DH Sorensen
Undone Science Abstract

Abstraction, Specification, and Proof

There are three concurrent, rapidly-advancing fields of computer science which share a vocabulary but do not agree on the meaning of terms. The disciplines are verified computation via zero-knowledge proofs [1], formal methods based in interactive theorem provers (ITPs) [13], and the formalization of mathematics [2, 12], and the terms are *abstraction*, *specification* (or *proposition*), and *proof*. A lack of unity in meaning between these disciplines is a matter of persistent vulnerability in software due to incorrect specification. We illustrate by first noting the differences, and then discussing the corresponding vulnerabilities.

First, consider the term *abstraction*, which has two meanings [3]. From the computer scientist's standpoint, abstraction is an activity of information *hiding*. Abstractions are made to simplify complex computational machinery so that one can more readily reason about computation, even informally, and write correct specifications. This results in layers in a software development stack, abstract data types, and high-level diagrams which describe system architecture. In contrast, mathematical abstraction is a means of information *neglect*. When studying a mathematical object, such as a group, a mathematician seeks to discover the core, minimal properties needed for desired behavior or results to hold. In this sense, the process of abstraction is a process of forgetting irrelevant details, differing in substance from computer scientific abstraction [3, 5].

Both meanings of abstraction are present among the three mentioned disciplines' notions of *specification* [5]. Mathematical abstraction shows itself most strongly in formalized mathematics, driving how mathematical propositions are articulated (or specified), and most weakly in the specification of verified software. We see mathematical abstraction in software specification particularly when that software uses mathematical concepts—for example, in the specification of a system's topology or operations on elliptic curves in zero-knowledge proofs [8]. On the other hand, computer-scientific abstraction is ubiquitous in the engineering aspects of software, thus it presents itself most strongly in software which can be formally verified or whose execution can be verified to have occured. Even so, one still finds it in the formalization of mathematics, since formalization is ultimately an engineering effort and features the likes of abstract data types [7].

Both meanings of abstraction are also present among the three mentioned disciplines' notions of *proof*, and it is through specification that these notions of proof relate to each other. Our contrasting notions of proof are *formal proof*, which is a mathematical proof derived within a formal system, and *zero-knowledge proof*, which is a cryptographic proof that a program has executed [4]. The meaning of a zero-knowledge proof is found precisely in the executed program's specification, thus in this context one knows the meaning of a zero-knowledge proof to the degree that one formally understands the behavior of said program.

Because formal methods applies formal proof to show software correct with regards to a specification, and the act of specification itself features both notions of abstraction, we might expect to see both notions of abstraction in formal specifications [5]. We would be disappointed. Anyone experienced in ITP-based formal methods will know that the actual formation of a formal specification is a translation from informal to formal specification, and thus features very little by way of mathematical abstraction [6, 10, 11]. For example, the formal specification of a system's topology will specify process APIs or communication protocols, but will almost certainly not formalize any graph theoretic description of the system.

The undone science for which we advocate is a study of how we can unify these notions of abstraction and specification in order to rigorously evaluate the correctness of software specifications. Software will have as many errors due to incorrect specification as it will to incorrect code, and while a great deal of work has been done in formal methods to show software correct with regards to a specification, relatively little has been done to evaluate the correctness of a specification itself [5, 9, 14]. The example of formally specifying a system's topology by specifying APIs or communication protocols shows us a problem that needs addressing: in practice, mathematical aspects of software design are not specified with mathematics, but instead undergo an informal translation into a prose specification. We argue that this informal translation is an immense source of avoidable errors. With ever-growing libraries of formalized mathematics, there is no reason why formal specifications need to undergo such informal translations, and improving the rigor with which we specify software will ultimately help us reason about the correctness of software specifications.

References

[1] Nada Amin, John Burnham, François Garillot, Rosario Gennaro, Daniel Rogozin, Cameron Wong, et al. Lurk: Lambda, the ultimate recursive knowledge. *Cryptology ePrint Archive*, 2023.

[2] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The hott library: a formalization of homotopy type theory in coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 164–172, 2017.

[3] Timothy Colburn and Gary Shute. Abstraction in computer science. *Minds and Machines*, 17:169–184, 2007.

[4] James H Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, 1988.

[5] Luciano Floridi. *The Blackwell guide to the philosophy of computing and information*. John Wiley & Sons, 2008.

[6] M-C Gaudel. Formal specification techniques. In *Proceedings of 16th International Conference on Software Engineering*, pages 223–227. IEEE, 1994.

[7] Georges Gonthier. Engineering mathematics: the odd order theorem proof. *ACM SIGPLAN Notices*, 48(1):1–2, 2013.

[8] C. A. R. Hoare. Mathematics of programming. *Byte*, pages 135–154, August 1986.

[9] Ralf Kneuper. Limits of formal methods. *Formal Aspects of Computing*, 9:379–394, 1997.

[10] Axel van Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159, 2000.

[11] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying formal specification in industry. *IEEE software*, 13(3):48–56, 1996.

[12] mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, pages 367–381, 2020.

[13] M Saqib Nawaz, Moin Malik, Yi Li, Meng Sun, and M Lali. A survey on theorem provers in formal methods. *arXiv preprint arXiv:1912.03028*, 2019.

[14] Bruce W Weide, William F Ogden, and Stuart H Zweben. Reusable software components. In *Advances in computers*, volume 33, pages 1–65. Elsevier, 1991.