Profiles: A Compositional Mechanism for Performance Specification

Joan Krone, William F. Ogden, and Murali Sitaraman

Technical Report RSRG-04-03
Department of Computer Science
451 Edwards Hall
Clemson University
Clemson, SC 29634-0974 USA

June 2004

Profiles: A Compositional Mechanism for Performance Specification

J. Krone[1], W. F. Ogden[2], and M. Sitaraman[3]

Abstract

A system for engineering component-based software must include mechanisms for specifying abstractly both the functionality of components and their performance. Using the profile construct introduced here for performance specification, 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 a larger scale component using a composition of the profiles of its constituent (possibly as yet unimplemented) components. To ensure scalability, the profile construct facilitates abstraction in performance specifications as well as formation of profiles for generics from the profiles of their parameters.

Key Words: Abstraction, components, generics, memory space, time.

[1] Dept. Mathematics and Computer Science , Denison University, Greenville, OH 43023;
E-mail: krone@denison.edu.
[2] Dept. Computer and Info. Science, Ohio State University, Columbus, OH 43210;
E-mail: ogden@cis.ohio-state.edu.
[3] Dept. Computer Science, Clemson University, Clemson, SC 29634-0974;
E-mail: {murali}@cs.clemson.edu.

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 implicitly suppresses irrelevant others. The functional aspect is typically the most important, and so developing a framework for its specification has been the focus of much of software engineering research. However, a system for developing components is not complete until its specificational framework includes a mechanism for describing component performance.

Factoring out performance specifications seems to be a common practice in engineering 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. We introduce the *profile* as a specificational mechanism for recording such performance characteristics. Profiles have the virtue of allowing the designer of a component implementation to summarize its expected performance concisely, while at the same time, allowing a prospective client for the functionality of a component to use the profiles of the 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 exist independent of both. This makes it a notion that's independent of top-down or bottom-up methodology. A good profile will typically be one that is developed with simultaneous attention to the needs of the clients and considerations of what implementers could provide, regardless of whether the client or implementation code existed at the time. Serving as contracts, profiles help decouple component developers from clients, and facilitate independent software development. In the same way as abstract specifications of functional behavior serve as essential building blocks in development and analysis of component-based systems in a modular fashion, profiles facilitate modular construction while accounting for performance behavior. Using profiles of constituent components, it becomes possible to define the expected performance of a larger scale component.

Since a performance profile is expressed in the context of functional behavior, it is not surprising that performance specification and reasoning engender every complexity that arises in functionality specification and reasoning, because they depend on corresponding functionality aspects. In addition, since performance is more intricately tied to implementations than functional behavior, its specification poses several new challenges. One of them is maintaining the abstract nature of specification so that users can ignore intricate implementation details in their analysis. Another is the expression of the performance of a generic component whose parameters are not fixed at the time of specification. Our examples illustrate how the expressions can be kept abstract and generic, providing precise descriptions of what is required for the implementer to follow

and, at the same time, showing clients what they can expect regarding performance. We outline how performance information can be composed.

Section 2 introduces performance profiles. It includes the functionality specification of a parameterized concept and a performance profile for one class of implementations of that specification. Section 3 explains the need for supplementing mathematical models describing functional behavior of data abstractions to express performance estimates with a suitable degree of abstraction. It presents such a profile for an alternative class of implementations where objects hold invisible storage. Section 4 outlines how compositionality can be achieved using profiles. Section 5 contains a discussion of related work and our conclusions.

2. Component Specification

Except in the simplest situations, performance specifications can be expressed only in the context of corresponding functionality specifications, and therefore, we begin with an abstract specification of functionality. We present our examples in RESOLVE, a language that provides support for both specification and implementations, including multiple implementations of generic components [Sitaraman 94a, Sitaraman 00]. However, the results are general, and any behavioral specification language such as Larch, VDM, or Z summarized in [Wing 90] can be extended with profiles.

2.1 Functional Basis

To illustrate both abstract functionality specification issues and concrete implementation issues necessary for specifying performance and to introduce profiles for expressing performance, we choose a generic *Stack* data abstraction as an example, because it is a simple and familiar concept. However, the ideas are designed to generalize for expressing performance specifications of any (parameterized) components.

