Performance Analysis Based upon Complete Profiles

Joan Krone
Denison University
Granville, Ohio 43023
+1 740 587 6484
krone@denison.edu

William F. Ogden
The Ohio State University
Columbus, Ohio 43210
+1 614 292 6004
ogden@cse.ohio-state.edu

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

ABSTRACT
A system for engineering and verifying component-based software must include mechanisms for specifying abstractly not only the complete functionality of components but their exact performance as well. This paper introduces *profiles* as a first-class construct for complete, independent specification of performance in higher-level languages. Using profiles, a developer can select from an assortment of implementations for a particular functionality the one that best suits his needs with respect to speed and memory usage. Equally importantly, he can define the expected performance of larger scale components using compositions of the profiles of their constituent (possibly as yet unimplemented) components. To support scalability, the profile construct facilitates abstraction in performance specifications as well as performance composition and analysis.
.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *performance measures*

General Terms
Software Engineering, specification, verification.

Keywords
Components, performance, reuse, specification.

1. INTRODUCTION
In order to have an effective system for engineering component-based software, it is essential to have a specificational framework that supports description of those aspects of a component that are relevant to its deployment and that implicitly supports suppression of other irrelevant aspects. The functional aspect is typically the most important, and so developing a framework for its specification has been the focus of much research. However, a framework is not adequate until it includes a mechanism for completely describing component performance.

SAVCBS 2006. November 10-11, Portland, Oregon, USA.

Factoring out performance specifications seems to be a common practice in the engineering of components. An auto manufacturer, for example, sets functional limits on the dimensions of tires that can be used but leaves to tire suppliers such performance specifications as traction, tread life, etc. As in the case of auto tires, a good conceptualization of functional behavior will admit a broad assortment of realizations with varying performance characteristics.

Currently performance specifications for software components are usually treated in a rather off-hand manner. Often they're given as gross Big-O estimates, typically in terms of imprecisely-specified parameters ill suited to object oriented programming (a problem we addressed in [11]). Alternatively, they're presented as exact durations for particular "representative" examples run on particular hardware, which data isn't ordinarily of much use for predicting behavior in future applications of components. In [11], we introduced language mechanisms for including exact performance specifications within each realization for a given component. We used an enhancement for a stack component to emphasize the important role of abstraction by showing that our approach permitted performance specification to be established without knowledge of how the stack component was implemented.

Subsequently we have found that there are important advantages to separating performance specifications not only from the component concepts but also from the realizations for the concept. Since the functionality of a component can be employed independent of the performance characteristics of its various implementations, those various performance specifications obviously don't belong in a component's conceptualization, where all its functional characteristics are formally specified. The principal advantage of separating performance specifications from particular realizations is that it supports additional reuse of specifications. As we've discovered, the performance of alternative implementations for a component often differs only in ways that can easily be parameterized in an appropriately abstract specification. Such a separation of specifications also makes it easier to document the performance of hardware components that are often constituents of larger (embedded) systems.

We introduce the *profile* then as a first class specificational construct for recording performance characteristics. Profiles have the virtue of allowing the designer of a component implementation to summarize its expected performance in a concise form that masks implementation details. At the same time, a prospective client for the functionality of the component

can use the profiles of its various implementations to select the one that best suits his performance objectives.

Since a profile is to act as a *performance contract* between client code and implementation code, it should become an artifact of the software development process with an importance similar to that of a functionality specification contract for a component. This makes it an entity that is independent of top-down or bottom-up development methodologies. Typically, development of a a good profile demands simultaneous attention to the desires of the clients and to the possibilities open to the implementers, regardless of whether implementation or client code exists at the time. In the same way that abstract specifications of functional behavior provide essential guideposts in development of modular component-based systems, profiles provide analytic yardsticks for checking the adequacy of system performance.

Since a performance profile is of necessity expressed in the context of a functional specification, it is not surprising that performance specification and verification potentially involves every complexity that can arise in functionality specification and verification. Moreover, since overall performance depends upon every detail of an implementation, its specification poses several new challenges. One of them is to aggregate these details into simplified, abstract specifications so that that clients can keep their focus on the larger picture as they the higher level code. Another is to formulate expressions for the performance of generic components, since performance specifications for their parameters are not fixed when their profiles are written. Our examples illustrate how to cope with all this complexity.

