

Connecting Discrete Mathematics and Software Engineering

James P. Cohoon¹ and John C. Knight²
 University of Virginia, Department of Computer Science,
 Charlottesville, VA 22904

Abstract - Modern systems are critically dependent on software for their design and operation. The next generation of developers must be facile in the specification, design and implementation of dependable software using rigorous developmental processes. To help prepare this generation we have developed a teaching approach and materials that serve a two-fold purpose: promote an understanding and appreciation of the discrete mathematical structures (DM) that are the foundation of software engineering (SE) theory; and provide motivation and training of modern software development and analysis tools. Our teaching approach and materials form a two-course sequence establishing mastery of important SE areas: DM underlying modern SE theory (e.g., static analysis and proof); formal languages and their use (e.g., software specification languages); application of DM to SE (e.g., composition of theory and application for artifact development); and mathematics for advanced SE engineering needs (e.g., finite state machines and graph theory).

Index Terms – discrete mathematics, software engineering, specification, verification

INTRODUCTION AND BACKGROUND

Typical Computer Science education curricula do not coordinate the teaching of discrete mathematics and software engineering. Both discrete mathematics and software engineering courses are usually required in university programs, but the place where they merge—courses in the area typically called formal methods—are usually optional if they are offered at all. The standard discrete mathematics courses provide minimal motivation and material application. The standard software engineering courses provide little if any application of discrete mathematics, and the formal method courses are usually optional and late in the education of a Computer Science major. These circumstances result in students with minimal training in modern, mathematically based software-engineering theory and practice. Furthermore, the students are not given a chance to appreciate the utility and merit of this material and, quite literally, wonder why they are required to take discrete mathematics courses at all.

The discrete mathematics pedagogy has a rich background. Literature dates back even to the first proposed

computing curricula [1-4]. Tool use in discrete mathematics courses has also been encouraged [5-8].

A similar history is present for software engineering. A large number of textbooks have been likewise developed, and universities include training in software engineering in their undergraduate computing programs. There is even a conference held annually on the subject—the Conference on Software Engineering, Education and Training (CSEE&T).

The central importance of discrete mathematics as the basis for formal approaches to software development has been noted by many including Dijkstra, Gries, and Schneider [9-10]. This position continues to be espoused by the ITiCSE working group [11] among others [12-15]. The idea that discrete mathematics can be viewed as software engineering mathematics has been popularized by the Woodcock and Loomes textbook [16].

The application of discrete mathematics to the software development problem has been the subject of extensive research. Much of the initial effort was directed to formal verification, the process of showing the equivalence of two software system representations. As part of this effort, a number of formal languages have been developed for software specification (e.g., Z [17] and PVS [18]).

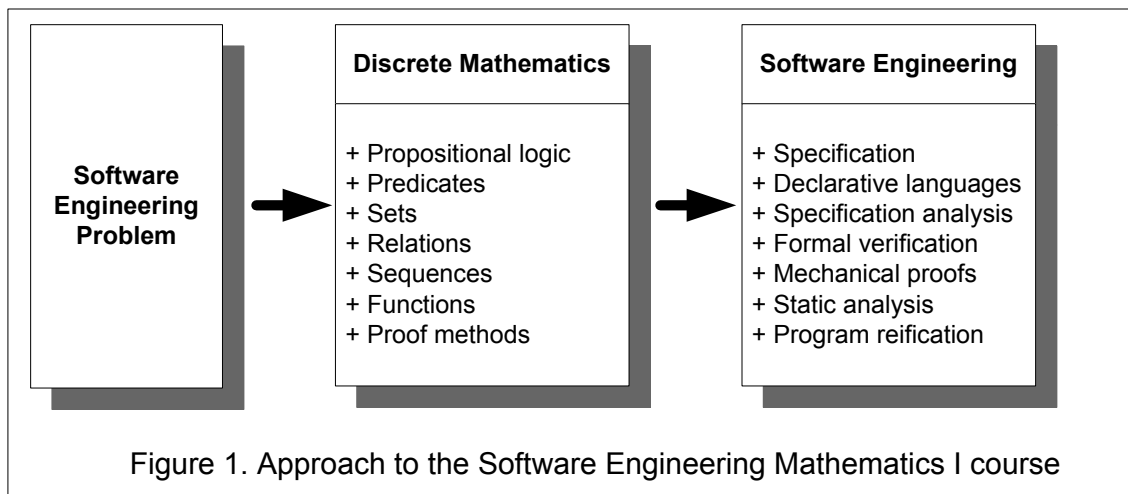
Because specification has proven the most error-prone phase of software development, recent formal methods work has focused on the application of formal languages to software specification. It has been shown experimentally that software specifications developed using formal languages possess several important advantages: they tend to contain fewer defects; they offer an improved level of communication between engineers, and they are amenable to many forms of analysis.

