N89-16285 167030

10P.

FORMERLY 34

FORMAL VERIFICATION AND TESTING: AN INTEGRATED APPROACH TO VALIDATING ADA PROGRAMS

Norman H. Cohen

SofTech, Inc.
One Sentry Parkway, Suite 6000
Blue Bell, Pennsylvania 19422-2310

NCohen@Ada20

Formal verification is the use of mathematical proof to confirm that a program will behave as specified when it is executed. Formal verification can produce a much higher level of confidence in a program than testing. Nonetheless, formal verification requires large amounts of skill, human time, and computer time, so it would be impractical to verify formally an entire Ada program for a typical embedded computer application.

We propose an integrated set of tools called a validation environment to support the validation of Ada programs by a combination of methods. The validation environment exploits the Ada distinction between module interfaces and module implementations to validate large Ada programs module by module. The proposed validation environment is called the Modular Ada Validation Environment, or MAVEN. MAVEN does not yet exist, nor have efforts begun to construct it. Rather, MAVEN is our vision of the context in which Ada formal verification should be applied. A more complete discussion of MAVEN can be found in [1].

Our vision of MAVEN is based on several requirements that we have identified for the validation of Ada programs. These requirements are based on the recognition that Ada programs for mission-critical applications are large, that skilled software engineers are in short supply, that the construction of a verifier is an expensive undertaking, and that the use of a verifier may be time consuming. Our requirements are as follows:

1. Formal proofs should not be based on the behavior of a particular implementation.

2. It should be possible to validate a large program module by module.

3. For typical mission-critical applications, verification will have to be integrated with other forms of validation.

4. It should be easy to request the proof of certain critical properties which, while they do not imply correctness of a module, significantly raise our confidence in its reliability.

See [2] for a more complete discussion of these requirements.

When software engineers use the term "validation and verification," they usually do not have formal verification in mind. To avoid confusion, this paper uses the terms validation and verification in two distinct and precise senses:

> Verification is the use of formal proof, checked by machine, to establish properties of a program's run-time behavior.
>
> Validation is the process of increasing one's confidence in the reliability of a program. Formal proof is one of many methods for validating software.

Confusion may also arise from our use of the term environment. Ada Programming Support Environments (APSE's) already exist, and have functions that overlap those we propose for a validation environment. We do not envision MAVEN as a full APSE or as a tool set independent of an APSE. Rather, we view MAVEN as an integrated tool set embedded within an APSE. It can be thought of as a "subenvironment." Many APSE tools, including an Ada compiler, may be used both for validation and for other purposes.

1 Integration of Multiple Validation Methods

One reason for validating programs module-by-module is so that different modules can be validated in different ways. There are many software unit validation methods, all of which have been used successfully in the past. These include:

- formal proof generated with machine assistance and checked by machine
- informal proof carried out by hand
- code walkthroughs
- unit testing
- acceptance of a software component as trustworthy, based on experience using the same component in a previous system

It is not necessary for a project to choose one of these validation methods for use throughout a program. Given the right framework, different methods can be combined in an effective symbiotic relationship to ensure the quality of a system.

While formal verification is the most effective means of ensuring consistency between a program and its specifications, it has limitations. These include the problem of validating that the specifications themselves specify what the customer wants; and the cost -- in both machine time and the time of skilled personnel -- of developing and checking the proof. The manufacture of software, like any manufacturing process, entails a tradeoff between cost and level of quality assurance. In some programs there are modules for which any form of validation less powerful than formal proof would be socially irresponsible. Sometimes the same program also contains many modules for which formal proof would be a wasteful misallocation of resources.

Furthermore, there may be some modules that cannot be verified because they use features of the language for which there are no proof rules. Features may be excluded from the "verifiable subset" of Ada even if there are occasional legitimate uses for such features. Such legitimate uses can be isolated in modules that are validated by some means other than formal proof. In particular, low-level features of the Ada language are inherently machine dependent and thus not characterized by proof rules. Low-level features can be isolated in interface modules, allowing the rest of a system to be validated by formal proof.

Many factors combine to determine the most appropriate form of validation for a module. The cost of formal proof must be compared with the possible impact of an error in the module. Low-level, target-dependent interface modules might best be validated by informal proof. For certain hard-to-specify modules, for e ample a graphics display builder whose desired output is specified pictorially, testing might be not only the cheapest, but also the most reliable form of validation. For modules that are not particularly critical, and for which test drivers would be difficult to write, code walkthroughs might be most appropriate. Software might simply be trusted (until integration testing) if it has been extracted from a working system in which it has functioned reliably.

