Writing and Using Program Specifications

Steven P. Reiss
Department of Computer Science, Brown University, Providence, RI. 02912, USA
(401)-863-7641
spr@cs.brown.edu

ABSTRACT

While there are a growing number of tools that demonstrate interesting and important uses of program specifications in software development, these tools are not widely applicable to today's software. A major problem is that it is difficult or impossible to write program specifications for most components in modern applications. In this white paper we look at the reasons for this and then propose possible research directions that can address the problem.

Categories and Subject Descriptors

D2.1 [Requirements/Specifications]: Tools; D2.4 [Software/Program Verification]: Formal methods, Validation..

General Terms

Design, Reliability, Security, Verification.

Keywords

.Software specification; Testing; Specification languages; Programming tools.

1. MOTIVATION

Program specifications have a wide variety of uses in software engineering. They have been used as a basis for writing programs [17]. They are essential for proving program correctness and model checking [6,30]. They are used for program understanding, description, and visualization. They are used for finding potential system problems [2] and have been proposed for automated debugging [27,32]. They are the basis for work in automated test case generation [22]. They are used recording requirements, specifications, and agile scenarios. Moreover, the uses of such specifications continue to grow as researchers find them more available and are able to develop new tools and techniques based on them such as code search [24], automatic connection of web services [9,23], model driven development [10,26], and automatic code generation [19].

We have attempted to use program specifications for three particular problems in the past few years. The first involves using semantics as the basis for selecting, binding, and using external components such as web services [23]. Specifications here defined what the program depended upon from the external component and let the system recheck the external components as they evolved to ensure they still met the original program's needs. The second involved semantic-based code search [24]. Here we let programmers define the semantics of what they want to search for and we attempt to find and transform publicly available code to meet their specifications. The third involved providing visualizations of a complex system in terms of the system, for example showing the threads, tasks and transactions involved in a complex server [25]. Here the problem involves building the program model the user wants to visualize and mapping this model to the actual code.

The problem we have come up against in each of these cases is that while there have been a variety of techniques used for program specification, none is suitable for describing much of what we wanted to describe. Today's specification languages and techniques are not appropriate for today's applications.

FoSER 2010, November 7-8, 2010, Santa Fe, New Mexico, USA.

2. BACKGROUND

A wide variety of different types of formal specifications have been proposed over the years. Much early work was done with executable specifications using formalisms such as VDL [29]. These were difficult to use both for formal program checking, often were as difficult to write as the actual program, and were difficult to verify. Formal, mathematical models have been more common, with frameworks such as Larch [12] and Z [31] as well as successors such as Alloy [18] being widely used. These are easier to use for proving program correctness, but turn out to be difficult for programmers to write and difficult to determine if they actually say what the writer intended. Another approach that is often taken is to use a domain-specific language. Here much of the complexity of the specifications is embedded in the language rather than in the program. Programs are relatively simple and hence much easier to verify. A classic example of this is the use of SQL for accessing databases. This approach, however, is limited to well-defined and understood domains.

Other approaches attempt to provide partial specifications of programs. Here the specifications concentrate on particular aspects of correctness or program behavior. To accommodate the movement toward software model checking, a variety of finite-state formal models have been developed [1,7,8,11,14-16]. These are often more intuitive and can be easier to write. They are typically used for proving relatively simple properties of programs, but have been used for describing complex systems [11]. The most common use of formal specifications today is in the form of contracts, generally preconditions and postconditions, either as part of a programming language as in Eiffel [21] or Spec# [3], or as an add-on as with JML [4]. These are useful for doing local validation, but do not provide comprehensive program semantics.

Because formal specifications have proven difficult to write correctly, researchers have tried other approaches. Informal specifications, for example in English or even UML are the most widely used. They are easy to write and generally can be

understood by most programmers. However, they are often ambiguous and incomplete and are rarely suitable for use in proving program properties or by other tools.

A compromise that forms the basis for agile development is the use of test cases. Test cases are inherently easier for the programmer to write and understand since they use the same language and constructs as the program itself. There are, however, incomplete in that they do not specify everything about the program and even what they specify is inherently ambiguous. This is often not a problem as they can demonstrate "sufficient correctness" to the programmer [28]. Moreover, experience has shown that a relatively small number of test cases are often enough to validate a program. This follows from the fact that while there are many possible programs that could satisfy the test cases, the one that is simplest and most likely is generally going to be the one that was intended by the programmer. Moreover, the test cases that are specified generally reflect both the common cases and the unusual cases foreseen by the programmer.

3. DIFFICULTIES

To take advantage of the growing set of uses for software specifications, one needs to be able to create specifications for both complete applications and for the various components that comprise today's systems. However, both formal specifications and formal test cases are difficult or impossible to write for most modern programs or program components. This occurs in a variety of ways.

One case arises when the user's specification is purposefully imprecise. As a simple example, consider a routine to convert an integer to a roman numeral. The user might want to specify precisely what should happen for numbers between 1 and 5,000. However, for values that are too small or too large, they would be willing to accept a variety of "nice" errors (e.g. different exceptions thrown), or reasonable default values (e.g. returning the arabic representation). However, they do not want the program to abort, go into an infinite loop, or return something meaningless or inappropriate.

