A Language for Building Verified Software Components

Gregory Kulczycki[1], Murali Sitaraman[2], Joan Krone[3], Joseph E. Hollingsworth[4], William F. Ogden[5], Bruce W. Weide[5] , Paolo Bucci[5], Charles T. Cook[2], Svetlana V. Drachova-Strang[2], Blair Durkee[2], Heather Harton[6], Wayne Heym[5], Dustin Hoffman[5], Hampton Smith[2], Yu-Shan Sun[2], Aditi Tagore[5], Nighat Yasmin[7], and Diego Zaccai[5]

[1] Battelle Memorial Institute, Arlington, VA, USA
kulczyckig@battelle.org
[2] School of Computing, Clemson University, Clemson, SC 29634, USA
{murali,ctcook,sdracho,bdurkee,hamptos,yushans}@clemson.edu
[3] Mathematics and Computer Science, Denison University, Granville, OH 43023, USA
krone@denison.edu
[4] Computer Science, Indiana University Southeast, New Albany, IN 47150, USA
jholly@ius.edu
[5] Computer Science and Engineering, Ohio State University, Columbus, OH 43210, USA
{ogden,weide,bucci,heym,hoffmand,
tagore,zaccai}@cse.ohio-state.edu
[6] Integrated Support Systems, Seneca, SC 29672, USA
hkeown@g.clemson.edu
[7] Computer Science, University of Mississippi, Oxford, MS 38677, USA
yasmin@clemson.edu

Abstract. Safe and secure reuse is only achievable by deploying formally verified software components. This paper presents essential design objectives for languages for building such components and highlights key features in RESOLVE—a prototype for such languages. It explains why the language must include specifications as an integral constituent and must have clean and rich semantics, which preclude unconstrained aliasing and other unwanted side-effects. In order that arbitrarily complex components can be given concise and verification amenable specifications, an adequate language must also include an open-ended mechanism for incorporating additional mathematical theories. Given these essential characteristics, safe and secure reuse cannot be attained within popular languages, such as C++ or Java, either by constraining them or by extending them. Better languages are necessary.

Keywords: assertions, clean semantics, components, reuse, specification.

1 Introduction

In order to achieve maturity as a field and to build safe and secure high assurance systems, software engineering must move from its current "cut-and-try" approach to a rigorous mathematically based system for engineering software. This engineering requires a language carefully designed to facilitate construction of verifiable and

J. Favaro and M. Morisio (Eds.): ICSR 2013, LNCS 7925, pp. 308–314, 2013.

reusable software components and a verifying compiler—a compiler that checks that code is correct and generates executable code. This is unarguably a grand challenge for the computing community [1]. This paper motivates and delineates the essential features of a language or framework for building verified components. Understanding the features will help not just language designers, but also component developers in existing languages, informing them of potential pitfalls when features of their language are in conflict with the goal of verification.

While it is difficult to retrofit currently popular languages with features amenable for verification, the features themselves are not unrealizable. Automated verification efforts summarized in [2], for example, include one or more of these features. RESOLVE is a more comprehensive effort [3, 4, 5, 6]; its web IDE (available at www.cs.clemson.edu/group/resolve) allows reuse of existing components (ranging from ones for Arrays to Maps, Prioritizers, and Pointers) and construction of new ones [7]. A key question for every verification effort is one of scale. In answering this question, the important observation is that specifications for capturing human-understandable component behavior are necessarily simple in a language with clean semantics [8], given an extensible mathematical language [9] and that reasoning of correctness for any code that is straightforward for humans, is also straightforward for automated verifiers, given suitable annotations [10]; no deep thinking is necessary. The challenge is in investing the effort in devising suitable specifications and annotated implementations.

2 Essential Features of a Language for Verified Components

The essential features of a system for building verified software components must clearly include a language in which sophisticated, clean software can be written, and a specification system in which concise, precise intentions for the behavior of software components can be expressed. To insure the soundness of the verifying compiler, the specification and the programming mechanism must be fully coordinated in every detail, and about the only way to guarantee this is to integrate them into a single assertive 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 seemingly verified software can be avoided. The need that specifications be an integral feature is the first of numerous indications that current languages are not adequate for meeting the grand challenge.

Another common problem occurs when verification is attempted in programming languages that lack clean semantics [8]. By clean semantics, we mean that all operations constructible in the language can only affect the objects to which they appear to have access. Without such semantics, seemingly "verified" constituents may not behave correctly when employed in a larger system. For a language to have clean semantics, its built-in data structures and composition mechanisms must be clean, and unconstrained aliasing must be avoided. Unfortunately, merely constraining an existing language to a clean subset will lead to an impractically weak language.

