the academics in the SCM field have enjoyed long and fruitful relations with the SCM industry, through consulting, sabbaticals, joint projects, or spear-heading start-up companies. Moreover, industrial contacts are a source for new problems to solve and act as a corrective of what is important and what is not.

Last but not least, workshops and conferences have had a significant impact on the SCM community. Given the competitive nature of the software business, it has fallen to the academics to organize vendor-independent meetings in which the latest results in SCM are presented and new, unpublished ideas discussed. It is unlikely that even the authors of this report would all have met without the SCM workshops that brought together researchers and developers in the SCM field for over a decade.

3. BACKGROUND

3.1 A Brief History

Configuration management (CM) is the discipline of managing change in large, complex systems. Its goal is to manage and control the numerous corrections, extensions, and adaptations that are applied to a system over its lifetime. Software configuration management (SCM) is configuration management applied to software systems. The objective of SCM, then, is to ensure a systematic and traceable software development process in which all changes are precisely managed, so that a software system is always in a well-defined state at all times.[1]

What sets SCM apart from other applications of CM (e.g., product data management, content management, . . .) is its focus on managing files and directories. Most current software engineering tools still rely on the presence of the file system view and many software developers still rely on actually seeing and manipulating files in the file system for their day-to-day operation. While a recent trend in SCM is to hide this view from developers through sophisticated user interfaces and integrations with programming environments such as Eclipse

[1]Ideally, SCM also enforces the quality attributes of the software under control, but clearly there is a gap between this theoretical ideal and the current state of the practice.

[2004], the file system view remains (in the foreseeable future) one of the core underpinnings of SCM systems.

The discipline of configuration management initiated in the aerospace industry in the 1950s, when production of spacecraft experienced difficulties caused by inadequately documented engineering changes. Several decades later, software started to pose some of the same challenges in terms of managing change. It became clear that similar techniques could be used to manage any *textual* software system. At first, the relatively primitive method of using different colored punch cards indicated changes. As an alternative, special "correction cards" were used on the UNIVAC-1100 EXEC-8 operating system in the late 60s. Fortunately, software quickly became an on-line entity and hence could easily be placed under the control of a dedicated and automated SCM system. Unfortunately, most knowledge regarding the early SCM systems has disappeared. Scientific conferences for what is now known to be software engineering did not exist, SCM was largely integrated in the operating systems of that time (which are now obsolete), and any documentation describing the early SCM systems is difficult to find.

SCM (re-)emerged as a separate discipline in the late 1970s, with the advent of tools such as SCCS [Rochkind 1975]; RCS [Tichy 1982]; Make [Feldman 1979], and Sablime [Bell Labs 1997]. Each system targeted a specific functionality and focused either on what is now known as "version control" or on supporting an efficient build process for generating an executable program out of source files. In a relatively short time thereafter, these functionalities were integrated in single SCM systems with the following main functionalities: (1) to manage the files involved in the creation of a software product, (2) to track changes to these files in terms of their resulting versions, and (3) to support the building of an executable system out of the files. Fundamentally, this focus has not changed to date.

Pushed by increasingly complex software development and maintenance practices, advances in hardware and software technology, demands by other software engineering tools, and pressures in the ever-changing business environment, SCM functionalities evolved over time. For example, throughout the 1980s, debates raged on the most efficient type of storage and retrieval mechanism, which lead to the development of various, text-based delta algorithms (these can be used inside SCM tools to store differences among versions instead of entire versions; see Section 4.2). However, in the 1990s, management of nontextual objects became much more common and new algorithms were required for efficiently storing and retrieving those objects. By 2000, disk storage became so inexpensive, CPUs so fast, and nontextual objects so common, that the use of deltas became unimportant—many new tools simply use (zip-like) compression.

As illustrated in Figure 1, the context in which SCM systems operate has changed significantly. First, they were used for managing critical software by a single person on a mainframe. This resulted in a need for versioning and building support, which was typically provided by some homegrown system. Use of SCM systems then changed to primarily supporting large-scale development and maintenance by groups of users on a Unix system. This resulted in a need for workspace management, which was quickly provided by newer, more ad-hoc

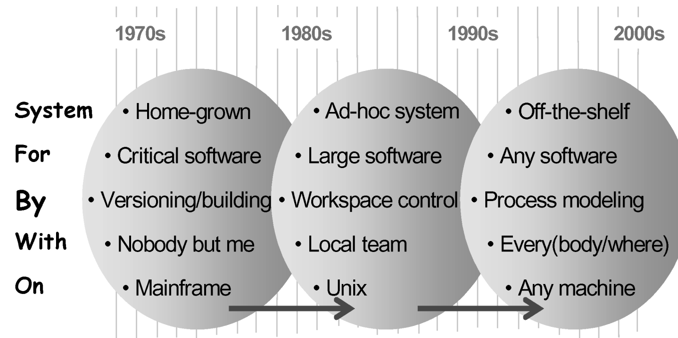

Fig. 1. Evolution of the context of SCM systems.

SCM systems. Now, SCM systems manage the evolution of any kind of software by many different people in many, perhaps distributed locations utilizing many kinds of machines. This often requires explicit process support, which today's advanced, off-the-shelf SCM systems integrally provide. Indeed, SCM is one of the few successful applications of automated process support.

The overall result is that modern SCM systems are now unanimously considered to be essential to the success of any software development project (CMM). The basic tools are pervasive, and the underlying techniques are no longer just found in SCM systems, but also in many other environments such as 4GL tools, web protocols, programming environments, and even word processors such as Microsoft Word. Indeed, the change-tracking facility in Word has been used to produce countless drafts of this article. Furthermore, there is a lively international research community working on SCM, and a billion dollar commercial industry has developed (see Table I; note that this table represent tool purchases only and does not include expenses on consultancy, dedicated staff, and so on).

3.2 SCM Spectrum of Functionality

As compared to early SCM systems, modern SCM systems provide a wide range of high-level functionality. Dart [1991] classified this functionality into eight closely related categories (shown in Figure 2), where a category is identified as a box, and where a box on top of another one indicates that the top box relies on features provided by the underlying one. Each of Dart's category addresses a distinct user need [Dart 1991]:

- ***Components.*** SCM systems must support their users in identifying, storing, and accessing the parts that make up a software system. This involves managing multiple versions (both revisions and variants), establishing baselines and configurations, and generally keeping track of all aspects of the software system and overall project.
- ***Structure.*** SCM systems must support their users in representing and using the structure of a software system, identifying how all parts relate to each other in terms of, for instance, interfaces.

Table I. Market Share of Modern SCM Tools ($M, Heiman 2003)

Worldwide Software CM, 2001, 2002, and 2003 ($M)				Revenue by Vendor	
	2001	2002	2003	2003 Share (%)	2002, 2003 Growth (%)
IBM	325.40	291.19	340.70	36.7	17.0
Computer Associates Intl. Inc.	66.69	110.00	115.00	12.4	4.5
Merant PLC	114.29	102.00	107.79	11.6	5.7
SERENA Software	91.29	89.00	96.52	10.4	8.5
Telelogic AB	56.50	63.09	71.94	7.7	14.0
Microsoft Corp.	35.61	35.49	38.15	4.1	7.5
MKS	24.39	26.99	27.97	3.0	3.6
Borland Software Corp.	47.00	35.00	24.20	2.6	−30.9
Perforce Software	10.29	13.00	16.20	1.7	24.6
Quest Software	2.70	9.50	10.62	1.1	11.8
Visible Systems Corp.	—	2.30	2.50	0.3	8.7
Cybermation Inc.	0.79	1.80	1.50	0.2	−16.5
McCabe & Associates	2.00	2.29	1.48	0.2	−35.5
AccuRev	0.40	0.82	1.25	0.1	52.5
Subtotal	777.33	782.47	855.80	92.1	9.4
Other	91.36	81.95	73.03	7.9	−10.9
Total	868.70	864.42	928.83	100.0	7.5

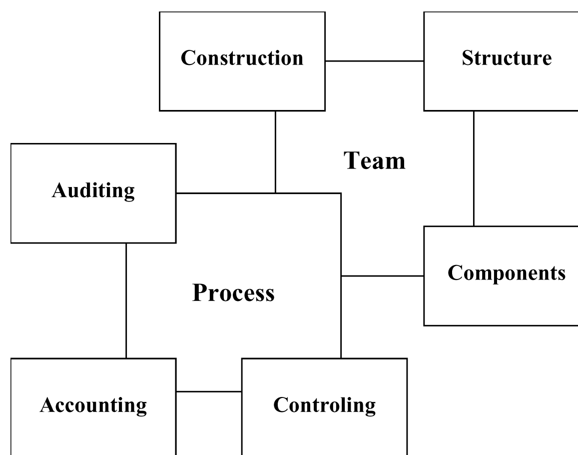

Fig. 2. Functionalities of SCM systems [Dart 1991].

- ***Construction.*** SCM systems must support their users in building an executable program out of its versioned source files, and doing so in an efficient manner. Moreover, it must be possible to regenerate old versions of the software system.
- ***Auditing.*** SCM systems must support their users in returning to earlier points in time and determining which changes have been performed, who performed those changes, and why the changes were performed. The SCM system should serve as a searchable archive of everything that happened.

- ***Accounting.*** SCM systems must support their users in gathering statistics about the software system being developed and the process being followed in so doing.
- ***Controlling.*** SCM systems must support their users in understanding the impact of a change, in allowing them to specify to which products a change should apply, and providing them with defect tracking and change request tools such that traceability exists from functional requirements to code changes.
- ***Process.*** SCM systems must support their users in selecting tasks to be done and performing such tasks within the context of the overall lifecycle process of software development.
- ***Team.*** SCM systems must support their users in closely collaborating with each other. This involves the need to be able to identify conflicts, resolve conflicts, and generally support both individual and group workspaces.

Not surprisingly, it took considerable time and effort to implement all these categories to their full extent; Dart published her paper in 1991, but it was not until recently that high-end SCM systems emerged that provided good support in each category. Now, high-end systems are all slowly but surely converging in terms of their coverage of the spectrum of functionality identified by Dart. While the specific mechanisms with which they provide this functionality may differ, the net effect is the same: users have at their disposal a powerful arsenal of techniques with which to manage the evolution of their software systems.

Of note is that virtually every SCM system is carefully designed to be independent from any programming languages or application semantics. In taking the best of simple representations and algorithms (like the version tree with locking [Rochkind 1975] and simple heuristics (like line-based textual merging [Buffenbarger 1995]), SCM systems are general, yet efficient and powerful tools that have been successful in avoiding the intrinsic complexity of syntactic and semantic issues. SCM systems tend to serve as a framework upon which the tools that need syntactic or semantic knowledge (like syntactic editors or compilers) build. In fact, SCM workspaces often serve as the common ground for those tools. We believe this is a strong contributor to the success of SCM systems: they remain "universally" applicable, while still providing a useful abstraction layer upon which other software engineering tools operate and integrate their results.

3.3 SCM Technical and Research Areas

The categories identified in Section 3.2 are closely related. In fact, the techniques underlying SCM systems often address concerns in more than one category. For instance, a number of SCM systems provide a process engine that, in governing an overall process, also incorporates support for defect tracking, controlling who has access to which artifacts, and when particular tasks are performed. Therefore, we perform our analysis of research impact not based on Dart's categorization of functionality, but on a categorization of underlying

techniques (see below). It is our belief that a much clearer picture of impact can be obtained that way.

From a technical perspective, some basic functionality described in Section 3.2, such as auditing or accounting, is easy to implement and has consequently received little research interest from vendors or researchers. This report therefore concentrates on topics that received significant attention from vendors and/or researchers, and examines the technical approaches underneath those topics.

We partition the technical approaches that address the remaining areas of functionality into three major support areas: product, tool, and process. These three areas represent roughly how the field as a whole evolved, first only supporting file management, then integrating sophisticated tool support, and finally incorporating advanced process control. Each support area can be broken down into a number of technical dimensions, which are the dimensions along which the next sections examine the impact of SCM research. We briefly introduce each dimension here.

3.3.1 *Product Support.* Product management forms the core functionality of SCM. From the beginning, one of the primary responsibilities of SCM systems was to manage the many files that constitute a software system as well as the many versions of those files that result because of changes. The different dimensions of product support are the following:

- ***Versioning.*** Versioning concerns the need to maintain a historical archive of a set of artifacts as they undergo a series of changes, and forms the fundamental building block for the entire field of SCM.
- ***System models and selection.*** Managing a project file-by-file is not very efficient nor very effective. It is necessary to support aggregate artifacts, with relationships among artifacts and attributes that help to enforce consistency in large projects. This is the task of system models, which provide the concept of configurations (aggregates of configuration items that can themselves be versioned). This, in turn, raised the need for supporting a user in selecting exactly to which parts and to which versions of such aggregate artifacts they want access at a given point in time.

3.3.2 *Tool Support.* Given their critical role in the software development process, SCM systems must provide facilities through which other tools (and users) can interact with and manipulate the given artifacts. We examine research impact in this category along two dimensions: workspace control, the primary mechanism used to gain access to artifacts, and building, one of the external tools that generally is considered an integral part of SCM:

- ***Workspace control.*** SCM systems implement workspaces to provide users with an insulated place in which they can perform their day-to-day tasks of editing and using external tools to manipulate a set of artifacts. Important considerations are whether workspaces must support

distributed (or even disconnected) users, and how activities in independent ("parallel") workspaces are eventually integrated back in the SCM repository.

- ***Building.*** Deriving an executable program from a set of source files is the task of build tools. Build tools were initially developed independently from SCM systems. However, derived artifacts have to be as precisely controlled as source artifacts. Efficient building also relies on information regarding which artifacts were modified. Together, this lead to solutions in which SCM systems integrally support building.

3.3.3 *Process Support.* Slowly but surely, the task of SCM systems evolved from managing just files to managing people collaborating in the development and maintenance of software. Initially, this entailed just providing support for change control, but recent SCM systems have incorporated support for general processes.

Early SCM systems already supported a process, namely the predefined way according to which artifacts could be manipulated. Usually this process was hard-wired within the tools, and could not be enhanced or altered without writing extensive scripts on top of the tool (in fact, this is how CVS started: as a set of scripts on top of RCS to alter the process via which artifacts could be manipulated from pessimistic and locking-based to optimistic and merging-based). This deficiency, combined with a desire to also support and capture the rationale about changes (e.g., defect corrections and feature requests), lead to a natural evolution in which SCM system incorporated functionality for the overall process of governing change.

Modern, high-end SCM systems push process support even further. They do not just support change control, but allow organizations to design and enforce general development processes throughout the enterprise. Exactly how this support is provided and integrated with SCM functionality is the subject of our last dimension along which we evaluate SCM research impact.

