Some Usability Hypotheses for Verification

David J. Pearce
School of Engineering and Computer Science, Victoria University of Wellington, New Zealand
djp@ecs.vuw.ac.nz

Abstract

The idea of specifying and verifying software to eliminate errors has been studied extensively over the last three decades or more. Recent advances in automated theorem proving have given rise to a range of new verification tools being developed. Despite this, very little is known about the effect of using such tools on software development. In this paper, we present several verification-related usability hypotheses which we believe warrant further investigation. These hypotheses are based on observations from the literature, as well as our own experiences in developing and using the Whiley verification system.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Programming by contract; H.1.2 [*User/-Machine Systems*]: Human factors

General Terms Languages, Verification, Human Factors

Keywords Software Verification, Usability

1. Introduction

The idea of specifying and verifying software goes back a long way to the likes of Hoare [1], Dijkstra [2], Gries [3] and others [4]. Whilst many theoretical works have been developed, much less has been achieved in terms of practical tooling. This is perhaps most evident by the number of books written on the subject, of which almost none are based around an actual tool (this is perhaps analogous to books on programming which are based only on some ad-hoc pseudocode) [3, 5–9].

Disappointed by developments on the practical side, Hoare created the Verifying Compiler Grand Challenge as an attempt to spur new efforts [10]. According to Hoare's vision, a verifying compiler "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles*" [10]. The earliest systems that could be reasonably considered as verifying compilers include that of King [4], Deutsch [11], the Gypsy Verification Environment [12] and the Stanford Pascal Verifier [13]. Following on from these was the Extended Static Checker for Modula-3 [14]. Later, this became the Extended Static Checker for Java (ESC/Java) — a widely acclaimed and influential work in this area [15]. Building on this success was the Java Modeling Language (and its associated tooling) which provided a standard notation for specifying functions in Java [16]. Since Hoare issued his challenge, a variety of new tools have blossomed in this space, including Spec# [17], Dafny [18], Why3 [19], VeriFast [20] and Whiley [21, 22].

At this juncture, we now return to consider the goal of this paper. Despite the large amount of theoretical work on software verification and the availability of some usable tooling, very little is really known about the effects of using tools such as verifying compilers on software development. The general assumption is that such systems will have an overwhelmingly positive impact on software development. For example, Bowen and Hinchey state [23]:

> *"It is clear to "Formal Methodists" like ourselves, for some of whom formal methods can be something of a "religion", that introducing greater rigor into software development will improve the software development process, and result in software that is better structured, more maintainable, and with fewer errors"*

Likewise, Meyer says something similar when talking about the future of software [24]:

> *"It is clear to all the best minds in the field that a more mathematical approach is needed for software to progress much."*

Whilst the present author doesn't necessarily disagree with these views, there is certainly a lack of experimental evidence to support them. Part of the problem here is simply that the use of such tools is far from widespread and, indeed, very few people actually have experience with them. Therefore, in this paper, we present a number of hypotheses regarding the specification and verification of software. We distinguish the act of *specifying* a program (e.g. annotating it with pre-/post-conditions, etc) from *verifying* it using some kind of tool (e.g. a verifying compiler). These hypotheses stem from commonly found views in the literature, as well as our own experiences in developing and using the

PLATEAU'15, October 26, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3907-0/15/10...
http://dx.doi.org/10.1145/2846680.2846691

Whiley programming language and its accompanying verifying compiler. We have also had the opportunity to observe students using Whiley to verify simple programs as part of a second-year paper on formal methods.

2. Hypotheses

We now present our verification-related hypotheses in two parts. First, we consider those related primarily to the act of specifying software and then we consider those related to the act of verifying software.

2.1 Specification

Our first hypothesis related to specifying software is seemingly straightforward:

HYPOTHESIS 1. *Specifications are no different from other aspects of programming.*

By this we mean that writing specifications will itself exhibit all the usual problems and issues found when writing software. For example, specifications can themselves contain bugs. Backhouse highlights this in his introduction [6]:

> *"Of course, the science guarantees correctness only if it is used correctly, and people will continue to make mistakes. So, testing is still wise"*

Likewise, specifications can be written in ways which are or are not more readable and will suffer from the general issues of software maintenance. They will not always reflect what the user wanted and will need to be developed alongside the user to ensure the right things are specified. Specifications will also need to be developed incrementally with the rest of the program, as Gries acknowledged [3]:

> *"A program and its proof should be developed hand-in-hand, with the proof usually leading the way."*

That is, as the program's design inevitably changes, its specifications will need to be updated accordingly. We note this appears to contradict the view taken by proponents of the "Correct by Construction" approach to development. For example, Lamport states [9]:

> *"it's a good idea to specify a program before implementing it"*

Similarly, Kourie and Watson state the essence of their book is to derive code from specifications, more specifically [8]:

> *"Once a problem has been specified, a number of refinement laws can be deployed to refined incrementally the specification"*

However, we do not believe there really is a contradiction here. For example, Lamport does not necessarily advocate developing complete specifications beforehand and, in later work, says *"formal specification is just one end of a spectrum. An architect would not draw the same kind of blueprint for a toolshed as for a bridge"* [25]. Likewise, Kourie and Watson explicitly acknowledge they do not expect *"all and sundry"* to specify programs before implementing them; rather they are hoping to foster more rigorous thought processes during development.