Figure 1 shows a formal, functionality specification (or a conceptual client view) of a generic bounded *Stack* component, parameterized by the type of items 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.) Given the support for generic components in modern languages, such as C++ and the most recent versions of Java and C#, the example motivates the need to address the complexity of developing parameterized, compositional performance profiles.

```
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| );

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

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

Figure 1: Specification of Stack_Template

The specification in Figure 1, termed a **concept**, uses mathematical String_Theory, a formalization of which is given in [Ogden 00b]. Using *Str* (the mathematical space named in String_Theory), the specification indicates that the value space of variables of type *Stack* is a subset of mathematical strings of entries. The term **Type_Family** is used (as opposed to type) to highlight the generic nature of the concept that exports a family of types depending on the particular *Entry* type and *Max_Depth* parameters supplied at the time of instantiation.

A string is similar to, but simpler than, a "sequence" because it does not explicitly include the notion of a position. The notations from String_Theory used in this specification are straight forward: 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*. Model-based specifications of other concepts use a combination of standard mathematical models such as integers, sets, functions, relations, and tuples, in addition to strings.

Using an **exemplar** stack variable S, the specification states that the value of every stack variable, which is a string, is constrained to have a maximum depth no greater than *Max_Depth*. Every stack variable is specified to be initialized automatically to be the empty string, as given in the **initialization ensures** clause. (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.

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, requires that the stack object S is not full. 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 post-conditions to be loose to allow maximum flexibility for implementers. *Pop* operation **replaces** the value in the parameter R with the entry in the top of the stack after removing it. These particular specifications for *Push* and *Pop* make it unnecessary for implementations to copy arbitrarily large entries.

Pre- and postconditions of the *Pop*, *Depth*, Rem_*Capacity* and *Clear* operations specify in string theoretic terms precisely what they will do. The **restores** mode in the specification of *Depth* 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 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 requirements and guarantees stated in the specification.

2.2 Adding a Performance Profile

Different kinds of implementations of generic concepts such as stacks provide different performance trade-offs. To illustrate performance ramifications, we consider two general implementation strategies that offer space-time trade-offs for implementing any data structure, not just stacks. The one considered in this subsection is more storage conscious than the one discussed in the next section which is faster. We begin the discussion with the profile of the former which is simpler.

Figure 2 contains selected parts of a performance profile, named *SSC*, 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 given in terms of these parameters.

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;
Defn 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} + (Max_Depth – 1)·Entry.**I_Disp**
+ Entry.**IM_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};
manip_disp SSC_{MPu};
…
end SSC;

Figure 2: A Performance Profile

A performance profile is used to document the behavior of a class of implementations in terms understandable to users of the concept without knowledge of concrete implementation details. A profile provides the following information. For each operation, there is a **duration** clause – a positive real number – 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 – that states the minimum additional space that is necessary to execute the operation above and beyond what is occupied by all objects currently in scope. Since space usage may increase and decrease during the execution of a procedure, this clause expresses the "high water mark" in terms of the parameters to the operations. In order to use this information to determine if there is enough space to execute the next call with a certain set of arguments, a caller needs to be able to determine the space occupied by all current objects. Therefore, profiles for implementations that provide types and therefore, creation of objects, additionally 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 with this clause.

The Object Displacement Clause

To make our discussions concrete, we consider implementations that allocate and initialize an array of entries of size Max_Depth when 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 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 to provide concrete details for a reader of this paper, a performance profile must be understandable to users based on the mathematical conceptualization of stacks as strings as given in Figure 1. Therefore, the displacement clause in the 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 a constant SSC_D and it represents the constant 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 given inside 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 used by the entries that have been pushed on a stack. To express this term, we have defined and used a definition *Cnts_Disp(S)* which counts the number of times an entry *E* occurs in the a string *S* (the abstract value of a stack) and multiplies it by the displacement for that entry E, denoted by *Entry.**Disp**(E)*. In the definition of *Cnts_Disp* in Figure 2, we have used *Occurs_Ct* – a mathematical definition that gives the number of times a given entry occurs in an arbitrary string – defined in *String_Theory*.

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 to have initial values under this profile). 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***.

We note that the same ideas discussed here suffice whether or not stacks are bounded a priori. For example, if the stack elements are allocated as and when needed instead of using an array, then the displacement will be less and it would not include the last term. Alternatively, if the convention of keeping unused array locations initialized is not followed, the third term will become more expensive.

Performance 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 to initialize 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 a 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 as and 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. Suppose 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 to be 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.

Performance Specification of Pop

To explain the expressions for Pop, we consider the following code that might be written in 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 the 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 elsewhere [Harms 91]: 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 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 corrupt the convention. At the end of the code, the local variable that now 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:

