Why (Meta-)Theories of Automated Software Design Are Essential: A Personal Perspective

Don Batory
Department of Computer Science
University of Texas at Austin
Austin, Texas, USA
batory@cs.utexas.edu

***Abstract*—Program generators are tools that automatically construct customized programs in a particular domain. Generators mechanize implicit "theories" of how a domain expert would go about writing an efficient program. Abstracting the core activities of a domain expert and automating them is analogous to creating and evaluating theories in physics and other natural sciences. Theories have a revered place in natural sciences; eventually theories will assume a comparable place in automated software design. The reason is simple economics: generators will remove the burden of difficult or mundane tasks from an engineer to a machine.**

***Index Terms*—automated software design, relational query optimization, semantic modularity, features, generators.**

I. Theory and Software Engineering

Consider the first two definitions of "science" from dictionary.com:

1) a branch of knowledge or study dealing with a body of facts or truths systematically arranged and showing the operation of general laws: the mathematical sciences.
2) systematic knowledge of the physical or material world gained through observation and experimentation.

The dominant paradigm today in *Software Engineering (SE)* is for referees to insist on a rigorous hypothesis evaluation of a proposed technique. A set of tests (observations) must be conducted by an author and a careful analysis of one or more hypotheses must be presented. This is the scientific method. It closely matches Definition 2 and the intended use of experimental methods in SE. To me, these are "pre-theory" activities.

To put this into perspective, a colleague once told me: "Empirical studies helped design spacecraft, but it was the theory of gravity that took us to the moon". Theories are the big ideas in science, not empirical studies. Empirical studies help shape and determine the validity of laws, and may indeed trigger the development of new and improved theories. But the big ideas (to me) are the theories.

II. Theories of Software Design

A colleague once asked me: "What could be more interesting and more fun than writing a program?" His answer: "Writing a program that writes other programs". The depth of this challenge belies its simple description. Such a program $\mathcal{G}$ must be able to produce many programs that vary in predetermined ways. $\mathcal{G}$ must have an input language—however primitive—for users to specify what program to output. Ideally, the specification is declarative, much like the way people select their dinners from restaurant menus or select features to identify a product to buy. $\mathcal{G}$ must be able to reason about a specification and understand how to map it to an efficient implementation. In the late 1970s, this challenge was given a name: *automatic programming*. The initial attempts to solve it provided a sobering glimpse of its difficulty [1].

It is common in physics for there to be different and poorly-related phenomena. A theoretical physicist would select a set of phenomena and seek a theory that unifies them as manifestations of the same underlying concepts. The broader the initial set, the fewer the concepts, the more general and significant the theory might be. An initial test of a theory is to check that it does precisely what it claims—not only reproduce or explain the phenomena of the initial set, but also explain and predict other phenomena as well.

The phenomena of interest to SE are programs with certain properties, and $\mathcal{G}$ is a program *generator* that is a concrete mechanization of an "implicit" SE theory for constructing domain-specific programs.

III. Meta-Theories of Software Design

History and experience has shown that such SE theories must be domain-specific to have any chance of success. Domain-specific design knowledge is often rich and deep, with few specifics transferable to other domains. It is somewhat ironic then that domain-specific theories are uninteresting to the general SE community. *Meta-theories* are more valued as their instances are domain-specific theories from which domain-specific generators can be developed. A meta-theory identifies *domain-independent* concepts or a framework to instantiate to create proper theories; these are the concepts that should be taught to our students; they will instantiate meta-theories to produce domain-specific generators of their own.

Meta-theories have been a part of SE education for years, although existing examples are informal and not very automatic or mathematical. Consider *object-oriented (OO)* frameworks [2], which are common in today's software libraries. Framework designers understand that a set (a.k.a. domain) of similar programs will be built frequently. They create an OO

framework to code the common objects and activities of a domain to minimize what others have to write. The concepts behind frameworks are fundamental (this is the meta-theory part), we teach these (meta-theory) concepts, and our students instantiate the concepts to create frameworks of their own.

UML is another example [3]. It asserts that an OO design can be documented in the languages of class diagrams, state machines, etc. (this is the meta-theory part). We teach UML (meta-theory) concepts to our students; they in turn, instantiate these concepts to design OO programs of their own. Meta-theories do indeed exist in today's SE curriculum. But meta-theories that focus on *automatic programming* ($\mathcal{G}$ programs) are hard to find.

It is unclear if automatic programming (meta-)theories were ever really part of core SE research. Key papers originally appeared in distant conferences (knowledge engineering, software reuse, artificial intelligence, programming languages, etc.) rather than flagship SE conferences. And for good reason: not everyone is interested in domain theories and meta-theories. Meta-theories tend to deal with concepts that are foreign to main-stream software engineers. Further, $\mathcal{G}$ programs—and what it takes to build them—are not the focus of popular SE texts and today's SE curriculum. Broadly speaking, a good SE text provides a well-organized recitation of proven SE techniques and analyses, and rarely (if at all) theories of automated software design.