HYPOTHESIS 2. *Specifications are rarely complete.*

As highlighted above, one view found in the literature is that aim is always to *completely* specify software. In most cases, completeness is taken to mean "functional completeness" and ignores issues such as real-time constraints or other resource constraints (e.g. memory). Nevertheless, many acknowledge the additional burden of complete specification may not be cost effective [26–28]. On this point Murphy-Hill and Grossman say [28]:

> *"First, formal specifications need not encompass all requirements. We can prove browser security without formalizing everything a web browser must do, which is essential since even specifying how to render HTML is surely intractable."*

Likewise, Bowen and Hinchey in their widely acclaimed paper say [26]:

> *"Thou shalt formalize but not over formalize ... Applying formal methods to all aspects of a system would be both unnecessary and costly."*

We speculate the effort required to specify a program increases considerably with the level of completeness required. Polikarpova *et al.* experimentally assessed the benefit of stronger versus weaker specifications [27]. As expected, they found stronger specifications uncovered more software faults. However, they also claimed anecdotally that the *"effort required to write the strong specifications was moderate"*, based on their experiences in developing strong specifications for their experiment. However, our personal experience contradicts this view as we have found simple specifications relatively easy to write, but more complete specifications considerably harder.

Our view is that we expect to see widely varying levels of specification in practice (i.e. when such tools become more common-place). This will be partly driven by the needs of programmers. If the goal is simply to eliminate common errors (e.g. array-out-of-bounds, division-by-zero, etc) then incomplete specifications will likely be sufficient. Another factor is that programmers themselves will sometimes be unaware that they have not fully specified a program. Since verification tools give no indication as to how complete a specification is, it will be up to the programmer to decide when to stop and varying levels of expertise will inevitably lead to varying levels of specification.

HYPOTHESIS 3. *Specification without verification provides only modest benefit.*

We believe that the act of specifying a program, regardless of whether or not it is verified, will lead to improved software quality. This is perhaps not surprising, given the general belief that even just documenting code improves quality [29]. However, like documentation, we believe that providing specifications which have not been verified in some way will provide only modest benefits.

Our personal experiences in developing a tool for checking non-null annotations supports this view [30]. In using this tool to verify parts of the Java standard library we found a large number of errors in the documentation. For example, methods in the widely used class `java.lang.String` are documented with respect to whether or not their parameters may accept **`null`**. However, at the time, we identified 83 out of 1101 public methods were mis-documented. This is particularly insidious because *most* being correctly specified gives false confidence that *all* are.

2.2 Verification

HYPOTHESIS 4. *Verification is challenging when it requires a creative step.*

Verifying a function meets it specification is, at times, quite challenging and our experience suggests one reason for this is the need for creative steps. The need for creative steps during verification is widely acknowledged [3, 8]. For example, it is well known that students find difficulty in writing loop invariants for this reason [5]. That is, when the required loop invariant does not match the loop's post-condition some creativity is needed to transform the post-condition into the correct form. Figure 1 illustrates such a function. However, in tools such as Dafny, Spec# and Whiley, there are other (less well known) kinds of creative step which can be required. For example, consider this simple function in Whiley:

```
function add([int] x,[int] y) -> ([int] z)
requires |x| == |y|
ensures |z| == |x|:
   //
   int i = 0
   while i < |x| where i >= 0:
        x[i] = x[i] + y[i]
        i = i + 1
   //
   return x
```

This function will not verify in Whiley as is because the loop invariant is not strong enough. The rules of Hoare logic tell us that the loop invariant needs to establish `|z| == |x|`. However, since `x` is returned, this corresponds to `|x| == |x|` which is seemingly nonsense. In fact, we must consider that `x` in the postcondition refers to its value *on entry to the function*, whilst `x` in the **`return`** statement refers to its value *at the end of the function*. Thus, the post-condition we must establish is really `|x'|␣==␣|x|`, where `x'` refers to its value at the **`return`** statement. Nevertheless, this still does not suggest a suitable loop invariant. In fact, a skilled operator will realise that `|x| == |y|` is sufficient since `y` is not modified by the loop and its size does match `x` on entry.

HYPOTHESIS 5. *Forwards reasoning is easier than backwards reasoning.*

Hoare logic underpins verification tools such as Dafny, Spec# and Whiley [1]. As such it is natural to connect such systems with Hoare logic when teaching. Furthermore, Hoare logic is indifferent to the direction of reasoning and can be used to reason about programs in either a forwards or backwards direction. Perhaps surprisingly, many advocate for approaches based on backwards reasoning for two reasons:

1. Reasoning in the forwards direction requires the use of Floyd's rule [31] which introduces existentials.[1]
2. One can often infer loop invariants by reasoning backwards from a function's postcondtion.