```
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));
```

The duration expression is straightforward. It includes the time to initialize a new *Entry* variable. Finalization depends on the Entry that is finalized, and therefore, the time to finalize is given in terms of the incoming value of parameter *R*. The definition for the constant term 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: ℝ^≥0 = Dur_Call(2) + 2·Array.Dur_:=: + 6·Record.Dur_. + Int_:= + Int_.;
```

The constant includes the time to call a procedure with 2 parameters, denoted by $\boldsymbol{Dur_{Call}(2)}$, array and record accesses, and Integer operations. This definition is relegated to the implementation because it is too much information to include in a profile for clients, and it is written in terms of implementation details which 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.

What 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[4], the ensures clause tells a user that E will be initialized after a call to *Push(E, S)*. While this information that appears in the performance profile cannot be used by a client program in establishing functional correctness, it can be used for reaching space/duration conclusions as illustrated in Section 4.

The duration and displacement expressions for *Push* and other *Stack* operations are constants, except *Clear*. To satisfy the space-conscious convention of keeping unused array entries initialized, the implementation of *Clear* needs to release the storage for all *Stack* entries at the time of the call and make sure that the corresponding array locations have initial entries.

3. Profiles with Supplementary Models

To express estimates of performance for some classes of implementations, it is necessary to augment the abstract functional model. The problem is quite general, and it arises, for example, when the concrete representation of an object includes hidden storage that is not visible in its abstract functionality specification. This situation is not atypical. Practical software components routinely trade off space for time, often using hidden data structures. To be able to provide abstract descriptions of performance for such components, additional specification machinery is necessary as explained in this subsection.

We now consider a specific example of a suitable performance profile for a class of *Stack* implementations for which speed is a higher priority than space usage. These implementations do *not* follow the space-conscious convention, i.e., internally-stored entries (in the array) that are not a part of the stack need not be initialized. This strategy leads to Stack implementations especially with a fast *Clear* operation, and hence, the corresponding profile is named *SFC* in Figure 3. To document the space occupied by a *Stack* object in this profile, it is necessary to talk about the hidden entries, and this is the motivation for supplementing the model of stacks as strings of Entries with a second string of (hidden or residual) Entries. Just as we were able to develop a satisfactory performance profile for the space-conscious implementations, subscribing to the view of stacks only as strings of entries, the supplemental model makes it possible to provide an abstract description of performance for the class of implementations discussed in this section.

[4] 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.

```
Profile SFC short_for Fast_Clear for Stack_Template;
    Defines  SFC_I, SFC_I1, SFC_F, SFC_Po, SFC_Pu, SFC_C, SFC_Dp, SFC_RC: ℝ^≥0;
    Defines  SFC_D, SFC_MI, SFC_MF, SFC_MPo, SFC_MPu, SFC_MC, SFC_MDp, SFC_MRC: ℕ;

  Type_Family Stack;
        Supplement SFC with Resid: Str(Entry);
            constraint |S.ipso| + |S.Resid| = Max_Depth;
        Defn Cnts_Disp( α: Str(Entry) ): ℕ = ( ∑_{E: Entry} Occurs_Ct(E, α)·Entry.Disp(E) );
        Displacement SFC_D + Cnts_Disp( S.Resid∘S.ipso );
        Defn Is_All_Init( α: Str(Entry) ): 𝔹 =
            ( ∀ E: Entry, if ⟨E⟩ Is_Substring α, then Entry.Is_Init(E) );
        Initialization;
            ensures  Is_All_Init( S.Resid );
            duration  SFC_I + (SFC_I1 + Entry.I_Dur)·Max_Depth;
            manip_disp  SFC_MI + (Max_Depth – 1)·Entry.I_Disp
                                                + Entry.IM_Disp;

    Oper Pop( replaces R: Entry; updates S: Stack );
        ensures S.Resid = ⟨#R⟩∘#S.Resid;
        duration  SFC_Po;
        manip_disp  SFC_MPo;

    Oper Push( alters E: Entry; updates S: Stack );
        ensures #S.Resid = ⟨E⟩∘S.Resid;
        duration  SFC_Pu;
        manip_disp SFC_MPu;

    Oper Clear( clears S: Stack );
        ensures  S.Resid = #S.Resid∘#S.ipso;
        duration  SFC_C;
        manip_disp  SFC_MC;
    …
end SFC;
```

Figure 3: A Profile with Supplementary Model

3.1 Specification of Supplementary Abstract Models

We begin the discussion considering the straightforward code of Pop in *Stack* implementations that do *not* maintain the space-conscious convention. Here, unlike the code for *Pop* given in the previous section, no (initialized) local variable is declared and swapped into the array to keep it "clean."

