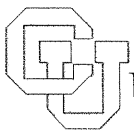
Specification and Static Evaluation of Sequencing Constraints in Software *

Kurt M. Olender
Leon J. Osterweil

CU-CS-335-86



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* Research supported by grants from the U.S. Department of Energy numbered DE-AC02-80ER10718 and from the National Science Foundation numbered DCR-8403341.

SPECIFICATION AND STATIC EVALUATION OF SEQUENCING CONSTRAINTS IN SOFTWARE

Kurt M. Olender
Leon J. Osterweil

CU-CS-335-86 June, 1986

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309-0430
U.S.A.

Research supported by grants from the U.S. Department of Energy numbered DE-AC02-80ER10718 and from the National Science Foundation numbered DCR-8403341.

ABSTRACT

This paper presents a flexible and general mechanism for specifying problems relating to the sequencing of events and mechanically translating them into dataflow analysis algorithms capable of solving those problems. Dataflow analysis has been used for quite some time in compiler code optimization. It has recently gained increasing attention as a way of statically checking for the presence or absence of errors and as a way of guiding the test case selection process. Most static analyzers, however, have been custom-built to search for fixed, and often quite limited, classes of dataflow conditions. We show that the range of sequences for which it is interesting and worthwhile to search is actually quite broad and diverse. We create a formalism for specifying this diversity of conditions. We then show that these conditions can be modeled essentially as dataflow analysis problems for which effective solutions are known and further show how these solutions can be exploited to serve as the basis for mechanical creation of analyzers for these conditions.

1. Introduction.

Determining and assuring quality in software relies — in the broadest sense — on the ability to demonstrate that software products behave in a fashion consistent with their specifications. Thus, demonstrating software quality requires a precise specification of intent, a precise determination of behavior, and the ability to show consistency between them.

In this paper, we develop and present a formalism capable of capturing those aspects of the intended behavior of software expressible as sequences of events. We then present statically derivable models of software behavior representing all possible event sequences that the software can perform. We fix upon one model particularly suitable for comparison to our event sequence specification formalism, thereby facilitating the sorts of consistency determination that should be quite useful.

2. Sequencing Constraints.

Taylor and Osterweil[1] argue that a significant class of errors can be modeled as the incorrect sequencing of the operations performed by a program because the inescapably algorithmic nature of software virtually guarantees that some requirements on that sequencing will be necessary. Although there are no formal studies classifying errors as due to faulty sequencing, in at least one study[2] nearly 50% of all errors in a set of programs were due to omitted or superfluous code. Clearly, when code is either omitted or superfluous, there is consequently also either an omission or oversupply of the program events contained in that code. A powerful and comprehensive event sequence analysis tool would be a strong aid to the detection of omitted and superfluous code errors, once the diagnostician had identified the appropriate events and precisely specified the desired and required sequences of those events. The ability to detect such errors seems to us to be a very important and useful one. Further, the consequent focus such a tool would place upon identification of key events and precise specification of how they should be sequenced is also most appropriate and useful in itself.

Let us take as an example a program that uses a module implementing operations for an output file with operations OPEN, CLOSE, and WRITE. Whenever we use a file, we must ensure that the file is opened before it is written or closed, and that an open file is closed before it is opened again. Additionally, we may want to know if the file is closed after opening without any intervening write being performed. While not strictly incorrect, this sequence would seem to be a strong indicator that either some code is missing (code to operate upon the file once it has been opened) or there is some superfluous code (the opening and closing of an unneeded file). We believe that software designers should be encouraged to think about and rigorize such specifications of desired and required event sequences so as to make as clear as possible the intent of code to be tested and analyzed. We further believe that the ability to translate such specifications into rigorous and effective analytic tests will be a very strong incentive for designers and coders to devise the specifications.

3. Previous Work.

3.1. Data Flow Anomalies.

Dataflow analysis is a powerful static analysis method that was first used in optimizing compilers. The information gathered was used to increase the performance of the object program by eliminating unnecessary computations. Since then, information gathered by dataflow techniques has been used to address an ever-increasing range of problems.

The first work that directly addresses the issue of systematic, diagnostic detection of violations of sequencing constraints was that of Fosdick and Osterweil.[3] They defined a **dataflow anomaly** as a sequence of the events reference (r), definition (d), and undefinition (u) of variables in a program that is either erroneous in itself or often symptomatic of an error. They considered that any program translated to machine code (from FORTRAN in particular) had an implicit specification of certain constraints on the sequencing of these operations.

Their DAVE[4] system statically detected instances of the anomalous sequences $<u,r>$ (uninitialized variables), $<d,d>$ and $<d,u>$ (redundant definitions) using variants of the dataflow algorithms previously used in compiler optimization. The first sequence listed is in fact an error. One cannot use a value that hasn't yet been computed. The last two sequences are not erroneous in themselves, but the value stored by the first definition is never used and its computation wasted. This may indicate either missing code between the two operations or that one of the operations is unnecessary and may be removed.

Since then, several other systems have been produced that use dataflow algorithms to detect these and other anomalous sequences of variable operations, e. g. FAST[5] and FORTVER[6] for FORTRAN programs and one for SETL programs.[7] This last system, for example, detects potentially non-terminating loops in SETL programs by statically detecting instances where no loop control variable is defined within the body of the loop.

3.2. More general sequences.

The systems mentioned in §3.1 detect the occurrence of a fixed set of sequences that are considered errors or anomalies over a fixed set of operations. Some work has been done on permitting the specification of sequencing constraints on program-defined operations.

The programming language Path Pascal[8] supports a facility to embed a specification of the correct sequencing of the procedures encapsulated in a monitor. This specification is then used to generate code for the dynamic detection of violations of the constraints. The notation is based on the path expressions of Habermann.[9] A similar scheme is proposed by Kieburtz and Silberschatz.[10] They also develop a set of proof rules to aid in the development of programs that satisfy the sequencing constraints.

