Interprocedural Static Analysis of Sequencing Constraints

KURT M. OLENDER and LEON J. OSTERWEIL
University of Colorado

This paper describes a system that automatically performs static interprocedural sequencing analysis from programmable constraint specifications. We describe the algorithms used for interprocedural analysis, relate the problems arising from the analysis of real-world programs, and show how these difficulties were overcome. Finally, we sketch the architecture of our prototype analysis system (called Cesar) and describe our experiences to date with its use, citing performance and error detection characteristics.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages*; D.2.2 [**Software Engineering**]: Tools and Techniques—*CASE*; D.2 4 [**Software Engineering**]: Program Verification—*validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*

General Terms: Algorithms, Reliability, Verification

Additional Key Words and Phrases: Error detection, interprocedural data flow analysis, sequencing constraints

1. INTRODUCTION

A common source of software failure is the incorrect sequencing of program events. Even when the specification of software functionality hides or omits explicit specification of legal or illegal sequences of events, specifications on sequencing are still inherent. The problems we use computers to solve are complex and require decomposition into smaller components. Solutions to these subproblems must cooperate in the solution of the overall task by sharing information in some way. This sharing creates a necessary partial order on the sequencing of computations, as the information must be produced before it can be used. Thus, some constraint on the sequencing of software events is necessary to properly specify (at some level of abstraction) a solution to the original problem.

This work was supported in part by National Science Foundation grant DCR-8403341, U.S. Department of Energy grant DE-FG02-84ER13283, and Defense Advanced Research Projects Agency grant DARPA-1537626.
Authors' addresses: K. M. Olender, Computer Science Dept. Colorado State University, Fort Collins, CO 80523; L. J. Osterweil, Dept. of Information and Computer Science, University of California, Irvine, CA 92717.

Even when the sequencing information is explicit, as in imperative programming languages like Pascal and in design notations like the structure charts used in the structured design methodology, the sheer size and volume of the software components can hide important sequencing phenomena, and consequently make more difficult the detection of violations of the constraints on this sequencing.

We can often detect violations of sequencing constraints automatically with static analysis. A number of tools have been built that statically uncover violations of a fixed set of sequencing constraints on a fixed set of program events, most commonly static data flow anomalies such as dead definitions or undefined references. In past work [22], we have argued that this kind of automatic static data-flow-based analysis can be extended to cover a much wider range of sequencing phenomena where the events of interest and the sequences themselves are specified not by the tool but by the program analyst. Such a capability appears to us to be an important addition to a software testing, analysis and verification environment.

While our previous work described a specification notation for sequencing constraints and gave an algorithm for their verification, we made some simplifying assumptions that must be overcome if we hope to analyze real-world programs. Such programs contain procedures, but our original algorithm assumed intraprocedural analysis. Real-world programs also contain many variables or program data objects upon which sequencing constraints may be individually applicable. Our original algorithm assumed just one such object.

This paper describes our solutions to these difficulties. After a brief review of our specification formalism, we relate our extensions to the algorithm of Olender and Osterweil [22] to handle interprocedural analysis and multiple objects on which the constraints must be verified. We also describe Cesar, our prototype analyzer that performs interprocedural static sequencing analysis on FORTRAN programs. We relate our experiences to date with the system, compare Cesar to some other static analysis systems, and finally outline our future plans.

2. CECIL: A SEQUENCING CONSTRAINT LANGUAGE

It will help the reader understand the purpose and context of the remainder of this paper if we first give a brief, intuitive overview of the language we use to specify sequencing constraints. Further detail can be found presented by Olender and Osterweil [22].

Cecil is a notation for expressing sequencing constraints based on regular expressions that is specifically oriented toward constraints that can be statically verified. A single term of a Cecil expression is an anchored, quantified, regular expression (AQRE). Each AQRE term consists of an *alphabet* of events for which sequencing is significant, the valid sequences or *traces* of those events, the events that bound or *anchor* the subtraces that must be constrained, and a *quantifier* of the possible bounded subtraces that must satisfy the constraint.

An example will make these ideas more concrete. Suppose we have a

program that requires a queue to store data. We construct an abstract data type (ADT) module that hides the implementation details of the queue and exports only the name of the type and a set of operations to *insert* a data element at the back the queue, *remove* the element at the front of the queue, return the element at the *front* of the queue, *create* an empty queue, and determine if a given queue *is empty*. Note that we follow Guttag's advice [15] and avoid operations that both change and query the state of the queue. Thus we define separate operations to tell us the next element in the queue and to remove that element from the queue.

A queue object must be created before any other operation can be performed on it. Additionally, removal or query of the front element fails if the queue is empty; there is nothing there. One possible sequencing constraint on queues, then, is that the first operation to be performed must be *create* and that every removal or front query must be preceded by an *is_empty* check that returns false or by an *insert* operation. Under these circumstances, we can be sure the queue is not empty. Obviously, it would be more accurate to constrain the number of insertions to be strictly greater than the number of removals. We know, however, that such a constraint cannot be statically checked as it requires knowledge of the number of executions of every loop, so we make a safe approximation that can be statically checked.

We might also want to prevent constructions that, while not failure-causing themselves, are frequently symptomatic of code defects. We might be suspicious, for example, if an element is removed from the queue before its value was queried by a front operation. This will not directly cause a failure, but why bother to store data in the queue if we never use the values we store? Perhaps the relevant code was mistakenly omitted. Perhaps we do not need the queue at all. In both cases, we would prefer to know this situation exists. We must be careful though. Front queries need not occur before every removal. We can easily imagine a program where elements might be removed without being used in certain situations. Our suspicions are raised only if there are *no* front queries on *any* execution path leading to the removal.

A Cecil constraint that expresses these concerns for a queue ADT is given in Figure 1. Events s and t respectively indicate the start and termination of program execution. The curly braces contain the alphabet, or the set of events whose sequencing we wish to constrain, the square brackets list the anchors, the events that bound the subtraces that will be constrained, and the quantifier (forall or exists) describes whether every possible bounded subtrace or at least one bounded subtrace must be in the language denoted by the regular expression.

Note that in the regular expression, a semicolon represents concatenation, a vertical bar represents union, an asterisk is reflexive-transitive closure, and a question mark is a wild card ranging over all single events in the alphabet (like the "any single character" wildcard used for file name "globbing" by the Unix command shell).

Since the events that bound subtraces need not necessarily be the same operations whose sequencing is constrained, events in the anchors need not

```
{insert, remove, create, front, empty, notempty} (
    [s] forall (create; ?*) [t]
    and [s] forall (?*; (notempty | insert)) [front,remove]
    and [s] exists (?*; front; notempty*) [remove]
    )
```

Fig. 1. A Cecil constraint for a queue ADT.

be in the alphabet. Our example uses the start and termination of program execution as anchors, but these events do not operate directly on a queue data object, and so are not listed in the alphabet.

In our example the *is_empty* predicate becomes two events, one for the true result and one for the false. It might seem that knowing the result requires run-time information. However, the typical use of such predicates is in the Boolean expression controlling a loop or conditional statement. The value is encoded directly into the control flow as we know which branches are taken under each condition, so we can statically distinguish between the empty and notempty events in our example under those conditions.

The first AQRE term requires that the *create* operation come first. We can read the AQRE as: "Every possible subtrace from program start to termination (uninclusive) must start with a create operation". We don't care what follows the create, as long as it comes first, so we allow any subsequent sequence of operations.

The second AQRE term prohibits front queries and removals from a (possibly) empty stack. We can read it as: "Every possible subtrace from program start ending at either a front or remove operation (uninclusive) must end with either an insert operation or an *is_empty* check that returns false. Since again we don't care what comes before that insert or nonemptiness check, effectively we have stated that the last queue operation before a front or remove event must be an insertion or nonemptiness check in every execution.

The third AQRE term requires that each removal be preceded on at least one execution path by a front query. We allow any number of nonemptiness checks after the front query as these do not change the state of the queue.

3. INTERPROCEDURAL SEQUENCING ANALYSIS

3.1 Background

Before launching into the details of our interprocedural sequencing analysis method, we will first briefly review some data flow analysis basics and introduce our notation.

A *data flow analysis framework* is a formal characterization of a class of data flow problems on a flowgraph that permits us to interpret the execution of the program in some abstract domain. Formally, let $\Lambda = (\nu, \sqsubseteq, \sqcap)$ be a meet semilattice. The set of values in the lattice is ν, partially ordered by $\sqsubseteq$, with a binary commutative, associative, and idempotent operation, $\sqcap$. Also let F be an operation space composed of a set of unary operations on ν.

Then $D = (\Lambda, F)$ is a data flow analysis framework if F satisfies certain conditions (discussed later) with respect to Λ. To satisfy these conditions, lattice Λ must contain a least, or *bottom* element, $\perp$. It may also contain a greatest, or *top* element, $\top$. Without loss of generality, we assume all lattices have a top element. We can always add an "artificial" top value to any set, if necessary.

Intuitively, ν forms the set of possible answers that we want to compute at each vertex in a flowgraph. We assume that program statements are associated with edges in the flowgraph. The vertices then represent control points between program statements. The value at any vertex is a function of the values at the predecessor vertices as transformed by the effects of the statements on the incoming edges. The operation space F describes the possible effects of statements in the abstract interpretation for the problem at hand, while the meet operation, $\sqcap$, describes how values from multiple incoming edges should be combined.

As an example, suppose we want to compute the smallest possible execution time for a given program. ν is the set of nonnegative real numbers $\mathscr{R}^+$ plus a special "infinite" value, ∞, to represent nontermination. The partial order is the usual "less than" operation, augmented so ∞ is greater than all other values. Since we want the smallest execution time, if two execution paths converge at a particular vertex, we take the minimum of the times for all converging paths. Thus, in this example, $\Lambda = (\mathscr{R}^+ \cup \{\infty\}, \leq, \min)$. The lattice bottom is 0 and the top is ∞. Each statement, regardless of its other computational effects, increases execution time by some amount, so F is the set of functions that adds a particular nonnegative real number to another value, that is,

$$F = \{ f_r : r \in \mathscr{R}^+ \cup \{\infty\} \wedge f_r(x) = x + r \}.$$

We assume that for all values $r \in \mathscr{R}^+ \cup \{\infty\}$, $r + \infty = \infty$.

