Performance Specification of Software Components

Murali Sitaraman
Dept. Comp. Science
Clemson University
Clemson, SC 29634, USA
+1 864 656 3444
murali@cs.clemson.edu

Greg Kulczycki
Dept. Comp. Science
Clemson University
Clemson, SC 29634, USA
+1 864 656 3444
gregwk@cs.clemson.edu

Joan Krone
Dept. Math. and Comp. Science
Denison University
Greenville, OH 43023, USA
+1 740 587 6484
krone@denison.edu

William F. Ogden
Dept. Comp. & Info. Science
The Ohio State University
Columbus, OH 43210, USA
+1 614 292 5813
ogden@cis.ohio-state.edu

A. L. N. Reddy
MomsDesk Commerce Corporation
341 Cobalt Way #208
Sunnyvale, CA 94089, USA
+1 408 736 3816
alnreddy@momsdesk.com

ABSTRACT

Component-based software engineering is concerned with predictability in both functional and performance behavior, though most formal techniques have typically focused their attention on the former. Reasoning about the (functional or performance) behavior of a component-based system must be compositional in order to be scalable. Compositional performance reasoning demands that components include performance specifications, in addition to descriptions of functional behavior. Unfortunately, as explained in this paper, classical techniques and notations for performance analysis are either unsuitable or unnatural to capture performance behaviors of generic software components. They fail to work in the presence of parameterization and layering.

The paper introduces elements of a compositional approach to performance analysis using a detailed example. It explains that performance specification problems are so basic that there are unresolved research issues to be tackled even for the simplest reusable components. These issues must be tackled by any practical proposal for sound performance reasoning. Only then will software developers be able to engineer new systems by choosing and assembling components that best fit their performance (time and space) requirements.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable software – models.

General Terms

Algorithms, Documentation, Performance, Design, Languages, Verification.

Keywords

Data Structures, Generic Objects, Time & Space Analysis, Storage Management.

1. INTRODUCTION

A major goal of software engineering is to introduce a modular style of software development by designing and deploying components that are easy to understand and reuse, are correct and efficient, and are easy to adapt and maintain. The focus of this paper is on component performance within this larger software engineering context. While correct functioning is a fundamental concern for any component, good performance is an essential property of a useful component [24]. Performance specifications are essential for two basic reasons:

- In a software component industry, multiple implementations from different vendors will be available to provide the same functional behavior. Clients will engineer new systems by choosing and assembling components that best fit their performance (time and space) requirements [1, 17, 21, 22]. To facilitate meaningful selection from among competing alternatives, software components should include descriptions of performance behavior in addition to functional behavior.
- We should be able to reason (formally or informally) that a component-based system satisfies its specified performance in a *modular* fashion. If components have performance specifications, then the performance of the system can be derived based on the components it directly uses; the component implementations need not be re-analyzed in each new context where they are used. Compositionality is essential for reasoning to scale up to component-based systems [19, 20].

Given these software engineering objectives, this paper addresses foundational questions in performance specification of generic, object-based software components. These and other inevitable questions [20] must be answered so that some day software practitioners will have the methods, tools, and the training necessary to reason soundly and compositionally about the performance of their systems. In the interim, this paper serves to warn developers of serious performance analysis problems that are typically swept under the rug in computing education and literature.

Ideally, the performance specification for an implementation will be expressed in conceptual terms that are understandable to users without a complete knowledge of its internal details. Estimates of both execution time durations as well as memory capacity requirements are needed to ensure that a component implementation is suitable for an application. Though historically the focus has been on time estimates, the importance of the latter for good software engineering cannot be overstated [9, 16]. We would like to be able to predict not only how long it will take our software to execute its various functions, but also if and when our systems will run out of memory.

While the performance specification problem naturally becomes complex due to non-deterministic computing and dynamic storage allocation, we note that the problem is non-trivial even for simple components that do not exhibit such behavior. To see why, we need to look no farther than an array-based *generic* stack component that provides Stack objects and operations to manipulate variables of Stack type. In particular, consider the following basic time/space questions:

- How much time does it take to copy or destroy (deeply) a given stack S?
- How much storage does a given stack S occupy?

A superficial big-O analysis might suggest O(|S|) as answers to the questions, where |S| denotes the depth of the stack. These answers are indeed reasonable for stacks containing simple objects, such as integers or characters. But to provide estimates for copying or destroying a stack containing, say trees, we need to account for the contents of the stack, not merely its depth. This is because a few large trees may take much longer to copy or destroy, and may occupy much more storage than a stack containing several empty trees. More importantly, for generic components, we need to provide performance expressions independent of (and even before we know) the type of parametric objects. Expressions such as O(|S|) are simply wrong for arbitrary generic stacks, whose content objects are yet to be determined.

