OO Big O

Joan Krone
Denison University
Department of Math and CS
Granville, Ohio 43023
740-587-6484
krone@denison.edu

W. F. Ogden
The Ohio State University
Neal Avenue
Columbus, Ohio 43210
614-292-6007
ogden@cis.ohio-state.edu

ABSTRACT
When traditional Big O analysis is rigorously applied to object oriented software, several deficiencies quickly manifest themselves. Because the traditional definition of Big O is expressed in terms of natural numbers, rich mathematical models of objects must be projected down to the natural numbers, which entails a significant loss of precision beyond that intrinsic to order of magnitude estimation. Moreover, given that larger objects are composed of smaller objects, the lack of a general method of formulating an appropriate natural number projection for a larger object from the projections for its constituent objects constitutes a barrier to compositional performance analysis.

Here we recast the definition of Big O in a form that is directly applicable to whatever mathematical model may have been used to describe the functional capabilities of a class of objects. This generalized definition retains the useful properties of the natural number based definition but offers increased precision as well as compositional properties appropriate for object based components. Because both share a common mathematical model, functional and durational specifications can now be included in the code for object operations and formally verified. With this approach, Big O specifications for software graduate from the status of hand waving claim to that of rigorous software characterization.

Categories and Subject Descriptors
D.2[Software Engineering], F.2[Analysis of Algorithms], F.3[Logics and Meanings of Programs]: Specifications, Models, Semantics – *performance specifications, performance analysis, performance proof rules.*

General Terms
Algorithms, Performance, Verification.

Keywords
Performance, Formal Specification, Verification, Big O.

1. INTRODUCTION
The past forty years have seen a great deal of work on the rigorous specification and verification of programs' functional correctness properties [2] but relatively little on their performance characteristics. Currently performance "specifications" for programs commonly consist of reports on a few sample timings and a general order of magnitude claim formulated in a Big O notation borrowed from number theory. As we have discussed elsewhere [6], such an approach to the performance of reusable components is no more adequate than the test and patch approach is to their functionality.

As with functionality, problems with performance usually have their roots in the design phase of software development, and it's there that order of magnitude considerations are most appropriately encountered. This means our order of magnitude notations are generally applied in a somewhat rough and ready fashion (which is probably why defects in our current ones have escaped notice for so long). However, if their formulation doesn't reflect the ultimate performance of the components under design accurately and comprehensibly, then marginal designs become almost inevitable. So the way to get an appropriate order of magnitude definition is to formulate one that meshes smoothly with program verification.

With the advent of object oriented programming and a component based approach to software, formal specifications of a component's functionality are considered to be critical in order for clients to make good choices when putting together a piece of software from certified components.

To meet the need for reasoning about performance as well as functionality, we introduce a new object appropriate definition of Big O. Object Oriented Big O, or OO Big O for short, allows one to make sensitive comparisons of running times defined over complex object domains, thereby achieving much more realistic bounds than are possible with traditional big O.

We cast our approach in a framework that includes an assertive language with syntactic slots for specifying both functionality and performance, along with automatable proof rules that deal with both. Equally important is the need for the reasoning to be fully modular, i.e., once a component has been certified as correct, it should not be necessary to reverify it when it is selected for use in a particular system.

Our approach is based on the software engineering philosophy that a component should be designed for reuse and thus include a general mathematical specification that admits several possible implementations – each with different performance characteristics [3, 7]. Of course, in order for a component to be reusable, it should include precise descriptions of both its functionality and its performance, so that prospective clients can be certain they are choosing a component that fits their needs.

It is also important that all reasoning about constituent components – including reasoning about performance – be possible without knowing implementation details. In fact, if one is using generic components, it should be possible to reason about those components, even before details such as entry types to a generic container type, are available.

With these considerations in mind, we make our new definition for Big O, and then we present an example illustrating how it can be used to supply a component with a formal summary of both its functionality and performance. Proof rules can then be applied to verify these specifications.

2. OO Big O Definition

The traditional Big O is a relation on natural number based functions defined in the following way:

Given f, g: _□ _, f(n) is O(g(n)) iff □ constants c and n_0 such that f(n) □ c·g(n) whenever n ≥ n_0. A program whose running time is O(g) is said to have growth rate g(n) [1].

