Charles Rattray
Savitri Maharaj
Carron Shankland (Eds.)

LNCS 3116

Algebraic Methodology and Software Technology

10th International Conference, AMAST 2004
Stirling, Scotland, July 2004
Proceedings

Charles Rattray Savitri Maharaj
Carron Shankland (Eds.)

Algebraic Methodology and Software Technology

10th International Conference, AMAST 2004
Stirling, Scotland, UK, July 12-16, 2004
Proceedings

Springer

Volume Editors

Charles Rattray
Savitri Maharaj
Carron Shankland
University of Stirling
FK9 4LA Stirling, UK
E-mail: {c.rattray,s.maharaj,c.shankland}@cs.stir.ac.uk

Library of Congress Control Number: 2004108198

CR Subject Classification (1998): F.3-4, D.2, C.3, D.1.6, I.2.3, I.1.3

ISSN 0302-9743
ISBN 3-540-22381-9 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11019428 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 4829

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Preface

This volume contains the proceedings of AMAST 2004, the 10th International Conference on Algebraic Methodology and Software Technology, held during July 12–16, 2004, in Stirling, Scotland, UK. The major goal of the AMAST conferences is to promote research that may lead to the setting of software technology on a firm, mathematical basis. This goal is achieved by a large international cooperation with contributions from both academia and industry. The virtues of a software technology developed on a mathematical basis have been envisioned as being capable of providing software that is (a) correct, and the correctness can be proved mathematically, (b) safe, so that it can be used in the implementation of critical systems, (c) portable, i.e., independent of computing platforms and language generations, and (d) evolutionary, i.e., it is self-adaptable and evolves with the problem domain.

Previous AMAST meetings were held in Iowa City (1989, 1991, 2000), Twente (1993), Montreal (1995), Munich (1996), Sydney (1997), Manaus (1999), and Reunion Island (2002), and contributed to the AMAST goals by reporting and disseminating academic and industrial achievements within the AMAST area of interest. During these meetings, AMAST attracted an international following among researchers and practitioners interested in software technology, programming methodology and their algebraic and logical foundations.

For AMAST 2004 there were 63 submissions of overall high quality, authored by researchers from Australia, Canada, China, the Czech Republic, Denmark, France, Germany, India, Iran, Israel, Italy, Korea, Portugal, Spain, Taiwan, The Netherlands, Turkey, the UK, and the USA. All submissions were thoroughly evaluated, and an electronic program committee meeting was held to discuss the reviewers' reports. The program committee selected 35 papers to be presented. This volume includes these papers, and abstracts or papers of invited lectures given by Roland Backhouse, Don Batory, Michel Bidoit, Muffy Calder, Bart Jacobs, and John-Jules Meyer.

We heartily thank the members of the program committee and all the referees for their care and time in reviewing the submitted papers, and all the institutions that supported AMAST 2004: the Edinburgh Mathematical Society, the Engineering and Physical Sciences Research Council, the London Mathematical Society, and the Formal Aspects of Computing Science specialist group of the British Computer Society.

May 2004

Charles Rattray
Savitri Maharaj
Carron Shankland

Program Committee Chairs

AMAST 2004 was organized by the following team at the department of Computing Science and Mathematics, University of Stirling, UK.

C. Rattray
S. Maharaj
C. Shankland

Steering Committee

E. Astesiano (Italy)
R. Berwick (USA)
M. Johnson (Australia)(chair)
Z. Manna (USA)
M. Mislove (USA)
A. Nijholt (The Netherlands)
M. Nivat (France)
C. Rattray (UK)
T. Rus (USA)
G. Scollo (Italy)
M. Sintzoff (Belgium)
J. Wing (USA)
M. Wirsing (Germany)

