Procedure Calls and Local Certifiability of Component Correctness

Bruce W. Weide
Wayne D. Heym
William F. Ogden

Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, OH 43210
Tel: (614) 292-1517 (Weide), fax: (614) 292-2911
Email: {weide,heym,ogden}@cis.ohio-state.edu

Abstract

Results from work involving formal methods in software engineering often can be interpreted in the context of practical software engineering problems, and thereby contribute to solutions. One example of this phenomenon involves an understanding of the need for restrictions imposed by "toy" languages that admit modular program verification systems. This analysis can be applied to "real" languages that are ostensibly intended to support component reuse (e.g., Ada and C++). The result is that unless we restrict procedure calls in ways that are not — indeed, cannot be — checked or enforced by compilers for these languages as presently defined, it is impossible to develop a modular system for reasoning about program behavior. However, by adopting an alternate explanation of parameter passing, call-by-swapping, we can remove the procedure-call impediment to a modular reasoning system without restricting calls in any way.

Keywords: Reuse, software component, local certification, proof of correctness, verification, aliasing, parameter passing, call-by-reference, call-by-swapping

Workshop Goals: Learning; networking; having an opportunity to get feedback from others on our ideas, and to provide same for others' ideas

Working Groups: Design guidelines for reuse, Reuse and OO methods, Reuse and formal methods, Reusable component certification, Reuse handbook, Education

1 Background

Our Reusable Software Research Group has done extensive work on the technical barriers to reuse, two examples of which are directly relevant to the remainder of this paper and are discussed in Subsections 1.1 and 1.2. This work has been supported by the National Science Foundation since 1988; current support is under grant CCR-9111892.

1.1 Local Certifiability

Last year at this workshop we discussed the idea of "local certifiability" and showed why it should be required of any serious software engineering discipline [1]. We proposed that support for local certifiability should be a litmus test for any proposed programming or software engineering methodology: If we design software in this way, can we reason about our programs in a modular (i.e., component-wise) fashion? The working group on design for reuse unanimously acknowledged the importance of designing for local certifiability [2] and Ware Myers' workshop summary [3] for *IEEE Software* highlighted local certifiability as one of the key technical issues discussed by the workshop participants.

Local certifiability of a property is the ability to establish that property for a software component out of the context of any particular client of the component. In this paper we deal with the property of correctness with respect to an abstract specification, and consider procedures as the components in question. So suppose we have an abstract specification of what a procedure is supposed to compute and an implementation of that procedure. With local certifiability, we can establish once and for all that the implementation is correct, put the procedure header and the body's *object* code in a reuse library, and henceforth trust that code to compute what the specification says it will compute; we don't have to look at the *source* code for the procedure body to reason about the caller's behavior. Without local certifiability, in some cases we might have to "expand" the source for the procedure body into the context of the client in order to determine what it will do for a particular call; we can't always rely on the procedure's abstract specification to tell us this.

Local certifiability of correctness is equivalent to the ability to reason *modularly*, on a component-wise basis, about software systems. Our argument for the intrinsic importance of local certifiability [1] was based on the intractability of dealing with the monolithic programs that would result from source-code expansion in large software systems. It is simply a practical reality that if one cannot reason modularly about a large system, one cannot really hope to reason about it at all.

1.2 Modular Proof Systems for Procedure Components

Suppose we wish to develop a formal system for program verification, i.e., proving programs to be correct with respect to an abstract specification. Any such reasoning process must have two logical properties [4, 5], roughly defined as follows:

- *Soundness* — If a program is incorrect if executed, then the reasoning system must be unable to predict it is correct.
- *(Relative) completeness* — If a program is correct if executed, then the reasoning system must be able to predict it is correct.

And of course in order to meet the test of local certifiability, the reasoning process must be modular in the sense explained in the first section.

One such sound and relatively complete modular verification system is Cook's [4]. This system is defined for a very simple, but still interesting, Pascal-like language. The most exciting part of Cook's language is that it has procedures (although no recursive ones) with parameters like those in Ada (later) termed "in" and "in out" mode. But here are some of the simplifications and compiler-checkable restrictions for this language:

- *Type restriction* — All variables are scalars of the same type, e.g., integer. There are neither type constructors such as arrays and pointers, nor user-defined types.
- *Repeated argument restriction* — In any call, all arguments whose corresponding formals have mode "in out" must be distinct variables.
- *Global variable restriction* — In any call, no argument whose corresponding formal has mode "in out" may be visible within the called procedure's body (or within the body of any procedure called directly or indirectly by P) by virtue of being global to it.

