The Use of Sequencing Information in Software Specifications for Verification

Richard N. Taylor
Leon J. Osterweil†

Technical Report #197

Department of Information and Computer Science
University of California
Irvine, California 92717 U.S.A.

†Department of Computer Science
University of Colorado
Boulder, Colorado 80309 U.S.A.

April 7, 1983

The Use of Sequencing Information in Software Specifications for Verification

Richard N. Taylor
Leon J. Osterweil†

Programming Environment Project
Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

†Department of Computer Science
University of Colorado
Boulder, Colorado 80309

Abstract

Software requirements specifications, virtual machine definitions, and algorithmic design all place constraints on the sequence of operations that are permissible during a program's execution. This paper discusses how these constraints can be captured and used to aid in the program verification process. The sequencing constraints can be expressed as a grammar over the alphabet of program operations. Several techniques can be used in support of testing or verification based on these specifications. Dynamic analysis and static analysis are considered here. The automatic generation of some of these aids is feasible; the means of doing so is described.

1. Introduction

At any stage of the design or coding of a program at least two distinct sets of specifications are present: a statement of the requirements the software must satisfy and a statement of the semantics of the "language" in which the solution is written. (At low levels of implementation the latter are the semantics of the programming language being used; at higher levels they are the specifications of abstract data types, procedures, functions, etc.) Additionally, if algorithmic design exists it represents a third set of specifications. Either implicitly or explicitly these three sets express the need to either obey or avoid certain orderings of operations in the solution statement (among other things). This paper discusses techniques whereby these sequencing requirements can be captured and used to verify the absence of corresponding classes of program errors. It shall be shown that there are a variety of ways in which correct sequencing can be checked, that the corresponding classes of errors are often very significant, and that efficient

and reliable detectors for these errors can often be constructed automatically.

The ideas presented have several strengths which are particularly noteworthy. First, the possibility of effective, efficient verification is offered, often by means of automatically generated verification tools. Second, some of the techniques appear equally applicable to designs as well as to code. Third, they are useful even when only incomplete specifications are available. Furthermore the notions are aesthetically appealing, as they present a framework in which the relationships between some earlier verification tools and specification techniques can be clearly seen. (Included here are static data flow analysis, dynamic data flow analysis, and abstract data type specification techniques.)

2. Sequencing Constraints

Any problem solution specification, be it program code or design, should be viewed as a program to be executed on a virtual machine whose instruction set, S, consists of certain primitive operators $S = \{o_1, o_2, ..., o_n\}$. If the solution specification is written in a higher-order language, the primitive operators are mapped onto actual machine operators by a compiler. If the solution specification is a design, then it must first be mapped into higher-order language by a programmer. In both of these cases the primitives are not actual hardware instructions, but rather abstract operators, employed to facilitate human reasoning and solution formulation. In all cases, the solution specification is an algorithm which, when fed a specific string of input data, will execute a certain specific fixed sequence of primitive operators, called a computation, $C = (o_{i1}, o_{i2}, ..., o_{im})$.

A *sequencing constraint* is a set of strings, or language, defined over an alphabet T, a subset of S. Let the members of T be $\{t_1, ..., t_n\}$. The language is given by a grammar G defined over this alphabet. By imposing various restrictions on the grammars used, classes of sequencing constraints are defined. Of primary interest will be the classes given by regular grammars and context-free grammars.

A particular computation C *obeys* a sequencing constraint G if and only if the sequence C(T) is a member of the language given by G. C(T) is the string C, with all operations in S - T deleted.

Computations can be grouped into classes or categories in a number of ways. The one of interest here is the grouping of computations according to whether they obey given sequencing constraints. We use the term *Abstract Computation Type* (ACT) to describe the class of all abstract computations which obey some specific sequencing constraint. In particular we denote by A(G), the class of all computation sequences C, such that C(T) is in G. The purpose of this paper is to discuss the problem of determining whether or not all possible computations of a given algorithmic solution specification are members of a specified ACT.