Howden[11] uses finite state machines to specify correct sequences and gives an algorithm for the static detection of violations of this sequencing.

Flow expressions are an extension of regular expressions developed by Shaw[12] to specify sequencing in concurrent software. These additions permit the definition of recursively enumerable languages, so we have no general method to detect violations of a flow expression specification.[13]

Guttag's axiomatic specifications[14] implicitly define sequencing constraints by defining the effects of particular sequences of the operations of an abstract data type. These effects may explicitly be described as an error or may be seen to have no net effect, which can be used to indicate a potentially unnecessary computation.

McLean[15] has developed a logic for the specification and formal analysis of the correctness of sequencing based on the work of Bartussek and Parnas.[16]

Hoare[17] uses sequences in the specification of processes in CSP and defines a formal system for verification of these specifications.

4. Static evaluation of sequencing constraints.

We have seen that specification of event sequences has been quite widely considered in connection with supporting dynamic and static analysis of software. We have also seen that there has been some work in using such specifications as the basis for the creation of practical tools for the static and dynamic detection of fixed, limited sets of errors and anomalies, defined as sequences of events. Several authors have also proposed sequencing specification methods that support the dynamic analysis of user-defined specifications, but only Howden gives a method for the static analysis of these specifications. Consequently we have developed a formalism capable of specifying a range of event sequences that is broader and sharper than these prior ones. This breadth and sharpness seems to be necessary for appropriately powerful and sophisticated software analysis. Some definitions are necessary before introducing our formalism.

Please note that our notation is defined in Appendix A. It may be helpful to briefly scan this appendix before continuing with the treatment in this section.

4.1. Definitions.

Following Taylor and Osterweil[1] we shall define a **program** as an algorithmic specification of a problem solution, using some set of operations for a virtual machine that we will call **events** We do not restrict the events to those of an actual computing machine. Thus, we are extending the concept of a program beyond that of the final implementation by code. A **computation** is a sequence of events performed during an execution of the program. We can characterize a program by the set of all its possible computations, Π.

We will define a **sequencing constraint** as a set of computations over some interesting subset of program events. Our analysis of a sequencing constraint can ignore the occurrence of uninteresting events in the program.

A program **obeys** a sequencing constraint if every computation of the program, with all uninteresting events removed, is in the sequencing constraint. If Π is the set of all possible computations for that program and S is a sequencing constraint over the subset of program events Σ, then Π obeys S when $\Pi/\Sigma \subseteq S$. We will also say that a program **violates** a sequencing constraint if there is some π in Π not in S.

The most obvious (and frequently suggested) approach to the problem of determining if a program obeys a sequencing constraint is to model both the sequencing constraint and the set of computations as formal languages. Determining whether a program obeys its sequencing constraint then becomes the problem of determining if one language is a subset of another. This problem is in general undecidable for arbitrary types of formal languages. Thus, it is necessary to identify languages that can be used to effectively model both computation and constraint sets and are also computationally tractable.

To do this, we restrict S to some particular class of languages, and model Π/Σ with a language P such that $\Pi/\Sigma \subseteq P$ and we can solve the problem $P \subseteq S$. If our analysis shows that indeed $P \subseteq S$ then we know by transitivity that $\Pi/\Sigma \subseteq S$, and the program obeys the constraint.

4.2. A dataflow model.

Dataflow analysis generally uses a flowgraph to represent the program. Let $G = (V, E, s, t, \Sigma, L)$ be the flowgraph for our program Π. Let Σ be the set of interesting events from our sequencing constraint. L_v for some vertex v is either the event from Σ occurring when v is entered during an execution, or the null event ϵ if no interesting event takes place at that vertex. Let P be the set of computations associated with all path tracings from s to t,

$$P = \{L_p \mid p \text{ is a path from } s \text{ to } t \text{ in } G\}.$$

Since all executable paths are a subset of all paths, Π/Σ must be a subset of P. We will use P then as our model of Π/Σ. The dangers and problems in doing this are well known. The inability of dataflow analysis to distinguish and effectively compensate for the effects of unexecutable paths is a classical problem with this technique. Here we shall indicate why we believe that increasing and sharpening the expressive power of sequence specification can address and alleviate this problem.

Suppose that a given sequencing constraint is violated on some paths through a program. The analyst is left to wonder whether the violation occurs only on unexecutable paths, on some executable paths, or perhaps only on executable paths. Clearly the ability to determine whether the violation occurs on all paths or some paths is most important, as violations occurring on all program paths must necessarily also occur on all actual executions. Some previous systems, such as DAVE, are often able to make these distinctions.

In the case where the violation only occurs on some paths, the analyst needs more information — especially in the (not unusual) case of a large program in which a dataflow analyzer may detect hundreds or thousands of violations that occur on some paths. If the analyst learns that there are some violations that occur on all paths into a particular statement, this would be useful elaborative information. If the statement is known to be executable, the importance of the violation is correspondingly increased as the violation is then identified as one that would actually occur on some program execution.

A number of these points can be elucidated with our output file example. We could specify a constraint for the proper sequencing of events as the language denoted by one of the regular expressions 4-1a or 4-1b.

$$(\text{OPEN WRITE}^{*}\ \text{CLOSE})^{*} \qquad (4\text{-}1a)$$

$$(\text{OPEN WRITE}^{+}\ \text{CLOSE})^{*} \qquad (4\text{-}1b)$$

The difference is that constraint 4-1b requires a file that is opened to be written at least once.

```
begin
   OPEN;
   CLOSE;
end;
```

Program 4-2.

