

NSF Workshop Position Paper

Science of Design – Software Intensive Systems

David Stotts

Department of Computer Science

University of North Carolina at Chapel Hill

<http://www.cs.unc.edu/~stotts/>**Position Summary**

If system design is to be a science, then it has to involve quantification, observation, measurement, analysis, deduction. It must be elevated from qualitative considerations alone and removed from an atmosphere of the practice of an art. We are applying these principles to the base levels of software design now... and moving our findings out into newer areas, such as aspect-oriented programming, that are moving our abilities to construct larger and more complex systems beyond what is practical with OO software technologies.

Design of large system requires clear semantics for expressing design characteristics and analysis of the design. Even current agile methods, many of which advocate design evolution rather than *a priori* full design, require some way to tell if an evolving design has gone astray... and hence requires re-factoring or re-design.

We envision a design calculus to provide the semantic foundation for knowledge encoding, knowledge capture, and reasoning about design. This would provide the basis for analysis tools, as well as agents and assistants, and also provide a knowledge base for design training and communication. A design calculus would provide the semantic foundation for automated refactoring tools -- tools that allow one design, expressed in one code realization, to be re-expressed in different code while still maintaining the proper computational behavior and still meeting the proper design constraints. While this is but one of many aspects of design for large software-intensive systems, it is an important one for quantifying design and moving it towards a science.

We are doing this now for OO design at the basic code level [1,2,3]. We use unambiguous semantics (modified sigma-calc) and automated inference (OTTER theorem prover) to capture (via compiler analysis) design characteristics of code and decide if an evolving design is adhering to the authors' intentions. To emphasize that our research goals are to create analytical methods that lead to practical applications, the following section summarizes our SPQR tool-set and its semantic foundations.

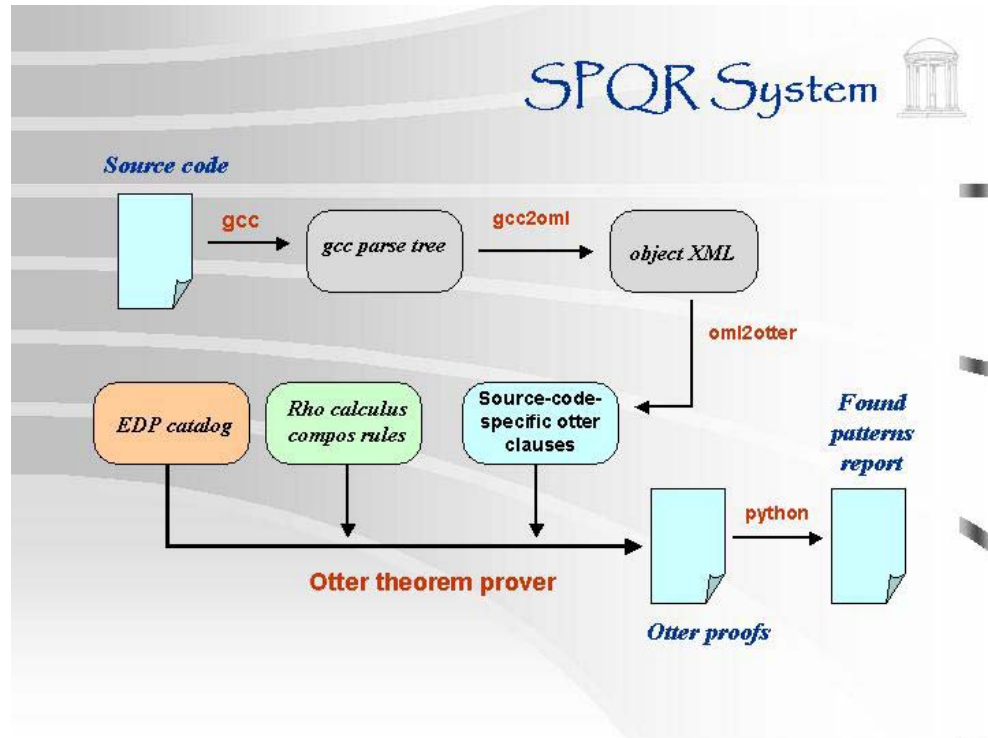
Research Background: Automating Design Discovery and Validation

