to appear, Future of Software Engineering Research Workshop, at FSE, November 2010

Thoughts on Automated Software Design and Synthesis

Don Batory
Department of Computer Science
The University of Texas at Austin
Austin, Texas, 78712 U.S.A.
batory@cs.utexas.edu

ABSTRACT

I summarize some personal observations on the topic of automated software design and synthesis that I accumulated over twenty years. They are intended to alert researchers to pitfalls that may be encountered and to identify goals for future efforts in advancing software engineering education and research.

Categories and Subject Descriptors

D.2.10 [**Design**]; D.2.2 [**Design Tools and Techniques**]

General Terms

Design

1. INTRODUCTION

My formal education in the late-1970s was in databases (DBs). In the early 1990s, my research migrated to *Software Engineering (SE)*. After a year and with trepidation, I admitted the move publicly to my department. The reputation of SE has since improved, but arguably remains one of the less-respected disciplines in Computer Science. SE is indeed engineering (nothing wrong with that); my core complaint is that the scientific foundations for particular subdisciplines of SE are weak, if not outright shabby. I focus my ire on software design and synthesis, my main interest. (Henceforth I use the term *program automation* for design and synthesis). Equating design with poetry does nothing to enhance its reputation in the eyes of scientists. While there is artistry in bridge-building, artistry is not what is taught in Mechanical Engineering [6]. Nor should it be. The same holds for software design. What distinguished DBs from SE was a communal agreement on fundamental concepts, core problems to solve, and a common vocabulary with agreed-upon meanings. This trifecta remains illusive in broad areas of program automation.

I take two positions in this paper. First, at the core of program automation lies a Science of Design or Algebra of Design that all CS students should be exposed. It may not help programmers in the trenches immediately, but it will provide a long-range conceptual foundation for which existing and new SE technologies can be cast, understood and related. And doing so will begin to shore up the scientific foundations of what we do and what we teach about design so that our students will be more literate, better prepared, and (hopefully) more appreciative of future advances in SE.

Second, there are communal roadblocks that next generation SE researchers may face: fundamentals of program automation, undergraduate education, and visibility and awareness of prior work. All are aimed at improving the scientific foundation and education of our discipline.

FoSER-18, November 7–8, 2010, Santa Fe, New Mexico, USA.

2. FUNDAMENTALS

One of the greatest achievements in program automation, and also software engineering, is *relational query optimization (RQO)* [5]. Prior to RQO, people had to access a database programmatically, which lead to a never-ending list of problems. Ad hoc queries were impractical, programmers had to understand far too many implementation details of data storage to write a program, programs would have to be modified if storage details changed, optimization of query evaluation programs was too costly, and worse, logic errors in query evaluation programs were detected far too late. To eliminate these difficulties required a solution to the *Automatic Programming (AP)* problem. That is, map a declarative specification of a query to an efficient program. This was no small challenge. RQO was considered at a time (late 1970s) when the Artificial Intelligence community was abandoning efforts on AP as it was too hard. The SE community was too nascent back then to have contributed significantly.

The genius of RQO was to represent query evaluation programs as relational algebra expressions. If you represent programs as expressions, you can optimize these expressions using algebraic identities, which in turn meant that you could optimize programs automatically. Voila! A domain-specific solution to AP. Today, database optimizers deal with many more operations than the classical Select-Project-Join, giving testament to the longevity, power, and practicality of the RQO paradigm. In effect, database researchers told us how to organize and conceptualize a domain so that efficient programs in that domain could be generated automatically from declarative specifications.[1]

It took me years to appreciate the significance and generality of RQO: it is a compositional paradigm for program synthesis and it fundamentally imprinted my view of program automation more than any software engineering course (in the 1980s and maybe even now) could have. I feel I can now point to notable successes generalizing this approach [1, 2, 4].

[1]The SQL and QUEL languages enabled declarative specifications of queries. Their compilers mapped queries to relational algebra expressions. This is another fundamental contribution of RQO.

