Modular Verification of Performance Correctness

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

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

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

Abstract
Component-based software engineering is concerned with predictability in both functional and performance behavior, though most formal techniques have typically focused their attention on the former. The objective of this paper is to present specification-based proof rules compositional or modular verification of performance in addition to functionality, addressing both time and space constraints. The modularity of the system makes it possible to verify performance correctness of a module or procedure locally, relative to the procedure itself. The proposed rules can be automated and are intended to serve as part of a system of rules that accommodate a language sufficiently powerful to support component-based, object-oriented software.

Categories and Subject Descriptors
D.2.13 [Software Engineering]: formal specification and verification of software performance.

General Terms
Verification, assertive language, formal specifications.

Keywords
Proof rule, performance, time and space.

1. INTRODUCTION

Predictability is a fundamental goal of all engineering, including software engineering. To show that a program predictably provides specified functional behavior, a variety of ways to apply a system of proof rules to a program for proving functional correctness have been studied since Hoare's work. More recent efforts address the special challenge of modular reasoning for object oriented, component based software [1, 5, 8, 9, 12]. These systems depend on programmer-supplied assertions that serve as formal specifications for the functional behavior of the software. While correct functional behavior is critical to any software system, in order to achieve full predictability, we must ultimately address the issue of performance as well.

A program that carries out the right job, but takes longer than available time to complete is of limited value, especially in modern embedded systems. Similarly, a program that is functionally correct, but that requires more space than the system can provide is not useful either. Cheng, Clemens, and Woodside note the importance of the performance problem in their guest editorial on *Software and Performance* [21]:

> "Performance is a problem in many software development projects and anecdotal evidence suggests that it is one of the principal reasons behind cases where projects fail totally. There is a disconnect between techniques being developed for software analysis and design and the techniques that are available for performance analysis."

Measurement during execution (e.g., using run-time monitoring) is a common approach for analyzing performance of large-scale systems [21]. The objective of this paper is to present static analysis (and hence, a priori prediction) as an alternative to measurement. In particular, the focus is on *modular* or *compositional performance reasoning*: Reasoning about the (functionality and performance) behavior of a system using the (functionality and performance) *specifications* of the components of the system, without a need to examine or otherwise analyze the implementations of those components [17].

Compositionality is essential for all analysis, including time and space analysis, to scale up. To facilitate compositional performance reasoning, we have introduced notations for performance specifications elsewhere [18]. Given functionality and performance specifications (and other internal assertions such as invariants), the rest of this paper describes a proof system for modular verification. Section II sets up the framework to facilitate automated application of rules, using a simple example rule. Section III contains proof rules for verification of procedure bodies and procedure calls, involving possibly generic objects with abstract models as parameters. Section IV contains an example to illustrate a variety of issues involved in formal verification. Section V has a discussion of related work and summary.

2. ELEMENTS OF THE PROOF SYSTEM

Though the underlying principles presented in this paper are language-independent and are applicable to any assertive language that includes syntactic slots for specifications and internal assertions, to make the ideas concrete we use the RESOLVE notation [15, 16]. RESOLVE is intended for predictable component-based software engineering and it includes notations for writing specifications of generic components that permit multiple realizations (implementations) of those components. It also includes notations for specifying time and space behaviors of an implementation. The implementations include programmer-supplied representation

invariants, loop invariants, progress metrics, and other assertions depending on the structure.

The proof rules have been designed so that an automated clause generator can start at the end of a given assertive program and back over the code replacing the executable language constructs with assertions about the mathematical domain over which the program has been written. The clause generator produces a clause that is equivalent to the correctness of the given program. The clause can then be evaluated manually, automatically by a theorem prover, or by a combination to determine whether the clause is provable in the appropriate mathematical domain, and thereby whether the program is correct (with respect to its specification). To illustrate the ideas, we begin with a simple example. First we consider functional behavior and then address performance for the following piece of assertive code:

Assume x = 3;
x := x + 1;
Confirm x = 4;

Exactly how such an assertive code comes into place, given a specification and an implementation, is explained in Section III. In this code segment, the programmer has supplied a pre-condition indicated by the **Assume** keyword and a post-condition following the keyword **Confirm** with some (assertive) code in between. To prove the correctness of this segment, consider the following automatable proof rule for expression assignment:

C \ Code; **Evaluate** (exp); **Confirm** Outcome_Exp[x ⇝ **M_Exp**(exp)]

C \ Code; x := exp; **Confirm** Outcome_Exp;

In this rule, *C* on the left side of both the hypothesis and the conclusion stands for *Context* and it denotes the collection of whatever information is needed about the code in order to reason about its correctness. For example, the types of variables and the mathematical theories on which those types are based would be in the context.