To ensure complete coverage, different forms of validation cannot be combined haphazardly. There must be a unifying discipline. One of the functions envisioned for MAVEN is to provide such a discipline.

2 Validation Libraries

The Ada language was designed to facilitate the construction of huge programs. A pervasive theme in the design of the language is the division of a program into units that can be understood individually yet checked for consistency with each other. If this theme is extended from unit compilation to unit validation, one unit of a program can be changed and revalidated without revalidating the rest of the program. This is especially important during program maintenance.

Module-by-module validation of a large program can be achieved in the same way as module-by-module compilation. Compilation of an Ada program unit consists not only of code generation, but also consistency checking. A unit's syntactic specification is compiled before either the unit's body or any external uses of the unit. This compilation puts information about the syntactic specification into a program library. Later, when either the unit's body or an external use of the unit is compiled, this information is retrieved from the program library and used for compile-time consistency checks.

The consistency checks that occur during compilation are limited to the information found in a unit's syntactic specification, such as the number, types, and modes of subprogram parameters. Except for this limitation, however, they are analogous to the checks that occur during unit validation. Just as a unit has a syntactic specification that is checked during compilation, it has a semantic specification that is checked during validation. Just as syntactic specifications are recorded in a program library, semantic specifications are recorded in a MAVEN validation library.

Semantic specifications are textually embedded in syntactic specifications in the form of structured comments like those found in Anna [3]. This unifies the notions of syntactic and semantic specifications. When MAVEN is directed to compile a specification, it invokes the Ada compiler to place the syntactic specification in the program library. If no compile-time errors are found, the semantic specification is then extracted from the structured comments and added to the validation library.

2.1 Validation Order

To facilitate compile-time consistency checks, the Ada language restricts the order in which units may be compiled. MAVEN imposes analogous restrictions on the order of validation. Specifically, a module's semantic specification must be entered into the validation library before the implementation or any use of the module is validated. Then the implementation and each use of the module may be validated in any order. Validation of the implementation establishes that the body fulfills the semantic specification. Validation of a use of the module involves assuming, while validating the using module, that the semantic specification is correctly implemented. This assumption is permitted as soon as the semantic specification is entered into the validation library, even before the body has been demonstrated to fulfill the semantic specification. (This is analogous to the compilation of a subprogram call after the subprogram declaration has been compiled but before the subprogram body has been compiled.) It implies that validation of one unit can proceed considering only the specifications of the units it invokes, without considering their bodies. This is the essence of module-by-module validation.

Some program units may be validated by fiat. That is, after a code walkthrough or simply on the basis of trust, a unit may simply be decreed to be "validated." This still must be done explicitly, by a request to MAVEN, and the usual validation order rules must be obeyed. In particular, a unit may not be decreed to be validated before the specifications it is meant to fulfill have been entered into the program library.

2.2 Revalidation Order

Just as the Ada language restricts compilation order, it imposes recompilation requirements to ensure that consistency checks have always been performed on the latest version of a program. If a syntactic specification is recompiled, all consistency checks based on the old syntactic specification are rendered invalid. The corresponding body and all uses of the unit must then be recompiled so that the consistency checks may be repeated with respect to the new syntactic specification.

MAVEN imposes analogous revalidation requirements. If a module's semantic specification is changed, both the implementation and all uses of the module must be revalidated if they have already been validated. This is relevant during program development and program maintenance.

In program development, failure to validate a body may mean either that the body does not correctly implement the corresponding logical specification or that the logical specification itself is incomplete. In the first case, the body can be corrected and validated. In the second case, the logical specification must be modified and all other units using that logical specification must be revalidated. This may require still further modifications and revalidations.

In program maintenance, revalidation requirements indicate which parts of a large program are potentially affected by a change. This can reduce or eliminate the "ripple effect" typically resulting from a change to a working program. All possible implications of the change will be flushed out by the ensuing round of revalidations, assuming the revalidation is sufficiently thorough. (If the revalidation is by unit testing, this process amounts to regression testing. Rather than blindly repeating all tests, however, we use validation dependency relationships to identify the tests that might possibly have been affected by the change.)

A unit validated by fiat is subject to the same revalidation requirements as any other unit, even if revalidation consists only of reissuing the decree by which the unit was originally validated. This encourages software engineers to consider whether the original decree is still valid given the new specifications. For example, it may be discovered that an off-the-shelf package originally thought to be applicable to the current application is inappropriate given the revised specifications.