The contributions of this paper are in raising a host of issues in generic software component performance specification, and providing a framework within which they can be addressed. The proposed framework includes a generalization of classical big-O definitions so that they are suitable and natural for expressing reusable software time/space characteristics. Using a detailed, but simple stack example, the paper illustrates our solutions to (some of) the key questions and directions for tackling others.

Use of a stack component often raises a valid question: If and how will the results generalize to larger or more complex components? This scalability question is important, because if the notations and the framework we propose here do not generalize, then ultimately the results will have little value or impact. A careful reading of the paper should convince a reader that this goal of scalability is exactly what has led us to seemingly more complex answers than might be sufficient to satisfy a stack user. The problems posed here arise in dealing with *any* generic component, and the solutions are meant to be generally applicable. We use a stack example more than anything else because the performance problem demands a close look at the specification and implementation of a component, and anything more complex would obfuscate the details of presentation and would make the results inaccessible to most readers.

The rest of the paper is organized into the following sections. Section 2 outlines alternative definitions/notations for expressing performance of reusable components. Section 3 contains a specification of Stack_Template. Section 4 discusses an implementation that minimizes space usage by trading off time, and provides its performance specification. Section 5 contains a summary and directions for further research.

2. PERFORMANCE NOTATIONS

In general, the execution time duration (and storage requirement) of a procedure depends on its parameters. Consider an arbitrary procedure with one parameter x of type T. Typically, we are interested in a function that is an upper bound on the execution time of the procedure, denoted by $D_p(x)$: $T \rightarrow \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers. In estimating approximate upper bounds on duration, it is appropriate to highlight factors that are significant and cover up intricate procedural details that affect the duration only marginally. In other words, in expressing execution time bounds we are usually more interested in *rate of growth* or order of growth than in particular constants. Big-O notations are generally used for expressing rate of growth of upper bounds. If the duration of a procedure P is $D_p(x)$, we might say that $D_p(x)$ **Is_O**(f(x)) to denote that f(x) is an approximate upper bound for the duration of P. In other words, $D_p(x)$ grows at most as fast as f(x).

The predicate **Is_O** essentially relates two arbitrary real number functions and roughly states that the first function is less than or comparable to the second one. It is formally defined below in infix notation.

Definition (f: Dom $\rightarrow \mathbb{R}$) **Is_O** (g: Dom $\rightarrow \mathbb{R}$): $\mathbb{B}$ =
($\exists$A: $\mathbb{R}^{>0}$, $\exists$ H: $\mathbb{R} \ni \forall$x: Dom, f(x) $\leq$ A*g(x) + H);

Here "Dom" stands for an *arbitrary* mathematical space and $\mathbb{R}^{>0}$ denotes the set of positive real numbers. The definition says that f(x) **Is_O**(g(x)) if there is some positive acceleration factor A and some handicap H such that for every domain value f(x) $\leq$ A*g(x) + H. This definition is of interest for typical infinite domains, because it holds for all functions f and g, when the domain is finite. It is always possible to pick a suitable handicap in those cases. The handicap is also what makes it possible to eliminate the usual provision for a finite exclusion set where the assertion does not hold. For arbitrary mathematical spaces, the exclusion set may need to be infinite. This observation in turn implies that straightforward generalization of a classical big-O definition from functions from natural number spaces to more general functions will not work. A classical definition based on [3, 10] (syntactically modified to make it easy to compare) is given below:

Definition (f: $\mathbb{N} \rightarrow \mathbb{R}$) **Is_O** (g: $\mathbb{N} \rightarrow \mathbb{R}$): $\mathbb{B}$ =
($\exists$c: $\mathbb{R}^{>0}$, $\exists n_0$: $\mathbb{N} \ni \forall$n: $\mathbb{N}$, n $\geq n_0 \Rightarrow$ 0.0 $\leq$ f(n) $\leq$ c*g(n));

To use this definition for generic components, arbitrary mathematical spaces must be metricized, i.e., mapped to the natural numbers. Even assuming it is possible, this additional step adds a non-trivial level of complexity, and renders resulting expressions unnatural. The more general definition given in this paper is compatible with the classical definition, when the domain turns out to be the natural numbers. For example, if a number-based computation has complexity **Is_O**(n) in the classical sense, it also has complexity **Is_O**(n) under the proposed definition (except in a few anomalous cases). General definitions of other order notations, including **Is_o** that defines a strictly less than relationship among two functions, **Is_Ω** that is the negation of **Is_o** relationship, and proofs that the definitions have desirable properties such as transitivity are discussed in [15].