3.3.4 *Summary.* The technical dimensions along which we examine SCM research impact in the next few sections of this article are closely related to the functional areas discussed in Section 3.2. Moreover, they are not independent. For instance, system models rely upon basic versioning facilities; building relies upon the availability of workspaces; and workspaces, in turn, are populated with a selection (query) over the system model. Many other such connections exist. An elaboration of these relationships is not the focus of this article, and can be found in Conradi et al. [1998]. Our strategy here is to discuss technical contributions in the primary category of applicability. For instance, make [Feldman 1979] and Odin [Clemm 1988], two systems for building an executable program out of a set of source files, are both discussed in Section 7 ("Building") even though they had influence in other categories as well ("System Building" being the primary one).

The following sections present—from a research, practice, and impact perspective—the different areas identified above.

4. VERSIONING

4.1 Classic Versioning

A usual technique to avoid confusion when an artifact evolves, is to issue a new identifier each time it changes. But issuing a new identifier for every change is insufficient to capture any relations that may exist among the resulting set of uniquely identified artifacts. For instance, we may want to record that a given artifact corrects certain defects in a previous incarnation of the artifact. The version control function of SCM records this kind of information by capturing artifacts as configuration items and tracking the desired relations among those configuration items in a structured way.[2] This structured way is termed a *version model*, which specifies the concepts and mechanisms used to structure the version space of configuration items and their potential relations [Katz 1990; Conradi et al. 1998]. Classic SCM systems, such as SCCS (developed at Bell Labs [Rochkind 1975]) and RCS (developed at Purdue University [Tichy 1982]), as well as virtually all major commercial CM systems, collect related configuration items into a set called a *version group* (alternatively called a *version graph*, a *component,* or an *element*) and manage the evolution of these sets. The items in a version group are organized into a directed, acyclic graph (DAG) in which arcs represent "successor" relations. Typically, three types of successor relations exist: (1) the relation *revision-of* records "sequential" or historical lines of development; (2) the relation *variant-of* connects "parallel" configuration items that interchange but differ in some aspect of function, design, or implementation, and (3) the *merge* relation indicates that changes made on several variants have been combined. This is the classic revision/variant/merge version model. For practical reasons, implementations of this model also address the issue of identity: they automatically increment a version number for every new item added to the version group.

Individual team members add new items to the version group by following the *checkout/checkin* protocol. Before commencing work on a configuration item, a developer performs a checkout. The checkout operation copies a selected version from its version group into the developer's workspace. There, a developer can carry out their modifications undisturbed by the activities of other developers (see Section 6). The checkout may also place a *reservation* (a lock) into the version group, giving the developer exclusive rights to a later checkin of a new version as a revision of what was checked out (extending the line of descent). Other developers wishing to modify the same original version can only create a variant of this version and must later merge their changes back into the original line of development. Thus, the reservation protects concurrent developers from inadvertently overwriting each other's changes. By the same token, however, it also restricts the amount of parallel work that may happen and typically leads to project delays when multiple developers must access and modify the same files. A trend in the research community, therefore, became to investigate

[2]A configuration item can pertain to *any* software lifecycle artifact—from requirements, source code, project plans, test data, scripts, and Makefiles (all in textual formats), to derived items like relocatable and executable programs (in binary formats).

mechanisms to relax the strict locking requirement. An early example of such an approach is NSE [Sun 1989], a system in which reservations are optional and represent "soft locks". This advance was quickly adopted by a large majority of modern SCM systems.

Given that a version group may contain hundreds or even thousands of revisions, and given that disk space was scarce and expensive, early SCM systems focused heavily on mechanisms to compress the space occupied by a version group. Typically, this involved using *delta storage* algorithms [Hunt and McIllroy 1996]. These algorithms only store the differences between versions rather than complete copies of each new version. Modern delta algorithms are able to reduce space consumption for source code objects to a small percentage of the full storage needs [Hunt 1996]. Whenever a version is needed in a workspace, the SCM system first reconstructs the necessary item from the differences. SCCS [Rochkind 1975] and RCS [Tichy 1982] are early version control systems that implement delta mechanisms. Variants of the delta techniques of these early systems are still used in modern commercial systems.

Changes in usage patterns drove a second wave of research in delta techniques. Users wanted to not just store text files, but also binary objects such as an executable or an image. The initial response was for SCM systems to leverage standard data compression techniques, such as *zip*, to save space for these kinds of objects. Usable binary delta techniques were eventually developed [Reichenberger 1991]. More recently, research has advanced the state of the art again by developing combinations of delta algorithms and compression techniques (e.g., bdiff [Hunt and McIllroy 1996] and vdelta [Korn and Vo 1995]). These algorithms are now being incorporated in commercial SCM systems (e.g., in ClearCase [Leblang 1994].

Many other approaches to delta techniques have been developed. Context-oriented [Knuth 1984], operation-oriented [Gentlement et al. 1989], semantics-oriented [Reps et al. 1988], and syntax-oriented [Westfechtel 1991] comparisons have been proposed to make differencing and merging algorithms more accurate [Buffenbarger 1995]. Syntactic and semantic comparisons seek to find precise, fine-grained differences so to ensure a merge will produce a consistent result, that is, a source file that compiles (syntactic merging) and exhibits the intended behavior of each of the merged changes (semantic merging). It is easy to prove that syntactic and semantic merging can avoid many of the problems introduced by the use of classic line-based merging [Horwitz et al. 1989]. Despite this clear advantage, commercial SCM systems have not adopted these kinds of algorithms; the need to remain neutral with respect to which kinds of artifacts are versioned prevents the systematic application of syntactic or semantic differencing and merging algorithms. Nonetheless, some of these systems now support the parameterization of which differencing and merging algorithms to use, depending on which artifacts are versioned. In addition, research results have made a transition into industry in a domain different from SCM. For instance, version-sensitive editors (also called "multi-version editors" [Kruskal 1984; Fraser and Myers 1987; Sarnak et al. 1988; Ant 2001]) have used the techniques successfully for several years now.

4.2 Advanced Versioning

Classic versioning limits the relations among configurations items to those in the version graph, that is, revision-of, variant-of, and merge. While certainly providing a solution that is adequate in many situations, another form of versioning exists that provides more flexibility to its users. This versioning model is called *change-set* versioning.

The key idea is to make changes a first class entity, inverting the relationship between configuration items and changes. Instead of ensuring that each version of a configuration item is uniquely stored and accessible (with or without the storage optimization of using deltas as organized in a version tree), the change set approach stores each change as a delta independently from the other changes. A version of the configuration item is constructed by applying a desired set of changes to a baseline version. Consider the following analogy. In classic versioning, configuration items are first-class objects manipulated directly by the users, from which changes can be derived indirectly (i.e., a user must request a diff). In change-set versioning, changes are first class objects directly manipulated by the users, and configuration items must be derived indirectly (i.e., a user must request a composition of a baseline and a set of change sets to get a "version").

Original implementations of the idea, as encoded in the industrial IBM and CDC update systems of the 1970s, relied on the use of conditional compilation techniques.[3] In particular, at checkin time, all changes in a file are tagged with the same label or combinations of such, which (upon checkout) allows the building of different versions of the file by providing a list of labels.

A critical property of change sets is that they are ideally suited to address the issue of multiple files. Whereas classical versioning must rely on the use of system models (see Section 5) to track-related changes in multiple files, change sets naturally lend themselves to contain multiple deltas covering multiple files. Using this approach, it becomes possible to build configurations that never existed before, by combining changes that were created independently from each other (note, through clever use of merging this is also supported indirectly by SCCS, but only at the individual file level—it does not easily scale to multiple files). To do so, however, requires one to also keep track of which change sets are built upon which other change sets. Rather than assuming that all change sets derive from the same baseline, therefore, the SCM tool Aide-de-Camp [1988] popularized the use of change sets that can build upon each other, in effect forming a graph of change sets. Although Aide-de-Camp is the first commercial implementation, the principal designer of Aide-de-Camp revealed that the implementation was highly influenced by the ideas of Hough [1981].

AT&T Bell Labs [1997] did an early implementation of change-set based versioning, called Sablime, designed for telecom applications. The goal was to allow for feature engineering [Turner et al. 1999] by mapping features onto

[3]The concept of conditional compilation is well-represented by *cpp*, the C preprocessor. *cpp* uses two special constructs, "#*if label*" and "#*endif*", that govern inclusion of text. If the condition in the *label* evaluates to *true*, the text between #*if* and its corresponding #*endif* is kept in the file; otherwise, it is removed.

change-sets. The result was that developers were able to configure a system by picking a subset of available functionalities. Internally, Sablime used a modified version of SCCS that could construct a desired version of a file by picking specific deltas out of its SCCS archive file.

Other early change-set systems include PIE [Goldstein and Bobrow 1980], a multi-version editor at IBM [Sarnak et al. 1988], and EPOS, which used the name *change-oriented versioning* [Lie et al. 1989]. EPOS differs from the change-set approach in expressing not only backward predecessors, but also forward visibilities of change-sets [Westfechtel et al. 2001]. The bi-directional approach makes version selection more powerful, and allows the SCM system to provide more assistance to users in composing particular versions, thereby reducing this otherwise unstructured task. Recently, Zeller and Snelting [1997] explored similar approaches in their ICE prototype, using feature logic to express the version rules.

Despite all the attention by the research community, SCM systems based on change-sets do not work well in practice, for a number of reasons. First, deltas sometimes overlap and conflict. Thus, although the SCM system can physically construct any combination of deltas, some combinations may not parse or compile (and it is difficult to predict in advance which versions do and which versions do not). EPOS addressed this problem by allowing "higher-level" change sets that consist of a series of other change sets and a "patch" to make the two work together. This approach, however, has not been adopted into practice yet.

A second problem is that, for certain binary objects (like Microsoft Word documents), there is no way to sensibly combine any deltas, except with the version on which they were based. This means that the change-set approach reduces to the classic versioning approach for these kinds of artifacts.

Finally, in a software project with tens of thousands of artifacts and hundreds of thousands of changes, the full flexibility of the change-set approach becomes unwieldy. That is, it is difficult to reliably identify each version of the system. Furthermore, only a tiny percentage of change-set combinations are actually useful. Commercial change-set systems such as TrueChange [McCabe 2000] and Bellcore's in-house Asgard system Micallef and Clemm [1996] overcome this problem by making heavy and frequent use of baselines to define periodic starting points for new changes. These baselines are usually tested and stable. As a result, a developer needs to only specify a baseline and a few additional changes at any point in time. Although this reduces the number of possible system permutations, it presents a blending of the classic and change-set approach to versioning, vastly simplifying version selection and thereby increasing confidence in the result.

After much debate about the advantages and drawbacks of change-sets, it was eventually understood that their functionality could be approximated in classic SCM systems by leveraging their diff and merge techniques. Called *change-packages* [Wiborg-Weber 1997], this approach has the critical advantage that it uses standard versioning technology, which is mature and efficient, but also allows the flexibility of the change-set approach in cases where it is needed. This is why this approach is rapidly gaining acceptance in current SCM tools.

4.3 Impact

The first significant and influential systems were SCCS and RCS, which came from Bell Labs and Purdue University, respectively. Delta and merge algorithms soon followed, as researched and fully developed in industrial laboratories. Despite the fact that these classic techniques are now more than 20 years old and have not changed much since their initial inception, almost all commercial systems still rely on them. Moreover, these techniques are now so pervasive that basic versioning techniques can be found in many non-SCM oriented tools today. This represents a remarkable achievement, and the impact of the early research is obvious.

The impact of later research on advanced versioning techniques is not so easy to discern. In fact, the most highly advanced versioning techniques (e.g., forward visibilities, feature-based change sets, formalizations in logic), that were a favorite topic in the research community, are not used in today's commercial systems. This is partially caused by the fact that the standard facilities are adequate for industrial users. In many situations, simple SCM tools (like SourceSafe (Microsoft [2000] or CVS [2000])) are sufficient. It is also caused by the fact that these techniques are complex and offer more flexibility than needed. For instance, software development organizations are often concerned with reducing, not increasing, the number of product variants (to save costs and improve quality). In addition, they often are more concerned with instituting proper change management and process control procedures than with deciding which versioning model to use.

Nevertheless, many commercial SCM systems have now added the ability to track changes at the logical level using change packages, albeit under different names (for marketing purposes). A brief survey reveals a plethora of terms: TrueChange uses “change-set”, CCC/Harvest uses “package”, CM/Synergy uses “task”, ClearGuide uses “activity”, PCMS uses “work package”, StarTeam uses “subproject”, and so on. Even lower-end SCM tools such as Intersolv's PVCS and Borland's StarTeam now have the ability to mark a source code change with the corresponding defect report or change request (a primitive way of approximating change packages). Within the next few years, we expect that this kind of traceability will be standard state of the practice for most commercial SCM systems [Wiborg-Weber 1997].

The driving force behind the acceptance of change-sets may be the desire of organizations to manage evolution in a natural, process-oriented way, rather than as a version selection issue. This process-oriented view entails leveraging baselines and capturing defect corrections and feature enhancements as change-sets, while frequently making new baselines. The change package approach matches this process naturally and is, as stated previously, a more practical form of change-sets.

Altogether, versioning in industrial SCM systems on the one hand shows a remarkable technical stability (the ubiquitous version tree) and on the other hand shows a willingness to gradually incorporate some of the advanced services with which the research community experimented. This transition typically only occurs if it does not compromise any of the efficient underlying technology, and

as a result usually rids itself of the complexity still present in the research approaches. Currently, most high-end systems have now adopted features similar to change-sets, but almost three decades were needed for that to happen.

5. SYSTEM MODELS AND SELECTION

5.1 Background

An SCM system does not operate just on individual artifacts, but has to support the evolution of entire configurations consisting of multiple artifacts. Therefore, it must support two critical functionalities: (1) aggregating multiple artifacts into higher-level artifacts, and (2) obtaining access to aggregate artifacts in a workspace such that they can be manipulated. The first problem is addressed in SCM systems by data models and system models, the second by advanced selection mechanisms.

5.2 Data Models

In the database community, typical data models include hierarchical, relational, entity-relationship, and object-oriented models. A file system is also a (simple) data model. In the early 1970s, version control tools such as SCCS and RCS simply relied on the file system to store artifacts as files, but since the mid-to-late 1980s, commercial SCM tools have largely been implemented on top of commercial database systems [Leblang 1984; SoftTool 1987; CaseWare 1989]. Despite this trend, research has certainly attempted to define specific data models dedicated to SCM by incorporating advanced modeling facilities for versioning, selection, and so on [Lamb et al. 1991]. For instance, both Adele [Estublier et al. 1984; Estublier and Belkhatir 1986] and CMA (Configuration Management Assistant) [Ploedereder and Fergany 1989]) added constructs to model these kinds of data. CMA, developed by Tartan Laboratories (a spin-off from Carnegie Mellon University), provided a data model with classes of attributes and relations with which it is possible to represent dependencies, consistency, compatibility, and so on.

