Algebraic Specifications and Sequencing: A Defect Detection Method

Kurt M. Olender James M. Bieman

Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
(303) 491-7015, (303) 491-7096
olender@cs.colostate.edu, bieman@cs.colostate.edu

March 20, 1995

Abstract

One class of program defects results from illegal sequences of otherwise legal operations in software implementations. Explicit statement of sequencing constraints, however, is not a common activity when specifying software even when using formal specification methods. This paper shows that constraints on program execution sequences can be derived directly from algebraic specifications. Results include heuristic methods for generating sequencing constraints and a generalization of these methods into automatable rules. The heuristics can be integrated into a specification methodology such as Larch. Engineers can use the generated sequencing constraints to detect sequencing defects in software even before dynamic testing begins. The method can be used to increase the reliability of software that is specified using algebraic methods.

Keywords: Software Defect Detection, Algebraic Specifications, Sequencing Constraints, Static Analysis, Larch

1 Introduction

Demonstrating that software implementations are correct is easier when as much of the correctness checking as possible is done automatically. Unfortunately, a completely automatic verification is impossible; the problem is undecidable.

Some investigators have developed tools (and languages) to support human analysts with the mass of tedious detail inherent in a complete formal verification. Some examples of these tools include Gypsy [1, 9], Larch and the Larch Prover (LP) [8], and the B-Tool (Edinburgh Portable Compilers Ltd) for checking Z specifications. While they address the full range of possible software behaviors and properties, formal proof techniques, even with tool support, require significant computation resources and highly skilled human analysts.

Another approach is to automate checkers for subsets of software behavior whose checking is automatable. Not all properties can be verified with automatic and tractable methods, but for

J. Bieman's research is partially supported by NASA Langley Research Center grant NAG 1-1461, the Colorado Advanced Software Institute (CASI), Storage Technology Inc., and Micro-Motion Inc. CASI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the state of Colorado. CATI promotes advanced technology teaching and research at universities in Colorado for the purpose of economic development.

those that can, no human intervention is required. In general, these properties must be statically visible in the software source code, since dynamic checks cannot confirm the absence of errors. A conservative approach is to use a static check to guarantee that certain errors cannot occur.

Sequencing of events during software execution is one important area where such static checks are possible. Data flow anomalies, which indicate erroneous or error-prone conditions, can be identified through a static check of the ordering of definitions and uses. Unfortunately, many existing automated static analysis tools check only limited kinds of behavior. For example, a tool may check only one constraint such as "in every program, no variable will be used before it is defined."

Some researchers have attempted to address this problem. Dillon *et al.* define *constrained expression* methods that can demonstrate, for example, deadlock sequences in distributed systems [2]. They analyze inequality systems over event occurrence counts in an execution trace. The QDA method of Howden and Wieand allows static detection of interface errors (essentially incorrect pairs of successive events) in software that has been manually annotated with specialized comments [16]. Of immediate interest is the work of Olender and Osterweil [22, 25], who show that static data flow analysis methods can be extended to user-defined constraints on the sequencing of events that should occur during program execution.

However, software systems are not typically specified solely in terms of sequencing. When a formal specification language like those mentioned above is used, one would like to specify the complete range of software behaviors and properties.

It is possible to gain the advantages of both general purpose specification languages and completely automatable checking of important behavior subsets like sequencing. Tools now being developed can automatically extract the sequencing properties from a specification written in a more general language, and then automatically check those sequencing properties in an implementation. Rather than burden the software specifier with a second specification language, the specifier need only learn and use one specification language, and can automatically check an implementation for conformance to desired sequencing properties. In addition, automatic extraction of sequencing constraints from general specifications can optimize the verification process. Rather than use expensive, mathematical methods to verify all details of an implementation, static methods can identify the potentially dangerous program fragments. The more labor intensive formal methods can be applied to verify these "dangerous" portions of the program.

The approach used in this investigation, then, which is similar to Leveson's approach to safety analysis of software [20], is to restrict the focus to an important subset of software behavior that can be more easily, and, in this case, automatically, verified. Leveson seeks methods to demonstrate that certain catastrophic failure conditions cannot occur. This investigation seeks methods to show that certain failure-inducing (or sometimes defect-symptomatic) event sequences do not occur. Even in cases when the ideal software sequencing behavior is not completely statically detectable, consistent, though somewhat more restrictive, constraints can be defined. These constraints can be used, in effect, as defensive programming tactics that can be automatically enforced.

The specific results described in this paper include the development of automatable techniques and heuristics to generate sequencing constraints from algebraic specifications, extending previous work on generating safety constraints to include the generation of liveness and parameterized constraints [26]. Also shown is how the method can be integrated into an algebraic specification methodology using Larch as an example.

2 Background

2.1 Algebraic Specifications and Larch

Formal specifications describe the behavior of systems using mathematical techniques. Several approaches can be used to write formal specifications including abstract modeling [17, 28], algebraic specifications [4, 10, 11], trace specifications [3, 21], and knowledge-based methods [7, 27]. Algebraic specifications are examined in the initial investigation.

Using algebraic specifications, a system is specified primarily as a collection of abstract data types (ADT), each of which includes a specification of the ADT operation syntax and semantics with type signatures and algebraic axioms. Illegal or undefined sequences of operations can either be specified explicitly using "error" or exception values as results in the axioms, or implicitly by omission of an axiom.

Figure 1 shows two example algebraic specifications of a stack ADT. The two examples demonstrate two ways to specify errors. The specification in Figure 1(A) uses the notation from Guttag [11]. In this example, error conditions are explicitly noted in the axioms. The specification in Figure 1(B) is written in LSL, the Larch Shared Language. Error conditions in this version are simply left undefined.

Larch, a "two-tiered" approach for software specification, has been used extensively in the specification research community [13]. A Larch specification has one component written in a *Larch Interface Language* and a second written in the *Larch Shared Language* (LSL). Larch Interface Languages specify communication between program units or between a program and its environment. They are tailored to fit the semantics of particular implementation languages. LSL is an algebraic specification language used to specify mathematical abstractions in a program. Because of their algebraic nature, specifications written in LSL are appropriate for the constraint derivation techniques described in this paper.

