A Methodology for Modular Termination Verification

K. Rustan M. Leino
Amazon Web Services
leino@amazon.com

Abstract

This paper describes a specification methodology for the sound modular verification of termination for programs with dynamically bound methods. The basic idea is to group methods into partially ordered clusters and to give precise termination specifications within each cluster. The paper proposes a general methodology where the grouping is specified manually, and a specialized methodology where the grouping is computed modularly from the call graph and some additional specifications.

0. Introduction

An important part of formally verifying programs correct is to prove that they terminate. In a modular setting, such verification is done module by module, proving for each module that it satisfies its specifications, under the assumption that imported modules satisfy theirs. Modularity is necessary for the verification to scale.

The contribution of this paper is to give a clear account of how to write *specifications* that allow modular formal verification to prove termination. I present a general methodology for understanding the issues at hand, as well as a specialized methodology that can be used with low specification overhead in an automatic program verifier. The overarching approach is to syntactically (that is, without theorem proving) determine clusters of mutually recursive methods and then to semantically (that is, using theorem proving) prove termination of the calls within each cluster.

There are two possible, orthogonal causes for non-termination in a program: infinite loops and infinite recursion. The termination specification for one loop is independent of the termination specifications of other loops. Consequently, how one such loop specification is written does not restrict the possible ways of writing specifications for other loops. In contrast, the termination specification for one method has an effect on the termination specifications of the method's callers and callees. At minimum, this requires some organizing principles—a *methodology*—when applied to a large program. This is especially dicey when the program may contain dynamically bound calls, because then it is not obvious which methods may be mutually recursive. In this paper, I address issues of writing termination specifications to prevent infinite recursion.

I consider a setting where a program is broken into modules. The modules declare dependencies among themselves by acyclic *import* declarations. Borrowing nomenclature from object-oriented programming, I'll refer to the code routines in each module as *methods*. In the first several sections of this paper, I'll look at termination and specifications more generally, without concern for modules. In other words, those sections are presented as if the entire program were written in one module. Then, starting in Section 5, I'll transition into how to write specifications in the presence of modules and information hiding.

I assume the termination checking is part of a deductive verifier that is capable of formally verifying assertions in programs. These assertions may come from programmer-supplied **`assert`** statements, but more generally come from proof obligations stipulated by the programming-language definition (like index bounds checks for arrays), proof obligations that arise from programmer-supplied specifications (like method preconditions), and proof obligations that enforce a programming discipline (like a methodology for ensuring termination). There are plenty of deductive verifiers of this kind—Why3 [4], SPARK [1], VeriFast [7], Viper [10], and Dafny [8, 9], to mention but a few.

Andreas Podelski has a long history of advancing the understanding of and state of the art in programming languages, specifications, and program analysis, including important results in the area of program termination. It is therefore an honor for me to contribute this article to the Festschrift volume that celebrates the scientific achievements of Andreas Podelski.

1. Fundamentals of Termination

1.0. Well-founded Orders

The fundamental idea behind termination proofs, introduced in a seminal paper by Floyd [5], comes down to mapping moments of a program execution to values of a *well-founded order*. A well-founded order $\succ$ is an irreflexive partial order that has no infinite descending chains; that is, there is no infinite sequence of values $a_0, a_1, a_2, \ldots$ such that

$$a_0 \succ a_1 \succ a_2 \succ \cdots$$

To prove the absence of infinite recursion, the moments of interest are method call boundaries. If successive moments that extend the call stack are mapped to decreasing values of the well-founded order, then the program is sure to terminate. In other words, such a construction ensures that no execution has an infinite call stack.

(For loop termination, the moments of interest are the program states at the top of each iteration of the same loop activation. But I will continue to ignore the orthogonal concern of specifying and verifying the termination of loops.)

To apply Floyd's fundamental idea, we include in the specification of each method an expression that is evaluated on entry to the method. I'll refer to this expression as a *(termination) metric*. To prove termination, a deductive verifier then generates, for each call, a proof obligation that the value of the callee's metric is smaller than the value of the caller's metric.

Proposition 0.
Let $(\mathcal{A}, \succ_{\mathcal{A}})$ and $(\mathcal{B}, \succ_{\mathcal{B}})$ be well-founded orders where the domains $\mathcal{A}$ and $\mathcal{B}$ are disjoint. Define $\succ$ on $\mathcal{A} \cup \mathcal{B}$ by

$$x \succ y \quad \equiv \quad x \succ_{\mathcal{A}} y \ \lor \ x \succ_{\mathcal{B}} y$$

Then, $(\mathcal{A} \cup \mathcal{B}, \succ)$ is also a well-founded order. This also holds for infinitary unions.

Proposition 1.
Let $(\mathcal{A}, \succ)$ be a well-founded order and let $\top$ denote an element not in $\mathcal{A}$. Define $\mathcal{A}_\top$ to be $\mathcal{A}$ extended with the element $\top$, and define an order $\succ_\top$ on $\mathcal{A}_\top$ by

$$x \succ_\top y \quad \equiv \quad (x = \top \ \land \ y \neq \top) \ \lor \ x \succ y$$

Then, $(\mathcal{A}_\top, \succ_\top)$ is also a well-founded order.

Proposition 2.
Suppose $(\mathcal{A}, \geq)$ is a partial order and $\mathcal{A}$ is a finite set. Let $\succ$ be the irreflexive counterpart of $\geq$. Then $(\mathcal{A}, \succ)$ is well-founded.

