The Human in Formal Methods

Shriram Krishnamurthi(✉) and Tim Nelson

Brown University, Providence, RI, USA
{sk,tn}@cs.brown.edu

Abstract. Formal methods are invaluable for reasoning about complex systems. As these techniques and tools have improved in expressiveness and scale, their adoption has grown rapidly. Sustaining this growth, however, requires attention to not only the technical but also the human side. In this paper (and accompanying talk), we discuss some of the challenges and opportunities for human factors in formal methods.

Keywords: Human factors · User Interfaces · Education · Formal methods

1 Humans and Formal Methods

Formal methods are experiencing a long-overdue surge in popularity. This ranges from an explosion in powerful traditional tools, like proof assistants and model checkers, to embeddings of formal methods in program analysis, to a growing recognition of the value to writing formal properties in other settings (like software testing). Whereas traditionally, corporate use was primarily in hardware (e.g., Seger [26]), now major software companies like Amazon [1,7,21], Facebook [6], and Microsoft [3,12] are growing their use of formal methods.

What does it take to support this growth? Researchers will, naturally, continue to work on formal techniques. We believe, however, that not enough attention has been paid to the *humans in the loop*. In this paper and accompanying talk, we discuss some of the challenges and opportunities in this area.

To set a context for what follows, our own work has focused largely on automated methods, specifically *model finding* [18,34], as typified by tools like Alloy [15] and SAT/SMT solvers. This is not to decry the value of other techniques, including deductive methods, which we have worked with in some of our research. However, we find that model-finding tools offer a useful sweet spot:

- Because of their automation, they provide a helpful separation between specification and proof, enabling the user to focus on the former without having to dwell very much on the latter. This separation of concerns is invaluable in training contexts, since it enables us to focus on one skill at a time.
- Because model-finders can be used without properties, they enable *exploration* in addition to verification and proof. Furthermore, this can start with small amounts of partial specification. This idea, which is one aspect of *lightweight formal methods* [16], is a powerful enabler for quickly seeing the value that formal methods can provide.

M. H. ter Beek et al. (Eds.): FM 2019, LNCS 11800, pp. 3–10, 2019.
https://doi.org/10.1007/978-3-030-30942-8_1

– The manifestation of these methods in tools like Alloy proves particularly convenient. An Alloy user can write a small part of a specification and click "Run" (an action already familiar from programming environments), and immediately get at least somewhat useful feedback from the system.

Due to these factors, in our experience, we have found these methods more accessible than others to a broad range of students. Since, in particular, our emphasis is not just on cultivating the small group of "hard core" students but to bring the "other 90%" into the fold, tools that immediately appeal to them—and hold their attention, while they are choosing between courses in formal methods and in other exciting areas such as machine learning—are important.

In the rest of this paper, we focus on two human-facing concerns: the human-factors qualities of model finding tools (Sect. 2), and education (Sect. 3). We believe both are vital: the latter to growing the *number* of people comfortable with formal methods, and the former to their *effectiveness*.

2 User Experience

We believe that the user experience of formal-methods tools has largely been understudied, although there have been promising past venues such as the Workshops on User Interfaces for Theorem Provers (e.g., [2]) and Human-Oriented Formal Methods (e.g., [19]). The majority of this work focuses on interactive tools such as proof assistants, which is to be expected. For instance, in deductive methods, the experience of stating and executing deduction steps is critical. (For early student-facing work, see the efforts of Barker-Plummer, Barwise, and Etchemendy [4]).

However, other formal tools could also benefit from user-focused research. For instance, model finders are often integrated into higher-level tools (with their model output presented in a domain-specific way). Thus, questions of quality and comprehensibility by lay users are key.

Our own work [8] has found that a model finder's choice of output and its presentation can make a major difference in user experience. Experiments with students found that output *minimality*, while intutively appealing, is not necessarily helpful for comprehending systems. Moreover, experiments with users on Amazon's Mechanical Turk crowdsourcing platform seem to suggest that providing a small amount of additional information alongside output can be helpful for comprehension.

3 Education

An equally important—and critically human-centric—problem is thinking about education. Numerous authors have *books* that present different educational viewpoints but, to our knowledge, most of these have not been subjected to any rigorous evaluation of effectiveness. Nevertheless, beyond books and curricula,

we believe much more attention should be paid to design methods and student-centric tools. There is a large body of literature on these topics in programming education, but its counterparts are often missing in formal methods education.