Each ADT is specified as a *trait* in LSL. The syntax and semantics of LSL does not differ greatly from the simple algebraic specification language used in Figure 1(A). One difference between the two notations is that type names can be parameterized. See `item` and `stack` in Figure 1(B). Another difference is that exceptions are not specified explicitly in LSL traits. LSL error conditions are implicit—operations that are used in an unspecified manner may result in failures or exceptions. For example, `pop(new)` can result in a failure since it is not specified in Figure 1(B). The sequence of operations `pop(pop(s))` is interpreted by first examining the embedded "`pop(s)`". If `pop(s)` returns stack s', and if s' is non-empty, then the equation `pop(push(s,i)) == s` applied to stack s' specifies the result. If s' is an empty stack, then there is no specification for `pop(s')` and a failure is implicitly specified.

Using Larch, developers specify the functionality of program implementation units using an *interface specification*, which uses a form of pre and post-conditions (the *requires* and *ensures* constructs). Because of the differences between implementation languages, interface specifications are written in interface languages designed for particular languages. For example, LCL is the interface language for C [12]. A different interface language would be used to specify the behavior of C++ programs [5].

In defining the functional behavior of implementation modules, a Larch interface specification references more abstract entities that are specified in LSL. For example, an LCL interface specification of an integer stack can reference the LSL stack trait with the `uses` clause:

```
uses set(int for item, intstack for stack);
```

Thus, `items` in the LSL trait are integers, and the axioms for the trait `stack` should hold for the implementation type `intstack`. LSL specifies the precise mathematical behavior of the entities

operation type signatures

```
    new:  stack
    push:  stack × item → stack
    pop:  stack → stack ∪ {error}
    top:  stack → item ∪ {error}
    isnew:  stack → boolean
```

semantic axioms

```
    isnew(new) = true
    pop(new) = error
    top(new) = error
    isnew(push(s,i)) = false
    pop(push(s,i))= s
    top(push(s,i))= i
```

(A) Specification with explicit error conditions

stack(item, stack): trait
introduces

```
    new:  → stack
    push:  stack, item → stack
    pop:  stack → stack
    top:  stack → item
    isnew:  stack → Bool
```

asserts ∀ `s:  stack, i:  item`

```
    isnew(new) == true
    isnew(push(s,i)) == false
    pop(push(s,i)) == s
    top(push(s,i)) == i
```

(B) LSL specification with parameters and implicit error conditions

Figure 1: Algebraic specifications of a stack

```
eventlist     ::=  event
               |   eventlist ',' event
anchor        ::=  '[' eventlist ']'
quantifier    ::= 'forall' | 'exists'
aqre          ::=  [anchor] quantifier regexp [anchor]
expr          ::=  ['not'] aqre
               |   expr 'or' expr
               |   expr 'and' expr
               |   '(' expr ')'
alphabet      ::=  '{' eventlist '}'
spec          ::=  alphabet expr
               |   spec 'or' spec
               |   spec 'and' spec
               |   '(' spec ')'
```

Figure 2: Syntax of Cecil

referenced by interface specifications. Thus, the interface specification serves as the connection between the implementation and the algebraic specifications

2.2 Sequencing Constraints

A sequencing constraint is an assertion that defines legal sequences of "events" that may occur during any particular program execution. For example, a specification that a variable must be assigned a value before it is referenced requires that an assignment event must occur before any referencing events on all possible event sequences representing all possible program executions.

Cecil is one language for expressing sequencing constraints in software [25]. It allows specification of program sequencing constraints in a way that is amenable to static determination of whether the constraint is satisfied.

A Cecil expression is an anchored, quantified, regular expression (AQRE). Each execution of a program generates a set of sequences or traces of events. Cecil expressions can specify events of interest and their valid traces. A Cecil constraint can be limited to particular subtraces of an execution bounded by *anchor* events, events that mark the start or end of a sequence. An AQRE can also quantify whether all or at least one of the subtraces between these anchors must satisfy the constraint. The grammar for Cecil is given in Figure 2. The *regexp* term is a regular expression. The *expr* non-terminal is provided for convenience. Olender and Osterweil provide a more detailed description of Cecil in reference [25].

Figure 3 is a Cecil constraint that expresses sequencing constraints for a write-only file, as originally given in reference [23]. Events `s` and `t` respectively indicate the start of program execution and its termination, while `open`, `write`, and `close` indicate the execution of the respective operations on some particular file.

Constraints can be separated into two categories: safety constraints and liveness constraints. Generally, any violation of a safety constraint results in program failure. Thus safety constraints must be met on all executions.

Liveness is defined by Lamport as the ability of a program to perform some useful work [18]. A

```
{open, close, write} (
    [s] forall (open; write*; close)* [t]
    and [open] exists ?+ [close] )
```

Figure 3: A Cecil constraint for a write-only file

violation of a liveness constraint does not directly cause program failure, but are often symptomatic of defects such as omitted code. It is usually satisfactory when there exists at least one path between anchors that satisfy liveness constraints.

In Figure 3, the first AQRE term expresses safety properties of the sequencing of file operations. On all executions, files must first be opened, then may be written, and must finally be closed. Violations of this constraint will cause program failure. The AQRE term is read as "Every possible subtrace (`forall`) from program start (`[s]`) to termination (`[t]`) must satisfy the regular expression which specifies the relative sequencing among the alphabet of interesting operations, {`open, close, write`} on write-only files."

The second AQRE term expresses a liveness constraint by specifying that it is possible for the program to perform "useful work" (at least one other event) at least one execution. when a file is opened and subsequently closed. The first AQRE ensures that other event is a `write` operation. The constraint does not require a write on every execution; many programs might produce no output for certain values of the input data. The impossibility of a write operation, however, while not directly causing program failure, should be examined further to determine if code was mistakenly omitted.

Cecil is a powerful language for specifying constraints on the sequencing of events. Given a Cecil constraint, the Cesar system automatically analyzes a program and indicates whether the Cecil constraints are satisfied. Olender and Osterweil describe the Cesar system in detail in reference [24].

3 Using Algebraic Specifications To Find Sequencing Defects

One can generate sequencing constraints directly from algebraic specifications, and then check whether the constraints are satisfied or violated in a software implementation. As with all static analysis methods, a detected constraint violation indicates only a potential defect, since invalid sequences may never actually execute. However, if there are no constraint violations, one can be sure that specific defects do not exist. Thus, a conservative approach is to eliminate all constraint violations. A complete system for generating sequencing constraints and finding sequencing defects requires the use of algebraic specifications, and a connection between specification and implementation objects.

The Larch development method seems well suited for applying this constraint checking approach, since, with Larch, a set of LSL traits are specified in an algebraic fashion, and interface specifications are used to map from implementation to specification entities. A system for finding sequencing defects can be designed to work with Larch using the following components:

1. A set of sequencing constraints for each trait. Sequencing constraints are not part of the Larch method. However, one can generate associated sequencing constraints (in Cecil) by analyzing the type signatures and the axioms of an LSL trait. The method for generating the constraints is the major topic of this paper.