3. AN EXAMPLE BEHAVIORAL SPECIFICATION

Without loss of generality, we have used a dialect of the RESOLVE specification and implementation notation [14, 17] to illustrate the issues throughout this paper. This is because it is necessary to distinguish abstract and implementation views of components. The same performance questions raised here need to be addressed in generic component development in any language, such as C++ or Java, with or without formal specifications.

Performance specifications need to be expressed in the context of functionality specifications, and therefore, we begin with specification of functionality. Figure 1 shows the functionality specification (or a conceptual outsider view) of a Stack component. Stack_Template is a generic **concept** (specification template), and it is parameterized by the type of items to be contained in stacks and by the maximum size to which a stack can grow. The type and operation names provided by a concept, in general, must be augmented with user-oriented explanations, because even the most meaningful names only hint at potential effects.

To reason mechanically or manually, but soundly, about a program that uses stacks, it is essential to have a mathematical conceptualization of Stack (and Integer) variables and operations. Using String_Theory, a mathematical unit that formally defines string notations [15], a (parameterized) Stack is conceptualized as a string of entries. A string is similar to, but simpler than, a "sequence" because it does not explicitly include the notion of a position. In String_Theory, Λ stands for an empty string, $\alpha \circ \beta$ denotes concatenation of two strings α and β in the specified order, and $|\alpha|$ denotes the length of a string α. "<*x*>" denotes the string containing the single entry *x*.

RESOLVE specifications use a combination of standard mathematical models such as integers, sets, functions, and relations, in addition to tuples and strings. The explicit introduction of mathematical models allows use of standard notations associated with those models in explaining the operations. Our experience is that this notation—which is precise and formal—is nonetheless fairly easy to learn to understand even for beginning computer science students, because they have seen most of it before in high school and earlier [19].

Using an **exemplar** stack variable S, the specification tells a client what is true of every stack variable. The **constraints** clause informs a client that a bounded stack cannot become too long. From the **initialization ensures** clause, a client has the guarantee that whenever a stack variable is declared, it has Λ for its initial value. (Occasionally, the initialization clause may specify a set of possible initial values, instead of a single value.) The practice of providing well-defined initial values, in situations where it is natural, is often helpful to avoid a class of routine software errors.

```
Concept Stack_Template( type Entry;
                                    evaluates Max_Depth: Integer);
        uses Std_Integer_Fac, String_Theory;
    requires Max_Depth > 0;

    Type_Family Stack  ⊆ Str(Entry);
        exemplar S;
        constraints |S| ≤ Max_Depth;
        initialization
            ensures S = Λ;

    Operation Push( alters E: Entry; updates S: Stack );
        requires |S| < Max_Depth;
        ensures S = ⟨#E⟩ ∘ #S;

    Operation Pop( replaces R: Entry; updates S: Stack );
        requires |S| > 0 ;
        ensures #S = ⟨R⟩ ∘ S;

    Operation Depth_of( restores S: Stack ): Integer;
        ensures Depth_of = ( |S| );

    Operation Rem_Capacity( restores S: Stack ): Integer;
        ensures Rem_Capacity = ( Max_Depth – |S| );

    Operation Clear( clears S: Stack );
end Stack_Template;
```

Figure 1: A Concept for Locally Bounded Stack_Template

The rest of the concept provides specifications of other Stack operations. Each operation is specified by a **requires** clause (precondition), which is an obligation for the caller; and an **ensures** clause (postcondition), which is a guarantee from a correct implementation. When clients violate the requirements, guarantees become void. Operation Push, for example, obligates the client that there be space on the stack object S for an additional entry. It guarantees that after Push is called, stack *S* will be updated to be the incoming value of entry (denoted by *#E*) concatenated with the incoming contents *#S* of the stack. Notice that the postcondition describes how the operation **updates** the value of *S*, but the return value of *E* (which has the mode **alters**) remains unspecified. In general, we allow postconditions to be loose to allow maximum flexibility for implementers.

In a similar fashion, the pre- and postconditions of the Pop, Depth_of, Rem_Capacity and Clear operations specify in string theoretic terms precisely what they will do. The **restores** mode in the specification of Depth_of and Rem_Capacity has the meaning that the stack S is unaffected by calls to these operations. Clear gives stack S the initial stack value Λ, and it gets this meaning, without an ensures clause, from the **clears** parameter mode. (The **evaluates** mode allows expressions to be passed as parameters.) The important point here is that by conceiving of stacks as strings, we can give a complete and coherent explanation of all of the operations on stacks. Absolutely no reference to details of any one implementation such as arrays, pointers, or linked lists is needed. Absence of such details simplifies understanding of the concept for clients yet provides developmental freedom for implementers of the concept, so long as clients and implementers adhere to the obligations and guarantees stated in the specification.