An *instance*, I, of framework D is a specific flowgraph with each edge mapped to its proper effect. Formally, let G be a single-exit flowgraph[1] (V, s, t, E) where V is the set of vertices, s is the *entry* vertex, t is the *exit* vertex, and E is the set of directed edges. Each edge starts at a *source* vertex, and ends at a *target* vertex. Also let $H: E \rightarrow F$ map each edge to its effect from the operation space. An instance I is the pair (G, H) which effectively defines a set of simultaneous equations for the lattice values to be computed for each vertex given by

$$\text{Value}(s) = \perp$$

$$\text{Value}(v) = \bigsqcap_{e \in E_i(v)} H(e)(\text{Value}(\text{source}(e))), v \neq s$$

[1] This is not a serious restriction as we can always add an extra vertex to create a single exit when necessary.

where $E_i(v)$ is the set of incoming edges to v in the flowgraph. A solution to these equations is a solution to the original data flow problem for that instance. There may be many solutions to this set of equations, but in general, we want to compute a solution that is at least a maximal fixed point (MFP).

There are a number of algorithms that compute the MFP solution. A simple one is to set Value(s) to $\perp$ and Value(v) to $\top$ for all other vertices v, and iterate through the equations in some order recomputing Value until no changes occur through a complete iteration. Improvements in efficiency arise if we select the order wisely.

The best solution to the equations is the *meet over all paths* (MOP) solution. We can extend the mapping H from edges to paths by composing the individual edge effect functions in path order. Thus, we can describe the effect of an entire execution path by a single function. The MOP solution at a vertex is the combination of the effects of all paths entering that vertex, or more formally,

$$\mathcal{M}(v) = \bigsqcap_{p \in \Pi_v} H(p)(\perp)$$

where Π_v is the set of paths from s to v in G, and H is extended as described above. If F is closed under composition, contains the identity function, contains some function to transform the bottom element of the semilattice into each other semilattice value, and each function in F distributes over the meet operation, the framework is *distributive* and the MOP solution equals the MFP solution. Any MFP algorithm can be used to solve for it.

Our original sequencing analysis framework used a deterministic finite state acceptor (DFSA) as the basis for the lattice and operation space definitions. Each Cecil expression contains a regular expression, which can be converted to a DFSA accepting the same language. Let a DFSA be defined as the tuple $(\Sigma, S, A, \iota, \delta)$ where Σ is the alphabet of the Cecil AQRE plus a null event ϵ, S is the set of states, A is the subset of accepting states, ι is the initial state, and $\delta : \Sigma \rightarrow S \rightarrow S$ is the state transition function.

If $\mathcal{P}(S)$ is the power set of DFSA states, then the lattice Λ is $(\mathcal{P}(S), \supseteq, \cup)$. The operation space F is the set of total unary functions over S extended to $\mathcal{P}(S)$ in the conventional way and augmented by function $\hat{\emptyset}: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ that returns the empty set regardless of its argument. An instance flowgraph is generated from the source for a given routine so that the edges are labeled by the events from Σ that correspond to the source statement for that edge[2]. So in an instance flowgraph, we associate events with edges by a labeling function, $L : E \rightarrow \Sigma$. If $L(e) = \epsilon$ for some edge e, then $H(e)$ is the identity function 1_S. If $L(e)$ is some other event σ, then $H(e) = \delta(\sigma)$.

We showed [22] that this framework satisfies the appropriate conditions and so can be used to statically determine satisfaction of a Cecil constraint by computing the MOP solution to an instance and subsequently comparing that

[2] We discuss how this is done in Cesar in Section 4

MOP value to A at the source vertex of each edge v labeled by an end anchor event. If the AQRE quantifier is forall then the constraint is violated at v if $\mathscr{M}(v)$ is not contained in A. When the quantifier is exists, the constraint is violated at v if $\mathscr{M}(v)$ is disjoint from A.

Olender and Osterweil's method [22] assumes the program can be represented by a single flowgraph and that the statements in the program either represent events appearing in the Cecil constraint or are ignored. Unfortunately, virtually all real, nontrivial programs contain calls to procedures and functions, which may contain statements that directly represent Cecil events or may call still other routines, and which may use various methods to pass information through the procedure call. An evaluation of such a program against a Cecil constraint must take these situations into account.

In the remainder of this section, we introduce the enhancements necessary to deal with these complications in stages, first allowing procedures while retaining a single implicit global object, and then adding multiple objects and allowing for parameter passing mechanisms.

3.2 The Interprocedural Analysis Framework

In an interprocedural setting, we have two sorts of events that may label edges in our flowgraph. *Primitive* events are those listed in a Cecil expression's alphabet or anchor set. They represent the basic events of interest. Procedures and functions may represent arbitrarily complex combinations of events, however, and a call to a routine invokes the combined effect of all these events. We name these *call* events.

3.2.1 *Summary Analysis.* We could evaluate a program with call events against a Cecil constraint by substituting a copy of the routine's flowgraph for an edge labeled by a call to that routine. The Omega data flow analysis system for C used this approach for example [31].

This is often not a practical choice. Calls may be nested or even recursive, potentially resulting in a combinatorial explosion in the size of the flowgraph to be analyzed, and consequently in the time required to do the analysis. It is preferable to perform only a single analysis of a routine and use the result in the analyses of routines that call it.

Just as we can define the effect of a single control flow path by a function from F, we can define the effect of a set of paths (including an entire routine) as a function from F. We define a *summary function* $\mathscr{S} : V \to (\mathscr{P}(S) \to \mathscr{P}(S))$ to describe the combined effect of all paths from flowgraph entry to each vertex v in the flowgraph. The application of that effect to the semilattice value at the entry vertex gives the value of $\mathscr{M}$ at v. For framework D, the summary function is given by

$$\mathscr{S}(v) = \bigcup_{p \in \Pi_v} H(p)$$

since the union of two functions over a set is defined to return the union of the images of the functions. Under these circumstances, we can use the value of S at the exit vertex of a routine as the effect of the routine as a whole.

Such unions of functions are not in the original sequencing framework operation space, so we create a new operation space by closing F under union of functions, thereby also creating a new data flow analysis framework. This new framework still satisfies the conditions for distributivity, so we have not lost the ability to compute the MOP solution. From this point onward, D and F will refer to the new framework and operation space.

A function from sets to sets can also be considered a relation over pairs of the elements. It will be more convenient if we think of $S(v)$ as a relation over $S \times S$ that defines the possible state transitions that might be taken by the DFSA along the paths of the flowgraph terminating at v. By the same token, the functions in F are also relations.

To compute $\mathcal{S}$ we use an adaptation of the functional interprocedural data flow analysis procedure of Sharir and Pnueli [24], a scheme similar to that independently developed by Fairfield and Hennell [11] for traditional data flow analysis of recursive procedures. Let R be a set of routines forming a program, $\{R_j : 1 \leq j \leq n\}$. Features of a particular routine are denoted by subscripting. The entry vertex for the flowgraph of routine R_k is s_k, while $\mathcal{M}_j$ is the MOP solution for routine R_j. Flowgraph edges labeled by primitive or null events have the same H values in an instance as before. Flowgraph edges labeled by call events however are assigned a relation from F that describes the possible state transitions from the entry of the routine to its exit. Thus when $L(e) = \text{call}_j$ for e in E_k, $H_k(e) = \mathcal{S}_j(t_j)$.

The system of equations to be simultaneously solved is the union of the equations for all routines R_j in the program. Sharir and Pnueli give an iterative algorithm to compute each $\mathcal{S}_j$ when the semilattice is finite and the operation space forms a bounded semilattice itself. Framework D fits these conditions exactly. The set of relations over a finite set is a finite (and therefore bounded) semilattice.

There are some dependencies among the equations of this system that can be exploited to reduce the size of the system to be solved at any one time. Any program consisting of multiple routines has a calling structure that can be described by a labeled flowgraph known as the *call graph*. The labeling function is a bijection from vertices to routines. Routine R_j precedes R_k in the call graph if R_j calls R_k. Routines at the leaves of the call graph call no other routines. The system of equations for each leaf routine can be solved without reference to the equations for any other routine as if it were an intraprocedural problem. Given the summary data for the leaves, routines that call only leaf routines can be solved, and so on. The reverse of a topological sort of the call graph vertices defines an ordering sufficient to assure the solution of the equation systems for each routine when the call graph is acyclic. This is sometimes called a *leaves-up* order, or a *postorder*.

If the call graph contains cycles,[3] the acyclic condensation formed by the strongly connected components (SCC) of the call graph can be used to order

[3] There may be a group of mutually recursive routines, for example.

the solutions. $\mathscr{S}$ for the routines in each SCC can be computed with an iterative data flow analysis algorithm.[4]

3.2.2 *State Propagation.* After the summary analysis phase, the values of the possible DFSA states at each vertex in all routines are computed during a *state propagation* phase. The value of $\mathscr{M}$ in each routine depends on the states in effect at the call sites; these are called the *initial states*, $\mathscr{I}$, so that

$$\mathscr{M}_j(v_j) = \mathscr{S}_j(v_j)(\mathscr{I}_j).$$

For the routine at the entry of the call graph (the *main* routine), $\mathscr{I}$ contains only the initial state of the DFSA. The value of $\mathscr{I}$ for other routines depends on the values of $\mathscr{M}$ at the call sites in the calling routines. Let $\mathscr{C}_j$ be the set of all sources of edges labeled with event call_j paired with the index of the routine in which the event occurs. These are the vertices whose values of $\mathscr{M}$ will be passed to R_j by the call events.

$$\mathscr{C}_j = \{(i, u_i) \mid \exists e : \text{source}(e) = u_i \wedge L_i(e) = \text{call}_j\}.$$

Both $\mathscr{I}$ and $\mathscr{M}$ can be computed for the routines in topsort order given $\mathscr{C}_j$ for each routine R_j.

