Using Algebraic Specifications To Find Sequencing Defects

Kurt M. Olender James M. Bieman

Department of Computer Science
Colorado State University
Fort Collins, CO 80523

Abstract

One class of program defect results from illegal sequences of otherwise legal operations in software implementations. Expressions that specify the correct sequences can be written in the Cecil sequencing constraint language. Programs can then be checked at compile time by the Cesar analysis system.

Explicit statement of sequencing constraints, however, is not a common activity when specifying software even when using formal specification methods. In this paper, we describe methods to derive constraints on program execution sequences from algebraic specifications. We provide heuristic methods for generating these constraints from the specifications and generalize the methods into automatable rules. Using these generated constraints, we can then detect sequencing defects in software before dynamic testing begins.

1 Introduction

If we hope to produce reliable software, we must have practical techniques to ensure that software performs its intended function. We can partition this general problem into three components: (1) creating complete and self-consistent specifications that accurately reflect the needs of the end users of the software, (2) producing implementations consistent with those specifications, and (3) verifying that the implementation is indeed consistent.

In this work, we attack portions of the first and third problems. These jobs are easier when as much of the checking as possible is done automatically from a formal specification. Unfortunately, a completely automatic verification is impossible; the problem is undecidable. However, we can restrict the kinds of behavior or properties to be checked to those that are completely automatable. Not all behaviors or properties can be verified with automatic and tractible methods, but for those that can, no human intervention is required. Static, compile-time methods that detect data flow anomalies or perform automatic type checking are examples. Many of the existing automated static analysis tools, however, can check only limited kinds of behavior. Often, the behaviors checked apply to all programs, and are built into the tool. Data flow anomalies, for example, translate to sequencing constraints like "in every program, no variable will be used before it is defined". Tools that verify these properties generally check those and only those properties.

Olender and Osterweil[1, 2] show that static data flow analysis methods can be extended so that user-defined constraints can be automatically checked. These constraints restrict the relative sequencing of computational events that occur during program execution. For example, we might want to place constraints on the order in which interface routines of a new module might be called. These are certainly not part of the programming language, and so a tool built to look for "canned" sequencing problems would not be able to check it.

While such constraints on operation sequencing are useful, software systems are not typically specified solely in terms of sequencing, but rather as informal natural language requirements, or more frequently than in the past, with semi-formal and formal specification languages. These specifications define much more comprehensive behavior than the order in which computations occur. A tool capable of automatically checking sequencing properties will be more convenient and useful if it can also automatically extract the sequencing properties from a specification written in a more general language. The specifier need only learn one language, but can still perform automated checking of a program for its conformance to at least a subset of the intended behavior.

Like Leveson's approach to safety analysis of software [3], we restrict our focus to an important subset of software behavior that can be more easily (and in our case automatically) verified with confidence. Leveson seeks methods to demonstrate that certain catastrophic failure conditions cannot occur. We seek methods to show that certain failure-inducing (or sometimes defect-symptomatic) sequences of computational events do not occur. Rather than burden the software specifier with a second specification language, we aim to automatically construct the appropriate sequencing constraints from an existing specification.

Thus, while the correct sequencing of events is only one component of a specification, it is a component that can be checked by machine. The focus of this paper is the generation of sequencing constraints from formal algebraic specifications, which can then be used to automatically find defects in the sequencing of computations at compile time.

1.1 Algebraic Specifications

Formal specifications describe the behavior of systems using mathematical techniques, with several approaches including abstract modeling [4, 5], algebraic specifications [6, 7], trace specifications [8, 9], and knowledge-based methods [10, 11]. Algebraic specifications were selected in our initial investigation because languages that support them, such as Larch [12], are widely known and are well-supported by tools. In addition, the equational nature of algebraic specifications appears to be well suited for our purposes.

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. In this paper, we assume that error conditions are explicitly noted in the axioms, so that the axiomatization is sufficiently complete.

Figure 1 is an example specification of a stack ADT. For simplicity, we use the notation from Guttag [7]. For additional information on algebraic specification techniques, see [13].