In our example, the `Outcome_Exp` is "x = 4." The `Code` preceding the assignment is the assertion "**Assume** x = 3." In the assertive clauses, the 3 and 4 are the mathematical integers, while the assignment statement is performing an increment on a computer representation of an integer. (The use of mathematical integers in specifying computational Integer operations is documented in *Integer_Template* that specifies Integer objects and operations, and it is assumed to be in the context.)

Applying the proof rule on the example leads to the following assertive code:

Assume x = 3; **Evaluate**(x + 1); **Confirm** x + 1 = 4.

This is the result of substituting the expression "x + 1" for x, the meaning of [x ⇝ **M_Exp**(exp)]. **M_Exp** denotes putting in the mathematical expression that corresponds to the programming expression, thus keeping our assertions over mathematical entities, rather than programming ones. There is a rule for **Evaluate** that causes the expression to be evaluated by the verifier. Similarly, the verifier would simply continue backing through the rest of the code, applying appropriate proof rules, eliminating one more constructs in each step.

Now we augment the above rule to prove functional correctness, with performance-related assertions. Suppose we need to prove the correctness of the following assertive code:

Assume x = 3 ^ **Cum_Dur** = 0 ^ **Prior_Max_Aug** = 0 ^ **Cur_Aug** = 0;
x = x + 1;
Confirm x = 4 ^ **Cum_Dur** + 0.0 = $D_{:=}$ + D_{Int_+} ^
Max(**Prior_Max_Aug**, **Cur_Aug** + 0) ≤ $S_{:=}$[1];

Here, $D_{:=}$ denotes the duration for expression assignment[2] (excluding the time to evaluate the expression itself). $S_{:=}$ denotes storage space requirement for expression assignment (excluding the storage space needed to evaluate the expression itself and the storage for variable declaration of x which is outside the above code). The units for time and space are assumed to be consistent, though we make no assumptions about the units themselves. The rest of the terms (whose need may not become fully clear until after the discussion of procedures in Section III) are explained in the context of the following rule for expression assignment:

C \ Code; **Evaluate**(exp); **Confirm** (Outcome_Exp ∧
Cum_Dur + $D_{:=}$ + Sqnt_Dur_Exp ≤ Dur_Bd_Exp∧
Max(**Prior_Max_Aug**, **Cur_Aug** + $S_{:=}$ +
Fut_Sup_Disp_Exp) ≤ Aug_Bd_Exp))
[x⇝**M_Exp**(exp)];

C \ Code; x := exp; **Confirm** Outcome_Exp ∧ **Cum_Dur** + Sqnt_Dur_Exp ≤ Dur_Bd_Exp ∧
Max(**Prior_Max_Aug**, **Cur_Aug** + Fut_Sup_Disp_Exp) ≤ Aug_Bd_Exp;

The new rule includes everything needed for functional correctness, and also includes new clauses about time and space performance. In spite of past attempts in the literature, it is just not possible to develop rules for performance correctness independently of functional correctness, because in general, performance depends on values of variables (which come from analyzing functional behavior) [17, 18]. In the example and in the rule, terms in bold print are keywords and the terms ending with "_Exp" represent expressions to be supplied by the programmer and kept up to date by the verifier.

---

[1] We have added terms "+ 0.0" and "+ 0" in the expressions here so that it is easy to match the syntactic structure of the rule given next.

[2] In RESOLVE, the right hand side of an assignment statement is restricted to be an expression. In particular, x := y is not allowed on variables of arbitrary types. For copying y to x, the assignment statement needs to be x := Replica(y). This interpretation is implicit for (easily) replicable objects such as Integers for programming convenience. This is what justifies the time analysis in the present rule. To move the value of y to x efficiently on all objects large and small, and without introducing aliasing, RESOLVE supports swapping (denoted by ":=:") as the built-in data movement operation on all objects [3].

