Verified Software: The *Real* Grand Challenge

Ramesh Bharadwaj

Center for High Assurance Computer Systems
Naval Research Laboratory
Washington DC 20375 USA
ramesh@itd.nrl.navy.mil

Abstract. This position paper addresses, and attempts to propose solutions for, critical issues in software engineering that need to be resolved before the Verified Software grand challenge as proposed by Professor Tony Hoare can be usefully exploited in industry to increase the assurance of software intensive systems.

Introduction

The following assumptions about programs and their correctness (which I refer to in the sequel as "assumptions") are implicit in the problem description of the Verifying Compiler Grand Challenge: 1) Associated with each program are types, assertions, and other annotations[1] that are readily available. 2) They are unassailable, inviolable, and invariant. 3) Their = correctness is both necessary and sufficient for the correctness of the programs they annotate. In this paper, I argue that for programs that are intended to solve real-world problems, the subject of my research for more than fifteen years, none of these assumptions necessarily holds. I proceed to explain how this problem may be addressed, and conclude with what I think are more realistic expectations on the impact of the grand = challenge problem and its solutions on real-world software development projects.

Problem Statement

For programs whose behavior is easily specified as mathematical = functions, it is conceivable that the assumptions are valid. An example of such a program is one that implements the 4-coloring algorithm for planar = graphs. If we assume that program annotations can characterize the function being computed by the program, the proof of its correctness is probably = derivable from the proof of correctness of the 4-coloring problem. However, even = for such programs, the correctness of its annotations is often predicated = upon extraneous factors in the program's execution environment, such as the = word length of the processor, the size of the address space, or the amount of available memory. This is because program code is generically written = for an abstract machine; the program may

[1] I shall loosely use the term "annotations" to refer to this redundant information.

execute on a real machine that may = not correctly implement some of these abstractions. In such an event, the = program will fail in unexpected ways. Also, program annotations may never be = able to capture quantitative aspects such as the space and time requirements of the program. Such properties are central to the program's = "correctness" since correctness often entails user expectations about the time and = space requirements for successful execution on specific data sets. Even if we assume that it is feasible to precisely characterize such machine = requirements and non-functional properties, it is not clear to me how their = correctness could possibly be established by a verifying compiler.

The situation becomes hopeless for programs the correctness of whose annotations depends upon extraneous factors. This is the case even when the specification of a program is precisely characterized as a mathematical function; the problem is, it is often impossible to ascertain with 100% accuracy what this function is. My favorite example is sales tax computation. In a bygone era, when I used to write programs for a living, I was under the naive impression that the precise nature of mathematical logic makes the problem of program correctness a mere = exercise in calculation. Imagine my surprise when, in response to my Management's decision to start charging for certain transactions, I had my first brush with sales tax laws. In the United States, for businesses that conduct transactions with customers in more than one state, correctly figuring = out the sales tax for a specific transaction can be a daunting challenge [12]. Sales tax collection falls within the purview of more than 7,500 state and local administrations, each with its own specific = set of rules and regulations. A business located in the United States is required to comply with all the regulations in effect at the location of *each* of its customers. Clearly, computing the correct sales tax is crucial to the very survival of the business.

Consider a program that is required to compute the sales tax associated with a sale: the correct tax rate varies with the location of the sale (which may not necessarily be the location of the computer on which the program is run), the sales tax to be levied at that location, and all applicable legislation(s) pertaining to the transaction[2]. For example, California law provides for the exemption of sales tax on food products subject to the following restrictions:

> Sales of food for human consumption are generally exempt from tax unless sold in a heated condition (except hot bakery items or hot beverages, such as coffee, sold for a separate price), served as meals, consumed at or on the seller's facilities, ordinarily sold for consumption on or near the seller's parking facility, or sold for consumption where there is an admission charge.

It is inconceivable that the above conditions and restrictions could be specified precisely in the form of program assertions. For example, how does one formalize notions such as "except hot bakery items" or "near the seller's parking facility?"

