Refocusing the Verifying Compiler Grand Challenge

Joan Krone, William F. Ogden, Murali Sitaraman, and Bruce W. Weide

Technical Report RSRG-08-01
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

May 2008

Refocusing the Verifying Compiler Grand Challenge

Joan Krone[1], William F. Ogden[2], Murali Sitaraman[3], Bruce W. Weide[2]

[1] Denison University, Granville, OH 43023, USA
krone@denison.edu
[2] The Ohio State University, Columbus, OH 43210, USA
{ogden, weide}@cse.ohio-state.edu
[3] Clemson University, Clemson, SC 29634, USA
murali@cs.clemson.edu

Abstract. The ideal goal of this grand challenge should be a future in which no production software is considered properly engineered unless it has been fully specified and fully verified as satisfying its specifications. The verifying compiler then becomes the essential central artifact necessary to achieve this outcome, and its characteristics are determined by the overall goal. From this perspective, the nature of programming languages that a verifying compiler could process becomes an immediate issue, and we present several critical features that such a programming language must possess. Specifically, it must include specifications as an integral constituent, and it must have clean semantics, which preclude unexpected side-effecting, aliasing, etc. It must include mechanisms for writing reusable components that are amenable to verification, and consequently, it must include an open-ended mechanism for adding arbitrarily sophisticated mathematical theories in order to specify large software components concisely. Because the current programming languages lack these essential characteristics, the verifying compiler grand challenge will not be met unless it redirects its focus to include the development of a suitable programming language within which full verification is possible.

Keywords: assertive language, specification, verification, clean semantics

1 Introduction

In order to achieve maturity as a field, software engineering must move from its current cut-and-try approach to a rigorous mathematically based system for engineering software. The essential features of such a system must clearly include a programming language in which sophisticated, clean software can be written, and a specification system in which concise, precise intentions for the behavior of software can be expressed.

To insure the soundness of the overall software verification system proposed in the grand challenge [1], the specification mechanism and the programming mechanism must be fully coordinated in every detail, and about the only way to guarantee this is to integrate them into an assertive programming language. The correctness objectives for the verification system and for the compiler can then be unified via a shared

semantics for the language, and the all too common problem of incorrect behavior by "verified" software can be avoided. The need for a programming language within which specifications are an integral feature is the first of numerous indications that current programming languages are not adequate for meeting the grand challenge.

Another common problem occurs when verification is attempted in programming languages that lack clean semantics. By clean semantics, we mean that all operations constructible in the language can only affect the objects to which they appear to have access. Typical cleanliness flaws arise from object aliasing, passing repeated arguments to operations, etc. The problem is that putatively "verified" constituents may not behave correctly when employed in a larger system.

For a programming language to have clean semantics, the built-in data types must be clean, and the composition mechanisms must preserve cleanliness. Constructing such a language by merely restricting oneself to a sublanguage of a language that didn't have clean semantics as a primary design goal will essentially guarantee the creation of an unsatisfactory product because the resulting language will be impractically weak. A new language is required.

A major source of problems confronting verification is scale. Viewed from the specificational perspective, one such problem is that descriptions of the intended effects of programs would seem to have to grow roughly in proportion to the size of the code. Given the limitation of human cognitive capacity, this is a very serious concern.

The solution to the coding side of this scaling problem is to provide modularization mechanisms that support a divide and conquer approach via componentization of software, with programming taking place using progressively more powerful components. An analogous approach is required on the specificational side, with descriptions of more powerful components being formulated in terms of more sophisticated theories. The net effect is that, in order to support the growth of software driven by the rapidly increasing power of hardware, the collection of mathematical theories used in specifying software must remain open ended. An immediate corollary is that machinery for developing mathematical theories must be a third constituent of the language for programming and specifying software; it just is not possible to know *a priori* which theories will ultimately be needed.

Even with well-conceived program verification machinery, the cost of evolving poorly structured software into correct software is bound to remain prohibitively high. One of the primary strategies used by more mature engineering disciplines for achieving sound products at reasonable cost is to rely upon a comparatively small collection of highly reusable components, and this must surely be an approach that's strongly supported by a language for software verification.

There are several key features that the machinery provided for supporting reusable components must have. Certainly it must exhibit the clean semantics mentioned above, since modular understanding is always essential to reuse. Second, it must provide the potential for a high degree of genericity, since keeping catalogues of reusable components of an intellectually manageable size is important. Third, it must provide an interfacing mechanism for presenting the object types and operations together with their abstract specifications, since information hiding is critical to keeping the specifications of higher-level code as simple as possible. Fourth, it must support the development of alternative implementations of an abstract interface, since

different implementations of the same functionality are necessary to meet different performance goals, and if a component interface does achieve the desired degree of reusability, then it must be possible in a large system to deploy it in numerous places where varying performance requirements hold.