First we consider timing. The keyword **Cum_Dur** suggests cumulative duration. At the beginning of a program the cumulative duration would be zero. As the program executes, the duration increases as each construct requires some amount of time to complete. The programmer supplies an over all duration bound expression, noted by `Dur_Bd_Exp`. This is some expression over variables of the program that indicates an amount of time acceptable for the completion of the program. As the verifier automatically steps backward through the code, that expression gets updated with proper variable substitutions as the proof rules indicate.

For example, in the above rule, when the verifier steps backward over an assignment, the variable, "x," receiving the assignment is replaced by the mathematical form of the given expression, "exp," in all of the expressions included within the parentheses.

`Sqnt_Dur_Exp` stands for the subsequent duration expression, an expression for how much time the program will take starting at this point. This expression is updated also automatically by the verifier, along with other expressions in the rule.

The duration (timing) for a program is clearly an accumulative value, i.e., each new construct simply adds additional duration to what was already present. On the other hand, storage space is not a simple additive quantity. As a program executes, the declaration of new variables will cause sudden, possibly sharp, increases in amount of space needed by the program. At the end of any given block, depending on memory management, storage space for variables local to the block, may be returned to some common storage facility, causing a possibly sharp decrease in space.

The right operation for duration is addition and for storage it turns out to be taking the maximum over any given block. It is reasonable to assume that for any given program, there will be a certain amount of space needed for getting the program started. This will include the program code itself, since the code will reside in memory. Assuming real, rather than virtual memory, the code will take up a fixed amount of space throughout the execution. With this in mind, we think of some fixed amount of space for any given program that remains in use throughout the execution. Our rules are written to deal with the space that augments the fixed storage and increases and decreases as the program executes. **Prior_Max_Aug** stands for "prior maximum augmentation" of space. At the beginning of any program, the prior maximum will be zero, since only the fixed storage is in use. As the program executes, over each block, a maximum of storage for that block is taken to be the **Prior_Max_Aug**. At any point in the program, there will be a storage amount over the fixed storage. We call that the current augmentation of space, **Cur_Aug**. Of course, there will be some overall storage bound to represent what is acceptable. We call that the augmentation bound expression, Aug_Bd_Exp. Finally, just as there was an expression to represent how much additional time would be needed, there is an expression for how much storage (displacement) will be needed in the future, the future supplementary displacement expression, `Fut_Sup_Disp_Exp`.

3. PROCEDURES

We examine a more complicated procedure construct in this section, having introduced basic terminology using the expression assignment proof rule. We present a rule for procedure declarations and one for procedure calls. These rules apply not only to ordinary code when all variables and types are previously defined, but to generic code as well, i.e., code written for variables that have not yet been tied to a particular type or value. This capability to handle generic code is critical for reusable, object-based components.

3.1 Procedure Declaration Rule

Associated with every procedure is a heading that includes the name, the parameter list, and assertions that describe both functional and performance behavior:

P_Heading:

Operation P(**updates** x: T);
requires P_Usg_Exp<x\>;
ensures P_Rslt_Exp<x, #x\>;
duration Dur_Exp<x, #x\>;
manip_disp M_D_Exp<x, #x\>;

This heading is a formal specification for procedure P. We use separate keywords **Operation** to denote the specification and **Procedure** to denote executable code that purports to implement an operation. We have included only one parameter on the argument list, but of course, if there were more, they would be treated according to whatever parameter mode were to be indicated. The **updates** mode means that the variable is to be updated, i.e., possibly changed during execution.

In the heading, the type `T` may be a type already pinned down in the program elsewhere, or it might represent a generic type that remains abstract at this point. The **requires** and **ensures** clauses are pre and post conditions respectively for the behavior of the operation, and the angle brackets hold arguments on which the clauses might be dependent. Due to page constraints, the rule does not include other potential dependencies such as on global variables.

Details of performance specification are given in [18]. **Duration** is the keyword for timing. `Dur_Exp` is a programmer-supplied expression that describes how much time the procedure may take. That expression may be given in terms of other procedures that `P` calls and it may be phrased in terms of the variables that the operation is designed to affect. We may need to refer both to the incoming value of *x* and to the resulting value of `x` in these clauses. We distinguish them by using `#x` for the value of `x` at the beginning of the procedure and `x` as the updated value when the procedure has completed. The last part of the Operation heading involves storage specification. Here, **manip_disp** (termed **trans_disp** in [18]) suggests manipulation displacement, i.e., how much space the procedure may manipulate as it executes.