The set $\mathscr{C}_j$ is likely to contain more than one element since routine R_j may be called from several sites in the program. $\mathscr{M}$ must reflect the possible states a DFSA might be in over all paths into a vertex. What does a vertex in a routine called from several places represent? The answer will affect the definition of the paths into such a vertex and consequently our analysis.

One view is that a vertex in a routine should represent a unique location to which all calls lead. Thus, "all paths" includes all paths from all possible call sites, while "at least one path" is defined as at least one path from at least one call site. We let $\mathscr{I}_j$ be the union of the states obtained from all possible calls of R_j.

$$\mathscr{I}_j = \bigcup_{(k, u_k) \in \mathscr{C}_j} \mathscr{M}_k(u_k).$$

A Cecil constraint is satisfied in this view if the MOP solution at the end anchor vertices has the desired relation to the accepting states of the DFSA.

An alternate interpretation is that a vertex in the flowgraph of a routine represents many vertices, one for each call, as if a copy of the flowgraph of the routine were substituted at each call edge to create a single flowgraph for the program. The set of all paths into one of these "replicated" vertices is the set of all paths that pass through one particular call site. Since each call site may be embedded in a routine that is in turn called from multiple sites, we must maintain a separate $\mathscr{I}_j$ for each possible chain of calls leading into R_j and compute a separate value of $\mathscr{M}_j$ for each of these chains. This appears to provide the same potential for combinatorial explosion caused by direct substitution of flowgraphs for call events. It is not necessary, however, to

[4] Since F forms a bounded semilattice itself, we can easily form a dataflow framework with F as the lattice component.

```
PROCEDURE P IS
BEGIN
    create;
    insert(i);
    IF Condition THEN
        j := front;
        Compute(j);
        Q; -- 1
    ELSE
        Q; -- 2
    ENDIF;
END P;

PROCEDURE Q IS
BEGIN
    remove;
END Q;
```

Fig. 2. A program with multiple procedures.

distinguish among call chains that provide the same value of $\mathscr{I}$ to a routine. We can maintain a set of $\mathscr{I}$ values and compute the resulting set of $\mathscr{M}$ values. $\mathscr{I}_j$ is defined as in the above equation, but the values become power sets of states. The values of $\mathscr{M}_j$ must also become power sets of states. The maximum size of these power sets depends only on the number of states in the DFSA and not on the number of calls to a given routine. A Cecil constraint would be violated at some end anchor vertex v in V_j if some element of $\mathscr{M}_j(v)$ failed to have the proper relation to the set of accepting states in the DFSA.

These two views are equivalent when the Cecil constraint is universally quantified, but not when it is existentially quantified. Under the first interpretation, an existentially quantified Cecil constraint is satisfied if at least one call chain causes an accepting state in $\mathscr{M}_j(v)$. The second requires that every call chain have that effect.

The second view appears to be more in line with our usual expectations. Procedure P in Figure 2 satisfies our Cecil constraint for queues under the first interpretation. A front query precedes the call to Q at statement 1 and therefore precedes the close event it encapsulates. It is more probable, though, that an analyst would prefer to consider the call at statement 2 a violation of our constraint. It is still impossible for a write event to precede the close event at this point, and therefore possible either that the call to Q is superfluous or that code to query front was unintentionally omitted in the else clause of P's if statement.

Thus we must extend our data flow analysis framework again. Our new framework has lattice values $\mathscr{P}(\mathscr{P}(S))$, and the functions in F must be extended in the natural way to this new lattice. Since we have simply extended the framework to sets of the original values, the conditions for a distributive data flow analysis framework remain satisfied.

```
begin
    - Summary analysis
    for each SCC_j, in reverse topsort order loop
        Compute {S_k | R_k ∈ SCC_j};
    end loop;

    - State Propagation
    for each SCC_j, in topsort order loop
        Compute {C_k | R_k ∈ SCC_j};
        Compute {I_k | R_k ∈ SCC_j};
        Compute {M_k | R_k ∈ SCC_j};
    end loop;

    - Violation Detection
    for each routine R_k in the call graph in any order loop
        for each v such that ∃e : v = source(e) ∧ L_k(e) ∈ α loop
            if ∃m ∈ M_k(v) : m ⋠ A then
                Report a violation at v;
            end if;
        end loop;
    end loop;
end;
```

Fig. 3. An interprocedural analysis algorithm.

When the call graph contains cycles, the computation of $\mathscr{I}$ can be done simultaneously for all routines in a SCC of the call graph using an iterative method, since some of these routines call others within the SCC. New state sets added to $\mathscr{I}$ in one routine may cause new values of $\mathscr{I}$ to be propagated to the routines it calls. This algorithm must terminate since it would never remove state sets from $\mathscr{I}$. In effect, this is a data flow subproblem, and an appropriate data flow algorithm will compute the result.

Our interprocedural dataflow analysis method is briefly summarized as the program of Figure 3. In that algorithm, SCC_j is the jth strongly connected component of the call graph. $\mathscr{S}_k$, $\mathscr{I}_k$, $\mathscr{C}_k$ and $\mathscr{M}_k$ are respectively the summary relation, initial state set, call location set, and MOP solution for a routine R_k in SCC_j. A is the set of accepting states for the DFSA generated from the AQRE, α is the set of end anchor events from that AQRE, and $\preceq$ is either set containment or nondisjointness depending on the AQRE quantifier.

3.2.3 *An Example.* We will evaluate the program of Figure 2 against the third AQRE in the Cecil constraint of Figure 1 to illustrate our interprocedural algorithm.

```
{insert, remove, create, front, empty, notempty}
        [s] exists (?*; front; notempty*) [remove]
```

The DFSA for the regular expression is in Figure 4, while Figure 5 gives the flowgraph for this program with edges labeled by the appropriate events, and

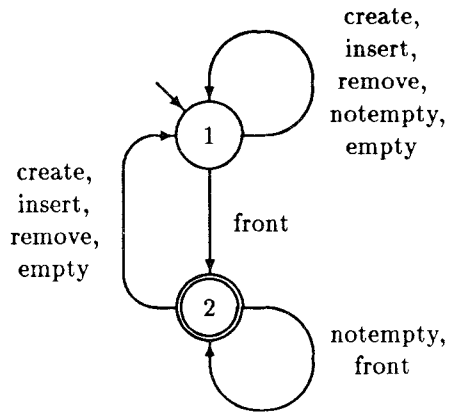

Fig. 4. DFSA for (?*; front, notempty*).

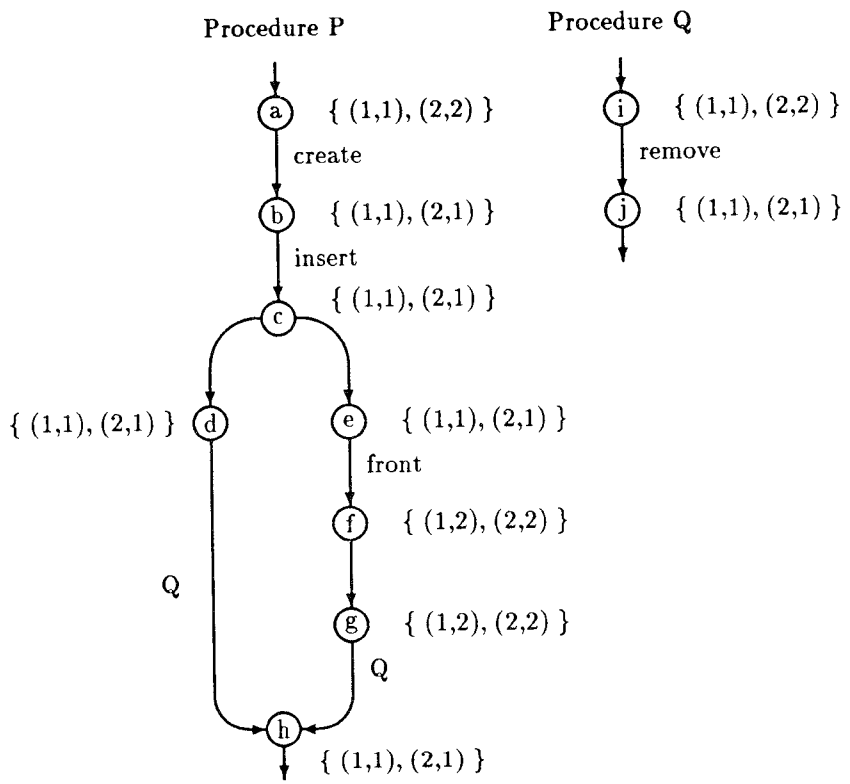

Fig 5. Example program summary data.

vertices labeled by the state transition relations computed for $\mathscr{S}$. Any edge not marked in the figure is assumed to be labeled with the null event.

In Figure 5 we see that the entry vertices to each routine are labeled with the identity relation. The routine can have no effect before it begins. Each value at a succeeding vertex is the union of the values at their respective

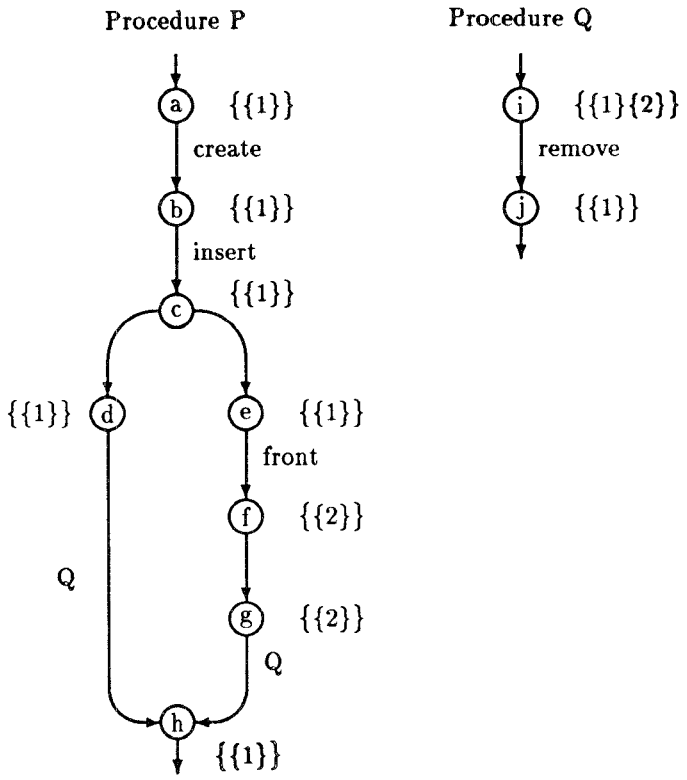

Fig. 6. Example state propagation data.

predecessors composed with the relation for the possible state transitions of the primitive or call event labeling the edge. Vertex b has only one predecessor, a. From the DFSA diagram in Figure 4, the possible state transitions that can be taken for the incoming *create* event, $\delta(create)$, are $\{(1,1),(2,1)\}$. Thus, $\mathcal{S}_P(a) \circ \delta(create)$ gives us the value of $\mathcal{S}_P(b)$ in Figure 5.