A major source of problems confronting verification is scale. From the specificational perspective, one such problem is that descriptions of the intended effects of programs might have to grow roughly in proportion to the size of the code. Given the limitation of human cognitive capacity, this is a 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 large system construction 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 the collection of mathematical theories used in specifying software must remain open ended in order to support the growth of software driven by the rapidly increasing power of hardware. So an immediate corollary is that machinery for developing mathematical theories must be a third constituent of the language for specifying software that we want to program.

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 is strongly supported by a language for software verification.

There are several key features that the machinery provided for generating 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 to 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, and so, component implementation machinery must provide for translating this information up to the external interface.

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 [8], so a language with clean semantics is a basic requirement if verification is to succeed.

The biggest impediment to clean semantics in a language is unconstrained and avoidable aliasing. As reference copying is the main cause of such aliasing, to support clean semantics without sacrificing efficiency, the language must support mechanisms to avoid reference copying (e.g., swapping or transfer) and parameter aliasing [8, 11]. However, this does not mean all pointers and aliasing can or should be avoided [12].

4 Language Support for Specifications

If languages or systems that do not share a common design are combined to specify, 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.

A clean specification of List abstraction (devoid of complications due to pointers) is given in [3] and an updated, verification-friendly version of that specification, can be found in the RESOLVE Web IDE [7]. In the specification, a list is conceptualized as an ordered pair: a mathematical string of entries that precede the insertion point (denoted by *Prec*) and a string of entries that remain past the insertion point (denoted by *Rem*). A part of this specification is shown in the screen insert to the right in Figure 1. The screen insert at the bottom shows a formal specification of a list reversal operation, named Flip_Rem; this operation is an enhancement (or extension) to the list component. In this specification, Reverse is a mathematical function that reverses a string; its formal definition is given in the next section. It is specified to take a list, such as (<>, <1,2,3,4>) and produce (<4,3,2,1>,<>).

The meaning of correctness of an implementation of Flip_Rem (in Figure 1) depends on its specification and the specifications of operations it uses; i.e., the same code may be correct or wrong, underscoring that specifications should be an integral part. Stated more formally, pre and post conditions 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 does not meet the operation's precondition, then the resulting state is MW. Similarly if the code for an operation does not 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 does not 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, provided by the software engineer, to make intermediate deductions "obvious." Besides operation specifications, the language must support invariants and termination progress metrics for its looping constructs, representation invariants and abstraction relations for data abstraction implementations, among others [2].

5 Reusable Mathematical Theory Constituent

It is unlikely 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 [13], unless it rests on a firmer foundation than citations into the mathematical literature. In short, the mathematics used in software specification and verification must 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 [9]. For example, while a theory of mathematical strings could be codified in a specification language and employed for effective verification [5], in general, the language should make it possible to define and use new theories without modifying the verifier.

Several ideas from reusable component 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 (theorems) 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 of components. A second such idea is reuse itself. 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.

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 is plain to see that it is correct [10]. 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.

6 Verified Reusable Components

When reusable components are fully specified and verified, the cost can be amortized over a large base of usage. This is possible if verification supportive component interfacing machinery cleanly decouples the implementations of components from their deployments. This decoupling is the motivation for the abstract specification of List component. Using that specification, it becomes possible to cleanly verify the realization (code) in Figure 1 is correct with respect to the specification of the enhancement operation Flip_Rem, also shown in the figure.