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
```

Figure 1: An algebraic specification of a stack

1.2 Cecil

Cecil is a language for expressing sequencing constraints in software. A sequencing constraint is an assertion that defines legal sequences of particular computations, or "events", that may occur during some execution of a program. For example, if we are concerned that variables are assigned values before those values are used, then the events are assignments to and uses of that variable. If we are instead concerned with operations on stacks, then those events are the code fragments that implement the operations. A sequencing constraint will restrict the relative order in which these events can occur, for example, that assignment events for a given variable always precede use events in every possible execution of the program. Cecil allows us to specify such program sequencing constraints in a way amenable to static checking.

We can view the execution of a program as if it generates a sequence, or *trace*, of the events in which we are interested. Cecil allows us to specify the events and write regular expressions that the traces of these events must match. Since we may be interested only in certain segments of a trace, Cecil also allows us to specify bounding events, or *anchors*, that delimit the subtraces of interest. Lastly, Cecil allows us to constrain either all, or perhaps only some of the bounded subtraces. So our constraints will be independent of the programming language used to code a program, the mapping of the event names to program statements is specified separately. A detailed exposition of Cecil is outside the scope of this paper. We briefly illustrate the important properties with a simple example. A detailed description of Cecil can be found in in [2].

Figure 2 is a Cecil sequencing constraint for a write-only file. The events `open`, `write`, and `close` indicate the execution of the respective operations on some particular file. We are also interested in the start and

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

Figure 2: A Cecil constraint for a write-only file

termination of program execution, which are denoted by the events `s` and `t` respectively. The constraint in the figure expresses a safety property of file operations, since a violation of the sequencing will cause program failure. Namely, we must ensure that files are opened before they are written, and closed when we are finished with them. We can read the expression as: every possible subtrace (`forall`) from program start (`[s]`) to termination (`[t]`) must satisfy the the regular expression as given. Again, the mapping of the event names (e.g. `open`) to program statements would be specified separately.

Cecil is a powerful language for specifying constraints on the sequencing of computational events. The Cesar system can be used to enforce these Cecil constraints. Given a Cecil constraint, the Cesar system automatically determines if a given program satisfies the constraint. Currently Cesar can check Cecil constraints in FORTRAN programs, although other languages can also be supported. The Cesar system is described in detail in [1].

1.3 Organization of the Paper

In the remainder of this paper, we address the problem of generating Cecil sequencing constraints from algebraic specifications. We develop heuristic methods for generating Cecil constraints and generalize these methods into automatable rules when possible.

Section 2 defines categories for operations in an algebraic specification. Heuristics for generating safety constraints that guard against those event sequences that cause program failure are described in Section 3. To simplify our presentation, we describe our method in terms of a simple example, a stack data abstraction. The technique can, however, be applied to more complex problems. Our conclusions and plans for future work are in Section 4.

2 Operation Categories

We assume that equations are interpreted as rewrite rules and that the set of equations is sufficiently complete and convergent (thus disallowing an axiom of commutativity, for example). We 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.

We partition the operations in an algebraic specification into relevant categories based on their type signatures and semantic equations. It is relatively easy to determine the appropriate category for any given operation. We define a set of signature and axiom patterns, one for each operation category.

This classification scheme is similar to those used by others. Guttag [7] partitions operations into constructors, modifiers, and selectors to demonstrate a method 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 [12]. Our classifications are intended to assist the generation of sequencing constraints and so may not be identical to other schemes.

In defining the operation categories, we 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, we allow $\times$ to 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 is equivalent to the unit type. Square braces represent an optional portion of a signature pattern. 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. We 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.

We 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.

2.1 Creator operations

A *creator* is an operation that generates a value of the type being defined out of whole cloth, that is, it converts arguments of other types into the type of interest. A creator operation has a signature that matches the pattern, $V^* \rightarrow T$. The `new` operation for stacks is a creator operation. We denote the set of creator operators for an ADT by C.

2.2 Modifier operations

Operations that transform an existing value of an ADT are termed *modifiers*. They have signatures that match the pattern, $[?^* \times]T \rightarrow T$, 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. We denote the set of modifiers for an ADT by M. The union of the creators and modifiers are *constructors* which we denote by $K = C \cup M$.

2.3 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, $[?^* \times]T \rightarrow V$. We must know that V 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, we consider the type of v to be 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.

2.4 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, $T \rightarrow V$. The `isnew` operation is a discriminator in the stack example.

2.5 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 $[?^* \times]V \times T \rightarrow \mathcal{B}$, 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.

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

Figure 3: Stack ADT operators

2.6 Stack ADT operation categories

The categorization of the operations in our stack ADT example are given in Figure 3. Given these classifications, we can begin to construct the sequencing constraints for the ADT.

3 Generating Constraints

The algebraic specification for a stack in Figure 1 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. In this section, we codify some heuristics for specifying sequences of operations that cannot result in such errors. Note that for the purposes of denoting sequences of operations, we ignore the arguments to the operations. Assume that in the above sequence `new;pop`, the stack result of `new` is passed by unspecified means as an argument to `pop`. When non-stack arguments are irrelevant for the discussion, we shall omit those as well. For example, the sequence `push; pop` indicates we first push some unspecified value onto a stack, and then pop the same stack.

3.1 Requiring Initializations

The value of an variable of any type must be defined before other operation 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 "cout" and "cin" iostreams 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. We use the lowercase c, m, s, i, d and o to represent some specific operation in each respective category.

Using Cecil, we specify that a creator operation always precedes any other operation with the expression:

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

In this expression, 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 ($C = \{c_1, \ldots, c_n\}$) 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. For the stack example, we write