The computation of $\mathcal{S}_P(h)$ is slightly more complicated. Vertex h has two predecessors. By coincidence, both incoming edges are labeled with call events for procedure Q. We use the summary relation value previously computed for the exit vertex of Q, $\mathcal{S}_Q(j)$, as the effect of this call, so $\mathcal{S}_P(h) = (\mathcal{S}_P(d) \circ \mathcal{S}_Q(j)) \cup (\mathcal{S}_P(g) \circ \mathcal{S}_Q(j))$, again leading to the value shown in Figure 5.

Figure 6 gives the values of the state sets computed during state propagation. Since P is the root of the call graph for this example and the DFSA initial state is 1, $\mathcal{I}_P = \{\{1\}\}$. We apply that value as argument to $\mathcal{S}_Q(v)$ for each vertex v in Q to obtain the sets of possible states the DFSA might be in when scanning the sequence of events along any path into that vertex. For procedure Q, we note that $\mathcal{C}_Q = \{(P, g), (P, d)\}$. Thus $\mathcal{I}_Q = \{\mathcal{M}_P(g), \mathcal{M}_P(d)\}$ and $\mathcal{M}_Q(j) = \mathcal{S}_Q(j)(\mathcal{I}_Q)$.

We detect violations of the sequencing constraint by examining the elements of $\mathcal{M}$ at the sources of edges labeled by end anchor events, *remove* in the example. Vertex i in procedure Q is the sole end anchor vertex, so we examine the elements of $\mathcal{M}_Q(i) = \{\{1\}, \{2\}\}$. Since the AQRE quantifier is exists, we check each of these elements for disjointness with $A = \{2\}$ and note that a violation does occur. We trace the source of this violation back to its location in P by noting that it is the call on edge dh that passes $\{1\}$ to Q.

3.3 Selection of an MFP algorithm

From an intraprocedural standpoint, the selection of a specific MFP algorithm is relevant only for efficiency; all produce the same result. We selected the fast path algorithm of Tarjan [26] for implementation in Cesar.

Tarjan's algorithm is efficient, with an almost linear time complexity in the number of vertices in the flowgraph when the flowgraph is reducible. This algorithm offers other technical advantages as well. Some forms of Cecil expressions are more efficiently analyzed by propagating sequencing information in reverse, i.e., from the exit to the entry of the flowgraph. Tarjan's algorithm makes this easier and also requires only that the flowgraph or its reverse be reducible. Programs with single entry, multiple exit loops are reducible themselves, but have reverses that are not reducible. Some other methods would not permit the analysis of these irreducible reverses. We may also implement this algorithm generically as an algebra over Λ with union, concatenation, and closure operators, making our analysis code more reusable.

3.4 Objects

The analysis of real programs raises important issues other than multiple procedures. Sequencing constraints are defined in terms of events, which are actions performed on objects. A real program contains specific data objects on which specific operations are performed. A sequencing constraint must be satisfied for each of these objects independently. Thus, a program containing several queues must satisfy our Cecil constraint for each individual queue.

The events whose sequencing we analyze define a view of the behavior of the program. To select an analysis view, we must define not only the operations of interest, but also the objects on which they act. The data flow anomaly detection performed by Fosdick and Osterweil's DAVE system [13] considered every variable, regardless of type, to be an object. Freudenberger's SETL data flow anomaly system defined an object (in the context of finding potentially nonterminating loops) as a variable referenced in the loop control predicate [14]. The constraint of sequences of operations on an ADT like a file or a stack requires that an object be defined as the data structure that implements an instance of the ADT.

It is conceptually straightforward to analyze programs under such circumstances by creating a different copy of the flowgraph of a subprogram for each distinct object, where the flowgraph's edges are labeled only by events that act on that object. This is essentially the program slicing notion introduced by Weiser [29]. We can analyze each slice of a program that affects an individual object independently.

This approach is not necessarily inefficient. In a typical program any single object will be affected only by some small subset of program statements. We can eliminate single-entry, single-exit subflowgraphs where all the edges are labeled by the null event for some particular object. Thus, each slice will have an associated flowgraph much smaller than the original flowgraph. The cumulative size of all flowgraph slices, and consequently the cost of analysis, will be a function only of the total number of events and not of the total number of objects.

Thus, we consider a routine to represent the slice of a particular subprogram for a distinct object. Each routine's effect is computed from the corresponding flowgraph slice. We can construct a new *slice call graph* from the call relationships among the routine slices. The slice call graph is larger than the original, but has important redeeming properties. A single subprogram call typically does not affect every object in the calling subprogram, and so will not generate a call event in every slice. Mutually recursive groups of subprograms need be iteratively solved only for those slices that have recursive dependencies. Slices for local objects not affected by a recursive call are leaves in the new call graph, even if the original subprogram was recursive. Thus the slice call graph has a larger proportion of leaves and smaller strongly connected components, thereby decreasing the dependencies among the equation systems.

Early data flow analysis tools did not exploit these properties because the use of bit vector operations made the analysis independent of the number of program objects. The data flow analysis framework required for Cesar's more general sequencing evaluation cannot use bit vectors to represent the data flow summaries computed for all objects, but this slicing scheme makes the cost of our more general sequencing analysis independent of the number of program objects as well.

3.5 Objects in an Interprocedural Setting

The method in which an object is passed between two subprograms affects how we treat the call event during sequencing analysis. We classify objects either as *global* (passed implicitly via scoping rules) or *parameter* objects (passed explicitly in the call statement). Parameter objects are affected by the programming language's parameter passing mechanism. While various languages may have different parameter passing mechanisms, in the absence of aliasing, we can classify them as either *in*, *out*, or *in-out*. In the following, let P be a subprogram that calls Q such that object ω in P is potentially affected by the call. We will denote the routine corresponding to the slice of P for ω as P_ω. We also assume that no aliasing takes place.

If ω is global to Q, then the summary relation for Q_ω at Q's exit vertex describes the effect of Q on ω in any calling routine, so $\mathcal{S}_{Q_\omega}(t_{Q_\omega})$ is the effect of any edge labeled by a call to Q_ω used during computation of $\mathcal{S}_{P_\omega}$. During state propagation, $\mathcal{I}_{Q_\omega}$ is obtained from all calls of Q_ω in all routines.

If ω is passed instead through an *in-out* parameter μ of Q, the computation is identical, except that we consider P_ω to call Q_μ and so during summary analysis we use $\mathcal{S}_{Q_\mu}(t_{Q_\mu})$ as the effect of Q on ω in P_ω.

If μ is an *in* parameter, ω cannot be affected in P by the call. The sequencing effect in P_ω is the identity relation. During summary analysis we can ignore a call to a subprogram with ω matched to an *in* parameter. In Q, μ receives the state of ω, however, so during state propagation it is treated the same as an *in-out* parameter object.

The last situation is when μ is an *out* parameter. Object ω is affected in P_ω by the call to Q_μ, but Q_μ does not receive initial states from any of its callers; it always begins with the initial state of the DFSA just as the main routine does. Because of this, the sequencing effect of an *out* parameter overrides any accumulated effect on paths leading up to the call site. It cannot simply be composed into $\mathscr{S}$ during summary analysis. *Out* parameters therefore require more than a cosmetic alteration to the treatment of call events. They require a new definition of summary to incorporate this overriding effect—effectively the definition of a new algebra for interpretation by Tarjan's algorithm.

Suppose the values in this new algebra are pairs of relations (N, O) with operations of composition, union, and closure defined in terms of relational composition, union, and closure as follows

$$(N_1, O_1) \circ (N_2, O_2) = ((N_1 \circ N_2), (O_1 \circ N_2) \cup O_2)$$
$$(N_1, O_1) \cup (N_2, O_2) = ((N_1 \cup N_2), (O_1 \cup O_2))$$
$$(N_1, O_1)^* = (N_1^*, (O_1 \circ N_1^*))$$

The N relation represents a *normal* effect that is composed into the sequencing effect of a path as described previously. The O relation represents *overriding* effect that replaces any sequencing effect from prefixes of the path with itself.

Let (N, O) be the computed sequencing effect of Q on formal parameter μ at the exit vertex for slice Q_μ and let $X = N \cup O$. Then we take the effect of Q_μ in a caller P_ω as

$$\mathscr{S}_{Q_\mu}(t_{Q_\mu}) = \begin{cases} (X, \emptyset), & \text{if } \mu \text{ is } \textit{in-out} \\ (1_S, \emptyset), & \text{if } \mu \text{ is } \textit{in} \\ (\emptyset, X), & \text{if } \mu \text{ is } \textit{out}. \end{cases}$$

From the definition of composition for these relation pairs, we can see that the O component erases any N component and asserts itself. This pair algebra, fortunately, is required only during summary analysis. During state propagation and when we pass a summary relation upward in the call graph to a caller during summary analysis, we use $N \cup O$ as the sequencing summary relation. At any given vertex, it is possible that only some incoming execution paths contain *out* parameter call events, so the cumulative effect must be the union of the two components. State propagation for slices corresponding to *out* formal parameters must take $\mathscr{I}$ from the DFSA initial state rather than from the MOP values from the call sites.