1.1. Programs

I consider programs of the following form, where I'm only expanding those grammar productions that are relevant at this time.

```
Program ::= Declaration*
Declaration ::= Type | Method | ...
Method ::= method Id "(" Param*, ")" ":" Type Specification* Body
Param ::= Id ":" Type
Specification ::= Precondition | TerminationMetric | ...
Precondition ::= requires Expr
TerminationMetric ::= decreases Expr ...
Body ::= "{" BodyFragment* "}"
BodyFragment ::= MethodCall | ...
MethodCall ::= Id "(" Expr*, ")"
```

I will omit the method return type if it is not relevant in examples.

1.2. Generating Verification Conditions

For illustration, let's define $\succ$ as the following well-founded order on any integers x and y:

$$x \succ y \quad \equiv \quad x \geq 0 \ \land \ x > y$$

When x is non-negative, $\succ$ is like arithmetic greater-than, and $\succ$ does not determine a relationship when x is negative. Consider the following method (where I'm only showing the control flow and recursive calls):

```
method M(x: int)
  decreases x
```

```
{
  if x ≤ 0
    ...
  else if x is even
    ...
    ⓪  M(x / 2)
    ...
  else
    ...
    ①  M(x - 5)
    ...
}
```

The method is specified with the termination metric `x`, as shown by the **`decreases`** clause. Each call in the method body gives rise to a termination proof obligation. The simplest way to prescribe these proof obligations is to instrument the program with assertions placed just before each call. With reference to the markers in the example program, these are

$$\begin{array}{ll} ⓪ & x \succ x/2 \\ ① & x \succ x-5 \end{array}$$

Applying standard weakest preconditions [3], these assertions give the logical verification conditions

$$\begin{array}{ll} ⓪ & \neg(x \leq 0) \ \wedge \ x \text{ is even} \implies x \succ x/2 \\ ① & \neg(x \leq 0) \ \wedge \ \neg(x \text{ is even}) \implies x \succ x-5 \end{array}$$

Note that the left-hand side of each $\succ$ is the metric of the caller, as evaluated on entry to the caller. The right-hand side of each $\succ$ is the metric of the callee, instantiated with the actual parameters of the call and evaluated at the time of the call.

As any modern verifier can ascertain, these are valid formulas. By simply supplying this termination metric (that is, **`decreases`** `x`) for the method, we thus prove that the program terminates.

From here on, I will take it for granted that each termination proof obligation is generated in the context of the respective call. Therefore, I will mostly just talk about calls from one method to another, leaving implicit the control-flow guards and program changes that occur during the execution of a method body.

1.3. Summary of Well-founded Termination

I've described the general structure of termination proofs: proving that the metrics decrease with each call, or more precisely, that every new activation record placed on the call stack has a metric that is smaller than the previous. The comparisons (in the previous sentence, "smaller") are always done with respect to some well-founded order that is fixed for the entire program. Since a well-founded order has no infinite descending chains, the proof obligations associated with each call imply the termination of the program.

I'm following the approach of uniformly checking termination with *every* call. This makes it simple to prescribe which two metrics should be compared in the well-founded order. An alternative, which Podelski and his colleagues have made good use of, is to consider paths of calls that lead back to the same method [2].

2. Dynamically Bound Methods

Dynamically bound methods are found in object-oriented languages, where a popular name for them is *virtual methods*. In such languages, a class can declare a method that its subclasses can replace. When the method is invoked on an object, the call at run time is redirected to the method implementation that is given for the allocated type of that object.

The details of which method implementation is selected for the redirection is not relevant to this paper. So, I'll abstract over that selection mechanism and simply view virtual methods as having a *set* of implementations. I'll say *virtual method* to refer to the declaration of a dynamically bound method and speak of each of its implementations as an *override*.

I'll write methods as follows:

```
method StaticallyBound(...) ... { ... }
virtual method DynamicallyBound(...) ...
method DynamicallyBound' overrides DynamicallyBound(...) ... { ... }
```

Giving a new name for each override is convenient for this paper, because it will let me refer to each declaration uniquely by the name it introduces. By convention, I will use a name like `M'` as the name for an override of a virtual method `M`. I'll say just "method" to refer to either of these or to a statically bound method.

The redirection performed by a virtual method is akin to a call. That is, it is as if the virtual method had a body that selects one of the overrides and then makes a call to that override. To prove termination, we have to account for this call as well. We could demand that each override have a metric that is smaller than that of the virtual method. However, it is convenient to think of a virtual method and all its overrides as using the same metric. That is what I will do. Formally, one can justify this "stuttering" in various ways; for example, by a lexicographically ordered pair where the second component counts down with every redirection. But the stuttering is also easy to justify by noting that a redirection always immediately follows a call, so the number of stuttering calls is no more than the number of decreasing calls.

3. Adding Structure to the Well-founded Order

In theory, what I've described is all there is to proving termination of a program with dynamically bound methods. In practice, however, it is useful to impose some structure on the well-founded order. We want the specifications to be easy to use and to be abstract enough to support information hiding and program evolution.

3.0. Lexicographic Tuples

An indispensable tool in writing termination specifications is the lexicographically ordered tuple, or *lexicographic tuple* for short. Such a tuple is ordered component by component, with the left-most component being most significant. For example, given three domains $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, each with its own ordering $\succ_{\mathcal{A}}$, $\succ_{\mathcal{B}}$, and $\succ_{\mathcal{C}}$, respectively, the lexicographic order $\succ$ on $\mathcal{A} \times \mathcal{B} \times \mathcal{C}$ tuples is defined as

$$\begin{array}{ll} \lfloor a_0, b_0, c_0 \rceil \succ \lfloor a_1, b_1, c_1 \rceil & \equiv \\ \quad (a_0 \succ_{\mathcal{A}} a_1) \ \vee & \\ \quad (a_0 = a_1 \ \wedge \ b_0 \succ_{\mathcal{B}} b_1) \ \vee & \\ \quad (a_0 = a_1 \ \wedge \ b_0 = b_1 \ \wedge \ c_0 \succ_{\mathcal{C}} c_1) & \end{array}$$

I've used $\lfloor \ldots \rceil$ brackets to enclose the list of components of each lexicographic tuple.