Program Committee

V.S. Alagar (USA)
G. Barthe (France)
M. Bidoit (France)
R. Bland (UK)
P. Blauth Menezes (Brazil)
G. v. Bochmann (Canada)
C. Brink (South Africa)
M. Broy (Germany)
M.L. Bujorianu (UK)
C. Calude (New Zealand)
C. Choppy (France)
R.G. Clark (UK)
A. Fleck (USA)
M. Frias (Argentina)
J. Goguen (USA)
N. Halbwachs (France)
A. Hamilton (UK)
A. Haxthausen (Denmark)
P. Henderson (UK)
M. Hinchey (USA)
A. Lopes (Italy)
M. Mislove (USA)
P. Mosses (Denmark)
M. Nesi (Italy)
R. De Nicola (Italy)
A. Nijholt (The Netherlands)
F. Orejas (Spain)
D. Pavlovic (USA)
T. Rus (USA)
S. Schneider (UK)
G. Scollo (Italy)
S. Seidman (USA)
D. Smith (USA)
C. Talcott (USA)
A. Tarlecki (Poland)
K.J. Turner (UK)
J. van de Pol (The Netherlands)
P. Veloso (Brazil)
L. Wallen (UK)
K. Williamson (USA)
M. Wirsing (Germany)

Referees

C. Adams
V.S. Alagar
P. Baillot
G. Barthe
S. Berghofer
M. Bidoit
L. Blair
G.v. Bochmann
C. Brink
H. Bruun
M.L. Bujorianu
M.C. Bujorianu
M. Calder
C. Calude
N. Catano
M. Cerioli
C. Choppy
B. Daou
M. Debbabi
G. Dufay
N. Evans
A. Fleck
M. Fränzle
M. Frias
R. Giacobazzi
J. Goguen
N. Halbwachs
R.R. Hansen
M.J. Healy
P. Henderson
M. Hinchey
J. Hodgkin
M. Huisman
M. Huth
S.B. Jones
J. Kleist
H.H. Løvengreen
S. Maharaj
F. Mancinelli
P.E. Martínez López
I.A. Mason
I. Mastroeni
F. Mehta
M. Mislove
P.D. Mosses
M. Nesi
H. Nilsson
K. Ogata
F. Orejas
J. Ouaknine
V. de Paiva
D. Pavlovic
P. Pelliccione
L. Petrucci
M. Poel
J. van de Pol
M Rappl
C. Rattray
G. Reggio
E. Ritter
N. Schirmer
S. Schneider
G. Scollo
R. Segala
S.B. Seidman
C. Shankland
R. Sharp
D. Smith
M. Spichkova
M. Strecker
A. Suenbuel
C. Talcott
A. Tarlecki
H. Treharne
K.J. Turner
M. Valero Espada
L. Wallen
K.E. Williamson
T. Wilson
M. Wirsing
J. Worrell
H. Yahyaoui
J. Zwiers

Sponsoring Institutions

Edinburgh Mathematical Society
Engineering and Physical Sciences Research Council
Formal Aspects of Computing Science Specialist Group of the British Computer Society
London Mathematical Society

Lecture Notes in Computer Science

Sublibrary 2: Programming and Software Engineering

For information about Vols. 1– 4214
please contact your bookseller or Springer

Vol. 4849: M. Winckler, H. Johnson, P. Palanque (Eds.), Task Models and Diagrams for User Interface Design. XIII, 299 pages. 2007.

Vol. 4834: R. Cerqueira, R.H. Campbell (Eds.), Middleware 2007. XIII, 451 pages. 2007.

Vol. 4829: M. Lumpe, W. Vanderperren (Eds.), Software Composition. X, 281 pages. 2007.

Vol. 4824: A. Paschke, Y. Biletskiy (Eds.), Advances in Rule Interchange and Applications. XIII, 243 pages. 2007.

Vol. 4807: Z. Shao (Ed.), Programming Languages and Systems. XI, 431 pages. 2007.

Vol. 4799: A. Holzinger (Ed.), HCI and Usability for Medicine and Health Care. XVI, 458 pages. 2007.

Vol. 4789: M. Butler, M.G. Hinchey, M.M. Larrondo-Petrie (Eds.), Formal Methods and Software Engineering. VIII, 387 pages. 2007.

Vol. 4767: F. Arbab, M. Sirjani (Eds.), International Symposium on Fundamentals of Software Engineering. XIII, 450 pages. 2007.

Vol. 4764: P. Abrahamsson, N. Baddoo, T. Margaria, R. Messnarz (Eds.), Software Process Improvement. XI, 225 pages. 2007.

Vol. 4762: K.S. Namjoshi, T. Yoneda, T. Higashino, Y. Okamura (Eds.), Automated Technology for Verification and Analysis. XIV, 566 pages. 2007.