We can see that the slice call graph used for state propagation could be different than the one used for summary analysis, when we take parameter modes into account. Slices for *out* parameters could become roots in a forest of call relationships in the state propagation graph. Slices for *in* parameters could be eliminated from the summary analysis call graph. This separation would help avoid some recomputation of sequencing effects when a program is changed, as we shall see later.

3.6 Pragmatic Considerations and Limitations

Given these enhancements to the interprocedural evaluation method discussed in Section 3.2, programs containing procedure calls and multiple instances of objects can be evaluated against a sequencing constraint expressed in Cecil. There are some pragmatic considerations, however, that place restrictions on this analysis.

A major consideration is the representation of the state transition relations that form $\mathcal{S}$. This representation must be compact and yet allow an efficient implementation of union, composition, and closure. One possible representation is a Boolean adjacency matrix. If the DFSA has n states, $\mathcal{S}$ requires n^2 bits per object per vertex. Union of two Boolean matrices take unit time if bit vector operations are used and the elements of the matrix are stored contiguously. Composition and closure take $\mathcal{O}(n^2/\log(n))$ time using bit vector operations and the "Four Russians" Boolean matrix multiplication algorithm [2]. In addition, the number of state sets that must be maintained in $\mathcal{I}$ and $\mathcal{M}$ to handle multiple call sites in exponential in n. This places a practical limit on the size of the DFSA and therefore on the regular expressions in the Cecil constraint. The current implementation of Cesar, for example, limits n to 8, requiring 64 bits per flowgraph vertex for the representation of $\mathcal{S}$ and 256 bits per vertex for $\mathcal{I}$ and $\mathcal{M}$.

We must also realize that parameter modes cannot be recognized solely by examination of the mode specifier in the procedure or function definition. The data type of the parameter must also be considered. Pointer types passed as *in* parameters are effectively *in-out* parameters for the data pointed to.

Aliasing is a third major consideration, one that is a common concern of all static analysis and evaluation methods. A parameter object may represent the same object as another global or parameter object for one particular call, but not for others. The analysis method defined here summarizes the sequencing effects of a routine in a way that yields inaccurate results in the presence of aliasing. The summary analysis algorithm assumes that the objects are distinct. The summary relations reflect the relative sequencing of events acting only one one object. If two objects are sometimes aliases for one another, we must create a new slice for each aliasing pattern to compute the proper summary relation. This may be expensive, but we have no data as it was not implemented in Cesar. A related problem concerns arrays and pointer variables. In general, an array must be considered a single object since one cannot always statically determine which element a given array reference may denote [13]. Pointer variables may also represent sets of

objects, although data flow analysis methods can be used to determine those sets [31].

4. CESAR: A STATIC SEQUENCING EVALUATION SYSTEM

Cesar is an experimental implementation of the static interprocedural sequencing analysis algorithm described in Section 3. Given a program, a Cecil constraint, and some additional information, it can determine if (and where) a violation of that Cecil constraint occurs.

Cesar now supports sequencing evaluation of FORTRAN programs. Additional tools to support analysis and evaluation of C and Ada programs are under construction. The system consists of 45 tool fragments comprising over 30,000 lines of Ada and 5,000 lines of FORTRAN, coordinated by the Odin system [8], which is used both as an object manager and user interface. Of those tool fragments, 14 implement the language independent analysis and evaluation system comprising the heart of Cesar. The remainder are either tools to support a particular programming language or to view the data structures produced.

4.1 The Architecture of Cesar

Figure 7 shows the organization of the tool fragments comprising Cesar. The tools grouped in the *PL front end* subsystem produce parse trees, attribute tables or other information from program source code that might be used for a variety of tools in an environment, such as pretty printers, compilers or debuggers. The FORTRAN front end tools were obtained from the Toolpack project [23] and are written in FORTRAN. Sequencing evaluation of FORTRAN programs was supported first because of the ready availability of these tool fragments and interfaces to the data structures they produce.

From that front end information, the *graphing* subsystem produces the labeled flowgraph upon which our interprocedural sequencing analysis is based. Since the front end tools and their results may vary for different programming languages, each language will require its own set of graphing subsystem tools. The Cecil subsystem produces a syntax tree and semantic information necessary to drive our interprocedural algorithm, such as the deterministic finite state acceptors for the regular expressions in the AQRE terms.

Since a Cecil constraint is language independent, the analyst must specify which programming language constructs correspond to the events in a Cecil constraint. An event is specified as a pattern to be matched in a syntax tree of the program to be analyzed. The Tepee tree pattern specification language associates these patterns with the event identifiers used in a Cecil expression. The Tepee subsystem produces the internal form for these pattern associations required by the graphing and analysis subsystems.

Cesar must also be capable of dealing with programs for which not all source code is available. Real programs often use predefined subprograms from object libraries, which fall into this category. Programs still under development may have some subprograms stubbed and so full source is again

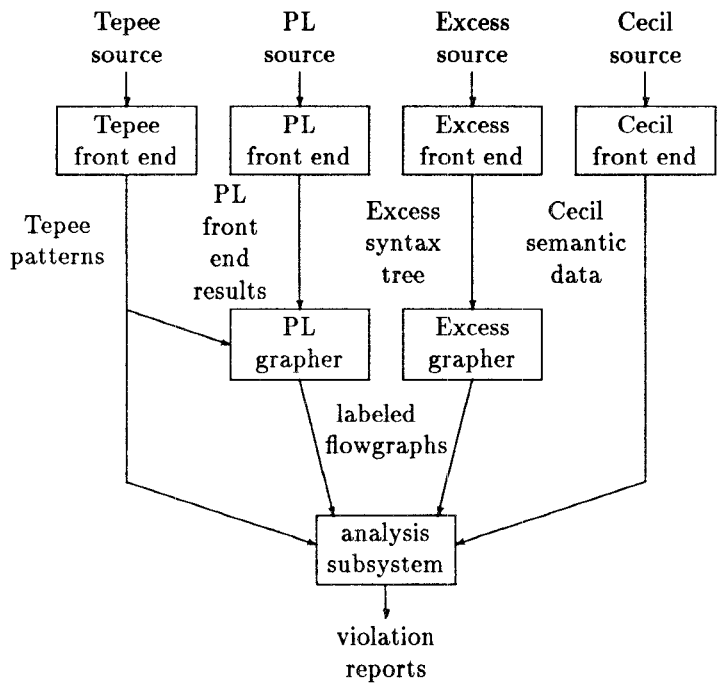

Fig. 7. Top-level Cesar architecture.

unavailable. The Excess language allows the assertion of a sequencing effect for these "external" procedures. The Excess subsystem produces the necessary labeled flowgraph information for inclusion in the analysis of a program.

The analysis subsystem performs the sequencing analysis as described in Section 3. It is based on the labeled flowgraph information produced by the graphing tools and is independent of the programming language used to write the programs to be analyzed.

4.2 Tool Integration and Object Management Through Odin

Cesar's fine scale architecture is heavily influenced by the Odin system and the desire to limit the amount of recomputation required when a change is made to the source code of a program. While a static analysis system tends to be less expensive to execute than a mechanical formal verification system or full suite of dynamic tests, it is still not cheap. It is advantageous to reuse as much still-valid information as possible during reanalysis when source code changes are made.

Odin is a tool integration system that facilitates the elimination of unnecessary recomputation of derived information. Odin supports the straightforward and effective introduction of new tools into that environment (even though those tools may not have been specifically designed for use within an environment) through the management of the data objects created and used

by those tools. A command to Odin is a request for a data object. Knowledge of how an object is derived from some set of base *source* objects is encoded in a *derivation graph* specified as an extended production system. When a request for an object is received, Odin checks an object repository (its *cache*) for the presence of the object. If the object is present and has not been made obsolete by the alteration of the objects on which it is based, the object is fetched to satisfy the request. If the object does not exist or is obsolete, Odin requests the data needed to build it. If one or more of the required input objects are nonexistent or obsolete, Odin recursively requests the objects necessary to build them. Once all data required to build a product object are available, Odin invokes the proper tool, traversing up the tree of requests building new objects and maintaining them in its cache until the initially requested product is built.

Like *make* [12], Odin will only rederive objects that have been made obsolete by alteration of their respective base source objects. Unlike *make*, it will recognize when one of these rederived objects is identical to its previous incarnation and halt the rederivation at that point. It also creates key object dependencies automatically from information in the derivation graph and possibly in the source objects themselves. The Odin system thus provides a simple user model for an arbitrary suite of software tools (specifiable by the user) with object management that limits rederivation in the face of change.

5. EXPERIENCE WITH CESAR

5.1 Effectiveness of Cesar

As we shall see later, our prototype implementation of Cesar is not yet adequate to provide the wide base of analysis results necessary to reach a definitive conclusion about its effectiveness as a sequencing evaluation system in a real software development setting. Some indications can be obtained from an analysis of the errors that were detected during the testing of the components of the Cesar system itself. Since the necessary Ada graphing tools are not yet available, we cannot apply Cesar to itself. We can, however, examine the errors found during development of Cesar to determine if they could be characterized by statically detectable sequencing constraints.

We kept a comprehensive log of the faults discovered during development of the Cesar component tools recording their location, effect and cause. These faults were classified as

(A) Statically detectable as classical data flow anomalies
(B) Statically detectable, program independent sequencing anomalies.
(C) Statically detectable, but program specific anomalies
(D) Not sequencing errors, or not statically detectable errors.

Faults were considered statically detectable sequencing faults if they were caused by incorrect or anomalous sequences of program operations which are themselves statically recognizable. Thus both the event and the sequence must be statically apparent. If the definition of an event includes information that can be known only at run-time (e.g., an argument to a procedure call

Error Class	Number	Percent
A	27	12.1
B	62	27.8
C	51	22.9
D	83	37.2

Fig. 8. Faults found during Cesar development.

must have some predefined value and the actual argument is a variable), then the event was not considered statically recognizable.

The class A faults were traditional data flow anomalies such as undefined references and dead definitions which have been statically detected by other tools as well as by Cesar. Class B faults were statically detectable sequencing faults where the sequencing constraint is independent of the program or algorithm implemented, essentially sequencing constraints on operations that implement an ADT. Counted in this group, for example, were null pointer references that could have been detected statically by constraining every pointer deference to be preceded either by an allocation (Ptr := new Block) or predicate test for inequality to the null pointer (if Ptr /= null then . . .).