Proposition 3.
If the orders for all components of the tuple are well-founded, then so is the lexicographic order of such tuples.

3.1. A Specific Order

I will now be more specific about the well-founded order to be used. The domain of this order will be tuples whose type is

$$\mathcal{S} \times \underbrace{\mathcal{U}_\top \times \mathcal{U}_\top \times \cdots \times \mathcal{U}_\top}_{k}$$

for some fixed k. The domain $\mathcal{S}$ denotes a set of *strata*, which, starting in the next section, will be the main subject of the rest of the paper. The domain $\mathcal{U}$ is the union of all types in the program. For example, $\mathcal{U}$ will contain boolean values, integer values, algebraic-datatype values, pointer values, etc. As we saw in Section 1.0 (Proposition 1), the subscript $\top$ says that $\mathcal{U}$ is extended with a unique top element, $\top$. By defining a well-founded order on each type of the program, we obtain, by Propositions 0 and 1, a well-founded order on $\mathcal{U}_\top$.

Obtaining $\mathcal{U}$ by defining a fixed well-founded order on each type in the programming language has three advantages. It is a simple way to allow all program values to be part of termination metrics. The ordering is consistent for all programs written in the language. And (unlike the situation in many verification systems based on type theory), it saves users from having to define their own orders and proving them to be well-founded, which is a complicated matter best reserved for experts. The approach to using a single, fixed, language-defined order has shown to be effective in Dafny [8].

3.2. Example

To illustrate the use of lexicographic tuples, here are two methods for computing the sum of the triangle numbers of the elements of a sequence. The triangle number of n is the sum of the first n natural numbers (or is 0 if n is negative). I'm writing `|s|` to

denote the length of a sequence `s`. I haven't yet talked about strata, so this example arbitrarily uses the value `Yellow` for the first component of the termination metric.

```
method SumOfTriangles(s: seq<int>, i: int): int
  requires 0 ≤ i ≤ |s|
  decreases Yellow, |s| - i, ⊤
{
  if i = |s|
    return 0
  else
    return Triangle(s, i, 0)
}
method Triangle(s: seq<int>, i: int, j: int): int
  requires 0 ≤ i < |s|
  decreases Yellow, |s| - i, s[i] - j
{
  if j < s[i]
    return j + Triangle(s, i, j + 1)
  else
    return SumOfTriangles(s, i + 1)
}
```

The termination proof of this program goes as follows: The call from `SumOfTriangles` to `Triangle` decreases the third component of the lexicographic tuple, since $\top$ is above every integer. By passing in a larger value for `j` (bounded by `s[i]`), the recursive call of `Triangle` decreases the third component of the lexicographic tuple. Finally, by passing in a larger value for `i` (bounded by `|s|`), the call from `Triangle` to `SumOfTriangles` decreases the second component of the lexicographic tuple.

3.3. Fixed Tuple Length

When I introduced the ordering in Section 3.1, I said it is to have k components of type $\mathcal{U}_\top$. How is this k determined and how can one make sure that all termination metrics of the program use the same tuple length? For example, suppose the two methods of the `SumOfTriangles` example are used in a larger program that uses a larger tuple length. Then, is it necessary to change these termination specifications to make their tuples longer?

To address this concern, we'll employ a simple syntactic rewriting: Each programmer-supplied **`decreases`** clause is implicitly right-padded with $\top$ elements to reach the desired tuple length. For a given program, the number k is picked to be the maximum length of any given **`decreases`** clause. (Since a program is finite, this will pick k to be finite.) Since all tuples will be extended by just $\top$ elements, it is never necessary to know the exact value of k—to compare two **`decreases`** clauses, pad them with enough $\top$ elements to make the clauses the same length, and then perform the lexicographic comparison.

With this syntactic rewriting in place, we can simplify the `SumOfTriangles` example above by removing the explicit mention of $\top$. This turns out to be a natural thing

to do for common programming styles.

Remark: The Dafny language uses the top-element padding just described. In an early version of Dafny, the padding had instead used "bottom" elements. However, a lively discussion I had with Andreas Podelski about these matters came to the conclusion that top-padding corresponds to more common programming patterns. As a result of that discussion, Dafny was changed to use top-padding. Thanks, Andreas!

4. Strata

Many methods in a program are not recursive at all. To a programmer, it may be "obvious" that such methods terminate. It would be nice if we could achieve the same simplicity in the corresponding formal justification. Even when methods are recursive, most pairs of methods in a program are not mutually recursive. For two methods that are not mutually recursive, it would be nice to specify the termination of one independently of the termination specification of the other. This is where we'll get help from the strata $\mathcal{S}$ that I mentioned in Section 3.1. In a nutshell, the idea is to write termination specifications that place methods into layers, or *strata*.[0]

4.0. Grouping Methods by Stratum

Consider the following example, which for each number `x` of an algebraic list computes the factorial of `Twice(x)`. Values of the algebraic (inductive) datatype are ordered by structural inclusion, which is well-founded.

```
datatype List<X> = Nil | Cons(X, List<X>)
method Compute(xs: List<int>): int
  decreases Pink, xs
{
  match xs
  case Nil ⇒ return 0
  case Cons(x, tail) ⇒ return Factorial(Twice(x)) + Compute(tail)
}
method Twice(x: int): int
  decreases Pastel
{ return 2 * x }
method Factorial(x: int): int
  decreases Peach, x
{
  if x ≤ 0
    return 1
  else
    return x * Factorial(x - 1)
}
```

[0] *stra·tum*, pl. *strata*. a) A layer of rock in the ground. b) A group of methods that is partially ordered with respect to other groups of methods.