Note that Ada and C++ — and similar "real" languages widely used to develop component-built software systems — are similar to Cook's language in many ways, but also differ from it in several respects. They permit recursive procedures; they have many scalar types; they have record, array, and pointer type constructors; they permit user-defined abstract types; and they do not ask the compiler to enforce either the repeated argument or global variable restriction on procedure calls. The question immediately arises: Does the existence of a sound and complete modular proof system for Cook's simple language say *anything* about the feasibility of modular reasoning about Ada or C++ programs?

We have developed modular proof rules for a language that includes modules, user-defined abstract types, and a variety of other more advanced constructs [6]. However, for purposes of this paper we wish to concentrate on the better-known, more classical work of Cook to illustrate our positions.

2 Position

It is apparent that some of the simplifications of Cook's "toy" language might be merely convenient to make the language and proof system smaller and more easily understandable; there would be no fundamental technical problems if these restrictions were relaxed. But others might be technically essential to obtain the results; they are imposed because, without them, the proof system would fail to be sound or relatively complete. As is often the case with such papers, Cook's discussion does not distinguish explicitly between these cases. His audience was mathematicians, not software engineers.

The "meta-position" of this position paper is that it is dangerous to err in either direction in characterizing these language simplifications as merely convenient or technically essential. Some software engineers tend to downplay the value of formal methods (especially in program verification) because they guess that nearly all the simplifications are essential; that the "toy" language is therefore so far removed from "real" ones that the results based on it can't possibly say anything interesting about the practical world. Others tend to guess that nearly all the simplifications are merely convenient; that the results for the "toy" language therefore apply to "real" ones that are similar. The only way to tell for sure (not just to guess) is to understand the proof system and the proof of its soundness and relative completeness; in other words, Cook's entire paper and then some. We have tried to do that, and to interpret two of our observations as specific positions to defend for this workshop:

1. *There can be no sound and relatively complete modular reasoning system for the class of legal programs written in Ada or C++.*

2. *If a language is defined to pass parameters using call-by-swapping [7], not call-by-reference or call-by-value-result, then the repeated argument and global variable restrictions on procedure calls are not necessary to obtain a sound and relatively complete modular reasoning system.*

Space limitations prevent us from defending these claims in any detail in this short position paper. Instead we note the relevance of the above claims to issues of software component reuse. We will, of course, be prepared to argue the validity of these positions at the workshop itself.

The first claim says that the full-fledged Ada and C++ languages permit legal programs (i.e., ones that compile without error) that may not submit to local certification of correctness. We cannot rely on the compiler to "weed out" the offending programs. If we want to be sure of the ability to reason modularly about programs — a prerequisite to successful component-oriented reuse [1] — then we *must* subscribe to a strict personal discipline or programming methodology such as that described by Hollingsworth for Ada [8]. Understanding which language simplifications are technically essential for modular reasoning tells us what guidelines this discipline must contain.

The second claim says that two of the restrictions in Cook's language that are most commonly considered to be technically essential are, in fact, not essential if one adopts a non-traditional parameter passing mechanism.

Ada includes a curiously indirect approach to these restrictions. An execution of a program whose effect depends on whether certain parameters are passed using call-by-reference or call-by-value-result is termed an "erroneous execution." A call violating the repeated argument or global variable restriction almost certainly can distinguish between these mechanisms for some, but not necessarily all, values of its arguments. So defining dependence on the parameter passing mechanism as a source of erroneous execution is, in effect, a roundabout way of telling programmers not to violate the repeated argument and global variable restrictions. Note also that because array entries can be used as though they were variables, it is not always possible for an Ada compiler to detect violations of the repeated argument restriction. But an Ada compiler must accept a program as "legal" even when it is possible to determine at compile time that there is the *potential* for erroneous execution, because it is the execution that is defined to be erroneous, not the program.