Despite its advantages, the utility and importance of discrete mathematics in software engineering is rarely taught to undergraduates in an integrated manner. As a result the great benefits of *appropriate* mathematics application to software development is neither appreciated nor practiced by the vast majority of Computer Science graduates. This omission is partly responsible for software being more expensive to develop than necessary and of poorer quality than required [19].

The curriculum development that we describe here remedies the disconnect between software engineering and discrete mathematics through a pragmatic approach using

¹ James P. Cohoon: cohoon@virginia.edu

² John C. Knight: knight@virginia.edu



courseware and tool development. In part, our approach was motivated by the work of Ince [20].

CLOSED LABORATORIES

Our curriculum remedy is just one part of a continuous ongoing evolutionary effort by the Department of Computer Science at the University of Virginia to provide a state-of-the-art undergraduate education.

The foundation of our curriculum was constructed in 1991. This initial activity was undertaken because the then current curriculum at Virginia did not prepare students well for either industry or graduate study. As part of the curriculum transformation, a new curriculum philosophy and materials were developed and continued to be developed [21-23]; and a strong emphasis was placed in the new curriculum on excellence in software development and mathematics [24]. In the initial activity, seven new core courses were developed. In a later, second phase of enhancement an additional four upper-level courses were developed [25].

A central feature of our curriculum and in particular to the new additions described here is the incorporation of a variety of different *closed laboratories*; initially in four core courses and subsequently with several other courses.

By a closed laboratory we mean an exercise activity that meets at an assigned time and place, and with a set agenda to be performed by the students. The simplest analogy for the concept is the familiar type of student laboratory exercise in physics and chemistry where students undertake some form of appropriate activity at a fixed time each week (usually) in a specially equipped, dedicated facility.

There are four types of laboratories taking place in conjunctions with courses:

- *Prescribed exercises*: Students work through a prescribed sequence of small interrelated steps that are checked off by a teaching assistant as each step is completed. Help is immediately available from the staff.

- *Artifact analysis*: Examination of carefully selected artifacts with a view to developing suitable comprehension and analysis skills.
- *Studio presentation*: Students present their solutions to a given problem and discuss them with their peers and instructors.
- *Case studies*: Many innovative approaches in other disciplines, such as business studies, make extensive use of case studies—the systematic study of artifacts. Our curriculum also utilizes this approach (e.g., students are provided with design documents and source code for a system with special or desirable characteristics and then engage in its analysis).

SEM1 AND SEM2

We limit our curriculum development description to two software engineering mathematic courses—SEM1 and SEM2—in our second-year computing curriculum. To provide clarity and context, our second year students also take a software development course; a digital logic course introducing computer architecture; and a course in program and data representation. In addition, in their first year they have taken an introductory Computer Science course.

The cornerstone of SEM1 and SEM2 is the use of problems that arise in software development to motivate the mathematics. Whereas the majority of undergraduate students are not sufficiently familiar with mathematics to appreciate its value fully, they are familiar with some of the difficulties that arise in developing software and are aware of the need for better technology.

Together SEM1 and SEM2 allow students to achieve mastery in the following areas:

- *Discrete mathematics underlying modern software engineering theory*: Propositional and first-order predicate logic; reasoning; proof techniques, induction; finite set theory; relations and graphs.

- *Formal languages and their use in software specification and verification*: Typical formal languages such as PVS and Z for software specification; proof techniques as they are applied to development of specification properties and in verification.
- *Application of discrete mathematics to practical software engineering issues*: Composition of the theory and application areas to develop artifacts in rigorous ways.
- *Mathematics derived from advanced software engineering needs*: Finite state machines; Turing machines; grammars; and tree and graph theory.