The termination specifications in this program use three strata, which I called `Pink`, `Pastel`, and `Peach`. If these strata are ordered so that `Pink` is larger than `Pastel` and `Peach`, then we can prove that the program terminates. Notice that, other than the respective stratum components, each method just mentions what is necessary to prove that its own recursion terminates. (It is instructive—perhaps even startling—to try to write integer-valued termination metrics—that is, not using strata or lexicographic tuples—for these methods.)

The strata let us organize the methods into groups. Within each group, the rest of the lexicographic components are used to prove termination. Between groups, all we need to know is some ordering relation between the strata. In other words, if the stratum components of the caller and callee are different, then the proof obligation for termination can be phrased independently of the subsequent k components of the caller and callee metrics. On the other hand, if the caller's and callee's strata are the same, then proving termination comes down to (semantically) proving that the other k components exhibit a lexicographic decrease.

So, we're going to allow a program to declare strata and to order them. The order must be verified to be a partial order (which, on behalf of Proposition 2, gives us the well-founded order we need).

The first component of each termination metric is a stratum. In my use of them, this stratum will be fixed for each method. Therefore, it will be natural to speak of "the stratum of a method". In this way, each stratum acts as a name for a group of methods.

4.1. Automatic Stratification

In what I've described so far, the $1 + k$ components of each metric are supplied (by the programmer) as part of the program text. I will continue to build on this foundation. However, for just this subsection, let's consider how the stratum component of every metric can be inferred automatically by a whole-program analysis.

Given an entire program (or, stated differently, a one-module program), here is one way we can assign strata systematically. It makes use of the program's *call graph*, so let me first define it.

The vertices of a program's call graph are the program's (statically bound, virtual, and overriding) methods. There is an edge from method A to method B if the body of A contains a call to B. Also, if A can redirect to B, then the call graph has an edge from A to B as well as an edge from B to A; by making this edge bidirectional, the automatic stratifications in this paper will end up assigning the same stratum to a virtual method and its overrides.

The strongly connected components (*SCCs*) of a directed graph form a partial order. More precisely, in the nomenclature of graph theory, the condensation of a graph under its SCC equivalence relation yields a directed acyclic quotient graph. The SCCs of the call graph are in one-to-one correspondence with nests of mutually recursive methods (modulo the fact that overrides are always in the same SCC as their corresponding virtual method).

The systematic stratification follows a condensation of the call graph: Introduce one stratum for each SCC. Then, for every edge from an SCC S to an SCC T, order

the stratum of S above the stratum of T; that is, $S \succ T$. Since the SCCs are partially ordered, we obtain a partial order also for the strata.

The resulting stratification has the following property, which is important enough to deserve a name:

> **Sympathetic Stratification.** For any call from a method A to a method B, either $\mathrm{stratum}(A) = \mathrm{stratum}(B)$ or $\mathrm{stratum}(A) \succ \mathrm{stratum}(B)$.

If $\mathrm{stratum}(A) = \mathrm{stratum}(B)$, then a proof of termination needs to consult the remaining lexicographic components of the termination metrics of A and B. If $\mathrm{stratum}(A) \succ \mathrm{stratum}(B)$, then the proof obligation for termination is satisfied on account of the strata alone.

The beauty of a sympathetic stratification is that those are the only two cases. This means that the calls whose termination a programmer may consider "obvious" do not require any further specification or proof from the programmer. Furthermore, since we can perform this stratification automatically, the stratum component of termination metrics can be tacit in **`decreases`** clauses. The strata are used to explain the soundness of the methodology for writing termination specifications, but they need never appear in the programming-language syntax.

Two remarks are in order.

First, it is possible to use fewer strata than this stratification gives rise to. For example, the stratification will give 3 strata for the 3 methods of the `Compute` example in Section 4.0. But since `Twice` and `Factorial` never call each other, those two methods could use the same stratum. There is no benefit to using fewer strata, however. In the other direction, it is not possible to use more strata and still verify the program. The reason is that, if an SCC used more than one stratum, then some call within the SCC will fail the termination proof obligation because the strata do not go in the right direction. In other words, if we use more strata, we will not get a sympathetic stratification. In conclusion, this automatic stratification is as good as we can hope for.

Second, a reminder: the stratification I just described assumes it is possible to compute the call graph of the entire program. Alas, in the presence of modules, information hiding, and modular verification, this is not possible. Next, let us return to having explicit strata in the program and dive into the modularity problem. I'll return to the subject of automatic stratification in Section 8.

5. Modularity

Modularity facilitates organization and abstraction. We'd like termination specifications at module interface boundaries to be expressive enough to be useful for the clients ("importers") of the module, and at the same time be abstract enough to give the module implementation some freedom to evolve.

I'll start this section by reviewing the definition of modular verification. Then, I'll go into key aspects of modules.

5.0. Modular Verification

The *modular verification* of a program consists of the separate verification of each module in the program as well as simple *link-time checks* on the composition of modules. The verification of a module is performed using the contents of the module plus the information it obtains from its imported modules. For this modular reasoning to be well-founded (sound), it is important for the import relation to be acyclic.

The link-time checks are performed on the whole program, but are admissible within the definition of modular verification on the grounds that they are simple enough not to require any logical reasoning. For example, if the programming language separates module interfaces from module implementations, then link-time checks will enforce that every module interface has exactly one implementation and that the program's module import relation is acyclic.

5.1. Module Declarations

To be more precise about how program declarations are partitioned into modules, let me update the grammar:

```
Program ::= Module*
Module ::= module Id "{" Declaration* "}"
Declaration ::= Import | Export | Stratum | Type | Method | ...
```

I will detail `Import`, `Export`, and `Stratum` declarations in the next few sections.