Class C reflects sequencing constraints that are dependent on the particular algorithm being implemented. The sequencing of operations necessary to correctly perform a master file update from a transaction file is one example. This could certainly be encoded as a regular expression, and the operations on the master and transaction files are statically recognizable.

The class D faults were those that were either not sequencing faults (e.g., an incorrect constant or arithmetic operator in an expression) or where the events were not statically recognizable as defined above.

Faults from classes A, B, and C are statically detectable. Our current implementation, however, is oriented toward constraints that apply across all programs that use some set of operations. Algorithm-specific constraints tend to have larger defining DFSA's, and as mentioned above, the size of the DFSA is critical to the performance of Cesar. Thus, while our algorithms could detect class C faults, given an appropriate constraint and definitions of the events, we take the conservative view and consider Class C faults impractical for Cesar and do not count them as Cesar-detectable.

Figure 8 summarizes the results of this analysis on the 223 faults discovered during testing. Faults discovered as a result of the type checks and other analyses performed by the Ada compiler are not counted among these. All compiler detected faults are statically detectable by nature, but since they were detected by a tool other than Cesar, we count them neither for nor against Cesar.

The number of class A faults found roughly corresponds to that found by previous data flow anomaly systems [13]. The number is perhaps somewhat less than usual, but we attribute that to Ada's strong typing. A typical source of data flow anomalies is a misspelled variable name. An Ada compiler would detect many of these as undeclared variables or type clashes, and we did not

record compiler-detected faults. Our analysis of the faults discovered that a further 27.8% of them were characterized by program-independent sequencing constraints where the events are statically recognizable. Thus, a Cesar for Ada could have detected 39.9% of the faults discovered during testing of the Cesar system given the proper Cecil specifications for the abstract data types or data flow anomalies involved.

While this fault analysis is far from conclusive proof that Cesar can add a significant capability to software evaluation, it does encourage us to continue experimentation. Forty percent is a significant proportion of faults.

5.2 Performance of Cesar

Figure 9 gives representative timings on a Sun 3/260 workstation running SunOS 4.1 for an analysis of a selection of FORTRAN programs. The executables of the Cesar tools were produced by the Alsys version 5.2 Ada compiler.

The "dave" and "matrix" examples are analyses of a Cecil description of traditional data flow anomalies as described by Fosdick and Osterweil [13]. The "load-ur" example is similar, except that the analysis is only of undefined references. The "file", "load-io", and "stack" examples demonstrate Cesar's abilities to detect violations of sequencing constraints on user-defined abstract data types. The two "load" examples are analysis of different sequencing constraints within the same program, one of data flow anomalies and the other of operations on files. They differ in number of routines because six of the routines implement primitive events in the "load-io" view, and Cesar need not examine their internal structure.

In Figure 9, the first block of lines gives information about each test case. The middle block lists the important resources, CPU time and disk space, consumed by the analysis. The last block breaks the time measurement into the proportion used directly by Cesar tools to perform the analysis, that used by Odin for object management (Odin direct), and the time used by Cesar tools required to generate the dynamic data dependencies needed by Odin to control the data derivations (Odin indirect) as described in Section 5.3. Since we use the term "object" in two senses in this paper, we will distinguish between them by referring to an external data object generated by a Cesar tool and managed by Odin as an *Odin object*, while an internal program data object on which an event acts will be called a *Cesar object*, unless it is clear by context which sense is intended.

Figure 10 gives some typical performance results for reanalysis necessitated by small source changes once a full-run analysis has been completed. All examples are reanalyses of the "load-io" test. The "dual" test changes one of the Cecil AQRE terms to its dual, effectively switching only the quantifier in the term. In this case summary analysis and state propagation need not be redone; only comparison of the state sets to the DFSA accepting states is necessary. Case "top-1" changes the main program's summary relation, but not its flowgraph, while "top-2" changes the flowgraph, but not the summary relation. Cases "leaf-1" and "leaf-2" do respectively the same for a frequently called leaf routine in the call graph. In these cases, the front-end, graphing, and resolution subsystem tools must also be rerun, but

	stack	dave	file	matrix	load-io	load-ur
Lines of Code	38	12	18	130	6557	6557
No. of Routines	2	2	4	4	134	142
No. of Cesar Objects	1	4	2	22	67	715
No. of Odin Objects	136	178	381	307	11691	7147
No. of AQRE's	1	2	3	2	3	1
CPU time (sec)	114	143	276	405	11100	19500
Disk space (kB)	161	151	284	750	10612	29148
Cesar (%)	41.2	43.3	43.8	68.1	39.7	81.1
Odin direct (%)	50.9	47.6	46.1	26.9	51.7	15.9
Odin indirect (%)	7.9	9.1	10.1	5.7	8.6	3.0

Fig. 9. Sun-3/260 full-run performance data for Cesar.

	dual	top-1	top-2	leaf-1	leaf-2
Routines affected (/134)	134	1	5	100	1
CPU time (sec)	795	420	1220	6110	255
Cesar (%)	47.0	27.4	28.3	42.6	18.4
Odin direct (%)	53.0	72.6	50.8	46.2	81.6
Odin indirect (%)	0.0	0.0	20.9	11.2	0.0
% of load-io full-run	7.2	3.4	11.0	55.0	2.3

Fig. 10. Sun-3/260 reanalysis performance data for Cesar.

for only some routines. The table summarizes the time required to reanalyze the source code, apportioned between Cesar and Odin, along with the relative number of routines in the call graph affected by the change. Since Odin can detect when an intermediate result is identical to its previous incarnation, the number of routines for which different tools must be rerun will vary. We report the relative percentage of routines for which the final report is regenerated.

Object management time dominates unless the number of Cesar objects becomes large. Overall, the number of tool invocations and Odin objects is proportional to the number of routines in the program and the number of AQRE terms in the Cecil expression. Each AQRE requires a separate invocation of the analysis phase described in Section 4, and a consequent multiplication of the number of data objects generated during that phase.

The number of Cesar objects affects the running time of each analysis phase that occurs. Our prototype implements neither the flowgraph slices nor the different call graphs for summary analysis and state propagation as described in Section 3.4. Rather, we implemented a single flowgraph for each routine, labeled by events acting on all objects in the routine, with a single call graph used for all analysis phases. This strategy increased the size of the flowgraphs that must be analyzed, but decreased the number of Odin objects

	stack	dave	file	matrix	load-io	load-ur
Lines of Code	38	12	18	130	6557	6557
No. of Routines	2	2	4	4	134	142
No. of Odin Objects	62	34	50	50	1416	1458
CPU time (sec)	27	28	28	40	767	823
Disk space (kB)	71	34	51	750	3320	3584
Cesar (%)	44.4	39.1	42.8	57.5	70.9	72.2
Odin direct (%)	55.6	57.2	57.2	42.5	29.1	27.8
Odin indirect (%)	0.0	0.0	0.0	0.0	0.0	0.0

Fig. 11. Sun-3/260 performance data for FORTRAN front-end.

that must be managed, in an attempt to balance the tradeoff between the savings obtained in reanalysis and the reduced object management overhead discussed later in Section 5.3. A proper implementation would treat each flowgraph slice as an independent Odin object and would have dramatically increased the number of Odin objects, and consequently, the object management overhead. Depending on the density of Cesar objects and events in a program, the number of Cesar objects and events acting on them can increase to the point that the data flow algorithms dominate the running time. We see this effect begin to occur in the "load-io" and "matrix" cases and dominate the "load-ur" case of Figure 9. If we had implemented slices, both the CPU time and data disk space consumed by Cesar would be smaller by a factor of the average number of objects accessed per subprogram, but we believe that the Odin object management overhead would have more than erased any gains, at least in total execution speed.

The reanalysis figures are as expected. When a routine at the bottom of the call graph has its summary sequencing effect changed, this alteration can propagate widely up the call graph, potentially changing the effects of a large number of other routines. The "leaf-2" case changed only the flowgraph. This can only alter the location of call sites for routines lower in the call graph. Since the routine is at the bottom, no effects are propagated. This effect also accounts for the comparatively long reanalysis time for case "top-2". The altered flowgraph changed the flowgraph location of call sites for other routines, and this propagates itself down one level in the call graph during state propagation. All routines directly called by the main program, which are numerous in this example, must redo state propagation analysis. Odin then notes that few routines have new state propagation data and continues recomputation only for those few.

Overall, these results show our prototype analysis tool to be slow and storage-consuming. Not all this lack of performance can be attributed to Cesar, however. For comparison purposes, Figure 11 lists the performance data to run only the FORTRAN front end and graphing tools, which should be roughly comparable to the speed of compilation under the same object management environment. The use of Odin has a large impact on the running times, both because of the overhead of Odin's object management and

because of architectural and implementation constraints necessitated by the use of Odin. We discuss the effect of Odin further in Section 5.3. In Section 7 we describe how performance of the Cesar tools themselves could be improved by as much as an order of magnitude or more.

5.3 Object Management

Odin was selected as the object manager and user interface for Cesar because it provides flexibility in environment definition and a partly incremental approach to recomputation in the face of change. It provides these benefits, but not without some drawbacks.

The current implementation of Odin is limited in three major ways. First, Odin stores each data object as a separate operating system file and cannot cache a recently or commonly used data in main memory between execution of tool fragments. The high volume of disk operations slows the computation. Second, to obtain the necessary flexibility to add arbitrary new tools, Odin invokes tools with a Unix shell script, generated for each invocation by macro substitution from a template. The process of generating and invoking a Unix command script, which in turn invokes a number of subprocesses, some of which must individually initialize the Ada run-time system also slows the computation. Third, to determine if an existing object is obsolete, Odin must determine and update dependencies throughout its object store. Odin can be swamped when a request involves the production or update of a large number of objects all derived from a common source object. The savings Odin achieves in avoiding recomputation of unchanged objects must be balanced against these costs. An Odin environment designed to minimize recomputation typically generates many small objects by many tool invocations, while an environment that avoids object management overhead manipulates few large objects with comparatively few tool invocations. Clemm's experience with Odin has shown that the optimum size of an Odin object is about five kilobytes [7].