Case-in-point: The most significant advance in automatic programming is *relational query optimization (RQO)*, ironically accomplished in the late 1970s when most others were giving up on automatic programming in droves [4]. A user writes a data retrieval specification as a declarative SQL query; an SQL parser maps a query to an inefficient relational algebra expression. An optimizer uses algebraic identities to rewrite the expression, never changing the semantics of the original, to find a more efficient way to execute it. A code generator translates an optimized expression to executable code. This is an elegant solution to automatic programming. RQO revolutionized databases, bringing it out of the stone-age 1960s to the omni-present and sophisticated technology we know today. Yet, find one contemporary SE text that explains RQO, its paradigm, or its connection to automated software development. I have not found a single text. Not one. It is as if the result or topic did not exist.

It seems evident that automating the development of well-understood software should be a prime goal of SE—capturing and mechanizing the knowledge of domain experts so others can benefit. But we do not teach design (meta-)theories for automation. So why should we be surprised that such (meta-) theories have had little impact or are hard to find?

Another point: theories are *not* small results—they are not new algorithms or new engineering techniques that one can "evaluate" easily. Theories are most effective for well-understood domains. Even so, theories often take a long time to develop; their generators can take months or years to build. They embody new ways of thinking about old problems. It takes time and effort to understand their strengths and limitations. To evaluate a theory properly can take years or decades—it cannot be done in a single paper (unless to show where the theory is wrong). Pre-theory and theory activities seem substantively different; it is not clear that they should be evaluated in the same way. I suspect that they are.

Offhand, what is an indicator of a good theory or meta-theory? I have found that if you can explain complex designs in a simple way, you're on the right track. Further, external indicators of success are comments like:[1]

1) Ok, but so what? What's the difficulty?
2) That's nice. But I can't see how it generalizes to anything of interest to me.
3) My software is too complicated for this to work.

Comment 3) is reminiscent of a point Tony Hoare made in his 1980 Turing Award Lecture [5]:

> *There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.*
>
> *The first method is far more difficult.*

Yet another irony: creating a $\mathcal{G}$ program is a good (if not great) *engineering* achievement. *Simplicity* counts. *Elegance* matters. Two words that one does not hear enough about SE results.

In summary, our current education system produces exactly what SE expects: software engineers. I ask the question: are we producing the scientists of tomorrow that we need? Are we producing engineers and future leaders who have an appreciation of theories? If your answer is "no" to either of these questions, then the theories I have described will be under-appreciated for years to come.

IV. What might a meta-theory look like?

Developing meta-theories for automated software development has been the focus of my career. I believe a small number of core concepts underpin powerful and practical meta-theories. My guide is the firm belief that programmers (including me) are geniuses at making the simplest things complicated; finding their underlying simplicity is the challenge.

The key question is how to express a meta-theory or theory? I use algebra. I was not educated as a mathematician; my educational background in the 1970s was software systems engineering (where mathematics played a non-existent role). I adopted mathematics as it was the only sensible way to explain my discoveries and ideas. I have since recognized a deeper reason: programs are sophisticated *structures*. Tools of a software engineer manipulate these structures: compilers map source code to byte code; refactoring tools restructure source code; *Model Driven Engineering (MDE)* is all about transforming models of one type to models of another. SE is replete with such examples. Mathematics is the science of structure and structure manipulation. Given this, it is not a big intellectual leap to believe there must be a fundamental

[1] Often these comments are part and parcel of negative reviews.

connection. Frankly, the use of mathematics should come as no surprise to any scientist or engineer outside of SE; within the sub-discipline of software design there is a very limited embrace of this connection.

My work has centered on *semantic modularity*—the modularization of semantic changes (typically increments) in program functionality. Semantic modularity is not code modularity: if you add new functionality to a program, you have to update a program consistently in lots of different places. And if you remove this functionality, all of these updates must be removed simultaneously, much like a database transaction. Modularizing such changes as an atomic unit is the goal.

Semantic modularity—called *features*—has been known for over 20 years [6]. Different communities have pursued their own agendas, terminologies, and distinctive takes on these ideas. With few exceptions, most do not express their meta-concepts algebraically.

To give a sense of what I'm talking about, I briefly illustrate two core ideas that are elegantly expressed algebraically, and of an algebraic theory that has had a modicum of success in *Software Product Lines (SPLs)*.

