Research Statement

Yuanfang Cai

My primary research goal is to enable quantitative and automatic reasoning about software design structure and related outcomes early in the development process.

The goals of software engineering research include reducing the cost of software development and evolution, reducing the time to complete development tasks, and improving software quality. To a significant degree, outcomes in all of these dimensions depend on design structure. In particular, designers seek to structure their systems to better accommodate expected changes, have parts that can be developed and evolved without further coordination, and to ease the understanding of complex designs through abstraction of details hidden within modules. My major research interest is to reason about how software design structure influences related outcomes.

The problem that I address is that current design representations and analysis techniques do not enable software engineers to reason effectively about design structures or their implications in these dimensions, especially early in the development process—before code has been produced. We have various ways to analyze program structures once code is written. However, in many cases, designers have to answer some important questions before the system is implemented. For example, will my design accommodate envisioned changes easily? Given a project deadline, is it worthwhile investing in making a system more flexible, for example, by refactoring? Today's design modeling notations, such as the Unified Modeling Language (UML), were not designed to support, and do not adequately support, design analysis of this kind. My research goal is to develop design representations and automated, quantitative analysis techniques that significantly improve engineers’ ability to reason about design structure and related outcomes early in the development process.

My major contributions to date include: (1) an approach to extracting dependence structures from abstract design models; (2) a model to substantiate the concept of information hiding as a measurable criterion for abstract designs; (3) an approach to quantitatively analyzing the changeability of a design represented using my logical design model. My research has imported existing engineering techniques and economic analysis into software [FSE 2001]. In a joint effort with several colleagues, my approach was applied to modeling and analyzing a new form of information hiding interfaces for aspect-oriented programming [FSE 2005]. I have formalized these techniques and implemented a prototype tool called Simon [ASE 2005] to automate these abilities. The evaluation of the framework in a set of case studies has produced promising results.

1 Formal Modeling and Automated Analyses

In my dissertation research, I contribute an approach with three main components. I first model high-level software designs formally, explicitly representing both environment conditions and formerly implicit design decisions. I also contribute an abstract representation that captures how design configurations evolve within a design space, driven by changes to individual design decisions, under design constraints. Third, I show that the formal models of software design and design evolution enable a series of dependence and evolution analysis techniques that are rigorous, quantitative, and automated. Finally, I import existing engineering techniques from other design fields to benefit software designs.

1.1 Formal Design and Evolution Modeling

Software evolution is usually driven by the changes in environment conditions and design decisions. However, prevailing software design models, such as UML, do not support the modeling of environment conditions and important design decisions. We need more general design modeling techniques to support the changeability analysis of software designs. I borrowed the idea of logical constraint networks (CNs) to model software designs.
However, there are at least two design phenomena that are indispensable in any design activity, but are not readily expressed using a pure constraint network: the aggregation of decisions into modules and the dominance relation among design decisions. For example, agreed interfaces often dominate decisions within modules. I augmented a constraint network with additional data structures that model these phenomena and achieved an analyzable software design model, which I call an Augmented Constraint Network (ACN).

Design abstraction itself is not enough to analyze design evolution phenomena. Although we have plenty of ways to abstract a design, we lack abstract models to handle the complexity of design evolution. I created a design evolution model called the Design Automaton (DA) to represent design variations driven by the changes of environment conditions or design decisions. I also devised an algorithm to derive DAs from abstract design models. The abstraction of software design and design evolution has enabled me to create a series of analysis techniques that answer important design questions. I introduce two of them in the next sections.

1.2 Automated Dependence Structure Derivation

My work first enables automatic dependence structure extraction from abstract design models. Dependence structure is the key to design modularization and evolution. People have explored numerous ways of deriving dependence structures from source code. However, the dependence structures in high-level design models, which precede coding and determine the quality of implementation, have not been well studied. Based on our abstract design model, ACN, and evolution model, DA, I contributed an algorithm to derive the dependence structures from high-level design models, and implemented it in Simon [ASE 2005].

I also explored the relationship between source code dependence structures and design dependence structures [ECOOP 06 submission]. I showed that source code dependence structures have limited capability to serve as proxies for design dependence structures. The former are seriously limited by the language in use and the representation granularity of the extraction tool. Important design dimensions, design rules and decisions that cannot be captured by program constructs were missed. Design dependence structures derived by Simon can be a valuable complement for the design structures recovered from source code.

1.3 Automated Design Impact Analysis

Another type of analysis enabled by my framework is design impact analysis (DIA). This analysis intends to address a number of important questions in design: given a design and a sequence of expected changes, what are the consequences? How many ways are there to accommodate the changes? Based on our abstract design evolution model, I formalized the problem, solved it formally, and implemented it in our tool, Simon. Using DIA, I have automated and quantified Parnas’s qualitative changeability analysis of the KWIC system [ASE 2005]. Abilities such as these have the potential to help with rigorous analysis of important and difficult decisions in software design. Associating costs with change, for example, would allow one to analyze the long-term cost-of-change implications of a choice among possible compensating actions today.