A second case occurs when the result itself might be imprecise. Consider a routine that takes as input a news article and returns a probability vector of what countries the article is about. For most articles, a wide variety of answers are possible and correct under some reasonable interpretation of "about". Attempting to provide precise specifications or even test cases here is quite difficult.

Another common case for today's applications occurs when the application or component involves a visual result. For example, this occurs when the component represents a user interface widget or when the result is supposed to be a visualization of some sort. Current specification languages and most testing systems do not allow the specification of the user interface or graphical results. This is further complicated by the fact that there is considerable flexibility in what might be considered an acceptable user interface or graphical representation, and by the fact that correctness here might need to be measured in terms of user interactions and satisfaction.

A fourth case arises when the correct behavior of the component in questions depends on intricate behaviors of other components that may or may not be user-written code. For example, understanding and specifying the behavior of an edit operation in a Java swing framework depends on the implementation of the underlying document, elements, and views as well as the various document and undoable edit callbacks, the handlers have been registered with those callbacks dynamically, and the order of that registration.

Other cases arise because of complex behaviors. There are cases when the task to be done is effectively programmatic and thus a program or at least an abstract program is the best description of what should be done. An example here might be a web service that provides the distance between an address and a zip code. In other cases, the specification can get quite tricky to write while a corresponding program might be much simpler. Consider checking a robots.txt entry. The programs that do this are generally under one page of code; the specification is two pages of ambiguous but dense English text.

While these examples might seem unusual, the cases they point out are not. Most parts of most modern applications fall into one of these categories and are thus difficult or impossible to semantically specify.

4. RESEARCH

While there are a growing set of tools and applications for machine-readable specifications, the applicability of these applications is severely limited by the impracticality of developing appropriate specifications to describe modern systems. We believe that addressing these issues is and will continue to be a fruitful area for future software engineering research.

There are several directions that such research could take. One approach that might be tractable for tools that involve interaction, for example code search and semantic checking, is to let the user be part of the specification. While it might be impossible to characterize all possible outcomes of giving zero to a roman numeral routine, it is relatively easy for a user to tell if a particular result is appropriate or not. Similarly, a user would be able to tell relatively quickly if a widget or visualization looks correct or is obviously wrong. We have started to look at this both for code search and for dynamic visualization. For example, our code search front end lets the user judge whether test cases should be considered failing after they see either the textually or graphically output.

Another possible research direction involves deriving the specifications from the system. Initial work has been done here by looking at what is considered "good" code, for example established libraries, and using static or dynamic analysis to find the usage patterns for the functions in a library or the methods of a class [5]. These patterns provide a model or partial semantics specification describing how applications should behave. These specifications can then be checked either statically using techniques such as model checking or dynamically with appropriate code instrumentation.

These two approaches can also be combined. For example, our Dyview visualization system uses a combination of dynamic and static analysis to build a model of the behavior of threads, transactions and tasks in a user application [25]. However, in order to do so it needs user input to determine what are the transactions and what tasks are appropriate for understanding and hence for the visualization.

Beyond these simple approaches, we believe there is a large opportunity for new research. One approach is to develop new formalisms for specifications. For example, there have been efforts at extending finite state models with stack-based components to provide a more powerful framework both for specification and checking. Alternatively, one could look at probabilistic semantic specifications or test cases where the

results are evaluated probabilisticly. This is a step up from randomized testing [13], and is used in restricted domains such as cryptography [20]. Another direction might involve combining existing models in different ways, for example, allowing model-checking specifications as contracts.

A related research direction would involve defining specifications for components with graphical results. This could involve incorporating user interface expertise, building user interface models based on usage (e.g. looking at error rates), graph grammars, programmer feedback, and other techniques.

Another possibility would be use the program itself as the semantics and then have the user validate that these semantics are correct. This implies converting the actual program into a higher-level model that the user can then understand. This could involve providing a clear view of the program's actions for particular test cases. It could also involve using automatic test case generation to build a suite of test cases and then having the programmer validate that the outputs for the test cases are what one would expect.

This could also be done only partially. For example, one could use a combination of dynamic and static analysis to build a model of the program automatically and excluding the component of interest. This semantics of the component could then be described relative to this model.

None of these approaches solves all the problems, nor do we believe that these ideas cover all possibilities. However, they do provide a feel for what is possible and what is needed.

5. IMPACT

The availability of more program specifications (including test cases) and the ability of the programmer to either write these in a simple fashion or to have the automatically generated, will also lead to research into new research directions on how these specifications can be used to improve the programming process.

Many things are possible here. For example, if the specifications were test cases, could these be used by the compiler to do additional checking or better code generation. Specifications could be used to build models and then show visualizations that offer an understanding of the actual behavior of a complex program. Model-based specifications can be used to answer what-if questions, for example, predicting the behavior of a system if the number of threads or the number of cores available to it were to change. Finally, if we can write the specifications, why not have the computer write the code, either directly using what is considered automatic programming, or using new ideas such as extracting the functionality by doing genetic adaptation of code gleamed from the repository of over a billion lines of open source code.