2 A PROFILE EXAMPLE

In order to ensure the generality of our profile mechanism proposed in this paper, we have tested it by creating performance specifications for a variety of software components, including a layered component-based system which addresses the issue of scalability. Since our objective here is to introduce the basic ideas in developing complete profiles that only make sense in the context of a thoroughly understood component, we will forego complexity of more sophisticated components and instead use the familiar generic stack component, as we did in [11]. For it, the functional specifications are given in Figure 1 in Resolve.

```
Concept Stack_Template( type Entry;
                        evaluates Max_Depth: Integer );
        uses 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| );
      ⋮
end Stack_Template;
```

Figure 1: Specification for a Stack_Template

Figure 1 shows a formal, conceptual client view of a generic bounded Stack component, parameterized by the type of entries to be contained in stacks and the maximum depth to which a stack can grow. (The **evaluates** mode is used to indicate that an expression may be passed as the maximum Stack depth.)

The Stack Template concept uses mathematical String Theory, a development of which is given in [20], to formalize stacks. The notation **Type_Family** is used where the stack formalization is introduced in order to highlight the generic nature of the concept by reminding that it involves a whole family of Stack types, which differ depending upon the particular Entry type and *Max_Depth* parameters supplied at the time of instantiation.

The concept provides specifications of typical Stack operations, each specified by a **requires** clause (precondition), which is an obligation for all callers, and an **ensures** clause (postcondition), which is a result guarantee from any correct implementation.

For example, the *Pop* operation updates the value of the stack parameter *S* by removing its top entry and using it to replace the value of the parameter *R*. This result is guaranteed by the **ensures** clause $\#S = \langle R \rangle \circ S$ once we know that $\#S$ refers to the previous value of *S*, that $\langle R \rangle$ is the single entry string containing *R*, and that $\circ$ is the concatenate operation for strings. The *Clear* operation gives stack *S* the initial stack value Λ (empty), and it gets this specification not based upon an **ensures** clause but instead based upon the **clears** parameter mode.

The important point here is that, by conceiving of stacks as strings, it is possible to give a complete and coherent explanation of all of the operations on stacks. Absolutely no reference to details of any particular implementation such as arrays, pointers, or linked lists is needed. This hiding of client irrelevant information by reconceptualization of objects is an equally critical feature for any satisfactory performance specification mechanism.

2.1 A Performance Profile for the Stack

In [11], we addressed the basic problems of adding performance specifications to **realization** code and of developing a reasoning system to verify that such specifications are accurate. That work was sound as far as it went and served as the basis for subsequent work on specification of performance properties in JML, and analysis of dynamic heap space usage in [1]. However, the earlier work doesn't fully address the larger software engineering scalability concerns of separating out concise and comprehensible summaries of the performance of component implementations and of structuring them in such a fashion that they support the derivation of analogous specifications for large components produced as compositions of smaller ones. Here we write performance specifications called ***profiles*** that represent a class of implementations, thereby removing these specifications from individual realizations, and remaining at a level of abstraction allowing for multiple realizations.

Since some alternative implementations of generic concepts such as stacks provide substantively different performance trade-offs, they will of necessity have different ***profiles***. The performance ***profile*** in Figure 2, named *SSC*, is suitable for a class of *S*tack implementations that are *S*pace-*C*onscious, i.e., ones that consider space to be more important than time. The profile is written without making any assumptions about the generic type *Entry* or *Max_Depth*, and therefore, the expressions have to be compositional and presented in terms of these parameters.

One of the key elements in the specification of a ***profile*** that's free of unnecessary implementation details, is the notion of a **defines** specification clause. Whereas a typical (mathematical) definition provides an immediate definiens for its definiendum, the **defines** clause allows a profile to name a definiendum for use within the profile, but to defer to each implementation the provision of a particular definiens. An implementation can then provide a specific definiens for each **defines** deferred definiendum based upon its exact code. So the **defines** construct provides a second mechanism whereby profiles can achieve appropriate independence. Whereas *Entry* and *Max_Depth* are traditional parameters whose values come down from clients, the deferred constants SSC_I, SSC_D, etc. seen here can be viewed as parameters whose values come up from implementations.

A performance ***profile*** is intended to document the behavior of a class of implementations in terms understandable to clients of the concept and generally simpler than an exhaustive description of each implementation. A ***profile*** provides the following information. For each operation, there is a **duration** clause – a non-negative real number valued expression – that places a bound on the time taken by the operation in terms of the parameters supplied to the operation.

For each operation, there is a manipulation displacement clause (abbreviated as **manip_disp),** a natural number valued expression that bounds the minimum additional space that is necessary to execute the operation above and beyond what is occupied by all objects currently in scope. Since memory usage may increase and decrease during the execution of a complex procedure, this clause expresses the "high water mark" in terms of the parameters to the operations. In order to use this information to determine whether there is enough space to execute the next call with a certain collection of arguments, a caller needs to be able to determine the space occupied by all current objects. Thus, ***profiles*** for implementations that provide types (and therefore permit creation of objects) include a displacement clause – also a natural number – that describes how much space is used by a variable (e.g., a Stack variable), given its abstract value (a string of entries). We begin the discussion with this clause, following Figure 2.