3. The Sources of Sequencing Information

Software engineering as a discipline has matured to the point where several software development principles have emerged and become widely recognized. Chief among these is the notion that software should be thought of as a product to be developed through a sequence of phases, called the software lifecycle (e.g. see [10]). Each lifecycle phase results in the creation of a new enhancement or elaboration of the product. This new elaboration or enhancement is viewed as the basis for work in the next phase of production. In order for this to be acceptable as a safe basis for future development, the newest elaboration or enhancement must be "verified" to be correct. This process is essentially the process of comparing the new artifact to its predecessors and determining that it is consistent with them. After this verification is complete, the new artifact can be accepted as the basis for development in future phases.

There are a large number of techniques and approaches that have been suggested and employed towards the goal of being able to carry out verification effectively [1]. It is important to observe that the process of determining whether an algorithmic solution specification adheres to a sequence specification can and should be viewed as a verification technique also. Further, as noted earlier, this technique appears to be applicable at each stage of the software production lifecycle. From our perspective, each of these phases can be thought of as being, at least potentially, the source of either sequencing specifications, algorithmic sequences, or both.

In the following subsections we substantiate this notion and show that different techniques for carrying out the various lifecycle phases differ in the effectiveness with which they can be used as the basis for verification of correct sequencing.

3.1. Requirements

Requirements analysis is generally agreed to be the first phase in the software development lifecycle. As such it should be viewed as a primary source of the specifications of sequencing constraints in which we are interested. A requirements specification should, for our purposes, contain a specification of S, the language of primitive program operations, as well as specifications of a variety of sequencing constraints. Each of these must consist of a specification of T, the set of operators over which the sequencing constraint is defined, as well as a specification of G, the set of strings which is the constraint. Requirements are, of course, expressed in the language of the application area, and not in terms of program operations. Thus key to successful use of the sequencing constraints in verification is the ability to trace them from high-level "requirements operations" to operations in the program design.

Informal specifications are the most common type of requirements specification found in current practice. They are rather unsatisfactory as sources of the rigorous specification information described above, as they oblige the would-be sequence verifier to infer T, and both S and the rigorous definition of G

for each desired sequence specification, as well as do the tracing. Even in such specifications, however, it is common to see statements explicitly noting required orders of events. For example, with the rising interest in "software safety" [7], informal requirements specifications often include such statements as "Never let the program do x and then y". The requirements here are of orderings which must never be allowed to happen. It is often not difficult to infer definitions of S, T and G from such informal statements, and use them as the basis for rigorous verification, as will be shown in subsequent sections. Clearly a more formal statement of requirements is desirable, however, as the basis for this sort of verification.

One example of such a more formal approach is shown in [5], where A-7 aircraft requirements are phrased formally using functions, boolean algebra, and finite state machine modeling techniques. The various aircraft operating conditions define the states of the machine. "Similar" states are grouped together in classes called modes. Given a state and the occurrence of an event (such as data acquisition), a transition to a new state is defined. Certain outputs are issued whenever various states are entered. The set of events can be taken as S, the set of virtual machine operations. The required functions of the system, and certain safety conditions are readily cast as either required or forbidden sequences of certain of the events. Thus each of these functions and conditions is expressible as a sequence, supplying the sequencing condition, G. The events used in such a G dictate the specification of T for that sequence specification.

3.2. The Design Phase

Following the requirements specification phase of the software lifecycle is a phase or sequence of phases devoted to design. The purpose of this activity is to produce an algorithmic specification of a procedure which meets the specifications of the requirements phase. As such, design provides an algorithmic specification which is capable of being verified against sequencing constraints specified in the requirements. This requires that the algorithm be stated in terms of operations that are traceable to the abstract machine primitives, S, that were used in the requirements.

In addition, moreover, the design specification is best written in terms of the facilities provided by a virtual machine [11]. It, of course, must be operated according to its "rules". These rules explicitly or implicitly fix the legal orderings of the machine's operations. Such designs thus lay down additional sequencing constraints which, in conjunction with, or in addition to, sequencing constraints in the requirements, establish sequencing specifications against which the design (and subsequent code) can be verified.