Vol. 4758: F. Oquendo (Ed.), Software Architecture. XVI, 340 pages. 2007.

Vol. 4757: F. Cappello, T. Herault, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface. XVI, 396 pages. 2007.

Vol. 4753: E. Duval, R. Klamma, M. Wolpers (Eds.), Creating New Learning Experiences on a Global Scale. XII, 518 pages. 2007.

Vol. 4749: B.J. Krämer, K.-J. Lin, P. Narasimhan (Eds.), Service-Oriented Computing – ICSOC 2007. XIX, 629 pages. 2007.

Vol. 4748: K. Wolter (Ed.), Formal Methods and Stochastic Models for Performance Evaluation. X, 301 pages. 2007.

Vol. 4741: C. Bessière (Ed.), Principles and Practice of Constraint Programming – CP 2007. XV, 890 pages. 2007.

Vol. 4735: G. Engels, B. Opdyke, D.C. Schmidt, F. Weil (Eds.), Model Driven Engineering Languages and Systems. XV, 698 pages. 2007.

Vol. 4716: B. Meyer, M. Joseph (Eds.), Software Engineering Approaches for Offshore and Outsourced Development. X, 201 pages. 2007.

Vol. 4680: F. Saglietti, N. Oster (Eds.), Computer Safety, Reliability, and Security. XV, 548 pages. 2007.

Vol. 4670: V. Dahl, I. Niemelä (Eds.), Logic Programming. XII, 470 pages. 2007.

Vol. 4652: D. Georgakopoulos, N. Ritter, B. Benatallah, C. Zirpins, G. Feuerlicht, M. Schoenherr, H.R. Motahari-Nezhad (Eds.), Service-Oriented Computing ICSOC 2006. XVI, 201 pages. 2007.

Vol. 4640: A. Rashid, M. Aksit (Eds.), Transactions on Aspect-Oriented Software Development IV. IX, 191 pages. 2007.

Vol. 4634: H. Riis Nielson, G. Filé (Eds.), Static Analysis. XI, 469 pages. 2007.

Vol. 4620: A. Rashid, M. Aksit (Eds.), Transactions on Aspect-Oriented Software Development III. IX, 201 pages. 2007.

Vol. 4615: R. de Lemos, C. Gacek, A. Romanovsky (Eds.), Architecting Dependable Systems IV. XIV, 435 pages. 2007.

Vol. 4610: B. Xiao, L.T. Yang, J. Ma, C. Muller-Schloer, Y. Hua (Eds.), Autonomic and Trusted Computing. XVIII, 571 pages. 2007.

Vol. 4609: E. Ernst (Ed.), ECOOP 2007 – Object-Oriented Programming. XIII, 625 pages. 2007.

Vol. 4608: H.W. Schmidt, I. Crnković, G.T. Heineman, J.A. Stafford (Eds.), Component-Based Software Engineering. XII, 283 pages. 2007.

Vol. 4591: J. Davies, J. Gibbons (Eds.), Integrated Formal Methods. IX, 660 pages. 2007.

Vol. 4589: J. Münch, P. Abrahamsson (Eds.), Product-Focused Software Process Improvement. XII, 414 pages. 2007.

Vol. 4574: J. Derrick, J. Vain (Eds.), Formal Techniques for Networked and Distributed Systems – FORTE 2007. XI, 375 pages. 2007.

Vol. 4556: C. Stephanidis (Ed.), Universal Access in Human-Computer Interaction, Part III. XXII, 1020 pages. 2007.

Vol. 4555: C. Stephanidis (Ed.), Universal Access in Human-Computer Interaction, Part II. XXII, 1066 pages. 2007.

Vol. 4554: C. Stephanidis (Ed.), Universal Acess in Human Computer Interaction, Part I. XXII, 1054 pages. 2007.

Vol. 4553: J.A. Jacko (Ed.), Human-Computer Interaction, Part IV. XXIV, 1225 pages. 2007.

Vol. 4552: J.A. Jacko (Ed.), Human-Computer Interaction, Part III. XXI, 1038 pages. 2007.

Vol. 4551: J.A. Jacko (Ed.), Human-Computer Interaction, Part II. XXIII, 1253 pages. 2007.