Profile SSC **short_for** Space_Conscious **for** Stack_Template;

Defines SSC_I, SSC_{I1}, SSC_F, SSC_{Po}, SSC_{Pu}, SSC_C, SSC_{C1}, SSC_{Dp}, SSC_{RC}: $\mathbb{R}^{\geq 0}$;

Defines SSC_D, SSC_{MI}, SSC_{MF}, SSC_{MPo}, SSC_{MPu}, SSC_{MC}, SSC_{MDp}, SSC_{MRC}: $\mathbb{N}$;

Type_Family Stack;

Definition Cnts_Disp(α: Str(Entry)): $\mathbb{N}$ = ($\sum_{E: Entry}$ Occurs_Ct(E,α)·Entry.**Disp**(E));

Displacement SSC_D + Cnts_Disp(S) + (Max_Depth – |S|)·Entry.**I_Disp**;

Initialization;

duration SSC_I + (SSC_{I1} + Entry.**I_Dur**)·Max_Depth;

manip_disp SSC_{MI} + Entry.**IM_Disp** + (Max_Depth – 1)·Entry.**I_Disp**;

Oper Pop(**replaces** R: Entry; **updates** S: Stack);

duration SSC_{Po} + Entry.**I_Dur** + Entry.**F_Dur**(#R);

manip_disp SSC_{MPo} + Max(Entry.**IM_Disp**, Entry.**FM_Disp**(#R));

Oper Push(**alters** E: Entry; **updates** S: Stack);

ensures Entry.**Is_Init**(E);

duration SSC_{Pu};

⋮

end SSC;

Figure 2: A Performance Profile

The Displacement Clause

We note that the same ideas discussed here suffice whether or not stacks are bounded a priori, as also noted by Atkey[1]. For example, if the stack elements are allocated only when needed instead of initially in an array, then the displacement will be less and it would not include the last term seen here. However, to make our discussions concrete, we consider implementations that allocate and initialize an array of entries of size *Max_Depth* whenever a new *Stack* is created. An implementation might use a simple representation such as the one shown below:

Type Stack = **Record**
Contents: **Array** 1..Max_Depth **of** Entry;
Top: Integer
end;

Within this context, one class of implementations can be characterized as placing high priority on minimizing space usage for a *Stack* variable, by following a space-conscious convention (or representation invariant): All entries in array locations beyond those that correspond to the conceptual stack value are kept initialized. For a stack containing complex objects such as trees, for example, this convention leads to minimal space usage because unused array locations contain only empty trees instead of arbitrary trees.

Though we have divulged the representation details above in order to provide a concrete example for readers of this paper, a performance profile must be understandable to users based only upon the mathematical conceptualization of stacks as strings as given in Figure 1. Accordingly, the displacement clause in this performance profile expresses the space occupied by a stack S using only its abstract string value:

Displacement SSC_D + Cnts_Disp(S) + (Max_Depth – |S|)·Entry.**I_Disp**;

There are three terms in this expression. The first term is the constant SSC_D, and it represents the fixed space overhead in any Stack object (e.g., an Integer index into the array that is used to keep track of the current top). The actual definition for this constant is implementation-specific and will be specified within the implementation; the profile merely provides a placeholder for this constant and others by listing them in the **defines** clause. The second term captures the space occupied by the entries that have been pushed onto a stack. To express this term, we have introduced a locally defined contents displacement function *Cnts_**Disp**(S),* which totals for each entry *E* in a stack *S* its displacement *Entry.**Disp**(E)* times *Occurs_Ct(E, S*), the number of times *E* occurs in *S*.

The last term in the displacement expression is the product (*Max_Depth* − |*S*|)·*Entry.**I_Disp***), and it accounts for the space taken by unused array entries (all of which are assumed by this profile to have initial values). Here, *Entry.**I_Disp*** denotes the space used by an entry with an initial value. Using the given expression, it is easy to see that for an empty stack with abstract string value Λ, the displacement *Stack.Disp(*Λ*)* becomes SSC_D + *Max_Depth*·*Entry.**I_Disp***.

Specification of Initialization

In the class of implementations under discussion here, when a Stack variable is initialized, *Max_Depth* number of entries are created and initialized. Therefore, initialization duration includes the factor *Entry.**I_Dur***·*Max_Depth*, which is the product of the duration for initializing a variable of type Entry, i.e., *Entry.**I_Dur*** and *Max_Depth*, the number of entries to be initialized. The expression includes additional constant overhead per entry, denoted by SSC_{I1}, as well an overall constant overhead denoted by SSC_I. The actual definitions for these implementation-specific constants will be given in the implementations. (If the Stack elements are allocated only when needed instead of using an array, then initialization will take a constant time, and the cost of object creation will be moved to the *Push* operation.)