Program 4-2 obviously does nothing of interest. Constraint 4-1a is nevertheless obeyed. The use of constraint 4-1b as a specification permits the diagnosis of this ineffectual sequence. Although no competent programmer would intentionally write such a program, if we allow the possibility that there could be a large amount of code between the OPEN and CLOSE events, or that they may not be in the same procedure or even in sections of code written by the same programmer, then this situation becomes much more plausible and the use of constraint 4-1b as a specification will detect it.

```
begin
   OPEN;
   while not Done loop
      Process;
      WRITE;
   end loop;
   CLOSE;
end;
```

Program 4-3.

Assuming *Done* and *Process* are not interesting in our constraint, Program 4-3 again obeys constraint 4-1a but not 4-1b. This is unfortunate. The program in the second example is quite common, and is generally not objectionable, but constraint 4-1b finds Program 4-3 equally as incorrect as Program 4-2. We believe it is desirable, if not essential, for the diagnostician to have at his or her disposal a specification formalism sufficiently powerful and flexible to readily distinguish differences between such types of programs. In this particular case, we could distinguish between these two situations if we could specify in some manner that <OPEN,CLOSE> is not permissible only if a WRITE event is missing from **all** paths from the OPEN statement to the CLOSE statement in the program.

Therefore, we believe that a great deal of important flexibility is gained by supporting the specification of both existential and universal quantification of the paths over which a sequencing constraint must be obeyed.

There are some additional desiderata as well. We believe that it is important to be able to specify quantified sequencing constraints over paths either entering a vertex or leaving a vertex (or both).

```
begin
   OPEN;
   if Condition then
      while not Done loop
         Process;
         WRITE;
      end loop;
      CLOSE;          -- #1
   else
      CLOSE;          -- #2
   end if;
end;
```

Program 4-4.

Consider Program 4-4. We again assume that *Done*, *Process*, and *Condition* are uninteresting in our constraint. As in Program 4-3, it is true that a WRITE event lies between the OPEN event and a subsequent CLOSE event on some, but not all, paths, but this new example differs in important ways. There are two CLOSE events following the OPEN and so two sets of paths from the OPEN to each following CLOSE. No paths from the OPEN to CLOSE #2 have WRITE statements on them. Another way of expressing this is that all paths into CLOSE #2 violate constraint 4-1b. This is in contrast to the fact that only some of the paths into CLOSE #1 and only some of the paths out of the OPEN violate 4-1b. Thus the ability to specify constraints and then quantify them over paths either into or out of specified vertices gives us additional power to elucidate anomalies.

In Program 4-4 without this specificational capability we could only learn that there are some paths from the OPEN to a CLOSE on which WRITE events do not occur. In view of our inability to determine which paths are executable this is of questionable importance. The WRITE-free paths may all be unexecutable. If, however, our analysis is able to pinpoint a statement that cannot be executed without the violation of a stated constraint (as with CLOSE #2 of Program 4-4) a far more serious problem is indicated. We believe that it is reasonable to assume that some program paths are unexecutable, but that it is far less reasonable to assume that some statements are unexecutable and an anomaly on all paths into or out of a statement seems particularly worth detecting and reporting. It is this additional specificational capability that we are now suggesting.

5. A sequencing constraint language.

5.1. Syntax and semantics.

The examples in §4 suggest that a language for the specification of sequencing constraints to a static analysis tool should be able to support the specification of a set of desired event sequences, the sets of paths where those sequences must be obeyed, and a quantification over those path sets.

We shall use the notation for regular expressions to denote the desired sequences. Figure 5-1a gives a BNF grammar for regular expressions extended with complement and transitive closure operators. Non-terminal symbols will be indicated by identifiers and terminal symbols will be enclosed in double quotes, e. g. "(". Brackets enclose optional symbols and braces indicate 0 or more repetitions of the enclosed symbols. Please note that some of the BNF meta-symbols e. g. the brackets and parentheses, are similar to the regular expression terminals. This is unfortunately unavoidable. We will place each separate alternative on a separate line to avoid large numbers of meta-parentheses.

```
RE-primary  ::= Event
             ¦ "(" RE ")"

RE-factor   ::= RE-primary [ "*" ¦ "+" ]
             ¦ "¬" RE-primary

RE-term     ::= [RE-term] RE-factor

RE          ::= [RE "|"] RE-term
```

Figure 5-1a. Regular Expression Grammar

We also want to quantify over the paths on which the desired sequences must occur. We will use $\Subset$ for universal quantification (all paths in a specified set must obey the sequencing constraint given by the regular expression) and $\leftarrow$ for existential quantification (at least one path in the set must obey the regular expression).

We specify the set of paths by means of sets of vertices in the flowgraph. We call these vertices **anchors,** and call the expressions defined by the syntax of Figure 5-1b Anchored Quantified Regular Expression (AQRE) constraints.

```
QRE   ::= (≪ ¦ ←) RE
A-set ::= "{" Anchor { "," Anchor } "}"
AQRE  ::= [A-set] QRE [A-set]
```

Figure 5-1b. AQRE Grammar

The set of anchors preceding the QRE is called the **start anchor** set and the set following is the **end anchor** set. Either or both of these anchor sets may be omitted. If so, the start or end anchor set is implicitly $\{s\}$ or $\{t\}$ respectively. Omitted anchor sets have additional semantics explained in §5.2, so an AQRE constraint that explicitly gives $\{s\}$ or $\{t\}$ as an anchor set is not equivalent to one that uses $\{s\}$ or $\{t\}$ implicitly.