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

In the above implementation, when *Pop* is called, the top *Stack* entry replaces the value in R. At the same time, the (arbitrary) value that happens to be in parameter R is swapped and left in the array. If R happens to be non-trivial tree, for example, then it continues to be held in the array though it is not a part of the conceptual stack. After a series of calls to *Push* and *Pop*, it is possible that a stack is empty, though several residual objects are left over in its internal array. These residual entries correspond to a second string of entries, and this string is introduced in the *SFC* profile as a supplementary model of a Stack variable.

```
Supplement Resid: Str(Entry);
        constraint |S.ipso| + |S.Resid| = Max_Depth;
```

In this profile, the abstraction for a *Stack* variable is a pair of strings: a string of visible entries and a string of residual entries. Since the maximum depth of any Stack variable is fixed to be *Max_Depth* in the concept, the constraint specifies that the total number of entries in a *Stack* object, residual and others, add up to *Max_Depth*. In this expression, S.**ipso** (a Latin word that means "for itself") is a placeholder for the (string) view of *Stack* given in the specification of *Stack_Template* concept.

Specifying Changes to the Supplemental Model

When a profile introduces a supplementary model, it must also specify how each operation affects this supplement. Without it, there would be no way of knowing what the supplement contains. The initialization operation includes an additional guarantee that all the residual entries are initialized. A local definition, *Is_All_Init*, based on the predicate *Entry.**Is_Init**(E)*, is used for expressing this clause.

```
Initialization;
        ensures  Is_All_Init( S.Resid );
```

Similarly, the additional ensures clause for the Pop operation makes it explicit that the incoming Entry supplied to a call becomes a part of the residue.

```
Oper Pop( replaces R: Entry; updates S: Stack );
        ensures S.Resid = ⟨#R⟩∘#S.Resid;
```

The specification of the Clear operation in the profile makes it apparent that there is no difference in the storage space occupied by a Stack S before and after a call.

```
Oper Clear( clears S: Stack );
        ensures  S.Resid = #S.Resid∘#S.ipso;
```

The corresponding code in the implementation merely resets an index and is a constant-time operation. The code allows arbitrary entries to remain in the array (possibly wasting space) as a trade-off for speed.

```
Procedure Clear( updates S: Stack );
    S.Top := 0;
end Clear;
```

3.2 Performance Specification Using Supplements

Given the supplemental model, the profile in Figure 3 states how much storage space is occupied by a *Stack* object, and it is essentially the aggregate displacement of all the entries.

Displacement SFC_D + Cnts_Disp(S.Resid∘S.**ipso**);

It is important to note that the expression is given purely in terms of the information in the concept and the supplementary model in the profile without any reference to implementation details. In general, a profile may introduce arbitrarily complex supplements necessary to capture performance behavior. Although the supplements may reveal more information than the concept does, typically, it is possible to avoid introducing implementation details, keeping the performance specification at the conceptual level. In general, it may be necessary to introduce vastly different supplements to capture the performance characteristics of different implementations (or no supplement at all as illustrated in the case of the profile in Figure 2).

Based on the *Stack* initialization ensures clause in the concept and the profile, and the object displacement clause, we can deduce that a stack *S* has the displacement SFC_D + *Max_Depth*·*Entry*.***I_Disp***, immediately after creation. The duration and manipulation displacement clauses for initialization are identical to those given in the first profile in Figure 2, though they will be different for finalization. Other than initialization and finalization, all *Stack* operations, including *Pop* and *Clear*, are documented to take constant time and manipulation space in this profile.