Despite this, we believe that students struggle with reasoning in a backwards direction and there is perhaps some evidence to support this from the debugging literature. For example, Katz and Anderson observed that novices tend to prefer debugging in a forwards direction, especially on code which is unfamiliar to them [32]. Fitzgerald *et al.* later replicated their studied (albeit in a slightly modified form) and drew the same conclusion [33].

HYPOTHESIS 6. *Verification is comparable with double checking.*

Verification provides a mechanism for checking our programs meet their specifications. We have no doubt that this will improve software quality. However, we also suspect that one of the main benefits from verification is simply that it forces us to look carefully at our program. If we had another mechanism which forced us to look carefully, we suspect this might give comparable benefits to verification. Such a mechanism might be to implement a given method twice — perhaps once in an imperative style, and once in a functional style. Doing this would seem to require careful thinking about what the function does and how it does it.

3. Conclusion

In this paper, we presented a number of usability hypotheses related to specification and verification. Our hope is that, as verification tools become more accessible, researchers will begin to critically examine the trade-offs they offer. And, of course, there are many more questions than we have had space to consider here. For example, understanding how

[1] We note that the need for existential quantifiers in Floyd's rule can be mitigated if students are taught an approach where existentials are immediately skolemised (i.e. so they never see them) and, for example, skolems are named along the lines of Static Single Assignment form

```
function zeroOut([int] xs) -> ([int] rs)
ensures |rs| == |xs|
ensures all { i in 0..|rs| | rs[i] == 0 }:
  //
  int i = 0
  [int] rs = xs
  //
  while i < |xs|
  where i >= 0 && i <= |xs|
  where |xs| == |rs|
  where all { j in 0..i | rs[j] == 0 }:
    rs[i] = 0
    i = i + 1
  //
  return rs
```

Figure 1. Illustrating a simple loop whose loop invariant requires a quantifier.

much overhead is imposed using a particular verification system seems important.

References

[1] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[2] E. W. Dijkstra. Guarded commands, nondeterminancy and formal derivation of programs. *CACM*, 18:453–457, 1975.

[3] D. Gries. *The science of programming*. Springer-Verlag, 1981.

[4] S. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.

[5] K. Broda, S. Eisenbach, H. Khoshnevisan, and Steven Vickers. *Reasoned Programming*. Prentice Hall, 1994.

[6] Roland Backhouse. *Program Construction*. Wiley, 2003.

[7] J.B. Almeida, M.J. Frade, J.S. Pinto, and S Melo de Sousa. *Rigorous Software Development, An Introduction to Program Verification*. Springer-Verlag, 2011.

[8] Derrick G. Kourie and Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, 2012.

[9] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[10] C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *JACM*, 50(1):63–69, 2003.

[11] L. Peter Deutsch. *An interactive program verifier*. Ph.D., 1973.

[12] D. I. Good. Mechanical proofs about computer programs. In *Mathematical logic and programming languages*, pages 55–75, 1985.

[13] D. Luckham, SM German, F. von Henke, R. Karp, P. Milne, D. Oppen, W. Polak, and W. Scherlis. Stanford Pascal Verifier user manual. Technical Report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.

[14] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 1998.

[15] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, 2002.

[16] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, March 2005.

[17] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *CACM*, 54(6):81–91, 2011.

[18] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer-Verlag, 2010.

[19] J. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *Proc. ESOP*, pages 125–128, 2013.

[20] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. pages 41–55. Springer-Verlag, 2011.

[21] D. J. Pearce and L. Groves. Whiley: a platform for research in software verification. In *Proc. SLE*, pages 238–248, 2013.

[22] D. J. Pearce and Lindsay Groves. Reflections on verifying software with Whiley. In *Proc. FTSCS*, pages 142–159, 2013.

[23] J. Bowen and M. Hinchey. Ten commandments of Formal Methods … ten years later. *IEEE Computer*, 39(1), 2006.

[24] Ted Lewis. Where is software headed? A virtual roundtable. *IEEE Computer*, 28(8):20–32, August 1995.

[25] L. Lamport. Who builds a house without drawing blueprints? *CACM*, 58(4):38–41, 2015.

[26] J. Bowen and M. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, 1995.

[27] N. Polikarpova, C. Furia, Y. Pei, Y. Wei, and B. Meyer. What good are strong specifications? In *Proc. ICSE*, pages 262–271, 2013.

[28] Emerson Murphy-Hill and Dan Grossman. How programming languages will co-evolve with software engineering: a bright decade ahead. In *Proc. FOSE*. ACM, 2014.

[29] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *SIGDOC*, pages 68–75. ACM Press, 2005.

[30] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Formalisation and implementation of an algorithm for bytecode verification of @NonNull types. *Science of Computer Programming*, pages 587–568, 2011.

[31] R. W. Floyd. Assigning meaning to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, 1967.

[32] I. R. Katz and J. R. Anderson. Debugging: An analysis of bug-location strategies. *HCI*, 3(4):351–399, 1987.

[33] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, 2008.