An AQRE, then, is a specification that either all or at least one of the paths from each start anchor vertex to each end anchor vertex obey the RE, depending upon the quantifier. More formally, the AQRE $\{X\} \lessdot \rho \{Y\}$ is obeyed if and only if

$$\forall x \in X\text{: } \forall y \in Y\text{: } \forall \text{ path } p \text{ from } x \text{ to } y\text{: } L_p \in \mathbb{L}(\rho)$$

The use of $\leftarrow$ as the AQRE quantifier changes the innermost logical quantifier to $\exists$. If either anchor set is empty (omitted sets implicitly have one element), the specification is trivially obeyed, since there are no paths.

For the purposes of this paper, we will use identifiers to denote vertices and sets of vertices in the anchor sets. An analysis tool will of course require a formal declaration of these identifiers and their meaning, but we will ignore that requirement here and ensure that the meaning of the anchor elements is clear from the context. We will use s to denote the entry vertex and t for the exit vertex of the flowgraph. If the identifier is the name of an event, then the set of vertices denoted is the set of all vertices labeled with that event. The anchor set (or A-set) in Figure 5-1b is the union of all vertices denoted by anchor identifiers in the list.

5.2. Examples from object code optimization problems.

We will illustrate these semantics further using some examples. First we show how our AQRE constraint language can be used to specify two classical types of object code optimization problems — available expressions and live variables.

5.2.1. Available expressions.

In classical object code optimization an expression e is said to be **available** at a vertex v if it is **generated** and not subsequently **killed** on **all** paths into that vertex. From our perspective, we shall let *gen* be the event consisting of the generation of expression e and *kill* represent the event consisting of the killing of expression e. The specification we must write is that the last interesting event on every path into v from the entry vertex s must be *gen*. In the language we have just defined that is

$$\{s\} \lessdot (kill \mid gen)^* gen \ \{v\} \tag{5-2}$$

In effect, we specify a required suffix to the sequence of interesting events along all paths from s to t. Our experience indicates we must often create specifications that assume s is the start anchor for the set of paths and that the constraint of interest must be a suffix of the event sequences along those paths. Thus for notational convenience and simplicity we let the omission of the start anchor from an AQRE imply that s is the start anchor and that the set of event sequences is $\Sigma^* \rho$, where ρ is the regular expression from the AQRE and Σ^* is the set of all sequences of interesting events. We may also write our specification for available expressions as

$$\lessdot gen \ \{v\} \tag{5-3}$$

5.2.2. Live variables.

In classical object code optimization a variable is defined to be **live** at a vertex v if it is subsequently referenced before being defined or undefined on at least one path leaving the vertex. Using r, d, and u to represent the reference, definition and

undefinition of variable x, then x is live at v if there exists at least one path starting at v on which the first interesting event is an r event. As an AQRE, that is

$$\{v\} \leftarrow r(d \mid r \mid u)^* \{t\}. \tag{5-4}$$

In this case we specify a required prefix of the sequence of events. As in the available expression example, we find that it is notationally convenient to omit one of the anchor sets. Here the end anchor is assumed to be $\{t\}$ and the set of event sequences is $\rho\Sigma^*$. Thus, we could also specify that x is live at v with

$$\{v\} \leftarrow r \tag{5-5}$$

If both anchor sets are omitted, we apply both rules. The constraint is that elements of $\mathbb{L}(\rho)$ must be a subsequence of either all or at least one path from s to t, depending upon the quantification. The AQRE $\leqslant d$ would thus specify that every path from s to t must have a d event somewhere along it.

5.3. The file example revisited.

Let us go back to our file module example. We will see that writing a correct constraint may be somewhat tricky. Although at first glance, it looks quite satisfactory, we cannot simply use the AQRE:

$$\{\text{OPEN}\} \leftarrow \text{WRITE}^+ \{\text{CLOSE}\} \tag{5-6}$$

To see this, consider Program 5-7.

```
begin
   OPEN; WRITE; CLOSE; -- line 1
   OPEN; WRITE; CLOSE; -- line 2
end;
```

Program 5-7.

To obey constraint 5-6, there must be at least one path from every OPEN to each CLOSE obeying WRITE^+, but the path from the OPEN in line 1 to the CLOSE in line 2 violates the constraint because of the intervening <CLOSE,OPEN> sequence. We might rectify this problem with the constraint:

$$\{\text{OPEN}\} \leftarrow (\text{WRITE}^+(\text{CLOSE OPEN WRITE}^+)^*) \{\text{CLOSE}\} \tag{5-8}$$

Unfortunately, although Program 5-9 is obviously incorrect, it trivially obeys this constraint because there are no OPEN events, and so the start anchor set is empty.

```
begin
   WRITE;
   CLOSE;
end;
```

Program 5-9.

We can however, write three short constraints that together give us what we desire.

$$\{s\} \lessdot (\text{OPEN WRITE}^* \text{ CLOSE})^* \{t\} \quad \text{(5-10a)}$$

$$\{\text{OPEN}\} \leftarrow \text{WRITE} \quad \text{(5-10b)}$$

$$\leftarrow \text{WRITE } \{\text{CLOSE}\} \quad \text{(5-10c)}$$

AQRE 5-10a is constraint 4-1a written in our language. Recalling that constraint, it permitted the ineffective Program 4-2. We add constraints 5-10(b,c) to close that gap by requiring that every OPEN event lead to a WRITE event on at least one path and every CLOSE event be entered from a WRITE event on at least one path.

5.4. Even more general sequencing constraints.

The example of 5-10(a-c) shows that there will arise situations where we must combine several AQRE constraints, in that case, a logical conjunction. In general, it appears to be useful to enable the combination of AQRE constraints with logical conjunction, disjunction or negation. We will call such an expression a Path Quantified Sequencing (PQS) constraint.

Conjunction ($/\backslash$) and disjunction ($\backslash/$) of PQS expressions are obeyed if either both or at least one of the components is obeyed, respectively. Negation ($\neg$) is obeyed if the component PQS is not obeyed.