4. Compositionality Using Profiles

The performance profiles are useful for component clients, enabling them to select from among a variety of implementations that provide interesting performance trade-offs for the same concept. They are also necessary for independent development and modular analysis of component-based systems in the same way as 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 reverse a stack is given

below. It is an **enhancement** or conceptual extension to the *Stack_Template* specification in Figure 1. 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;
```

An implementation of the *Flipping_Capability* specification is given below. It contains code for *Flip* using Stack operations, without relying on any particular implementation of the *Stack_Template* concept.

```
Realization Obvious_F_C_Realiz for Flipping_Capability;
    Procedure Flip( updates S: Stack );
        Var Next_Entry: Entry;
        Var S_Flipped: Stack;

        While ( Depth_of( S ) ≠ 0 ) do
                Pop( Next_Entry, S );
                Push( Next_Entry, S_Flipped );
        end;
        S :=: S_Flipped;
    end;
end Obvious_F_C_Realiz;
```

To analyze the functional behavior of the code, only the functionality specification of stacks given in Figure 1 is necessary. Modular reasoning about the functional correctness of the code can be done regardless of the implementation chosen for *Stack_Template*. We have presented a concrete implementation to motivate the resulting performance expressions in the discussion, but the emphasis in on developing abstract performance specifications that are understandable to clients of the *Flipping_Capability* without knowledge of its implementation details.

The same implementation of *Flip* has different performance behaviors, depending on whether a *Stack* implementation with *SSC* profile or *SFC* profile is used. Therefore, it is not possible to present a single profile for its specification. The example is typical, and it illustrates that multiple profiles for a single implementation can result if it is built reusing one or more other components. To specify performance in these more common situations in a component-based setting, we generalize the idea of profiles to *profile synthesizers* – constructs that make new profiles by using existing profiles for the concept under consideration. Figure 4 contains the skeleton of a profile synthesizer for *Flip*. It leads to either profile *SFF* or *STF* for *Flipping_Capability* depending on the *Stack* implementation profile. The motivation for (apparently conflicting) naming of the profiles will become clear in subsequent discussion.

Profile_Synthr STF **short_for** Typical_Flip **for** Flipping_Capability
for Stack_Template **with_profiles** SSC, SFC;
Profile SFF **short_for** Faster_Flip **matches** STF **with** SSC;
Profile SSF **short_for** Space_Saving_Flip **matches** STF **with** SFC;

Defines SFF_{F1}, SFF_{F2}, …: $\mathbb{R}^{\geq 0}$;
Defines SFF_{FMC1}, SFF_{FMC2}, …: $\mathbb{N}$;

Operation Flip(**updates** S: Stack);
duration_for SFF: …
duration_for SSF: …
…
end STF;

Figure 4: A Profile Synthesizer to Specify Profile Dependence

The synthesizer, *STF*, which can be realized with a single piece of code, provides performance specifications for both the *SSC* and *SFC* profiles of *Stack_Template*. This supports an economy of reuse of the same code for different profiles, those put together by this synthesizer.

4.1 Specification and Analysis of Flip for Stack Implementations with SSC Profile

Shown below is the performance behavior for *Flip*: The case *SFF*, where *Flip* is matched with *Stack_Template* implementations with *SSC* profile.

Operation Flip(**updates** S: Stack);
duration_for SFF: (SFF_{F1}+ Entry.**I_Dur** + Stack.**I_Dur** +
(Entry.**F_IV_Dur** + Stack.**F_IV_Dur** +
(SFF_{F2}+ Entry.**I_Dur** + Entry.**F_IV_Dur**)·|#S|;
manip_disp_for SFF: (SFF_{FMC1} + Entry.**I_Disp** + Stack.**I_Disp** +
Max (SFF_{FMC2}, Entry.**IM_Disp**, Entry.**F_IVM_Disp**));

The duration expression for *Flip*, in addition to a constant term SFF_{F1}, has three parts: duration for local variable initialization, local variable finalization, and loop execution. First we assume that a *Stack* component with profile *SSC* (Figure 2) 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. In this code, 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)* from the extended ensures clause for *Push* operation in the

profile *SSC* guarantees that after *Push* the parametric *Entry* is initialized. Therefore, the duration to finalize the Entry at the end of the code is *Entry.**F_IV_Dur*** – the time to finalize an entry with initial value.