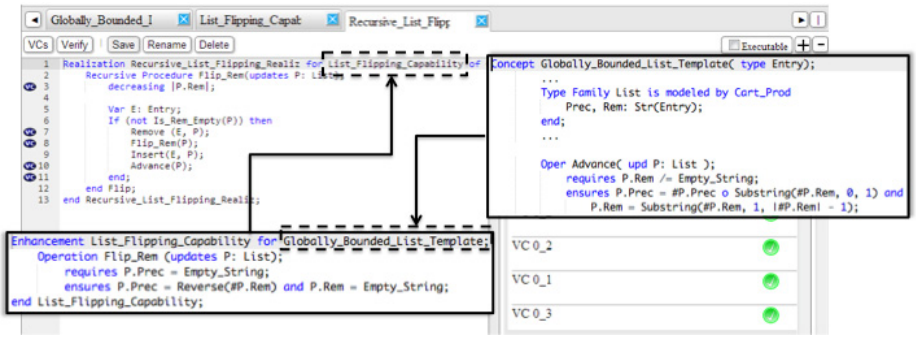

Fig. 1. Example Reusable Component Verification

The code in Figure 1 is the same as the one presented in [1], except that this one has now been mechanically verified. The syntactic slot for the decreasing clause in this recursive procedure enables a software engineer to annotate the code and facilitate automatic proof of termination. Here, in the blue ovals down the left hand side, the term VC stands for verification condition. The RESOLVE verifier has analyzed "Flip_Rem" code using its specification and specifications of operations it reuses, generated the VCs for total correctness (detailed in [3]), and proved them. Though not all components at the RESOLVE web IDE are verified or even verification amenable because this is still ongoing research, it is a useful prototype.

7 Conclusions

To meet the challenge of building verified software components, a verification-driven language design is necessary. Characteristics of such a language do not match those of currently popular languages. Recognizing this means that neither constraining an existing language nor adding on is a viable strategy. In particular, the overarching soundness requirement means that mechanisms for both specification and mathematical development of the theories used in these specifications must be an integral part of the language.

Succeeding in the goal of building verified software components still will not 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. Regardless, developing a verifying compiler and building verified components would certainly represent a major advance for our field, but the challenge will only be met when the realities of what is involved are squarely faced. Among these realities is the need to educate the next generation of software engineering workforce on the principles of verified software construction [14].

Acknowledgments. We wish to thank our research groups for their contributions to the ideas discussed here. We also acknowledge United States National Science Foundation grants CCF-0811748, CCF-1161916, DUE-1022191, and DUE-1022941.

References

1. Hoare, C.A.R.: The Verifying Compiler. A Grand Challenge for Computing Research. JACM 50(1), 63–69 (2003)
2. Klebanov, V., et al.: The 1st Verified Software Competition: Experience Report. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011)
3. Sitaraman, M., et al.: Reasoning About Software-Component Behavior. In: Frakes, W.B. (ed.) ICSR 2000. LNCS, vol. 1844, pp. 266–283. Springer, Heidelberg (2000)
4. Sitaraman, M., Adcock, B., Avigad, J., Bronish, D., Bucci, B., Frazier, D., Friedman, H.M., Harton, H.K., Heym, W., Kirschenbaum, J., Krone, J., Smith, H., Weide, B.W.: Building a Push-Button RESOLVE Verifier: Progress and Challenges. Formal Aspects of Computing 23(5), 607–626 (2011)
5. Adcock, B.: Working Towards the Verified Software Process. Ph. D. thesis, Computer Science and Engineering, The Ohio State University (2010)
6. Harton, H.K.: Mechanical and Modular Verification Condition Generation for Object-Based Software. Ph. D. Dissertation, Clemson University, 305 pages (2011)
7. Cook, C.T., Harton, H.K., Smith, H., Sitaraman, M.: Specification engineering and modular verification using a web-integrated verifying compiler. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 1379–1382 (2012)
8. Kulczycki, G.: Direct Reasoning. Ph. D. Dissertation, Clemson University, 183 pages (2004)
9. Smith, H.: Engineering Specifications and Mathematics for Verified Software. Ph. D. Dissertation, Clemson University, (to appear, 2013)
10. Kirschenbaum, J., Adcock, B., Bronish, D., Smith, H., Harton, H., Sitaraman, M., Weide, B.W.: Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping? In: Edwards, S.H., Kulczycki, G. (eds.) ICSR 2009. LNCS, vol. 5791, pp. 31–40. Springer, Heidelberg (2009)
11. Harms, D.E., Weide, B.W.: Copying and Swapping: Influences on the Design of Reusable Software Components. IEEE Transactions on Software Engineering 17(5), 424–435 (1991)
12. Kulczycki, G., Smith, H., Harton, H., Sitaraman, M., Ogden, W.F., Hollingsworth, J.E.: The Location Linking Concept: A Basis for Verification of Code Using Pointers. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 34–49. Springer, Heidelberg (2012)
13. DeMillo, R.A., Lipton, R.J., Perlis, A.J.: Social Processes and Proofs of Theorems and Programs. Comm. ACM 22(5), 271–280 (1979)
14. Cook, C.T., Drachova, S., Sun, Y.-S., Sitaraman, M., Carver, J., Hollingsworth, J.E.: Specification and reasoning in SE Projects Using a Web IDE. In: Proceedings Conference on Software Engineering Education & Technology, CSEE&T (2013)