Adele [Estublier 1985], originally a research system developed at the University of Grenoble, has developed a series of different data models over time. It uses an active, object-oriented and versioned data model to represent the artifacts of the software system. Objects can represent any kind of entity; attributes can be primitive, compound, or predefined like files, activities, and functions. Relations are independent entities that model associations such as derivation, dependency, and composition. Furthermore, triggers can be attached to objects and/or relations [Belkhatir and Estublier 1987; Belkhatir et al. 1991; Estublier and Casallas 1994].

Several commercial SCM systems also developed more advanced data models, the focus being on object-oriented approaches. Aide-De-Camp (ADC, later ADC Pro and now TrueChange) incorporated a data model with extended attributes and relations, as well as an innovative technique of storing changes independent of the file in which they were made [Aide-de-Camp 1989].

CaseWare/CM (later CCM, then Continuus/CM, and now CM/Synergy) developed an object-oriented data model in 1989 that uses objects, attributes, and relations to model complex systems [Wright and Cagan 1992; Cagan 1993, 1994]. An object's type defines the behavior of its instances using platform-independent, run-time interpreted methods. Types are organized in a type hierarchy with single inheritance. Relation and attribute types can also be defined, and attached to objects dynamically.

5.3 System Models

System modeling initiated in the early 1970s, even before the first SCM systems were developed. Addressing the needs of programming in the large, system modeling was best characterized by DeRemer and Kron in 1976:

> "We argue that structuring a large collection of modules to form a 'system' is an essentially different intellectual activity from that of constructing the individual modules . . . We refer to a language for describing system structure as a 'module interconnection language' (MIL). . . "

In response, a number of MILs were developed in academic settings, including MIL75 [DeRemer and Kron 1976], INTERCOL [Prieto-Diaz and Neighbor 1986], Mesa [Geschke et al. 1977], and Cedar [Swinehart et al. 1986]. Over time, the functionality of these MILs evolved to include facilities for modeling the interfaces of modules as sets of required and provided functionalities, the hierarchical construction of modules out of other modules, the specification of behaviors of interfaces in terms of pre- and post-conditions, and other such relations among modules. MILs eventually evolved into architecture description languages (ADLs) [Shaw and Garlan 1996], which provide object-oriented notations specifically to model the high-level design of a system. Aspects of MILs also appeared in more general design notations, such as UML [Booch et al. 1999] and SDL [Bræk and Haugen 1993].

The advantage to integrating system modeling into the SCM system was quickly recognized by SCM researchers. In particular, they had the ambition to abstract away from the underlying data model (often the file and directory structure) to provide users with an SCM system in which they could describe and manage the "real" organization of the software product. This requires the ability to model entities of all kinds of granularity (e.g., product, subproduct, configuration, component, file, file fragment) with all kinds of different dependencies and relations that together support computing completeness, consistency, and other relevant properties [Jordan and van de Vanter 1995]. Using this information, the benefit is that it becomes possible to model "semantic" information and use it to provide advanced functionality. For example, using the required and provided interfaces of each module, one can analyze the consistency of a particular configuration (as supported by Inscape [Perry 1987]. Interfaces are also used to deal with variability, for instance by allowing alternative implementations of a module that each support the same interfaces in different ways as allowed in Gandalf in the 1980s [Habermann and Notkin 1986] and in Adele from 1982 until 1992 [Estublier et al. 1984; Estublier and Casallas 1994]). This is particularly important in support of the building process, because this process requires

consistent configurations as its input. Some approaches, therefore, evolved to make the system model the central entity along which they supported the build process (i.e., Vesta [Heydon et al. 2001] and SHAPE [Mahler and Lampen 1988].

Over time, it was recognized that evolution was a major problem in using system models. As an implementation evolves, the system model must be kept in sync, and vice versa. Additionally, old versions of the system model must be kept to allow referencing of old versions of the implementation. Therefore, SCM systems using system models evolved to also apply versioning techniques to the system model itself (e.g., PCL [Mahler and Lampen 1988], POEM [Lin and Reiss 1995], and Adele [Estublier and Casallas 1994]).

5.4 Selection

The selection problem complements the use of data and system models: How to obtain a desired set of artifacts in the workspace without having to resort to requesting each artifact individually? Most simple systems, such as Microsoft SourceSafe [2000], SCCS, and RCS, simply default to placing the latest version of the principal variant in the workspace and leave it up to the user to fetch any artifacts that are exceptions to this rule. This is rather limited, as any time one wants to go back to a previous version of the system most artifacts become "exceptions": a version other than the latest must be retrieved manually. RCS allows the association of tags to a particular version and the checkout of versions based on these tags, but this is still limited as there are no mechanisms for specifying any complex conditions or relations.

Much research was performed to allow users to easily populate a workspace with the desired versions of the desired artifacts. Some representative resulting approaches are the following:

- ***Hierarchical workspaces***. First, introduced commercially by NSE (formerly Teamware) [Sun 1989], this selection mechanism relies on the hierarchical structuring of a series of workspaces. Each workspace can specify local versions of certain artifacts, but inherits versions of other artifacts from its parent and grandparent workspaces. This structure typically is used to mimic the structure of the tasks to be performed, with each workspace responsible for a subtask.
- ***General queries.*** Adele [Estublier et al. 1984] supports the association of arbitrary and multiple attributes to different versions of an artifact, and then allows general queries over the set of attribute values to specify a particular configuration with which to populate a workspace. For example, the query *"(status = approved AND owner = Jacky) OR (date > 6.20.83)"* populates a workspace with the latest artifact version that is approved and authored by Jacky, or else that is created later than June 20th, 1983. Queries are interpreted as an ordered set of imperative, preference or default rules.Aggregate selection, leveraging the data captured in the data and system model, is also supported by Adele (and other systems, such as Inscape [Perry 1987], EPOS [Lie et al. 1989], and ICE [Zeller and Snelting 1997]), as is the use of relations in queries. In general, this approach is flexible and compact in its expressiveness.

- ***Leverage change-sets.*** When change-sets are used to store changes, the act of populating a workspaces is simple (in fact, that is the intended use of change-sets in the first place). Systems such as McCabe TrueChange [McCabe 2000] and Asgard [Michaleff and Clemm 1996] therefore use change-set compositions in support of workspace population (e.g., "*baseline 2.5 + bug-fix-283 + bug-fix-2 + feature-12*"). We note, however, that the result is not always as simple—as we discussed previously in Section 4.2.
- ***Rule-based.*** Rule-based systems such as SHAPE [Mahler and Lampen 1988] and ClearCase [Leblang 1994] use an ordered set of rules to specify versions. For example, it is possible to use a layered request that is structured as follows:

  - First, my checked-out versions
  - Otherwise, the latest versions on my branch
  - Otherwise, the latest versions on the main branch

  Rules may be qualified by patterns and use wildcards, and are sufficiently expressive to in essence form general queries such as those supported EPOS, or ICE. The main difference with general queries is that rules impose an ordered structure that simplifies specification of desired behavior as compared to one general query full of and's and or's.

Many more approaches were designed, but all more or less fall in these categories. In all cases, the objective is to make selection as automated as possible and relieve as much of the burden from the user as possible. In most cases, this was achieved (fully automatic selection could take place), but users ended up using only a small portion of the features. A common weakness is that the result of automatic selection is often not easy to predict and, in a workspace with thousands of files, difficult to determine a-posterior. As a result, users tend to be very cautious and avoid any complex specifications.

5.5 Impact

Logically, researchers hypothesized that a more powerful data and system model would allow the SCM system to provide better support for precisely capturing the evolution of the artifacts it manages. This simple idea fostered a vast number of contributions of dedicated data and system models for SCM, models in which everything is versioned, including files, attributes, general relations, configurations, and workspaces [Estublier et al. 1985; Dittrich et al. 1987; Boudier et al. 1988; Thomas 1989; Gulla et al. 1991; Estublier and Casallas 1994]. While this research certainly had some impact, it did not have a great amount of impact on industrial practice and tools to date. The design of both CCM's and ClearCase's data modeling facilities, for instance, were influenced by PCTE [Thomas 1989] and other object-oriented approaches, but neither has adopted the full spectrum of available research technology. An explanation can be found in the following:

- SCM systems, to date, mostly care about file management and system building and not about system specification or detailed behaviors.

- For workspace management reasons, there is a need to define the translation back and forth between the system model and a file system structure. This is far from trivial, if not impossible, when the system model is too rich, at least compared with its underlying file system model.
- A substantial amount of additional effort is required to define and maintain the system model description. Unless the system model can be automatically updated, the additional effort easily outweighs the expected benefits, especially since compilers catch most interface mismatches. This is why MILs disappeared, and why the use of architecture description languages has not caught on much in industry.

Additional impact did happen when market demand for the ability to manage relations among configuration items grew due to the old U.S. Department of Defense DoD-2167A *requirements tracing* specification (and later, due to the ISO 9000 standard and the SEI Capability Maturity Model [Paulk et al. 1993]). Virtually all high-end commercial SCM systems provide hyperlinks and attributes to model such relations. They do not, however, provide facilities for versioning those hyperlinks and attributes. Industrial trials showed this would complicate the versioning task too much. Even though the research systems demonstrated the technical feasibility, thus, practical feasibility turned out to not be possible at this point in time.

Not surprisingly, given the complementary role to data and system models, research in advanced selection mechanisms have had the same limited amount of impact. While some of the mechanisms have certainly made their way into industrial systems (witness the rule-based selection mechanism in ClearCase or the adaptation of a change-set based selection process in CM/Synergy), as a practical matter it is difficult for a user to comprehend a configuration that varies too greatly from a known baseline. Users therefore tend to populate their workspaces with configurations that vary little from a known baseline and frequently build, test, and create new baselines. For this reason, the approach of general queries is not too popular. Analogously, users of change-set SCM systems tend to specify a baseline plus only a handful of changes, and users with a rule-based SCM system typically have only a small number of rules. This creates a manageable level of version selection, and reflects how development teams typically perform cooperative updating and integration.

In general, thus, we can observe that research has produced a number of highly advanced approaches to composition and selection, but that industry and its customers have only adopted slowly and very partially those research results. Even in cases where the SCM system supports advanced versioning selection (e.g., ClearCase), this feature rarely tends to be used to its full extent. However, newer SCM systems are starting to use version selection hidden under other aspects of SCM, such as process control, change management, and system modeling, which seems to be a very promising approach to hiding the complexity and letting the SCM system take care of all the details while the user can simply concentrate on following the appropriate process.

Additionally, the use of UML models is maturing rapidly in industry; and modeling in general has recently received much more attention. A new

community, both from academia and industry, is developing around the OMG Model Driven Architecture initiative (MDA) [Soley et al. 2000; Bézivin 2001; D'Souza 2001], which increases significantly the likelihood to see operational models used in industry in the future. This is a new opportunity for SCM system to get and leverage, for "free", system models from automatic transformations of standard designs.

6. WORKSPACE CONTROL

6.1 Background

Versions stored in the repository of an SCM system are immutable, that is, they cannot be changed. When source files are to be modified, they are usually checked-out into a *workspace*. A workspace may be as simple as the user's home directory or it may be a complex structure managed by a specialized tool and a database. In any case, the workspace typically performs three essential functions:

- Sandbox—the workspace provides a place to put checked-out files so that they can be freely edited. Locking (hard or soft) of the original file objects in the repository may or may not be required.
- Building—the workspace is usually the place where software is compiled and derived objects (binaries) are placed. Since an SCM system generally stores source files in a compressed form (such as deltas), the workspace usually must expand compressed files to full-fledged source files.
- Isolation—a typical project usually has at least one workspace per developer. This allows each user to make changes, compile, test, and debug without interfering with other developers (such as by overwriting source changes or derived objects).

These features are often also available in advanced programming or software development environments, but SCM systems tend to be the central place for managing these facilities, since the services can then be available for all tools in a transparent way.

Classic source code control systems such as SCCS [Rochkind 1975] and RCS [Tichy 1985] do not provide workspace management features. It was soon recognized that this was a rather large void: it was difficult for developers to share the editing of files in the same location. Initially, this problem was overcome by implementing some turn-key scripts on a situation by situation basis. Soon, it was realized that the need to develop these scripts could be replaced by standard features. Workspace management as an integral part of SCM functionality was born. CVS [2000], for instance, was originally implemented as a set of scripts on top of RCS, but later these scripts became an integral part of the source code. CVS provides simple workspace management by creating an isolated directory with source files and Makefiles needed to build a subcomponent. A user commits completed changes back to the SCM system. CVS detects and helps to merge changes that conflict with those made by other users. SourceSafe [Microsoft

2000] is similar, but additionally places copies of newly checked-in files in a common area.

In order to build large software systems, a compiler typically must access many thousands of source objects. These must be efficiently extracted and decompressed from the SCM system and stored in a workspace. On a large project, with hundreds of individual workspaces, this can result in hundreds of thousands of source file copies. Initially, developers addressed this problem by having the workspace tool populate only the directories in which they will be making changes, relying on compiler search paths to find other sources residing in a shared workspace containing a full copy of all sources (these search paths must be kept as part of the workspace configuration!).

Clearly, this is a manual process that does not easily scale. SCM research recognized this problem and has provided several solutions. The first such solution leverages hierarchies of workspaces to improve sharing when multiple configurations are under development, as in Sun/Forte Teamware [Sun 1989, 2000]. A large project is organized as (nested) sub-projects and each sub-project has a variety of individual tasks. An individual user may have a personal workspace for local changes; the sub-project a workspace where individual changes are combined and integrated; and the overall project a workspace with a baseline version of the sources to which all changes from all subworkspaces are ultimately committed. Reconciling changes "up" the hierarchy may be as simple as checking-in a source file, but will often involve merging changes made to the same source files in different workspaces [Estublier et al. 2003].

Recognizing that even in hierarchical workspaces a large amount of copying remains, a second solution to the workspace scaling problem comes in the form of virtual workspaces. As opposed to a physical workspace, which contains copies of all needed files, a virtual workspace copies only those files that the user is actually editing. Called a "view" in ClearCase [Leblang and Chase 1984] and implemented *inside* the host computer's operating system, a virtual workspace provides access to all other necessary source files directly from the SCM system. In particular, when a compiler or other tool attempts to open a source file, the computer's operating system dynamically decompresses and constructs the requested version of the source file. From the user's point of view, the source files appear as an ordinary directory structure of plain files—writeable if checked-out; otherwise, read-only. Virtual workspaces are fast and inexpensive to create so users tend to create quite a few of them when it suits their purposes.