The loop executes |#S| times. The time for each iteration includes a constant term because of calls to *Depth*, *Push*, and loop branching. It 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, *Next_Entry* that is supplied to Pop is the entry resulting from the previous to call to *Push*. Since the ensures clause of *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***. This analysis leads us to the above lemma about the duration specification for *Flip*, where SFF_{F1} and SFF_{F2} are two constants defined in terms of the constants given in the *SSC* profile.

Using a similar analysis based on the *SSC* profile, minimum space necessary to call *Flip* beyond what it is occupied by its parametric stack is its manipulation displacement expression given above. The procedure requires space for local variable creation: *Entry.**I_Disp*** + *Stack.**I_Disp***. In addition, it requires the maximum space that might be manipulated by any of the called operations, *Entry* initialization, *Entry* finalization, *Depth_of*, *Push*, and *Pop*. Using the expressions in the SFC profile, this analysis leads us to that maximum given in the expression. In general, *Entry* finalization can take different amounts of time and space depending in the entry that is finalized. Here, *Entry.**F_IVM_Disp*** denotes the manipulation displacement necessary to finalize an entry with an initial value, and this is sufficient, because all entries that are on the finalized stack have initial values. In developing the formula, we have used the information that the finalization manipulation displacement for a *Stack* variable (not shown in the profile in Figure 2) is dominated by *Entry* finalization manipulation displacement.

4.2 Specification and Analysis of Flip for Stack Implementations with SFC Profile

Next we discuss the *Flip* procedure relative to the *SFC* profile in Figure 3 that includes the supplemental residue entries. The resulting profile is named *SSF*, as a short hand for *Space_Saving Flip* because it ensures that the residue of the incoming stack *S* is cleared to be all initial entries as a side-effect of flipping. This becomes clear from the additional ensures clause for *Flip* given below.

Operation Flip(**updates** S: Stack);
ensures Is_All_Init(S.Resid);
duration (SSF_{F1}+ Entry.**I_Dur** + Stack.**I_Dur** +
Entry.**F_IV_Dur** +
Cnts_F_Dur(#S.resid) + (Max_Depth - |#S.**ipso**|)·Entry.**F_IV_Dur** +
SSF_{F2}·|#S.**ipso**|;
manip_disp (SSF_{FMC1} + Entry.**I_Disp** + Stack.**I_Disp** +
Max (SSF_{FMC2}, Entry.**IM_Disp**,
Entry.**F_IVM_Disp**, Max (Entry.**FM_Disp**(E)));
E: Entry, β: Str(Entry) ∋
⟨E⟩∘β Is_Sufix #S.resid

The duration for initialization in this case leads to the same expression and it is: *Entry.**I_Dur*** and *Stack.**I_Dur***. To compute the time for finalizing the local stack variable, we use the assertions on the supplemental model in the profile in addition to the assertions in the functionality specification. Following this reasoning, we see that the local stack *S_Flipped* that is finalized at the end is conceptually empty and therefore, its residue is of length *Max_Depth* as per the constraints in the profile. In addition, we can reason that *S_Flipped.resid* is the residue of the parametric stack *#S.resid* concatenated with a string of all empty entries. Therefore, the duration to finalize *S_Flipped* is the duration to finalize all the entries in *#S.resid* (given using the definition *Cnts_F_Dur* in the formula) plus the duration to finalize empty entries. From the assertions, it is also possible to analyze that the local *Next_Entry* has an initial value at the end of the code, leading to the expression *Entry.**F_IV_Dur*** in the formula. All the operations invoked in the loop take a constant time when using the *SFC* profile, and therefore, the loop duration depends only on the length of the conceptual value of the parameter *S*.

The central difference between the manipulation displacement expression in this profile for *Flip* and the one corresponding to the *SSC* profile arises from the fact that all residual entries in the parametric stack *S* need to be finalized. Since the finalization manipulation displacement time depends on the value of the entry, we have to do a maximum among the entries in the residue of the incoming stack.

It is also possible to generate the performance expressions for Flip operation mechanically, but our experience in mechanical generation of performance formulas without human intervention shows us that the resulting expressions are too general and are unlikely to be comprehensible. But it is both possible and important to develop tools to check the correctness of specified formulas against the implementations.

5. Related Work and Discussion

The importance of performance considerations for software engineering (e.g., [CCW00, Smi90]), in general, and for software components (e.g., [Szy98, Mey03]), in particular, has been widely acknowledged. Designers of languages and developers of component libraries have emphasized the role of alternative implementations to provide performance trade-offs [Boo87, Mey97, MDS01,Win90]. The importance of generic programming is