```
{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 insure that this Cecil expression is not violated.

3.2 Preventing Exceptions

The semantic equations for the stack indicate that the sequences `new;pop` and `new;top` result in an exception. With the (usually) reasonable assumption that exception results are to be avoided, we want to generate constraints that prohibit such sequences. These two sequences are not the only ones that can result in an exception, however. Applying either `pop` or `top` to any stack that happens to be empty (regardless of how it became so) will cause an exception. We must statically identify when these sequences might occur.

Some operations can return exception values. We call these operations *failure* operations and let F be the set of failure operations for ADT T. In the stack example, $F = \{$`pop, top`$\}$. Failure operations are not guaranteed to fail, they are simply those that might fail. We can see in our stack example that the sequences `new;pop` and `new;top` are guaranteed to fail, but that other sequences, such as `push;pop` would not.

Some non-failure operations are *safe* in that when they precede a failure operation, they will never cause failure. For example, the sequence `push;pop` or `push;top` will never produce an error for the unbounded stack in our algebraic specification. We call these safe operations *guards*. We denote the set of guards for failure operation f as $G(f)$. On the other hand, other non-failure operations are unsafe as they always cause a following failure operation to fail. The sequence `new;pop` will always fail since we cannot pop an empty stack. We call these unsafe operations *anti-guards*. The set of anti-guards for failure operation f is denoted $A(f)$. In our stack example, we know that $G($`top`$)$ includes `push` and $A($`top`$)$ includes `new`, but there may be other operations in these sets as well.

A suitable safety constraint for a failure operation f is constructed in the following manner. Given constructor operations K, $G(f) = \{g_1, \ldots, g_n\}$ and $A(f)$, we require f be preceded on all control paths by a guard operation with a Cecil constraint of the form:

$$\{K, G(f), A(f)\}\ \texttt{forall } (g_1 \mid \ldots \mid g_n)\ \texttt{[f]}$$

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

To write appropriate constraints, we obviously must determine the guard and anti-guard sets for each failure operation. In the ensuing discussion, let lowercase f and g represent an arbitrary failure or guard operation, respectively. We examine all of the semantic equations that define the result of f and identify the equations that produce valid results (that is, produce non-exception values). The operations in the ADT argument positions of these equations are guards. For example, `pop` is a failure operation when its argument is the result of `new`. However, when the result of `push` is the argument to `pop` a valid result is specified. Thus `push` is a guard for `pop`, and similarly for `top`.

Suppose an equation defines the result of a discriminator operation, d, when applied to the result of a known guard. (This analysis is also relevant when an interrogator operation is applied to a known guard.) Such an equation might look something like:

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

In the stack example, `isnew(push(s,i)) = false` is an equation defining the result of a discriminator applied to a known guard. We assume that value v above is *characteristic* of the guard call. The call of this discriminator producing a particular value can be defined as an "event", call it `d=v`. Since this event (with result v) is characteristic of the guard call, then `d=v` $\in G(f)$. Thus, `isnew=false` is a guard for `top` in the stack example.

We have partitioned d into two events: `d=v` and `d≠v` (`isnew=false` and `isnew≠false` for the stack `isnew` operation). We must statically determine that event `d=v` (or `isnew=false`) occurs in a given program. While this is in general undecidable, most 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 our ADT type. We know that 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`. Cesar associates such discriminator events with program control flow edges, and can thus statically recognize these events.