The initialization **manip_disp** clause expresses the minimum storage space necessary to create a new stack variable. Recall that *Entry.**I_Disp*** denotes the space taken by an entry with an initial value. To create a Stack representation with *Max_Depth* initial entries, the necessary displacement is roughly *Entry.**I_Disp***·*Max_Depth*. The expression given in the profile differs slightly because the procedure to create an initial entry might need more space than what is strictly necessary for storing an initial entry. This would be the case if Entry is a non-trivial type, and creating an initial value for it requires creation and use of other local variables. Therefore, suppose that *Entry.**IM_Disp*** denotes the manipulation space necessary for initial entry creation. Then the highest watermark in space usage during *Stack* initialization occurs when *Max_Depth* − 1 new entries have been created and the *Entry* initialization operation is being invoked to initialize the last entry. Therefore, this is the minimum space necessary to initialize a new *Stack*. The expression includes an implementation-specific constant as well.

Specification of Pop

To explain the expressions for *Pop*, we consider the following code that might have been written for a space-conscious implementation.

```
Procedure Pop( replaces R: Entry; updates S: Stack );
    Var Fresh_Val: Entry;
    R :=: S.Contents(S.Top);
    S.Contents(S.Top) :=: Fresh_Val;
    S.Top := S.Top – 1;
end Pop;
```

In this implementation, we have used the swap operator ":=:", instead of assignment, to move *Entry* values and to access array contents. The reasoning and efficiency advantages of swapping over reference assignment and representation assignment of arbitrary entries, respectively, are discussed in detail elsewhere [8]: Swapping enables reasoning without introducing aliasing; its implementation is efficient because compliers can represent large objects internally using references and merely exchanging the references in constant time. (If entries are copied, then the same principles of specifying performance expressions would still be adequate, except that the performance expressions need to account for copying.)

The second swap statement in the code is necessary to satisfy the space-conscious convention. By declaring a local Entry variable (which is automatically initialized) in the *Pop* procedure and swapping it into the array, we make sure that the arbitrary entry *R* that might have been supplied as the incoming parameter to *Pop* does not go into the array and violate the convention. At the end of the code, the local variable that then contains the incoming value of parameter *R* is released or finalized. The performance specification of Pop is expressed in user-oriented terms in the profile:

```
Operation Pop( replaces R: Entry; updates S: Stack );
    duration SSC_Po + Entry.I_Dur + Entry.F_Dur(#R);
    manip_disp SSC_MPo +
            Max( Entry.IM_Disp, Entry.FM_Disp(#R) );
```

The **duration** expression includes the time to initialize a new Entry variable. Finalization depends on the Entry that is finalized, and thus, the time to finalize is given in terms of the incoming value of parameter *R*. The definition for the deferred constant SSC_{Po} in the duration expression for *Pop* code is given internally in each implementation. For the present example, it might be defined as:

Definition SSC_{Po}: $\mathbb{R}^{\geq 0}$ = $\mathbf{Dur_{Call}}(2)$ + 2·Array.**Dur**$_{:=:}$ +
6·Record.**Dur**$_{.}$ + Int.**Dur**$_{:=}$ + Int.**Dur**$_{\_}$;

This constant includes the *time to call a procedure with 2 parameters, denoted by* $\mathbf{Dur_{Call}}(2)$, array and record accesses, and Integer operations. This definition is relegated to the implementation because it provides too much information to include in a profile for clients and it is expressed in terms of implementation details that should not be visible to them. Placing the definition in the **profile**, in addition to hard wiring it, would seriously compromise information hiding and hinder modularity in reasoning.

How much space is necessary to call *Pop* beyond what is already taken up by its parameters? It is the maximum of the displacement necessary to initialize a new variable, i.e., *Entry.**IM_Disp*** (Entry initialization manipulated displacement) or finalize the incoming parametric entry, i.e., *Entry.**FM_Disp**(#R)*.

One other aspect of interest in the performance **profile** is the additional **ensures** clause for the Push operation. In particular, using the predicate *Entry.**Is_Init**(E)* that is true only if *E* has an initial *Entry* value[1], the ensures clause tells a user that *E* will be initialized after a call to *Push(E, S)*. While this information, which appears only in the performance profile, cannot be used by a client program in establishing functional correctness, it can be used for reaching **displacement/duration** conclusions, as illustrated in Section 3. Unlike *Pop*, the *Push* and *Depth_of* procedures have constant performance expressions.