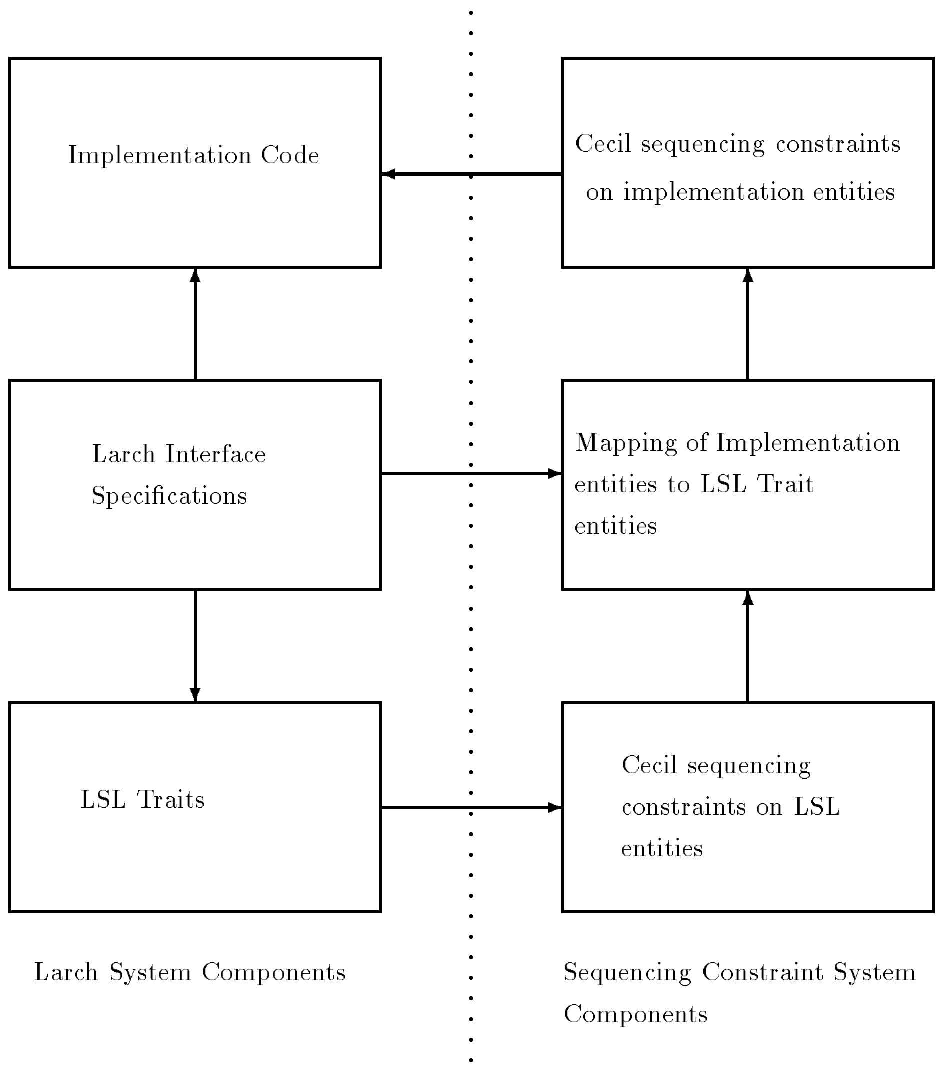
Implementation Code
Cecil sequencing constraints on implementation entities
Larch Interface Specifications
Mapping of Implementation entities to LSL Trait entities
LSL Traits
Cecil sequencing constraints on LSL entities
Larch System Components
Sequencing Constraint System Components

Figure 4: Overall process for generating and checking sequencing constraints.

2. The association between traits and implementation types. This association is defined in Larch interface specifications via the `uses` clause. This association can be used to generate a set of constraints that apply to relevant data types in the implementation.

3. Sequencing constraints for implementation entities. Well known data flow analysis techniques can be used to check code against constraints to indicate potential defects.

Figure 4 depicts the overall process for generating and checking the sequencing constraints. The process starts by generating Cecil constraints directly from LSL traits, which is the focus of this paper. These are constraints on LSL trait entities. Then mappings are derived from implementation entities to LSL entities using Larch interface specifications. Finally, Cecil constraints on implementation entities can be generated from the LSL constraints and the mapping. Thus, in concert with LSL and Larch interface specifications, one can generate sequencing constraints from a specification and check implementations for conformance with the constraints.

The process of generating and checking sequencing constraints can be automated. Prototype tools support portions of this process—the Cesar system can check sequencing constraints against Fortran programs [24]. Constraint checkers for other languages can be readily designed. A system is now being built to implement the heuristics developed in this paper. Thus, one can envision a system that can automatically generate sequencing constraints from a specification, and then analyze an implementation to ensure that there are no sequencing defects.

The first step in generating sequencing constraints is a detailed analysis of the nature of the specified operations.

4 Operation Categories

Assume that semantic equations in an algebraic specification are interpreted as rewrite rules and that the set of equations is sufficiently complete and convergent (thus disallowing an axiom of commutativity, for example). A *sufficiently complete* set of equations includes enough equations to to adequately specify the behavior of an abstract data type. A *convergent* set of equations is one such that, after applying rewrite rules a finite number of times no further rewrite rules can be applied. For now, also assume that no equations are conditional, that is, all rewrite rules are applicable when the form on the left hand side of the rule is present in an expression. Section 5 includes a discussion of the effects of allowing conditions that control the application of a rewrite rule.

The operations in an algebraic specification are partitioned into categories relevant to the analysis using operation type signatures and semantic equations. It is relatively easy to determine the appropriate category for any given operation by defining a set of signature and axiom patterns, one for each operation category.

This classification scheme is similar to those used by others. Guttag [11] partitions operations into constructors, modifiers, and selectors to demonstrate a heuristic for creating sufficiently complete axiomatizations. The Larch Shared Language allows operations to be declared as either generating or partitioning ADT values to aid analysis of Larch specifications [13]. The classifications are intended to assist the generation of sequencing constraints and so may not be identical to other schemes.