As I look through the collection of SE texts in my office (which admittedly is not voluminous), I find no references at all to this fundamental result. Not one. Even if I missed some, this says volumes to me about how things must change in the future. I wonder what others in SE are thinking about program automation (if at all), and if RQO was ever on their radar.[2]

RQO taught me something fundamental. Software engineers define structures called programs and use tools to transform, manipulate, and analyze them. Object orientation uses methods, classes, and packages to structure programs. Compilers transform source structures into bytecode structures. Refactoring tools transform source structures into other source structures, and metamodels of MDE define the allowable structures of model instances: transformations map metamodel instances to instances of other metamodels for purposes of analysis and synthesis. Software design and development is replete with such examples.

I am not a mathematician. On the contrary, I had no formal mathematical background and built systems just like others to validate ideas. But I needed a language in which to explain my ideas in a natural way. I found the marriage of algebra and program transformations to provide a perfect language to do so. Mathematics is the science of structures and their relationships. I use mathematics as an informal modeling language (not as a formal model) to define and explain the principles behind my work. And it naturally leads to expressing software design in terms of ancient concepts: programs are values, transformations map programs to programs, and operators map transformations to transformations. This orientation effortlessly leads to MDE, with its emphasis on transformations and models. This is a strong argument that we have been using MDE concepts for some time (although certainly not in their most general form). Viewing prior work through an MDE-lens places program automation more on an algebraic footing, and (I argue) is a better foundation for teaching SE concepts to undergraduates and graduates than non-algebraic foundations. Even if students never explore MDE in their undergraduate classes, the transition to MDE when they do see it should be straightforward.

3. UNDERGRADUATE EDUCATION

Experience in program automation makes the following point painfully clear: unless you have a deep understanding of both the conceptual and engineering issues of a domain, you can not seriously automate the essential tasks of design and development. The breadth of my work covers *Software Product-Lines (SPLs)*, layered abstractions, extensible designs, variability management, refactorings, program synthesis, and more recently *Model Driven Engineering (MDE)*. There have been significant advances in all these areas in the last decade, but the number of graduate students that appreciate (or have any clue about) SPLs and MDE still feels miniscule. In other words, they are not seeing these ideas in their undergraduate education. Magnify this with the fact that if students do not encounter an idea (such as SPLs or MDE) prior to entering the work force, they will be imprinted with legacy concepts that — in my experience — make it harder to adopt and/or advance SE technology later on.

Even worse, I do not see non-SE graduate students (who come to UT from other universities) applying standard SE knowledge in their work, say, of software architectural designs (e.g. pipes-and-filters) that were explored 15 years ago. Students do use the ideas — but the historical trail, terminology, and knowledge of that work seems to have been forgotten or maybe were even never known to them. This again suggests to me that it is time to refresh or rethink our undergraduate education in SE so that these "holes" are plugged.

A case in point: consider the future prominence of program automation. A domain is ripe for program automation when its body of knowledge and software implementations are well-understood. Ten years ago, there were many such domains. Now there are more. Ten years from now, there will be more still. Program automation will be familiar to SE practitioners in the future. What about automation should we teach? Tools and hacking are a poor start.

Algebraic principles should be driving force in teaching program automation much like relational algebra is the driving force behind query optimization. Domain analysis — the process by which a domain is codified — should follow classical activities that are undertaken in physics and chemistry: one develops theories that explain and predict — using as few concepts as possible — a set of disparate phenomena. I am not talking about the "wind, fire, water, earth" theory of the Greeks, which is best qualitative. Nor am I talking about the broad, informal surveys of what has been done or could be done that is popular today. Rather, I refer to modern theories that have an algebraic core (compositional, automatable) where mathematical reasoning — possibly including formal verification — underpins design reasoning. The broader the set of phenomena covered, the fewer the core concepts, the better. A theory is evaluated by implementing a generator to synthesize programs that exhibit predicted properties. Of course, there is a practical benefit to all this: tasks of programming that are well-understood are mechanized and higher-quality software is produced at less expense. To me, this is a core Science of Design activity [3] that will become progressively more important in undergraduate education, e.g. how to think about program automation and, later given appropriate background, how to use and improve automation tools.