Performance **profiles** are useful for component clients, enabling them to select prudently from among a variety of

[1] We use a predicate here instead of asserting *E* = *Entry.**Init*** or equivalent, because initializations may be specified to give an object one of many initial values.

implementations for a particular concept that provide interesting performance trade-offs. They are also important for independent development and modular analysis of component-based systems in the same way that abstract specifications of functional behavior are useful. For example, performance of other components that reuse the *Stack* **concept** can be derived from the performance profile of the chosen *Stack* implementation. To illustrate how profiles for a component built on other components can be presented parametrically, we analyze code for a component built on *Stack* objects and operations. The example specification for a *Flip* operation to invert a stack is given below. It is an **enhancement** or conceptual extension to the *Stack_Template* described previously. In the **ensures** clause, *Rev* denotes the mathematical string reversal operator.

Enhancement Flipping_Capability **for** Stack_Template;

Operation Flip(**updates** S: Stack);

ensures $S = \#S^{Rev}$;

end Flipping_Capability;

2.2 Profile Specification of Flip

A given implementation of *Flip* may exhibit different performance behaviors, depending on the profile of the *Stack* implementation that is used in conjunction with *Flip*. It becomes possible to express this performance dependence of one component upon another quite elegantly, if profiles are available as first class constructs in a language. To illustrate how this is done, we show profile *SSCF* for *Flip* based on the *SSC* profile of *Stack_Template*.

Profile SSCF **short_for** Space_Conscious_Stack_Flip **for**

Flipping_Capability **for** Stack_Template **with_profile** SSC;

Defines $SSCF_{F1}$, $SSCF_{F2}$: $\mathbb{R}^{\geq 0}$;

Defines $SSCF_{FMC1}$, $SSCF_{FMC2}$: $\mathbb{N}$;

Operation Flip(**updates** S: Stack);

duration $SSCF_{F1}$ + Entry.**I_Dur** + Stack.**I_Dur** + Entry.**F_IV_Dur** + Stack.**F_IV_Dur** + ($SSCF_{F2}$ + Entry.**I_Dur** + Entry.**F_IV_Dur**)·|#S|;

manip_disp ($SSCF_{FMC1}$ + Entry.**I_Disp** + Stack.**I_Disp**) + Max($SSCF_{FMC2}$, Entry.**IM_Disp**, Entry.**F_IVM_Disp**);

end SSCF;

The abstract performance specifications in the profile above are given in terms meaningful to clients of the *Flipping_Capability*. In particular, the profile of *Flip* can be understood, without knowing any implementation details of either the *Stack_Template* or the *Flipping_Capability* enhancement.

To motivate the specifics of the particular performance expressions in the profile, we consider a concrete implementation of *Flip* in this subsection. The implementation contains concrete definitions for constants used in the SSCF profile, such as $SSCF_{F1}$ and $SSCF_{F2}$. The loop is annotated with the **maintaining** (loop invariant) and **decreasing** (progress metric) clauses necessary for an automated system to prove that the code satisfies its functional specification for flipping the Stack. In addition, the loop specification includes **elapsed time** and **manipulated displacement** expressions [11] needed to prove the correctness of the code with respect to its performance **profile**.

Due to space constraints, we present and analyze just the timing-related assertions. Since the code for *Flip* relies only on the specification of operations in the *Stack_Template* and not on any particular implementation, modular reasoning about the functional correctness of the code can be done regardless of the *Stack* implementation chosen.

Realization Obvious_F_C_Realiz **for** Flipping_Capability

with_profile SSCF **of** Stack_Template **with_profile** SSC;

Definition $SSCF_{F1}$: $\mathbb{R}^{\geq 0}$ = ($\mathbf{Dur}_{\mathbf{Call}}(1)$ + (SSC_{Dp} + Int.$\mathbf{Dur}_{\neq}$) + $\mathbf{Dur}_{:=:}$);

Definition $SSCF_{F2}$: $\mathbb{R}^{\geq 0}$ = (SSC_{Dp} + Int.$\mathbf{Dur}_{\neq}$ + SSC_{Po} + SSC_{Pu});

Definition $SSCF_{FMC1}$: $\mathbb{N}$ = ⋯

Definition $SSCF_{FMC2}$: $\mathbb{N}$ = ⋯

Procedure Flip(**updates** S: Stack);

Var Next_Entry: Entry;

Var S_Flipped: Stack;

While (Depth_of(S) ≠ 0)

affecting S, S_Flipped, Next_Entry;

maintaining $\#S = S\_Flipped^{Rev} \circ S$ **and**

Entry.**Is_Init**(Next_Entry);

decreasing |S|;

elapsed_time ($SSCF_{F2}$ + Entry.**I_Dur** + Entry.**F_IV_Dur**)·|S_Flipped|;

manip_disp ⋯

do

Pop(Next_Entry, S);