5.2. Imports

If a module `M` wants to mention a name declared in another module `L`, then `M` must first *import* `L`. This is done with an **`import`** declaration. I'll write $\supset$ to denote the transitive closure of the import relation (e.g., `M` $\supset$ `L`) and $\supseteq$ to denote the reflexive closure of $\supset$.

The import ordering of modules in a program must be acyclic. That is, $\supseteq$ must be a partial order.

In typical programming languages, if a module `M` imports a module `L` that defines a name `x`, then this name is written as `x` inside `L` and as `L.x` in `M`. For brevity in this paper, I will omit the qualification "`L.`" and write just `x` for uses of `x` everywhere.

5.3. Exports

To support the good software-engineering practice of information hiding, a module is allowed to restrict which of its declarations are visible to importing modules. In some programming languages, this is done by selectively declaring some of the names defined by the module as "public". In this paper, I will refer to this module interface boundary as the module's *export set*.

An export set controls not only which names importers can see, but also what information about those names is provided to importers. Here, three things are of interest to this paper:

First, whenever the name of a method is exported, so is the signature and specification of the method. However, the body of the method is never exported. In other words, the details of a method's implementation remain private to the module. This allows the module to change the implementation over time without concern of breaking importing modules. A consequence of keeping method bodies private to a module is that it is not possible for importers to build a call graph for the whole program.

Second, the export set allows a module to choose which edges of the import relation are public information. A consequence of allowing imports to be private is that it is not possible for any single module to build the import relation for the whole program. Rather, the acyclicity of the import relation can only be enforced at link time.

Third, an export set can include stratum names separately from stratum ordering information. That is, just because a stratum name is exported does not mean that importers will know anything about the ordering of that stratum with respect to other strata. This lets a module decide which stratum ordering edges to make public.

6. Defining Strata

The declaration of a stratum `C` has the form

```
stratum C below AA above ZZ
```

where `AA` and `ZZ` are lists of previously defined strata. The **`below`** and **`above`** lists define edges in the ordering of strata. In particular, the declaration says that `A` $\succ$ `C` holds for every stratum `A` in the list `AA` and that `C` $\succ$ `Z` holds for every stratum `Z` in the list `ZZ`. I will omit the **`below`** or **`above`** keyword if the list that follows is empty.

The *origin* of a stratum is the module that declares the stratum.

The *upward closure* of a stratum `C` in a module `M` is the set of strata `B` such that `B` $\succ$ `C` is in the transitive closure of the $\succ$ relation induced by the stratum declarations given in `M`.

A stratum `C` in `M` is said to be *in the top tier of `M`* if all strata in the upward closure of `C` in `M` are declared in `M`.

6.0. Ensuring a Partial Order on Strata

Strata are supposed to form a partial order. To make sure that the given ordering clauses indeed define a partial order, a module's stratum declarations are checked, one by one in the order given, to satisfy `A` $\succ$ `Z` for every `A` in `AA` and `Z` in `ZZ`. If this condition holds for all such `A,Z` pairs, then the new stratum is admitted and its ordering information can be used in the checks for the remaining stratum declarations.

To check a condition `A` $\succ$ `Z`, a module is allowed to use stratum-ordering information in the export sets of its imported modules, as well as the stratum-ordering information induced by the module's previously admitted stratum declarations.

Discharging the `A` $\succ$ `Z` proof obligations requires some care. A simple strategy is to introduce the module's strata in a bottom-up fashion. In many cases, this will mean that each stratum declaration will have an empty **`below`** list, which makes the ordering check trivial. The one case where it is necessary to use a **`below`** clause is

when the desire is to define the stratum to be below a stratum declared in another module. In that case, it may be necessary to know about the other stratum's relative ordering information.

A module uses a declaration

```
export C ≻ D
```

to give its importers information about the stratum ordering. The condition in this export declaration is checked once all stratum declarations in the module have been admitted.

6.1. Stratum Ordering Induced by Module Ordering

When information is needed about the stratum ordering, the following inference is allowed:

> **Stratum-Module Axiom.** For any module `M` and strata `C` and `D`,
> `C` $\in$ TopTier`(M)` $\wedge$ `M` $\supset$ origin`(D)` $\implies$ `C` $\succ$ `D`.

That is, strata in the top tier of `M` are above strata declared in modules transitively imported by `M`. In other words, this axiom dictates that strata in the top tier of `M` implicitly be ordered above strata introduced by "earlier" modules.

The program stratum information available in one module is limited. If that information is enough to prove `A` $\succ$ `Z` in one module, then `A` $\succ$ `Z` also holds in light of the additional information available elsewhere in the program. In other words, the modular process I described for admitting stratum declarations results in a partial order for all strata in the program.

7. Composition Requirements

When discharging termination proof obligations in a module, a module has available the stratum-ordering information from the module itself and from (the export sets of) its imports. However, in the presence of method overrides, this is not always enough, even in well-structured programs. Let's take a look through an example.

7.0. Motivating Example

Consider the following program:

```
module Client {
  import Util, Math

  method Process' overrides Process(x: int): int
    decreases Umber
  { ... Sqrt(...) ... }
}
module Util {
```

```
  export Umber, Process
  stratum Umber
  virtual method Process(x: int): int
    decreases Umber
}
module Math {
  export Mauve, Sqrt
  stratum Mauve
  method Sqrt(x: int): int
    decreases Mauve
}
```

The `Client` module uses two other modules, one for various utilities and one for math operations. Since `Process'` is an override for the virtual method `Process`, `Process'` has to use the same **`decreases`** clause as `Process`. The call from `Process'` to `Sqrt` gives rise to the termination proof obligation

`Umber` $\succ$ `Mauve`

Unfortunately, this condition is not provable in module `Client`. Indeed, it may not be true at all, for example if `Math` privately imports `Util` and privately declares `Mauve` to be above `Umber`. If a programmer believes that `Util` and `Math` are independent libraries that do not call each other or override each other's methods, then the programmer is often right. But, of course, we want to be sure.