I feel strongly that future research in SE cannot just be motivated by the need to solve immediate or newly arising technical problems. Solving such problems is admittedly important, but they are not the only problems of substance. Much of what I have seen in my 20+ years that passes for great SE are virtuoso performances — often by people outside the community — that expose researchers to new challenges that lie ahead. But we should also find ways of moving research results into our undergraduate classes.

Matthias Felleisen observed that as a community, we are good at finding research problems to keep us busy for years at a time. Equally important is research that keeps our undergraduates busy. Jay Misra noted that the quality of a research area is measured by the quality of its teaching material. We should follow their advice: the quality of a research area is measured by the quality of its undergraduate teaching material. Ask yourself (for a given paper or area) is there material that is worth presenting to undergraduates? If not, then perhaps something is wrong.

4. WHAT HAVE WE DONE?

I attended an NSF workshop in 2003 on the "Science of Design: Software Intensive Systems" where after a break-out session, Fred Brooks expressed frustration to what I have seen as a general problem: "How do we know what we are doing if we don't know what we have already done?" I know of research communities that are prone to reinvent the wheel. I worry that software engineering research matures at a slower rate than it does in other disciplines, because there may not be standard names for problems or concepts

[2]I recall in the early-1990s telling a pioneer in the SE community (who will remain nameless) that I was glad I did not know much about SE when I started my work on SPLs and program generation. The contemporary ideas in SE back then were irrelevant to RQO, in my opinion, and would have been put me on the wrong track. Of course, my statement went over like a lead balloon, but it still retains an element of truth today.

(making it harder to find related work), key results often do not appear in flagship conferences or journals (again, making it harder to find related work), and the role of journals in today's research climate may be diminishing. All reduce the impact of tomorrow's software engineering research, and this should be concern to all.

I was fortunate to have been a member of the 2003 ICSE program committee. When I received my batch of papers to review, I wondered what fraction of their citations (i.e. references to prior work) were to earlier ICSE and ACM FSE (the flagship SE conferences), and to ACM TOSEM and IEEE TSE (the flagship SE journals). Table 1 summarizes the findings.

Total Papers Submitted	311	
Citations Per Paper	21.3	
IEEE TSE Citations	298	4.5%
ACM TOSEM Citations	49	0.7%
ICSE Citations	202	3.0%
FSE Citations	48	0.7%
Others		91%

Table 1: Statistics on ICSE 2003 Submissions

I was surprised to see that FSE and TOSEM had tiny percentages. We all know how difficult it is to get publications in these venues, so I began to wonder if these the venues had the highest impact? Were the 2003 statistics outliers? Again with the help of my students, we collected comparable statistics on the accepted papers from two prior ICSEs (2000 and 2001). Table 2 shows the results:

	ICSE00	ICSE01
Total Papers	51	60
Citations Per Paper	19.9	21.0
IEEE TSE Citations	7.5%	4.5%
ACM TOSEM Citations	1.6%	0.8%
ICSE Citations	5.7%	4.7%
FSE Citations	1.1%	1.7%
Others	85%	88%

Table 2: Comparison of Prior ICSE Proceedings

In short, not much difference. FSE and TOSEM account for a negligible percentage of citations; ICSE and TSE account for about 10%. Note: TOSEM publishes about 12 papers a year, TSE 48. The factor 4 difference seems to hold for their relative citation rates (1% vs. 5-6%), which means TOSEM papers are on average cited as often as TSE papers. Never-the-less, 85-90% of papers cited were not from flagship venues didn't seem right.

Given my database background, I was curious to know similar statistics about my former community. At the time ACM TODS (Transactions on Database Systems) and ACM TKDE (Transactions on Knowledge and Data Engineering) were the flagship journals and SIGMOD, VLDB, and PODS the flagship conferences, where PODS was the more theoretical of the three. See Table 3.