The problem is exacerbated by the need to dynamically determine dependencies among the data objects generated by Cesar tools. Analysis of any given routine will often require analysis results from called or calling procedures. These relationships cannot be encoded directly into the derivation graph since they depend on the call graph. Additional tools are needed to determine the data dependencies. These tools use call graph information to generate requests that force Odin to derive data appropriately. Besides the computation time needed to generate the data dependencies themselves, these additional tools tend to produce a large number of small objects, increasing object management overhead.

All these problems are reflected in the large and complex derivation graph that captures the relationships among the tools comprising Cesar. The derivation graph specification was a significant part of the development effort. A 565-line specification that defines 66 Odin tools is needed to orchestrate the application of Cesar's 45 tool fragments. Environments with static dependencies among data objects have a much smaller expansion factor and are simpler to integrate with Odin.

6. COMPARISON OF CESAR TO OTHER WORK

As mentioned earlier, data flow anomaly detection tools can be considered sequencing evaluation tools for a fixed sequencing constraint [5, 13, 14, 31]. These tools are based on data flow analysis frameworks where the information propagated is the answer to a simple question, "Does a variable have an anomaly at this vertex?" Only a single bit is required per variable per vertex to answer this question. The mechanism for interprocedural static sequencing evaluation described in this work potentially requires much more space.

Traditional data flow anomalies can be described as event sequences of length two. Thus, for example, a Cecil constraint that can be used to study certain kinds of undefined reference phenomena is

```
{ref, def, undef} [undef] forall def,?* [t]
```

This constraint requires every undefinition event to be followed on all paths by a definition. The DFSA for this regular expression is a simple two-state automaton in which the initial state is also the sole accepting state. In this case and when the quantifier is known (or predetermined as in DAVE), a significant optimization can be made. It is no longer necessary to propagate information on both states throughout the graph. For any universally quantified Cecil expression (where the evaluation comparison is set containment), one need only note that it is possible to be in a nonaccepting state to determine that a violation exists. The DFSA state transitions can be computed with only that Boolean datum if there is only one possible accepting state and one possible nonaccepting state. For the same reason, existentially quantified constraints require knowledge only of whether it is possible to be in an accepting state. Again, the simple DFSA permits us to simplify the data propagated to a single Boolean per object. Earlier data flow analyzers exploited this property. Since Cesar permits specification of longer event sequences, which may have DFSA's with multiple accepting and nonaccepting states, this optimization is in general not possible.

In the special case when the sequence specified is of length two, we could produce a more efficient algorithm based on propagating bit vectors rather than summary relation data. This, however, requires knowledge of the AQRE quantifier during summary analysis as we must know which state is important, as described above. Cesar does not require that knowledge until after both summary analysis and state propagation have been performed. The advantage of this deferral is that the summary analysis and state propagation results need not be recomputed when analysis is performed for the dual constraint, one where only the quantifier is different. As we can see above, reanalysis for the dual constraint takes comparatively little time. This is expected to be a useful feature, since evaluation of the dual constraint for a Cecil AQRE term can often eliminate the necessity for further evaluation to determine executability or nonexecutability of paths. Thus, the higher cost of the initial analysis may often be counterbalanced by the lower cost of later reanalysis.

As an example, consider the Cecil specification for queues given earlier in Figure 1. Suppose a program violates the first AQRE term, so that there is

some flowgraph path on which a creation is not the first queue event. The flowgraph path or paths on which the violation occurs may be unexecutable. The program in this case is correct, although our static analysis says not. If we change the quantifier to exists and reanalyze, we can obtain more information about the possibility that an erroneous sequence of operations may occur during some execution of the program. If the program still violates this altered constraint then every path through the flowgraph violates our constraint, and the failure *must* occur on some execution. If the quantifier is not needed until after summary analysis and state propagation, we save a considerable amount of reanalysis, and obtain the additional information at low cost. Further experimentation will show whether higher efficiency during the initial analysis or during reanalysis of altered specifications is more important.

Howden's functional analysis work [17, 18] is also comparable to Cesar. His original general static analysis paper [16] was the source of our notion to base sequencing analysis on propagation of DFSA states through a flowgraph. He proves that under certain conditions a language generated by a flowgraph can be determined to be a subset of a language defined by a DFSA by examining the set of all label pairs on adjacent edges. This set can be found with a simple and normally efficient algorithm based on depth first search. In such cases, the constraints are sets of sequences of length two. Our experimentation with Cesar supports his claim that sequences of this type will form the bulk of the constraints one will use. Howden's method does not, however, provide support for analyses of constraints that incorporate existential quantification over flowgraph paths as Cesar does. It is unknown whether a concrete implementation of Howden's method for functional analysis in these situations, extended to an interprocedural framework, would be more efficient in practice than the data flow analysis algorithm used by Cesar or whether the increased flexibility and generality of Cesar are worth whatever extra cost might be incurred. Again, only further experimentation will tell. As with bit vector data flow analysis algorithms, it may be possible to incorporate Howden's algorithm, should it prove more efficient, into Cesar's scheme, when the form of the constraints permits.

Werner and Howden have implemented some of these ideas in a sequencing analysis system for COBOL programs [30]. Their system permits user definition of the events, but not the constraints. The sequencing constraint used is a generalization of the traditional data flow analysis constraint to sequences involving initialization, use and finalization operations. These events are inserted into the code at appropriate locations as special comments. This scheme allows analysis of this fixed constraint in a more algorithm-specific manner than Cesar and also permits the analysis algorithm (a variant of that used by DAVE) to account for some unexecutable paths, something of which Cesar is currently incapable.

Howden has recently extended this approach with his *comments analysis* [20]. This notion is based on a thorough and explicit annotation of a program with a special comment language intended for later static analysis. Comments may assert that particular events take place at a given program

location, that certain events must have taken place beforehand, or may make other statements about the control or data flow of the program. With this approach, Howden hopes to avoid the difficulties discussed in Section 3.6 involved with statically recognizing events (including dealing with aliasing, arrays and pointers), in creating algorithm-specific constraints rather than general constraints on abstract data types and in handling infeasible paths. This approach has been implemented and applied to an assembly language real-time control system [19].

Strom and Yemini [25] included a mechanism for enforcement of sequencing constraints directly into the compiler for their programming language, NIL. Their mechanism is rather similar to that of Werner and Howden above in that the constraints were based on the same initialize, use, finalize paradigm and annotations in the program were used to signal state changes. The NIL mechanism, however, was more sophisticated in that it allowed several levels of initialization. A dynamically allocated record, for example, could be unallocated, allocated with various combinations of fields undefined, or be allocated with all fields defined. Finite state machines were constructed by the compiler from the data structure definitions to take this substructure into account. The DFSM states were partially ordered so that the meet operation would select the greatest lower bound of the DFSM states at flowgraph vertices where execution paths come together. The NIL compiler then statically computed this "best possible guaranteed state" during compilation based on the control flow and annotations. The mechanism did not allow user defined or existentially quantified constraints. Procedures were handled by annotations that asserted the sequencing effect of the procedure at a call site rather than by using a computed effect, so that NIL would be independently compilable.

The work on constrained expressions by Avrunin et al. [3, 4] and Dillon et al. [9] takes an entirely different approach to analysis than Cesar. Since this work is aimed at analysis of distributed systems, the methods proposed must handle the iterated shuffle operator which extends the specifiable sequences to the recursively enumerable languages. Thus there is no general algorithm for their evaluation. So far, the work has concentrated on solving systems of inequalities between the numbers of occurrences of specific events in a constrained expression. The analysis and evaluation methods used in Cesar could be used to evaluate a constrained expression representing a program against another expression representing a constraint as long as neither contains the iterated shuffle operator.

7. FUTURE WORK

Our experience with Cesar has shown that general static sequencing evaluation can be performed in an interprocedural setting, and suggests that it can provide a significant addition to a software analysis and evaluation capability. The poor performance, the primitive nature of the user interface provided by Odin, and the current limitation to analysis of FORTRAN make the prototype unsuitable for large scale experimentation with Cesar as a component of a software analysis environment.

7.1 Improvements to Cesar

An important performance improvement would come from implementation of the sliced flowgraphs described in Section 3.4. As noted earlier, the time to analyze a single AQRE term can be reduced at least by the average number of Cesar objects accessed per subprogram. Given that some Cesar objects are accessed in more than one routine, this would be more than a factor of ten in the "load-ur" example above, and of about five in the "load-io" case. Additional work to tune the implementation will also improve performance, though not by such large factors.

A second effective improvement to Cesar would be a more efficient object management system, capable of caching objects in memory, efficiently storing and retrieving objects from disk, and invoking tool components through a more efficient mechanism than command scripts. We can then avoid much of the current I/O overhead as well as that generated by invoking Unix processes and run-time initialization for each tool fragment invocation.

An alternative is to implement Cesar as a monolithic tool that does no object management. We find that unattractive as it makes reanalysis of a slightly altered software system as costly as the initial analysis. Object management systems for software environments are under investigation by other researchers. We shall select an appropriate object management system when one that meets our needs becomes available.

In retrospect, given the use of Odin as the object manager, an intermediate strategy of storing all the slices from a single routine in a single Odin object may be a reasonable near-term alternative. While this would not give the full potential savings during reanalysis we might otherwise expect, it would decrease the size of the flowgraphs in the analysis and evaluation phases, improving performance during those stages, and would also eliminate flowgraph vertices and edges that have no effect on the computation, assisting Odin's ability to detect unnecessary recomputation.

The text-based user interface provided by Odin is also a barrier to more widespread use and experimentation. Since Cesar is based on a flowgraph model, a more reasonable interface might display the information obtained by Cesar graphically, using Odin or some other object manager as a data server. We are currently investigating means to make the user interface more friendly and effective with graphical information displays.

Finally, we must support analysis of languages other than FORTRAN. Currently, work is proceeding on the necessary graphing subsystem tools to permit analysis of C and Ada programs. This will give Cesar a wider audience and allow more data to be collected on its usefulness and applicability.

