VERIFYING PERFORMANCE REQUIREMENTS

By
Dr. Joseph Cross
Sperry Corporation
St. Paul, MN
(612) 456-7316

INTRODUCTION

The thesis presented in this paper is that today, it is in general impossible to verify that the performance requirements on a software program will be met. An approach to a partial solution to this problem is presented.

The next section of this paper, Problem Definition, defines the problem to be addressed, and defines related terms as they are used below.

The following section, Obstacles to Verifying Performance Requirements, presents the reasons why performance requirements are, today, difficult to verify.

The section on Methods for Verifying Performance Requirements briefly presents methods in use today, and proposes an alternative approach to overcome some of the remaining difficulties.

PROBLEM DEFINITION

A "performance requirement" is a requirement on the speed of a function performed by software. Much of the following applies equally well to requirements on the amount of memory used by a software function. An example of a performance requirement is "The interval between updates to each track shall be on the average at most two seconds, and in no case longer than five seconds." Note that while performance requirements are, at the user level, generally stated in elapsed time, these requirements may be recast at lower levels of design into units of processor utilization.

"Verifying" a specific requirement on a specific software development work product refers to determining whether that requirement is fulfilled by that work product. The requirements on the work products of each phase of software development are results of the preceding phase, except for the system requirements, which are input to the entire software development process. A work product WP is said to satisfy a requirement R if any system produced according to the requirements set forth in WP

(and its sibling work products, if any) will meet the requirement R. Verifying a software development work product in its entirety also entails checking its completeness, consistency, feasibility, and testability [1].

For example, to verify that a detailed design satisfies a requirement, such as the example requirement above, is to determine whether any system produced in accord with that detailed design could fail to exhibit the required behavior. Moreover, verifying the entire detailed design requires determining whether there is at least one system that can be built in accord with that detailed design.

Of course, what work products are produced and what are the phases of software development depend on the approach to software development in use. In the conventional approach, the phases are requirements analysis, design (often subdivided into high-level design and detailed design), and implementation; the work products of which are a requirements specification, a design document (or documents), and code, respectively. In the operational approach to software development, the first phase produces a prototype/executable specification, which is intended to satisfy/define all requirements except performance requirements; then a second phase transforms that prototype/executable specification into a program with the same behavior except that the performance requirements are met [2].

In order to minimize the dependence of the following discussion on the approach to software development in use, it will be assumed below that the work product on which performance requirements are to be verified is a body of compiled but untested Ada (tm) code. (Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.) This body of code could represent a detailed design in the conventional approach, or an intermediate step in the transformation of a prototype/executable specification in the operational approach.

In order to make the issues involved in the verification of performance requirements as simple as possible, it will be assumed that a target machine is given and fixed throughout the discussion. Here "target machine" refers to the virtual machine on which compiled code is to run: one or more processors, memories, communication channels, together with run-time support software such as operating systems. This target machine is assumed to be the target of a valid implementation of the Ada language.

Note that while the assumption of a single, known, target machine is reasonable in the Space Station environment, it is not reasonable in other environments in which the target machines that will execute the software may be unknown. We are fortunate in this regard.

The term "mapping" will be used to refer to the association between design-level objects and run-time objects. For example, a subprogram may be mapped onto a segment of memory-resident machine code, or it may be mapped into many similar segments of

machine code (as it would be if it were inlined), or it may be mapped into nothing (as may be the case for type conversion functions). As another example, a data object may be mapped into a location in the main memory of one computer, into a register of one computer, or into several locations in several computers (as would be the case if redundant data were being maintained).

The function of mapping a program is generally distributed among the compiler, linker, loader, and run-time system.

Note that one of the goals of the Ada language design was to include in the source code all details of the program that define its semantics, except for performance issues. That is, by examination of only the source code of an Ada program, without considering other information such as linker directives, it is possible to determine (within limits) its behavior as an input-output process; however, it is not possible to determine its timing. For this reason, it is necessary to use additional information, above and beyond the source code, to verify performance requirements.