Operation Signature and Semantic Patterns. The definitions of operation categories use the following notation for patterns of operation signatures. Let T match the type being defined. Let V match any other type, and let ? match any type at all. Pattern $X \times Y$ matches the cross-product of types matching X and Y in either order. Thus, for pattern purposes, $\times$ can be commutative. The unit or empty type matches the pattern (). Z^* matches zero or more cross-products of types matching Z. The nullary cross-product, a set with one element, is equivalent to the unit type. Square braces represent an optional portion of a signature pattern. The pattern $[?^*\times]T$ means either T or a Cartesian product of one or more types with T. Thus the pattern $[?^*\times]T \rightarrow V$ matches the signature of any function that has at least one argument of the ADT being defined (and possibly arguments of any other types, in any order) and returns a result of some other type. Also assume that $\mathcal{B}$ is the Boolean type, and that T is never $\mathcal{B}$. Boolean types are nearly always directly provided by programming languages, and so need not be specified as an ADT.

The definitions also use patterns on the semantic equations. An identifier matches an operation name. An ellipsis $(\ldots)$ matches zero or more arguments to an operation. Thus, the pattern, $d(\ldots, v, \ldots) = x$ matches an equation defining the result of a single operation with at least one argument.

ADT Subcomponents. The "parts" of an ADT object are its *subcomponents*. For example, a stack ADT is built out of `item` objects, and a tree ADT is built out of subtrees and leaf items. ADT Subcomponents may be of the same type as the ADT itself, e.g., subtrees are trees, or they may be objects of a different type, e.g., stack items. The concept of "subcomponent" helps to classify ADT operations.

4.1 Creator operations.

A *creator* is an operation that initializes a value of the type being defined; it converts arguments of other types into the type of interest. A creator operation has a signature that matches the pattern $V^* \to T$. The `new` operation for stacks is a creator operation. C denotes the set of creator operators for an ADT.

4.2 Modifier operations.

Operations that transform an existing value of an ADT are termed *modifiers*. They have signatures that match the pattern $[?^* \times]T \to T$, converting at least one value of type T into some new value of T. In the stack example, `push` and `pop` are modifiers since they convert an input `stack` into a new `stack` value. M denotes the set of modifiers for an ADT. The union of the creators and modifiers are *constructors* which is denoted by $K = C \cup M$.

4.3 Selector Operations.

The set of *selectors*, S, are the operations that return some subcomponent of a type that is not of that same type itself. Thus, a selector has a signature matching the pattern $[?^* \times]T \to V$. V must be is a subcomponent of T to distinguish between a selector operation and other categories subsequently to be defined. If a semantic equation exists of the form $s(..., k(..., v, ...), ...) = v$, so that a non-ADT value, v, used to construct an ADT value (via constructor k) is directly recovered by the operation s, the type of v is considered a subcomponent of the ADT and s to be a selector. A cursory examination of the semantic equations is sufficient to determine this. In the stack example, `top` is a selector.

4.4 Discriminator Operations.

Functions that take a single argument of T and return a value of some other type and are not selectors are included in the *discriminator* set, D. Often, these operations are used to partition values of the type into equivalence classes based on some intrinsic property of the value. Discriminators match the signature pattern $T \to V$. The `isnew` operation is a discriminator in the stack example.

4.5 Interrogator Operations.

Predicates taking multiple arguments, including at least one of the ADT T in question, and at least one of a subcomponent type (as defined above), are *interrogators*, denoted by I. These operations are normally used to determine the existence or non-existence of some property of the value. The stack example has no interrogator operations, but the membership predicate in the algebraic specification of a set is an interrogator. Interrogator operations match the signature pattern $[?^* \times]V \times T \to \mathcal{B}$, where there exist corresponding constructor and selector operations that add and extract values of type V from T. Again, a cursory examination of the semantic equations can show whether V is a subcomponent of T.

$$\begin{aligned} C &= \{\text{new}\} \\ M &= \{\text{push, pop}\} \\ K &= \{\text{new, push, pop}\} \\ S &= \{\text{top}\} \\ D &= \{\text{isnew}\} \\ I &= \emptyset \end{aligned}$$

Figure 5: Stack ADT operators

Interrogator operations are used to check the state of an ADT value against some parameters, while discriminators cannot include operations that involve parameters other that the ADT value itself. Distinguishing interrogators from discriminators provides greater flexibility in generating constraints.

4.6 Stack ADT operation categories

The categorization of the operations in the stack ADT example are given in Figure 5. These classifications are used to construct the sequencing constraints for the ADT.

5 Generating Constraints

The algebraic specification for a stack in Figure 1(A) clearly states that popping a new stack is an error. The equation returns an exception value rather than a stack. Thus, the sequence `new;pop` directly results in an error. The discussion in this section codifies some heuristics for specifying sequences of operations that cannot result in such errors. These heuristics are expressed as templates for Cecil constraints. An analyst can generate the constraints by manually following the heuristics. A prototype tool to automatically generate the constraints directly from LSL traits is now being developed.

5.1 Constraints to Require Initializations.

The value of a variable of any type must be defined before other operations can be performed on it, so a creator operation must precede any other operation on every execution path. In some cases, the creator operation may not be directly visible in the code, but it must be present. C++, for example, allows the definition of "constructor" operations that implicitly initialize a value of a type defined by a class at the declaration of a C++ variable or constant. Also, some objects may be initialized by the run-time system. For example, the "cin" and"cout" (stdin and stdout) standard I/O streams are not explicitly opened in a C++ program source. These implicit events still occur, however, and the Cesar analysis system accounts for them.

In the following discussion, let C, M, S, I, and D be the set of creators, modifiers, selectors, interrogators, and discriminators, respectively for a particular ADT. Let O be the set of all ADT operations. Lower case c, m, s, i, d, and o represent some specific operation in each respective category.

A Cecil constraint that specifies that a creator operation always precedes any other operation expressions matches Heuristic Template 1:

Heuristic Template 1 $\{O\}$ `[s] forall` $(c_1 \mid c_2 \mid \ldots \mid c_n)$`;?* [`$O \setminus C$`]`

In this template, O is substituted by a comma-separated list of all operations, $O \setminus C$ is the list of all non-creator operations, c_1 through c_n are the creator operations ($C = \{c_1, \ldots, c_n\}$), `?*` is substituted by zero or more operations of any type, and s represents the start of program execution. This Cecil specification requires the first operation on every path from the start of execution leading to a non-creator operation to be a creator. A constraint for the stack example is

```
{new,push,pop,top,isnew}
    [s] forall new;?* [push,pop,top,isnew]
```

The Cesar system can examine the program code that uses the stack implementation to ensure that this constraint is not violated.