Vol. 4550: J.A. Jacko (Ed.), Human-Computer Interaction, Part I. XXIII, 1240 pages. 2007.

Vol. 4542: P. Sawyer, B. Paech, P. Heymans (Eds.), Requirements Engineering: Foundation for Software Quality. IX, 384 pages. 2007.

Vol. 4536: G. Concas, E. Damiani, M. Scotto, G. Succi (Eds.), Agile Processes in Software Engineering and Extreme Programming. XV, 276 pages. 2007.

Vol. 4530: D.H. Akehurst, R. Vogel, R.F. Paige (Eds.), Model Driven Architecture - Foundations and Applications. X, 219 pages. 2007.

Vol. 4523: Y.-H. Lee, H.-N. Kim, J. Kim, Y.W. Park, L.T. Yang, S.W. Kim (Eds.), Embedded Software and Systems. XIX, 829 pages. 2007.

Vol. 4498: N. Abdennahder, F. Kordon (Eds.), Reliable Software Technologies - Ada-Europe 2007. XII, 247 pages. 2007.

Vol. 4486: M. Bernardo, J. Hillston (Eds.), Formal Methods for Performance Evaluation. VII, 469 pages. 2007.

Vol. 4470: Q. Wang, D. Pfahl, D.M. Raffo (Eds.), Software Process Dynamics and Agility. XI, 346 pages. 2007.

Vol. 4468: M.M. Bonsangue, E.B. Johnsen (Eds.), Formal Methods for Open Object-Based Distributed Systems. X, 317 pages. 2007.

Vol. 4467: A.L. Murphy, J. Vitek (Eds.), Coordination Models and Languages. X, 325 pages. 2007.

Vol. 4454: Y. Gurevich, B. Meyer (Eds.), Tests and Proofs. IX, 217 pages. 2007.

Vol. 4444: T. Reps, M. Sagiv, J. Bauer (Eds.), Program Analysis and Compilation, Theory and Practice. X, 361 pages. 2007.

Vol. 4440: B. Liblit, Cooperative Bug Isolation. XV, 101 pages. 2007.

Vol. 4408: R. Choren, A. Garcia, H. Giese, H.-f. Leung, C. Lucena, A. Romanovsky (Eds.), Software Engineering for Multi-Agent Systems V. XII, 233 pages. 2007.

Vol. 4406: W. De Meuter (Ed.), Advances in Smalltalk. VII, 157 pages. 2007.

Vol. 4405: L. Padgham, F. Zambonelli (Eds.), Agent-Oriented Software Engineering VII. XII, 225 pages. 2007.

Vol. 4401: N. Guelfi, D. Buchs (Eds.), Rapid Integration of Software Engineering Techniques. IX, 177 pages. 2007.

Vol. 4385: K. Coninx, K. Luyten, K.A. Schneider (Eds.), Task Models and Diagrams for Users Interface Design. XI, 355 pages. 2007.

Vol. 4383: E. Bin, A. Ziv, S. Ur (Eds.), Hardware and Software, Verification and Testing. XII, 235 pages. 2007.

Vol. 4379: M. Südholt, C. Consel (Eds.), Object-Oriented Technology. VIII, 157 pages. 2007.

Vol. 4364: T. Kühne (Ed.), Models in Software Engineering. XI, 332 pages. 2007.

Vol. 4355: J. Julliand, O. Kouchnarenko (Eds.), B 2007: Formal Specification and Development in B. XIII, 293 pages. 2006.

Vol. 4354: M. Hanus (Ed.), Practical Aspects of Declarative Languages. X, 335 pages. 2006.

Vol. 4350: M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude - A High-Performance Logical Framework. XXII, 797 pages. 2007.

Vol. 4348: S. Tucker Taft, R.A. Duff, R.L. Brukardt, E. Plödereder, P. Leroy, Ada 2005 Reference Manual. XXII, 765 pages. 2006.

Vol. 4346: L. Brim, B.R. Haverkort, M. Leucker, J. van de Pol (Eds.), Formal Methods: Applications and Technology. X, 363 pages. 2007.

Vol. 4344: V. Gruhn, F. Oquendo (Eds.), Software Architecture. X, 245 pages. 2006.

Vol. 4340: R. Prodan, T. Fahringer, Grid Computing. XXIII, 317 pages. 2007.