[2] An interesting discussion on Sales and Use taxes for transactions carried out via the Internet in the United States is found at [12].

How can one ascertain that the formulations are correct? How will a programmer devise algorithms for their computation? How is the correctness of these algorithms = established? Automatically?

One could argue that since the tax code is vague, confusing, and open to interpretation, the above example does not invalidate the aims of the grand challenge. Therefore, the argument goes, no methodology, formal or informal, can ever produce a system that is correct with respect to such an imprecise specification. An optimist may therefore suggest that one of the benefits that society will derive from this grand challenge would be to make the notion of precise and ambiguous specification a widely known and accepted concept of humankind. But, I'm too wizened and all too familiar with the fraility of human beings that I remain a skeptic. I put forth three arguments in my defence: (1) The pace of current day systems development, coupled with ever-changing requirements, and the non-technical background of major decision makers, precludes such optimistic thinking. There's never going to be enough time or money to maintain two distinct, yet accurate descriptions of the same system. (2) Even if we assume that we have the time and money to maintain a mathematically precise specification, how (who) is this going to be maintained (by)? Let's face it, we're never going to become a technocratic society. Technology is, and will continue to be understood by a small minority whose job is, and will continue to be, the "dumbing down" of systems to make them usable by the masses. Since specifications are never intended to be "run" (I consider the term "executable specification" to be an oxymoron) the social processes necessary to weed out bugs are never going to be in place. According to reliable sources [11], the "specification" in Z of the IBM CICS system was understood and read by only one member of the project team, i.e., the writer of the specification; subsequent attempts to wean developers away from their informal specification proved futile, which is when, as a last resort, English text was derived manually from the formal specification for developers to comment upon. (3) It is not the case that specifications are always clearer, more concise, or more comprehensible than the corresponding implementations (i.e., code). Case in point: In the '70s, incompleteness in the formal specification of something as trival as the routine "sort" went unnoticed by several great minds, including members of this august body. Therefore, it is my firm belief that the social processes needed to weed out the bugs in specifications are likely to be more expensive (and unnecessary) in comparison to the weeding out of bugs in the code. Specifications may not be always worth the trouble.

Requirements Specifications

My exposure to programs that solve real-world problems led me to the world of software engineering, where one addresses the problem of determining customer needs and their precise characterization in the form of a specification. By *specification* I mean a description of the *required behavior* of a system, sub-system, or component. In general, a specification describes *what* is being computed, omitting details of *how* this is achieved. Two important goals are to make the specification of a system understandable to the users of the system (to enable its *validation*) and making it precise, i.e., avoiding overspecification (also known implementation bias) as well as underspecification[3].

This is a tall order, since the two goals are often in conflict: On the one hand, the specification must be understandable to the users; therefore, its vocabulary must only include user-visible (or = environmental) quantities and exclude variables and other artefacts used in the implementation. On the other hand, since a specification is also a "build-to" document, i.e., it is the specification of the behavior of the implementation, its vocabulary must be linked to implementation = detail. One solution to this conundrum is to specify a mapping between the two behavioral descriptions (the so-called refinement mapping) – the specification and the implementation. However, providing this is infeasible in practice and I advocate instead an approach [5, 6, 9] where the implementation vocabulary includes the user vocabulary, i.e., environmental quantities associated with the externally visible behavior of the system.