Push(Next_Entry, S_Flipped);

end;

S :=: S_Flipped;

end Flip;

end Obvious_F_C_Realiz;

2.3 Durational Analysis of Flip

The **duration** expression for *Flip*, in addition to a constant term $SSCF_{F1}$, has three parts: duration for local variable initialization, for local variable finalization, and for loop execution. First we assume that a *Stack* component with profile *SSC* is used. The duration expression to initialize the two local variables – an entry and a stack – is straightforward, and it is the sum of *Entry.**I_Dur*** and *Stack.**I_Dur***. Unlike initialization, the time for finalization of the two local variables depends on the values of the local variables at the time of finalization. Therefore, we need to understand what their values would be at the end of the code. Here, the Stack *S_Flipped* that is finalized is empty, because *S* is empty just before the swap statement. Therefore, the duration expression also includes the term *Stack.**F_IV_Dur*** – the time to finalize a stack with initial value. The local variable *Next_Entry* also has an initial value just before finalization. To

see why, notice that the loop maintains the invariant *Entry.**Is_Init**(Next_Entry)*, based on the extended **ensures** clause for the *Push* operation in the profile *SSC*, which in our version of Stack, guarantees that after *Push* the parametric Entry is initialized. Therefore, the duration of finalizing the Entry at the end of the code is *Entry.**F_IV_Dur*** – the time to finalize an entry with an initial value.

The loop executes |#S| times. The time for each iteration includes a constant term arising from calls to *Depth_of*, *Push*, and the loop branching activity. In addition, we note from the *SSC* profile that every call to *Pop(R, S)* takes time SSC_{Po} + *Entry.**I_Dur*** + *Entry.**F_Dur**(#R)*. In the code given above, the *Next_Entry* that is supplied to Pop is the entry resulting from the previous to call to *Push*. Since the ensures clause for *Push* in SSC profile guarantees that *Push* initializes its *Entry* parameter, we are guaranteed that *Pop* is only supplied initial entries in every call. Therefore, *Pop* needs to finalize only initial entries and the time for each call to *Pop* simplifies to SSC_{Po} + *Entry.**I_Dur*** + *Entry.**F_IV_Dur***. Given these considerations and the matching definitions of constants $SSCF_{F1}$ and $SSCF_{F2}$, the elapsed time estimate for the loop is documented in the implementation as:

($SSCF_{F2}$ + Entry.**I_Dur** + Entry.**F_IV_Dur**)·|S_Flipped|

2.3 Validity of the Elapsed Time Estimate

This elapsed time estimate is used in proving the performance correctness of *Flip*. A part of the proof that verifies that the given elapsed time estimate is valid is given in the table below.

State	Path Condition	Assume	Confirm
While (Depth_of(S) ≠ 0) **affecting** S, S_Flipped, Next_Entry; **maintaining** #S = S_Flipped^{Rev}∘S **and** Entry.**Is_Init**(Next_Entry); **decreasing** |S|; **elapsed_time** ($SSCF_{F2}$ + Entry.**I_Dur** + Entry.**F_IV_Dur**)·|S_Flipped|; **do**			
2	$\|S_2\| \neq 0$	Entry.**Is_Init**($Next\_Entry_2$) ∧ $\mathbf{ET}_2$ = ($SSCF_{F2}$ + Entry.**I_Dur** + Entry.**F_IV_Dur**)·$\|S\_Flipped_2\|$ ⋯	…
Pop (Next_Entry, S);			
3	$\|S_2\| \neq 0$	$S_2 = S_3 \circ \langle Next\_Entry_3 \rangle \wedge$ $S\_Flipped_3 = S\_Flipped_2 \wedge$ $\mathbf{ET}_3 = \mathbf{ET}_2 + (SSC_{Po}$ + Entry.**I_Dur** + Entry.**F_Dur**(Next_Entry2)) ⋯	…
Push (Next_Entry, S_Flipped);			
4	$\|S_2\| \neq 0$	Entry.**Is_Init**($Next\_Entry_4$) ∧ $S_4 = S_3 \wedge S\_Flipped_4 = S\_Flipped_3 \circ \langle Next\_Entry_3 \rangle \wedge$ $\mathbf{ET}_4 = \mathbf{ET}_3 + SSC_{Pu}$ ⋯	…
Confirm ET4 = ($SSCF_{F2}$+ Entry.**I_Dur** + Entry.**F_IV_Dur**)·$\|S\_Flipped_4\| \wedge$ ⋯			
end;			