By first mastering these areas our students can then in their third year pursue additional mastery:

- *Mathematics and its role in the software development process* through a software-engineering capstone course that applies techniques to a project involving end-to-end system development.
- *Algorithms* through a course that applies the discrete mathematics techniques to the design and analysis of algorithms.

With these six masteries we believe that our students are well-prepared for careers in industry or graduate studies.

SEM1

Like traditional discrete mathematics courses, the core mathematical topics in SEM1 are an introduction to mathematical reasoning, sets, sequences, propositional and first order predicate logic, relations, functions and proof techniques including induction. However, the approach to their introduction is completely revised. As the topics are introduced, we take the opportunity to build on the knowledge that the student has gained by first discussing the application of the topics. As the course proceeds, we combine topics to introduce new ideas in software engineering such as declarative languages, formal specification, and formal verification.

The process by which we introduce a topic begins with a review of an important software problem. We then introduce the mathematical topic. Finally, we show how the mathematical topic contributes to dealing with the software problem. The process is as follows (see Figure 1):

- Present a general problem from software engineering, and expand and explain the general problem using examples from existing software artifacts.
- Introduce the relevant topic from discrete mathematics, showing informally how it might help, and then present the topic in appropriate depth.
- Present the mechanisms by which the topic from discrete mathematics can be used to address the software problem, and introduce the specific techniques used in practice to address the software engineering problem.

An example of this approach designed around finite set theory takes the following form:

- Motivating software problem: “In a software specification, how can we ensure that groups of items that

have to be processed by a computer program are defined precisely?” Explanatory example: “All files that are beyond a critical size and that are not owned by root have to be moved to a secondary storage device.”

- Set theory at an appropriate level is introduced with all relevant concepts and operations.
- Examples taken from software specification and design are presented to link the mathematics to the problem, and the role of sets in formal languages is discussed.

As different mathematical topics are covered, the synthesis of those topics in a software engineering concept is introduced. The process followed is essentially the same as is used for a single topic except that more time is spent developing the idea and explaining its use.

As an example, consider the topic of declarative languages for specification. These languages typically include many areas of discrete mathematics, but it is possible to adequately introduce the notion with just a few topics. The combination of finite set theory, propositions, and relations allows the specification of simple computer applications using a language like Z.

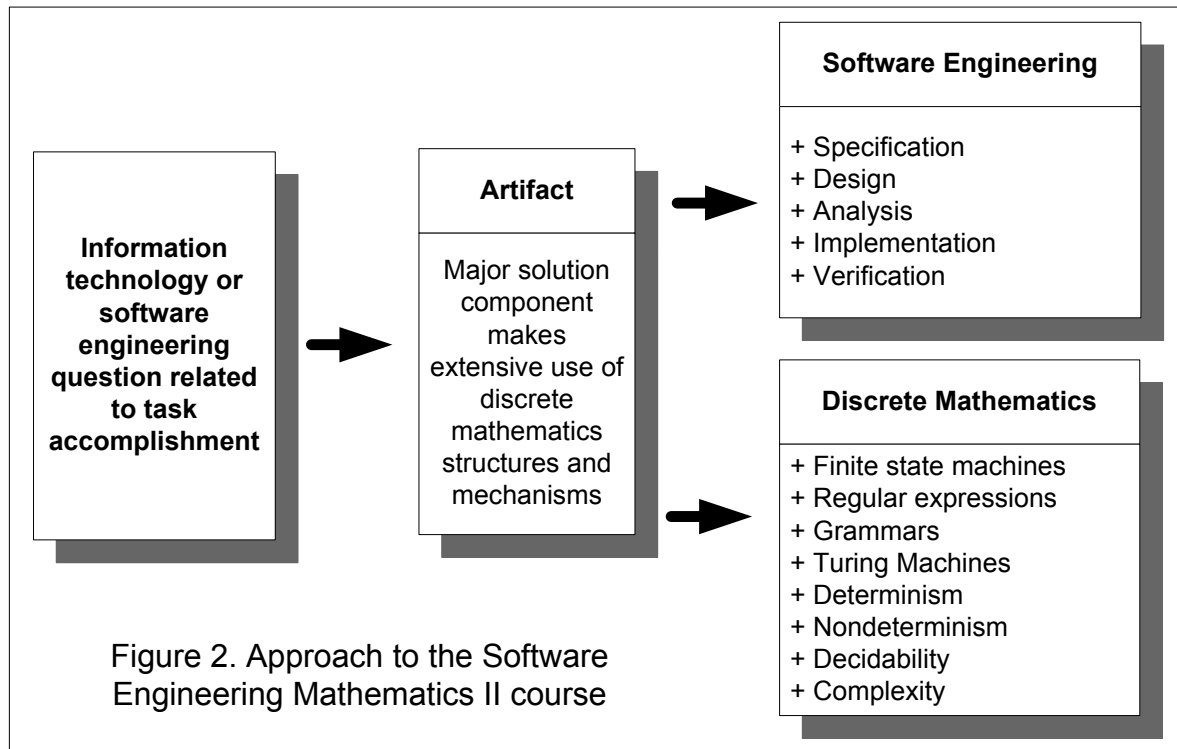
In SEM1, once the basic idea of formal specification is covered, we proceed to a more detailed treatment of Z with an introduction to the overall Z syntax and schema calculus. We also introduce the notion of types, and discuss the value and importance of both syntax and type checking in languages like Z.