5.2 Constraints to Prevent Exceptions.

The semantic equations for the stack indicate that the sequences `new;pop` and `new;top` result in an exception. One can easily generate constraints that prohibit such sequences. However, many other sequences can result in an exception. Applying either `pop` or `top` to an empty stack raises an exception, regardless of how it became empty. The concepts of failure operations, guards, and antiguards are used to develop a Cecil template that can be used to generate appropriate constraints.

Failure Operations. Operations that can return exception values are the *failure* operations. Using the stack example, the set of failure operations, $F = \{$`pop, top`$\}$. Failure occurs only when executing `pop` and `top` with arguments equivalent to `new`. Note that `push` would be a failure operation in a constrained stack.

Guards and Anti-Guards. Failure operations do return valid (non-exception) values when they are immediately preceded by certain operations. For example, the sequence `push;pop` or `push;top` will not produce an error even when one of these sequences is preceded by a `new` operation. The set of operations that may always safely precede a given failure operation, f, are *guards*, denoted $G(f)$. In the stack example, `push` is in both $G($`pop`$)$ and $G($`top`$)$, but there may be other guards as well.

Discriminator and interrogator operations may also be guards. The result of a discriminator (or interrogator) operation, d, might be defined as an equation of the form

```
d(..., g(...), ...)  = v
```

when applied to the result of a known guard, g. Assume that value v above is *characteristic* of the guard call. An "event" `d=v` occurs when a discriminator produces value v. This event is characteristic of the guard call, thus `d=v` is a guard for operation f (`d=v` $\in G(f)$). In the stack example, `isnew(push(s,i)) = false` defines the result of a discriminator applied to the known guard `push`. Thus, `isnew=false` is a guard for `top`.

Consider a partition of d into two events: `d=v` and `d`$\neq$`v` (`isnew=false` and `isnew`$\neq$`false` for the stack `isnew` operation). A static analysis must determine that event `d=v` (or `isnew=false`) occurs in a given program. This problem is undecidable in general. However, intuition suggests that many discriminator (or interrogator) calls appear in conditional expressions of control statements, resulting in constructs such as:

```
if ( d(x) = v ) then ...else ...end if;
```

where x is a variable of the ADT type. The "then" clause is only executed when d is equal to v and so the control path to the "then" is associated with event `d=v`. Events of this form can be found using static analysis. For example, Cesar associates such discriminator events with program control flow edges.

Anti-guards are the complementary events to the guard discriminator (and interrogator) events. Whenever event `d=v` is added to $G(f)$, the complementary operation `d≠v` must be added to the set of anti-guards $A(f)$, since these operations may be characteristic of failures. In the stack example, since event `isnew=false` is a guard for `top` and `pop`, `isnew≠false` is an anti-guard for these operations.

Heuristics for Preventing Exceptions. A suitable safety constraint for a given failure operation is constructed in the following manner. Identify all constructor operations K, all guard operations for the given failure operation, $G(f) = \{g_1, \ldots, g_n\}$, and all *anti-guards* for the failure, $A(f)$. Using these sets, an heuristic would require the failure operation to be preceded on all control paths by a guard with a Cecil constraint that matches the template:

Heuristic Template 2 $\{K, G(f), A(f)\}$ `forall` ($g_1 \mid \ldots \mid g_n$) `[f]`

On all execution sequences that include constructors and other known guards and anti-guards, the failure operation must be immediately preceded by a guard. In other words, no non-guard constructor or anti-guard can occur in an execution sequence between the guard and the failure operation f. Operations not in K, $G(f)$, or $A(f)$ are not relevant to the sequencing as they cannot change the state of an ADT value or otherwise protect against failure.

Applying the Heuristic to the Stack Example. To write appropriate constraints, one must determine the guard and anti-guard sets for each failure operation. Failure operations `pop` and `top` and several guards and antiguards have already been identified. Completion of the analysis requires an examinination of all remaining constructor operations not yet in $G(pop)$ and $G(top)$. Essentially, one must prove that for all values of the ADT as arguments to the constructor, a known discriminator guard returns a value characteristic of its application to a known constructor guard. This is simple in the stack example. The remaining unexamined constructor operation is `pop`. The equations include the counterexample `pop(push(new,x)) = new`. Therefore, `pop` is not a guard.

The final constructor, guard, anti-guard, and failure sets for the stack ADT are:

K = {`new, push, pop`}
F = {`pop, top`}
A(`pop`) = A(`top`) = {`isnew≠false`}
G(`pop`) = G(`top`) = {`push, isnew=false`}

Since two or more failure operations share the same guard sets, their separate constraints will be isomorphic—a property-preserving mapping can convert the constraint for `pop` into the constraint for `top`. Thus, one can define a single AQRE with both failure operations listed in the end anchor. The resulting Cecil constraint for the stack example is:

```
{new,push,pop,isnew=false,isnew≠false}
   forall (push | isnew=false) [pop,top]
```

This means that `push` or `isnew=false` must immediately precede `pop` or `top` in any execution sequence that contains a constructor or discriminator operation call. If the above constraint is satisfied, `pop` and `top` will not produce exception values.

This is a conservative estimate of the sequencing constraint for a stack. An implementation satisfying the constraint cannot fail when `pop` or `top` executes. However, an implementation that does not satisfy the constraint may never or rarely fail because the unsafe paths are never or rarely followed. Typical defensive programming practices require guard operations that protect against possible failure. The Cecil constraint above codifies such a defensive programming practice and the Cesar system allows for its automatic enforcement. Thus, this Cecil constraint is reasonable, though more restrictive than might be required.

6 Liveness Constraints

Certain program anomalies make up a class of *possible* program defects. For example, redundant operations or values that are computed but never used may not cause immediate program failures, but often indicate defects such as omitted code. If there is no error, then the redundant or unused values could be safely removed by an optimizing complier. Cecil "liveness constraints" guard against such anomalous sequences. Liveness constraints can be generated directly from algebraic specifications.

Generally, a "liveness" condition states that some necessary event takes place. The event, however, need not occur during all executions; it is usually satisfactory when there exists at least one possible execution during which the event occurs in the proper context. Using Cecil terms, there must exist a path between anchors that contains the event of interest.

The following discussion examines two different classes of program anomaly—unused and redundant computations, describes the liveness constraints that guard against these anomalies, and demonstrates how to generate such constraints from algebraic specifications.