Given the operation heading, we next consider a rule for a procedure declaration to implement an operation.

C ∪ {P_Heading} \ **Assume** P_Usg_Exp ∧
Cur_Dur = 0.0 ∧
Prior_Max_Aug = **Cur_Aug** = **Disp**(x);
P_Body;
Confirm P_Rslt_Exp ∧ **Cur_Dur** + 0.0 ≤ Dur_Exp ∧
Max(**Prior_Max_Aug**, **Cur_Aug** + 0) ≤ M_D_Exp;
C ∪ {P_Heading} \ Code; **Confirm** Outcome_Exp;

C \ P_Heading; **Procedure** P_Body; **end** P;
Code; **Confirm** Outcome_Exp;

As in the assignment rule, `C` stands for the context in which the procedure occurs. Note that `P_Heading`, the specification of Operation P, is added to the context making it possible for reasoning about the procedure to take place. The conclusion line of the rule allows the procedure declaration to be made and followed by some code and a clause to confirm after the code.

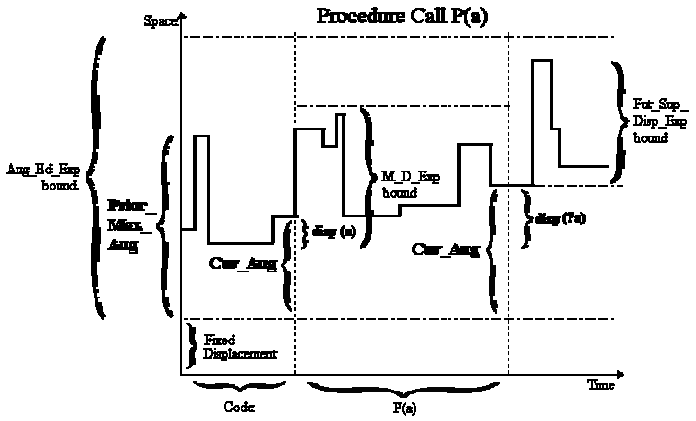
The hypotheses of the rule indicate that the procedure is to be examined abstractly, proving that no matter what value for the parameter is passed in, the result will satisfy both the functional and performance requirements.

The first hypothesis checks functional behavior by showing that if the **requires** clause is met, then the **ensures** clause is satisfied upon completion of the procedure body. For timing, we set the **Cum_Dur** to 0 thereby localizing the proof to just this procedure, avoiding the pitfall of having to consider the entire program when proving correctness for just this procedure. After the procedure body, we confirm that the **Cum_Dur** remains below Dur_Exp, the bound expression given in the specifications. It is assumed that the **Cum_Dur** acts like an auxiliary variable updated automatically at each step.

Finally, we address the storage requirements. Before the procedure body, we set the **Prior_Max_Aug** and the **Cur_Aug** both to be the amount of space required by the parameter, `x`. (Alternatively, the displacement of parameters at the beginning could be subtracted at the end.) This is necessary to retain the local nature of the proof process. The only concern that the procedure rule has about space is what the procedure uses above what has already been used in the past and what might be used in the future. After the body, the rule checks that the max over the stated values is within the specified bound.

3.2 Procedure Call Rule

A picture serves to motivate space-related assertions in the procedure call rule. The timing aspects of the rule are more straightforward and they are not shown in this picture.



Along the lower part of the picture the "fixed displacement" represents some amount of storage necessary for the program to run, an amount that does not vary throughout execution. The code itself is included in this fixed storage. Above the fixed storage the execution of the code requires a fluctuating amount of space, increasing when storage for new variables is allocated and decreasing when it is released.

The auxiliary variable, **Cur_Aug**, represents at any point what the current amount of storage is over and above the fixed storage. Note that the same variable appears twice on the picture, once at the place where a call to procedure `P` is made and again at the point of completion of `P`. **Cur_Aug** has a value at every point in the program and is continually updated. Similarly, as the execution procedes, **Prior_Max_Aug** keeps track of the maximum storage used during any interval. In the picture at the point where the call P(a) is made, **Cur_Aug** is shown, as is **Prior_Max_Aug**. Of course, as the code execution progresses, the value for **Prior_Max_Aug** is updated whenever a new peak in storage use occurs.