The value of formal specification can be seen easily using this approach. Students can be shown simple analysis like type checking, and they appreciate quickly the value of types made possible by using a mathematical approach. In addition, even with simple specification examples written in Z, students quickly see the possibility of finding errors in specifications by inspection. The mathematical representation is both compact and more familiar than they realize initially, and so inspecting a mathematical artifact is seen to be useful and actually fun.

Proof techniques are challenging for students to master, and, in our experience, it is difficult to motivate students in the area of proof. At the level that proof techniques have been traditionally introduced, their perceived applicability to students’ expected professional activities is remote.

In SEM1, we are dealing with these issues in two ways. First, using the process outlined above, we motivate proof using several examples of software circumstances for which we need a high level of assurance. Second, we bring mechanical proof checking into the course material so as to relieve the students’ concern about the effort involved.

Formal verification is introduced to combine formal specification and the various notions of proof that have been presented. At present the coverage of formal verification is limited to discussion and examples. The students do not undertake formal verification of real software. Our goal is to ensure that they have the necessary appreciation of the problem and the available approaches.



We have begun a process of designing closed laboratories for SEM1. These laboratories are constructed in a traditional fashion with a series of steps for the students to undertake and with the requirement that a teaching assistant check the student's work at each step.

For example, the first set of SEM1 laboratories deals with the topics that we have found give students the most difficulty: creation and use of truth tables, and proof techniques. These laboratories are as follows:

- *Truth Table Manipulation.* In this laboratory, a computerized tool is employed to create truth tables for specific expressions. The exercises include establishing simple results in propositional logic by inspection of generated truth tables.
- *Simple Mechanical Proof.* In this laboratory, a mechanical proof system (PVS) is employed to allow students to proof simple results in propositional logic. The proofs have been presented and discussed in the classroom using traditional human proof processes.
- *Complex Mechanical Proof.* In this laboratory, PVS is used to assist in the development of more complex proofs including induction.

These laboratories and other laboratories are still being refined as further experience is gained and additional student opinions solicited.

Based on the experiences with a prototype version of SEM1, we propose a module-based implementation of the course with software development being the driver for the course. The analysis, design and implementation of software will require of the students new discrete mathematical knowledge and skills that the course will provide.

We next consider SEM2. SEM2 was the course to which the authors first turned their attention. Since the initial SEM2 offering by one of the authors, both authors have individually conducted the SEM1 course. Based both on the combined experiences with leading SEM1 and the students increased experience with the software engineering milieu by completing SEM1, the SEM2 course remains the subject of development.

SEM2

Our approach to the SEM2 course is also module based (see a depiction of the overall SEM2 approach in Figure 2.). The driving force behind a module is an artifact. The selected artifacts are significant in their own right. They are not just there to motivate discrete mathematics and show the various roles discrete mathematics plays and contributes to software engineering. The artifacts are important tools to which computing and software engineering professionals should be familiar.

The important discrete mathematics structures, mechanisms, and theory introduced in SEM2 are:

- Deterministic and nondeterministic finite state machines; regular expressions; closures; and equivalences;
- Grammars – especially regular and context-free; pumping lemmas; and the Chomsky hierarchy;
- Turing machines as computers, recognizers, and generators; Church-Turing hypothesis; recursively enumerable and nonrecursively enumerable languages; recursive sets;
- Decidability; and Rice's Theorem;
- Complexity; NP-hardness; and NP-completeness.