When the object classes central to modern programming are formally modeled, they are viewed as essentially arbitrary mathematical domains, which generally have little to do with natural numbers. So the natural expression of the duration f(x) of an operation on object x is as a function directly from its input domain to the real numbers. Clearly any gauging function g that we might want to use as an estimate for f should have the same domain. Accordingly, the Is_O relation between functions (f(x) Is_O g(x)) is defined by:

Definition: (f: Dom□ _) Is_O (g: Dom□ _): B = (□A: $\_^{>0}$, □ H: _ □□ x: Dom, f(x) □ A·g(x) + H).

In other words, for two timing functions f and g mapping a computational domain Dom to the real numbers, to say that f(x) Is_O g(x) is to say that there is some positive acceleration factor A and some handicap H such that for every domain value x, f(x) □ A·g(x) + H. If we think of f and g as representing competing processes, f being big O of g means that g is not essentially faster than f. If f is run on a processor A times faster than g's processor and also given a head start H, then f will beat g on all input data x.

Of course, in order to use this definition, it is necessary to have mathematical support in the form of theorems about the revised definition of Is_O. For example, we need an additive property so we can apply our analysis to a succession of operation invocations:

Theorem OM1: If $f_1(x)$ Is_O $g_1(x)$ and $f_2(x)$ Is_O $g_2(x)$, then

$f_1(x) + f_2(x)$ Is_O Max($g_1(x)$, $g_2(x)$).

A development of appropriate theorems and definitions appears in [5]. We turn now to formal specification of functionality and performance for components.

3. ABSTRACT OBJECTS

If you want to produce rigorously specified and verified software components that support genericity, facilitate information hiding, and can be reasoned about in a modular fashion, it is necessary to adhere carefully to certain guidelines and principles [7].

As a prelude to presenting an example illustrating our approach to Big_O, we will briefly discuss a component that does adhere to these principles. For any component developed in our system we use a module construct called a ***Concept*** to record formally its functionality specifications. A concept serves on the one hand as a conceptually simple yet fully precise description of functionality for a client and at the same time as a functionally complete yet maximally flexible requirements document for the implementer. A concept will typically be generic to facilitate maximum reuse.

Our example will be the component concept that captures the "linked list." Because one of our guidelines is to tailor a concept to simplify the client's view, we call this concept a one-way list template and the objects it provides list positions. We describe list positions mathematically as pairs of strings over the entry type. The first string in a list position contains the list entries preceding the current position and is named *Prec*; the second string is the remainder of the list, *Rem*. Since the operations on list position (Insert, Advance, Reset, Remove, etc.) all have easy specifications in terms of this model, and since the underlying linking pointers are cleanly hidden, reasoning about client code is much simplified.

Although variations in list implementation details are usually insignificant, our system allows for the possibility of a multiplicity of different realizations (implementations) for any given concept. Each **Realization**, with its own potentially distinct performance characteristics, retains a generic character, since parameter values such as the entry type for lists have yet to be specified. The binding of such parameters only takes place when a client makes a **Facility**, which involves selecting the concept and one of its realizations along with identifying the appropriate parameters.

When designing concepts for maximal reusability, our guidelines prescribe that only the basic operations on a class of objects should be included, so for lists we only include Insert, Advance, etc., but not Search, Sort, etc. In order to have a rich enough Big O example, we will consider such a sorting operation, so we need to examine the **Enhancement** construct used to enrich basic concepts such as the one-way list.

Well-designed enhancements also retain to the extent possible the generality we seek in our concepts, but often they do add constraints that prevent their use in certain situations. Providing a Sort_List operation, for example, requires that list entries possess an ordering relation _, so certain classes of entries would be precluded from lists if Sort_List were one an operation in the basic list concept.

EXAMPLE APPLICATION OF BIG □

To clarify the setting in which Big □ performance specifications must work, we begin our example by examining the sorting enhancement that lays out the functional specifications any implementation must satisfy.

This enhancement's name is *Sort_Capability*, and it maintains the generic character of the concept (which allows entries to be of arbitrary type) by importing an ordering relation _ on whatever the entry type may be. A **requires** clause insists that any imported _ relation actually be a total preordering on whatever the entry type is.