Eventually, it was realized that even this process could be further optimized: using information from previous builds (as captured in bill of materials, BOMs, see the next section), systems such as ClearCase routinely wink in object files from other builds to avoid having to recompile source files that have not changed.

Virtual workspaces are highly reliant on a fully connected network, but most implementations provide an "undock" facility that can be used to create a physical workspace (which can then be disconnected and, for example, taken home on a laptop). Complementing the undock facility is a "redock" facility that synchronizes and re-virtualizes a workspace.

6.2 Impact

First, we note the enduring popularity of workspaces that are constructed simply by copying files and directories into the file system (CVS and Subversion being the prime examples). The simplicity of these approaches and the compatibility with existing tools that rely on the files and directories being in the files system are the primary reasons for the continuing use of this approach. However, for medium- and high-end systems, the concept of a workspace has deeply evolved over the years.

Workspace management involves a large number of topics. At first sight, there seems to be little new as they borrow many concepts and techniques from databases (views, classic transactions, long transactions, etc.). A closer look, however, reveals that research has made a significant impact on workspaces in SCM technology. Two particular research contributions stand out. First, virtual workspaces are now a mainstay in high-end SCM systems. They were brought to development and use within Bell Labs under the name 3DFS [Korn and Krell 1990] and later nDFS [Roome 1992; Fowler et al. 1994]. Earlier research, however, experimented with replacing the IO library (Inversion [Olson 1993]), providing a Translucent File System [Hen 1990], using NFS, or other, semivirtual workspaces (e.g., PCTE [Thomas 1989], CAPITL [Adams and Solomon 1995]). Surprisingly, most vendors claim they were unaware of this early work, and that they (re)developed solutions. Nonetheless, this involved a significant amount of research to make the solutions broadly applicable. It is, however, true that virtual workspaces became popular (and appreciated) only after being available in major commercial products.

Perhaps the most significant contribution of research into SCM workspaces lies in the recognition that it is a unifying technology that brings together file management, versioning, selection, building, change control, and software process. Early research experimented with combining various subsets of these technologies, but the end-result is clear: the workspace is the central technology in SCM systems and its role is crucial in providing users with an effective software development process. Indeed, a modern workspace is created “behind-the-scenes” to perform a particular user-selected task, and all interaction with the workspace contents happens through the advanced interface of the SCM system—hiding the details of selection, versioning, and building. Although the resulting solution may seem “simple”, it represents an important success in applying and making usable (and popular!) advanced technologies. This illustrates how the research had major impact in shaping today’s software engineering practices.

7. BUILDING

7.1 Background

Most SCM systems support the building of software. A change in a common interface description (often captured in an “include file”) can require thousands of other source files to need recompilation. The classic tool for building is Make

[Feldman 1979]. Originally developed at Bell Labs for the Unix platform, Make (and Make-like) tools are still the most widely used tool for system building. Make relies on a textual Makefile, which contains a simple, user-specified system model consisting of file names, composition rules, and derivation relations. A certain derived file is rebuilt (compiled) in correct topological order when Make is called, if a file on which it depends is newer than the current derived file already built in the workspace. Because most changes are limited to a few derived files, Make is impressively fast in rebuilding a previously compiled application.

Over time, various improvements and additions have been made to Make. These include new grammars for pre-processor macros, auxiliary tools for reading source files and generating Makefile dependencies, and parallel distributed building. In fact, there is an entire alphabet of "Makes" (e.g., dmake, gmake, gnumake, imake, pmake, smake) as well as other Make-like tools (e.g., Jam [Wingerd and Seiwald 1997], Odin [Clemm 1988, 1995], and the approach in Water [1989]), which are particularly interesting as they avoid some of the deficiencies of Make. In particular, they do not rebuild based on date, but only rebuild based on the content of a dependency file (see below). They also provide automatic computation of include dependencies, freeing the user from having to specify those by hand. Despite these improvements, Make's core syntax and semantics have remained basically the same for decades, and are a de-facto standard for building. More recently, in the Java world, the Ant system [2000] is slowly changing this fact by providing a build tool that relies on the Java language, introspection, and source code extractors. This creates a higher-level build tool that may replace Make and its siblings in the long run.

7.2 "Same as" versus "Newer than"

Make does not always rebuild when it should. For example, if a user were to checkout a file, change it, build it, and then revert to its original version via an un-checkout, Make would not rebuild the target because it would be newer than the source file now in the workspace (after the un-checkout, the file has as its date the date of its last check-in). This problem is more severe when variant branches or change-sets are used.

One major improvement to Make, employed by virtually all commercial tools, involves a change that modifies the semantics of Make but keeps compatible syntax. In particular, objects are rebuilt only if any of the source versions now seen in the workspace is not exactly the same as those used in the last build (rather than just "newer" than). In order to do this, SCM systems maintain a "bill of materials" (BOM) for each target object built. The BOM lists the exact versions of all source files that went into the target object. ClearCase and Endevor for instance, capture this information though operating system hooks that intercept the build process and record its exact steps. Endevor inserts the data into the derived object file, while ClearCase stores the BOM in its database. As another example, PVCS does not use operating system hooks but scans each

source file for dependencies and writes the BOM into the corresponding object file. Some systems go even further than that, and include the versions of the tools used in the derivation process, as well as the version of the operating system and the hardware configuration.

An analogous problem involves compiler options. If a user were to switch a global conditional compilation flag from, for example, "LANGUAGE=French" to "LANGUAGE=English", Make would see no reason to rebuild. Storing the build script that is used to make a target in a BOM and matching it against the build script currently requested solves this problem.

Note, that the use of BOMs has a secondary effect by providing a reliable audit trail. Using this audit trail, users can repeat the version selection process and building steps at a future time in exactly the same fashion as it occurred in the past. This is required to be able to return to previous releases and perform defect correction.

7.3 Language-Based Building

Programming languages like Ada support semantics for rebuilding that is more fine-grained than an entire file. Depending on the support environment, Ada source code can be rebuilt on a procedure-by-procedure basis. Various attempts have been made to provide similar incremental compilation facilities for other languages, such as Pascal and C++. While such fine-grained, incremental recompilation initially was deemed beneficial, the high-speed of coarse-grained recompilation has made it no longer worth the effort and complexity. An exception can be found in rapid-prototyping environments, where the editor and compiler are integrated and the code is semi-interpreted. Microsoft has made, for instance, the Visual Basic development environment incremental in its support. However, its development environment for Visual C++ still relies on compilation and requires Makefiles.

Smart recompilation [Tichy 1986] is a variant of fine-grained rebuilding that relies on semantically analyzing the changes made in a file to determine if it, and any of its dependent files, must be recompiled [Schwanke and Kaiser 1988; Adams et al. 1994; Bret 1993]. This technique ensures that a minimum number of files are recompiled. But, it requires deep knowledge of the actual programming language and, for performance reasons, any program analysis must be performed incrementally. In practice, simplified implementations of this technique are found only in some of the aforementioned programming environments. The CHIPSY environment for Chill was one of the first systems supporting smart recompilation. Initially, it operated coarse grained, and stopped further recompilations only if a module's interface did not change. This was later extended to be fine-grained, by analyzing the local use of each program entity (symbol) from an imported module interface.

Another form of smart recompilation is a technique called "winking in", which was pioneered in ClearCase. The technique has the advantage of being language independent, reusing binaries in the virtual file system across workspaces to optimize the build process. Builds by someone else, therefore, can help speed up a local build.

7.4 Impact

Commercial SCM systems continue to rely on variants of Make for system building. For example, both ClearCase [2004] and CCM (presently CM/Synergy) [Wright 1990] explored advanced build systems that deviated from the standard Make approach. However, neither uses these facilities now. CCM, in fact, abandoned a “smarter”, more powerful facility in favor of Make. Similarly, ClearCase’s ancestor, DSEE [Leblang and Chase 1984], initially leveraged a full-fledged system model to support a parallel build process. ClearCase, however, abandoned that approach in order to use the industry standard of Make. However, both these systems have modified their respective version of make to include hooks for the automated support of BOMs. Many of the advanced SCM systems on the market today have followed that example, and BOMs are now a ubiquitous and often used feature in SCM systems.

With the exception of BOMs, however, market demand for Make-compatible build systems has severely limited the possible impact of research into advanced build systems. Nonetheless, systems such as Odin and a few “super” Makes had some impact in the field of system modeling. The file dependencies found in a Makefile are nothing more than a formal expression of relations among entities, limited to compile time relationships.

Although explored quite a bit in research SCM systems, semantics-based approaches have not made much foray into industrial SCM systems. We believe this is because SCM systems tend to need general, multilanguage support, and thus far cannot afford to be specific to just a single language. The fate of the syntax-based editors of the 1980s is also telling. Again, language-independent solutions are preferred and have generally more impact. The winking in mechanism originated by ClearCase, for instance, is now also provided by the open source tool ccache, which implements equivalent functionality and is used quite frequently.

8. PROCESS SUPPORT

8.1 Background

The software process has been defined as the sequence of activities performed during the creation and evolution of a software system [Dowson et al. 1991]. Since SCM controls software evolution, it is fundamentally related to the software process.

Process support takes its roots in the belief that software quality and software projects’ cost and delays can be controlled only if the process by which the software is built is controlled [Paulk et al. 1993]. Process support consists of formally describing the intended process (the process model) and executing that process to provide automated help and guidance for developers to adhere to the process in their day-to-day activities.

In SCM, process support primarily targets the control of software changes. Nonetheless, the broader goals of process support in SCM systems also aim at reducing development cost and delay, improving quality, promoting integration with other company processes, increasing repeatability of development

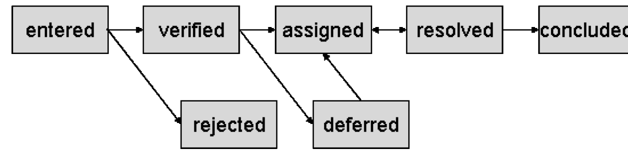

Fig. 3. A change request lifecycle.

successes, and even avoiding blunt development failures (as prescribed by the SEI Capability Maturity Model [Paulk et al. 1993]).

8.2 Change Control

From the early days of SCM, it was clear that change control is a central issue in any SCM process. In the beginning (1960s and 70s), *change requests* (a description of a modified requirement for the product) and *trouble reports* (a description of a malfunction in the product) were handled in paper form. Today, change requests and trouble reports are stored in the SCM repository and have relations to the actual changes; change control is now automated.[4]

The first automated change control systems imposed specific change control processes. Later on, change requests and trouble reports could be modeled as a process using a formalism based on state transition diagrams [Continuus 1993; Whitgift 1991]. The following diagram is an example of a typical set of states through which a change request or trouble report may move.

In general, state transition diagrams describe, for various types of objects, the legal succession of states, the operations valid in a particular state, and which actions produce a transition from one state to another. The underlying technology implementing state transition diagrams in SCM systems relies either on straightforward use of triggers with an associated scripting language (Adele [Belkhatir and Estublier 1987] and CM/Synergy [Wright 1990]) or on a more powerful interpreter (a generic process engine; see below). Most high-end SCM change control tools use definable state transition diagrams, even if the experience shows that customers are rarely defining their own change control models. In any case, the process is typically accompanied by human decision making, such as done by a Change Control Board.

8.3 Generic Process Support

General process support is a research topic that started in the early 1980s in a series of academic workshops, the ISPW series (1984–1999), followed by the EWSPT workshop series (1990–2003) and the ICSP conferences series (1992–1998). These forums were essentially discussions about the real nature of processes ("software processes are software too" [Osterweil 1987]) and their position in software development. It is worth mentioning that the major SCM vendors were present at some of these events.

[4]Numerous commercial change request tools exist.

Many formalisms for process modeling were proposed, and process support became a hot research topic in the 1990s [Derniame et al. 1999; Finkelstein et al. 1994; Di Nitto and Fuggetta 1998]. Although over-enthusiastic promises were common at that time [Katayama 1991], this line of work did identify fundamental concepts and experimented with various techniques. Many papers on process support explicitly focused on SCM as the target application [Estublier et al. 1997]) and generic process engines eventually migrated into SCM systems (e.g., ClearGuide [Leblang 1997]). However, users deemed the full flexibility too daunting and difficult to customize for their own purposes. Early experience with customers showed that process modeling was too much for most people—they would rather buy an established process than build their own. Generic process engines, therefore, eventually were "hidden" by the vendors in favor of predefined, "best practices" standardized processes. For instance, Rational leveraged the full functionality of ClearGuide to provide users with a standardized change process called UCM [Flemming et al. 2003]. This process manages, in concert, the ClearQuest change request database [Magee 2003] and ClearCase workspaces to provide users with a seamless (albeit only limited) process experience.

While the primary domain of process modeling in SCM remained change control and trouble reporting, experiments also took place that used process models to manage workspaces. In particular, the experiments addressed the needs of managing concurrent engineering activities [Estublier et al. 2003].

8.4 Impact

Change control was present and mature in many SCM tools by the early 1990s [Dart 1991], and is now found in all but the simplest SCM tools [Estublier 2000]. Some support only weak associations among changes and change requests, others provide a tighter integration by storing change requests in the same repository as the SCM data and relating the change requests to the objects that changed [Cagan and Weber 1996]. Little research is expected in the future with regards to change control. Based on its wide acceptance and implementation, it is considered a mature technology. The area of growth concerns a better integration with other tools, such as Project Management, Call Tracking (Help Desk), and Customer Relationship Management (CRM).

Since the beginning, generic software process research has mostly focused on activity-based modeling. Here, activities play the central role and process models express data and control flows among the activities. This contrasts with the product-based system modeling traditionally used in SCM, and the current state of the practice does not integrate the two models. The impact of generic process modeling is, thus, relatively low. However, the success of change control as a process-based technique within the field of SCM has lead to the widely accepted observation that SCM is one of the software engineering domains in which process support has proven to be most successful [Conradi and Westfechtel 1998].