4. A GENERIC IMPLEMENTATION AND ITS PERFORMANCE SPECIFICATION

Figure 2 shows an array-based implementation (or realization) of Stack_Template in RESOLVE implementation notation. To employ a component in RESOLVE, a client programmer needs to instantiate the corresponding concept by supplying appropriate arguments and then needs to pick one of its implementations. The implementation given in Figure 2 is straightforward, and it provides a representation for type Stack and procedures for Stack operations using that representation. To understand the implementation, two aspects of the default array concept[1] in RESOLVE, both with performance implications, must be noted: (i) Default arrays are automatically initialized and (ii) the basic data movement array operator is swapping, i.e., you can swap in or out an entry at a specified location in the array using :=: notation. The performance and reasoning advantages of swapping over deep and shallow copying, respectively, are discussed elsewhere [7]. Swapping works without introducing aliasing, and it is always efficient if all large objects are represented implicitly using references by the compiler.

The **correspondence** or abstraction relation clause explains how to read the value in the *Contents* array as the string described in conceptualization. Here it says that the conceptual level string **Conc.**S corresponds to the string formed by concatenating together the *S.Contents* array entries from 1 to *S.Top* and then reversing this string. The mathematical definitions for string concatenation and reversal, denoted by Π and *Rev*, respectively, are formally defined in String_Theory. The correspondence clause is necessary to check that the code for procedures such as Push and Pop based on the array representation of a stack satisfy their respective specifications based on a string model of a stack in Stack_Template concept[2]. More generally, to facilitate mechanical or manual reasoning that a realization is correct with respect to its specification, a realization needs to be annotated with a variety of assertions (including invariants for loops).

The **conventions** or representation invariant clause contains constraints on the realization records for stacks that are essential to interpret them as strings, in addition to information that is needed to keep the implementations of all the various operations consistent. Here the convention is that *S.Top* always have a value that makes sense in the correspondence. It also has an additional representation convention of keeping all unused array elements "clean"[3]. By this convention, any array entry that is not a part of the stack has an initial value for the Entry type. In expressing the convention, we have defined and used a local mathematical predicate *Array_Is_Clean* on the stack representation. This definition is in turn based on the language-defined predicate **Is_Initial** for type Entry. Each procedure may assume that the Stack representation satisfies the convention when the procedure is called. At the end, each procedure needs to leave the representation in a state that satisfies the convention. The convention is the reason why a local (initialized) Entry variable is used in the Pop procedure.

4.1 Duration Specification

We begin this discussion by emphasizing that performance estimates should be described in terms that are meaningful to clients, without the knowledge of internal realization details[4]. In the displacement expression for type Stack, and in the duration expressions for finalization and Clear, for example, the reference to Stack S in the performance expressions stands for its conceptual string value, though the expressions are given inside the realization body. More exact estimates, however, may demand realization-specific details [18].

For a generic realization such as the Clean_Array_Realization of Stack_Template, the duration of exported procedures also depend on the parameters to the generic module, Max_Depth and Entry. For example, the initialization of a stack involves initialization of a record that contains an array of entries with size Max_Depth. The duration of initialization of a standard array depends on Max_Depth and the maximum time to initialize an object of generic type Entry, denoted by rough duration $\mathbf{R\_Dur}_{\mathit{EntryInitialization}}$. This leads to the **duration** clause **Is_O**(Max_Depth * $\mathbf{R\_Dur}_{\mathrm{Entry.Initialization}}$) for the implicit initializations in Stack initialization procedure, though the procedure itself has no code.

$\mathbf{R\_Dur}_{\mathit{Entry.Initialization}}$ is a rough approximation of duration $\mathbf{Dur}_{\mathit{Entry.Initialization(E)}}$. In general, for a procedure P with an input parameter X and an output parameter Y with a (relational) specification that allows one of several values to be returned in Y, we define "output independent" duration $\mathbf{R\_Dur}_{\mathrm{p}}$(X) to denote the maximum duration $\mathbf{Dur}_{\mathrm{p}}$(X, Y) for any result value Y. Entry initialization, like any other operation, may have a specification that allows one among several values to be the result of initialization, and hence, the duration for a call to Entry initialization technically depends upon the particular result value.

[1] RESOLVE includes a variety of array concepts. Amortized cost array implementations that do not initialize in "batch style" at the beginning are among them. To limit the scope of this presentation, we use the default arrays.