We are focusing primarily on the task of *writing specifications*, because:

- It is a near-universal requirement shared between different formal methods—indeed, it is perhaps a defining characteristic of the field.
- Specifications are sufficiently different from programs that we cannot blindly reuse existing knowledge about programming education, though of course there are many problems in common and we should try to port ideas. If anything, we conjecture that the need for formal methods to consider *all* possible behaviors, thanks to attributes like non-determinism, might make it *harder* than programming.
- Specifications are useful even outside traditional formal methods settings, such as in property-based testing, monitoring, etc. Hence, they increasingly affect a growing number of programmers, even ones who don't think of themselves as using traditional formal methods.

We will in turn discuss design methods (Sect. 3.1) and tools (Sect. 3.2).

3.1 A Design Recipe for Writing Specifications

One of the challenges every author faces is the "blank page syndrome" [9]: given a problem statement, they must fill a blank page (or editor) with magical incantations that match the given statement. For many students, this can be a daunting and even overwhelming experience; ones for whom it is not are sometimes merely overconfident in their abilities.

However, in other design disciplines—from electrical engineering to building architecture—designers produce not just one final artifact but a series of intermediate artifacts, using a range of representations with distinct viewpoints that hide some aspects and make others salient. What might that look like in our discipline?

One answer is provided by *How to Design Programs* [9], which breaks down the programming process into a series of steps called the Design Recipe. These steps incrementally build towards a solution, alternating abstract and concrete steps that build on previous ones. For programming, these steps are:

1. Data definitions: translating what is given in the problem statement into abstract descriptions for the computer system.
2. Data examples: constructing examples of each data definition to ensure the student understands it, has created a well-formed definition, and can cover the cases the problem demands.
3. Function outline: translating the function expected in the problem into an abstract computational representation, including type signatures, purpose statements, and a function header.

4. Function examples: constructing input-output examples of the function's use, using the data examples and the function outline components. These ensure the student actually understands the problem before they start working on it. These are usually written using the *syntax* of test cases, so they can eventually be run against the final function, but they are conceptually different: they represent exploration and understanding of the *problem*.
5. Function template: Using the data definition and function outline to create a skeleton of the body based purely on the structure of the data.
6. Function definition: Filling in the template to match the specific function definition, using the examples as a guide.
7. Testing: Constructing tests based on the chosen implementation strategy, checking for implementation-specific invariants. The goal of tests, in contrast to function examples, is to falsify the purported implementation.

There is significant cognitive theory backing the use of this recipe. The process corresponds to Bruner's notion of *scaffolding* [31], while the steps reflect Vygotsky's theory of *zones of proximal development* [29]. The progression from data through examples to code and tests provides a form of *concreteness fading* [13]. Completed sequences form *worked examples* [28] that students can apply to new problems. The templates are a form of *program schema* [22,27] that students can recall and reuse in constructing solutions to new problems.

How can we translate this from writing programs to writing specifications? We believe many of the steps carry over directly (and serve the same purpose), while others need some adaptation, depending on what students are authoring (the process for specifications would look different than that for models given to a model-checker, etc.). For instance, the "function examples" stage translates well to students creating concrete instances of behavior that they believe should or should not satisfy the eventual specification.

We will not go here into the details of how to adapt this process to different settings, especially authoring specifications. However, we believe the basic ideas are fairly universal: of proceeding in a step-wise way with new artifacts building on old artifacts; of proceeding from the concrete to the abstract; of writing illustrative, concrete examples of preceding abstract steps to test well-formedness and understanding; and so on.

3.2 Tools

Researchers and developers have invested significant effort into formal methods tools, many of which are then brought into the classroom. On the one hand, industrial-strength tools tend to be robust and performant, and are endowed with authenticity, which can make a difference for some students. On the other hand, they may expose too much power: they accept full and complex languages that contain features that may confuse students, they produce errors and other feedback with terminology that students may not understand, and so on. In light of this, projects have advocated the use of *language levels* [5,10,14], arguing that

students would benefit from a graduated introduction through a sequence of sub-languages (and corresponding tool interfaces), each sub-language presenting an epistemic closure that corresponds to a student's learning at that point.