PQS-primary ::= AQRE
| "(" PQS ")"

PQS-factor ::= ["¬"] PQS-primary

PQS-term ::= [PQS-term "/\"] PQS-factor

PQS ::= [PQS "\/"] PQS-term

Figure 5-11: PQS Constraint Grammar

5.5. Some properties and identities of PQS expressions.

These definitions give PQS expressions some properties and identities that are useful in analysis. We will say that two PQS constraints are equivalent if they are both obeyed with the same sequence sets on the same set of paths on all possible flowgraphs. One constraint implies another if it is obeyed along the paths and with the same sequences as the other on all possible flowgraphs, but not necessarily vice versa. To save space, we omit the proofs of these properties. Let σ, σ_1, and σ_2 be events, ρ, ρ_1, and ρ_2 be regular expressions and X and Y be anchor sets.

Formula (a) is an example of an identity that may permit the the coalescence of two or more AQRE components into a single AQRE and lead to more efficient algorithms for determining whether or not subject programs obey the constraints. The remaining formulas show the relationships between the quantifiers. Formula (b) shows that, as one would expect, all paths is a more restrictive condition than at least one path. Formula (c) shows that for a fixed sequence, the occurrence of that sequence on all paths into a vertex labeled with the last event in the sequence is not equivalent to a quantification over all paths out of the first event of the sequence, while (d) shows that

existential quantification in both directions is equivalent.

$$\text{(a)} \leftarrow\!(\rho_1 \mid \rho_2) \equiv \leftarrow\!\rho_1 \bigvee \leftarrow\!\rho_2$$

$$\text{(b)} \lessdot\rho \Rrightarrow \leftarrow\!\rho$$

$$\text{(c)} \lessdot\sigma_1\rho\{\sigma_2\} \not\equiv \{\sigma_1\}\lessdot\rho\sigma_2$$

$$\text{(d)} \leftarrow\!\sigma_1\rho\{\sigma_2\} \equiv \{\sigma_1\}\leftarrow\!\rho\sigma_2$$

6. An algorithm for static analysis of PQS constraints.

In this section, we will show how we can statically analyze programs for obedience to a PQS constraint. We will do this by first showing how we can analyze constraints of the form $\{s\}\lessdot\rho\{t\}$ where ρ is some regular expression. We will then show how we can modify that method to give us an algorithm for the analysis of AQRE constraints.

6.1. The basic algorithm.

We have seen that we model a program Π with a language P defined as the set of computations associated with all paths from s to t in a flowgraph G for the program. P obeys a constraint language S if $P \subseteq S$ and by transitivity Π/Σ obeys S. If $S = \mathbb{L}(\rho)$, then the PQS constraint is $\{s\}\lessdot\rho\{t\}$, and so analyzing for obedience to this PQS constraint is equivalent to determining if $P \subseteq S$.

We can obtain an algorithm for this problem if we can cast it into a dataflow analysis framework. We assume the reader is familiar with the definitions of a partially ordered set and a meet semi-lattice. If not, Chapter 2 in Hecht[18] contains a brief treatment.

6.1.1. Dataflow analysis frameworks.

A dataflow analysis framework is a characterization of a class of dataflow problems. In any instance of such a framework, we wish to compute some information at each vertex of the flowgraph. The events labeling the vertices in the graph are viewed as functions that transform the information as execution passes through the vertex. The information value at some particular vertex is a function of the values at its immediate predecessors as transformed by the labels at those predecessors. This gives us a set of simultaneous equations whose solution gives the values at each vertex.

The dataflow analysis framework consists of the set of all possible information values, a partial ordering of these values, the set of all possible transformation functions that might label a vertex, and an operation to combine the values from the immediate predecessors.

An instance of a dataflow framework consists of a particular flowgraph with a particular assignment of transformation functions to the vertices, thus giving a particular set of simultaneous equations to be solved.

In establishing a particular dataflow analysis framework, we are free to define the value set, ordering, combining operation and function set as convenient for the general problem at hand. As long as the values, partial order and combining operation form a meet semi-lattice and that the set of transformation functions is monotone with respect to the partial order then there exist efficient algorithms to compute solutions

to this set of equations.

If the set of transformation functions is also distributive with respect to the semi-lattice meet operation, then there are efficient algorithms to compute the best solution (called the **meet over all paths** (MOP) solution) as well.

Many algorithms exist for the MOP solution for dataflow problems, some of which are faster than others. Our own research is focusing on determining where still sharper and faster analytic algorithms might be automatically generated with the guidance of the characteristics and structure of the user's specific PQS expressions. This research is still ongoing and little can be reported here.

We now proceed to establish a dataflow analysis framework that will be sufficient to determine the obedience of sequencing constraints.

6.1.2. The value set.

Let $M = (\Sigma, K, i, A, \delta)$ be a deterministic finite state acceptor for S. If M scanned and accepted all computations in P, then our constraint S would be obeyed. P may be an infinite set, so we cannot enumerate its computations and test them with M.

Suppose we computed for each vertex v in G the set of states from M, call it $K(v)$, that M could be in after scanning the computation associated with any path from s to v in G. More formally,

$$K(v) = \{\delta_{L_p}(i) \mid p \text{ is a path from } s \text{ to } v \text{ in } G\}$$

Now consider $K(t)$. If every state in $K(t)$ is an accepting state in M then every computation is accepted by M. Thus, if $K(t) \subseteq A$ our constraint is obeyed.

6.1.3. The rest of the framework.

The value set then is the set of non-empty subsets of K (A deterministic finite state machine must always be in some state). The meet operation reflects how values are combined when two paths come together. Since we want all possible states that M could be in, we will choose set union for meet. This requires us to choose superset as the ordering relation for the semi-lattice and therefore K will be the bottom element.