The **uses** clause indicates that this component relies on a mathematical theory of order relations for the definitions and properties of notions such as total preordering. Note that an automated verifier would need such information.

The mathematical definition *In_Ascending_Order* is introduced to make later assertions easier to express. In this case, the **ensures** (post condition) clause for the operation *Sort_List*, is stated in terms of this definition and indicates that the *Prec* string of the list must be ordered according to the _ relation passed in.

In the *Sort_List* operation, **upd** denotes the updates parameter mode, indicating that this operation may change the *List_Position* parameter *P*.

The second part of the **ensures** clause guarantees that the entries in the list after the operation *Sort_List* has taken place are exactly the same entries as those before the operation took place; the @ symbol indicating the value of *P* at the beginning of the operation.

Enhancement Sort_Capability(**def const** (x: Entry) _ (y: Entry): B);

for One_Way_List_Template;

uses Basic_Ordering_theory;

requires Is_Total_Preordering(_);

Def const In_Ascending_Order(□: Str(Entry)): B =

(□ x, y: Entry, **if** □x□□y□Is_Substring □, **then** x _ y);

Oper Sort_List(**upd** P: List_Position);

ensures P.Prec = □ **and**

In_Ascending_Order(P.Rem) **and**

P.Rem Is_Permutation @P.Prec_@P.Rem;

end Sort_Capability;

A client who wishes to order a list would be able to choose this list enhancement on the basis of these functional specifications. However, before choosing among the numerous realizations for it, a client should be able to see information about their performance. Rather than giving such timing (**duration**) information a separate ad hoc treatment, we introduce syntax for formally specifying **duration** as part of each **realization.** In short, we associate with every component not only a formal specification of its functionality but of its performance as well, so that a potential client can choose a component based on its formal specifications rather than on its detailed code.

To see how the new Big O definition can improve performance specifications, we will look at an insertion sort realization for the Sort_List operation. Since our focus here is on formal specifications of timing for object-oriented components, we go directly to the parts of the realization that are most immediately relevant to such specifications.

Because a realization for a concept enhancement relies upon the basic operations provided by the concept, its performance is clearly dependent on their performance, and that can vary with the realization chosen for the concept. Fortunately performance variations for a given concept's realizations seem to cluster into parameterizable categories, which we can capture in the **Duration Situation** syntactic slot. The normal situation for a one-way list realization, for example, is that all the operations have □(1) durations. Of course realizations of lists with much worse performance are possible, but we wouldn't ordinarily bother to set up a separate duration situation to support analyzing their impact on our sort realization.

Duration situations talk about the durations of supporting operations such as the Insert and Advance operations by using the notation $\mathbf{Dur}_{\mathrm{Insert}}$(E, P), $\mathbf{Dur}_{\mathrm{Advance}}$(P), etc. So we can use our Big O notation to indicate that the performance estimates labeled "normal" only hold when $\mathbf{Dur}_{\mathrm{Insert}}$(E, P) Is_O 1, etc.

The realization is next, followed by additional explanation. We suggest reading the Duration Situation, then skipping to the procedure and returning to the definitions and theorems section as you read the subsequent explanation.

Realization Insertion_Sort_Realiz(

Oper Lss_or_Comp(**rest** E1, E2: Entry): Boolean;
ensures Lss_or_Comp = (E1 _ E2);)
for Sorting_Capability;

Duration Situation Normal: $\mathbf{Dur}_{\mathrm{Reset}}$(P) Is_□ 1 **and**

$\mathbf{Dur}_{\mathrm{Length\_of\_Rem}}$(P) Is_□ 1 **and**

$\mathbf{Dur}_{=}$(i, j) Is_□ 1 **and** $\mathbf{Dur}_{\mathrm{Remove}}$(E, P) Is_□ 1 **and**

$\mathbf{Dur}_{\mathrm{Advance}}$(P) Is_□ 1 **and**

$\mathbf{Dur}_{\mathrm{Insert}}$(E, P) Is_□ 1 **and** Entry.**Init_Dur** Is_□ 1 **and**

List_Position.**Init_Dur** Is_□ 1 **and**

$\mathbf{Dur}_{\mathrm{Lss\_or\_Comp}}$(E1, E2) Is_□ 1;