Beyond this, we argue that educational settings have one key advantage that conventional industrial use does not: the presence of *ground truth*, i.e., someone already knows the answer! In industry, users rarely build a whole new specification that precisely matches one that already exists. In education, however, that is exactly what students do almost all the time. Therefore, we can ask:

How does the presence of a ground truth affect formal tools in education?

We argue that "knowing the answer" especially helps in light of the Design Recipe discussed above, because we can build tools to help with each step. We discuss a concrete manifestation of this below. These should be thought of as training wheels to help beginners become comfortable with formal methods; naturally, we need to study how to wean students so that they can engage in the more authentic experience of writing specifications and models un-aided.

Understanding Before Authoring. A growing body of literature in programming education [17,25,30] shows that students frequently start to write programs before they have understood the problem. As a result they "solve" the wrong problem entirely. Not only is this frustrating, it also leads to learning loss: the stated problem presumably had certain learning goals, which the student may not have met as a result of their misdirection.

Recent work [24,32] has begun to address this issue by devising techniques to make sure students can check their understanding of the problem before they embark on a solution. These critically rely on having intermediate artifacts authored by the student in the process of authoring, precisely matching the intermediate steps proposed by the Design Recipe. In particular, function examples are a valuable way for them to demonstrate their understanding; because they are written in executable form, they can be run against implementations.

We especially draw on the perspective of Politz et al. [23] and Wrenn et al. [33], which think of tests (and examples) as *classifiers*. That is, the quality of a suite of tests or examples can be judged by how well they classify a purported implementation as correct or faulty. If we want a quantitative result, we can compute precision and recall scores to characterize these classifiers. Thus, students can rapidly obtain concrete feedback about how well they are doing in terms of understanding the problem, and our evidence in the context of programming [32] suggests that they take great advantage of this. Initial explorations for specification authoring suggests that this phenomenon carries over.

More Artifacts. More broadly, there are several artifacts that can be produced on both sides for specification-authoring assignments, including:

- Student's concrete examples
- Student's properties
- Student's completed specification

– Instructor's concrete examples
– Instructor's properties
– Instructor's completed specification

the latter three of which are ground truth components. Furthermore, instructional staff can be pressed to produce multiple kinds of each of these, such as correct and faulty specifications to enable classification.

Given this rich set of artifacts, it is instructive to consider all their (at least) pairwise combinations. For example, consider the point where the student believes they have completed their specification. This can now be compared for *semantic difference* [11,20] against the instructor's specification, with the differences presented as concrete examples that the student has to determine how to incorporate to adjust their specification. There are several interesting questions of *mechanism design*, i.e., how to structure rewards and penalties for students using these modes.

4 Conclusion

In sum, we believe there are large human-facing aspects of formal methods that have not yet been explored, and that exploring them is vital for the field to thrive. With enough emphasis, we believe formal methods can be democratized and made accessible to large numbers of users—not only scientists and trained operators, but even the general public, from children to retirees. Even the most non-technical user has to make consequential decisions every time they set a configuration option on a system, and would hence benefit from the specification and state-exploration powers that characterize our field. These problems are intellectually exciting and challenging, and serious progress requires wedding technical results to cognitive and social ones.

Acknowledgements. This work was partially supported by the U.S. National Science Foundation. We are grateful for numerous valuable conversations with Daniel J. Dougherty, Natasha Danas, Jack Wrenn, Kathi Fisler, Daniel Jackson, and Emina Torlak.

References

1. Amazon Web Services: Provable security. https://aws.amazon.com/security/provable-security/. Accessed 5 July 2019
2. Autexier, S., Benzmüller, C. (eds.): User Interfaces for Theorem Provers, Proceedings of UITP 2006, Electronic Notes in Theoretical Computer Science, vol. 174. Elsevier (2007)
3. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: technology transfer of formal methods inside microsoft. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24756-2_1