Like the module study in SEM1, a SEM2 module study begins with a motivating question or questions. The typical question asks how and why some strategic software engineering or information technology activity is normally accomplished. A noted software artifact and its accompanying practices are then introduced, examined, and evaluated. This coverage of the artifact takes the following form:

- The utility of the artifact in the student's professional or personal life is demonstrated.
- Students use the tool to become familiar with both its specification and operation.
- Guided analysis of the artifact leads to the discovery and appropriateness of additional discrete structures and mechanisms.
- Informal demonstrations show how the discrete structures and mechanisms work.

A module study then presents the discrete mathematics structures and mechanisms in suitable depth. To ensure appropriate student mastery of the discrete mathematics, students always perform exercises that manipulate and extend the discrete mathematics.

For selected modules the students will in parallel also design and implement a related software artifact. Students will also use the specification skills learned from SEM1 to properly capture program requirements. This activity coupled with similar activities in courses, helps ensure that the formal method practices and tools introduced in the SEM1 course become mastered.

An example of this approach designed for exploring finite state machines and regular expressions takes the following form.

- *Motivating professional and recreational questions:* "How does an operating system perform text and file searching?", "How can I find all occurrences of an identifier within a program?", and "How can I quickly search a word list to determine which words are possibilities for a particular crossword solution configuration?"
- *Proposed underlying solution – grep:* There are several freely available `grep` tools with excellent documentation (e.g., `cygwin grep` [26] and `wingrep` [27]). An examination of the pattern matching specification leads naturally to languages, nondeterminism, regular expressions and Kleene closure.
- *Guided analysis:* A discussion of how an implementation of a regular expression pattern matcher might take forms leads to both finite state machines and nondeterminism.
- *Informal demonstrations:* Use visual depictions of finite state machines for particularly noteworthy regular expressions and show the results of applying `grep`. Ad hoc evidence from the initial SEM2 offering indicates student interest in searching word lists for uncommon words; e.g., words with just one canonical occurrence of the vowels along with `y` (abstemiously and facetiously) and the longest English word with one vowel (strengths);

Classroom contact is then devoted to deterministic and nondeterministic finite state machines, regular expressions; the

expressiveness and limitations of regular languages, equivalences and closure properties.

In the initial offering of SEM2, laboratory contact hours were devoted to studio activities that resulted in the design of a basic `grep`-like tool and to discussions of implementation and testing issues. With the existence of SEM1, an important part of the next offering of SEM2 will include specification and other formal software engineering activities introduced in SEM1.

Noncontact hours are used both for gaining experience with using and reasoning on finite state machines, equivalences, and closure properties; and for the specification, design, analysis, and implementation of a basic `grep`-like tool. The implementation and verification exercise are done in small teams of two or three students.

Other modules have been developed and still others are being developed. Because none of the modules have yet been conducted with even the prototype of SEM1 being in place, the modules require further refinement and additional solicitation of student opinions.

For other module examples, the motivation for the grammars module is program languages and compilation, and the artifact is a `yacc`-like parser. The student laboratory and project activity is the development of a tool for compiling arithmetic expressions.

As other examples, the motivation for Turing Machines is bytecode interpreters. For decidability, a red herring is considered – a heuristic to recognize whether another program produces output. For NP-completeness, a satisfiability approach for solving Sudoku solutions is being developed. Other modules concerned with graph matching and file compression were prototyped but are not being pursued further because of changes in the expected discrete mathematics coverage of SEM2.

CONCLUSION

Our initial experience with this teaching approach in which the introduction of discrete mathematics is motivated by problems in software engineering indicates that the approach is feasible. Perhaps of more importance is the firmly established tie between software engineering and mathematics that this approach imparts to students – a link that they can exploit in numerous professional situations.

ACKNOWLEDGMENT

We thank the National Science Foundation and the Microsoft Corporation and its MSDN Curriculum Repository for their past and on-going support for curriculum development at the Department of Computer Science at the University of Virginia.