2.3 Other Information in the Validation Library

A validation library contains information besides the semantic specifications of program units. A validation plan can be entered into the library in advance, stipulating how a unit will be validated once it is written. The validation library also records which units have been validated, and according to which validation plans.

Each module may have its own validation plan. The plan specifies the validation method applied to the unit (testing or formal proof, for example) and the details of the validation criteria (which files contain the test driver or test data, algorithms for evaluating test results, or which properties are to be proven, for example). A validation plan may specify several rounds of validation, all of which must succeed for the unit to be considered validated. For example, a plan may call for testing to find and eliminate obvious errors, followed by formal proof to ensure the absence of more subtle errors. No one round of validation need provide complete coverage of the unit's semantic specification. Some parts of a unit's semantic specification may be proven valid, some validated by testing, and some simply assumed to be valid, for example.

Besides allowing MAVEN to enforce validation and revalidation order dependencies, the data kept in the validation library allows MAVEN tools to generate reports on the progress of system validation to date. The reports indicate which units have been validated and how rigorously. During development, validation of units can be tracked and compared with schedules. When an error arises, information about the validation methods applied to each unit and the properties validated for each unit can help pinpoint suspect modules. The revalidation implications of a proposed change can quickly be estimated.

3 Other Components of a Validation Environment

A verifier is only one of the tools that a validation environment should provide. We have already mentioned the need for a validation library. This implies the need for library management tools, including the report-generation tools discussed above. Other tools can assist in the writing of specifications, the retrieval of reusable software from a large catalogue, and the execution and analysis of tests.

Formal specifications are at the heart of MAVEN, but they are difficult for the typical software engineer to write. Therefore MAVEN must supply tools to help the software engineer express his intent. These tools are collectively called the specification-writer's assistant. One component of the specification-writer's assistant is a knowledge-based tool that will construct formal specifications based on a dialogue with the user. The specification-writer's assistant also includes an interpreter for a logic programming language, similar to PROLOG but providing the higher level of data abstraction found in the Ada language. This tool can be used for rapid prototyping, to test specifications as they are written.

The Ada language is meant to encourage the reuse of general-purpose software components. This approach can only have a significant impact on software development costs if there is a large corpus of general-purpose software available for reuse; but such a large corpus presents an awesome information-retrieval problem. While software retrieval is not usually thought of as a validation problem, Platek [4] has noted that formal specifications and verification can form the basis of a retrieval tool. In addition to a validation library, MAVEN might include a catalogue of general-purpose, reusable software components, all of which have been formally specified. Given the semantic specification of a module required in the design, a MAVEN tool would search the catalogue for reusable components that can be proven to have compatible specifications.

Because testing is the most frequently used validation method, MAVEN contains tools specifically supporting testing. These include tools to generate subprogram stubs, tools to generate test drivers, tools to generate test data, and tools to analyze test results. All of these tools can base their outputs at least in part on the semantic specifications found in the validation library. For embedded applications, there should be software simulation tools and tools providing interfaces with hardware mockups. A related tool would administer tests automatically, based on the validation plans found in the validation library. Such a tool could also revalidate those units validated entirely by testing, whenever revalidation is required. In essence, this automates regression testing.

4 MAVEN and the Software Life Cycle

MAVEN tools are primarily concerned with unit validation. This can lead to the impression that the benefits of MAVEN are primarily reaped during the unit validation stage of the life cycle. In fact, the use of MAVEN imposes a discipline on software development and provides benefits throughout the software life cycle. This section walks through a typical waterfall model of the life cycle and describes the impact of MAVEN on each stage.

4.1 Requirements Analysis

The specification-writer's assistant supports the formal expression of requirements. Requirements can be entered into a new MAVEN validation library as the semantic specifications of the main program and of tasks declared in library packages. These formally stated requirements can be checked for consistency using a verifier. They may later become the basis for design verification and code verification. An integration-testing plan may be derived from the formal requirements and stored in the validation library until software integration time.

4.2 Design

During high-level design, the modular decomposition of a system is determined and the specifications of each module are written. Algorithms for top-level modules may also be written. MAVEN can play four roles at this stage -- design documentation, recording of unit validation plans, software-component retrieval, and design verification.

Design documentation consists of entering the semantic specifications for each design module into the validation library. The specification-writer's assistant again comes in handy here. The semantic specifications entered at this stage become the basis for later verification of module bodies. The appropriate time to formulate unit validation plans is just after unit semantic specifications have been identified. One of the responsibilities of an Ada designer is to look for existing software that can be incorporated in a design. As noted earlier, formal specifications might provide the basis for software automated software retrieval. The top-level algorithms of a high-level design can expressed in executable Ada code verifiable in the same way as lower level modules. Using only the specifications of the main system modules (the main program and tasks declared in library packages), it can be proven that the top-level algorithms correctly implement the system specifications.