Theorem 6-1: $\Lambda = (\mathbb{P}(K)-\varnothing, \supseteq, \cup)$ forms a meet semi-lattice.
Proof: Trivial from the definitions. □

We must now define the set of transformation functions, F, with which the vertices may be labeled. Let F be the set of all functions from K to K.

Theorem 6-2: The pair (Λ,F) is a distributive dataflow analysis framework.
Proof: See Appendix B. □

6.1.4. Deriving instances of this framework.

We must now show how to derive any particular instance of the above dataflow framework. Let $G' = (V', E', s', t, L')$ be a labeled flowgraph derived from G by adding a new entry node s' labeled with $\#$ and one edge from s' to s. Let $\delta_{\#}$ be the function that maps every state in K to the initial state i, δ_{ϵ} be the identity function, and δ_{σ} for any $\sigma \in \Sigma$ be the function from K to K obtained by applying σ to δ. $\delta_{\#}$ simulates putting M into its initial state, δ_{ϵ} simulates vertices where no interesting event takes place, and δ_{σ} gives all the transitions in M that might be taken when the event σ is scanned. We assign the function $\delta_{L'_v}$ to every vertex v. Note that all these functions are in F.

Theorem 6-3: The MOP solution to an instance of a dataflow framework as derived above gives a resultant value of $K(v)$ that is the set of states that M could be in after scanning the computation associated with any path from s to v.

Proof: For some path p from s' to v, the composition of the functions labeling the vertices on that path first puts M into its initial state and then transforms the state exactly as M would. Since the algorithm computes the MOP solution and the meet operator is set union, then the value $K(v)$ computed at each vertex v is the union over all paths of the states M would be in by reading the sequence of events labeling vertices along those paths. □

6.2. Analysis of an arbitrary AQRE.

Now we must show how our algorithm can be adapted to analyze an arbitrary AQRE constraint.

6.2.1. Path quantification.

We found that all paths obeyed the constraint defined by the regular expression in §6.1 if $K(t) \subseteq A$ at the conclusion of the MOP algorithm. Suppose instead that after the conclusion of the MOP algorithm we found that $K(t) \cap A \neq \emptyset$. In other words, M could be in at least one accepting state. This means that there is at least one path from s to t that is associated with a computation in the language denoted by the regular expression in the constraint. Analyzing for existential rather than universal quantification over paths then is a matter of altering the test we perform at the conclusion of the MOP algorithm.

In fact, there are other comparisons between $K(t)$ and A we could perform, but these are not required for the specification language defined in this paper. We are currently investigating how these other comparisons might enhance the power of the specification language.

6.2.2. End anchors.

We determined that a constraint was obeyed at t by examining $K(t)$ at the conclusion of the MOP algorithm. If the end anchor is a set of vertices, X, then we determine if the constraint is obeyed at all $x \in X$ by performing the appropriate comparison of $K(x)$ and A at all such x.

If $X = V$ and every such x obeyed the constraint, we have effectively determined that the set of prefixes of every computation defined by the flowgraph obeyed the constraint. Hoare[17] uses the prefixes of computations (which he calls traces) in the specification of CSP processes. We can analyze a CSP process for correct sequencing using our method if we can write a regular expression for the traces.

6.2.3. Start anchors.

In the method of §6.1, sequences start at s because $K(s)$ contains i. We caused this by adding an edge in G' from the #-labeled vertex s' to s. We could define a set of vertices Y such that we added edges $s'y$ for all $y \in Y$ to G'. If at the conclusion of the MOP algorithm, $K(t) \subseteq A$ then every path from every y to t matches the regular expression. Unfortunately, if $K(t) \not\subseteq A$ we cannot tell which y may be at the head of the path that caused the non-accepting state in $K(t)$. That information would almost certainly be required in a practical analysis tool. Existential quantification gives an even more serious problem. If we know that $K(t) \cap A \neq \emptyset$ at the conclusion of our MOP algorithm, then we only know that there is a path from at least one start anchor to t that obeys the regular expression. Our semantics state that there must exist at least one path from every start anchor to t obeying the regular expression. Thus, our dataflow algorithm does not meet the semantics we have defined for our language. We can handle both problems however by considering the direction of the paths.

6.2.4. Direction.

To this point we have quantified over a set of paths entering a vertex. In the previous section, we saw that this causes difficulties. Suppose that instead of analyzing a constraint like $\{X\}\leftarrow\rho$ on G, we analyzed $\leftarrow\rho^r\{X\}$ on G^r. In other words, the reverse of the constraint on the reverse of the graph. We turn the start anchor into an end anchor, which we can solve. In this way we quantify over all the paths **leaving** a vertex and ending at t.

6.2.5. Multiple anchors.

We can handle the situation where there are multiple end anchors and a single start anchor by examining each end anchor's K set. We transformed the situation of multiple start anchors and a single end anchor into the first situation by reversing the graph and constraint. Suppose that we have both multiple end anchors and multiple start anchors. Given the dataflow framework we have developed here, we cannot perform an analysis that satisfies the semantics we have given for our language with a single execution of an MOP algorithm for the same reasons we were forced to reverse the graph and constraint to handle the second situation above; we cannot tell which start anchor (or anchors) cause a given state to be in $K(v)$ for each end anchor v. Reversing the graph doesn't help. We have multiple start anchors in both orientations. With our existential quantification test, we still can only determine that there is at least one path from at least one start anchor to the end anchor we are currently examining. Therefore, we must run our MOP algorithm once for each start anchor. We may considerably reduce the size of the flowgraph for many of these start anchors, since not all vertices will be reachable from every start anchor, but even so, we almost certainly greatly increase the computation resources required to perform an analysis.