The guard discriminator (and interrogator) events are added to the respective sets $G(f)$ so that if $g \in G(f)$ and `d=v` is characteristic of g then `d=v` $\in G(f)$. We must also keep tabs on the complementary events, however, as they will be significant in the sequencing. Thus, whenever we add event `d=v` to $G(f)$, we must add the complementary operation `d≠v` to the set of anti-guards $A(f)$, since these operations may be characteristic of failures.

In the stack example, the equations show that `isnew` is false for the known guard `push`, so we add event `isnew=false` to the G sets for `top` and `pop`, and `isnew≠false` to the corresponding A sets.

To complete the analysis, we examine all remaining constructor operations not yet in $G(f)$. Essentially, we must prove a theorem 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 our stack example. The remaining constructor operation is `pop`. We can easily find the counterexample `pop(push(x,new)) = new` from the equations. 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 = {`isnew≠false`}
G(`pop`) = G(`top`) = {`push, isnew=false`}

We note that when two or more failure operations share the same guard sets, their separate constraints will be isomorphic, and we can merge them into a single AQRE with both failure operations listed in the end anchor. The resulting Cecil constraint for our 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, which is actually a context-free language. However, it is impossible to statically check the sequencing against the proper context-free constraint. 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 necessary. We have presented our method in the context of a simple stack data abstraction, but it is more generally applicable to a rich set6 of specifications.

3.3 Other constraints

The preceding section describes the generation of constraints that are intended as safety conditions. We wish to avoid exceptions during program execution. Thus, we check the order of operations on the data abstraction to ensure that sequences resulting in exceptions cannot occur. In other situations, we may want to verify that particular sequences do occur, at least on some program executions. We characterize these latter constraints as describing *liveness* properties for the software. As one example, we may want to ensure, in the file example from Section 1.2, that there is at least one possible execution in which a `write` event occurs. Otherwise, we may wonder why we bother opening the file!

A second complication is that the correct sequencing of the events may depend on the value of the arguments to an operation. Table data abstractions, for example, require a successful lookup of a key in a table be preceded by a successful insertion of the same key. In these situations, we must parameterize the events in the constraints.

We have studied both the generation of liveness constraints and the situations where parameterized events are needed, but our results are outside the scope of this paper. Interested readers may consult [14].

4 Conclusions

We demonstrate a technique to generate sequencing constraints from algebraic specifications. These constraints can be used to discover execution order defects in software. Using the method, we identify operations that may result in software failures, and specify the event traces that guarantee that the failure will not occur. A program can be statically analyzed to determine if the sequences of operations in the implementation satisfy the constraints. Thus, certain classes of program errors can be ruled out without running the program.

These methods do have certain limitations. Obviously, any static compile time method must be able to find the events of interest at compile time. As we briefly mentioned above in Section 3.3, there are situations where this may be difficult or impossible. However, in many cases, we can design the software to reduce or eliminate these cases. Apart from the advantage of enhancing automatic checking, designing with statically-checked constraints in mind will aid the readability and well-structuredness of the software and promote "defensive programming".

Our method also assumes that the syntactic form of the algebraic specifications is indicative of the semantic meaning of the equations. In general, this may not be true. It is not hard, for example, to write an algebraic specification for a stack that will fool our method. However, we believe that such degenerate specifications run counter to the principles of good software engineering. We desire easily understood code and specifications. This is greatly helped when the syntactic forms and semantic meaning are as closely aligned as possible. Thus, for the same reason that we choose meaningful names for variables and procedures in code, we should choose specifications whose syntax reflect their semantics. Our informal examinations of existing sets of algebraic specifications, such as those distributed with the Larch toolset, appear to bear out this assumption.

We are currently planning the implementation of an automatic constraint generator from algebraic specifications. We also plan to expand our investigation to include other specification techniques, for example abstract model specifications written in languages like Z or VDM's *meta iv* [4, 5].

References

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

[2] 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.

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

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

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

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

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

[8] 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.

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

[10] 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.

[11] 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.

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

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

[14] K. M. Olender and J. M. Bieman. Generating sequencing constraints from algebraic specifications. Technical Report CSU–TR–91–104, Colorado State University, Dept. of Computer Science, February 1991.