OBSTACLES TO VERIFYING PERFORMANCE REQUIREMENTS

This section describes several reasons why verification of performance requirements is not a straightforward task, even given a design that has been carried to the level of compiled Ada code, and a well-defined target machine.

UNSPECIFIED MAPPING ONTO THE TARGET

Perhaps the major obstacle to verifying performance requirements on a design presented as Ada code is that lack of information concerning the mapping of the program onto the target machine. It is only in the mapping information that performance-critical issues such as the following are dealt with:

* Optimizations. These include low-level optimizations such as dead code detection and constraint check elimination, and high-level optimizations such as subprogram inlining and monitor task optimizations.

* Target resource allocation. This includes the assignment of tasks to processors (whether the assignment is static or dynamic), the allocation of data to memory (registers or main memory, resident or non-resident, and arrangement into memory banks or pages), and backup and casualty configurations.

* Implementation dependencies. These include all the implementation dependencies allowed by the Ada language definition, such as the number of task priorities, task scheduling algorithm within a priority, and interrupt handling methods.

As an example of the importance of these issues, note that it is possible to construct an Ada program that will deadlock under one legal task scheduling algorithm, but not under another legal task scheduling algorithm.

Note that a large amount of the optimization and target resource allocation data can change as a result of an apparently small change in the design. For example, the declaration of a small data object can cause the allocation by the compiler of static data to memory banks to be significantly revised, with potentially important changes to timing. This effect is particularly pronounced if a globally optimizing compiler is used.

NON-CATEGORICAL SPECIFICATIONS

A "categorical" specification is one which defines only one target system. Of course, design specifications are generally intended to be non-categorical, that is, to permit substantial freedom in their implementation.

The problem of non-categorical specifications is that if too much freedom of implementation remains, there can be a combinatorial explosion in the number of cases requiring examination in order to verify a requirement. For example, consider a target machine that consists of 3 dissimilar processors connected by communication channels. If the program contains 12 tasks, and if the design does not constrain the allocation of tasks to processors, then there are 12 to the third power (1728) configurations, each of which requires verification. Each choice left open by the design potentially multiplies the number of configurations that must be dealt with in verification.

NON-INVERTIBLE DATA DEPENDENCIES

The processing time for some operations depends on the input conditions to those operations (i.e., input data and retained data). For example, the time required by a track processing operation may depend on the number of currently live tracks. For a given operation, let the function that maps input conditions to processing time of that operation be called its data dependency function.

Data dependency functions are often invertible, at least in the rough sense that the set of input conditions that result in processing times less than some limit can be determined. For example, it might be determined that the time necessary to search a track file will be less the 25 milliseconds if there are no more the 100 live tracks to be searched. This sort of inversion of the data dependency function is often sufficient to verify whether the operation meets its performance requirements.

Unfortunately, data dependency functions are found in practice that are not invertible. That is, there are operations for which the processing time depends on the input conditions, but the

dependency is too complex to invert. Phrased otherwise, it is impossible in practice to define the set of input conditions on which the operation will complete within its prescribed time.

Examples of such non-invertible data dependencies can be found in combinatorial algorithms, and in artificial intelligence paradigms. Specifically, consider a backtracking algorithm -- depth first search for an optimum value using bounding functions. It may happen that a long series of nodes will be generated and expanded before it is discovered that this series does not lead to an optimum, and must be discarded (the "garden path" phenomenon). It is not in general possible to give a simple condition defining those sets of input data that give rise to this phenomenon. In such cases, it is impossible to discriminate input conditions for which processing will be fast from input conditions for which processing will be slow.

NON-DETERMINISTIC BEHAVIOR

Non-deterministic behavior of a program is behavior that cannot be predicted from the input conditions. Non-determinism can arise from the hardware level, as when two processors race for access to a memory word, from the run-time software level, as when the operating system takes varying times to respond to a service request due to the varying activity of peripherals or to the varying activity of other programs under its purview, and from the software level, as from the Ada select statement and Ada arithmetic, which are defined as (potentially) non-deterministic.