Def. const (E1: Entry) _ (E2: Entry): B = (E1 _ E2 **and** ¬ E2 _ E1);

Inductive def. on □: Str(Entry) **of**

const Rank(E: Entry, □): _ **is**

(i) Rank(E, □) = 0;

(ii) Rank(E, ext(□, D)) = $\begin{cases} \mathrm{Rank}(E, á) + 1 & \text{if } D \text{ p } E \\ \mathrm{Rank}(E, á) & \text{otherwise} \end{cases}$;

Theorem IS1: □ E: Entry, □ □, □: Str(Entry), Rank(E, □_□) = Rank(E, □) + Rank(E, □);

Theorem IS2**:** □ E: Entry, □ □: Str(Entry), Rank(E, □) =

$$\left(\sum_{\text{C: Entry □ C p E}} \text{Occurs\_Ct(C, á)}\right);$$

Theorem IS3: □ E: Entry, □ □, □: Str(Entry),

if □ Is_Permutation □, **then** Rank(E, □) = Rank(E, □);

Theorem IS4: □ E: Entry, □ □: Str(Entry), Rank(E, □) □ |□|;

Inductive def. on □: Str(Entry) **of const** P_Rank(□): _ **is**

(i) P_Rank(□) = 0;
(ii) P_Rank(ext(□, E)) = P_Rank(□) + Rank(E, □);

Theorem IS5: □ □: Str(Entry), P_Rank(□) =

$$\left(\sum_{\substack{\text{E: Entry, á: Str(Entry) □} \\ \text{á○□E□Is\_Prefix â}}} \text{Rank(E, á)}\right);$$

Theorem IS6: □ □: Str(Entry), P_Rank(□) □ |□|·(|□| □ 1)/2;

Def. const Len(P: List_Position): _ = (|P.Prec _ P.Rem|);

```
Proc Sort_List( upd P: List_Position );
   Duration Normal:
    Is_□ Max( Len( @P ), P_Rank( @P.Prec _ @P.Rem ) );
   Var P_Entry, S_Entry: Entry;
   Var Sorted: List_Position;
   Aux Var Processed_P: List_Position;
   Reset( P );
  While Length_of_Rem( P ) ≠ 0
        affecting P, P_Entry, Sorted, S_Entry, Processed_P;
        maintaining Sorted.Prec = □ and
          In_Ascending_Order( Sorted.Rem ) and
          Processed_P.Prec _ P.Rem = @P.Prec _ @P.Rem
        and
          Sorted.Rem Is_Permutation Processed_P.Prec;
        decreasing |P.Rem|
        elapsed_time Normal:
      Is_□ P_Rank(Processed_P.Prec) + |Processed_P.Prec|;
  do
        Remove( P_Entry, P );
        Remember
          Iterate
            affecting Sorted, S_Entry;
            maintaining
              Sorted.Prec _ Sorted.Rem = @Sorted.Rem and
            □ E: Entry, if □E□Is_Substring Sorted.Prec
                then E _ P_Entry;
          decreasing |Sorted.Rem|;
          elapsed_time Normal: Is_□ |Sorted.Prec|;
        when Length_of_Rem( Sorted ) = 0
                     do exit;
              Remove( S_Entry, Sorted );
            when Lss_or_Comp( P_Entry, S_Entry )
                     do Insert( S_Entry, Sorted ) exit;
              Insert( S_Entry, Sorted );
              Advance( Sorted );
          repeat;
       forget;
       Aux Comp Insert( Replica(P_Entry), Processed_P );
                     Advance( Processed_P ) end;
      Insert( P_Entry, Sorted );
      Reset( Sorted );
    end;
    P :=: Sorted;
  end Sort_List;
end Insertion_Sort_Realiz;
```

For the *Sort_List* procedure, the strategy of the outer loop is to place successive values *P_Entry* from the list *P* into their proper place in the list *Sorted*. Ultimately, the two lists, *P* and *Sorted* will be swapped so the original list *P* is replaced by the same elements rearranged to be in the specified order. The task of the inner loop is to position the *Sorted* list appropriately for the insertion of *P_Entry*.