This approach has two limitations: it does not address the problem of legacy systems; it also unnecessarily constrains design choices. A more general solution to this problem (also known as the "traceability problem" in requirements engineering) remains a daunting challenge. By traceability we mean a formal argument that establishes a relationship between two artefacts that describe a system at different levels of abstraction. We do not mean the manual generation and maintenance by developers of ad-hoc links (akin to hyperlinks in html) whose semantics are not interpreted or captured by the analysis tools. The set of problems whose solutions remain elusive are: 1) Reverse Engineering: Given a legacy implementation, how can one automatically extract a user-understandable description? 2) System verification: Given a user-visible specification of system behavior, how does one ensure that an implementation satisfies the specification? 3) Refinement Mapping: How are relations between user-visible and system-specific vocabularies established? 4) Requirements Traceability: Given an instance of user-visible behavior, which components of the implementation are responsible for implementing this behavior? 5) Trojans and Dead Code: Given a requirements specification, which components, sub-systems, or lines of code in the system are irrelevant to the correct operation of the system?

Domain Models

I have also explored another area in software engineering where precise notation and mathematical analysis prove to be very useful. This is in *requirements engineering*, i.e., the processes and methods employed by users and system developers to gain an understanding of the problem being solved in building the system. This is complementary to the specification based approach above and can be

[3] In other words, every implementation that satisfies the specification must be acceptable to the customer and the specification must describe every acceptable implementation.

used in addition to or instead of requirements specifications. In contrast to the conventional approach, which can be costly and time consuming, requirements engineering advocates the creation and analysis of "domain models" just for the purpose of answering specific questions about the domain [8]. The effort involved in creating such models is minimal and is comparable to the effort required to peruse prose requirements to find answers to the same questions (which often turn out to be incorrect). Using domain models, not only is there the advantage of arriving at the right answer with mathematical certainty, but as an added bonus, they uncover anomalies and raise issues about the domain that informal approaches do not. Research challenges in this area include the automatic transformation of domain models into requirements specifications, their verification, validation, and = maintenance. Other challenges are related to the challenges I enumerate above = pertaining to Requirements Specifications.

Architectural Patterns

Today's systems are built using highly reusable software or hardware components using the so called "system of systems" approach. Systems are typically built by integration of highly disparate components that interact with one another via a middleware infrastructure = [14]. Some of these components may be Commercial Off The Shelf (COTS) or = standard IP hardware components which may have been developed without taking into account the requirements of the system in which they are deployed. Further, during the design of a component, consideration of = non-functional requirements such as reliability may complicate the design. Therefore, satisfying certain requirements of the system, such as fault-tolerance, is better done at later stages of the development cycle during hardware/software integration. Since the sub-components are not easily modified during system integration, the only alternative is to implement these requirements by appropriately configuring the components so as to alter their behavior at run-time. Architectural patterns are a means to rapidly develop such mechanisms by reusing existing solutions to "similar" requirements. Using such patterns, the system integrator can quickly develop architectural models by assembling existing patterns to meet specific dependability requirements of an application. The research challenges include the automatic translation of these models into efficient runnable code, automated deployment of code on a secure, perhaps distributed platform, and initiation of repair actions in the case of hardware, network, or software failures.

We have conducted an initial study in formal verification of = architectural patterns in support of dependable distributed applications [10]. This initial study has shown that it is relatively straightforward to associate safety properties with generic modules that implement such architectural patterns. Proofs of these properties were carried out = using the standard induction technique [7] using an = assumption/guarantee proof system for compositional reasoning similar to [13]. = Although we have automated the proofs of safety properties for concrete instances = of an architectural pattern, an open problem is to develop automatic proof strategies for the generic case. Also required is a polymorphic type = system and generalized proof methodologies in support of *architectural frameworks*, which are the generators of architectural patterns.