Reusable components are the setting in which the specificational simplification derived from changing to more sophisticated mathematical theories frequently occurs. So part of the machinery in a component implementation must provide the specification of a correspondence relation that properly matches the behavior of the entities at the implementation level with functionality prescribed for the more abstract entities presented by the component's external interface. Performance specification and verification capability is also essential for a satisfactory software verification system, and accordingly, component implementation machinery must provide for translating this information up to the external interface.

2 Language Support for Specifications

If systems that don't share a common design are combined to write and verify software, the slightest of inconsistencies in their semantics could easily vitiate apparent correctness results. Consequently, we need one language that treats the development of software systems as an integrated whole. In particular, software's specifications should be viewed as an essential part of the software, and not as an add-on sideshow that might or might not describe the actual code.

For verification purposes, all operations need syntactic slots where software engineers can express the pre and post conditions that describe the situations in which an operation can be invoked and the outcomes that it is designed to produce. These specificational constructs can produce effects involving two special semantic states: a vacuously correct state, VC, and a manifestly wrong state, MW. If, for example, some code attempts to invoke an operation in a state that doesn't meet the operation's precondition, then the resulting state is MW. Similarly if the code for an operation doesn't meet its post condition, then the outcome is also MW. The VC state is introduced when the code for an operation is started in a state that doesn't meet its pre condition. A program is semantically correct only if under no circumstances can it produce the MW state. In such an integrated approach, the potential for a verification system to be unsound is vastly reduced.

A compiler is only going to be capable of verifying the correctness of assertive code if that code includes sufficient hints in the form of justificational specifications, which are provided by the software engineer, to make intermediate deductions "obvious." Besides pre and post conditions for operations, the language will need to support invariants for its looping constructs, invariants and abstraction relations for its component modularizing mechanism, etc. To account for tricky code, constructs for adding specificational claims as supplemental hints will be necessary. The semantics of such specificational constructs can also be described easily using the VC and MW states.

Other specificational constructs will be required to account for the ordinal valued progress metrics required to ensure the termination of loops and recursive procedures.

Similarly, no verification system would be considered adequate in the long run if it didn't account for performance. So there must be slots for integer valued expressions used to specify the memory displacements required by objects and transitorily by operations. The slots that account for the durations of operations, loops, etc. will hold real-number-valued expressions, so it's clear that the verification system must deal with a variety of number systems before it even begins to work with anything introduced to specify particular components.

3 Clean Semantics

Correct reasoning about software, both formal and informal, is critically dependent on "separation of concerns." If a piece of code appears to be working on only a small portion of the overall state space, then any efficient verification system must be safe in restricting its attention exclusively to the code's effect on that subspace. Languages that restrict the effects of each programming construct to just the objects that are syntactically targeted by the construct are said to have clean semantics [2], so a language with clean semantics is an important requirement if verification is to succeed.

Languages that rely upon reference copying, permit out-of-range referencing, etc. lack clean semantics, and any sound system for reasoning about them is going to be impossibly complex. So if well-engineered software is to be verified software, legacy software is best viewed as being merely prototype software.

A language suitable for verification must be designed to encapsulate pointers within components that are carefully engineered to present clean, efficient interfaces to programmers. It must also have an efficient way to pass parameters to procedures such that repeated arguments don't introduce aliasing.

4 The Mathematical Theory Constituent

In the thirty or so years since the De Millo, Lipton, and Perlis critique of the prospects for program verification [3], our understanding of automated proof systems has made modest progress, our computers' speed has made dazzling progress, and the social "proof" process for verifying the correctness of general mathematics has changed hardly at all. Consequently, results found in the mathematical literature still somewhat resemble software in that the parts that have been heavily used are pretty much correct, while both the rarely used and newly developed parts are more suspect.

There's no particular reason to believe that the most appropriate theories for specifying the full compass of software applications will inevitably lie within the well worked parts of mathematics, so the reliability of the general software verification process becomes quite suspect unless it rests on a firmer foundation than citations into the mathematical literature. In short, the mathematics used in software specification and verification needs to be industrial strength rather than craftsman formulated. A system for developing, checking, and cataloguing mathematical theories then becomes an essential component of a software verification system.

Brief reflection reveals that the notations and results developed in mathematical theories are not independent of program specifications or of the verification process. In fact, it's essential that the specifications and the mathematical theories share a common semantics, and that makes the mathematical theories and the syntactic constructs used in their formulation a part of the language used for programming and specification.

Several ideas from software engineering are also appropriate for structuring mathematical theories. The first is separation of concerns. A client using a theory to formulate specifications only needs a summary or précis of the definitions and results for that theory, but not anything about proofs for the results, so the précis should be in a separate syntactic unit from the proofs. This is analogous to separating interfaces and implementations in the programming constituent of the language in that a client of an interface in both cases only needs to know that the supporting unit exists but can safely ignore the messy details left to another specialist.