Common components of virtual machine specifications are *Abstract Data Types* (ADTs). The semantics of an ADT determine the legal orderings for application of the access functions that it provides. For example, one sequencing constraint present in the specification of a stack ADT is that the "pop" operation may never be applied to an empty stack.

The sequencing constraints implied by an ADT are derived from its semantic specifications. The algebraic specification technique [4] implicitly defines the permissible orderings of events. The technique shows the legal sequencing of operations through definition of legal functional compositions and illegal sequencings of operations through definition of functional compositions mapping onto "error" elements of range spaces. An algebraic specification of the ADT "unbounded integer stack" is shown in Figure 1. Note that for this example T = {push, pop, create, top}. The grammar G expressing all legal sequences of operations can be given as follows:

SEQUENCE ::= create {OPERATIONS}*
OPERATIONS ::= push {OPERATIONS} {top}* {pop}

The specification of the constraint describing the need for there to be an equal number of pushes and pops at the end of any execution would only require a minor change to this grammar.

An ADT may also be specified through the use of *traces* [2] [8]. With this technique the legal orderings are much more explicit. Trace specifications are used to show basic legal orderings. Equivalence relationships are used to allow reduction of long traces to the basic ones. If a trace from a given program is reducible to the legal orderings, the path given by the trace is correct with respect to the ADT specification. The technique is equivalent in power to the

---

create: –> stack
push: integer X stack –> stack
pop: stack –> {stack ∪ stack_error}
top: stack –> {integer ∪ int_error}

Declare s: stack
i: integer

pop(push(i,s)) = s
top(push(i,s)) = i
pop(create) = stack_error
top(create) = int_error

Figure 1.
Algebraic definition of the ADT infinite stack of integers

---

algebraic technique.

Finally, it should be observed that there is no reason that sequencing requirements cannot be expressed in terms of the accessing primitives of more than one ADT. Thus constraints introduced during the design of an algorithm which uses several ADTs can be expressed and used in the verification process.

3.3. Coding

The final phase in the software development process is coding. During this phase actual computer software code is produced in accordance with the dictates of the requirements specification, and as dictated by the algorithmic specifications in the design. In the context of this paper we see that this phase is one during which a specification of all of the execution sequences of the final solution is produced. Hence the code produced during this phase is to be compared against all of the sequencing specifications and requirements which may have been inferred from the specifications of the earlier phases.

It is interesting to observe that the sequencing specified by code can also be compared to sequencing constraints laid down by the semantic rules of the coding language. That is, the programming language is a virtual machine that must be used in accordance with certain sequencing rules. For example, the Dave system [9] was created specifically to determine whether Fortran programs ever violated certain semantic rules of that language. In particular, the Dave analyzer checked to see if a Fortran program ever referenced any of its variables before they had been initialized, or if it ever defined a new value of a variable immediately after a value had just been defined for the same variable. In terms of our formalism, the set of primitive language operations, T, is {undefinition-of-x, definition-of-x, reference-to-x, undefinition-of-y, ..., reference-to-y, ...}, i.e. these operations as applied to each variable in the program. The grammar G excluded all sequences of the form (undefinition-of-x, reference-to-x) and (definition-of-x, definition-of-x).

The error analysis performed by this system proved to be quite useful, but it should be noted that, in the context of this paper, it performed a relatively weak and straightforward analysis. It is far more interesting to consider the possibility of carrying out analysis to determine whether a given body of code always obeys sequencing specifications dictated by the higher level designs and requirements. For example, if the code implements a stack, and the design specification for the stack indicates that a newly created stack is never to be popped, it seems very useful to be able to verify adherence of the code to this specification.

4. Methods Appropriate for Verifying Adherence to Sequence Specifications

The preceding sections have indicated that there are several ways in which the ability to verify sequence specifications could be of considerable value in assisting the software developer in producing software that is demonstrably free of certain important classes of errors. In this section we address the issue of how such verification can be made effectively and reliably.