[2] To write realization assertions, mathematical models of the structures used in the realizations such as arrays and records are essential. In the default array concept, an array of entries is modeled as a mathematical function from integers to entries, and hence, *S.Contents* is treated as a function in the correspondence assertion. A record with 2 elements is viewed as an ordered pair in the record concept.

[3] A more efficient implementation without this convention is discussed in Section 5. However, the performance specification of such an implementation introduces the need to enhance the abstract model, and is slightly more complex.

[4] Performance estimates should be separated from details and readily accessible to clients without access to the implementation. We have included them as a part of the implementation in this paper so a reader can easily relate the estimates with the code that follows. It is clear that a support system can mechanically extract and display the performance information (along with definitions such as *Dur_Init_Entry_Fin*).

```
Realization Clean_Array_Realiz for Stack_Template;
        uses Record_Template, Static_Array_Template;

    Type Stack = Record
                Contents: Array 1..Max_Depth of Entry;
                Top: Integer
            end;
        Definition Array_is_Clean( SR: Stack ): $\mathbb{B}$ = (
            $\forall$ i: $\mathbb{Z}$, if SR.Top ≤ i ≤ Max_Depth then
                        Entry.Is_Initial( SR.Contents(i) );
        conventions
            0 ≤ S.Top ≤ Max_Depth and Array_is_Clean( S );
        correspondence
            conc.S = $\left( \prod_{i=1}^{S.Top} \langle S.Contents(i) \rangle \right)^{Rev}$ ;
        displacement Is_RO( $\sum_{E:\,Entry}$ Occurs_Ct(E,S)*Disp(E) +
                ( Max_Depth – |S| )*Entry.init_disp );

        initialization
            trans_displacement Is_RO( $\mathbf{Disp}_{Stack}(\Lambda)$ );
            duration Is_O( Max_Depth*
                        $\mathbf{R\_Dur}_{Entry.Initialization}$ );
        end;

        Definition Dur_Init_Entry_Fin: $\mathbb{R}$ =
        ( $\mathop{Max}_{E:\,Entry \ni Entry.Is\_Initial(E)} \left( \mathbf{Dur}_{Entry.Finalization}(E) \right)$ );

        finalization
            trans_displacement Is_RO( 0 );
            duration Is_O (
        $\sum_{E:\,Entry}$ Occurs_Ct(E, S) * $\mathbf{Dur}_{Entry.Finalization}$(E) +

            ( Max_Depth – |S| )*Dur_Init_Entry_Fin );
        end;

    Procedure Push( clears R: Entry; updates S: Stack );
        trans_displacement Is_RO( 0 );
        duration Is_O( 0 );

        S.Top := S.Top + 1;
        R :=: S.Contents(S.Top);
    end Push;

    Procedure Pop( replaces R: Entry; updates S: Stack );
        trans_displacement Is_RO( Entry.init_disp );
        duration Is_O( $\mathbf{R\_Dur}_{Entry.Initialization}$ +
                    $\mathbf{Dur}_{Entry.Finalization}$( #R ) );

        Var Fresh_Val: Entry;

        R :=: S.Contents(S.Top);
        S.Contents(S.Top) :=: Fresh_Val;
        S.Top := S.Top – 1;
    end Pop;

    Procedure Depth_of( preserves S: Stack ): Integer;
        trans_displacement Is_RO( 0 );
        duration Is_O( 0 );

        Depth_of := S.Top;
    end Depth_of;

    Procedure Rem_Capacity( preserves S: Stack ): Integer;
        trans_displacement Is_RO( 0 );
        duration Is_O( 0 );

        Rem_Capacity := Max_Depth – S.Top;
    end Rem_Capacity;

    Procedure Clear( clears S: Stack );
        trans_displacement Is_RO( $\mathbf{Disp}_{Stack}(\Lambda)$ );
        duration Is_O( Max_Depth*$\mathbf{R\_Dur}_{Entry.Initialization}$ +

        $\sum_{E:\,Entry}$ Occurs_Ct(E, #S) * $\mathbf{Dur}_{Entry.Finalization}$(E) +
        ( Max_Depth – |#S| )*Dur_Init_Entry_Fin );

            Var Fresh_Stk: Stack;

            S :=: Fresh_Stk;
    end Clear;

end Clean_Array_Realiz;
```

Figure 2: An Implementation Including Duration and Displacement Estimates

Where we are not interested in exact durations, we use approximate durations for operations with potentially relational behavior. We have included special notation **R_Dur** for output-independent duration approximation, because we expect it to be used frequently. The rough duration **R_Dur** can be derived mechanically from the duration expression for a procedure. For procedures with a purely functional specification, the rough duration approximation equals actual duration.