4. Barker-Plummer, D., Barwise, J., Etchemendy, J.: Language, Proof, and Logic, 2nd edn. Center for the Study of Language and Information/SRI, Stanford (2011)
5. du Boulay, B., O'Shea, T., Monk, J.: The black box inside the glass box. Int. J. Hum.-Comput. Stud. **51**(2), 265–277 (1999)
6. Calcagno, C., et al.: Moving fast with software verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 3–11. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_1
7. Cook, B.: Formal reasoning about the security of amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 38–47. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_3
8. Danas, N., Nelson, T., Harrison, L., Krishnamurthi, S., Dougherty, D.J.: User studies of principled model finder output. In: Cimatti, A., Sirjani, M. (eds.) SEFM 2017. LNCS, vol. 10469, pp. 168–184. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66197-1_11
9. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs, 2nd edn. MIT Press, Cambridge (2018). https://www.htdp.org/
10. Findler, R.B., et al.: DrScheme: a programming environment for Scheme. J. Funct. Prog. **12**(2), 159–182 (2002)
11. Fisler, K., Krishnamurthi, S., Meyerovich, L., Tschantz, M.: Verification and change impact analysis of access-control policies. In: International Conference on Software Engineering, pp. 196–205 (2005)
12. Fogel, A., et al.: A general approach to network configuration analysis. In: Networked Systems Design and Implementation (2015)
13. Fyfe, E.R., McNeil, N.M., Son, J.Y., Goldstone, R.L.: Concreteness fading in mathematics and science instruction: a systematic review. Educ. Psychol. Rev. **26**(1), 9–25 (2014)
14. Holt, R.C., Wortman, D.B.: A sequence of structured subsets of PL/I. SIGCSE Bull. **6**(1), 129–132 (1974)
15. Jackson, D.: Software Abstractions: Logic, Language, and Analysis, 2nd edn. MIT Press, Cambridge (2012)
16. Jackson, D., Wing, J.: Lightweight formal methods. IEEE Comput. (1996)
17. Loksa, D., Ko, A.J.: The role of self-regulation in programming problem solving process and success. In: SIGCSE International Computing Education Research Conference (2016)
18. McCune, W.: Mace4 reference manual and guide. CoRR (2003). https://arxiv.org/abs/cs.SC/0310055
19. Milazzo, P., Varró, D., Wimmer, M. (eds.): STAF 2016. LNCS, vol. 9946. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4
20. Nelson, T., Ferguson, A.D., Krishnamurthi, S.: Static differential program analysis for software-defined networks. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 395–413. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_25
21. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
22. Pirolli, P.L., Anderson, J.R.: The role of learning from examples in the acquisition of recursive programming skills. Canadian Journal of Psychology/Revue canadienne de psychologie **39**(2), 240–272 (1985)
23. Politz, J.G., Krishnamurthi, S., Fisler, K.: In-flow peer-review of tests in test-first programming. In: Conference on International Computing Education Research (2014)

24. Prather, J., et al.: First things first: providing metacognitive scaffolding for interpreting problem prompts. In: ACM Technical Symposium on Computer Science Education (2019)
25. Prather, J., Pettit, R., McMurry, K., Peters, A., Homer, J., Cohen, M.: Metacognitive difficulties faced by novice programmers in automated assessment tools. In: SIGCSE International Computing Education Research Conference (2018)
26. Seger, C.H.: Combining functional programming and hardware verification (abstract of invited talk). In: International Conference on Functional Programming (ICFP) (2000)
27. Spohrer, J.C., Soloway, E.: Simulating Student Programmers. In: IJCAI 1989, pp. 543–549. Morgan Kaufmann Publishers Inc. (1989)
28. Sweller, J.: The worked example effect and human cognition. Learn. Instr. **16**(2), 165–169 (2006)
29. Vygotsky, L.S.: Mind in Society: The Development of Higher Psychological Processes. Harvard University Press, Cambridge (1978)
30. Whalley, J., Kasto, N.: A qualitative think-aloud study of novice programmers' code writing strategies. In: Conference on Innovation and Technology in Computer Science Education (2014)
31. Wood, D., Bruner, J.S., Ross, G.: The role of tutoring in problem solving. J. Child Psychol. Psychiatry **17**, 89–100 (1976)
32. Wrenn, J., Krishnamurthi, S.: Executable examples for programming problem comprehension. In: SIGCSE International Computing Education Research Conference (2019)
33. Wrenn, J., Krishnamurthi, S., Fisler, K.: Who tests the testers? In: SIGCSE International Computing Education Research Conference, pp. 51–59 (2018)
34. Zhang, J., Zhang, H.: SEM: a system for enumerating models. In: International Joint Conference on Artificial Intelligence (1995)