Design verification and code verification under our proposal are defined in terms of these proofs against a common application model, although additional interpretation of the model may be necessary for code proofs. In some cases, special tools may be needed to conduct code verification, such as information flow analyzers for specific programming languages. Existing techniques appear to be adequate for accomplishing a substantial amount of code proof; it is primarily tools for specific languages that are lacking.

In essence, our proposal is a plea for flexibility in the way we apply current verification technology. Prevailing notions of design verification and code verification, while obviously valid, should not be prescribed as constituting the only permissible methodology. Until such time as someone produces a uniqueness proof for the validity of a given verification methodology, we should remain open to different approaches for applying our admittedly limited capabilities. The problems we face are difficult enough. We should not add to them by acquiescing to false intractabilities that result from our insistence on a particular approach. With a little bit of imagination, our existing knowledge and tools can be stretched to achieve far greater results than are currently perceived possible.

Position Paper: Verkshop III

Beyond Functional Behavior: Combining Methods to Specify Different Classes of Properties of Large Systems

*Jeannette M. Wing**
Computer Science Department
University of Southern California
Los Angeles, California 90089-0782

1. Introduction to a Problem

Most of the past specification and verification technology has concentrated on the functional behavior of a piece of software. Once one begins to explore the possibility of the practical use of such technology for larger and larger pieces of software, one finds that showing the correctness of only the functional behavior of a system is not usually sufficient to satisfy the customer. Verifying other properties, such as reliability, security, performance, and real-time behavior, is in as much of the customer's interest as verifying functional behavior.

Work done on specifying and verifying one of these properties is typically done at best with the assumption that the other properties are satisfied or at worst in complete disregard of them. Little work has addressed the language and model issues of integrating the specification and verification of all these properties. A few exceptions include work done on fault-tolerance [Wensley 78] and more recently, on concurrency (e.g., [Lamport 83]), reliability [Cristian 83], and performance [Zave 84]. Properties such as response time and space efficiency, which both are of critical importance when analyzing the behavior of a large program, have thus far been ignored by most in the specification and verification communities.

We need to study the unaddressed task of combining the specifications of these different classes of properties. Such a combination should consequently yield a specification that more completely describes the behavior of a large program than a traditional specification that describes only the program's input-output behavior. Furthermore, we need to reassess the verification process in light of the proofs of these other properties. Performing the proofs themselves raises some challenging problems. Moreover, these proofs will not necessarily be independent of one another, and thus, the formal meaning of their combination, raises other theoretical and practical problems for verification technology.

In what follows I will dwell on only the specification end of the problem. In the third section, I will make some final remarks and briefly mention some of the corresponding problems at the verification end.

2. Towards a Solution: Combining Methods

Combining methods, languages, and models of specifications is a reasonable approach toward handling the problem of specifying properties of different classes. For example, for concurrency, a definitional method of specifying concurrent properties, e.g., via temporal logic, should blend in well with a definitional method of specifying (sequential) functional behavior, e.g., via algebraic specifications. In the same spirit, CCS and Meta-IV of the Vienna Definition Method would also be a good blend (see [Folkjar and Bjorner 80] for combining CSP and Meta-IV) of operational methods. With some work, it similarly should be possible to combine methods, languages, or models of specifications of reliability, performance, etc., with those of the input-output behavior of programs.

* This work was supported in part by the National Science Foundation under Grant No. ECS-8403905.

What is essential to keep in mind when focusing on large systems (and what makes the problem more difficult) is modularity. We must aim to build a system specification from specifications of its pieces. Ideally, we should be able to derive and add to reliability, performance, etc., requirements of the system from the corresponding requirements stated in the specifications of the system's pieces. That is, let P_i, for $1 \leq i \leq n$, be specifications of properties such as functionality and reliability, and let S_j, for $1 \leq j \leq m$ be specifications of a system's pieces. If we form a specification $S_{b,g}$ from the S_j specifications, then roughly speaking, we want the relationships:

$$S_{b,g} = S_1 + \cdots + S_m$$

where

$$S.P_1 = S_1.P_1 + \cdots + S_m.P_1$$

...

$$S.P_n = S_1.P_n + \cdots + S_m.P_n$$

for some undefined "+" (composition operator) for the S_j and for each P_i. Notice that these "+"'s may be defined in terms of the specifications themselves (their syntax) or what they denote (their semantics), or some combination of the two. It is important to realize that there are a lot of different "+"'s and that they all need formal definitions.

Each of the S_j's, in fact, may not have components for each of the properties of interest. What are these S_j's? Typically, they are specifications of program modules, e.g., procedures and functions, and of abstract data types. For concurrent systems, they may also be specifications of processes; for distributed systems, of nodes and networks. Each of these different kinds of specifications[1] will include requirements that make sense for its kind and may not for other kinds.

To be a little more concrete, let's look at a possible template for a piece of a specification that includes specifications of many classes of properties. Figure 1 shows what one of the S_j's might look like for a procedure or function. The template has space for four different P_i's (functionality, reliability, security, and performance) to be filled in.