For each loop, we record the loop invariant in the **maintaining** clause. To express its invariant, it helps to be able to refer to the value of the list *Sorted* at the beginning of the inner loop. For this purpose, we employ the **Remember-forget** construct. The effect of **Remember** is to record the current value of *Sorted* in adjunct variable *@Sorted, P* in *@P*, etc. To preserve the values of *@P*, etc. previously remembered, these values go into the adjunct variables *@@P*, *@@Sorted*, etc., where we can reference them as needed. The effect of **forget** is roughly the opposite of **Remember**.

Since the inner loop just changes the values of a couple of the variables, we can simplify the **maintaining** clause by listing those affected variables in the **updating only** list. The **decreasing** clause provides a place for a progress metric, necessary for proving termination and thereby establishing total correctness.

To specify the performance of the procedure we need to supply a **duration** clause, a formula that is synthesized from the durations of the constituent parts of the procedure, in this case, a nested loop. Accordingly, we examine first the inner loop and then the outer loop to get a duration estimate for each. Then we put those estimates together to get the duration clause to associate with the procedure.

For the inner loop, we have filled in the **elapsed time** expression using our new OO Big O with the expression **Is_O** |Sorted.Prec|, indicating that whenever we are at the beginning of the loop, the time that has elapsed since we entered the loop construct can be estimated by the length of the *Prec* string of the *Sorted* list. Remember that our one way list is modeled as a pair of strings, *Prec* and *Rem*. The correctness of this specification can be verified simply by first verifying that |Sorted.Prec| = 0.0 when we first arrive at the inner loop, since Sorted.Prec = □ at that point. Then, we check that the sum of the durations of the operations in the loop body Is_O of the difference between the elapsed time gauge function at the end of the loop body and its value at the beginning. Here |Sorted.Prec| increases by one on each iteration and there are five order of 1 duration operations in the body, so this boils down to checking that (5)Is_O(1) for the "normal" situation. Now we turn our attention to the outer loop, where problems with traditional Big_O manifest themselves. For the elapsed time expression here, we again need an estimate of the time at the beginning of each iteration that's elapsed since the beginning of the construct. In each iteration, an entry is removed from the original list, the inner loop advances to its proper place in the *Sorted* list. and then it is inserted at this point in the *Sorted* list.

Clearly the elapsed is going to depend heavily upon the order of the elements in the original list @P, but traditional natural number based Big_O analysis would require that we project the @P list onto a natural number "n" and express our gauge function in terms of that n (e.g. n^3). Typically that n would be the length of a list (what we've formally defined as Len(P) so that n = Len(@P)). Since Len(@P) is totally insensitive to the order of the entries in @P, we could at best end up with a duration estimate for Sort_List of n^2.

To exploit the increased precision of the OO Big O definition, we need to define a function on strings of entries α that counts how many entries in α are less than an entry E and hence would be skipped over when positioning E after α has been sorted, and that's why our realization includes the definition of the Rank(E, α) function and states some of its properties. Since the elapsed time of the outer loop depends upon the cumulative effect of positioning successive entries in @P, we also need to define a "preceding rank" function P_Rank(α).

When we attempt to use the P_Rank function to specify the elapsed time of our outer loop, we notice that constructing the Sorted list has scrambled the data from P that determines the elapsed time, so we add an auxiliary variable, *Processed_P*, which we use to keep track of that otherwise obliterated ordering data. Auxiliary variables and the auxiliary code that updates them are just used in reasoning about programs and are never compiled, so it is syntactically incorrect for them ever to influence the values ordinary variables.

Using these definitions, we express our **elapsed time** bound for the outer loop as

Is_O |Processed_P.Prec| + P_Rank(Processed_P.Prec).

When the loop is entered, Processed_P.Prec is empty, so |Processed_P.Prec| + P_Rank(Processed_P.Prec) = 0.0. Understanding the inductive step involves noticing that if E is the next P_Entry and α is the current Processed_P.Prec, then after executing the loop body, Processed_P.Prec = α_$\langle$E$\rangle$, so that the difference of the before and after values of the gauge function is 1 + P_Rank(α_$\langle$E$\rangle$) $-$ P_Rank(α) = 1 + Rank(E, α). The duration of the outer loop body is easily seen to be Big O of this quantity, since the execution time of the inner loop Is_O Rank(E, α), and durations of the other four operations in outer loop body are all Big O of 1.