Vol. 4336: V.R. Basili, H.D. Rombach, K. Schneider, B. Kitchenham, D. Pfahl, R.W. Selby (Eds.), Empirical Software Engineering Issues. XVII, 193 pages. 2007.

Vol. 4326: S. Göbel, R. Malkewitz, I. Iurgel (Eds.), Technologies for Interactive Digital Storytelling and Entertainment. X, 384 pages. 2006.

Vol. 4323: G. Doherty, A. Blandford (Eds.), Interactive Systems. XI, 269 pages. 2007.

Vol. 4322: F. Kordon, J. Sztipanovits (Eds.), Reliable Systems on Unreliable Networked Platforms. XIV, 317 pages. 2007.

Vol. 4309: P. Inverardi, M. Jazayeri (Eds.), Software Engineering Education in the Modern Age. VIII, 207 pages. 2006.

Vol. 4294: A. Dan, W. Lamersdorf (Eds.), Service-Oriented Computing – ICSOC 2006. XIX, 653 pages. 2006.

Vol. 4290: M. van Steen, M. Henning (Eds.), Middleware 2006. XIII, 425 pages. 2006.

Vol. 4279: N. Kobayashi (Ed.), Programming Languages and Systems. XI, 423 pages. 2006.

Vol. 4262: K. Havelund, M. Núñez, G. Roşu, B. Wolff (Eds.), Formal Approaches to Software Testing and Runtime Verification. VIII, 255 pages. 2006.

Vol. 4260: Z. Liu, J. He (Eds.), Formal Methods and Software Engineering. XII, 778 pages. 2006.

Vol. 4257: I. Richardson, P. Runeson, R. Messnarz (Eds.), Software Process Improvement. XI, 219 pages. 2006.

Vol. 4242: A. Rashid, M. Aksit (Eds.), Transactions on Aspect-Oriented Software Development II. IX, 289 pages. 2006.

Vol. 4229: E. Najm, J.-F. Pradat-Peyre, V.V. Donzeau-Gouge (Eds.), Formal Techniques for Networked and Distributed Systems - FORTE 2006. X, 486 pages. 2006.

Vol. 4227: W. Nejdl, K. Tochtermann (Eds.), Innovative Approaches for Learning and Knowledge Sharing. XVII, 721 pages. 2006.

Vol. 4218: S. Graf, W. Zhang (Eds.), Automated Technology for Verification and Analysis. XIV, 540 pages. 2006.

Table of Contents

Invited Speakers

Contributed Talks

Algebraic Approaches to Problem Generalisation

Roland Backhouse

School of Computer Science and Information Technology, University of Nottingham, Nottingham NG8 1BB, England
rcb@cs.nott.ac.uk

Abstract. A common technique for solving programming problems is to express the problem in terms of solving a system of so-called "simultaneous" equations (a collection of equations in a number of unknowns that are often mutually recursive). Having done so, a number of techniques can be used for solving the equations, ranging from simple iterative techniques to more sophisticated but more specialised elimination techniques.

A stumbling block for the use of simultaneous equations is that there is often a big leap from a problem's specification to the construction of the system of simultaneous equations; the justification for the leap almost invariably involves a *post hoc* verification of the construction. Thus, whereas methods for solving the equations, once constructed, are well-known and understood, the process of constructing the equations is not.

In this talk, we present a general theorem which expresses when the solution to a problem can be expressed as solving a system of simultaneous equations. The theorem exploits the theory of Galois connections and fixed-point calculus, which we briefly introduce. We give several examples of the theorem together with several non-examples (that is, examples where the theorem is not directly applicable). The non-examples serve two functions. They highlight the gap between specification and simultaneous equations – we show in several cases how a small change in the specification leads to a breakdown in the solution by simultaneous equations – and they inform the development of a methodology for the construction of the equations.

Application of the technique in the case of the more challenging problems depends crucially on finding a suitable generalisation of the original problem. For example, the problem of finding the *edit distance* between a word and a context-free language is solved by computing the edit distance between each segment of the given word and the language generated by each nonterminal in the given grammar.