Within the procedure body, some local variables may be declared. This augmented displacement is denoted in the figure by a spike in the line representing space allocation for the procedure code. The specifications of the procedure include `M_D_Exp`, an expression that limits the supplementary storage a procedure may use. The procedure must stay within that limit in order to be considered correct in terms of performance. As the picture shows, the `M_D_Exp` is an expression about only local variables and whatever parameters are passed in. These are the only variables under the control of the procedure and they are the only ones the procedure should need to consider for specification and verification purposes.

Disp is an operator that extracts the amount of storage for a given variable. This operator gets its value in the displacement clause given in an implementation of an object-oriented concept, and it is usually parameterized by the object's value [18]. At the

point where the call `P(a)` is made the picture shows **Disp**(a), to denote that `a`'s space allotment is part of the current augmentation displacement. Upon completion of the procedure call, the new value of `a`, shown as `?a` may be different and may require a different amount of space from what its value needed at the time of the call. **Disp**(?a) is part of the current augmentation at the point of completion. `Fut_Max_Sup_Exp`, as noted before, describes a bound on the storage used by the remaining code, i.e., code following the current statement under consideration.

Given his explanation, the procedure call rule follows:

C ∪ {P_Heading} \ Code; **Confirm** P_Usg_Exp[x⇝a] ∧
∀ ?a: **M_Exp**(T), **if** P_Rslt_Exp[#x⇝a, x⇝?a] **then**
Outcome_Exp[a⇝?a] ∧
Cum_Dur + Dur_Exp[#x⇝a, x⇝?a] +
Sqnt_Dur_Exp[a⇝?a] ≤ Dur_Bd_Exp[a⇝?a] ∧
Max(**Prior_Max_Aug**, **Cur_Aug** ,
Max(M_D_Exp[#x⇝a, x⇝?a],
Disp(?a) + Fut_Sup_Disp_Exp[a⇝?a]) – **Disp**(a))
≤ Aug _Bd_Exp[a⇝?a];

C ∪ {P_Heading} \ Code; P(a); **Confirm** Outcome_Exp ∧
Cum_Dur + Sqnt_Dur_Exp ≤ Dur_Bd_Exp ∧
Max(**Prior_Max_Aug**, **Cur_Aug** + Fut_Sup_Disp_Exp) ≤
Aug_Bd_Exp;

The heading for P is placed in the context, making available the specifications needed to carry out any proof. In the conclusion line, a call to `P` with parameter `a` is made at the point in the program following `Code`.

In modular reasoning, verification of this code that calls an operation P is based only on the specification of P. The functional behavior is addressed in the top line of the hypothesis part of the rule. To facilitate modular verification, at the point in the code where the call to `P` is made with parameter `a,` it is necessary to check that the **requires** clause, `P_Usg_Exp` with `a` replacing `x` holds. The second hypothesis, also about functional behavior, checks to see that if the procedure successfully completes, i.e., the **ensures** clause is met with the appropriate substitution of variables, then the assertion `Outcome_Exp` holds, again with the appropriate substitution of variables. These substitutions make it possible for the rules to talk about two distinct times, one at the point where a call to the procedure is made and one at the point of completion. The substitution of what variables need to appear at what points in the proof process avoids the need ever to introduce more than two points in the time line, thereby simplifying the process.

It is important to note here that the specification of Operation P may be relational, i.e., alternative outputs may result for the same input. Regardless of what value results for parameters after a call to P, the calling code must satisfy its obligations. This is the reason for the universal quantification of variable `?a` in the rule.

The next hypothesis in the rule is about timing, and it checks, after variable substitution, that any result from the procedure will lead to satisfaction of specified time bounds for the client program. It is not surprising that any reasoning about time or space must be made in terms of the variables being manipulated, since their size and representation affect both.

Finally, the displacement hypothesis considers the maximum over several values. To understand this hypothesis, the picture helps by illustrating the prior maximum augmentation, current augmentation both at the point of the call and at the point of the return. The picture also shows the displacement for actual parameter `a` at the beginning of the procedure call and the displacement of `?a` at the end.