Module <module name>
Uses <list of specifications of data types of objects that module manipulates>

Functionality
<Input-output specification>
Reliability
<E.g., Probability of failure>
Security
<E.g., Security class>
Performance
<Worst-case behavior, average-case behavior, upper bounds, etc.>

Figure 1: A Template for the Specification of a Piece of a System

[1] I will use the terms "kinds of specifications" for the different S_j's and "classes of properties" for the different P_i's.

The other kinds of specifications would have different templates. For example, a process specification might have an additional clause for **Concurrency** properties; a node specification might have additional clauses for **Shared Resources** and **Processes**; a network specification might have an additional clause for **Configurations** (strictly speaking, a structural, not behavioral, property) to include how nodes may be networked together. Finally, a template for a system specification might end up looking like that in Figure 2.

System
Uses <list of module, data type, process, node, and network specifications>

Configurations
<Structural information about how nodes and networks may be networked together>
Functionality
<Constraints additional to those of the Used specs>
Reliability
<Additional constraints>
Security
<Additional constraints>
Performance
<Additional constraints>

Figure 2: A Template for a System Specification

3. Final Remarks

Even with reasonable ways of specifying different classes of behavioral properties of systems, the problem of verifying them remains. Ways of writing and combining specifications do not at all necessarily reflect ways of verifying them.

As for the specification process, for verification, it is important to distinguish between proving each piece of the specification (e.g., the S_j's) and proving each different class of property (e.g., the P_i's) all of which combined make up the proof of the system with respect to the entire specification (e.g., $S_{b,g}$). In both cases, but for possibly different reasons, the way a specification is modularly built may not lead to a modular verification process. The lines along which a specification is broken may not be the same lines along which the verification process would be performed. Having a modular specification may not even make the verification process easier or more efficient, let alone modular.

More generally, we must grapple with the problem that methods of writing specifications may lead to specifications that serve well for system design, but not for system verification. This is not surprising since specifications are meant to serve many purposes throughout the software life cycle and when the focus is at one phase, e.g., system design, techniques are tailored for that phase and not the others. Specifications that are easy for humans to understand are rarely of the same form as those easy for machines to manipulate, and vice versa. Forging ahead into the territory of modularly constructed specifications of large systems further magnifies this problem.

In summary, I believe that in order to demonstrate the practicality of specification and verification, we must be able to specify and verify large systems, and thus, to show how to specify and verify many classes of properties besides functional behavior. We must be able to do both specification and verification in a piecemeal fashion in order to come to gripe with the inherent problem of scale. To do both requires some challenging work in defining the pieces: what class they are in, how to write them, what their formal models are, how to put them together.

References

[Cristian 84] Cristian, F., "A Rigorous Approach to Fault-Tolerant System Development," IBM Res. Rep. RJ3754, Jan. 1983, will appear in *IEEE-TSE*.

[Folkjar and Bjorner 80] Folkjar, P., and D. Bjorner, "A Formal Method of a Generalized CSP-like Language," Proceedings of IFIP 1980, Tokyo, North-Holland Publishing, pp. 95-99.

[Lamport 83] Lamport, L., "Specifying Concurrent Program Modules," *ACM TOPLAS*, 5:(2), April 1983, pp. 190-222.

[Wensley 78] Wensley, J.H., et al, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," Proceedings of the IEEE, Vol. 66, No. 10, October 78, pp. 1240-1255.

[Zave 84] Zave, P., "Specification and Analysis of Worst-Case Delay Requirements," submitted to *IEEE-TSE*.

Ada Verification Using Existing Tools

A Position Paper for the Verkshop

John McHugh - Research Triangle Institute
Karl Nyberg - Verdix Corporation

1. Introduction

DoD directive 5000.31 [DoD] requires that new mission critical computer programs written for the department of defense be written in Ada[1] [Ada]. The statutory definition of mission critical (10 USC 2315) includes security applications specifically. Computer security has been one of the principle driving forces for applied verification work in recent years. These factors lead us to one of two conclusions: 1) The time is rapidly approaching when it will be necessary to apply verification techniques to programs written in Ada; or 2) DoD 5000.31 will have to be modified to exclude secure systems. While there exists a well known antipathy towards Ada within parts of both the verification and the computer security communities, it is unlikely that the DoD policy towards Ada will undergo substantial change in the near future. If this is the case, it will be necessary to develop an Ada verification capability in the near future.

There are several ways in which such a capability could be developed. A first option would be to start from scratch, using any of the formal models of program specification and verification and build a system specifically designed to verify Ada programs. A second option is to ignore the Ada specific aspects of the problem entirely. Under the current certification criteria of the DoDCSC, it is not necessary to deal with the implementation language for a system in a formal manner, so it could be argued that current systems are just as suitable (or unsuitable) for Ada as for any other language. In this case, it is only necessary to provide a convincing argument for the conformance of the Ada implementation code to the verified formal top level specification of the system in question. Finally, it is possible to adapt an existing verification system to deal with Ada.