We are currently considering a modification to the dataflow framework given here that uses sets of states tagged with a vertex as its value set and determining if the additional computation to maintain this larger value set at each vertex offsets that required to perform multiple executions of an MOP algorithm.

6.3. Analysis of an arbitrary PQS constraint.

We cannot directly translate an arbitrary PQS expression into a single dataflow instance. However, we can combine the solution of the problems for each component AQRE of a PQS constraint using the definitions we have provided for the logical operations.

7. Summary and Conclusions.

We have developed a notation capable of describing broad and diverse classes of sequencing problems, and also capable of driving the mechanical creation of analyzers for effectively solving these problems.

Our research has indicated that the precise definition of sequencing problems is more difficult than we had earlier supposed, as a sequencing problem such as "undefined reference" from Fosdick and Osterweil[4] is actually a family of related problems. The precise specification of any particular member of this family requires some notational intricacy. The notation developed here seems to be sufficiently precise and effective. Further research aimed at applying the notation to the description of a variety of diagnostic problems is needed in order to confirm that the notation is effective.

Another important feature of the notation is that it can be used as the basis for mechanical creation of effective analyzers. Here our work is based upon showing that sequencing problems are profitably viewed as dataflow analysis problems for which good algorithms already exist. Our research has shown how such algorithms can be adapted to solve specific sequencing problems expressed in our notational formalism, where the adaptation process is also guided by the formalism.

We are currently implementing a system capable of recognizing sequencing constraints expressed in our formalism and converting them into effective dataflow analyzers. We shall experiment with this system to further determine the practicality and effectiveness of our ideas.

Ultimately we see the most important application of these ideas as being in flexible, adaptable software environments. We believe that users of such environments will need to quickly and easily create powerful diagnostic aids to study program phenomena which may evolve and become apparently dynamically, and perhaps suddenly. At such times, users will wish to use sequence analysis techniques to come to a better understanding of the structure of their programs. We believe that our formalism is the basis for the rapid, precise specification of such phenomena. We further believe that the mechanical analyzer generation method that we have developed would then be effective in quickly analyzing the user's program and returning valuable diagnostic information.

Accordingly we propose to embed our notation and procedures in an interactive software development environment in an attempt to evaluate these ideas and to also

come to a better understanding of how they complement more classical dynamic testing techniques.

8. Acknowledgments.

The authors gratefully acknowledge the support of the National Science Foundation and the U.S. Department of Energy in making this research possible.

Appendix A. Definitions and terminology.

Set operations.

We will use $\mathbb{P}(S)$ for the power set of set S and standard operators for other operations and set constants. Uppercase italic or greek letters will usually stand for sets and lowercase symbols for an arbitrary element of the set named by the corresponding uppercase symbol, but this convention will not be exclusively followed.

Sequences.

A sequence will be enclosed in angle brackets, with its elements separated by commas, e. g. $<a,b,c>$. Concatenation of sequences will be indicated by juxtaposition. Thus $<a><b> \equiv <a,b>$. If π is a sequence over some alphabet A and $\Sigma \subseteq A$, then π/Σ is the sequence π restricted to elements of Σ, in other words, the sequence π with all symbols not in Σ removed. So, for example,

$$<h,c,d,b,a>/\{a,b,c\} = <c,b,a>.$$

If Π is a set of sequences, then Π/Σ is the set of all sequences in Π restricted to Σ. For a sequence $\pi = <\pi_1, \cdots, \pi_n>$, we will also define $first(\pi) = \pi_1$, $last(\pi) = \pi_n$, $head(\pi) = <\pi_1, \cdots, \pi_{n-1}>$, $tail(\pi) = <\pi_2, \cdots, \pi_n>$, $j\text{-}th(\pi) = \pi_j$, and $\pi^r = <\pi_n, \cdots, \pi_1>$.

Finite state acceptors.

A **deterministic finite state acceptor** is the 5-tuple $(\Sigma, S, i, A, \delta)$ where Σ is a set of symbols called the alphabet, S is a finite non-empty set of states, $i \in S$ is the initial state, and $\delta: \Sigma \rightarrow (S \rightarrow S)$ is the transition function defining how the state of the machine changes when a symbol is scanned. We will denote the state change function when $\sigma \in \Sigma$ is scanned as δ_σ. For some sequence of symbols, $\alpha = <\alpha_1, \cdots, \alpha_n>$ we define δ_α as the identity function if α is the empty sequence, otherwise as the composition of the functions for each individual symbol in the sequence,

$$\delta_{\alpha_1} \circ \cdots \circ \delta_{\alpha_n}$$

Sequence α is **accepted** when $\delta_\alpha(i) \in A$. We will use $\mathbb{L}(M)$ to denote the language accepted by a finite state machine M and $\mathbb{L}(\rho)$ for the language denoted by a regular expression ρ.

Flowgraphs.

A **vertex-labeled single-exit flowgraph** G is the 6-tuple (V, E, s, t, Σ, L) where V is some set of vertices, $E \subseteq V \times V$ is a set of edges, $s \in V$ is the entry vertex, $t \in V$ is the exit vertex, Σ is some set of labels, and $L: V \rightarrow \Sigma \cup \{\epsilon\}$ is a function that

assigns a label from Σ, or the null label ϵ to each vertex. We will denote the label assigned to vertex v by L_v. We will always use the term flowgraph to indicate an vertex-labeled single-exit flowgraph.