REFERENCES

- [1] Berztiss, A. T., "The why and how of discrete structures", *SIGCSE Technical Symposium on Computer Science Education*, 1976, pp. 22-25,
- [2] Prather, R. E., "Another look at the discrete structures course", *SIGCSE Technical Symposium on Computer Science Education*, 1976, pp. 247-252.

- [3] Tremblay, J. P., and R. Manohar, "A first course in discrete structures with applications to computer science," *SIGCSE Technical Symposium on Computer Science Education*, 1974, pp. 155-160.
- [4] A. Tucker, (editor), "Computing curricula 1991: report of the ACM/IEEE-CS Joint curriculum task force", ACM Press, 1991.
- [5] Knight, J.C., K.S. Hanks, and S.R. Travis, "Tool Support for Production Use of Formal Techniques", *International Symposium on Software Reliability Engineering*, Hong Kong, November 2001.
- [6] Rodger, S. H., A. O. Bilaska, K. H. Leider, M. Procopiuc, O. Procopiuc, J. R. Salemme, and E. Tsang, "A collection of tools for making automata theory and formal languages come alive, *ACM SIGCSE Bulletin*, March 1997, pp. 15-19.
- [7] Warford, J. S., "An experience teaching formal methods in discrete mathematics," *SIGCSE Bulletin*, 1995, pp. 60-64.
- [8] Ross, R., "Making Theory Come Alive", ACM SIGACT News, 1996, pp. 58-64.
- [9] E. W. Dijkstra, "On the cruelty of really teaching computing science." *Communications of the ACM*, December 1989, pp. 1398-1404.
- [10] Gries, D., and F. B. Schneider, *A logical approach to discrete math*, Springer-Verlag, New York, 1993.
- [11] Almstrum, V. L., C. N. Dean, D. Goelman, T. B. Hilburn, and J. Smith, "ITiCSE 2000 working group report: support for teaching formal methods," *SIGCSE Bulletin*, June 2001.
- [12] Fleury, A. E., "Evaluating discrete mathematics exercises", *SIGCSE Technical Symposium on Computer Science Education*, 1993, pp. 73-77.
- [13] McGuffee, J. W., "The discrete mathematics enhancement project", *Journal of Computing in Small Colleges*, 2002, pp. 162-166.
- [14] Saiedian, H., "Towards more formalism in software engineering education", *SIGCSE Technical Symposium on Computer Science Education*, 1993, pp. 193-197.
- [15] K. Heninger, "Specifying Software Requirements Complex Systems: New Techniques and Their Application", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 1, January 1980.
- [16] Woodcock, J., and M. Loomes, "Software Engineering Mathematics" *Software Engineering Institute, Series in Software Engineering*, 1988.
- [17] Spivey, J. M., *The Z Notation: A Reference Manual* (Prentice-Hall International Series in Computer Science)
- [18] Crow, J., S. Owre, J. Rushby, N. Shankar, and M. Srivas, *A Tutorial Introduction to PVS*, SRI International, 1998.
- [19] National Institute of Standards and Technology, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, <http://www.nist.gov/director/prog-ofc/report02-3.pdf>
- [20] Ince, D., *An Introduction to Discrete Mathematics, Formal System Specification, and Z*, Oxford University Press.
- [21] Knight, J. C., J. C. Prey, and W. A. Wulf, "Undergraduate Computer Science Education: A New Curriculum Philosophy & Overview", *SIGCSE Technical Symposium on Computer Science Education*, March 1994, pp. 155-159.
- [22] Knight, J. C., and J. C. Prey, "Exploring the software development process using a security camera in a CS2 Course. *Frontiers in Education*, November 1994, pp. 284-286.
- [23] J. P. Cohoon and J. W. Davidson, *Java Program Design*, McGraw-Hill, 2003.
- [24] Prey, J. C., J. P. Cohoon, and G. Fife, "Software engineering beginning in the first computer science course", SEI CSEE Conference, January 1994.
- [25] Knight, J. C., J. C. Prey, and W. A. Wulf, "Teaching Undergraduate computer science using closed laboratories", *Unity in Diversity, AAAS Annual Meeting and Science Innovation Exposition*, February 1995.
- [26] Faylor, C.G., and C. Vinshcen, *Cygwin information and installation*, <http://www.cygwin.com>, 2006.
- [27] Millington, H., *Windows Grep Search Utility Home Page*, <http://www.wingrep.com>, 2006.