OO design patterns [5] have become a common vocabulary in which software developers discuss and communicate design and architecture ideas. Finding design patterns in source code helps in maintenance, comprehension, refactoring and design validation during software development. SPQR (System for Pattern Query and Recognition) [1,2,3] is a toolset for the automated discovery of design patterns in source code. SPQR uses a logical inference system to reveal large numbers of patterns and their variations from a small number of definitions. A formal denotational semantics is used to encode fundamental OO concepts (which we term elemental design patterns), and a small number of rules (which we call reliance operators) for combining these concepts into larger patterns. These reliance operators, when combined with the sigma-calculus [5], provide a formal foundation we call the rho-calculus.

SPQR improves on previous approaches for finding design patterns in source code. Other systems have been limited by the difficulty of converting something as abstract as design patterns into concrete expressions without being overly restrictive. A single design pattern when reduced to concrete code can have myriad realizations, all of which have to be recognized as instances of that one pattern. Other systems have had difficulty due to their reliance on static definitions of patterns and variants. SPQR overcomes this problem by using an inference system based on core concepts and semantic relationships. The formal foundation of SPQR defines base patterns and *rules for how variation can occur*; the inference engine is then free to apply variation rules in an unbounded manner. A finite number of definitions in SPQR can match an unbounded number of implementation variations.

The semantic foundation of SPQR is composed of two parts: the fundamental concepts of object-oriented programming and design (*Elemental Design Patterns*), and the rules for their variation and composition (*rho-calculus*). The EDPs were deduced through careful analysis of the Gang of Four (GoF) design patterns [4] for use of core object-oriented language concepts (such as inheritance, delegation, recursion, etc); this analysis produced 11 EDPs from which the GoF patterns can be composed. It also produced a design space that was filled out to produce 16 comprehensive EDPs. We speculate that these additional EDPs will prove useful in defining design patterns from other sources

The SPQR tool chain is shown in the right-side figure. From the engineer's point of view, SPQR is a single automated tool that performs its analysis from source code and produces a final report. A script chains together several modular component tools, centered around the tasks of *source code feature detection*, *feature and rule description*, *rule inference*, and *query reporting*. Compiler output of the syntax tree of a code base (one current input source is gcc) is transformed into our



formal notation encoded as input to the OTTER theorem prover [6]. This fact set is combined with the pre-defined rho-calculus encodings for the EDP Catalog and the relationships rules of rho-calculus to form the knowledge base for the system. OTTER operated on this base and finds instances of design patterns, which are then reported to the user in terms of the code from which the facts came. SPQR is language and domain independent, and would be impractical without the formal foundations of our EDPs and rho-calculus.

References

- [1] Smith, J., and D. Stotts, "SPQR: Flexible Automated Design Pattern Extraction from Source Code," *IEEE Conf. On Automated Software Engineering 2003*, Montreal, Oct. 6-10, 2003, accepted to appear.
- [2] Smith, J, and D. Stotts, "Elemental Design Patterns: A Formal Semantics for Composition of OO Software Architecture," *27th Annual IEEE/NASA Software Engineering Laboratory Workshop*, Greenbelt, MD, Dec. 5-6, 2002, 8pp.
- [3] Smith, J, and D. Stotts, "Elemental Design Patterns and the Rho-calculus: Foundations for Automated Design Pattern Detection in SPQR", UNC CS Dept. [TRO3-032](#), submitted to *Int'l Conf. On Software Engineering 2004*.
- [4] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison Wesley, 1995.
- [5] Abadi, M., and L. Cardelli, "A Theory of Objects," Springer-Verlag, New York, 1996.
- [6] McCune, W., "Otter 2.0 (theorem prover)," in M.E. Stickel, ed., *Proc. Of the 10th Int'l Conf. On Automated Deduction*, pp. 663-664, July 1990.