An interesting addition to this discussion concerns something that happened at SCM9 [Whatihead 1999]. Attendees were asked to list what they considered

Table II. Landmark Contributions with Great Impact

<table>
<tr><th></th><th>Academic Research</th><th>Industrial Research</th><th>Industrial Product</th></tr>
<tr><td>1972</td><td></td><td>SCCS (Bell Labs)</td><td></td></tr>
<tr><td>1976</td><td></td><td>Diff (Bell Labs)</td><td></td></tr>
<tr><td>1977</td><td></td><td>Make (Bell Labs)</td><td></td></tr>
<tr><td>1980</td><td>Variants, RCS (Purdue University)</td><td></td><td></td></tr>
<tr><td>1980</td><td></td><td>Change-sets (Xerox Parc)</td><td></td></tr>
<tr><td>1982</td><td>Merging, and/or graph (Purdue University)</td><td></td><td></td></tr>
<tr><td>1983</td><td></td><td colspan="2">Change-sets (Aide-de-Camp)</td></tr>
<tr><td>1984</td><td>Selection (Grenoble University)</td><td></td><td></td></tr>
<tr><td>1985</td><td></td><td colspan="2">System model (DSEE)</td></tr>
<tr><td>1988</td><td colspan="3">First International SCM workshop</td></tr>
<tr><td>1988</td><td colspan="2">Process support (Grenoble University)</td><td></td></tr>
<tr><td>1988</td><td colspan="3">NSE Workspaces (Carnegie Mellon University; Sun)</td></tr>
<tr><td>1990</td><td></td><td>3DFS, nDFS virtual file system (Bell Labs)</td><td></td></tr>
<tr><td>1994</td><td></td><td colspan="2">Virtual file system (ClearCase)</td></tr>
<tr><td>1994</td><td></td><td colspan="2">MultiSite (ClearCase)</td></tr>
<tr><td>1996</td><td></td><td>Activity-oriented SCM (Asgard, Bell Core)</td><td></td></tr>
<tr><td>2000</td><td colspan="2">WebDAV/DeltaV (University of California, Irvine, Microsoft, ClearCase,. . .)</td><td></td></tr>
</table>

the most appreciated and the most deficient features in SCM. Surprisingly, process support was number one in *both* categories! This indicates the progress and the frustration that process support has introduced in SCM.

9. SCM EVOLUTION AND IMPACT

We now summarize the preceding discussion with an overall view of the evolution and impact of SCM research. We first discuss the set of contributions that we believe has had unequivocal impact, and present some observations regarding the context of those contributions. We then examine in more detail why we believe some research had impact and other research did not, by examining some of the success stories in the field and some examples of research that has not made the transition into industry. We conclude this section with some of our expectations for future SCM research and practice.

9.1 Landmark Contributions

Table II presents a list of what we believe are the landmark contributions in the field of SCM, that is, contributions that have had clear and direct impact on SCM tools and practice. It is these contributions that have the greatest name recognition in the field, significantly influenced large numbers of future contributions, and increased the bar of what is considered to be standard SCM practice.

Several observations are in place about the list we identified. First, it is a conservative list. While we recognize that many significant contributions exist, we only included those for which we can unambiguously point to direct and strong influence on the current state of the practice, especially in the form

of the tools that are currently available. After all, this is the mission of this article.

Second, we recognize that these contributions were not made in isolation. The context in which the research was performed shaped the contributions. We can, for instance, point to research tools and prototypes such as Inscape [Perry 1987], COV [Lie et al. 1989], Odin [Clemm 1988], and PCL [Mahler and Lampen 1988], which have been widely recognized as very influential in the academic community, but for which there is little-to-no evidence of direct industrial impact. Nonetheless, the context, discussions, and mutual influences among the research typically lead to these kinds of contributions having had indirect influence and impact.

Similarly, a number of papers such as DeRemer and Kron [1976], Feiler [1991] and Dart [1991] have set "theoretical ideals": they demonstrate the level of support (and the spectrum of functionality) that is ultimately possible and therefore they challenge the community to achieve their vision. However, the theoretical ideals are generally not practically achievable in the form in which they are proposed. Much industrial research is needed to make the ideas work in practice; sometimes this happens, sometimes it does not (see Sections 9.2 and 9.3).

The context is further formed by previous research. Academic and industrial research contributions all cite previous publications. It can therefore be presumed that the research presented in those previous publications had influence on the idea in some kind of form. Similarly, industrial releases of systems are based on existing contributions—either existing SCM systems or academic contributions. It is interesting in this regard to examine a patent issued to the makers of DSEE (the leading SCM system in the 1980s, eventually becoming ClearCase, a current market leader) and an early scientific paper authored by the makers of what is now CM/Synergy (also a current market leader). The citations in the patent and paper are presented in Appendices A and B, respectively. We can clearly observe that the work was influenced by significant academic contributions (consider the listing of Gandalf and CEDAR in the patent for DSEE, and the listing of Inscape and NSE in the paper describing for CM/Synergy), further illustrating that there certainly is a healthy exchange between (academic and industrial) research and the industrial products that result.

Third, we observe that even though we made an attempt to separate academic research, industrial research, and industrial impact, this is not always possible. Only some industrial research is performed in a truly research-oriented setting (i.e., SCCS, Make, nDFS), but most industrial research is performed in the context of actual needs from an accompanying industrial product (e.g., the system model underlying DSEE, ClearCase MultiSite). Similarly, the distinction between academic research and industry (whether research or product development) is not as clear either. For example, NSE was developed as a collaboration between CMU and Sun Microsystems. As another example, research on Adele was performed in an academic setting, but the Adele team behaved much like a company in also selling the product. The bottom line is that it is not possible to always strictly separate where research is performed.

We consider this a strength in our field and one of the primary reasons why significant impact has occurred: because this kind of cross-pollination exists in the field, problems addressed by researchers tend to have a high level of industrial relevance.

Finally, we note that each of the contributions listed in Table II did not necessarily have immediate impact on industry. Often it took time for an idea to develop, for the market to be ready, and in general for the conditions to be in place for the transition to occur from research into industrial practice.

9.2 Some Successful Transitions

As summarized at the end of each of the preceding sections, SCM research has had significant impact in shaping the current SCM tools and their usage. A question that remains, though, is why certain research contributions have such strong impact and others do not. In this section, we take a look at example areas in which research was successful in having impact; the next section discusses several examples of research that did not. While we certainly cannot claim we have come to a complete answer, we believe the examples illustrate several critical factors that are in play when it comes to the successful transition from research into industrial use.

9.2.1 *Change Sets.* As opposed to some research that had immediate and long-lasting impact (SCCS, Make, RCS), change sets represent an example of a research contribution that took quite some time to have impact. As discussed in Section 4.2, changes sets were first introduced to the SCM market by Aide-de-Camp [1988], and then were heavily studied, extended and formalized by academic research [Lie et al. 1989; Zeller and Snelling 1997; Westfechtel et al. 2001]. For long a curiosity and looked upon as an interesting but completely nonpractical solution, change sets slowly have become a standard feature in high-end SCM systems. However, this has occurred in a much simplified form under the name *change package*, which refers to the practice of associating a change set with the concept of a task. Indeed, focusing on the concept of a task has two major consequences. First, the grain for a version is the complete workspace, thus "naturally" a change-set. Second, a selection is based on a known release, and a version is a change with respect to that release. This represents a much more natural way of working and explains why the simplified change-set approach, in its change-package implementation, is gaining acceptance and why the more general change-set approach remains esoteric and unused.

Here, thus, it was the transformation of an initially good idea to match an already accepted practice of undertaking work on a task-by-task basis. Once this mapping had been established, and once the underlying technological support had been sorted out, the idea could transfer.

As the current trend towards hiding low-level mechanisms (revisions, variants, branches, and so on) and relying on higher-level concepts (workspace, task) continues, we believe more-and-more of the advanced, well-researched features of change sets and other advanced versioning and selection mechanisms will make their way into commercial SCM systems. Customer

readiness and a need to incrementally adopt new features make this a slow process.

9.2.2 *Process Support.* Process support in SCM systems is a noteworthy success, and has become one of the major selling arguments of vendors as well as one of the major expectations of clients [Conradi and Westfechte 1998]. SCM is one of the very few fields in which process support proved extremely successful and even critical.

This success, however, did not come "easy". The incorporation of powerful process engines in SCM systems did not succeed initially, because users found it too cumbersome and difficult to define their own processes properly. An example of this is ClearGuide [Leblang 1997], which is a powerful, generic process engine integrated with ClearCase. Despite its flexibility and advanced support for modeling and enforcing virtually any kind of process (allowing users to tailor the SCM system to their own needs), most users of ClearCase prefer its standard, out-of-the-box UCM process, which is marketed as embodying best practices. It took a number of years for this realization, and the advent of UCM, to happen. Initially, ClearGuide was heavily marketed yet customers did not purchase it.

This is another excellent example of technology "ahead of its time". As with change sets, the abilities afforded with generic process engines simply were greater than what customers were prepared to handle at the time (and even now). Progress, both in customer maturity and in simpler process support concepts and interfaces, will be required before generic process support will be widely used.

9.2.3 *Differencing and Merging.* Initial SCM systems relied on traditional differencing and merging technology that involves the comparison of lines of ASCII text [Hunt and McIllroy 1976; Meyers 1986]. While these algorithms were not invented within the SCM domain, Section 4 discusses a variety of SCM research that has since been undertaken to improve on these simple line comparison algorithms. This research can be categorized into two classes: incorporating more semantics to provide more accurate differencing and merging technology, and extending the differencing and merging algorithms to handle binary artifacts.

Research in both classes can be described as attempting to provide better functionality rooted in strong theoretical foundations. Semantic-based merging requires dealing with conflicts at the language-level, differencing and merging of binary artifacts requires algorithms that can simultaneously detect block moves and compress any kind of file. Despite this similarity, despite a serious amount of research devoted to both topics, and despite the fact that both research directions provide serious advantages to customers, an interesting juxtaposition exists: whereas semantic-based differencing and merging has had virtually no impact, new binary differencing and merging algorithms are now an integral part of virtually all leading SCM systems.

The reason for the success of binary delta algorithms is twofold. There is a direct and customer-driven benefit, in terms of space and the ability to version

binary configuration items efficiently. Second, the delta algorithms practically operate "under the hood", as there is no visual difference to the customers in terms of the way in which they interact with the SCM system. For semantic-based differencing and merging, these two factors are not present: customers are currently not asking for this level of support, and their use of the SCM system would change should the support be available. It is clear that only when the research advances do not stray too far from the general and established objectives will impact occur.

9.2.4 *Distributed and Remote Development.* Distributed and remote development represents an area in which industrial research clearly took the lead in solving the problem. Initial academic approaches focused on adding a simple web interface to an SCM system to provide remote access to a repository with artifacts. A first breakthrough shortly thereafter was the addition to CVS of a client-server protocol. This rather simple extension enabled the seamless creation and use of wide-area, distributed workspaces, a concept that is now in widespread use and constitutes a major supporting technology for open source software development.

ClearCase then researched and developed MultiSite [Allen et al. 1995], a solution that relies on peer-to-peer repositories that are periodically synchronized with each other. Similar solutions are now in use in almost every high-end SCM system and research on the topic has quieted down as the solution is satisfactory, resulting from well thought-out industrial research, as well as from reusing timestamp, synchronization, and backup techniques from the fields of databases and distributed systems.

It is interesting to note that SCM techniques have found their way in other distributed settings and solutions as well—providing yet another kind of success story. As a first example, website management closely resembles software development. It involves rapidly changing resources that are authored and managed by a multitude of people (possibly in different geographic locations). Moreover, the periodic releases of the resources must be closely controlled. Although it is curious that none of the SCM vendors is among the current market leaders of content management tools (such as BroadVision, Vignette, ATG, Allaire, InterWorld, Interwoven, BSCW, and DreamWeaver), it is not surprising that these tools incorporate SCM techniques to manage the evolution of web sites. They use a different data model, but the basic principles and techniques are still the same as originally devised for SCM.

WebDAV and DeltaV [Whitehead and Golan 1999; WebDAV 1999, 2002] provide a second example of a success story in the relm of the web. WebDAV is a protocol that extends HTTP with distributed authoring facilities. DeltaV is a second extension that adds advanced versioning capabilities. SCM research has had a definite impact in this arena: early incarnations of the interface functions in the protocol of WebDAV were partially inspired by NUCM [van der Hoek et al. 1996], and DeltaV is actively being developed under the partial leadership of SCM vendors. Clearly, the field is having its impact and the two standards incorporate many of the good practices that have been researched and developed over time.

As to why impact occurred, distributed SCM support seems to have been the result of both a push and a pull situation. Customers quickly moved from centralized development to distributed development, and needed as strong SCM support in the distributed setting as they did in the centralized. At the same time, vendors (and standards groups) managed to develop new technology that supported these kinds of situations better than before—allowing them to push their solutions onto the customers through aggressive marketing techniques and promotion of the standards.

9.2.5 *Summary.* In looking at these and other examples of successful impact, we conclude that successful impact seems to be possible when three factors align: (1) a desire of customers to have a certain issue addressed, (2) the ability of SCM tool producers to provide the needed feature, and (3) a readiness of the customer to accept the potentially additional burden that comes with using the new feature. This last factor is often overlooked, but is of critical nature. For instance, some customers may not be able to absorb the extra cost of hiring a full-time database administrator that is needed when they want to use an advanced SCM system. As another example, despite significant potential benefits, most customers will not use a system model if the dependencies among artifacts must be manually specified and maintained. The apparent burden simply is too high. New SCM features, thus, must be brought into an organization slowly, incrementally, and in a carefully managed manner with appropriate automated support. Only when all of these factors are met, will there be strong potential for eventual success, and thus impact, of newly researched and developed SCM features.

9.3 Some Failed Transitions

SCM research also has produced a number of ideas that have not been able to succeed in making the transition to industrial practice. Here, we examine three such ideas and discuss why we believe that, even though the ideas are conceptually sound and useful, they did not successfully transition to commercial or even free SCM systems—despite serious attempts at doing so.

9.3.1 *Semantic-Based Recompilation.* Semantic-based recompilation, as discussed in Section 7.3, is clearly an appealing idea: it reduces the amount of work needed to recompile a system to an absolute minimum. Much research was performed on the subject, a survey of which is presented in Adams et al. [1994]. Industrial trials of the subject were performed, but in the end, none of the more successful commercial or freely available SCM systems have features that resemble smart recompilation. The reasons are twofold:

(1) Semantic-based recompilation techniques require implementations of SCM systems that are language dependent. Many new languages are forthcoming, and the "popular" language of choice changes frequently. Supporting all the languages would require a significant effort on behalf of the vendors, an effort that was too costly compared to other, more pressing needs in advancing the functionality of their products.

(2) In hindsight (certainly not at the time!), semantic-based recompilation algorithms are not needed with today's fast hardware. In the general case, brute-force, entire recompilation "as is" is fast enough. So even if some marginal products were supporting semantic-based recompilation, that need has disappeared and the impact was merely temporary.