The displacement hypothesis involves a nested max situation. We consider the inner max first. Here we are taking the maximum over two items. The first is the expression from the procedure heading that identifies how much storage the procedure will need in terms of the local variables and the parameters. The second is the sum of the amount of space required by the final value of the updated parameter referred to as `?a` and the amount of space for the rest of the program represented by `Fut_Sup_Disp_Exp`. From the second quantity we subtract the displacement of `a`, since it was accounted for in the current augmentation. Finally, we take the max over the two items and show that it remains within the overall bound.

The technique used in parameter passing naturally affects the performance behavior of a procedure call. In the rule, we have assumed a constant-time parameter passing method, such as swapping [3]. An additional degree of complication is introduced when an argument is repeated as a procedure call, because extra variables may be created to handle the situation. The present rule does not address this complexity.

4. AN EXAMPLE

In this section, we present a more comprehensive example of a generic code segment, including appropriate expressions for describing time and space. In our example, we reproduce Stack_Template concept from [18], where a detailed explanation of the notation may be found:

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

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

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

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

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

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

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

This specification is for a generic family of stacks whose entries are left to be supplied by clients and whose maximum depth is a parameter. It exports a family of stack types along with the typical operations on stacks. Any given stack type is modeled as a collection of strings over the given type *Entry* whose length is bounded by the *Max_Depth* parameter.

In order to promote both component reuse and the idea of multiple implementations for any given concept, our design guidelines include the recommendation that concepts should provide whatever operations are necessary to support whatever type is being exported and operations that allow a user to check whether or not a given operation should be called. In the stack example both *Push* and *Pop* must be present because those are the operations that define stack behavior. The *Depth_of* and *Rem_Capacity* enable a client to find out whether or not it is alright to Push or to Pop. These are called primary operations.

Our guidelines suggest that secondary operations, ones that can be carried out -- efficiently -- using the primary ones, should be in an enhancement. An enhancement is a component that is written for a specific concept. It can use any of the exported types and operations provided in that concept. For example, we might write an enhancement to reverse a stack. In it would be an operation whose specifications indicate that whatever stack is passed into the procedure is supposed to be reversed. Given below is the functionality specification of such an enhancement:

```
Enhancement Flipping_Capability for Stack_Template;
    Operation Flip(updates S: Stack);
        ensures S = #S^Rev;
end Flipping_Capability;
```

The advantage of writing this capability as an enhancement is that it is reusable, i.e., it will work for all Stack_Template realizations. For an example of a Stack_Template realization, a reader is referred to [18].

In our implementation, given below, we have included both the code (it is purely generic since any realization of the given stack concept may be used for the underlying stack type) and the performance specifications that deal with time and space.

```
Realization Obvious_F_C_Realiz for
Stack_Template.Flipping_Capability;
  Duration Situation Normal: ∃ C_Pu, C_Po, C_IE, C_EI, C_SIS: ℝ^>0 ∋
        C_Pu = LUB(Dur_Push[Entry×Stack]) and
        C_Po = LUB(Dur_Pop[Entry×Stack]) and
        C_IE = LUB(Dur_Is_Empty[Stack]) and
        C_EI = Dur_Entry.Initialization and
        C_SIS + Max_Depth∗C_EI = Dur_Stack.Initialization;
    Defn const C_1: ℝ^>0 = (C_IE + C_Po + C_Pu);
    Defn const C_2: ℝ^>0 = (Dur_Call(1) + C_EI + C_SIS + C_IE + C_:=:);
    Defn const Cnts_Disp( S: Str(Entry) ): ℕ =
    ( ∑ Occurs_Ct(E,S) ∗ Disp(E) );
                E:Entry
    Displacement Situation Normal: ∃ D_SD, D_EID: ℕ ∋
      D_EID = Disp_Entry.Init_Val and
       ∀ S: Stack, Disp(S) = D_SD +
            D_EID∗(Max_Depth – |S|) + Cnts_Disp( S ) and
       ∀ E: Entry, Disp(E) ≥ D_EID and
       Is_Nominal(Mnp_Disp_Pop(E, S)) and
       Is_Nominal(Mnp_Disp_Push(E, S)) and
       Is_Nominal(Mnp_Disp_Is_Empty(S));

    Procedure Flip( upd S: Stack );
        duration Normal: C_1∗|#S| + Max_Depth∗C_EI + C_2;
         manip_disp Normal: 2∗D_SD + D_EID∗(2∗Max_Depth +
                                1 – |@S|) + Cnts_Disp( @S );
        Var Next_Entry: Entry;
        Var S_Flipped: Stack;
        While ¬ Is_Empty( S )
          updating S, S_Flipped, Next_Entry;
          maintaining #S = S_Flipped^Rev∘S and
          Entry.Is_Initial(Next_Entry);
          decreasing |S|;
          elapsed_time Normal: C_1∗|S_Flipped|;
          max_manip_space 2∗D_SD + D_EID∗(2∗Max_Depth
                            + 1 – |#S|) + Cnts_Disp( #S );
        do
                Pop( Next_Entry, S );
                Push( Next_Entry, S_Flipped );
        end;
        S :=: S_Flipped;
    end Flip;
end Obvious_F_C_Realiz;
```