Note that, if `Util` and `Math` are indeed independent libraries, then it is not reasonable to expect that `Util` would declare or export the information `Umber` $\succ$ `Mauve`. So, a practical methodology for modular verification of termination must allow a module like `Client` to proceed in the absence of knowing `Umber` $\succ$ `Mauve`.

7.1. The Design of Composition Requirements Declarations

As a way out of the quandary above, I propose allowing a module to declare *composition requirements*. Such requirements are assumed when verifying the module. The composition requirements are enforced at link time.

Here is how it works. Module `Client` needs to prove `Umber` $\succ$ `Mauve`. However, if we allowed a composition requirement to mention these strata directly, then the link-time checks would need complete information about the program's strata. That seems like too much information for a link-time check. Indeed, the complete stratum ordering information is similar in information volume to the whole-program call graph. So, we reject this candidate design.

Instead, composition requirements will be on modules. After all, link-time checks already have to check the acyclicity of the import ordering (which is far more sparse than the stratum ordering or any whole-program call graph). So, we'll allow `Client` to declare

`requires` `Util` $\supset$ `Math`

which says that `Client` can be composed with other modules only if `Util` $\supset$ `Math` is possible. More precisely, the role of the composition requirement is to introduce an

edge in the module ordering. If the addition of such an edge leads to a cycle, then the link-time checks will fail and the program will be rejected.

For any stratum `C` mentioned in a module `M`, we have `M` $\supseteq$ origin`(C)`. In our example, we thus have `Math` $\supseteq$ origin`(Mauve)`. The call from `Process'` to `Sqrt` has the proof obligation

`Umber` $\succ$ `Mauve`

This condition follows from the composition requirement `Util` $\supset$ `Math` and the Stratum-Module Axiom, *provided* `Umber` is in the top tier of `Util`.

7.2. Top-tier Exports

To determine if a stratum is in the top tier of a module requires knowing all the stratum declarations in that module. This information is available only in the module itself, so the only way to communicate this information to importers is by another export declaration. For this purpose, we need a new kind of export,

```
export TopTier(C)
```

which can be declared in the module that declares stratum `C`. Like the other stratum-ordering export declarations, this one is checked after all the module's stratum declarations have been admitted.

7.3. The Final Program

Applying what I've discussed in this section, we can verify the example program from Section 7.0 by declaring

```
export TopTier(Umber)
```

in module `Util` and declaring

```
requires Util ⊃ Math
```

in module `Client`.

8. Automatic Stratification

Now that we have a general methodology for writing and verifying termination specifications using explicitly declared strata, I will define a specialized methodology that does not use any explicit strata. Instead, the idea is to generate stratum declarations automatically from module-local call graphs and to replace strata by method names in export information. The automatic stratification will be sympathetic, so the termination proof obligation for a call will either come down to the non-stratum components of **`decreases`** clauses or will hold trivially. Since the stratum component of **`decreases`** clauses will always be handled automatically, programs need only provide the other components.

8.0. Module-local Call Graphs

Akin to what we considered in Section 4.1, we first build a call graph. The call graphs here will be computed locally to module. A *module-local call graph* for a module contains all whole-program call-graph edges that either start or end in the module (plus any call-graph information gleaned from exports, as I will describe below).

8.1. Generating Strata

The algorithm to assign strata to a module's methods is as follows. Order the SCCs of the module-local call graph topologically from bottom to top. Then, process each SCC S in order:

Case 0
If S consists only of external vertices (that is, methods declared outside the module), then the methods in S have already been assigned some stratum. Furthermore, S has no outgoing edges into the current module, because the only edges in that direction are bidirectional and S does not contain internal vertices. So, no action is necessary for S.

Case 1
If S consists only of internal vertices (that is, methods declared inside the module), then introduce a new stratum, say `C`, and place it above the strata of all of S's outgoing edges. That is, generate the declaration **`stratum`** `C` **`above`** `...`. Assign the new stratum to all methods in S.

Case 2
If S consists of both internal and external vertices, then first insist that all external vertices have the same stratum, say `C` (more about this soon). If it is not possible to determine that these vertices all have the same stratum, then reject the module. Next, assign `C` to the methods in S. Finally, for every outgoing edge from S, say to an SCC with stratum `D`, do: If the origin of `D` is the current module, then augment the declaration of `D` to add the clause **`below`** `C`. If the origin of `D` is not the current module, then check `C` $\succ$ `D`; reject the module if this condition cannot be determined.

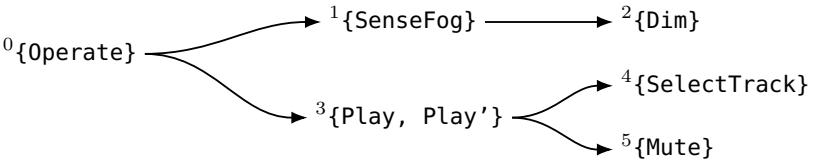
It is worthwhile to note that the third case applies only if the module contains a method override for a virtual method declared in a different module. Without any such override, all call-graph edges are in the same direction as the module ordering, so no SCC would have both internal and external vertices. So, in the absence of any such inter-module overrides, only the first two cases apply, and they satisfy all ordering relations trivially. (For years, this was the situation in Dafny, because of a restriction on the use of traits [0].)