Certain exceptions exist in the form of extremely large-scale systems for which an entire recompilation can take days. Semantic-based recompilation may be a good solution there, but a much more straightforward solution is typically employed: specialized build systems that utilize multiple machines in parallel tend to achieve satisfactory performance results and reduce the total recompilation time from days to hours. Similarly, the winking in approach of ClearCase provides pragmatic, language-independent benefits in this regard. From an SCM point of view, thus, semantic-based recompilation has not had very much impact.

Nonetheless, semantic-based recompilation cannot be labeled a failure. The idea is appealing and has started to be incorporated in language-dependent programming environments. In these environments, syntactic information is available "for free". The Ada and Chill programming environments, for example, were among the first to adopt semantic-based recompilation techniques [Bret 1993]. Now, most modern programming environments, including the popular Microsoft Studio for Visual Basic, rely on these techniques. Semantic-based recompilation is therefore an area of research that has found its way to industry in another domain than SCM.

9.3.2 *Advanced System Models.* Extending and generalizing versioning capabilities has been a core topic of SCM research since its early beginnings. Much work has been dedicated to creating advanced system models and associated selection techniques, including a number of formalizations of these approaches [Bielikova and Navrat 1995; Navrat and Bielikova 1996; Zeller and Snelting 1997]. Yet, the system models used by today's commercial SCM systems only capture the files and directories that represent a software product, the compile time dependencies, and a small set of attributes. This leads to an interesting disparity. From a researchers' point of view, research has continuously improved the state of the art by offering new or alternative modeling capabilities. From a practitioners' point of view, however, a sizeable portion of these approaches are overkill. They provide more power than is actually needed, at the cost of extra complexity and reduced efficiency.

Two simple but important reasons explain this discrepancy. First, no commercial database exists that can support these kinds of advanced models. And building such a database, either from scratch or on top of a commercial relational or object-oriented database, is a daunting undertaking [Westfechtel et al. 2001]. Second, the models are simply too complicated. We have offhand and informal information that some vendors did provide, to select customers, prototyped new versions of their SCM systems with extended relation versioning capabilities. Development of these prototype efforts into full-fledged system functionality was abandoned, since these early experiences showed that the

task of managing the relations was too cumbersome and that users eventually simply did not use the feature. Unless automated techniques are developed that support users in updating and maintaining advanced data and system models, they will probably never be put into practice.

9.3.3 *Generic Platform.* SCM is meant to provide a generic, language-independent platform that can handle any kind of software artifact. The focus of SCM research, however, has largely been on managing source code only, leading to platforms that tend to be more specific than desired. Limitations in the data and system model, even though necessary from the functional perspective described above, make it difficult to manage complex structures. Too much additional manual work is needed to capture the extra information in, for example, supporting product data management (PDM) [Estublier et al. 1998]. Similarly, existing SCM tools cannot effectively support web site management or document management. Even integration with existing tools in those domains that already include some of the basic SCM services (e.g., data and version management, process management, rebuilding) has proven to be difficult.

The reason for research in these areas to not have had much impact seems to be a combined technical and organizational reason. Demand certainly exists [Crnkovic et al. 2003], but technically it is a daunting challenge to build SCM support that handles all these different kinds of artifacts and their relations, yet maintains ease-of-use. This requires advanced system models with automated support procedures for maintaining consistency and up-to-date relations among elements in the model—something discussed in the previous section as too technically challenging for the moment. Organizationally, companies are not ready yet to fully delve into all aspects of managing these kinds of situations.

There is hope, however. The driving force is that software is increasingly an integral part of virtually any complex manufactured object. The need to consistently and conjointly manage software and hardware in such situations, combined with the fact that current makeshift solutions are ineffective and tend to be disjoint, may force SCM researchers to reconsider the situation and initiate a renewed research program into providing generically-applicable facilities. Some research has started to look into this direction, but much additional research will be required to reach a satisfactory result.

9.3.4 *Summary.* The reasons why some SCM research has had little-to-no impact varies, and can lie in the level of complexity required to master an idea or in the level of effort required by the customer to use a feature. Furthermore, a typical customer may not see a need for a feature (at that particular point in time) and sometimes vendors think a feature is technically too difficult to implement or outside the scope of SCM. Not surprisingly, in all these cases, industry will ignore the idea. In some cases, changes in technology or in customer practices make it reasonable to later (re)introduce functionalities making use of the idea. Even then, industry must find a way to transform the (typically complex) idea into an easy-to-use feature that hides most of the actual complexity. None of these conditions are granted nor can they be easily forecasted, leading to a typically cautious approach in adopting new ideas.

9.4 What Is Next?

SCM research has addressed many different topics and it is fair to say that, by now, the basic principles, concepts, and techniques are set. Consensus also seems to be emerging regarding more advanced functionality, evidenced by the fact that most high-end SCM systems are closing in on satisfying the spectrum of functionality laid out by Dart [1991]. Nonetheless, many research issues remain to be addressed [Conradi and Westfechtel 1999]. In particular, the field as a whole is now sorting out how to better fit in the overall picture of software development, rather than always assuming being able to provide a standalone application that somehow seamlessly fits with existing tools, approaches, and practices.

This research is breaking two fundamental assumptions that underlie current SCM systems: (a) the focus on managing the implementation of software and (b) the basic philosophy of SCM being programming language and application independent. Breaking the first assumption requires careful management of artifacts produced earlier in the lifecycle (e.g., requirements and design) and later in the lifecycle (e.g., deployment, dynamic evolution and reconfiguration). Breaking the second assumption involves the integration of SCM functionality into particular environments (e.g., integrated development environments) and representations (e.g., product line architectures).

At the forefront seem to be the issues of unifying SCM and PDM, managing component-based software development, and understanding the relation between SCM system models, software architecture, and component-based software engineering [van der Hoek et al. 1998a, 1998b; Crnkovic et al. 2003; Westfechtel and Conradi 2003]. Nowadays, an increasing amount of software is no longer developed from scratch or in isolation, but rather assembled for combined hardware-software systems from multiple components authored by multiple different organizations. To survive, SCM systems must evolve to address the new concerns raised by these situations and stay abreast of new developments, trends, and technologies.

A side-effect of the popularity and long-term use of SCM systems has been the recent discovery that they serve as an excellent source of data on all sorts of aspects regarding the software development process and products that result from it. A new field, mining software repositories [Zimmermann et al. 2004], has sprung up that is concerned with such issues as reverse engineering, software understanding, finding historical trends, discovering actual processes versus prescribed processes, and leveraging past activities in providing guidance for present activities. Without SCM systems, this entire field would not be in existence today.

Of course, we can in no way predict the future of SCM. It may be that sorting out some of these issues turns out to be trivial, not relevant, or far too difficult for practical application. It may also be that other research trends and other communities, such as content management, CSCW, operating systems, advanced automated cooperation policies for distributed and mobile systems, and others, turn out to be more successful in making an impact. This, however, is the way of research. Providing an answer to the above questions is

what is important, even if those answers sometimes close, rather than open, doors.

10. DISCUSSION

A report like this can easily generate a lot of dissention. Both industry and academia often claim ownership of ideas, and the factual tracking of origins and traces of impact over such a long period of time with so many different players is virtually impossible. This report is only an attempt at documenting the impact of research in the field of SCM; an attempt that we hope comes as close as possible to historic reality. Based on our discussions with many individuals who have contributed to the field, on careful tracing of ideas in papers, and on general observations of trends in SCM, we hope we have done justice to the history of the field.

Agreement between industry and academics as to where ideas originated is generally easy to find in areas of direct impact. For example, published techniques that are used "as is" in industrial products can easily be traced. RCS, Make, and delta techniques are examples of that category. SCM is probably one of the few software engineering domains in which contributions in this category form the core of most, if not all, of today's tools—even those residing at the high end of the market.

Agreement is more difficult to find in those cases where published techniques are re-implemented, augmented, and polished for use in real-world settings with demanding customers. Over time, vendors tend to feel the original idea was trivial because of the significant amount of work needed to really make the idea work. Merging techniques fall into this category: SCM vendors have invested very significant amounts of time in creating merge tools that are effective at the level of scale and user friendliness required.

Often, in fact, a company performs its own research to facilitate the transition from research to practice. In adopting a basic idea, but implementing its "flavor" in a manner adjusted to what is considered useful or accepted by actual practice, the original idea is significantly transformed to address such issues as reliability, scalability, usability, and efficiency. Data models, selection mechanisms, and process support all fall into this category. Change-sets are another excellent example. Although the idea has been around for over two decades, only now major SCM vendors are switching their products to be based on change packages, a more realistic and practical incarnation of change-sets.

In some cases, it is industrial research that is ahead of academia, as shown in Table II. Although regularly overlooked, industrial research is as important as academic research and therefore receives significant credit in this report.

For ourselves, a caricaturist but frank starting point in our discussions was that the researchers in the team claimed ownership of virtually all ideas (dismissing industrial tool realization as "engineering") and that the vendors in the team argued they had to (re)invent everything they needed from the ground up (dismissing research concepts, ideas, and architectures as "academic"). From the intensive discussion that followed, a much more balanced perspective emerged. We eventually came to the conclusion that both research

and development require engineering *and* creativity, and that tracking down where ideas came from and directly establishing what was influential on what is virtually impossible. However, broad trends could be identified, and the influence of research on those trends could certainly be established. Based on this, we also rapidly agreed that research has been fundamental in the success of SCM and that industrial research from corporate research laboratories and vendors had a definitive influence. Accordingly, we have attempted to document the flow of ideas, based on evidence from publications and tool realizations; an attempt that we hope comes as close as possible to reality.

Finally, although conferences, workshops, and personal interactions undoubtedly play a tremendous role in research transition, it is impossible to quantify their impact. Nonetheless, continual attendance of the SCM workshop series by chief architects of some of the most influential SCM systems, the transition of academic researchers to industry and vice-versa, and anecdotal evidence brought forth in our personal interviews indicate that these kinds of interactions are absolutely necessary for any kind of research impact to occur. Conferences and workshops create a community of researchers and practitioners, raise new issues to be addressed, set high-level expectations for new directions, and, in the case of SCM-1, have set the standard terminology still in use today [Winkler 1988].

11. CONCLUSION

SCM is arguably one of the most successful software engineering disciplines. It is difficult to imagine this kind of success would have prevailed without research fueling continuous innovations. This report demonstrates that the impact of this research, whether industrial or academic in nature, is undeniable—most fundamental techniques underlying current SCM systems were first published in one form or another.

Like any other field, SCM research has had its successes and failures. Certain ideas are universally adopted, others have had limited impact, and yet others never saw fruition. Timing has been critical: whereas some contributions could immediately be related to practical, day-to-day problems, others were too early for their time and not practically relevant for the problems then at hand. Nonetheless, the actual evolution of the field demonstrates that most of those ideas eventually were useful. As shown by the remarkably long delay in the adoption of change-sets, it is often market readiness, along with substantial rework to make the idea practical, that determines success. In the long term, however, a significant fraction of ideas have trickled through.

The academic research community has contributed many ideas to the field of SCM. But, more importantly, it has provided a forum for the publication and discussion of ideas of both academic and industrial research. An international workshop series dedicated to the topic, as well as many general conferences such as ICSE and FSE, enabled people from academia and industry to interact and exchange ideas. While over time the set of important ideas may be changing, the need for an active research community remains constant and essential.

The SCM basic concepts and technologies may have been settled, but substantial work remains to be done. In particular, the field as a whole is now sorting out its relation to other domains, such as, among others, product data management, component-based software engineering, CSCW, distributed and mobile computing, and software architecture. We look forward to the advances that will come from this research, and are proud to be a part of a field with such a rich legacy as SCM.

APPENDIX A

DSEE PATENT NUMBER 4,809,170 4-JUN-1987

REFERENCES

Here are the references provided in the DSEE patent of 1987. It provides an illustration of the major sources of inspiration, as seen by the major vendor at that time.

FELDMAN, S. I. 1979. Make—A program for maintaining computer programs. *Softw. Pract. Exper.* (Apr.).

HABERMANN, N., ET AL. 1982. *The Second Compendium of GANDALF Documentation*. CMU Computer Science Dept. (May).

HECKEL, P. 1978. A technique for isolating differences between files. *Comput. ACM* (Apr).

IVIE, E. L. 1977. The programmer's workbench. *Commun. ACM* (Oct.) .

LAMPSON, B. AND SCHMIDT, E. 1983. Organizing software in a distributed environment. In *Proceedings of the SIGPLAN: ACM Special Interest Group on Programming Languages* (San Francisco, CA), ACM, New York, SBN:0-89791-108-3.

LAMPSON, B. AND SCHMIDT, E. 1983. Practical use of a polymorphic applicative language. In *Proceedings of the 10th POPL Conference* (Jan.).

LEACH, P., LEVINE, P., DOROUS, B., HAMILTON, J., NELSON, D., AND STUMPF, B. 1983. The architecture of an integrated local network. *IEEE J. Sel. Areas Commun.* (Nov.).

LEBLANG, D. B. 1982. Abstract syntax based programming environments. In *Proceedings of the ACM/AdaTEC Conference on Ada* (Washington DC, Oct.). ACM, New York.

LEBLANG, D. B., ET AL. 1984. Computer-aided software engineering in a distributed workstation environment. In *Proceedings of the ACM/SIGPLAN/SIGSOFT Conference on Practical Software Development Environments* (Apr.). ACM, New York.

OSTERWEIL, L. J. AND COWELL, W. R. 1983. The TOOLPACK/IST programming environment. IEEE/SOFTFAIR (July). IEEE Computer Society Press, Los Alamitos, CA.

SANDEWALL, E. 1978. Programming in an interactive environment: The "LISP" experience. *Comput. Surv. 10*, 1 (Mar.).

SOFTWARE ENG. SYMPOSIUM ON HIGH-LEVEL DEBUGGING. 1983. ACM/SIGSOFT/SIGPLAN. New York.

SOURCE CODE CONTROL SYSTEM USER'S GUIDE. 1981. UNIX System III Programmer's Manual (Oct.). CMS/MMS: Code/Module Management System Manual. Digital Equipment Corporation.

TEITELBAUM, T. 1981. The why and wherefore of the cornell program synthesizer. In *SIGPLAN*. ACM, New York.

TEITELMAN, W. 1983. Cedar: An interactive programming environment for a compiler-oriented language. In *Proceedings of the LANL/LLNL Conference on Work Stations in Support of Large Scale Computing* (Mar.).

TEITELMAN, W. AND MASINTER, L. 1981. The Interlisp programming environment. *Computer* (Apr.).