6.1 Unused Computations

A constraint can require operations that create an ADT value to be followed by an operation that uses the value. For example, operations that produce stack results are the constructors `push`, `pop` and `new`. The operations that reference stacks are `top`, `isnew`, `push`, and `pop`. The following Cecil constraint requires that there be at least one execution path with a stack reference between a stack constructor and the end of the program:

```
{top, isnew, push, pop} [push,pop,new] exists ?+ [t]
```

Similar liveness constraints can be derived from other ADT specifications. The set of constructor operations K have already been defined. Only creator operations C do not have an ADT parameter as an input argument, so the operations that use a stack are in $O \setminus C$. Thus, the following Cecil expression template is a generalization of the unused computation constraint

Heuristic Template 3 $\{O \setminus C\}$ `[`K`] exists ?+ [t]`

Unused computations are also created when a program modifies data object V creating V' and a subsequent change to V' recreates V with no possible intervening use of V'. Such a program might contain a defect. Assuming that the creating of V' has no additional side-effect, either the code that constructs V' is unnecessary, or the code to use V' is missing. As before, if there is even a single execution path between the two state change operations on which a use of V' occurs, a defect is less likely.

The above situation occurs most frequently when an ADT specification contains two operations that are, in some sense, inverses of one another. In the stack example, `push` and `pop` undo one another's effects. When `pop` removes a value from a stack, the stack is returned to its state just before the most recent `push` operation. One of the `push` axioms directly states this relationship.

To ensure that the `push` operation was necessary, a stack created by a `push` must be referenced on some path before it is popped. As before, all stack operations but `new` use a stack in some way. A Cecil expression for this constraint is

```
{top, isnew, push, pop} [push] exists ?+ [pop]
```

To generalize the technique to arbitrary ADT specifications, one must first identify the potential anomalous sequences from the algebraic specification. In some constructor/modifier the modifier undoes the work of the constructor. As shown in Section 4, an operation is a selector if there existed an equation matching the pattern $s(..., k(..., v, ...), ...) = v$. Operation s in that case returned a non-ADT subcomponent of the ADT. When s is instead a modifier that returns an ADT subcomponent of the ADT, the situation is analogous, and there is exactly the relationship between k and s need seek for a liveness constraint. In the stack example, the equation

```
top(push(s,i)) = s
```

indicated that `top` was a selector. Similarly, the equation

```
pop(push(s,i)) = i
```

indicates that `pop` is a modifier that undoes the effect of a `push`.

Thus, to identify such constructor/modifiers pairs one would examine the specification axioms looking for equations of the form

$$m(\ldots, k(\ldots, t, \ldots), \ldots) = t$$

If a program applies k and then m in exactly this pattern, with no other operation intervening, no useful work is performed. The result of the constructor is immediately undone by the modifier and never used.

For each such pair (k, m), a constraint can require at least one other operation that uses the ADT value on some execution path between the constructor and modifier. Such a constraint would match the Cecil expression template

Heuristic Template 4 $\{O \setminus C\}$ `[`k`] exists ?+ [`m`]`

6.2 Redundant Computations

A computation that is repeated even though the earlier value is still available may be a redundant computation. Such an occurrence may indicate an error. Either the redundant computation is unnecessary or the code to change the earlier value is missing. For example, it is reasonable to expect that on at least one path between two `top` operations that some operation may change the state of the stack. Otherwise, the value computed by the first `top` operation can be used and the second `top` operation is unnecessary. (Of course, for a stack, it may require fewer resources to repeatedly use the `top` operation rather than to store the value, but in many situations, such redundant computation is at least suspicious.) A Cecil constraint can specify that a state change must be possible between any two `top` operations:

```
{push, pop, new} [top] exists ?+ [top]
```

One of the earlier safety constraints would ensure that the last constructor before the second `top` is not `isnew`. Here the focus is on preventing (or identifying) redundant operations.

A similar constraint can be specified to prevent redundant `isnew` operations:

```
{push, pop, new} [isnew] exists ?+ [isnew]
```

Olender and Osterweil showed that such constraints can sometimes be inefficient to analyze as a separate invocation of Cesar for each individual location of an event in the start anchor is required [25]. Equivalent expressions that avoid this difficulty are

```
{push, pop, new, top}
    [s] exists ?*; (push | pop | new) [top]

{push, pop, new, isnew}
    [s] exists ?*; (push | pop | new) [isnew]
```

Non-constructors other than `top` are ignored by removing them from the alphabet and requiring that the `top` operation be immediately preceded only by a constructor.

Selector, discriminator and constructor operations can be used to generate similar constraints from other specifications. Let $K = \{k_1, \ldots, k_n\}$ be the constructors and let o be either a discriminator or a selector with only the ADT as an argument. Then for each such o a Cecil constraint matches the template

Heuristic Template 5 $\{K,\ o\}$ `exists` $(k_1 \mid \ldots \mid k_n)$ $[o]$

7 Parameterized Events

In this section, methods described in Section 5 and Section 6 are used to derive sequencing constraints from an example specification. The example specification is written in the Larch Shared Language (LSL), which is described in Section 2.1. The method is demonstrated, and parameterized events that enhance the ability to specify and check safety and liveness constraints are introduced.

7.1 A Table Specification in Larch

The example table ADT trait specification from the LSL documentation [13] helps demonstrate the approach for dealing with parameterized events. Figure 6 shows the LSL trait for ADT table, with the infix $\in$ operator replaced by the prefix operator `in` for convenience in writing a Cecil specification. There is no semantic difference.

A table object (of type `Tab`) is initialized using the `new` operation, which creates an empty table. Table entries, `(Ind, Val)` pairs, are added via the `add` operation. The `lookup` operation returns the `Val` component of an entry associated with a particular `Ind`. The `isEmpty` determines whether the table has any entries, and the `size` operation returns the number of entries in the table. The `in` operator determines whether a particular `Ind` is an index for some table entry. The table trait in the LSL documentation [13] does not include a "remove" operation. The inclusion of such an operation would increase the complexity of the example.

```
Table: trait
introduces
    new: → Tab
    add: Tab, Ind, Val → Tab
    in: Ind, Tab → Bool
    lookup: Tab, Ind → Val
    isEmpty: Tab → Bool
    size: Tab → Card
asserts ∀i, i' :Ind, val:Val, t:Tab
    lookup(add(t, i, val), i') ==
        if i = i' then val else lookup(t, i')
    ¬ in(i,new)
    in(i,add(t, i', val)) == i = i' ∨  in(i, t)
    size(new) == 0
    size(add(t, i, val)) ==
        if in(i, t) then size(t) else size(t)+1
    isEmpty(t) == size(t) = 0
```

Figure 6: An LSL specification of a table

7.2 Table Operation Categories

The procedure described in Section 4 generate the safety constraints of Section 5.