4.3 Unit Development

There is not a clear dividing line between design validation and unit validation. The same techniques applied to the top-level modules during design validation are applied to lower-level modules during unit validation. The unit validation plan placed in the validation library during system design is retrieved and applied. A round of validation is repeated until it is successful, and then the next round specified in the validation plan is begun. The validation plan is restarted from the first round any time a change is made to the unit, its semantic specification, or the semantic specifications of the modules that the unit invokes.

Validation can uncover implicit assumptions that underlie the correct functioning of a module, especially when validation is by formal verification. Such assumptions must be added to a module's semantic specifications if the module is to be verified. Thus the validation process contributes to the development of complete and up-to-date specifications.

4.4 Integration Testing

The main impact of MAVEN on integration testing will be a drastic reduction in integration problems. The Ada compiler will already have checked all units for syntactic consistency with each other. MAVEN will already have checked all units for consistency with their own semantic specifications and the semantic specifications of the modules they invoke. The few integration problems that remain will arise from incomplete module specifications (for example, specifications that address functional requirements but not performance requirements) and insufficiently rigorous unit validation (for example, use of code walkthroughs as the sole means of validation or the use of tests that do not provide adequate coverage).

4.5 Maintenance

MAVEN will reduce the costs and risks of program maintenance. Both the data MAVEN collects during program development and the discipline MAVEN imposes on program modification will help confine the "ripple effect" of a change. MAVEN will also keep documentation up to date after changes have been made.

The most frequent problem associated with program maintenance is a change that violates an implicit assumption upon which a different part of the program depends. This problem is less likely to arise when using MAVEN for two reasons. First, the validation process applied during program development has served to make implicit assumptions explicit. The documentation will warn the maintenance programmer right from the start that certain changes must be disallowed unless further changes are made in other modules. Second, if the semantic specification of a module is changed, MAVEN will enforce the revalidation of all modules that may be affected by the change. The revalidation dependencies alone clarify the potential impact of a contemplated change. The actual revalidation, which may follow the original unit validation plan created during the initial design, leads the maintenance programmer to discover which potential impacts are truly significant, to revise the affected modules, and to validate the revisions. If the revised modules can themselves affect other modules, revalidation of these other modules will also be required. If sufficiently rigorous, revalidation anticipates and averts all possible ripple effects.

MAVEN keeps documentation current during program maintenance in the same way that it does so during initial development. Every time a unit's semantic specification changes, MAVEN records the fact. This makes the next round of maintenance easier.

5 Conclusions

We have described our vision of a Modular Ada Validation Environment, MAVEN, to propose a context in which formal verification can fit into the industrial development of Ada software. While proof of correctness is unquestionably the most rigorous and effective form of validation, there are contexts in which it is inappropriate. Nonetheless, formal proof can be effectively combined with other validation methods to raise confidence in a program's reliability.

MAVEN offers software engineers a continuum of more and less rigorous validation methods. This continuum makes a wider variety of validation methods available to a larger group and applicable to a greater number of modules. MAVEN provides a unifying framework in which different validation methods may be applied to the same program. By exposing software engineers to more rigorous methods than those they may be familiar with, MAVEN encourages learning and promotes wider use of formal methods in the situations where they are appropriate.

MAVEN includes components that are at and beyond the state of the art. We do not propose that construction of MAVEN in its entirety should start today. Rather, MAVEN can serve as framework for the specification, design, and construction of individual tools, including a verifier. If such tools are viewed as eventual MAVEN components and if the MAVEN philosophy is kept in mind when the tools are specified, then MAVEN can be assembled over a number of years from independently developed components.

REFERENCES

1. Cohen, Norman H. MAVEN: The modular Ada verification environment. Proceedings, 3rd IDA Workshop on Ada Verification, Research Triangle Park, North Carolina, May 1986

2. Cohen, Norman H. The SofTech Ada Verification Project. AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, Long Beach, California, October 1985, 399-407

3. Luckham, David C., von Henke, Friedrich W., Krieg-Brueckner, Bernd, and Owe, Olaf. Anna, A Language for Annotating Ada Programs: Preliminary Reference Manual. Technical Report 84-261, Stanford Computer Systems Laboratory, July 1984

4. Platek, Richard. Formal specification. Proceedings of the First IDA Workshop on Formal Specification and Verification of Ada, Alexandria, Virginia, March 1985, paper C