8.2. Example

To illustrate the algorithm, consider the following example modules (where, for brevity, I'm omitting method parameters):

```
module Car {
  export Operate
  import Lights, Radio
  method Operate() { ... SenseFog() ... Play() ... }
  method SenseFog() { ... Dim() ... }
  method Play' overrides Play() { ... Mute() ... SelectTrack() ... }
  method SelectTrack() { ... }
}
module Lights {
  export Dim
  method Dim() { ... }
}
module Radio {
  export Play, Mute
  virtual method Play()
  method Mute() { ... }
}
```

The module-local call graph for module `Car` is



Going bottom-up in this call graph,

- SCC 2 and SCC 5 only have external vertices, so Case 0 applies. I will refer to the strata of those SCCs as `E2` and `E5`, respectively.
- For SCC 1 and SCC 4, Case 1 applies, so we introduce new strata `S1` and `S4` and declare `S1` to be above `E2`.
- For SCC 3, Case 2 applies, so we assign the stratum of `Play`, say `E3`, to `Play'`. Because of its outgoing edge to SCC 4, we augment the definition of `S4` to place it below `E3`. Because of the outgoing edge to SCC 5, we note that we need to check `E3` $\succ$ `E5`.
- Finally, for SCC 0, Case 1 applies, so we introduce stratum `S0` and declare it above `S1` and `E3`.

Had the augmentations in the example led to a stratum with other strata both above and below, then we would need to conduct the `A` $\succ$ `Z` checks mentioned in Section 6.0 to ensure that the strata form a partial order. Luckily, that works out trivially in this example. For SCC 3, I noted that we need to check `E3` $\succ$ `E5`. Unfortunately, there is not enough information to confirm `E3` $\succ$ `E5`, so module `Car` is rejected.

The rejection of module `Car` is sensible, because nothing rules out the possibility that `Mute` would call `Play`, in which case there would be a call cycle among `Mute`, `Play`, and `Play'`.

8.3. Ordering of the Strata of Methods

As the `Car` example reminded us, Case 2 in the algorithm above prescribes some checks on the strata of imported methods. To deal with such checks, it is necessary for modules to share information about the strata of methods. With the explicit strata in Section 6, we had added stratum-ordering exports (Section 6.0). The analogous declaration in the absence of explicit strata is

```
export P > Q
```

where `P` and `Q` are methods. This declaration expresses $\mathrm{stratum}$`(P)` $\succ$ $\mathrm{stratum}$`(Q)`.

With explicitly named strata, it is immediately clear when two methods have the same stratum. In the absence of explicit strata, such information requires an export that reveals the relationship between such methods. For this reason, we'll allow a declaration

```
export P ~ Q
```

which expresses $\mathrm{stratum}$`(P)` = $\mathrm{stratum}$`(Q)`.

As before, these ordering exports need to be checked for each module that declares them. But these two new export declarations play an additional role, which is to *add* edges to the module's call graph. This is necessary if a module wants to force relationships between the strata of its methods, because the module itself may not always contain calls that would add those call-graph edges.

In summary, a module `M` can use these two kinds of **`export`** declarations. They affect the construction of `M`'s module-local call graph by adding an edge from `P` to `Q` (for **`export`** `P > Q`) and adding two edges between `P` and `Q` (for **`export`** `P ~ Q`). The automatic stratification in `M` then uses this module-local call graph, as described in the three cases in Section 8.1. As a final step, the information conveyed by the **`export`** declarations is checked to hold in the module-local call graph.

8.4. Car Example Revisited

To verify module `Car` in Section 8.2, module `Radio` must export more information. In essence, `Radio` must ensure and reveal that `Mute` does not call `Play`. To allow overrides of `Play` to call `Mute`, we add

```
export Play > Mute
```

to module `Radio`. Now, the importing module `Car` has enough information to confirm `E3` $\succ$ `E5`, so that module is now accepted.

As a variation of the example, suppose `Radio` does want to call `Play` from `Mute`. Then, the export declaration above would be rejected, since

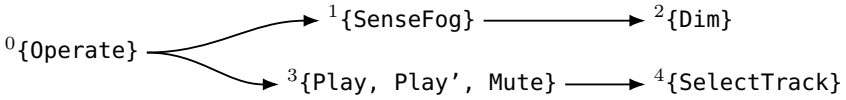
$\mathrm{stratum}$(`Play`) $\succ$ $\mathrm{stratum}$(`Mute`)

would not hold in the call graph for `Radio`. We would then have to remove this export declaration (in which case module `Car`, with its call from `Play'` to `Mute`, would be rejected, as we saw in Section 8.2) or we can instead add

```
export Play ~ Mute
```

With this export declaration, the call graph in `Radio` places `Play` and `Mute` in the same SCC, so these methods need to be given **`decreases`** clauses to enable the termination proof of the call from `Mute` to `Play`. In other words, the termination proof needs to look beyond the stratum component of the termination metric.

Moreover, if `Radio` exports `Play ~ Mute`, then the call graph in `Car` would change to



This, in turn, will cause of the automatic stratification to assign to `Play'` the same stratum as `Play` and `Mute`. Therefore, the termination of the call from `Play'` to `Mute` will also need to consult the **`decreases`** clause. In other words, it is now recognized that the call from `Play'` to `Mute` is a mutually recursive call.

8.5. Top-tier Methods

Lastly, the export declaration for top-tier strata from Section 7.2 needs an analogous form for methods. A direct analog would be to support **`export`** `TopTier(stratum(P))` for any method `P`. However, since most strata *are* top tier, a more frugal alternative is to insist that every exported method *without* a top-tier stratum be exported together with that information. More precisely, if a module `M` exports a method `P` that is either statically bound or virtual (that is, not a method override) and the stratum of `P` is something other than a top-tier stratum of `M`, then this has to be declared in the export set. This will be declared using

```
export M > P
```

And with that, we have a full methodology for writing termination specifications. It uses no explicit strata, and, for accepted modules, the initial stratum component of **`decreases`** clauses is generated automatically to be a sympathetic stratification.