Figure 7 classifies the table operators into creators (C), modifiers (M), selectors (S), discriminators (D), and interrogators (I) as defined in Section 4. From this classification, the constraints are derived.

7.3 Requiring Table Initializations

Using the procedure in Section 5.1, one can generate a constraint to force a creator to precede other table operations. Since `new` is the sole creator, the following Cecil expression is derived

```
{new,add,lookup,isEmpty,size,in}
    [s] forall new;?* [add,lookup,isEmpty,size,in]
```

$$\begin{aligned}
C &= \{\texttt{new}\} \\
M &= \{\texttt{add}\} \\
K &= \{\texttt{new, add}\} \\
S &= \{\text{lookup}\} \\
D &= \{\texttt{isEmpty, size}\} \\
I &= \{\texttt{in}\}
\end{aligned}$$

Figure 7: Table ADT operators

To satisfy this constraint a `new` operation must precede any other operation.

7.4 Preventing Table Exceptions

The process for preventing exceptions is to identify the failure operations, the failure guards, and then derive a Cecil constraint.

One difference between LSL and the specification language of Figure 1 is that exceptions are usually not explicitly defined via axioms in LSL, so the axiomatization may not be sufficiently complete. Exceptions or failures result when operations are applied in undefined ways.

Failure operations are identified by determining whether the behavior of any operation is defined via **asserts** equations for all inputs that fit the type signatures in the **introduces** component of the specification. This process is not difficult; Guttag and others describe heuristics to create a set of sufficiently complete semantic equations [11]. The assumption is that any equation missing (according to Guttag's heuristics) from the Larch specification raises an exception when the associated sequence of operations is executed. From examining the **asserts**, the only operation that may fail is `lookup`. Thus, for the table ADT, $F = \{$`lookup`$\}$.

Finding the guards for `lookup`, G(`lookup`) requires examining each semantic equation that includes `lookup` to identify those that may produce a valid result. The process is complicated because Larch allows conditional results of an **assert**:

$$\texttt{lookup(add(}t, i, val\texttt{), } i'\texttt{) ==}$$
$$\textbf{if } i = i' \textbf{ then } val \textbf{ else } \texttt{lookup(}t, i'\texttt{)}$$

This equation is guaranteed to give a valid result when the operation executed before `lookup` was `add`, and the same `Ind` value was an argument to both operations. The condition in the equation result gives an additional condition that a guard must satisfy to ensure safety. Since Cecil makes an implicit assumption that each instance of an ADT variable must independently satisfy the constraint, this event is characterized as `add(i)` and is included as a guard for event `lookup(i)`. That is, `add` is an effective guard only when `lookup` and `add` use the same index value as an argument.

Other possible guards are found by examining the discriminator and interrogator operations to determine whether any of these return a value characteristic of the guard `add(i)`. Any such operations must have an `Ind` argument to allow for the comparison of index values, since the only known guards are contingent on an `Ind` argument.

None of the discriminators give a characteristic value since they do not have `Ind` arguments. The interrogator `in` does give a characteristic value, however; it returns `true` when its `Ind` argument is an argument to a prior `add` operation. Thus, `in` is potentially characteristic of the guard `add`. More precisely, for `Tab t` and `Ind i`, event `in(i)=true` is characteristic of guard `add(i)`. There are no other suitable characteristic interrogators, so the final event sets are

$$
\begin{aligned}
K &= \{\texttt{new, add}\}\\
F &= \{\texttt{lookup(i)}\}\\
G(\texttt{lookup(i)}) &= \{\texttt{add(i), in(i)=true}\}\\
A(\texttt{lookup(i)}) &= \{\texttt{in(i)}\neq\texttt{true}\}
\end{aligned}
$$

The Cecil constraint becomes

$$\{\texttt{new,add(i),in(i)=true,in(i)}\neq\texttt{true}\}$$
$$\texttt{forall (add(i)}|\texttt{in(i)=true) [lookup(i)]}$$

The constraint states that on all possible execution sequences, the `lookup` of a given index must be preceded with either an `add` or an `in` with the same index as argument. The parameter `i` for the index value has a scope that includes the entire expression.

7.5 Parameterized Liveness Constraints

Constraints with parameters can also identify potential program anomalies. Redundant operations may involve interrogator operations and selector operations with arguments in addition to an ADT argument. The example table ADT LSL specification can be used to demonstrate liveness constraints which use parameters.

Cecil expressions that require a path with a state change between two `in` operations or two `lookup` operations that use the same index are

```
{new, add(i), in(i)}
    [s] exists ?*;(new | add(i)) [in(i)]

{new, add(i), lookup(i)}
    [s] exists ?*;(new | add(i)) [lookup(i)]
```

The enforcement mechanism will consider two arguments to `in` or `lookup` to be the same only if there is a path between the operations in which the argument must be the same, and is not modified by any other operation.

7.6 Conditional Semantic Equations

The previous example includes conditional equations in algebraic specifications. The conditions simply form additional constraints on the sequencing. Typical conditions are equality or inequality of arguments as above, or are the results of discriminator or interrogator calls. In the first case, the condition is reflected in the final sequencing constraint by parameterized events. Otherwise, the appropriate discriminator or interrogator guard is inserted into the required sequence in the Cecil constraint.

7.7 Extensions to Cesar

The Cecil constraint for the table ADT clearly restricts the occurrence of failures or anomalous sequences resulting from the execution of operations with particular indices. Expressing and checking these constraints requires extending the earlier definition of Cecil [25]. The current implementation of Cesar needs to be upgraded to check these parameterized Cecil events [24].

Adding parameters to the events is a natural extension to Cecil, but impose difficulties as the events may not always be statically recognizable. In general, one cannot statically know that the arguments to two different operations have the same value. One can, however, gather that information in some cases, using a safe approximation that, as with Cecil constraints in general, can be used to guide defensive programming practices. It is clear, for example, that the values are the same when the arguments are the same variable and there is a definition-free path from the guard to the failure operation. Static analysis might sometimes report an error that does not exist, but as mentioned before, defensive programming standards often require that all failure operations, such as `lookup`, be appropriately guarded. Under many circumstances, these guards will use the same variable name. Cesar can provide a mechanism to enforce such defensive programming practices. In the table example, one can speculate that the most common occurrence of a lookup call would be in a construct such as

```
if (in(i,t))
then ...; v = lookup(i,t); ...
else ...
end if;
```

Static analysis similar to constant propagation, which tracks values through sequences of assignments, allows some relaxation of the need for consistent variable names in the guard and operation.