The duration expression for stack finalization has two parts, because to finalize a stack, entries in the array that correspond to elements of the stack as well as other array entries need to be finalized. In the expression, we have used *Occurs_Ct*, a mathematical definition that gives the number of times a given entry occurs in an arbitrary string. The first part of the duration expression sums up the durations to finalize all Stack entries using

Occurs_Ct. This expression is naturally based on the duration for finalization of an entry that in turn depends on the entry that is finalized. To express the duration to finalize the rest of the array entries that are all clean, we have defined and used a local approximate expression *Dur_Init_Entry_Fin*, which is the maximum duration to finalize an initial Entry with any initial value.

Procedure Push takes a constant time, denoted by **Is_O**(0)[5], only because the constant-time swap operator is used to transfer parameters, and to transfer an entry to and from the array. The duration for Pop involves finalization of the Entry supplied as a parameter to Pop, in addition to an Entry initialization. Duration expressions need to account for local variable initialization and finalization, though these procedures are invoked automatically. The Clear procedure is expensive in the realization presented in Figure 2 because it involves the destruction of a stack and creation of a new one. In an alternative realization, outlined in Section 5, it would be only necessary to set *S.Top* to 0.

Notice that the domains of the duration expressions such as those for initialization, finalization, Push and Clear are cross products involving a number of mathematical spaces. Mapping these spaces to natural numbers, even where possible, is both unnatural and complicated, justifying the big-O notations proposed in this paper.

4.2 Storage Displacement Specification

The problem of estimating storage capacity differs from duration estimation because unlike time, storage capacity displaced by objects of a program increases and decreases during execution. Whether the next procedure can be executed without running out of memory depends upon the capacity displaced by objects in scope at the time of the call and the additional storage capacity needed to execute the procedure. To be able to do such an analysis, each implementation module must have a capacity estimate containing a capacity usage specification for each type of object it provides and the incremental displacement required by each of its procedures. The capacity estimates for objects are presented as displacement functions from values of objects to storage units in natural number. Ideally these displacement functions are given in terms of the abstract values of objects, though precise estimates generally demand additional representation knowledge.

The capacity requirement of a procedure depends on the capacity required for its local variables as well as capacity needed for any changes the procedure makes to its parametric objects. In addition, the capacity requirements of lower-level procedures that are invoked should be considered. Based on these considerations, we provide a single "transitional" displacement (not transient) estimate for each procedure that specifies the additional or incremental storage displacement needed to execute the procedure successfully.

Unlike in the case of duration estimates, variance in capacity estimates of different realizations of a concept is usually more subtle. The **Is_O** relationship that we have used to approximate durations is usually too sloppy for displacement specification. For example, a container object represented using a two-directional list needs twice as much capacity to store addresses as one based on a one-directional list. In both cases, the displacement requirements are linear in terms of the number of elements in the container.

[5] To bound constant duration procedures, because of the handicap provision in the new definition, it is sufficient to write **Is_O**(0), though **Is_O**(1) is also accurate.

Similarly, the capacity requirement distinctions for different competing tree representations of objects cannot be shown using the **Is_O** relationship. In general, when objects are possibly large and arbitrary, the number of objects becomes important in capacity analysis.

We introduce a restricted ordering relationship, termed **Is_RO** for capacity estimation. This relationship holds only between expressions that agree in the degree of their leading terms and in the coefficients of those terms, unlike the **Is_O** relationship that is silent about the coefficients. However, the definition is sufficiently sloppy to ignore lower-order terms. This relationship, defined below, allows for only one additive fudge factor, that is more appropriately termed a *dunnage* allowance D (instead of a handicap H):

Definition (f: Dom→ℝ) **Is_RO** (g: Dom→ℝ): 𝔹 =

$$(\exists D: \mathbb{R} \ni \forall x: \text{Dom}, f(x) \le g(x) + D);$$

The displacement expressions in Figure 2 are given in **Is_RO** terms and are not meant to be exact. The figure provides a **displacement** function for objects of Stack type and **transition displacement** (using shorthand keyword **trans_displacement)** requirements for each procedure. The displacement expressions naturally depend on the generic parameters Max_Depth and Entry. The displacement calculation for a particular stack needs to account for both the entries in the array that correspond to a conceptual stack and the rest of the array entries that do not. To calculate the displacements of arbitrary entries of the generic Entry type, we use the Entry displacement function $\textbf{Disp}_{\text{Entry}}$ defined in the module that provides objects of type Entry. (It is simply referred to as **Disp** because from its parameter E, the subscript is obvious.)