becoming increasingly clear [JLM00] as seen from improvements to Java and C#. Even more sophisticated approaches that permit development of systems to allow parameterization and choice of implementation have been proposed [BaG97, SFS97]. However, 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 [BDI04], 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 specify performance is a step in integrating performance considerations in software development.

Elsewhere we have explained that classical big-O notations [CLR90, Knu68] can be strengthened to provide performance estimates in terms of values of objects, not merely their sizes, in order to be suitable for components [SKO01]; other complexity calculus approaches have been proposed [ScZ94]. But big-O notations require metricization, and are often too restrictive for specifying performance precisely. 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 make the ideas discussed in this paper quite different from the work in the real-time community (e.g., [HaU01, Hoo91, JuT02, LyV95, Mah92]), where timing deadlines and concurrency are the focus.

Our work is complementary to the work in performance engineering based on actual measurement in that our profiles are conceptual. This observation suggests that an important future direction is to explore how measurement can be employed to "test" that the validity of component profiles for specific instances. One of the efforts along this direction is the inclusion of performance-related assertions into JML – Java Modeling Language [LBR99, LPC03] based on the notations discussed in this paper. Current JML tools can check if functionality-related assertions hold at run-time. Using run-time monitoring and measurement, we hope that performance assertions can be "tested" in a similar fashion. Based on the principles presented in this paper, we believe that a performance specification framework for other behavioral specification language and implementation language combinations can be developed, as long as the particulars of the language features are accounted for carefully in specification.

An important future direction is to extend previous work on formal static analysis of functional behavior to performance. Working within the context of a procedure, Shaw has proposed elements of a system for timing analysis at the source code level [Sha89]. Following Shaw's work, whereas Lim, et al., have focused on hardware aspects and executable code to provide tight analysis (e.g., [LBJ95]), Liu and Gomez have addressed various other aspects of source code analysis, including loops, parameters and recursion [LiG98, GoL02]. Hehner has built on the work of Mary Shaw [Sha79], to formalize time

and space analysis of a recursive procedure at the source code level [Heh99]. 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 [HoJ03, USL01].

Ultimately, for compositional performance analysis of components with performance profiles such as the ones outlined in this paper, current approaches for formal analysis need to be combined with advances in compositional verification of functional behavior in the presence of data abstractions (e.g, [EHO94, Lei95, LeW95, MuP00, SOW97, SAK00]), because assertions for functional correctness are necessary in establishing performance correctness. To automate the process, additional annotations for loops in the spirit of invariants for time and space and additional abstraction relations to supplementary models will be necessary. While we have motivated the need for supplementary models in performance specifications in this paper using a simple example, the more general nature of the problem can be seen from the discussions and examples in [Sit94] on alternative levels of precision, expressiveness of specifications in [WOS01], and amortized cost and shared realizations in [EHO94].

We have introduced a framework within which it is possible to specify both functionality and performance behaviors of software components at appropriate levels of abstraction, providing a vocabulary for stating time and space constraints. The new framework supports both generics and compositionality.

Acknowledgments

Several people have contributed important ideas to this work. We would especially like to thank past and present members of the *Reusable Software Research Groups* at Clemson University and Ohio State University. Our thanks are due to Gary Leavens and Bruce Weide for their comments on topics related to this paper. We gratefully acknowledge financial support from the National Science Foundation under grant CCR-0113181.

References

[BaG97] Batory, B., and Geraci, B. J., "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, 23(2), February 1997, 67-82.

[BDI04] 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.

[Boo87] Booch, G. *Software Components With Ada.* Benjamin/Cummings, Menlo Park, CA, 1987.