A focus of the talk is the use of Conway's factor theory [Con71] in generalising a class of problems we call "bound" problems. Since its publication in 1971, Conway's work has been largely ignored, but its relevance to program analysis has recently been observed by Oege de Moor and his colleagues [MDLS02,SdML04]. We show how factor theory underpins De Moor's work as well as the well-known Knuth-Morris-Pratt pattern matching algorithm [KMP77]. We also speculate on how further study of factor theory might have relevance to a broader class of problems.

This talk is based on [Bac04], where further details can be found.

C. Rattray et al. (Eds.): AMAST 2004, LNCS 3116, pp. 1–2, 2004.

References

[Bac04] Roland Backhouse. Regular algebra applied to language problems. *Submitted for publication in Journal of Logic and Algebraic Programming*, 2004.

[Con71] J.H. Conway. *Regular Algebra and Finite Machines.* Chapman and Hall, London, 1971.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:325–350, 1977.

[MDLS02] O. de Moor, S. Drape, D. Lacey, and G. Sittampalam. Incremental program analysis via language factors. Available from http://web.comlab.ox.ac.uk/work/oege.de.moor/pubs.htm, 2002.

[SdML04] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larssen. Incremental execution of transformation specifications. In *POPL'04*, 2004.

A Science of Software Design

Don Batory

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78746
batory@cs.utexas.edu

Abstract. Underlying large-scale software design and program synthesis are simple and powerful algebraic models. In this paper, I review the elementary ideas upon which these algebras rest and argue that they define the basis for a science of software design.

1 Introduction

I have worked in the areas of program generation, software product-lines, domain specific languages, and component-based architectures for over twenty years. The emphasis of my research has been on large-scale program synthesis and design automation. The importance of these topics is intuitive: higher productivity, improved software quality, lower maintenance costs, and reduced time-to-market can be achieved through automation.

Twenty years has given me a unique perspective on software design and software modularity. My work has revealed that large scale software design and program synthesis is governed by simple and powerful algebraic models. In this paper, I review the elementary ideas on which these algebras rest. To place this contribution in context, a fundamental problem in software engineering is the abject lack of a science for software design. I will argue that these algebraic models can define the basis for such a science.

I firmly believe that future courses in software design will be partly taught using domain-specific algebras, where a program's design is represented by a composition of operators, and design optimization is achieved through algebraic rewrites of these compositions. This belief is consistent with the goal of AMAST. However, I suspect that *how* I use algebras and their relative informality to achieve design automation is unconventional to the AMAST community. As a background for my presentation, I begin with a brief report on the 2003 Science of Design Workshop.

2 NSF's Science of Design Workshop

In October 2003, I attended a *National Science Foundation (NSF)* workshop in Airlie, Virginia on the "Science of Design" [11]. The goal of the workshop was to determine the meaning of the term "Science of Design". NSF planned to start a program with this title and an objective was to determine lines of research to fund. There were 60 attendees from the U.S., Canada, and Europe. Most were from the practical side of

C. Rattray et al. (Eds.): AMAST 2004, LNCS 3116, pp. 3–18, 2004.

software engineering; a few attendees represented the area of formal methods. I was interested in the workshop to see if others shared my opinions and experiences in software design, but more generally, I wanted to see what a cross-section of today's Software Engineering community believed would be the "Science of Design". In the following, I review a few key positions that I found particularly interesting.

Richard Gabriel is a Distinguished Engineer at Sun Microsystems and one of the architects of Common Lisp. He described his degree in creative writing – in particular, poetry – and demonstrated that it was far more rigorous in terms of course work than a comparable degree in Software Engineering (of which software design was but a small part). He advocated that students should be awarded degrees in "Fine Arts" for software design. I was astonished: I did not expect to hear such a presentation at a *Science* of Design workshop. Nevertheless, Gabriel reinforced the common perception that software design is indeed an art, and a poorly understood art at that.

Carliss Balwin is a Professor at the Harvard Business School. She argued that software design is an instance of a much larger paradigm of product design. She observed that the processes by which one designs a table, or a chair, or an auditorium, are fundamentally similar to that of designing software. Consequently, software design has firm roots in economic processes and formalisms. Once again, I was not expecting to hear such a presentation at a *Science* of Design workshop. And again, I agreed with her arguments that software design can be viewed as an application of economics.