In writing performance specifications, there is a trade-off between generality and simplicity. Given that the space/time usage of a call to every operation could depend on the input and outputs values of its parameters at the time of the call, a general version of performance specification can be quite complex. But we can simplify the situation, if we make some reasonable

assumptions about the performance of reusable operations. While the performance specification language should be sufficiently expressive to handle all possibilities, in this paper, we present simplified performance expressions making a few assumptions. When the assumptions do not hold, the performance specifications do not apply.

There may a variety of ways in which time and space are handled, such as the straightforward allocation of space upon declaration and immediate return upon completion of a block as one method, and amortization as another. Here we use the term **Duration Situation** followed by Normal to indicate the former. A specification may also give performance behavior for more than one situation.

We provide constants that represent durations for each of the procedures that might be called, taking least upper bound when those durations might vary according to contents. For example, $\mathbf{Dur}_{\text{Push}}$ stands for the amount of time taken by a Push operation. Since that might vary depending on the particular value being pushed, the least upper bound is used to address that fact.

The way this approach allows the use of generic code is to have specifications that can be given in terms of the procedures they call. We think of initialization as a special procedure, one for each type, that is called when a variable is declared. For example, $\mathbf{Dur}_{\text{Stack.Initialization}}$ means the duration associated with the initialization of a stack. We do not know nor do we need to know what particular kind of stack will be used here, rather our specifications are completely generic, allowing the specific values to be filled in once a particular stack type has been designated.

All of the constants at the beginning of the realization are presented as convenience definitions so that the expressions written in the **duration** and **manip_disp** clauses will be shorter to read.

Just as we have identified what **duration** constants are needed for specifying the duration of the reversing procedure, we also set up definitions to make the storage (**manip_disp**) expression shorter to read. We can now see how the duration and manipulation displacement expressions associated with each procedure can be used when scaling up and using those procedures in a larger program.

In verifying the correctness of the procedure, for the loop statement, the programmer supplies the following information:

- An **updating** clause that lists variables that might be modified in the loop, allowing the verifier to assume that values of other variables in scope are invariant, i.e., not modified;
- A **maintaining** clause that postulates an invariant for the loop;
- A **decreasing** clause that serves as a progress metric to be used in showing that the loop terminates;
- An **elapsed time** clause for each situation assumption in the duration specification to denote how much time has elapsed since the beginning of the loop; and
- A **max_manip_space** clause that denotes the maximum space manipulated since the beginning of the loop in any iteration.

The proof rule for while loop (not given here) checks that each of the programmer-supplied clauses is valid and then employs them in the proof.

In this short version of the paper, we have omitted discussion of several important issues, including proof rules for loop statements as well as other constructs. We have also not explained how the system can accommodate dynamic and/or global memory management, though the framework allows for those complications. Finally, the non-trivial aspects of a framework within which to discuss the soundness and completeness of the proof system need to be presented.

5. RELATED WORK AND SUMMARY

The importance of performance considerations in component-based software engineering is well documented [7, 19, 20, 21]. Designers of languages and developers of object-based component libraries have considered alternative implementations providing performance trade-offs, including parameterization for performance [2]. While these and other advances in object-based computing continue to change the nature of programming languages, formal techniques for static performance analysis have restricted their attention to real-time and concurrency aspects [6, 10, 11, 20].