The property of being non-deterministic differs from being non-categorical in that non-determinism may be a property of the behavior of a single system, whereas only a specification can be non-categorical. The property of being non-deterministic differs from having a non-invertible data dependency function in that the data dependency function of a non-deterministic process can only be defined statistically, and that function may or may not be invertible.

One example will suffice to demonstrate the difficulties presented by non-determinism to the verification process. Consider a program that is deterministic except that the select statement is implemented non-deterministically. That is, when several rendezvous are possible, the choice of which to accept is made at random. The state space of such a program branches each time a select with two or more open accept branches is executed. Therefore the number of distinct possible program behaviors can grow rapidly with time, and it must be verified that all these behaviors meet the requirements.

ADAPTIVE BEHAVIOR

Adaptive behavior refers to the aspects of a program's behavior that change relatively slowly over time, for the purpose of improving its performance. Examples of adaptive behaviors are load balancing functions in distributed systems, and programs that learn from experience.

Adaptive behavior can be implemented in a straightforward manner, as by changing a vector of locations, and adaptive behavior can be implemented by highly sophisticated means, as in some learning programs that, in effect, modify the code that performs some of their functions.

If the set of possible behaviors of an adaptive program is reasonably small, then adaptation causes no great problems for verification: each of the possible behaviors must be verified to satisfy the requirements. If, on the other hand, the set of possible behaviors is large, then verification may become difficult or impossible.

METHODS FOR VERIFYING PERFORMANCE REQUIREMENTS

Substantial work has been done in the area of dealing with performance requirements. SREM [3] is a method of expressing requirements, including performance requirements. SREM also provides a means to simulate the behavior of the specified system. Unfortunately for our present purposes, the SREM methodology is not well suited to producing Ada programs.

The Model system [4] generates programs (in PL/1) of a restricted form from a specification expressed in an ad hoc language. The system then estimates the performance of the resulting system, using data generated as a by-product of the program generation process together with inputs from the user on the times of the target machine for "input, output, arithmetic, comparison, and function operations."

Several methods support performance estimation based on queueing theory. Examples are PAISLey [5] and SARA [6], and Petri net approaches [7]. Such methods are effective when a network of queues is an acceptable model of the execution behavior of the software, and when statistical estimates of timing (as opposed to guaranteed worst-case values) are acceptable.

Note that none of the preceding techniques is intended to solve exactly the problem addressed by this paper: validating performance requirements on a detailed design expressed as Ada code.

One popular non-method for dealing with performance requirements needs to be noted. There is some feeling that any concern for performance is improper, almost immoral, during program design. This attitude will be called the DEMO methodology (for DEliver Me from Optimizations). The DEMO

methodology calls for programs to be designed exclusively for correctness, modifiability, and maintainability, and that efficiency will taken care of later. The claim is that whatever degree of efficiency is called for can be provided, automatically, after the completion of detailed design, by one of three means:

* Compiler optimizations. "Any decent implementation" of the Ada language will provided extensive, global, optimizations, resulting in a system that will be as efficient as if it had been optimized by hand.

* Recoding hot-spots into low-level code. Since most of the execution time in many programs is taken up by a small proportion of the lines of code, those blocks of code may be recoded into assembly code, and good efficiency thereby obtained at small cost.

* Hardware. If the program does not run fast enough, a faster computer should be used. It does not matter if no such computer is available today, since it will be available soon.

The DEMO attitude probably developed in response to the older, pre-software engineering attitude that what makes software good was first, being efficient, followed closely by meeting spec, and all other values, such as maintainability, were of insufficient importance to deserve mention. If DEMO is a reaction to that attitude, it is largely justified, but nevertheless it is an overreaction. Consider each of the preceding three points:

While extensive, global, optimizations are within the state of the art, no Ada compiler known to this author provides the facilities previously demanded of "decent implementations" of the language. This is due to two factors: the demand for reasonably fast compilation, and the separate compilation facilities of the language. The result is that locally, generated code is not as particularly good, and global optimizations are not performed at all. Hence we cannot depend on compilers to solve our efficiency problems today.