Did the workshop bring forth the view of is software design as a *science*? I did not see much support for this position. Attendees were certainly using science and scientific methods in their investigations. But I found little consensus, let alone support, for software design as a science. The most memorable summary I heard at the workshop was given by Fred Brookes, the 1999 ACM Turing Award recipient. He summarized the conclusions of his working group as "We don't know what we're doing, and we don't know what we've done!".

The results of the workshop were clear: if there is to be a science of software design, it is a very long way off. In fact, it was questionable to consider software design a "science". Although I do not recall hearing this question posed, it seemed reasonable to ask if design is engineering[1]. For example, when bridges are designed, there is indeed an element of artistry in their creation. But there is also an underlying science called physics that is used to determine if the bridge meets its specifications. So if software design is engineering, then what is the science that underlies software design? Again, we are back to square one.

After many hours of thought, I realized that the positions of Gabriel and Baldwin were consistent with my own. Software design is an art as Gabriel argued, *but not always*. Consider the following: designing the first automobiles was an art – it had never been done before, and required lots of trial and error. Similarly, designing the first computer or designing the first compiler were also works of art. There were no assembly lines for creating these products and no automation. What made them possible was craftsmanship and supreme creativity. Over time, however, people began building variants of these designs. In doing so, they learned answers to the important questions of *how* to design these products, *what* to design, and most importantly, *why* to do it in a particular way. Decision making moved from subjective justifications to

[1] Thanks to Dewayne Perry for this observation.

quantitative reasoning. I am sure you have heard the phrase "we've done this so often, we've gotten it down to a science". Well, that *is* the beginnings of a science.

A distinction that is useful for this paper is the following: given a specification of a program and a set of organized knowledge and techniques, if "magic" (a.k.a. inspiration, creativity) is needed to translate the specification into a program's design, then this process is an *art* or an *inexact science*. However, if it is purely a mechanical process by which a specification is translated into a design of an efficient program, then this process follows an *exact* or *deterministic science*.

Creating one-of-a-kind designs will always be an art and will never be the result of an exact or deterministic science, simply because "magic" is needed. Interestingly, the focus of today's software design methodologies is largely on creating one-of-a-kind products. The objective is to push the envelope on a program or component's capabilities, relying on the creativity and craftsmanship of its creators – and not automation. In contrast, I believe that an exact science for software design lies in the mechanization and codification of well-understood processes, domain-expertise, and design history. We have vast experiences building particular kinds of programs, we know the *how*, the *what*, and the *why* of their construction. We want to automate this process so that there is no magic, no drudgery, and no mistakes. The objective of this approach is *also* to push the envelope on a program or component's capability but *with emphasis on design automation.* That is, we want to achieve the same goals of conventional software development, *but from a design automation viewpoint.*

The mindset to achieve higher levels of automation is unconventional. It begins with a declarative specification of a program. This specification is translated into a design of an efficient program, and then this design is translated to an executable. To do all this requires significant technological advances. First, how can declarative specifications of programs be simplified so that they can be written by programmers with, say, a high-school education? This requires advances in *domain-specific languages*. Second, how can we map a declarative specification to an efficient design? This is the difficult problem of *automatic programming*; all but the most pioneering researchers abandoned this problem in the early 1980's as the techniques that were available at that time did not scale [1]. And finally, how do we translate a program's design to an efficient executable automatically? This is *generative programming* [9]. Simultaneous advances on all three fronts are needed to realize significant benefits in automation.

To do all this seems impossible, yet an example of this futuristic paradigm was realized *over 25 years ago*, around the time that others were giving up on automatic programming. The work was in a significant domain, and the result had a revolutionary impact on industry. The result: *relational query optimization (RQO)* [12].

Here's how RQO works: an SQL query is translated by a parser into an inefficient relational algebra expression. A query optimizer optimizes the expression to produce a semantically equivalent expression with better performance characteristics. A code generator translates the optimized expression into an efficient executable. SQL is a prototypical declarative domain-specific language; the code generators were early examples of generative programming, and the optimizer was the key to a practical solution to automatic programing.

In retrospect, relational database researchers were successful because they automated the development of query evaluation programs. These programs were hard to write, harder to optimize, and even harder to maintain. The insight these researchers had was to create an *exact* or *deterministic science* to specify and optimize query