Hehner and Reddy are among the first to consider formalization of space (including dynamic allocation) [4, 13]. Reddy's work is essentially a precursor to the contents of this paper, and its focus is on performance specification. The proof system for time and (maximum) space analysis outlined in [4] is similar to the elements of our proof system given in section 2 of this paper. Both systems are intended for automation. In verification of recursive procedures and loops, for automation, we expect time remaining and maximum manipulated space clauses to be supplied by a programmer, though the need for the clauses is not made apparent in the examples in Hehner's paper. Our rules for these constructs are, therefore, different. Other differences include performance specification of generic data abstractions and specification-based modular performance reasoning. This becomes clear, for example, by observing the role of the displacement functions in the procedure call rule in Section 3.

This paper complements our earlier paper on performance specification in explaining how performance can be analyzed formally and in a modular fashion. To have an analytical method for performance prediction, i.e., to determine a priori if and when a system will fail due to space/time limits, is a basic need for predictable (software) engineering. Clearly, performance specification and analysis are complicated activities, even when compounding issues such as concurrency and compiler optimization are factored out. Bringing these results into practice will require considerable education and sophisticated tools. More importantly, current language and software design techniques that focus on functional flexibility need to be re-evaluated with attention to predictable performance.

ACKNOWLEDGMENTS

It is a pleasure to acknowledge the contributions of members of the Reusable Software Research Groups at Clemson University and The Ohio State University. We would especially like to thank Greg Kulczycki, A. L. N. Reddy, and Bruce Weide for discussions on the contents of this paper. Our thanks are also due to the referees for their suggestions for improvement.

We gratefully acknowledge financial support from the National Science Foundation under grants CCR-0081596 and CCR-0113181, and from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office.

REFERENCES

1. 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.
2. *Generic Programming*, eds. M. Jazayeri, R. G. K. Loos, and D. R. Musser, LNCS 1766, Springer, 2000.
3. Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Transactions on Software Engineering,* Vol. 17, No. 5, May 1991, pp. 424-435.
4. Hehner, E. C. R., "Formalization of Time and Space," *Formal Aspects of Computing*, Springer-Verlag, 1999, pp. 6-18.
5. Heym, W.D. *Computer Program Verification: Improvements for Human Reasoning.* Ph.D. Dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, 1995.
6. Hooman, J., *Specification and Compositional Verification of Real-Time Systems*, LNCS 558, Springer-Verlag, New York, 1991.
7. Jones, R., Preface, *Proceedings of the International Symposium on Memory Management, ACM SIGPLAN Notices 34,* No. 3, March 1999, pp. iv-v.
8. Leavens, G., "Modular Specification and Verification of Object-Oriented Programs", *IEEE Software,* Vol. 8, No. 4, July 1991, pp. 72-80.
9. Leino, K. R. M., *Toward Reliable Modular Programs*, Ph. D. Thesis, California Institute of Technology, 1995.
10. 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.
11. Lynch, N. and Vaandrager, F., "Forward and backward simulations-Part II: Timing-Based Systems," *Information and Computation*, 121(2), September 1995, 214-233.
12. 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.
13. Reddy, A. L. N., *Formalization of Storage Considerations in Software Design,* Ph.D. Dissertation, Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, 1999.
14. Schmidt, H. W. and Chen, J. Reasoning About Concurrent Objects. In *Proceedings of the Asia-Pacific Software Engineering Conference*, IEEE, Brisbane, Australia, 1995, 86-95.
15. Sitaraman, M., and Weide, B.W., eds. Component-based software using RESOLVE. *ACM Software Eng. Notes 19,4* (1994), 21-67.
16. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E., "Reasoning About Software-Component Behavior," *Procs. Sixth Int. Conf. on Software Reuse*, LNCS 1844, Springer-Verlag, 2000, 266-283.
17. Sitaraman, M., "Compositional Performance Reasoning," *Procs. Fourth ICSE Workshop on Component-Based Software Engineering: Component-Certification and System Prediction*, Toronto, CA, May 2001.
18. 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.
19. Szyperski, *C., Component Software: Beyond Object-Oriented Programming,* Addison-Wesley, 1998.
20. Special issue on Real-Time Specification and Verification, *IEEE Trans. on Software Engineeering*, September 1992.
21. Special section: Workshop on Software and Performance, Eds., A. M. K. Cheng, P. Clemens, and M. Woodside, *IEEE Trans. on Software Engineeering*, November/December 2000.