6. REFERENCES

1. Godmar Back and Dawson Engler, "MJ - a system for constructing bug-finding analyses for Java," *Stanford University Computer Systems Laboratory*, (2003).

2. Thomas Ball and Sriram K. Rajamani, "The SLAM project: debugging system software via static analysis," *Proc. POPL 2002*, (2002).

3. Mike Barnett, Rustan M. Leino, and Wolfram Schulte, "The spec# programming language: an overview," *CASSIS 2004*, (2004).

4. Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan, M. Leino, and Erik Poll, "An overview of JML tools and applications," *Intl. Journal on Software Tools for Technology Transfer* Vol. **7**(3) pp. 212-232 (June 2005).

5. Jacob Burnim and Koushik Sen, "DETERMIN: inferring likely deterministic specifications of multithreaded programs.," *ICSE 2010*, pp. 415-424 (May 2010).

6. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled, *Model Checking*, The MIT Press (1999).

7. J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil, "FLAVERS: A finite state verification technique for software systems," *IBM Systems Journal* Vol. **41**(1) pp. 140-165 (2002).

8. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett, "Patterns in property specifications for finite-state verification," *Proc. ICSE 99*, pp. 411-420 (1999).

9. Naeem Esfahani and Sam Malek, "Social computing networks: a new paradigm for engineering self-adaptive pervasive software systems," *ICSE 2010*, pp. 159-162 (emscnnp).

10. Object Management Group, "Model driven architecture (MDA)," *Document ormsc/2001-07-01*, (July 2001).

11. Yuri Gurevich, Benjamin Rossman, and Wolfram Shulte, "Semantic Essence of AsmL," *Microsoft Research Technical Report MSR-TR-2004-27*, (2004).

12. J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch family of specification languages," *IEEE Software* Vol. **2**(5) pp. 24-36 (March 1985).

13. Dick Hamlet, "When only random testing will do," *Intl. Symp. on Software Testing and Analysis*, pp. 1-9 (2006).

14. Klaus Havelund and Jens Ulrik Skakkebaek, "Applying model checking in Java verification," *Proc. 5th and 6th SPIN Workshop, Lecture Notes in Computer Science* Vol. **1680** pp. 216-231 Springer-Verlag, (1999).

15. Klaus Havelund and Thomas Pressburger, "Model checking Java programs using Java Pathfinder," *Intl Journal on Software Tools for Technology Transfer* Vol. **2**(4)(April 2000).

16. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre, "Lazy abstraction," *Proc. POPL '02*, pp. 58-70 (2002).

17. C. A. R. Hoare, "Proof of a program: FIND," *CACM* Vol. **14**(1) pp. 39-45 (January 1971).

18. Daniel Jackson, "Alloy: A lightweight object modeling notation," *ACM Trans. Software Engineering and Methodology* Vol. **11**(2) pp. 256-290 (April 2002).

19. Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari, "Oracle- guided component-based program synthesis," *ICSE 2010*, pp. 215-224 (May 2010).

20. Catherine Meadows, "Formal methods for crytographic protocol analysis: emerging ideas and trends," *IEEE J. on Selected Areas in Communications* Vol. **21**(1) pp. 44-54 (January 2003).

21. Bertrand Meyer, "" "Applying "design by contract"," *IEEE Computer* Vol. **25**(10) pp. 40-51 (October 1992).

22. Sun Microsystems, *ADL Design Specification, Version 0.1*, Sun Microsystems (1993).

23. Steven P. Reiss, "A component model for Internet-scale applications," *Proc. ASE 2005*, pp. 34-43 (November 2005).

24. Steven P. Reiss, “Semantics-based code search,” *ICSE 2009*, pp. 243-253 (May 2009).

25. Steven P. Reiss and Suman Karumuri, “Visualizing threads, transactions, and tasks,” *PASTE 2010*, (June 2010).

26. Douglas C. Schmidt, “Model-driven engineering,” *IEEE Computer* Vol. **39**(2) pp. 25-31 (February 2006).

27. Ehud Y. Shapiro, *Algorithmic Program Debugging*, MIT Press (1983).

28. Mary Shaw, “Sufficient correctness and homeostasis in open resource conditions: how much can you trush your software system,” *Proc. 4th Intl. Software Architecture Workshop*, pp. 46-50 (2000).

29. Peter Wegner, “The Vienna definition language,” *ACM Computing Surveys* Vol. **4**(1) pp. 5-63 (March 1972).

30. Jeannette M. Wing and Mandana Vaziri-Farahani, “Model checking software systems: a case study,” *Software Engineering Notes* Vol. **20**(4) pp. 128-139 (October 1995).

31. J. B. Wordsworth, *Software Development with Z*, Addison-Wesley (1992).

32. Cernal Yilmaz and Clay Williams, “An automated model-based debugging approach,” *Proc. ASE’07*, pp. 174-183 (November 2007).