7.2 Mixed Language Program Analysis

Given the language-independent nature of the labeled flowgraphs used by Cesar, it should be possible to evaluate a single program that has components written in different languages [10]. Some of Cesar's current tool fragments are themselves mixed language programs. Ada programs call FORTRAN

subprograms to access data provided by the Toolpack tools. In one sense, Cesar already performs mixed language analysis. An Excess specification can be used to represent the behavior of a subprogram for which no source code is available. During analysis, flowgraph data derived from Excess stubs are indistinguishable from the data derived from FORTRAN subprograms.

An obstacle to a more realistic mixed language analysis capability is the need to use Tepee to recognize the events. Tepee itself is independent of programming language, but the statements that correspond to a particular event may have different syntactic forms in different languages and consequently have different representations in the respective intermediate forms. In Ada, a file is opened by a procedure with four parameters, in C by a function with two, and in FORTRAN the necessary statement is not a subprogram call at all. Different Tepee specifications for each language will be necessary. It should be possible to alter the Odin specification of Cesar to ensure that a program source file written in a particular language is mated with the appropriate Tepee pattern specification during event recognition.

A more difficult problem is that different languages require different actions from the run-time system on invocation and termination of a program. A mixed language program requires a mix of these actions. Thus, Cesar would require a simple abstraction of the concept of a run-time system that is sufficiently general to cover the actions performed by a wide range of programming languages.

7.3 Concurrent Program Evaluation

The sequencing evaluation of concurrent programs has received attention from both static and dynamic analysts. TSL is one example of a dynamic analyzer for event sequencing [21]. Other work has addressed the static detection of data flow anomalies in concurrent programs [5, 28]. The constrained expression work cited earlier is also intended primarily for the analysis of concurrent programs. This concurrent static sequencing analysis is performed either by creating a single flowgraph that reflects all possible interleavings of the atomic actions in the program, or by using special concurrent fork and join nodes to connect flowgraphs for the individual processes. These concurrent flowgraphs require that the fork and join vertices be treated differently from other vertices, complicating the data flow analysis. It is possible to use Cesar in its current state to perform analysis on flowgraphs that reflect all possible interleavings of atomic actions in the concurrent program. It is also possible to modify Cesar to account for these special fork and join nodes. The tradeoff here is between the cost of analysis of potentially combinatorially large, interleaved flowgraphs and the increased cost and complexity of a modified Cesar that treats some flowgraph vertices differently than others. Experimentation can establish which of these choices is most worthwhile.

A second approach to concurrent program evaluation was proposed independently by Apt [1] and Taylor [27]. These methods build a state graph of a concurrent system from which such information as absence of deadlock can be deduced. Taylor and Young propose an integrated static and symbolic

evaluation method to extend the usefulness of this analysis [32, 33]. Since Cesar is based on a graph model of execution, it may prove useful for certain kinds of analyses on these concurrent state graphs.

The form of Cecil expressions is reminiscent of interval formulations of temporal logic in some respects. Thus, a third possible direction in the application of Cesar to concurrent and distributed programs is the application of Cesar to finite state protocol models in the same way that Clarke et al. have applied a data flow analysis algorithm to model checking [6]. The relationship of Cecil to interval logics themselves is also an interesting topic for further research.

7.4 Summary

Cesar provides a flexible, interprocedural, static sequencing evaluating capability that has not previously existed. It allows evaluation of analyst-defined sequencing constraints on analyst-defined events. This prototype system will provide a basis for further development and experimentation in sequencing analysis as an evaluation technique for both sequential and concurrent systems and in the effective integration of different evaluation and analysis techniques. Other issues on the periphery of these topics, such as graphical user interfaces for evaluation tools, can also be investigated.

ACKNOWLEDGMENTS

We thank Geoff Clemm for his help with Odin, Michal Young for his comments on the relationship of Cecil to temporal logic, and the anonymous referees for their many helpful suggestions.

REFERENCES

1. APT, K. R. A static analysis of CSP programs. In *Proceedings of the Workshop on Logics of Programs* (Pittsburgh, Pa., June 6-8, 1983), pp. 1-17.
2. ARLAZAROV, V. L., DINIC, E. A., KRONROD, M. A., AND FARADZEV, I. A. On economical construction of the transitive closure of a directed graph. *Soviet Math. Doklady. 11*, 5 (1970) 1209-1210.
3. AVRUNIN, G. S., BUY, U. A., CORBETT, J. C., DILLON, L. K., AND WILEDEN, J. C. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Trans. Softw. Eng.* To appear.
4. AVRUNIN, G. S., DILLON, L. K., WILEDEN, J. C., AND RIDDLE, W E. Constrained expressions: adding analysis capabilities to design methods for concurrent software systems. *IEEE Trans. Softw. Eng. SE-12*, 2 (Feb 1986), 278-292.
5. BRISTOW, G., DREY, C., EDWARDS, B., AND RIDDLE, W. Anomaly detection in concurrent programs. In *Proceedings of the 4th International Conference on Software Engineering* (Munich, Sept. 17-19, 1979), pp. 265-273.
6. CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8*, 2 (Apr. 1986), 244-263
7. CLEMM, G. M. Personal communication, Mar. 1988.
8. CLEMM, G. M., AND OSTERWEIL, L. J. A mechanism for environment integration. *ACM Trans. Program. Lang Syst. 12* (Jan. 1990), 1-25.

9. DILLON, L. K., AVRUNIN, G. S., AND WILEDEN, J C. Constrained expressions: toward broad applicability of analysis methods for distributed software systems. *ACM Trans. Program. Lang. Syst. 10*, 3 (July 1988), 374–402.
10. EINARSSON, B., AND GENTLEMAN, W. M. Mixed language programming. *Softw Pract. Exper. 14*, 4 (Apr. 1984), 383–396
11. FAIRFIELD, P , AND HENNELL, M. A. Data flow analysis of recursive procedures. *SIGPLAN Notices. 23*, 1 (Jan. 1988), 48–57.
12. FELDMAN, S. I. Make—a program for maintaining computer programs *Softw. Pract. Exper. 9*. 4 (Apr 1979), 255–265
13 FOSDICK, L. D., AND OSTERWEIL, L. J DAVE—a validation, error detection and documentation system for FORTRAN programs *Softw. Pract. Exper. 6*, 4 (Sept. 1976), 473–486.
14 FREUDENBERGER, S. M. On the use of global optimization algorithms for the detection of semantic programming errors. PhD thesis, Courant Inst., New York Univ., 1984
15. GUTTAG, J. V., HORNUNG, E , AND MUSSER, D. R. Abstract data types and software validation. *Commun. ACM 21*, 1 (Jan. 1979), 1048–1064
16. HOWDEN, W E A general model for static analysis. In *Proceedings of the 16th Hawaii International Conference on System Sciences* (Honolulu, Jan. 1983), pp. 163–169.
17. HOWDEN, W. E A functional approach to program testing and analysis. *IEEE Trans. Softw. Eng. SE-12*, 10 (Oct. 1986), 997–1005
18. HOWDEN, W. E. *Functional Program Testing and Analysis*. McGraw-Hill, New York, 1987.
19 HOWDEN, W. E. Validating programs without specifications. In *Proceedings of the 3rd Testing, Analysis, and Verification Symposium* (Key West, Fla., Dec 13–15, 1989), pp 2–9.
20 HOWDEN, W E Comments analysis and programming errors. *IEEE Trans. Softw. Eng. 16* (Jan. 1990), 72–81
21 LUCKHAM, D. C., HELMBOLD, D. P., MELDAL, S., BRYAN, D. L , AND HABERLER, M A Task sequencing language for specifying distributed Ada systems (TSL-1). Tech. Rep CSL-TR-87-334, Computer Systems Laboratory, Stanford Univ., July 1987
22. OLENDER, K. M., AND OSTERWEIL, L. J. Cecil: a sequencing constraint language for automatic static analysis generation *IEEE Trans. Softw Eng. 16*, 3 (Mar 1990), 268–280.
23 OSTERWEIL, L. J Toolpack—an experimental software development environment research project. *IEEE Trans. Softw. Eng. SE-9*, 11 (Nov. 1983), 673–685
24 SHARIR, M., AND PNEULI, A Two approaches to interprocedural data flow analysis. In *Program Flow Analysis*. S S. Muchnick and N. D. Jones, Eds. Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 189–233.
25 STROM, R E., AND YEMINI, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans Softw. Eng. SE-12*, 1 (Jan. 1986), 157–171.
26 TARJAN, R E Fast algorithms for solving path problems *J. ACM. 28* (July 1981), 594–614.
27. TAYLOR, R. N A general-purpose algorithm for analyzing concurrent programs. *Commun. ACM 26*, 5 (May 1983), 362–376.
28 TAYLOR, R N., AND OSTERWEIL, L J Anomaly detection in concurrent software by static data flow analysis. *IEEE Trans. on Softw. Eng. SE-6* (May 1980), 265–277.
29. WEISER, M. Program slicing *IEEE Trans Softw. Eng. SE-10*, 4 (July 1984), 352–357
30 WERNER, L L , AND HOWDEN, W E Fault detection in COBOL programs by means of data usage analysis. Tech. Rep. CS-87-111, Dept. of Computer Science, Univ. of California, San Diego, Dec 1987.
31. WILSON, C., AND OSTERWEIL, L. J. Omega—a data flow analysis tool for the C programming language. *IEEE Trans. Softw Eng. SE-11*, 9 (Sept. 1985), 832–838.
32. YOUNG, M , AND TAYLOR, R. N. Combining static concurrency analysis with symbolic execution. In *Proceedings of the Workshop on Software Testing* (Banff, Alberta, July 15–17, 1986), pp. 170–180
33. YOUNG, M., TAYLOR, R. N., FORESTER, K., AND BRODBECK, D Integrated concurrency analysis in a software development environment. In *Proceedings of the 3rd Testing, Analysis, and Verification Symposium*, (Key West, Fla., Dec. 13–15, 1989), pp 200–209

Received August 1990; revised July 1991; accepted September 1991