In the initialization displacement expression, Entry.**init_disp** is a shorthand notation for maximum displacement of an initial entry, i.e., *Max* **Displacement** (E), **where** E: Entry ∍ Entry.**Is_Initial**(E). Since initial entry displacement comes up routinely, we have defined a keyword for the purpose. The definition of a maximal initial displacement for entries of arbitrary types raises an obvious question: Does there exist such a maximum even for types where no initial values are specified? The answer to this question becomes clear by recognizing that computational objects are constrained to be within bounds, such as computational Integers that are constrained to be within *Max_Int* and *Min_Int*. Since any allowed value of an object has to be within those bounds, maximal initial displacement is well defined. In particular, even for those objects commonly referred to as "unbounded" there are indeed computational bounds.

Stack initialization procedure requires transition displacement to create an empty stack. The expression uses function $\textbf{Disp}_{\text{Stack}}$ that refers to the displacement expression for Type Stack. It is easy to see that $\textbf{Disp}_{\text{Stack}}(\Lambda)$ becomes Max_Depth + Max_Depth * Entry.**init_disp**. Most other procedures have no significant storage requirements that show up in this rough displacement analysis. Parameter passing by swapping (instead of copying) is a key reason for this simplicity. The local variables in procedures Pop and Clear lead to the specified storage requirements for these procedures.[6]

5. RELATED WORK AND DISCUSSION

The importance of generic components [6] and the role of performance considerations in component-based software engineering are well documented [1, 5, 22, 24]. Designers of languages and developers of component libraries have considered alternative implementations providing performance trade-offs [2, 12, 13]. More sophisticated approaches that permit development of systems to allow parameterization and choice of implementation have been proposed [1, 21]. However, in addressing the basic question of how to specify reusable component performance, almost all previous results defer to classical big-O analysis [3, 10]. We have explained the deficiencies of classical estimates, and argued for a set of notations for specifying both space and time behaviors of reusable software components.

Though we have used big-O style notations to capture only upper bounds in this paper, the system proposed here is applicable regardless of the tightness of bounds. The duration and displacement clauses may contain tight expressions, in terms of number of low-level operations (e.g., integer assignments). To predict that a system would not come within a certain "safe limit" of exceeding its storage capacity, the displacement specifications also need to be precise and take into account language and compiler dependent overheads. The need for enhancing abstract models with more elaborate structures for tight time specification is discussed in [18]. Our work differs from the work in the real-time community, where the focus is typically on proving tight bounds of programs involving simple objects like integers [23]. Liu and Gomez, for example, consider parameters, loops, and recursion, and the issue of accurate time analysis [11]. Hehner is one of the first to consider formalization of space (including dynamic allocation) in a real-time setting [8]. However, issues of data abstraction and genericity are not raised or addressed.

The notations proposed here have natural extensions to handle situations where common storage pools are shared among multiple objects, as well as for dynamic, global storage allocation [4, 16]. The parameterized nature of the notations also makes them suitable for specification of non-trivial, performance-parameterized components [1, 21], though the resulting specifications are more complex. There are other complications as well [20]. We have considered a stack implementation that maintains the internal convention of keeping all unused array elements initialized. An alternative, more traditional, implementation might not use this convention. In that implementation, the Pop procedure may simply swap the entry in the array with whatever entry happens to be passed as a parameter to Pop. Clear will just set the Top index to 0. In this implementation, after a series of calls to Push and Pop, arbitrary entries will be found in the unused array locations. This implementation is faster than the one in Figure 2, but more space is wasted because arbitrary entries, in general, will occupy more space than initialized entries. To express the time to finalize a stack or the storage capacity used by a stack implemented in this way, it becomes essential to refer to the hidden contents, not visible from the outside. This situation is not atypical. Practical software components routinely trade off space for time.

Ideally, we would like to verify that a component or a component-based system satisfies not only the specified functional behavior, but also specified performance. In addition to specifications, a modular performance analysis system would also need internal assertions, such as "time elapsed" clauses for loops [15].

In summary, this paper offers a window into the much larger performance specification problem. Predictable component-based software engineering demands an engineering approach to ensuring predictability in performance. A first step in achieving this objective is in understanding how to specify the performance of reusable software components.

ACKNOWLEDGEMENTS

Several people have contributed important ideas to this work. We would like to thank past and present members of the *Reusable Software Research Groups* at Clemson University, West Virginia University, and Ohio State University. Our special thanks are due to Bruce Weide for his insightful technical comments as well as his suggestions for presentation of the material.

We also gratefully acknowledge financial support from our own institutions, from the National Science Foundation under grants CCR-9311702, DUE-9555062, and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project number P116B60717, and from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Education, or the U.S. Department of Defense.