The overall procedure Sort_List consists of only a few more Big O of 1 operations and variable declarations, so its normal duration bound simplifies to

Max(Len(@P), P_Rank(@P.Prec_@P.Rem).

Now one of the results about P_Rank is that P_Rank(α) $\leq$ |α|·(|α|$-$1)/2, so it follows that $\mathbf{Dur}_{Sort\_List}$(P) Is_O Len(P)^2 too, so we can get the much less exacting estimate produced by traditional Big O analysis if we wish. We're just not forced to when we need a sharper estimate. Another point to note is that besides being compatible with correctness proofs for components, the direct style of performance specification is much more natural than the old style using the often ill defined "n" as an intermediary.

4. THE CALCULUS FOR OO BIG O

Our Sort_List example illustrates how we can use the new Big O notation in performance specifications and indicates how such specifications could fit into a formal program verification system. The success of such a verification system depends upon having a high level calculus for Big O that allows verification of performance correctness to proceed without direct reference to the detailed definition of Big O.

Of course making such a calculus possible is one of the primary motivations for the new Big O definition, and in [4] we have developed a number of results like the earlier theorem OM1 to support this calculus. Another simple illustration of a property of the new Big O important for verification is dimensional insensitivity.

Theorem OM2: If f(x) Is_O g(x) and F(x, y) = f(x) and G(x, y) = g(x), then F(x, y) Is_O G(x, y).

Taken together, these results must justify both the proof rules for our program verification system and the expression simplification rules for the resulting verification conditions.

We should also note that Big O estimates are only the most obvious of order of magnitude estimates, and that we can readily extend our object based definitions and theorems to cover little o, Big Ω, Big Θ, and little ω. See [5].

5. CONCLUSION

A critical aspect of reusable components is assured correctness, an attribute attainable only with formal specifications and an accompanying proof system. Here, we claim that while functional correctness is absolutely necessary for any component that is to be reused, it is not sufficient. Reusable components need formally specified performance characteristics as well.

Traditional Big O order of magnitude estimates are inadequate because they deal only with the domain of natural numbers and offer no support for modularity and scalability.

Here we introduce a new mechanism for doing order of magnitude analysis for components, OO Big O. This new method is applicable to programs written over any domain and addresses the issues of generic data abstraction and specification-based modular performance reasoning.

If we want to design software components that can be reused, we claim that such components must have formal specifications for both functionality and performance associated with them and that there must be a proof system that addresses both. Moreover, to avoid intractable complexity, it must be possible to reason about these components in a modular fashion, so that one can put together programs hierarchically, each part of which can be reasoned about using only the abstract specifications for its constituent parts. To avoid the rapid compounding of imprecision that otherwise happens in such systems, it is also essential to use high precision performance specification mechanisms such as OO Big O.

To develop maximally reusable components, it is necessary to be able to reason about them in a generic form, without knowing what parametric values may be filled in when the component is put into use.

OO Big O satisfies all these criteria, supporting complete genericity, performance analysis of programs over any domain, and modular reasoning.

6. REFERENCES

1. Aho, A., Hopcroft, J., Ullman, J., Data Structures and Algorithms, Addison-Wesley, 1983.
2. de Roever, W., Engelhardt, K. Data Refinement: Model-Oriented Proof Methods and their Comparison, Cambridge University Press, 1998.
3. Krone, “The Role of Verification in Software Reusability.” Dissertation, The Ohio State University, 1988.
4. J. Krone, W. F. Ogden, and, M. Sitaraman, Modular Verification of Performance Constraints, Technical Report RSRG-03-04, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2003, 25 pages.
5. Ogden, W. F., CIS680 Coursenotes, Spring 2002.
6. Sitaraman, M., Krone, J., Kulczycki, G., Ogden, W., and Reddy, A. L. N., “Performance Specification of Software Components,” *ACM SIGSOFT Symposium on Software Reuse*, May 2001.
7. Weide, B., Ogden, W., Zweben, S., “Reusable Software Components,” in M.C. Yovits, editor, **Advances in Computers**, Vol 33, Academic Press, 1991, pp. 1 – 65.