Recoding of hot-spots into low-level code is of course a valuable technique as far as it goes. It does not help in two important cases: distributed inefficiency, and hot assembly code. The former refers to inefficiencies that are widely spread throughout a program; for example, a currently popular Ada compiler emits respectable code to reference arrays that have an index subtype such as 1..10, and highly inefficient code for arrays having the index subtype 0..9; no localized recoding will help. Hot assembly code refers to the case in which the program's hot-spots are in subroutines that are already in assembly code; in particular, when the hot-spots are in the run-time support code. For example, a program that is bound by task suspension and dispatch times cannot be helped by recoding into low-level code.

The hardware solution depends on cost-effectiveness. There is a balance between the cost of optimizing software, and maintaining that optimized software, against the cost of using of a faster computers, taking into account weight, power, and logistics issues. That balance cannot be casually tipped in either direction, no matter how convenient it would be for the software validation process.

The remainder of this section concerns a proposed approach to solving the problem defined above.

The basis of this approach is a change in viewpoint of the meaning of a design. A design is conventionally considered to define, roughly, an abstract computation (i.e., a function mapping inputs into outputs) together with a structure for the software. Note that the meaning of "structure of the software" is not entirely evident: the structure of the source code -- its hierarchical decomposition of a program into packages, tasks and subprograms and the separate compilation structure -- may be quite different from the structure of the software at run time. For example, code of one subprogram may be consolidated into the code of many others by means of inlining, and the program's static data may be divided up arbitrarily across several computers, and further into resident and non-resident segments. Conventionally, a "design" may specify any or all of these software structures.

For the purposes of verifying performance requirements, let us adopt the following viewpoint on the meaning of "design":

A DESIGN IS A CONSTRAINT ON THE INITIAL STATE OF THE TARGET MACHINE

That is, of the very large number of possible initial states for the target machine, a design selects a subset of those states, all of which presumably define programs that will perform according to the program's requirements. The word design will be used only in this sense below.

A design may be expressed as Ada code with annotations, or as Ada code with a separate data structure that constrains the mapping of the program onto the target machine. Examples of data that may reasonably be included in a design include the type and configuration of the target computer processors, memories, and communication channels, the mapping of static data onto memories, the mapping of tasks (or task types) to processors, and the identification of the run-time support code and parameters (such as task scheduling algorithm).

Even after the human designers have expressed all the information they have concerning the mapping of the program onto the target machine, additional information is required from the compiler concerning its mapping decisions. A form in which this information could be expressed will be presented shortly. This information includes data on the compiler's choices of optimizations, such as upmerging, inlining, and code motion.

When all of the available information on the source code and its mapping onto the target machine is available, then the verification of performance requirements can proceed. The essence of verifying performance requirements is to prove certain statements about the program behavior correct. The statements to be proven correct are the requirements ("The interval between updates to each track shall be on the average at most two seconds, and in no case longer than five seconds"), and the hypotheses are the available rules about the program and its mapping onto the target machine, together with some rules defining the behavior of the target machine itself.

Since the verification of a requirement is likely to be a long, but not particularly subtle, chain of reasoning, such verifications are likely candidates for automation. For this to be feasible, the data on the program will have to be expressed in a form acceptable to a theorem-proving system, such as a Prolog implementation [8] or a rule-based system [9]. For example, part of one set of rules presented to the verifier, which expresses the run-time structure of a subroutine, might have a semantic content (but not a form) such as

1) Subroutine S117 is completed when Block249 is completed and Loop98 is completed.

2) Loop98 is completed when Boolean4276 is false.

3) Block249 requires 79 milliseconds to complete.

4) Each iteration of Loop98 requires 182 milliseconds.