THALL, R. 1983. Large-scale software development with the Ada language system. In *Proceedings of the ACM Computer Science Conference* (Feb.). ACM, New York.

TICHY, W. F. 1982. Design, implementation and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering* (Sept.).

TUTORIAL: SOFTWARE DEVELOPMENT ENVIRONMENTS. 1981. IEEE/COMPSAC-81 (Nov.). STONEMAN: Requirements for AdA programming support environment. 1980. U.S. Department of Defense (Feb.).

APPENDIX B

CONTINUUS WHITE PAPER

(JANUARY 5, 1988)

REFERENCES

Here are the references provided in an early paper when Continuus was designed.

APOLLO COMPUTER, INC. 1985. Domain Software Engineering Environment (DSEE) Reference Manual., Order No. 003016, Rev.03 (July).

ARNOLD, K. 1986. C advisor. UNIX Review 5:6 (December).

DEREMER, F. AND KRON, H. H. 1976. Programming-in-the-large versus programming-in-the-small. *IEEE Trans. Softw. Eng. SE-2* (June), 80–86.

FELDMAN, S. I. 1978. Make: A program for maintaining computer programs. Bell Laboratories (Aug.).

FRASER, C. W. AND MYERS, E. W. 1987. An editor for revisioncontrol. *ACM Trans. Prog. Lang. Syst. 9*, 2 (Apr.), 277–295.

KAISER, G. E. AND HABERMANN, A. N. 1983. An environment for system version control. In *Digest of Papers Spring Comp-Con '83*. IEEE Computer Society Press, Los Alamitos, CA (Feb.), 415–420.

PERRY, D. E. 1987. Version control in the inscape environment. In *Proceedings of the 9th International Conference on Software Engineering* (Monterey, CA, Mar. 30–Apr. 2). IEEE Computer Society Press, Los Alamitos, CA. 142–149.

ROCHKIND, M. J. 1985. The source code control system. *IEEE Trans. Softw. Eng. SE-1*, 364–370.

TICHY, W. F. 1985. RCS—A system for version control. *Softw. Practice Exper. 15*, 7 (July), 637–654.

WINKLER, J. F. H. 1987. Version control in families of large programs. In *Proceedings of the 9th International Conference on Software Engineering* (Monterey, CA, Mar. 30–Apr. 2). IEEE Computer Society Press, Los Alamitos, CA, 150–161.

ACKNOWLEDGMENTS

We would like to thank everyone in the field of SCM, whether a researcher, an industrial vendor, or a customer. Knowingly or unknowingly, most if not all of you have advanced our field to where it is now—a standard and accepted part of any serious software development project.

REFERENCES

ADAMS, R., TICHY, W. F., AND WEINER, A. 1994. The cost of selective recompilation and environment processing. *ACM Trans. Softw. Eng. Meth. 3*, 1 (Jan), 3–28.

ADAMS, P. AND SOLOMON, M. 1995. An overview of the CAPITL software development environment. Software configuration management. ICSE SCM-4&5. Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, Berlin, Germany, 1–34.

AIDE-DE-CAMP: A SOFTWARE MANAGEMENT AND MAINTENANCE SYSTEM. 1988. *National Software Quality Assurance Conference*. Software Productivity Institute, Washington DC (April).

AIDE-DE-CAMP. SOFTWARE MAINTENANCE AND DEVELOPMENT SYSTEMS, INC. 1989. Aide-de-Camp Software Management System: Product Overview.

ALLEN, L., FERNANDEZ, G., KANE, K., LEBLANG, D., MINARD, D., AND POSNER, J. 1995. ClearCase MultiSite: supporting geographically-distributed software development. ICSE SCM-4 and SCM-5, Seattle USA (May).

ALAN, W. 1997. An holistic model for change control. In *Systems for Sustainability*. Plenum, New York, http://www.dis.port.ac.uk/~allangw/chng-man.htm.

ANT. The ANT rebuild system. Apache. http://jakarta.apache.org/ant/index.html.

ATKIN, D. 1998. Version sensitive editing: Change history as a programming tool. In *SCM 8* (Brussels, Belgium). Lecture Notes in Computer Science, vol. 1439. Springer-Verlag, New York.

BELKHATIR, N. AND ESTUBLIER, J. 1987. Software management constraints and action triggering in Adele program database. In *Proceedings of the 1st European Software Engineering Conference* (Strasbourg, France, Sept.). 47–57.

BELKHATIR, N., ESTUBLIER, J., AND MELO, W. L. 1991. Software process modeling in adele: The ISPW-7 example. In *Proceedings of the 7th International Software Process Workshop*, (San Francisco, CA, Oct.), I. Thomas, Ed. IEEE Computer Society Press, Los Alamitos, CA.

BELL LABS. 1997. *Sablime v5.0 User's Reference Manual*. Lucent Technologies, Murray Hill, NJ.

BEZIVIN, J. 2001. From object composition to model transformation with the MDA. TOOLS USA, Santa Barbara, CA (Aug.). (Available at http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda.)

BIELIKOVA, N. AND NAVRAT, P. 1995. Modelling software systems in configuration management. *Appl. Math. Computer Sci. 5*, 4, 751–764.

BOLCER, G. A. AND TAYLOR, R. N. 1996. Endeavors: A process system integration infrastructure. In *Proceedings of the 4th International Conference on the Software Process. (ICSP '96)* (Dec.), 76.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 1999. The unified modeling language user guide. Addison-Wesley Object Technology Series. Addison-Wesley, Reading, MA.

BOUDIER, G., GALLO, F., MINOT, R., AND THOMAS, I. 1988. An overview of PCTE and PCTE+. In *Proceedings of the ACM/SIGSOFT Software Engineering Symposium on Practical Software Development Environments* (Boston, MA, Nov.). ACM, New York, 248–257.

BRAEK, R. AND HAUGEN, Ø. 1993. *Engineering of Real Time Systems*. Prentice-Hall. Murray Hill, NJ.

BRET, B. 1993. Smart recompilation: What is it? Its benefits for the user, and its implementation in the DEC ADA compilation system. In *Conference Proceedings on TRI-Ada '93* (Seattle, WA, Sept.).

BUFFENEARGER, J. 1995. Syntactic software mergers. In *Proceedings of the SCM 5* (Seattle, WA, May). Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, New York.

CAGAN, M. 1993. Untangling configuration management. In *Proceedings of the Software Configuration Management, ICSE SCM4 and SCM5 Workshops* (Baltimore, MD, May) Selected Papers. Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, New York, 35–52.

CAGAN, M. 1994. Change management for software development. CaseWare, Inc. (later Continuus Software Corp. now Telelogic AB). http://www.continuus.com/developers/developersACED.html.

CAGAN, M. AND WEBER, D. 1996. Task-based configuration management. http://www.continuus.com/developers/developersACEA.html.

CASEWARE, INC. (now Continuus Software Corporation). 1989. Introduction to amplify control (later known as CaseWare/CM, then Continuus/CM).

CCACHE. http://ccache.samba.org.

CHRISTENSEN, F. T., ABBOTT, J., AND PFLAUM G. Rational clearcase UCM migration: A case study. Rational report. (Available at http://www-10.lotus.com/ldd/today.nsf/lookup/Rational_migration.)

CLEAR CASE current reference. (http://www.ibm.com/software/awdtools/clearcase/.)

CLEMM, G. 1988. The Odin specification language. In *Proceedings of the International Workshop on Software and Configuration Control*, J. Winkler, Ed. B.G.Teubner, Stuttgart, Germany.

CLEMM, G. 1995. The Odin system. In *Proceedings of SCM5* (Seattle, WA; June). Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, New York, 241–263.

CLEMM, G., AMSDEN, J., ELLISON, T., KALER, C., AND WHITEHEAD, J. 2002. RFC 3253. Versioning extensions to WebDAV (Web Distributed Authoring and Versioning) (March).

CONRADI, R., FUGGETTA, A., AND JACCHERI, M. L. 1998. Six theses on software process research. In *Software Process Technology, 6th European Workshop* (EWSPT'98) (Weybridge, UK, Sept.). V. Gruhn, Ed. Lecture Notes in Computer Science, vol. 1487. Springer-Verlag, New York, 100–104.

CONRADI, R. AND WESTFECHTEL, B. 1998. Version models for software configuration management. *ACM Comput. Surv. 30*, 2 (July), 232–282.

CONRADI, R. AND WESTFECHTEL, B. 1999. SCM: Status and future challenges. In *Proceedings of the International Workshop on Software Configuration Management* (SCM 99) (Toulouse, France, Sept.). J. Estublier, Ed. Lecture Notes in Computer Science, vol. 1675. Springer-Verlag, New York, 228–231.

CONTINUUS SOFTWARE CORPORATION (now Telelogic AB). 1993. Introduction to continuus/PT.

CRNKOVIC, I., ASKLUND, U., AND PERSSON, A. 2003. Implementing and integrating product data management and software configuration management. Artech House. Norwood, MA. ISBN 1-58053-498-8.

CVS PRODUCT DESCRIPTION. 2000. http://www.cvshome.org.
DART, S. 1991. Spectrum of functionality in configuration management systems. CMU/SEI-90-TR-11 ESD-90-TR-212. (http://www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90.)
DEREMER, F. AND KRON, H. H. 1976. Programming-in-the-large vs. Programming-in-the-small. *IEEE Trans. Softw. Eng. SE-2*, 2, 80–86.
DERNIAME, J.-C., KABA, B. A., AND WASTEL, D. EDS. 1999. *Software Process: Principles, Methodology, and Technology* (second book from PROMOTER project). Lecture Notes in Computer Science, vol. 1500. Springer-Verlag, New York, 307.
DINITTO, E. AND FUGGETTA, A., ED. 1998. *Process Technology*. Kluwer Academic Publishers Boston, MA.
DITTRICH, K. R., GOTTHARD, W., AND DAMOKLES, P. C. L. 1987. The database system for the unibase software engineering environment. *Data. Eng. 10*, 1 (Mar.).
DOWSON, M., NEJMEH, B., AND RIDDLE, W. 1991. Fundamental software process concepts. In *Proceedings of the 1st European Workshop on Software Process Modeling* (Milan, Italy). AICA Press.
D'SOUZA, D. 2001. Model-driven architecture and integration: Opportunities and challenges (Feb.).
ECLIPSE. http://www.eclipse.org/.
EIDNES, H., HALLSTEINSEN, D. O., AND WANVIK, D. H. 1989. Separate compilation in CHIPSY. In *Proceedings of the 2nd International Workshop on Software Configuration Management* (SCM) (Princeton, NJ). Lecture Notes in Computer Science. Springer-Verlag, New York, 42–45. ISSN:0163-5948.
ESTUBLIER, J., GHOUL, S., AND KRAKOWIAK, S. 1984. Preliminary experience with a configuration control system for modular programs. In *Proceedings of the 1st ACM SIGSOFT SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, PA, Apr.), P. B. Henderson, Ed. *ACM SIGPLAN Notices 19*, 5 (May), 149–156.
ESTUBLIER, J. 1985. A configuration manager: The Adele data base of programs. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large* (Harwichport, June).
ESTUBLIER, J. AND BELKHATIR, N. 1986. Experience with a data base of programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Conference on Practical Software Development Environments* (Palo Alto, CA, Dec.). ACM, New York, 84–91.
ESTUBLIER, J. AND CASALLAS, R. 1994. The Adele software configuration manager. In *Configuration Management*. W. F. Tichy, Ed. Wiley, New York, 99–133. http://www-adele.imag.fr/Les.Publications/bookChapters/ADELE1994Est.pdf.
ESTUBLIER, J., DAMI, S., AND AMIOUR, M. 1997. High level process modeling for scm systems. In *Proceedings of the International Workshop on Software Configuration Management* (SCM 7) (Boston, MA, May). Lecture Notes in Computer Science, vol. 1235. Springer-Verlag, New York, 81–98.
ESTUBLIER, J., FAVRE, J. M., AND MORAT, P. 1998. Toward PDM/SCM: Integration?. In *Proceedings of the International Workshop on Software Configuration Management* (SCM 8) (Brussels, Belgium, July). Lecture Notes in Computer Science, vol. 1439. Springer-Verlag, New York, 75–95.
ESTUBLIER, J. 2000. Software configuration management: A road map. In *The Future of Software Engineering* (supplement to *Proceedings of the 22nd International Conference on Software Engineering)* (Limerick, Ireland). ACM, New York, 279–289.
ESTUBLIER, J., GARCIA, S., AND VEGA, G. 2003. Defining and supporting concurrent engineering policies in SCM. In *Proceedings of the International Workshop on Software Configuration Management* (SCM-11) (Portland, OR). Lecture Notes in Computer Science, vol. 2649. Springer-Verlag, New York, 1–15.
FEILER, P. H. 1991. *Configuration Management Models in Commercial Environments*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
FELDMAN, S. I. 1979. Make—A program for maintaining computer programs. *Softw.—Pract. Exp. 9*, 3 (Mar.), 255–265.
FINKELSTEIN, A., KRAMER, J., AND NUSEBEIGH, B. 1994. *Software Process Modeling and Technology*. Wiley, New York (Advanced Software Development Series. ISBN 0 471 95206 0. (First book from PROMOTER project)).

FLEMMING, T., CHRISTENSEN, ABBOTT, J., AND PFLAUM, G. 2003. Rational ClearCase UCM Migration: A case study. Rational report. Available at http://www-10.lotus.com/ldd/today.nsf/lokkup/Rational_migration.

FOWLER, G. S., HUANG, Y., KOM, D. G., AND RAO, H. 1994. A user-level replicated file system. In *Proceedings of Summer USENIX* (June), 279–290.

FRASER, C. AND MYERS, E. 1987. An editor for revision control. *ACM Trans. Prog. Lang. Syst. 9*, 2 (Apr.), 277–295.

GALLI, M., LANZA, M., NIERSTRASZ, O., AND WUYTS, R. 2004. Ordering broken unit tests for focused debugging. In *ICSM 2004 Proceedings (20th IEEE International Conference on Software Maintenance)*. IEEE Computer Society Press, Los Alamitos, CA, 114–123.

GENTLEMENT, W., MACKEY, S., STEWARD, D., AND WEIN, M. 1989. Commercial real-time software needs different configuration management. In *SCM 2*. ACM, New York.

GESCHKE, C. M., MORRIS, J. H., AND SATTHERTWAITE, E. H. 1977. Early experience with mesa. *Commun. ACM 20*, 8 (Aug.), 540–551.

GOLDSTEIN, I. P. AND BOBROW, D. G. 1980. A layered approach to software design. Tech. Rep. CSL-80-5. XEROX PARC, Palo Alto, CA.