Dependable Middleware

A goal of the NRL dependable middleware project [1–4] is to develop infrastructure to support secure deployment, coordination, security, and encapsulation mechanisms for untrusted software COTS components. With such middleware, it should be feasible to compose and deploy untrusted components in mission-critical applications, while guaranteeing the compliance of the application with performance-critical properties. Such middleware is also the enabler in the creation of service-oriented architectures (SOAs), where organizations can delegate to other organizations the responsibility of implementing, deploying, and maintaining certain functions constituting a mission-critical application. For instance, most businesses routinely use third-party vendors for carrying out credit card transactions. Getting back to the problem of sales tax computation, a business may delegate to a third = party responsibility (and associated legal liability) for computing this = function within an application. The correctness of such an application is = obviously predicated upon the correctness of these outsourced functions. = Therefore, to ensure compliance, organizations must enter into Service Level Agreements (SLAs) that are legally binding contracts similar to design contracts in object oriented programming. Automatic discovery of = services relevant to an application's requirements, protocols for automatically drawing up service level agreements, ensuring the compliance of services provided by vendors with the SLAs, dynamic composition of available services to meet the requirements of a specific mission-critical application, and verifying that the composed application meets its performance-critical properties, are some of the multitude of challenges posed by application development for service-oriented architectures.

Conclusion

In this position paper, I have made an attempt to put into perspective the daunting challenges associated with Verified Software. In my opinion, the Verified Software grand challenge is merely a good start for developing methods and tools to solve the more challenging problems of the software development industry. It is hoped that the attendees of the IFIP Working Conference on Verified Software: Theories, Tools, Experiments, will give thought to these additional challenges, and propose a road map for tackling some of these more pressing problems. I think the Computer Science community has abdicated responsibility for improving the state-of-practice of software development many years ago. It is my earnest hope that this forum will serve as a springboard to invigorate the community into making genuine research contributions that have the potential to truly transform the software development process into an engineering activity. In other words, to put the "engineering" back into software engineering.

References

1. R. Bharadwaj. SINS: a middleware for autonomous agents and secure code mobility. In *Proc. Second International Workshop on Security of Mobile Multi-Agent Systems (SEMAS-02)*, Bologna, Italy, July 2002.
2. R. Bharadwaj. Verifiable middleware for secure agent interoperability. In *Proc. Second Goddard IEEE Workshop on Formal Approaches to Agent-Based Systems*, Greenbelt, MD, October 2002.
3. R. Bharadwaj. A framework for the formal analysis of multi-agent systems. In *Proc. Formal Approaches to Multi-Agent Systems*, Warsaw, Poland, April 2003.
4. R. Bharadwaj. Development of dependable component-based applications. In *In Proc. First International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004)*, LNCS. Springer, 2004.
5. R. Bharadwaj and C. Heitmeyer. Hardware/software co-design and co-validation using the SCR method. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT'99)*, San Diego, CA, November 1999.
6. R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the SCR requirements methods. In *Proc. 19^{th} IEEE Digital Avionics Systems Conference*, Philadelphia, PA, October 2000.
7. R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc.* 6^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 378–394, Berlin, March 2000.
8. Ramesh Bharadwaj. Formal analysis of domain models. In *Proc. International Workshop on Requirements for High Assurance Systems (RHAS'02)*, Essen, Germany, September 2002.
9. C. Heitmeyer and R. Bharadwaj. Applying the SCR requirements method to the Light Control Case Study. *JUCS*, 6(7), 2000.
10. R. L. Jeffords and R. Bharadwaj. Formal verification of architectural patterns in support of dependable distributed systems. In *Submitted.*, 2005.
11. David L. Parnas. *Private Communication.*
12. TurboTax. FAQs on Sales and Use Taxes and the Internet. URL-http://www.turbotax.com/articles/FAQonSalesandUseTaxesandtheInternet.html.
13. Fei Xie and James C. Browne. Verified systems by composition from verified components. In Paola Inverardi, editor, *Proc. Joint 9th Eur. Softw. Eng. Conf. (ESEC) and 11th SIGSOFT Symp. on Foundations of Softw. Eng. (FSE-11)*, pages 277–286, Helsinki,Finland, September 2003.
14. Stephen S. Yau, Supratik Mukhopadhyay, and Ramesh Bharadwaj. Specification, analysis, and implementation of architectural patterns for dependable software systems. In *Proc. 10th IEEE Int'l Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005)*, Sedona,AZ, February 2005.