The table shows only a part of an inductive proof: verification conditions corresponding to the inductive potion of the proof to confirm the invariance of the elapsed time estimate. In the table, which is based on [26]], we **assume** at the beginning of the loop (numbered state 2 in the figure) the elapsed time estimate holds. We then **confirm** at the end of the loop (state 4) that the estimate when evaluated there is correct. The assumptions in states 3 and 4 come from the functional and performance specifications of operations *Push* and *Pop*. Variable names are subscripted with the state number to distinguish their values in different states. The *verification variable* **ET** stands for the elapsed time. Given the assumptions, a verifier can conclude that $\mathbf{ET}_4$ satisfies its equation if $\mathbf{ET}_2$ satisfies its equation. We have omitted the base case for the inductive proof, assertions outside the loop, and functionality-related assertions, not necessary for the above proof.

3. SCALING UP

Two important scalability questions arise in generalizing the utility of the profile construct:

1. Can profiles for layered components be expressed abstractly?
2. How complicated will profiles get when components are used to put together a layered system?

To address these questions we designed and specified a *spanning forest* component that we built using a *prioritizer* and a *coalescable equivalence relation* component, among others, and specified all components fully for both functionality and performance.

We answer the first question affirmatively noting that it was possible to write a fully descriptive profile for the top layer of the system without filling in the details for the components upon which it was layered.

The second question is one of concern, since the stack example may give the impression that the number of lines of specifications in a profile may approach the number of lines of executable code. However, we note that the stack component has an unusually small number of lines of code, and that the complexity of the profile is dominated by its parameterization. Moreover, although it may seem counter-intuitive, it turns out that when layering up, the profile for a higher-level component is usually no longer than that for a lower level one, while the aggregate number of lines of executable code grows considerably. For example, in the case of the spanning forest, the ratio of lines of performance specification to executable code is closer to one to three, rather than one to one, indicating that the depth of layering in a system is not an indicator of the need for longer profiles.

Our research has also shown that the profile construct is essential for documenting concisely the various performance specifications of a layered component, such as the spanning forest component, that result when alternatives are considered for the performance of a constituent component such as the prioritizer.

4. RELATED WORK AND DISCUSSION

The importance of performance considerations for software engineering (e.g., [4], [14], [17]), in general, and for software components, in particular, has been widely acknowledged. Designers of languages and developers of component libraries have emphasized the need for alternative implementations in order to provide performance trade-offs [3], [16], [18]. The importance

of generic programming and of alternative implementations is being increasingly recognized, as is evident from the evolving designs of C#, C++, and Java.

In order for component users to choose from multiple implementations and analyze performance of component-based systems in a modular fashion, a formal system for performance specification is necessary. Balsamo, et al., in surveying various efforts in performance analysis [2], note that "Although several of these approaches have been successfully applied, we are still far from seeing performance prediction integrated into ordinary software development" and conclude that one of the unresolved problems is the lack of software notations that allow for easily expressing performance. The profile construct proposed here for extending specification (and programming) languages to support specifying performance is a contribution to integrating performance considerations into software development.

A general performance specification system should be flexible, allowing specifiers to express performance in terms of abstractions that are appropriate for the problem at hand. This emphasis on abstraction and generic components in specifying both time and space usage of components also makes the ideas discussed in this paper quite different from the work in the real-time community (e.g., [7], [23]) where timing deadlines and concurrency are the focus.

Expression of tight timing constraints is an active area of research [6], [15]. Elsewhere, we have detailed how the expressiveness issues that arise in tight specification of performance at the source code level can be addressed using intermediate abstraction models [28].

Hehner has built on the work of Shaw [22], to formalize time and space analysis of a recursive procedure at the source code level [9]. Our earlier work and the work of Schmidt and Zimmermann [21] have considered space complexity issues for components. Working within the context of functional programs Unnikrishnan, et al. and Hofmann and Jost have addressed issues in bounding the space usage of functional programs under various assumptions using program-level source code analysis [10], [27]. Ultimately, compositional performance analysis needs to be combined with advances in verification of functional behavior in the presence of data abstractions (e.g, [5], [19], [25], [26]) because assertions from functional correctness are necessary for establishing performance correctness.

We have introduced profiles as a first class language construct for modular specification and analysis, providing a vocabulary for stating time and space constraints. The construct supports both generics and compositionality. Based on the construct, as Atkey [1] has shown recently, mechanisms for other behavioral specification language and implementation language combinations can be developed, provided the particulars of the language features are carefully accommodated in specifications.

5. ACKNOWLEDGMENTS

Several members of our research groups have contributed important ideas to this work. Our special thanks are due to Gary Leavens and Bruce Weide for their comments. We gratefully acknowledge financial support from the U.S. National Science Foundation under grant CCR-0113181 and a grant from the U.S. National Aeronautics and Space Administration through the SC Space Grant Consortium.