30-40% of citations were to publications in premier DB venues, in contrast to 10-15% for SE. Note that citations to database journals weighed in at a paltry 1.7%, raising not only the question of the relevance of database journals, but journals in general.

In writing this paper, I wondered what had changed in the intervening years. So I grabbed a digital copy of the ICSE 2010 proceedings, and did a quick analysis summarized in Table 4.The citation numbers from flagship venues are better, rising from 10% to

	SIGMOD00	SIGMOD01
Total Papers	42	29
Citations Per Paper	23.5	20.7
ACM TODS	1.6%	1.2%
ACM TKDE	0.1%	0.2%
SIGMOD	20.7%	10.4%
VLDB	12.8%	5.6%
PODS	1.7%	10.6%
Others	62%	72%

Table 3: Statistics for ACM SIGMOD Conferences

20%. But this is not the whole story: there are many citations to papers in OOPSLA, ECOOP, ASE, GPCE, PLDI, POPL, and ISSRE which, not counted in Table 4, should raise the citation percentages to respectable levels. And of course, this is not a proper statistical survey; it represents just a few sample points.

Total Papers Accepted	54	
Citations Per Paper	27.1	
IEEE TSE Citations	84	5.7%
ACM TOSEM Citations	11	0.7%
ICSE Citations	127	8.7%
FSE Citations	24	4.8%
Others		80%

Table 4: Statistics for ICSE 2010

This raises a set of interesting questions that I hope someone (eventually) will answer: Are important papers so widely dispersed across different venues that it delays dissemination to others? Are flagship conferences bad at focussing the attention of people on the big ideas of the next few years? Are these conferences hurting our community because the topics that are covered are simply too broad and too numerous? Is it better for academic promotion to publish in lower-tier conferences where impact may be greater because the community is more focused?

How could we do a better job of disseminating results? One way to promote better awareness of prior work is to have digital libraries list the most significant papers in recognized subdisciplines of SE. As the publishers of these papers (ACM, IEEE, Springer, etc.) will likely be different, only links to these papers may be posted. But at least, there would be a clearing house of key papers that all should be aware. Committees that are representative of subdisciplines would vote to recognize these papers. (How to keep politics and religion out of this is not obvious). An indirect benefit might help bring different communities closer together.

5. CONCLUDING THOUGHTS

I believe the time has come to step back and see how we can improve SE as a discipline. We should ask if the relative lack of structure and coordination among the myriad SE conferences and journals is hurting us because key papers, terms, and concepts that define an area are sometimes hard to identify. We should re-examine what we teach at the undergraduate level, and pay more attention to updating our SE curriculums in light of more modern thinking. We should recognize the future importance of program automation and begin preparing our undergraduates for this future. And we should re-examine thoughts on design and move its foundations away from art and poetry toward more algebraic foundations, (ala query optimization) which will be the core of program automation.

Acknowledgements: I gratefully thank Mark Grechanik, Taylor L. Riché, Jay Misra, and Dewayne Perry for their comments on an earlier draft. I also appreciate the input of FoSER referees. This work was supported by NSF's Science of Design Project CCF-0724979.

6. REFERENCES

[1] D. Batory, M. Azanza, and J. Saraiva. The Objects and Arrows of Computational Design. In *Keynote at Model Driven Engineering Languages and Systems (MODELS)*, Oct. 2008.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, June 2004.

[3] N. S. Foundation. Science of design program. `http://www.nsf.gov/pubs/2005/nsf05620/nsf05620.htm`.

[4] G. Freeman, D. Batory, G. Lavender, and J. Sarvela. Lifting Transformational Models of Product Lines: A Case Study. *Software and Systems Modeling*, 9:359–373, 2009.

[5] P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Databasse System. In *ACM SIGMOD*, 1979.

[6] W. G. Vincenti. *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*. Johns Hopkins University Press, 1990.