A second such idea is reuse. Well-considered and well-developed mathematical theories are appropriate for a variety of specifications, and the cost of their formulation and proof can be amortized over all these uses.

The form and degree of development of libraries of theories is a primary determiner of whether the fully automatic verification of particular software can succeed. Results provided by the theories are to be proved off-line by mathematically proficient specialists, so they can be arbitrarily difficult. A verifying compiler itself, being completely mechanical, is never going to be able to prove anything but easy stuff. So it's up to the theory developers and the software specifiers to leave only a narrow gap between the strong results in the theory development and the specifications and specificational hints in the software. Given this objective, these theory developments may contain some results that are a bit too obvious for a traditional mathematical development, but are an essential aid for a verifying compiler.

Making the verification of production software routine depends on a taxonomic thesis about how software engineers create software that they "know" is correct. The thesis is that most of such code is straightforward and it's plain to see that it is correct. The remaining not-so-obvious parts are separable from the rest, and certainty of the correctness of each such part is developed through a serious individual process of abstract reasoning. If this thesis is correct, then a software verification system can achieve its objectives using two qualitatively different subsystems. The first addresses the not-so-obvious and is the general mathematics subsystem that handles theory and proof modules. The second is a code justification checker that examines the specifications embedded in code to determine whether they are "obviously" correct, given the specifications and annotations in the code and the definitions and theorems developed in the supporting theories.

A language for developing verifiable software must be simultaneously cognizant of the limits of mechanization and of the needs of language users. For example, both because mathematicians generally will be needed to develop the non-trivial proofs upon which some software relies and because traditionally educated software engineers will be required to read and write specifications, it is essential that the software verification system present theories and specifications in standard mathematical notation.

5 Reusable Component Support

Reusable components would seem to be inextricably connected to verification. If components are to be deployed across a large base of installed software, then it is essential that they be correct, since the cost assessment associated with any residual errors contains a huge multiplier. By the same token, when highly reusable components are fully specified and verified, the cost can be amortized over a large base of usage.

Yet nurturing this symbiotic relationship does not appear to have been a serious concern. Notions such as reuse via inheritance are widespread, when most uses of inheritance clearly exhibit terrible coupling properties and thoroughly undermine modular verification.

The component interfacing machinery supporting verification must cleanly decouple the implementations of components from their deployments. This involves interfacing mechanisms for clearly identifying all the entities, both fixed and parametric, that are employed by a component, the properties the parameters must possess, and the properties of the objects and operations provided by the component that can be relied upon in all deployments.

Because maximal generality is essential, the component parameterization mechanism must be designed to support full situational flexibility. For example, a parameterization mechanism that allows the passing of object types and their operations only as whole components forces stamp coupling in situations when only a few of the operations are needed, and this clearly restricts the number of situations in which such components could be employed. A reuse supportive mechanism would allow fine grained parameter passing for components.

The principal goal of the decoupling provided by component interfacing machinery is information hiding, and the abstract specifications provided for exported objects and operations provide the cover stories behind which myriad sordid implementation details can be hidden [4]. These "abstract" specifications become the concrete specifications when a component is used to construct higher level software, so keeping them as simple as possible is essential if the verification of such higher level software is to remain tractable.

Two properties of a verification system are the keys to allowing software engineers to achieve such simplicity in their component specifications. The first is that the system have clean semantics, for the reasons previously discussed. The second is for the system to provide semantics that are *rich*. By this we mean that if a software engineer can come up with any mathematical system in terms of which he can provide a complete, accurate explanation of the behavior of a component, then the verification system should fully support his use of that mathematical system in his specification of the component. The idea is that a verification system that has rich semantics can't preclude the development of component specifications that are the simplest possible. For example, forcing a programmer/specifier to express every component description in terms of a single theory, such as set theory, would lead to possibly accurate, but unreadablely complicated specifications.

We conjecture that there is no practical way to distinguish the mathematical theories that might ever be useful across the ever expanding sphere of software applications, and consequently that the compass of the mathematical theory

constituent of a software verification system would be coextensive with the domain of general mathematical theories. An interesting corollary is that, semantically, a verification system would have to support programming in arbitrary mathematical domains because, when any component is deployed, the code that uses it will be verified against a semantics based upon that component's mathematical model.

The use of different mathematical theories makes it possible to provide nicer cover stories for implementations using specifications based on uglier theories. This means that, when the system is verifying implementations, it's going to need programmer supplied specifications for correspondence relations between the two theories so that pre and post conditions written in the cover story theories make sense in terms of the theories employed at the implementation code level. If the mathematics in the cover stories really is simpler, then the extra verificational complexity introduced here will be recompensed many times over – especially if the component is highly reusable.

