

# Position Statement for NSF Science of Design Workshop

## Kathi Fisler

Software design requires interplay between human designers and the computer-based tools that support the design process. A sound science of software design must indicate how to build tools that enhance people's design abilities. Tools routinely automate tasks that are either too time-consuming or infeasible for people to perform manually. Design verification tools are one such example. Unfortunately, many verification tools fail to account for how a designer views the work being automated. Such tools effectively force a designer to change how she thinks about the design, which somewhat contradicts the goal of the tool enhancing the designer's process.

Although developing tools that support design reasoning appears a primarily cognitive problem, it also has a critical logical and foundational dimension. Consider the logics, models, and algorithms that underlie modern verification tools. The verification community has tended to design these around abstract models of the information to be analyzed, rather than concrete models of how designers use that information. The effort the designer expends to work with these general models distracts them from thinking about the design. Temporal logics provide a good example: while they capture the abstract information common to many functional requirements, designers often have substantial difficulty writing requirements in them. The increasing levels of research on user-friendly requirements languages underscores the importance of this problem in practice. A science of design should help tool developers avoid this problem.

I believe the software design community needs more research on creating formalisms that capture a designer's concrete perspective on tasks. Although our current abstract models allow generality across problems, the lack of domain-specific models hinders usability and often inhibits efficiency. The latter observation speaks particularly to the standard concerns about scalability in formal verification.

My own research provides two instances of computational benefits arising from bringing a designer's perspective into formal models and analyses.

**Diagrams as Formal Design Logics:** Hardware designers tend to use multiple notations, many of them diagrammatic (such as timing diagrams, circuit schematics, and state machines). Each of these notations explicates a certain kind of information; in combination they provide several focused perspectives on a design. Modern verification tools rarely support reasoning with information from multiple notations. GUIs support diagrams, but viewing diagrams merely as interfaces obscures the deep connections between the information explicit in the diagrams and verification algorithms. In other words, our GUI-based view of diagrams hinders building tools that reflect how designers work in practice.

My research develops verification logics and algorithms that operate at the level of the diagrams themselves. The resulting logics contain inference rules that mimic the reasoning steps that the designers take when using the diagrams [3]. The corresponding algorithms exploit the information made explicit in the diagrams [4]. These algorithms can be more efficient than those for the diagrams' nearest textual counterparts, even when those counterparts target the same kind of information (such as timing diagrams versus temporal logic). This work illustrates the computational consequences of our tendency to treat the designer's perspective purely as an interface issue.

**Feature-Oriented Verification:** Scalability is one of the crucial problems facing automated verification tools. In practice, verification engineers decompose designs into smaller components and attempt to derive properties of a design from properties proven of the smaller components. The smaller components generally reflect the modular structure of the implementation (such as the objects that comprise the implementation). To use this technique, a verification engineer must also decompose the *properties* around these

modules. This can be very difficult, because properties are inherently about function rather than implementation. Put differently, properties (requirements) are determined by users, and usually constrain features of a design (also driven by users). Thus, traditional compositional verification takes a design and properties both conceived in terms of features (the end-user's perspective) and recasts the problem in terms of the implementation.

Feature-oriented design modularizes a design around its features, rather than its implementation. Features cross-cut the objects and entities that comprise a system's implementation. Research efforts into programming with features [2, 7], separation of concerns [6], and aspects [1] are all exploring the effects developing software with cross-cutting information. Little work, however, has studied how cross-cutting impacts verification. Shriram Krishnamurthi<sup>1</sup> (Brown University) and I have developed verification algorithms for certain models of cross-cutting modules. In particular, we are developing verification methodologies that check properties against individual features (modules) and automatically generate interface constraints on features that are sufficient to preserve their properties at composition time [5]. This technique retains the alignment between properties and designs, avoids property decomposition, and supports an increasingly common product-line style of development (in which features are combined to form multiple products). Supporting the design-oriented view of systems as compositions of features is therefore leading to improvements in verification methodologies.

These two projects illustrate ways in which formal models could account for designers' and users' perspectives. In both cases, the new models offer *computational* benefits over earlier models, despite their more cognitive motivations. Our community needs additional such research in order to develop automated design tools that support designers' perspectives. Achieving this goal remains a key challenge in developing a science of software design.

## References

- [1] Aspect oriented programming (article series). *Communications of the ACM*, 44(10), October 2001.
- [2] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Transactions on Software Engineering and Methodology*, April 2002.
- [3] Kathi Fisler. Timing diagrams: Formalization and algorithmic verification. *Journal of Logic, Language, and Information*, 8:323–361, 1999.
- [4] Kathi Fisler. Diagrams and computational efficacy. In Dave Barker-Plummer, David I. Beaver, Johan van Benthem, and Patrick Scotto di Luzio, editors, *Words, Proofs, and Diagrams*, pages 27–46. CSLI Publications, 2002.
- [5] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying cross-cutting features as open systems. In *Symposium on the Foundations of Software Engineering*, pages 89–98. ACM Press, 2002.
- [6] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717), IBM, April 1999.
- [7] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science. Springer-Verlag, 1997.

---

<sup>1</sup>Unable to submit/attend due to teaching constraints.