4.1. Dynamic analysis

One technique that can be used to support verification based on sequencing specifications is dynamic analysis. With this technique an executable version of the solution specification is run and the sequence of operations performed is checked against the constraints. (The executable specification is most likely code, but an "animatable" design would be appropriate too. For brevity we will refer to either as the program for the remainder of this subsection.) In this manner the correctness of a single path is checked. The automatic generation of the necessary checking apparatus is both feasible and desirable.

Two checking schemes are possible. The first, suggested by M. Majester and communicated by D. Parnas, is to record the sequence of operations performed during execution, then check the sequence for legality as a post-process. The second scheme is to do the checking dynamically, as each operation is performed, in order to catch errors as soon as possible. With this approach debugging tools or exception handling procedures could be brought into play at the point of error.

With either scheme there are two aspects to the checking apparatus. First the program P must be transformed to a program P' such that the functional effect of the program is unchanged, but the sequence of operations C(T) is recorded. Then an appropriate mechanism must be created to check the legality of the recoded sequence against G, the constraint, either dynamically or as a post-process. These two aspects are considered in turn below.

Generation of P' from P requires only specification of T, the operators of interest. The difficulty of the transformation corresponds to the difficulty of isolating the operators in the program. If all elements of T correspond to procedure calls, then it is easy to see how the program could be scanned to locate all calls on the procedures, and, immediately before each call, to insert a statement which would write a token corresponding to the operation on the recording sequence. This would happen in much the same way as the values of variables are traced with debugging tools.

If some of the elements of T correspond to function calls, then identification of the appropriate statements is still easy, but the instrumentation is slightly more involved. It must be assured that the sequence of tokens emitted corresponds to the order of function invocation prescribed by the language in which the program is written. No real problems are found here, though, and it can be seen how the use of abstract data type access functions lends itself to this form of checking.

A much more substantial difficulty arises when some of the elements of T correspond to generic program activities, such as reference and definition of variables. Then identification of all the appropriate points in the program requiring instrumentation may be quite difficult. The transformations themselves are not likely to be any more difficult than with function calls, however.

The difficulty associated with constructing the necessary mechanism for checking the correctness of the sequence recorded by P' corresponds to the

complexity of the grammar G, which specifies the sequencing constraints. If G is a regular grammar, then constructing a finite state acceptor is easy (though if many operators from S' are used it may be large). Such a finite state machine could operate equally well during or after program execution. If G is context free then more work is involved in constructing the acceptor, but automatic generation of acceptors is clearly practical. If G is neither regular nor context free then the path to automatic acceptor construction is not clear. The possibilities here depend on the specific techniques used in G (such as relative counts of operators).

The drawbacks to dynamic analysis are apparent. Only a single path is checked, the checking of all paths is infeasible, and when an error is discovered it may be "too late" to do anything but record the failure and quit. The technique has the merits of simplicity and ease of application however. Manually constructed transformers and checkers exist [6]; it remains to move on to this more widely applicable and useful approach.

4.2. Static Analysis

In section 3 it was noted that a static analyzer (called a data flow analyzer) was created in order to determine whether Fortran programs ever allowed for the possibility of execution sequences violating either of two sequencing constraints derived from the specifications of the Fortran language. It was noted that such an analyzer is best viewed as a specific instance of a class of analyzers capable of determining whether all execution sequences of a program must necessarily adhere to sequencing specifications derived, perhaps, from design or requirements.

In [3] a formalism for describing sequences which must be adhered to (or eschewed) is presented, and there is also a discussion of the sorts of data flow analysis algorithms which can be combined and configured to check for the presence or absence of such sequences. Here we suggest that this type of analysis can always be based upon executing some standard algorithms, adapted from the theory of global program flow optimization to flowgraph representations of the program, suitably annotated. In [3] the annotations considered were selected to correspond to occurrences of such language primitive operations as reference and definition. Clearly program flowgraphs can equally well be annotated to correspond to occurrences of higher level primitive operations such as those specified in an ADT definition. The algorithms described in [3] and adaptations of them seem to suffice to determine adherence of the code represented by such annotated flowgraphs to many sequencing constraints of interest. In particular, if verification of adherence to safety conditions (i.e. "event x must never follow event y") is of interest, then adaptations of current data flow analysis algorithms are sufficient.