On the basis of an admittedly superficial analysis, we conjecture that changing the explanation of parameter passing in Ada, from what is now in the language reference manual to call-by-swapping, would be an "upward-compatible" change in the following sense: (1) A legal Ada program that does not have the potential for erroneous execution under the current language definition would have the same effect if call-by-swapping were used. (2) A legal Ada program that does have the potential for erroneous execution due to dependence on the parameter passing mechanism under the current language definition would always give (modularly) predictable results if call-by-swapping were used. This change therefore could have important expressiveness consequences, especially in certain numerical and linear algebra applications where repeated arguments in procedure calls are both natural and desirable.

3 Comparison

Guaspari, *et al.* [9] note that their verification system for Ada does not handle procedures in a modular fashion, but they do not show why this is impossible without changes to the language definition itself.

Of course many programming language textbooks contain examples to illustrate pathological situations for call-by-reference, call-by-value-result, and other parameter passing mechanisms. But to our knowledge none discusses the consequences for modular reasoning, for software engineering in general, or for component reuse in particular. Nor does any suggest call-by-swapping as a possible alternative mechanism that makes seemingly pathological procedure call behavior predictable from procedure specifications alone, without the need for examining the called procedure's body.

There have been extensions to Cook's original system [4] having to do with procedure calls. For example, Clarke [10] suggests some relaxations of the repeated argument and global variable restrictions, but he does not suggest it is possible to do away with them entirely.

4 References

References

[1] B. Weide and J. Hollingsworth, "Scalability of Reuse Technology to Large Systems Requires Local Certifiability," in *Proceedings 5th Workshop on Software Reuse*, 1992.

[2] M. Griss and W. Tracz, "WISR '92: 5th Workshop on Software Reuse Working Group Reports," *Software Engineering Notes*, vol. 18, pp. 74–85, April 1993.

[3] W. Myers, "Workshop Participants Take the Pulse of Reuse," *IEEE Software*, vol. 10, pp. 116–117, January 1993.

[4] S. Cook, "Soundness and Completeness of an Axiom System for Program Verification," *SIAM Journal of Computing*, vol. 7, pp. 70–90, February 1978.

[5] G. Ernst, J. Menegay, R. Hookway, and W. Ogden, "Semantics of Programming Languages for Modular Verification," Tech. Rep. CES-85-4, Case Western Reserve University, Dept. of Computer Engineering and Science, October 1985.

[6] J. Krone, "The Role of Verification in Software Reusability," tech. rep., Ohio State University, Dept. of Computer and Information Science, August 1988.

[7] D. Harms and B. Weide, "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Transactions on Software Engineering*, vol. 17, pp. 424–435, May 1991.

[8] J. Hollingsworth, "Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada," tech. rep., Ohio State University, Dept. of Computer and Information Science, August 1992.

[9] D. Guaspari, C. Marceau, and W. Polak, "Formal Verification of Ada Programs," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1058–1075, September 1990.

[10] E. Clarke, "Programming Language Constructs for Which It Is Impossible To Obtain Good Hoare Axiom Systems," *Journal of the ACM*, vol. 26, pp. 129–147, January 1979.

5 Biography

Bruce W. Weide is Associate Professor of Computer and Information Science at The Ohio State University in Columbus. He received his B.S.E.E. degree from the University of Toledo and the Ph.D. in Computer Science from Carnegie Mellon University. He has been at Ohio State since 1978. Professor Weide's research interests include various aspects of reusable software components and software engineering in general: software design, formal specification and verification, data structures and algorithms, and programming language issues. He also has published recently in the area of software support for real-time and embedded systems.

Wayne D. Heym received his undergraduate degree from Miami University (Ohio) and his masterUs degree from Cornell University. He has also worked at Kodak Corporation. As a Ph.D. student at Ohio State he has worked on a variety of problems related to software reuse, including formal specification and verification and component testing, and has published in the latter area. His Ph.D. research involves development of a "natural" modular system for verification of assertive programs, including proofs of soundness and relative completeness of the system.

William F. Ogden is Associate Professor of Computer and Information Science at The Ohio State University in Columbus. He is a Ph.D. graduate of Stanford University whose work has resulted in significant advances in theoretical computer science, including the well-known "Ogden's lemma" and a proof of the exponential complexity of the circularity problem for attribute grammars. His most recent research has concentrated on modular program verification and on design and development of reusable software components based on abstract data types.