A. An Example of An Algebraic Meta-Theory

1) Basics of SPLs: A set or domain $\mathcal{D}$ of programs can be constructed from a feature set $\vec{\mathcal{D}} = \{\mathtt{F}_1 \ldots \mathtt{F}_n\}$. Each program in $\mathcal{D}$ is identified with a unique combination of features. Features are composed by an abstract operation $+$. So each program $\mathtt{P} \in \mathcal{D}$ is compactly written as the sum of a unique set of features (a.k.a. functionalities) from $\vec{\mathcal{D}}$, *e.g.* $\mathtt{P} = \mathtt{F}_4 + \mathtt{F}_3 + \mathtt{F}_1$, called a *feature expression* [7].

2) Implementations and Homomorphisms: Programs have many concrete representations: source code σ, documentation δ, makefile μ, etc. We want to construct each by module composition.

Suppose $\mathtt{P} = \mathtt{F}_4 + \mathtt{F}_3 + \mathtt{F}_1$. The source code of $\mathtt{P}$, namely $\sigma(\mathtt{P})$, is constructed by code-composing ($\oplus$) the code modules for each of $\mathtt{P}$'s features:[2]

$$\sigma(\mathtt{P}) \quad = \quad \sigma(\mathtt{F}_4 + \mathtt{F}_3 + \mathtt{F}_1) \quad = \quad \sigma(\mathtt{F}_4) \oplus \sigma(\mathtt{F}_3) \oplus \sigma(\mathtt{F}_1)$$

That is, we translate a feature expression into a source-code module expression to synthesize $\mathtt{P}$'s code. Mapping an expression in one algebra to an expression in another is a *homomorphism* [8]. Homomorphisms are at the core of recent SPL results, reviewed next.

3) Recent Instances of the Meta-Theory:

- Siegmund et al. [9] showed how to compute a performance estimate π for a given workload for any $\mathtt{P} \in \mathcal{D}$. Procedures were given to estimate the delta in performance that each feature contributes to a program. Assuming performance estimates of features are arithmetically added, their work relied on the identity:

$$\pi(\mathtt{A} + \mathtt{B}) \quad = \quad \pi(\mathtt{A}) + \pi(\mathtt{B})$$

  Surprisingly accurate predictions were reported using this simple approach.

- Apel et al [10] showed how different program representations can be encoded as syntax-trees and feature composition maps to syntax-tree composition. Given the grammar of a language λ and rules for composing λ syntax-trees, FeatureHouse generates a tool that implements the following homomorphism:

$$\lambda(\mathtt{A} + \mathtt{B}) \quad = \quad \lambda(\mathtt{A}) \;\; +_\lambda \;\; \lambda(\mathtt{B})$$

  That is, the generated tool parses the λ modules for $\mathtt{A}$ and $\mathtt{B}$ and composes them with the syntax-tree composition operation $+_\lambda$.

- The most sophisticated use to date of homomorphisms is by Delaware et al. [11], who showed how proofs of correctness of a program could be synthesized from its feature expression. The target domain, $\mathcal{FJ}$, contains dialects of Featherweight Java. An integral part of any type system are the meta-theoretic proofs that show type soundness—the guarantee that the type system statically enforces the desired run-time behavior of a language, typically preservation and progress.[3] Four different representations of each feature—syntax, typing rules for preservation, evaluation rules for progress, and the proofs—were encoded as separate modules in the Coq proof assistant [12]. The δ homomorphism (1) composes syntax, typing rule, and evaluation rule modules and the ψ homomorphism (2) composes proof modules, each operation implemented by a Coq library [13]:

$$\delta(\mathtt{A} + \mathtt{B}) \quad = \quad \delta(\mathtt{A}) \;\; +_\delta \;\; \delta(\mathtt{B}) \qquad (1)$$
$$\psi(\mathtt{A} + \mathtt{B}) \quad = \quad \psi(\mathtt{A}) \;\; +_\psi \;\; \psi(\mathtt{B}) \qquad (2)$$

  Each distinct module for feature syntax, feature typing rules, etc. is certified once by Coq (this is the expensive part) and reused as-is. Coq mechanically verifies the correctness of a composite proof by a simple interface check.

4) And More: There is over two decades of evidence that features can be used to specify programs in diverse domains. There is considerable evidence that the meta-theory described above has wide applicability.

There are many extensions to this meta-theory: the inclusion of feature interactions [14], feature models that define legal feature expressions (as not all features are compatible) and their relation to propositional formulas [15], and hierarchically recursive applications of algebras (that these same concepts apply at all levels of abstraction) [16]. As limitations are discovered, generalizations of this meta-theory are proposed, just like theories in physics and other natural sciences.

[2] Or more generally: $\sigma(\mathtt{A} + \mathtt{B}) \;=\; \sigma(\mathtt{A})\; \sigma(+)\; \sigma(\mathtt{B})$.