It is important to also observe that, because this technique is based upon the analysis of an annotated flowgraph, it is not restricted to analysis of code. Any algorithmic specification can be used as the basis for the derivation of a flowgraph. In particular, it is reasonable to consider the flowgraph derived from an algorithmic design specification. Here the nodes of the flowgraph represent

units of code to be written, specifying the high level operations to be realized. These operations can be indicated as annotations to the flowgraph, and this structure can then be used as the basis for data flow analysis as described earlier.

Other, different, static analysis techniques hold promise as well, such as modeling programs by finite state machines and comparing them with regular grammars expressing the constraints. Thus it appears that the notion of an abstract computation type can be viewed as the basic concept underlying a discipline of specifying and constructing static analyzers for verifying wide classes of sequencing specifications.

5. Future Directions

We are convinced that the Abstract Computation Type formalism, as described in the previous sections of this paper, may prove to be an important unifying influence in establishing systematic verification and testing techniques applicable at all phases of the software lifecycle. To determine whether this is so a great deal of additional work is in order. At present we have enunciated a formalism and have adduced a set of anecdotal examples showing that many apparently useful sequence specifications can be inferred from contemporary specifications and put into the framework of this formalism.

From here the following steps seem appropriate. 1) Determination of the power necessary to express the sorts of sequencing specifications that arise naturally in requirements and designs. 2) Derivation of appropriate notational techniques and supporting technology to aid in capturing these specifications and tracing their development. 3) Further development of analysis and testing procedures that could be created from and applied to the sequence specifications captured. 4) Evaluation of the overall utility of this approach.

Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada through grant A5538, the Defense Advanced Research Projects Agency of the United States Department of Defense under contract MDA-903-82-C-0039 to the Irvine Programming Environment Project, National Science Foundation grant MSC8000017, Department of Energy grant DE-AC02-80ER10718, and the U.S. Army Research Office grant DAAG29-80-C-0094. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government. (Whew!)

References

[1] Adrion, W. Richards, Martha A. Branstad, John C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *Computing Surveys, Vol. 14, No. 2,* June 1982, 159-192.

[2] Bartussek, W. and Parnas, D.L. Using traces to write abstract specifications for software modules. University of North Carolina Report No. TR 77-012, December 1977.

[3] Fosdick, L.D., and Osterweil, L.J. Data flow analysis in software reliability. *Computing Surveys, Vol. 8, No. 3,* September 1976, 305-330.

[4] Guttag, J.V. Abstract data types and the development of data structures. *Communications of the ACM, Vol. 20, No. 6,* June 1977, 396-404.

[5] Heninger, K.L. Specifying software requirements for complex systems: new techniques and their application. *IEEE Trans. on Software Eng., Vol. SE-6, No. 1,* January, 1980, 2-13.

[6] Huang, J. Detection of data flow anomaly through program instrumentation. *IEEE Trans. on Software Eng., Vol. SE-5, No. 3,* May 1979, 226-236.

[7] Leveson, Nancy G. Software Safety. *Software Engineering Notes, Vol. 7, No. 2,* April 1982, 21-24.

[8] Maclean, John. A formal foundation for the trace method of software specification. NRL Memorandum Report 4874, Naval Research Laboratory, Washington, D.C. September 1982.

[9] Osterweil, L.J. and L.D. Fosdick. DAVE - A validation, error detection and documentation system for Fortran programs. *Software - Practice and Experience, Vol 6.,* 1976, 473-486.

[10] Osterweil, L.J. A software lifecycle methodology and tool support, in *Software Development Tools,* W.E. Riddle and R.E. Fairly, eds., Springer-Verlag, 1980, 82-92.

[11] Parnas, D.L. Designing software for ease of extension and contraction. *IEEE Trans. on Software Eng., Vol. SE-5, No. 2,* March 1979, 128-137.