Another major benefit of information hiding is that it allows software designers to develop alternative implementations that all match the abstract cover story presented by a single component interface but that differ in structural details and performance characteristics. If a verification system supports plug compatible interchange in such situations, it can achieve significant savings because the functional reasoning about any deployment of the component depended entirely on the specifications in the component's interface and not at all on details in an implementation. Of course the details of an implementation do dictate what correspondence relation is needed to make them match with the cover story, so each implementation module is going to need its own correspondence specification.

There's also an advantage to offering strong support for developing broadly unifying component interfaces that bring together a multitude of alternative implementations, which now clutter "component" libraries with similar but not quite compatible entries, into a relatively few standard and highly reusable components that are much easier for programmers to understand and employ. The payoff for verification from the increased use of components is obvious.

Efficiency is also a critical consideration for verifiable software in general and for reusable components in particular. As noted, the programming constituent of a verification system should not be so bereft of built-in capabilities that it prevents software engineers from developing software that is essentially as efficient as that developed in current programming languages.

When it comes to components, there's a concern lest specificational deficiencies prevent software engineers charged with implementation from achieving full efficiency. For example, with many sorts of components there's an operation to return from a colletion object an item meeting certain criteria. From a client perspective, the collection may well hold several items all meeting his criteria and it's a matter of complete indifference which is returned, so the natural input/output specification is relational. If the verification system only handles functional specifications, then the operation's post condition will have to be made unnecessarily exacting (which may even involve complicating the model of the collection). The result would be over specification and the exclusion of potential implementations having better performance in either time or space. Clearly, a verification system needs to support relational specifications.

In component based software, as we've noted, "abstract" specifications for a component become concrete specifications where that component is deployed, and when component specifications are relational, this means that the semantics of the programming component of a verification system must become relational. Because traditional denotational semantics assume that the semantics will be functional (i. e., for an input state, produce a single output state), the semantics of a verification system must be established on a new basis. This is quite doable, but it does involve moving the fixed point result from the classical continuous function based argument to one involving taking limits over ordinals.

6 Conclusions

If the verifying compiler grand challenge is going to be met, then these are some key characteristics of any programming language that the compiler can target. They plainly don't match the characteristics of current languages. Recognizing this means that no more energy needs to be wasted trying to cope with some of the pathologically byzantine features of these languages, and that the fettered, de facto language redesign activity that is language subsetting can be abandoned in favor of a thorough, verification-driven language redesign.

In effect, language development constitutes the bulk of the project because, as we've noted, the overarching soundness requirement means that the language must include mechanisms for both code specification and mathematical development of the theories used in these specifications.

A verifying compiler then must process all these language constituents, so that it effectively includes components that perform the traditional syntax checking and code generation augmented by those that check the results in the mathematical theories and that check the correctness of claims made in the program specifications. As mentioned, the language must be cleanly modular so that separate compilation of supporting theory units, software components, etc. can be standard operating procedure. A more subtle point is that the verification system must maintain complete control of previously compiled modules if the integrity of verifications is to be maintained.

Succeeding with the verifying compiler project still won't mean that all the software the compiler processes is absolutely correct because that software may not have been properly specified to meet the objectives of the real world system in which it is to be embedded. The analysis of requirements and the specification of real world systems with embedded software will almost certainly require even more complex techniques than those for specifying and verifying correctness within the relatively pristine world of imperative programming. However, complete specification and verification of software would imply a clear separation of concerns. Questions about the correctness of systems in which software is embedded could be fully addressed by looking only at the specifications for that software, with absolutely no consideration of coding details, and questions about the correctness of software could be fully addressed with absolutely no consideration of the systems into which it is to be embedded.

Developing a verifying compiler would certainly represent a major advance for our field, but the challenge will only be met when the realities of what's involved are squarely faced.

Acknowledgments. We wish to thank our research groups for their contributions over the decades to the ideas discussed here. Particular thanks are due to Joe Hollingsworth for challenging us to provide a rationale for the high level requirements and design scheme for the verifying compiler. Special thanks are also due to Greg Kulczycki for championing the important objective of clean semantics. Various ideas presented here were developed under NSF grants DMS-0701260, CCR-0113181, CCR-0081596, CDA-9634425, DUE-9555062C, CR-9311702, CCR-92044611, CCR-9111892, CCR-8802312, and DUE-0633055.

References

1. Hoare, Tony: The Verifying Compiler, a Grand Challenge for Computing Research. JACM 50, 1 63--69 (2003)
2. Kulczycki, G., Direct Reasoning. Ph. D. Dissertation, Clemson University, 183 pages (2004)
3. DeMillo, Richard A., Lipton, Richard J., Perlis, Alan J.: Social Processes and Proofs of Theorems and Programs. Comm. ACM 22, 5, 271--280 (1979)
4. Parnas, D.L.: A Technique for Software Module Specification with Examples. Comm. ACM 15, 5, 330–336 (1972)