The first approach is possible, but would take an excessive amount of time and resources. Current verification systems represent investments of ten or more man years each, expended over periods of five to ten years. The second approach is representative of the practice followed for the Honeywell SCOMP, a product currently approaching A1 certification by the DoDCSC. It appears that the requirement for a convincing argument concerning the equivalence of the FTLS and the implementation resulted in an extremely complex and concrete FTLS and greatly increased the verification effort. Being able to verify an Ada based FTLS for an Ada based implementation should obviate these difficulties. Additionally, there is substantial interest in systems which go beyond the A1 criteria by requiring code verification for which second approach would not be viable. The third approach offers a chance to capture much of the investment in

[1] Ada is a registered trademark of the Ada Joint Project Office.

a current verification system while gaining experience with the verification of Ada. We argue for such an approach, based on the Gypsy [Good78] system, suggesting that it will lead to a prototype code verification system for Ada with minimum (although not insubstantial by any means) effort. Taking advantage of the Ada packaging mechanism, we feel that verified packages can function within a larger Ada environment, making possible the implementation of security kernels and the like.

The remainder of the paper discusses some of the problems associated with the verification of Ada, suggests ways in which these problems might be addressed, and indicates ways in which the Gypsy system could be combined with the front end of an Ada compiler and transformed into a prototype system for the verification of Ada.

2. Trouble spots in Ada

Although one of the early design objectives for Ada (in the days when it was still known as DoD-1) was to facilitate proofs of program properties, the committee nature of the requirements process resulted in a language which was required to carry a certain amount of the baggage of 1960s style programming languages. Among the potentially most troublesome of these are the presence of arbitrary control flow constructs i.e. the "go to" statement, and unrestricted access to global variables which, in addition to complicating proofs about sequential programs, render concurrent programs intractable under many circumstances. Other features of the language include the possibility of side effects from function invocations, exceptions during expression evaluation, and the lack of an explicit evaluation order for the operators of an expression. These factors, combined with the lack of a formal definition for the semantics of the language, have lead some workers to despair of verifying any aspect of the language. Indeed, it has been noted that given the proper Ada context, it may be impossible to prove anything about the value of X after the execution of so simple a statement as

```
X := 1;
```

We maintain that the situation is not quite as grim as indicated above. Just because a language contains a particular feature does not mean that all programs written in the language will contain that feature. The adverse interaction among features of the language, does not mean that all of them must be discarded, or that all occurrences of a feature in a given program are intractable. Although the word "subset" is an anathema to the Ada world, we feel that a useful set of Ada constructs and programming practices can be defined in such a way that realistic and functional programs can written and verified using them. Although the task is substantially more difficult, because of the extra complexity of the language, we feel that a theory of verifiable Ada can be developed in much the same way as Boyer and Moore developed their FORTRAN [Boyer80] theory. Platek [Odyssey84] and his colleagues at Odyssey Associates have recently defined an initial subset of Ada which they feel is suitable for verification. One feature which they rule out is the exception mechanism. We feel that the Ada exception mechanism is sufficiently like the Gypsy mechanism so that its verification is tractable, and we propose to include exceptions in our system.

Ada as currently defined has no specification mechanism. While it is possible to use an external specification mechanism, i.e. one in which the program and specification are joined only during the verification process, we are more comfortable with an internal mechanism, similar to that used in Gypsy. At the same time, we would like our verifiable code to be acceptable to a variety of Ada translators. An extension of Luckham's Anna notation [Luckham84] to accommodate exception returns from routines appears to be the most promising mechanism available at the present time, although a specification language using the Ada PRAGMA construct cannot be ruled out.

3. A hybrid system

We propose to base our prototype Ada verification system on a combination composed of an existing Ada compiler and an existing verification system. The Ada compiler is the one developed and recently validated by the Verdix corporation of McLean, Virginia, while the verification system is the Gypsy Verification Environment, developed at the University of Texas. There are several reasons for the choice of such a hybrid system. Ada is a large language with a complex syntax and semantics. Using an existing front end from a validated compiler eliminates much of the effort required to implement a front end for the verification system. It also provides a direct method for providing executable versions of the verified programs, as well as facilitating systems which contain mixtures of verified and unverified programs. The use of a modified version of the GVE as a back end for the Ada verification system offers similar advantages. We feel that the initial set of Ada constructs which can be verified will be roughly equivalent in power and flavor to the Gypsy language. Previous efforts to model Ada constructs in Gypsy [Akers83], and vice versa provide evidence for this assumption. Although Ada type rules are "stronger" than those for Gypsy, it is possible to write Gypsy as though it were typed like Ada. The Gypsy exception mechanism, though somewhat more tractable than the Ada exception mechanism is suitable for modeling Ada. Most of the Ada operators are already present in Gypsy.

The proposed hybrid consists of three primary components, the Ada front end, the intermediate form translator, and the verification back end. Each of these are described briefly in the sections which follow.

4. The Ada front end

As noted above, the front end of the proposed system is based on the parser and semantic checker of an existing, validated, Ada compiler. The parser and semantic checker will require some modifications to accept Ada with embedded specifications.