[3] *Preservation* says if expression $\mathtt{e}$ of type $\mathtt{T}$ evaluates to a value $\mathtt{v}$ then $\mathtt{v}$ also has type $\mathtt{T}$. *Progress* says expression evaluation does not get "stuck", i.e. there are no expressions that cannot be evaluated.

B. More General Meta-Theories

As in physics where theories are special cases of more general theories, the same holds here. Semantic modules can also be understood as program transformations—mappings of one program (representation) to another (representation). More general meta-theories are elegantly grounded in *category theory (CT)* [8]. It has been shown that SPLs and MDE are different manifestations of the axioms of CT, but at different levels of abstraction [17].

Having said the above, I know what most people who read this must be thinking. A few years ago a colleague said to me: "Using category theory is the kiss of death", meaning anything connected to CT is the perfect way to kill a line of research. This is understandable: existing texts on CT are impenetrable because they give impractical examples for software engineers to appreciate and understand. It takes effort and the right set of examples to bridge the gap.

Let me also remind critics that the relational data model was based on set theory—roughly the first couple pages of a set theory text. This was of great disappointment to mathematicians, but it was exactly the right language and exactly the right level of simplicity that database researchers grew to appreciate.

My prediction is that elementary CT will play a comparable role in automated software development, just as elementary set theory played a foundational role in relational databases. You don't have to be a mathematician to appreciate the impact of set theory on databases; the same will hold for CT and automated design. So to critics who say CT is irrelevant to software design: it is time to leave the dark ages.

V. Why (Meta-)Theories are Essential to SE

An essential activity of SE is program design. Program design involves abstraction: it is a process of distinguishing essential ideas from non-essential. A generator scales abstraction to a family of domain-specific programs. Meta-theories scale abstraction further to diverse domains, thereby laying the groundwork for a more economical production of programs.

Just as programs should not be hacked, generators should not be hacked. Both require thought and effort. The reasons why a generator works in one domain is likely the same reasons why other generators have worked in others. Meta-theories can be deep intellectual excavations to understand the reasons for why programs work the way they do, and that they do not work by accident.

A meta-theory will tell you how your tools should work. It will tell you that certain fundamental identities (*e.g.*, homomorphisms) must hold otherwise your tools, designs, or ideas are wrong [18]. In mature technical communities, there is an accepted way to think about problems and how to formulate solutions (*e.g.* type systems for programming languages, relational algebra and sets for databases). Meta-theories bring organization to what would otherwise be intellectual chaos.

VI. Closing Thoughts

The big ideas in science are theories, not empirical studies. Still, algebraic meta-theories will be resisted for many reasons. The primary reason is that people will need to learn something new and to appreciate the value in doing so. But isn't this the substance of scientific advances? SE today largely practices "pre-theory" science. The more important half of science remains to take its place in SE education, history, and discourse.

Acknowledgments. I am grateful for conversations with B. Delaware, P. Höfner, W. Lawvere, C. Lengauer, B. Marker, M. Myers, and B. Möller in shaping my view of algebra and the contents of this paper. I also gratefully acknowledge support for this work by NSF projects OCI-1148125 and CCF-1212683.

Any opinions, findings and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the NSF.

References

[1] R. Balzer, "A 15 year perspective on automatic programming," *IEEE Transactions on Software Engineering*, 1985.
[2] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, 1998.
[3] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2010.
[4] P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," in *ACM SIGMOD*, 1979.
[5] C. A. R. Hoare, "The emperor's old clothes," *CACM*, 1991.
[6] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (foda) feasibility study," 1990, cMU/SEI-90-TR-021.
[7] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM TOSEM*, 1992.
[8] B. Pierce, *Basic Category Theory for Computer Scientists*. MIT Press, 1991.
[9] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov, "Scalable prediction of non-functional properties in software product lines," in *SPLC*, 2011.
[10] S. Apel, C. Kästner, and C. Lengauer, "Featurehouse: Language-independent, automated software composition," in *ICSE*, 2009.
[11] B. Delaware, W. Cook, and D. Batory, "Theorem proving for product lines," in *OOPSLA/SPLASH*, 2011.
[12] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.
[13] B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers, "Meta-theory à la carte," in *POPL*, 2013.
[14] D. Batory, P. Höfner, and J. Kim, "Feature Interactions, Products, and Composition," in *GPCE*, 2011.
[15] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *SPLC*, 2005.
[16] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE TSE*, Jun. 2004.
[17] D. Batory, M. Azanza, and J. Saraiva, "The Objects and Arrows of Computational Design," in *MODELS*, 2008.
[18] G. Freeman, D. Batory, R. G. Lavender, and J. N. Sarvela, "Lifting transformational models of product lines: a case study," *Software and System Modeling*, 2010.