6. REFERENCES

[1] Atkey, J., "Specifying and Verifying Heap Space Allocation with JML and ESC/Java2", *Proceedings of the ECOOP Workshop Formal Techniques for Java-like Programs*, Nantes, France, July 2006; available at: http://www.disi.unige.it/person/AnconaD/FTfJP06/

[2] Balsamo, S., Di Marco, A., and Inverardi, P., "Model-Based Performance Prediction in Software Development: A Survey", *IEEE Transactions on Software Engineering*, 30(5), May 2004, 67-82.

[3] Booch, G. *Software Components With Ada.* Benjamin/Cummings, Menlo Park, CA, 1987.

[4] Cheng, A. M. K., Clemens, P., and Woodside, M., eds. Special section: Workshop on Software and Performance. *IEEE Trans. on Software Engineering 26*, 11/12, November/December, 2000.

[5] 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.

[6] Gomez, G. and Liu, Y. A., "Automatic time-bound analysis for a higher-order language," *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02)*, Portland, Oregon, USA, January 14-15, 2002, ACM SIGPLAN Notices 37(3), March 2002.

[7] Hayes, I.J. and Utting, M., "A Sequential Real-Time Refinement Calculus," Acta Informatica 37, 2001, 385-448.

[8] 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, 424-435.

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

[10] Hofmann, M. and Jost, S., "Static Prediction of Heap Space Usage for First-Order Functional Programs," *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2003, 185-197.

[11] Krone, J., Ogden, W. F., and Sitaraman, M., "Modular Verification of Performance Correctness", *SAVCBS Workshop Proceedings*, 2001, 60-67.

[12] Krone, J., Ogden, W.F., "Abstract OO Big O," *SAVCBS Workshop Proceedings*, 2003, 80-84.

[13] Leavens, G. T., Baker, A. L., and Ruby, C., "JML: A Notation for Detailed Design," *Behavioral Specifications of Businesses and Systems*, H. Kilov and B. Rumpe and I. Simmonds, eds., Kluwer Academic Publishers, Boston, 1999.

[14] Leavens, G.T., Abrial, J., Batory, D., Butler, M., Coglio, A., Fisler, K., Hehner, E., Jones, C., Miller, D., Peyton-Jones, S., Sitaraman, M., Smith, D.R., and Stump, A.: Roadmap for Enhanced Languages and Methods to Aid Verification. Department of Computer Science, Iowa State University, TR #06-21. July 2006.

[15] Lim, S-S, Bae, Y. H., Jang, G. T., Rhee, B-D, Min, S. L., Park, C. Y., Shin, H., Park, K., Moon, S-M, and Kim, C. S., "An accurate worst case timing analysis for RISC processors," *IEEE Transactions on Software Engineering,* Vol. 21, No. 7, July 1995, 593 - 604.

[16] Meyer, B., *Object-Oriented Software Construction,* Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

[17] Meyer, B., "The Grand Challenge of Trusted Components," *Procs. 25th Int. Conference on Software Engineering*, Portland, OR, May 2003, 660-667.

[18] Musser, D.R., Derge, G.J., and Saini, A. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[19] Muller, P. and Poetzsch-Heffter, A., "Modular Specification and Verification Techniques for Object-Oriented Software Components," in *Foundations of Component-Based Systems*, eds. G. T. Leavens and M. Sitaraman, Cambridge University Press, 2000.

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

[21] Schmidt, H. and Zimmermann, W., "A Complexity Calculus for Object-Oriented Programs," *Journal of Object-Oriented Systems*, 1994, 117-147.

[22] Shaw, M., *A Formal System for Specifying and Verifying Program Performance*, Carnegie-Mellon University Technical Report CMU-CS-79-129, June 1979.

[23] Shaw, A. C., Reasoning About Time in Higher-Level Language Software, *IEEE Transactions on Software Engineering 15*, 1989, 875-889**.**

[24] Smith, C. U., *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[25] Sitaraman, M., Ogden, W.F., and Weide, B.W., "On the Practical Need for Abstraction Relations to Verify Abstract Data Type Representations," *IEEE Trans. Software Eng 23*, 3, Mar. 1997, 157-170.

[26] 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*, IEEE Computer Society, 2000.

[27] Unnikrishnan, L., Stoller, S. D., and Liu, Y. A., "Automatic Accurate Live Memory Analysis for Garbage-Collected Languages," *Procs. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2001.

[28] Weide, B. W., Ogden, W. F., and Sitaraman, M., "Expressiveness Issues in Compositional Performance Reasoning," *Procs. Sixth ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, Portland, OR, May 2001, 85 - 90