Let u, v and w be vertices, and σ be an arbitrary label. Let uv represent the edge $(u,v) \in E$. Then we call u an **immediate predecessor** of v, and v an **immediate successor** of u. A path p is a sequence of vertices $<p_1, \cdots, p_n>$ where

$$\forall\ j\colon 1 \leq j \leq n-1\colon p_j p_{j+1} \in E$$

It **starts** at v when $first(p) = v$ and **ends** at v when $last(p) = v$. We will on occasion refer to a path that starts at u and ends at v as path uv. If path uv exists in G then we will refer to u as a **predecessor** of v and v as a **successor** of u. Vertex v is **on** path p when for some j, $j\text{-}th(p) = v$. The reverse of a flowgraph is $G^r = (V, E^r, t, s, \Sigma, L)$ where $uv \in E^r$ if and only if $vu \in E$.

Appendix B. Proof of Theorem 6-2.

>From Hecht[18] the theorem is true if F satisfies the following conditions:

1. F is closed over functional composition.
2. F contains the identity function
3. All functions in F are monotone with respect to the partial ordering of Λ.
4. Given some arbitrary element of Λ there is a function in F that can transform the bottom element K into that arbitrary element.
5. All functions in F are distributive over the meet operation.

Conditions 1 and 2 must be true since F contains all total functions over K. Condition 3 is implied by 5.

Proof of Condition 4. Formally, Condition 4 is

$$\forall\ X \subseteq K\colon \exists f \in F\colon X = f(K).$$

Consider some arbitrary element X of Λ. If $X = K$ then the identify function satisfies the above. So let $X = K-Y$. Define the function $f_{i \to j}$ as

$$f_{i \to j}(z) = (\textbf{if } z{=}i \textbf{ then } j \textbf{ else } z).$$

This function effectively removes the element i and adds j to its image. Now pick an arbitrary element x from X and let f be the composition of the functions $f_{y \to x}$ for all $y \in Y$ in some order. By condition 1, f is in F. The net effect is to remove all the elements of Y from the image of f and replace them with the same element x. So we have $f(K) = (K-Y) \cup \{x\} = X$. □

> Note: We exclude the empty set from the semi-lattice because there must be at least one element left in the set into which we can transform eliminated elements using the function defined above. The empty set is the image only of the empty set under any function in F.

Proof of Condition 5. Formally, this condition is

$$\forall\ f \in F\colon \forall\ X, Y \subseteq K\colon f(X \cup Y) = f(X) \cup f(Y).$$

If f is a function and X a subset of its domain then we define $f(X)$ as the image of X

under f, denoted $\mathrm{Im}_f(X)$, and equal to $\{f(x) \mid x \in X\}$. Let x be an arbitrary element of X. Then $f(x)$ is in $\mathrm{Im}_f(X)$ by definition. Since x is in $X \cup Y$ then x is in $\mathrm{Im}_f(X \cup Y)$ as well. But $f(x)$ is also in $f(X) \cup f(Y)$. The same argument holds for an arbitrary element of Y. Therefore any element of $f(X \cup Y)$ is also a member of $f(X) \cup f(Y)$ and vice versa. □

Since all conditions are fulfilled, we have a distributive dataflow analysis framework. □

References

1. R. N. Taylor and L. J. Osterweil, "Analysis and testing based on sequencing specifications," *Proc. of the 4th Jerusalem Conf. on Information Technology*, May 1984.
2. T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment," *The Journal of Systems and Software*, vol. 4, pp. 289-300, 1984.
3. L. D. Fosdick and L. J. Osterweil, "Data flow analysis in software reliability," *Computing Surveys*, vol. 8, no. 3, pp. 305-330, September 1976.
4. L. D. Fosdick and L. J. Osterweil, "DAVE — A validation, error detection and documentation system for FORTRAN programs," *Software — Practice and Experience*, vol. 6, pp. 473-486, 1976.
5. J. C. Browne and D. B. Johnson, "FAST: A second generations program analysis system," *Proc. of the 3rd Intl. Conf. on Software Engineering*, pp. 142-148, May 1978.
6. R. Conradi and D. Svanaes, "FORTVER — a tool for documentation and error diagnosis of FORTRAN-77 programs," Technical Report 1/85., Univ. of Trondheim, January 1985.
7. S. M. Freudenberger, "On the use of global optimization algorithms for the detection of semantic programming errors," Report No. NSO-24, Dept. of Computer Science, Courant Institute of Mathematical Sciences, New York Univ., 1984.
8. R. H. Campbell and R. B. Kolstad, "Path expressions in Pascal," *Proc. of the 4th Intl. Conf. on Software Engineering*, pp. 212-219, September 1979.
9. N. Habermann, "Path Expressions," Technical Report, Dept. of Computer Science, Carnegie-Mellon Univ., 1975.
10. R. B. Kieburtz and A. Silberschatz, "Access-right Expressions," *ACM Trans. on Programming Languages and Systems*, vol. 5, no. 1, pp. 78-96, January 1983.
11. W. E. Howden, "A general model for static analysis," *Proc. of the 16th Annual Hawaii Int'l. Conf. on System Sciences*, pp. 163-169, 1983.
12. A. C. Shaw, "Software descriptions with flow expressions," *IEEE Trans. on Software Engineering*, vol. SE-4, no. 3, pp. 242-254, May 1978.
13. T. Araki and N. Tokura, "Flow languages equal recursively enumerable languages," *Acta Informatica*, vol. 15, pp. 209-217, 1981.
14. J. V. Guttag, et. al., "Abstract data types and software validation," *Communications of the ACM*, vol. 21, no. 12, pp. 1048-1064, January 1979.

15. J. McLean, "A formal method for the abstract specification of software," *Journal of the ACM*, vol. 31, no. 3, pp. 600-627, July 1984.

16. W. Bartussek and D. L. Parnas, "Using traces to write abstract specifications for software modules," TR 77-02, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill, 1977.

17. C. A. R. Hoare, *Communicating Sequential Processes,* Prentice/Hall, 1985.

18. M. S. Hecht, *Flow analysis of computer programs,* North-Holland, 1977.