[BOS98] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P., "Making the Future Safe for the Past: Adding Genericity to the Java Programming Language," *Proc. OOPSLA 98,* ACM, 1998.

[CCW00] 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.

[CLR90] Cormen, T.H., Leiserson, C.E., Rivest, R.L., *Introduction to Algorithms*, The MIT Press, Cambridge, MA, 1990.

[EHO94] 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.

[GoL02] 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.

[HaU01] Hayes, I.J. and Utting, M., "A Sequential Real-Time Refinement Calculus," Acta Informatica 37, 2001, 385-448.

[HaW91] 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.

[Heh99] Hehner, E. C. R., "Formalization of Time and Space," *Formal Aspects of Computing*, Springer-Verlag, 1999, 6-18.

[HoJ03] 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.

[Hoo91] Hooman, J., *Specification and Compositional Verification of Real-Time Systems*, LNCS 558, Springer-Verlag, New York, 1991.

[JLM00] *Generic Programming*, eds. M. Jazayeri, R. G. K. Loos, and D. R. Musser, LNCS 1766, Springer, 2000.

[Jon98] Jones, R., Preface, *Proceedings of the International Symposium on Memory Management, ACM SIGPLAN Notices 34,* No. 3, March 1999, iv-v.

[JuT02] Juan, E. Y. T. and Tsai, J. J. P., *Compositional Verification of Concurrent and Real-Time Systems*, Kluwer, 2002.

[Knu68] Knuth, D. E., *The Art of Computer Programming*, Vol. 1, Addison-Wesley, 1968; Third Edition, 1997.

[LBR99] 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.

[LPC03] Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., and Kiniry, J., *JML Reference Manual (DRAFT)*, April 2003; available at http://www.cs.iastate.edu/~leavens/JML/documentation.shtml.

[LBJ95] 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.

[Lei95] Leino, K. R. M., *Toward Reliable Modular Programs*, Ph. D. Thesis, California Institute of Technology, 1995.

[LeW95] Leavens, G. T. and Weihl, W. E. "Specification and Verification of Object-Oriented Programs Using Supertype Abstraction," *Acta Informatica 32*, 8, Nov. 1995, 705-778.

[Liu98] 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.

[LyV95] Lynch, N. and Vaandrager, F. Forward and backward simulations, part II: timing-based systems. *Information and Computation 121*, 2, 1995, 214-233.

[Mah92] Mahony, B. P., *The Specification and Refinement of Timed Processes*, PhD thesis, Department of Computer Science, The University of Queensland, 1992.

[Mey97] Meyer, B., *Object-Oriented Software Construction*, 2nd Edition, Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

[Mey03] Meyer, B., "The Grand Challenge of Trusted Components," *Procs. 25th Int. Conference on Software Engineering*, Portland, OR, May 2003, 660-667.

[MDS01] Musser, D.R., Derge, G.J., and Saini, A. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[MuP00] 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.

[Ogd00] Ogden, W.F., *The Proper Conceptualization of Data Structures*, Dept. Computer and Information Science, Ohio State University, 2000.

[ScZ94] Schmidt, H. and Zimmermann, W., "A Complexity Calculus for Object-Oriented Programs," *Journal of Object-Oriented Systems*, 1994, 117-147.

[Sha79] Shaw, M., *A Formal System for Specifying and Verifying Program Performance*, Carnegie-Mellon University Technical Report CMU-CS-79-129, June 1979.

[Sha89] Shaw, A. C., Reasoning About Time in Higher-Level Language Software, *IEEE Transactions on Software Engineering 15*, 1989, 875-889**.**

[SKK01] Sitaraman, M., Krone, J., Kulczycki, G., Ogden, W. F., and Reddy, A. L. N., "Performance Specification of Software Components," *ACM SIGSOFT Symposium on Software Reuse*, May 2001.

[Smi90] Smith, C. U., *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[Sit92] Sitaraman, M. "Performance-Parametrized Reusable Software Components," *International Journal of Software Engineering and Knowledge Engineering 2*, No. 4, December 1992, 567-587.

[SiW94] Sitaraman, M., and Weide, B.W., eds. Special Section: Component-based software using RESOLVE. *ACM Software Eng. Notes 19,4* (1994), 21-67.

[Sit94] 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.

[SOW97] 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.

[SAK00] 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.

[SFS97] Sreerama, S., Fleming, D., and Sitaraman, M., "Graceful Object-Based Performance Evolution," *Software - Practice and Experience*, Vol. 27, No. 1, January 1997, 111-122.

[Szy98] Szyperski, *C., Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[USL01] Unnikrishnan, L., Stoller, S. D., and Liu, Y. A., "Automatic Accurate Live Memory Analysis for Garbage-Collected Languages," *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2001.

[Win90] Wing, J. M., "A Specifier's Introduction to Formal Methods," *IEEE Computer 29*, 9, Sep. 1990, 8-24.

[WOS03] 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.