GULLA, B., KARLSON, E.-A., AND YEH, D. 1991. Change-oriented version descriptions in EPOS. *Softw. Eng. J. 6*, 6 (Nov.), 378-386.

HABERMANN, A. N. AND NOTKIN, D. 1986. Gandalf: Software development environments. *IEEE Trans. Softw. Eng. SE-12*, 12 (Dec.), 1117–1127 (Special issue on GANDALF).

HEIMAN, R. 2003. IDC Bulletin #29613 (June).

HEN. 1990. A filesystem for software development. In *Proceedings of USENIX Summer 1990 Conference* (Anaheim, CA, June). pp. 333–340.

HENDRICKS, D. 1990. A filesystem for software development. In *Proceedings of the USENIX Summer 1990 Conference* (Anaheim, CA, June). 333–340.

HEYDON, A., LEVIN, R., MANN, T., AND YU, Y. 2001. The VESTA approach to software configuration management. Compaq Systems Research Center Research Report 168 (Mar.).

HORWITZ, S., PRINS, J., AND REPS, T. 1989. Integrating non-interfering versions of programs. *ACM Trans. Prog. Lang. Syst. 11*, 3 (July).

HOUGH, H. 1981. Some thoughts on source update as a software maintenance tool. In *Proceedings of the IEEE Conference on Trends and Applications* (May). IEEE Computer Society Press, Los Alamitos, CA, CH1631-1/81/0000/0163.

HUNT, J. AND MCILLROY, M. 1976. An efficient algorithm for differential file comparison. Tech. Rep. 41. Bell Labs (June).

HUNT, J., VO, K., AND TICHY, W. 1986. An empirical study of delta mechanisms. In *Proceedings of the International Workshop on Software Configuration Management* (SCM 6), (Berlin, Germany, Mar.). Lecture Notes in Computer Science, vol. 1167. Springer-Verlag, New York.

ISO STANDARDS COMPENDIUM: ISO 9000—Quality management. 2003, 10th ed., 380 p., ISBN 92-67-10381-4.

JORDAN, M. AND VAN DE VANTER, M. 1995. Software configuration management in an object oriented database. In *Proceedings of the USENIX Conference on Object-Oriented Technologies*. (Available online at http://www.sun.com/research/forest/COM.Sun.Labs.Forest.doc.coots_95.abs. html.)

KATZ, R. H. 1990. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv. 22*, 4 (Dec.), 375–408.

KATAYAMA T., ED. 1991. Support for the software process. In *Proceedings of the 6th International Software Process Workshop*. IEEE Computer Society Press, Los Alamitos, CA.

KLIEWER, C. 1998. Software configuration management. http://sern.ucalgary.ca/~kliewerc/SENG/621/SCM_Pres.htm.

KNUTH, D. 1984. Literate programming. *Comput. J.*, 97–111.

KORN, D. AND KRELL, E. 1990. A new dimension for the UNIX file system. *Softw. Pract. Exper. 20*, S1 (July), S1/19–S1/34.

KORN, D. AND VO, K. 1995. VDELTA: Efficient data differencing and compression.

KRUSKAL, V. 1984. Managing multi-version programs with an editor. *IBM J. Res. Devel. 28*, 1 (Jan.).

LAMB, C., LANDIS, G., ORENSTEIN, J., AND WEINREB, D. 1991. The objectstore database system. *Commun. ACM 34*, 10 (Oct.), 50–63.

LEBLANG, D. B. 1994. The CM challenge: Configuration management that works. In *Configuration Management*, W. Tichy, Ed. Wiley, New York, Chap. 1, 1–37.

LEBLANG, D. B. 1997. Managing the software development process with ClearGuide. In *Proceedings of the ICSE'97 SCM-7 Workshop* (Boston, MA, May). Springer-Verlag, New York, 66–80.

LEBLANG, D. B. AND CHASE, R. P. 1984. Computer-aided software engineering in a distributed workstation environment. In *Proceedings of the Symposium on Practical Software Development Environments* (*Special issue of SIGPLAN Notices, 19*, 5 (May)). ACM, New York, 104–112.

LETHBRIDGE, T. C. 2000. What knowledge is important to a software professional? *IEEE Comput. 33*, 5 (May), 44–50.

LIE, A., DIDRIKSEN, T. M., CONRADI, R., KARLSSON, E.-A., HALLSTEINSSEN, S. O., AND HOLAGER, P. 1989. Change oriented versioning. In *Proceedings of the 2nd European Software Engineering Conference* (Coventry, UK, Sept.). C. Ghezzi and J. McDermid, Eds. Lecture Notes in Computer Science, vol. 387. Springer-Verlag, New York, 191–202.

LIN, Y.-J. AND REISS, S. P. 1995. Configuration management in terms of modules. In *Proceedings of the Software Configuration Management*, *ICSE SCM-4 and SCM-5 Workshops*. Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, New York.

LIN, Y.-J. AND REISS, S. P. 1996. Configuration management with logical structures. In *Proceedings of the 18th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA. 298–307.

MAGEE, M. 2003. Good electronic records management (GERM) using IBM rational ClearCase and IBM rational ClearQuest. IBM report, available at: http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/germ.pdf.

MAHLER, A. AND LAMPEN, A. 1988. SHAPE—A software configuration management tool. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Jan.). B. G. Teubner, Grassau, West Germany, 228–243.

MCCABE/TRUE SOFTWARE. 2000. Documentation 2000. http://www.mccabe.com/products.htm.

MEYERS, E. 1986. An OND difference algorithm and its variations. *Algorithmica 1*, 2, 51–266.

MICALEFF, J. AND CLEMM, G. M. 1996. The Asgard system: Activity-based configuration management. In *Proceedings of the Software Configuration Management, ICSE'96 SCM-6 Workshop* (Berlin, Germany, Mar.), I. Sommerville, Ed. Lecture Notes in Computer Science, vol. 1167, Springer-Verlag, New York, 175–186.

MICROSOFT. 2000. Sourcesafe Product Documentation, Microsoft, Inc., Seattle, WA.

NAVRAT, P. AND BIELIKOVA, N. 1996. Knowledge controlled version selection in software configuration management. *Softw. Concepts Tools*. *17*, 40–48.

OLSON, M. A. 1993. The design and implementation of the inversion file system. In *Proceedings of the 1993 Winter USENIX* (San Diego, CA, Jan.). 205–218.

OSTERWELL, L. J. 1987. Software processes are software too. *ICSE*, 2–13.

PAULK, M. C., CURTIS, B., CHRISSIS, M. B., AND WEBER, C. V. 1993. Capability maturity model, Version 1.1, *IEEE Softw. 10*, 4 (July), 18–27.

PAULK, M. C., CURTIS, B., CHRISSIS, M. B., AND WEBER, C. V. 1995. The Capability Maturity Model for Software: Guidelines for Improving the Software Process. (SEI Series in Software Engineering). Addison-Wesley, Reading, MA, 640.

PERRY, D. E. 1987. Version control in the inscape environment. In *Proceedings of the 9th International Conference on Software Engineering* (Monterey, CA, Mar.). Springer-Verlag, New York, 142–149.

PLOEDEREDER, E. AND FERGANY, A. 1989. The data model for a configuration management assistant. In *Proceedings of the 2nd International Workshop on Software Comfiguration Management* (SCM-2) (Princeton, NJ, Oct.). (As a special issue of *ACM SIGSOFT Engineering Newsletter* (SEN)), ACM, New York, 5–14.

PRIETO-DIAZ, R. AND NEIGHBOR, J. M. 1986. Module interconnection languages. *J. Syst. Softw. 6*, 307–334.

REICHENBERGER, C. 1991. Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Workshop on Software Configuration Management* (Trondheim, Norway, June). ACM, New York, 144–152.

REPS, T., HORWITZ, S., AND PRINS, J. 1988. Support for integrating program variants in an environment for programming in the large. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany).

ROCHKIND, M. J. 1975. The source code control system. *IEEE Trans. Softw. Eng. SE-1*, 4, 364–370.

ROOME, W. D. 1992. 3DFS. A time-oriented file server. In *Proceedings of the USENIX Winter 1992* (San Francisco, CA, Jan.).

SARNAK, N., BERNSTEIN, B., AND KRUSKAL, V. 1988. Creation and maintenance of multiple versions. *Syst. Config. Manage.* 264–275.

SCHNAZE, J. L., LEGGET, J., HICKS D. L., AND SZABO, R. 1993. Semantic data modeling of hypermedia associations. *ACM Trans. Inf. Syst. 11*, 1 (Jan.), 27–50. ISSN:1046-8188.

SCHWANKE, R. W. AND KAISER, G. E. 1988. Smarter recompilation. *ACM Trans. Prog. Lang. Syst.* 627–632, ISSN:0164-0925.

SHAW, M. AND GARLAN, D. 1996. Software architecture—Perspectives of an emerging discipline. Prentice Hall, Englewood Cliffs, NJ, 242.

SOLEY, R. AND THE OMG STAFF. 2000. Model-driven architecture. White paper, Draft 3.2. Available at www.omg.org.

SOFTTOOL. 1987. CCC: Change and configuration control environment: A functional overview.

SUN MICROSYSTEM, INC. 1989. The network software environment (NSE), Sun Tech. Rep. Sun Microsystems, Inc., Mountain View, CA, 104.

SUN/FORTE. 2000. Teamware product documentation. Sun MicroSystems Inc, Mointain View, CA.

SWINEHART, D. C., ZELLWEGER, P. T., BEACH, R. J., AND HAGMANN, R. B. 1986. A structural view of the cedar programming environment. *ACM Trans. Prog. Lang. Syst. 8*, 4 (Oct.), 419–490.

THOMAS, I. 1989. PCTE interfaces: Supporting tools in software-engineering environments. *IEEE Softw. 6*, 6 (Nov.), 15–23.

TICHY, W. F. 1982. Design implementation and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering*.

TICHY, W. F. 1985. RCS—A system for version control. *Softw. Practice Exp. 15*, 7, 637–654.

TICHY, W. F. 1986. Smart recompilation. *ACM Trans. Prog. Lang. Syst. 8*, 3, 273–291.

TICHY, W. F., ED. 1994. *Configuration Management* (*Trends in Software*). Wiley, New York, ISBN 0-471-94245-6.

TRYGGESETH, E., GULLA, B., AND CONRADI, R. 1995. Modelling systems with variability using the PROTEUS configuration language. In *Proceedings of the Software Configuration Management—ICSE SCM4 and SCM5 Workshops*. Selected Papers (Seattle, WA, Apr.). Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, New York, 115–126.

TURNER, C. R., FUGGETTA, A., LAVAZZA L., AND WOLF, A. L. 1999. A conceptual basis for feature engineering. *J. Syst. Softw. 49*, 1 (Dec.), 3–15.

VAN DER HOEK, A., HEIMBIGNER, D., AND WOLF, A. L. 1996. A generic, peer-to-peer repository for distributed configuration management. In *Proceedings of the 18th International Conference on Software Engineering* (Berlin, Germany, Mar.).

VAN DER HOEK, A., HEIMBIGNER, D., AND WOLF, A. L. 1998a. Software architecture, configuration management, and configurable distributed systems: A ménage a trois. Tech Report CU-CS-849-98. U. Colorado.

VAN DER HOEK, A., HEIMBIGNER, D., AND WOLF, A. L. 1998b. System modeling resurrected. In *Proceedings of the International Workshop on Software Configuration Management* (SCM 8), (Brussels, Belgium, July). Lecture Notes in Computer Science, vol. 1439, Springer-Verlag, New York.

WATER, R. C. 1989. Automated software management based on structural models. *Softw. Pract. Exper.*

WEBDAV. 1999. HTTP extentions for distributed authoring. RFC 2518. http://andrew2.andrew.cmu.edu/rfc/rfc2518.htm. February.

WEBDAV WEB SITE COMMUNITY. http://www.webdav.org/.

WEBER, D. W. 1997. Change sets versus change packages: Comparing implementations of change-based SCM. In *Proceedings of the 7th International Workshop on Software Configuration Management* (SCM'7) (Boston, MA, May). Lecture Notes in Computer Science, vol. 1235. Springer-Verlag, New York.

WESTFECHTEL, B. 1991. Structure oriented merging of revisions of software documents. In *Proceedings of the International Workshop on Software Configuration Management* (SCM 3). ACM, New York.

WESTFECHTEL, B., MUNCH, B. P., AND CONRADI, R. 2001. A layered architecture for software configuration management. *IEEE Trans. Softw. Eng. 27*, 12 (Dec.), 1111–1133.

WESTFECHTEL, B. AND CONRADI, R. 2003. Software architectures and software configuration management. In *Proceedings of the Software Configuration Management—ICSE Workshops SCM 2001 and SCM 2003* Selected Papers. A. van der Hoek and B. Westfechtel, Eds. Lecture Notes in Computer Science, vol. 2649 Springer-Verlag, New York, 24–39.

WHATIHEAD, J. 1999. Goals for a configuration management network protocol. In *Proceedings of the International Workshop on Software Configuration Management* (SCM 9) (Toulouse, France, Sept.). Lecture Notes in Computer Science, vol. 1675. Springer-Verlag, New York, 186–204.

WHEELER, D. 2004. Comments on OSS/FS software configuration management systems. http://www.dwheeler.com/essays/scm.html.

WHITGIFT, D. 1991. *Methods and Tools for Software Configuration Management*. Wiley, London, England, ISBN 0-471-92940-9.

WINGERD, L. AND SEIWALD, S. 1997. Constructing a large product with JAM. In *Proceedings of the International Workshop on Software Configuration Management* (SCM7) (Boston, MA, May). Lecture Notes in Computer Science, vol. 1235. Springer-Verlag, New York, 36–49.

WINKLER, J. F. H., ED. 1988. In *Proceedings of the ACM Workshop on Software Version and Configuration Control* (Grassau, FRG). Berichte des German Chapter of the ACM, Band 30, 466 p., B. G. Teubner-Verlag, Stuttgart, Germany.

WRIGHT, A. 1990. Requirements for a modern CM system. CaseWare, Inc. (later Continuus Software Corporation, now Telelogic AB).

ZELLER, A. AND SNELTING, G. 1997. Unified versioning through feature logic. *ACM Trans. Softw. Eng. Meth. 6*, 4 (Oct.), 397–440.

ZIMMERMANN, T., WEISSGERBER, P., DIEHL, S., AND ZELLER, A. 2004. Mining version histories to guide software changes. In *Proceedings of the ICSE* (Edinburgh, Scotland, June).

Received April 2004; revised November 2004; accepted December 2004