It is to be expected that attempts to verify requirements by this method will initially fail, simply because the conclusion is not justified by the available information. That is, requirements will not be validated because there is not sufficient data to establish that those requirements will be satisfied by the final system. This is as it ought to be.

When requirements cannot be validated due to the lack of sufficient data, additional information must be made available. Examples of such information would be a conclusion that is justified by the available information but is too deep for the verifier to discover (such as that some iterative process must converge within a fixed number of iterations), or information that is added to the design in order to meet the performance requirement (such as that when Condition equals Red, then the availability of Processor Alpha to Program Zeta will be 100%.) If such additional information does not permit the truth of the requirement to be deduced, then that requirement must be reported as not satisfied.

This rule-based verification approach has the following strengths:

* Accuracy. If a requirement is verified by rule-based verification, it is highly probable that any system produced according to the design will satisfy the requirement. Also, if a requirement is not verified by this method, it is highly probable that some system can be produced according to the design that will not satisfy the requirement. The method is well suited to handling worst-case requirements.

* Ability to handle non-determinism. In contrast to simulation-based approaches, the rule-based verification approach does not require that state transitions be uniquely defined: a rule stating that under certain conditions, either Process Alpha or Process Beta will be dispatched is perfectly acceptable.

* Ability to accept non-categorical specifications. A rule-based verification process is well suited to handle non-categorical specifications.

* Ability to repeat a validation following a modification. After a change to a design, such as specifying pragma inline for a function, validation may be repeated for only the cost of computer time.

This rule-based verification approach has the following weaknesses:

* Required tool support. The major tool support required to use rule-based verification is the rule-based system processor, and the additional function required of the Ada compiler (viz. emission of information on mapping decisions). Rule-based system processors are commercially available, but the modification to the Ada compiler is not trivial.

* Required human effort. Substantially more effort than is traditionally expended will be required on the part of the verifiers and the designers to achieve verification under this approach.

* Inability to handle non-invertible data dependencies. The use of a rule-based system will not solve the problem of unpredictable processing time.

* Inability to handle adaptive behavior. The use of a rule-based system will not solve the problem of unpredictable processing.

SUMMARY

Today, it is impossible to verify performance requirements on Ada software, except in a very approximate sense. There are several reasons for this difficulty, of which the main reason is the lack of use of information on the mapping of the program onto the target machine.

An approach to a partial solution to the verification of performance requirements on Ada software is here proposed, called the rule-based verification approach. This approach is suitable when the target machine is well-defined and when additional effort and expense are justified in order to guarantee that the performance requirements will be met by the final system.

REFERENCES

[1] B. W. Boehm, "Verifying and Validating Software Requirements and Design Specifications," IEEE Software, pp. 75-88, Jan. 1984.

[2] P. Zave,"The operational versus the conventional approach to software development," Communications of the ACM, pp. 104-118, Feb. 1984.

[3] M. W. Alford, "A requirements engineering methodology for real-time processing requirements," IEEE Transactions on Software Engineering, vol. SE-3, pp. 60-69, Jan. 1977.

[4] J. S. Tseng et al., "Real-Time Software Life Cycle with the Model System," IEEE Transactions on Software Engineering, vol. SE-12, pp. 358-373, Feb. 1986.

[5] P. Zave, "An operational approach to requirements specification for embedded systems," IEEE Transactions on Software Engineering, vol. SE-8, pp. 250-269, May 1982.

[6] G. Estrin et al., "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems," IEEE Transactions on Software Engineering, vol. SE-12, pp. 293-311, Feb. 1986.

[7] M. K. Molloy, "Discrete time stochastic Petri nets," IEEE Transactions on Software Engineering, vol SE-11, pp. 417-423, Apr. 1985.

[8] M. R. Genesereth and M. L. Ginsberg, "Logic Programming," Communications of the ACM, vol. 28, pp. 933-941, Sept. 1985.

[9] F. Hayes-Roth, "Rule-Based Systems," Communications of the ACM, vol. 28, pp. 921-932, Sept. 1985.