REFERENCES

1. Batory, B., and Geraci, B. J., "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering,* 23(2): 67-82, February, 1997.
2. Booch, G. *Software Components With Ada.* Benjamin/Cummings, Menlo Park, CA, 1987.
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., *Introduction to Algorithms*, The MIT Press, Cambridge, MA, 1990.

[6] To be exact, the displacement expressions for Stack initialization and finalization operations should account for any transitional displacement required Entry initialization and finalization procedures. So should procedure Pop, where a local variable is declared. For Pop. For example, the transition displacement should be **Is_RO**(**Trans_Disp**$_{\text{Entry.Initialization}}$ + **Trans_Disp**$_{\text{Entry.Finalization}}$(#R)). In general, for arbitrary entries **Trans_Disp**$_{\text{Entry.Initialization}}$ may not be **Is_RO**(Entry.**init_disp**) , because local variables may be declared in the initialization procedure. Also, **Trans_Disp**$_{\text{Entry.Finalization}}$ may not be **Is_RO**(0), because local variables may be needed for finalization. An Entry with a recursive finalization procedure is an extreme case for where finalization requires considerable displacement. However, we have chosen to make simplifying assumptions here with the knowledge that the expressions work for most entry types and that they can be tightened in other situations as necessary.

4. Ernst, G. W., Hookway, R. J., and Ogden, W. F., "Modular Verification of Data Abstractions with Shared Realizations", *IEEE Transactions on Software Engineering 20,* 4, April 1994, 288-307.

5. Fleming, D., Sitaraman, M., and Sreerama, S., "A Performance Criterion for Object Interface Design," *Journal of Object-Oriented Programming,* July/August 1997, 52-63.

6. *Generic Programming*, eds. M. Jazayeri, R. G. K. Loos, and D. R. Musser, LNCS 1766, Springer, 2000.

7. Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Transactions on Software Engineering,* Vol. 17, No. 5, May 1991, pp. 424-435.

8. Hehner, E. C. R., "Formalization of Time and Space," *Formal Aspects of Computing*, Springer-Verlag, 1999, pp. 6-18.

9. Jones, R., Preface, *Proceedings of the International Symposium on Memory Management, ACM SIGPLAN Notices 34,* No. 3, March 1999, pp. iv-v.

10. Knuth, D. E., *The Art of Computer Programming*, Vol. 1, Addison-Wesley, 1968; Third Edition, 1997.

11. Liu, Y. A. and Gomez, G., "Automatic Accurate Time-Bound Analysis for High-Level Languages," *Procs. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LNCS 1474, Springer-Verlag, 1998.

12. Meyer, B., *Object-Oriented Software Construction,* Second Edition, Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

13. Musser, D.R.. and Saini, A., STL Tutorial and Reference Guide, Addison-Wesley, Reading, MA, 1996.

14. Ogden, W., F., and Sitaraman, M., Design, Specification, and Analysis of Software Components, WVU publications, 2000, 120 pages.

15. Ogden, W.F., *The Proper Conceptualization of Data Structures,* Dept. Computer and Information Science, Ohio State University, 2000.

16. Reddy, A. L. N., *Formalization of Storage Considerations in Software Design,* Ph.D. Dissertation, Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, 1999.

17. Sitaraman, M., and Weide, B.W., eds. Component-based software using RESOLVE. *ACM Software Eng. Notes 19,4* (1994), 21-67.

18. Sitaraman, M., "On Tight Performance Specification of Object-Oriented Software Components," *Proceedings of the 1994 International Conference on Software Reuse,* Ed. W. Frakes, IEEE Computer Society Press, November 1994, 149-157.

19. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E., "Reasoning About Software-Component Behavior," *Procs. Sixth Int. Conf. on Software Reuse*, LNCS 1844, Springer-Verlag, 2000, 266-283.

20. Sitaraman, M., "Compositional Performance Reasoning," *Procs. Fourth ICSE Workshop on Component-Based Software Engineering: Component-Certification and System Prediction*, Toronto, CA, May 2001.

21. Sreerama, S., Fleming, D., and Sitaraman, M., "Graceful Object-Based Performance Evolution," *Software - Practice and Experience,* Vol. 27, No. 1, January 1997, 111-122.

22. Szyperski, *C., Component Software: Beyond Object-Oriented Programming,* Addison-Wesley, 1998.

23. Special issue on Real-Time Specification and Verification, *IEEE Trans. on Software Engineeering*, September 1992.

24. Special section: Workshop on Software and Performance, Eds., A. M. K. Cheng, P. Clemens, and M. Woodside, *IEEE Trans. on Software Engineeering*, November/December 2000.