1.4 Dissertation Research Summary

My dissertation research takes the first step towards my goal. From an ACN model, a designer can extract the dependence structure of the system and analyze its changeability early in the development process automatically and quantitatively. Representing software designs using ACNs is idealized and not adequately expressive. However, its simplicity and generality make it a reasonable starting point.

2 The Structure and Value of Modularity in Software Design

The concept of information hiding modularity is a cornerstone of modern software design thought, but its definition remains informal. We need better models of information hiding for both their explanatory power
and prescriptive utility. One of the efforts that led to my dissertation work was a project, joint with my colleagues, to evaluate the potential of a new theory—developed by Baldwin and Clark to account for the influence of modularity on the evolution of the computer industry—to inform software design [FSE 2001]. The theory uses an existing engineering model called design structure matrices (DSMs) to model designs and an economic analysis model called net option value (NOV) to evaluate them. To test the potential utility of the theory for software, we mapped Parnas’s KWIC designs into DSMs. I then created a novel method to quantitatively analyze changeability of a design, which enabled rigorous economic analysis for software designs. Our quantified analysis confirmed Parnas’s conclusions with numbers. Our models showed that the information-hiding criterion does have a tangible and measurable form in a DSM.

My current work enables automatic DSM derivation from abstract design models. DSM modeling is widely studied and used for design task structuring and optimization in other design areas, such as vehicle design and civil engineering. The derivation of DSMs from ACNs has the potential to link precise, abstract logical design representations with tools and methods already developed around DSMs. My approach also enables automatic NOV economic analysis for abstract design models. In summary, my framework integrates intellectual contributions from three fields: software design, financial economics (NOV) and engineering systems design (DSM).

3 Information Hiding Modularity in Aspect-Oriented Programming

Aspect-oriented programming (AOP) languages such as AspectJ offer new mechanisms and possibilities for decomposing systems into modules and composing modules into systems. My colleagues and I explored the modularization ability of AspectJ [AOSD 2002]. My experiment exhibited a significant bound: component integration is not adequately modularizable in AspectJ. We further explored how to best modularize programs in which aspect-oriented composition mechanisms are used [FSE 2005, IEEE 2006]. I contributed to the efforts of applying the information-hiding criterion to aspect-oriented programming. We used DSMs to visually evaluate and compare the widely cited oblivious design approach with our own. I compared different designs quantitatively using the NOV analysis. Both DSM and NOV analyses showed significant weaknesses in the former and benefits in the latter.

4 Future Plans

Before coming to the United States and the University of Virginia, I worked as a senior software engineer for one of top computer research institutions in China. I participated in numerous projects involving dozens of programmers. I saw first-hand how large projects fail. Many projects became disasters due to the lack of careful design and necessary analyses in face of evolution. I seek to explore design modeling and analysis techniques for the power of description, prediction, and prescription, serving various development stages.

Between Design and Specification. A good specification itself does not imply a good design. A specification can be seen as the environment of a design. There are internal dependences among these environment conditions. A good software design description should reflect, respect, and modularize these dependencies. I have started working in this direction with software specification researchers.

Between Design and Value. My framework has the potential to link formal design representations with economic analysis. In addition to the NOV computation, extending the design impact analysis to support cost modeling would allow one to find the least expensive way to accommodate a given sequence of changes in a design. My research is motivated by a question from an industry practitioner: “Given the necessity to keep our feature delivery velocity, is it worthwhile investing in refactoring, as my engineers suggested?” My framework proposes a solution to such a problem, which has the following key elements: (1) developing ACN models of the current and proposed designs at a suitably high level of abstraction; (2) formulating an expected evolutionary scenario as a sequence of changes, or perhaps as a stochastic process generating...
change requests; (3) measuring the cost of change in both cases; (4) accounting for the switching cost to get from the current to the proposed new design. I plan to further evaluate and develop this idea.

**Between Design and Code.** Under the pressure of project deadlines, a project usually sacrifices many necessary design considerations and plunges to implementation prematurely. The problem is not that the design stage is ignored. It is that current design modeling and analysis techniques do not support fast and automated design evolution modeling and analysis. Software evolution should start with comprehensive consideration of the costs and benefits based on current and proposed design representations. These analyses should enable designers to select an optimal way to evolve the project, quickly and automatically. On the other hand, legacy code presents difficulties in many companies. Recovering designs from source code is important to save previous investments. I plan to explore approaches to extracting logical design models from source code, taking it as a subset of the full design, combining it with high level design models, and forming a full picture.

I believe that a scientific and analyzable design model is a key to handling the complexity of software design, modularization, and evolution. I am looking forward to continuing my research in this direction.