8 Conclusions

A set of heuristics can be used to generate Cecil sequencing constraints from algebraic specifications. These heuristics can identify operations that may result in software failures, and specify the sequences of events that guarantee that the failure will not occur. A program can be statically analyzed to determine whether the sequences of operations in the implementation satisfy the constraints. Thus, certain classes of program errors can be ruled out without running the program.

The analysis of sample algebraic specifications resulted in some sequencing constraints that are not currently supported by the Cecil/Cesar system. These constraints involve additional parameters for operator arguments. A proposed extension to Cecil will include these parameter values in the constraint expressions, and extensions to Cesar can check the new constraints.

The process of extracting the inherent sequencing constraints in algebraic specifications is an effective research approach. Results include the identification of new features to add to a sequencing constraint language to produce more useful tools for software analysis.

The development of a prototype tool to automatically generate sequencing constraints from LSL traits is underway. Mechanical extraction of the sequencing constraints from the algebraic specification of an ADT follows the developed heuristics. Most steps involve only an examination of the physical text of the signatures or semantic equations. In some cases, a theorem proof is necessary to determine constraints. These theorems should be, in general, simple to prove, since ADT interfaces typically provide a set of guard discriminators for operations that may fail. Such guards are needed to program defensively when using the ADT and its operations. Unfortunately, some guard operations may not be identifiable by a theorem prover. When this happens, it makes sense to err on the conservative side and assume the operation is not a guard. In some cases, classification of operations is simplified by explicit denotation of the classes in the specification. The Larch Shared Language, for example, allows the explicit classification of operations as "generating" or "partitioning" some ADT. These LSL classifications correspond only in part to the classes defined in this paper, however. Algebraic specification languages could be enhanced to include more specific annotations that aid the analysis more directly.

The `Table` specification example in Section 7 demonstrates that some sequencing errors depend on sequences of operations that use matching values. To protect against such sequencing errors, Cecil and Cesar must be enhanced to allow for parameterized events. Some of the constructs in the Prosper specification language may be appropriate [19].

In expanding this investigation, other specification techniques will be examined. It should be possible to derive sequencing constraints from abstract model specifications written in languages like Z or VDM-SL [17, 28]. The connections between abstract model specifications and dynamic testing have been explored [6, 14, 15]. The objective is to investigate the generation of statically checkable constraints from these specifications.

References

[1] A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch, and R. E. Wells. Gypsy: a language for specification and implementation of verifiable systems. *SIGPLAN Notices*, 12, March 1977.

[2] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Trans. on Software Engineering*, 1991. to appear.

[3] W. Bartussek and D. L. Parnas. Using traces to write abstract specifications for software modules. Technical Report 77-12, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill, 1977.

[4] J. A. Bergstra, J. Heering, and P. Kling, editors. *Algebraic Specification.* ACM Press, New York, 1989.

[5] Y. Cheong and G. Leavens. A quick overview of Larch/C++. Technical Report TR #93-18, Computer Science Dept., Iowa State University, June 1993.

[6] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. *Proc. Int. Symp. Formal Methods Europe (FME'93): Industrial Strength Formal Methods*, pages 268–284, 1993.

[7] M. S. Feather. Language support for the specification and development of composite systems. *ACM Trans. on Programming Languages and Systems*, 9(2):198–234, April 1987.

[8] S. J. Garland and J. V. Guttag. A guide to LP, the Larch prover. Technical Report SRC Research Report 82, DECSRC, December 1991.

[9] D. I. Good, B. L. Divito, and M. K. Smith. Using the Gypsy methodology. Technical Report CLI-2, Computational Logic, Inc., January 1988.

[10] J. V. Guttag, E. Horning, and D. R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21:1048–1064, January 1979.

[11] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[12] J. V. Guttag and J. J. Horning. Introduction to LCL, a Larch/C interface Language. Technical Report 74, Digital Systems Research Center, July 1991.

[13] J. V. Guttag, J. J. Horning, and Andrés Modet. Report on the Larch Shared Language: Version 2.3. Technical Report 58, Digital Systems Research Center, April 1990.

[14] J. Hagar and J.M. Bieman. Adding formal specifications to a proven V&V process for system-critical flight software. *Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, April 1995. (to appear).

[15] P A V Hall. Relationship between specifications and testing. *Information and Software Technology*, 33(1):47–52, January 1991.

[16] W. E. Howden and B. Wieand. QDA—A method for systematic informal program analysis. *IEEE Trans. on Software Engineering*, 20(6):445–462, June 1994.

[17] C. B. Jones. *Systematic Software Development Using VDM.* Computer Science Series. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1990.

[18] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3:125–143, March 1977.

[19] J. Leszczyłowski and J. Bieman. PROSPER: a language for specification by prototyping. *Computer Languages*, 14(3):165–180, 1989.

[20] N. G. Leveson. Software safety in embedded computer systems. *Communications of the ACM*, 34(2):34–46, February 1991.

[21] J. McLean. A formal method for the abstract specification of software. *Journal of the ACM*, 31:600–627, July 1984.

[22] K. Olender and L. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Trans. on Software Engineering and Methodology*, 1(1):21–52, January 1992.

[23] K. M. Olender and L. J. Osterweil. Specification and static evaluation of sequencing constraints in software. In *Proc. of the Workshop on Software Testing*, pages 2–9, July 1986.

[24] K. M. Olender and L. J. Osterweil. Cesar: A static sequencing constraint analyzer. In *Proc. of the 3rd Symp. on Software Testing, Analysis, and Verification*, December 1989.

[25] K. M. Olender and L. J. Osterweil. Cecil: a sequencing constraint language for automatic static analysis generation. *IEEE Trans. on Software Engineering*, 16(3):66–74, March 1990.

[26] K.M. Olender and J.M. Bieman. Using algebraic specifications to find sequencing defects. *Proc. Int. Symp. on Software Reliability Engineering*, pages 226–232, November 1993.

[27] D. R. Smith, G. B. Kotik, and S. J. Westfold. Research on knowledge-based software environments at Kestrel Institute. *IEEE Trans. on Software Engineering*, SE-11(11):1278–1295, November 1985.

[28] J. M. Spivey. *The Z Notation: A Reference Manual.* Computer Science Series. Prentice-Hall, Englewood Cliffs, NJ, 1989.