8.6. Top-tier Examples

As a final variation of the `Car` example from Section 8.2, suppose module `Car` wants to export method `SelectTrack`. With the frugal approach to top-tier exports, an attempt to declare only **`export`** `SelectTrack` would be rejected on the grounds that this method is not top tier. To remedy the situation, module `Car` also has to declare

```
export Car > SelectTrack
```

Under the frugal top-tier approach, exported methods are exported as top tier by default. In light of that default, automatic stratification, and a composition requirement, here are all of the declarations needed to verify the modules from Section 7:

```
module Client {
  import Util, Math
  requires Util ⊃ Math
```

```
  method Process' overrides Process(x: int): int
  { ... Sqrt(...) ... }
}
module Util {
  export Process
  virtual method Process(x: int): int
}
module Math {
  export Sqrt
  method Sqrt(x: int): int { ... }
}
```

For this particular program, no **decreases** clauses are needed at all, and the only declaration we had to add to facilitate termination verification is the composition requirement `Util` $\supset$ `Math`, which has to be checked at link time.

8.7. On Syntax

The syntax I have used in the section above presents the various ordering specifications separately. In a programming language, it may be more natural to fold that information into existing declarations. For example, composition requirements may be better notated as part of import declarations.

9. Related Work

To my knowledge, the only other work that addresses modular termination specifications of dynamically bound methods is by Jacobs et al. [6]. They use as the well-founded order a lexicographic pair. The first component of that pair is a multiset of method names, where method names have a predetermined order inside modules and method names from different modules are ordered according to the module import ordering. The second component of the pair is an ordinal number, which is typically constructed by mapping lexicographic tuples of numbers into polynomials in the limit ordinal ω. Although this second component is formulated in terms of a single ordinal, it is similar to the ready-to-use k components of $\mathcal{U}_\top$ in this paper. The more interesting difference lies in their first component versus the use of strata in this paper.

Using multisets of method names provides more flexibility than the strata proposed in this paper. For example, it is possible to give `Process` (Section 8.6) a specification that allows `Process'` to call `Sqrt` without needing to know anything about the relative ordering of modules `Util` and `Math`. On the other side of the ledger, the flexibility comes at a cost. To make use of the full flexibility requires specifications that relate objects to their "dynamic depth" [6].

As a final point of comparison with the work by Jacobs et al. [6], if there are no inter-module overrides (that is, if overrides are always in the same module as the corresponding virtual method), then one can obtain this paper's general methodology using their methodology by defining a stratum to be a multiset consisting of one method name. (One also needs to adjust their call rule to refund the permissions required to

make a call upon return from the call.) However, because their ordering of method names within and across modules is fixed, there is no analogous way to define a **`below`** `A` clause where the origin of stratum `A` is an imported module.

The specialized methodology in this paper does not use strata explicitly. This means that programs need to use **`decreases`** clauses only to explain why recursion and mutual recursion terminates. To reduce programmer-supplied information even more, the Dafny program verifier [8] uses a default **`decreases`** clause for any method that doesn't explicitly declare one. The default uses a lexicographic tuple consisting of the method's parameters, in the order given. In practice, this default turns out to be an effective way to reduce explicit termination specifications.

To go even further in reducing explicit termination specifications, one can employ some kind of termination analysis, like that by Cook, Podelski, and Rybalchenko [2]. Such analyses typically require the entire program, which would not qualify as modular verification. However, even applying such an analysis within a module would help reduce explicit specifications.

10. Conclusions

Termination proofs must show a decrease in some well-founded order. In this paper, I proposed using as the domain of that order a lexicographic tuple, where the first component layers methods into partially ordered strata and the remaining components are ordinary expressions of the language. In a verified program, the strata correspond to disjoint sets of methods, where every recursive or mutually recursive call goes between methods of the same stratum. The task of ensuring termination is divided into the subtask of partitioning methods into strata and the subtask of applying theorem proving to calls within each stratum. Calls between strata are non-recursive, so their termination follows trivially.

In the paper, I presented two methodologies for performing the first of the two subtasks. In the general methodology, strata are explicitly introduced by the programmer. In the specialized methodology, the strata are inferred from modular views of call graphs, some additional ordering specifications, and some restrictions. Both methodologies lend themselves to modular verification.

Acknowledgments

I am grateful to Fabio Madge, Dominic Mulligan, John Tristan, Remy Willems, and Stefan Zetzsche for discussions and feedback that improved this paper.

References

[0] Reza Ahmadi, K. Rustan M. Leino, and Jyrki Nummenmaa. Automatic verification of Dafny programs with traits. In Rosemary Monahan, editor, *Formal Techniques for Java-like Programs, FTfJP 2015*. ACM, 2015.

[1] Roderick Chapman, Claire Dross, Stuart Matthews, and Yannick Moy. Co-developing programs and their proof of correctness. *Communications of the ACM*, 67(3):84–94, March 2024.

[2] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 415–426. ACM, 2006.

[3] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[4] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[5] Robert W. Floyd. Assigning meanings to programs. *Proceedings Symposium on Applied Mathematics*, 19:19–31, 1967.

[6] Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015*, volume 37 of *LIPIcs*, pages 664–688. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2015.

[7] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.

[8] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning — 16th International Conference, LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, April 2010.

[9] K. Rustan M. Leino. Accessible software verification with Dafny. *IEEE Software*, 34(6):94–97, 2017.

[10] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation — 17th International Conference, VMCAI 2016*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.