Modular Verification of Performance Constraints

Joan Krone, William F. Ogden, and Murali Sitaraman

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

May 2003

Copyright © 2003 by the authors. All rights reserved.

Modular Verification of Performance Constraints

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

Abstract

Modular analysis of performance for component-based systems is the focus of this paper. The paper introduces performance contracts that specify time and space. The contracts are expressed in a modular fashion using a suitable coordinate system that admits the consequences of software engineering tenets such as abstraction and parameterization. The paper presents a modular verification system that is based on both contracts of functional behavior and performance. The system is designed to handle the necessary complexity in using non-trivial, generic objects, where performance estimates cannot be metricized (i.e., presented in terms of sizes of objects), and instead need to rely on values from an analysis of functional correctness. To enable automation, the system checks and employs programmer-supplied functional and performance invariants for loops.

Keywords: Components, formal specification, generic objects, predictability, proof rule, static analysis, time and space.

[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] **Address for correspondence**: Dept. Computer Science, Clemson University, Clemson, SC 29634-0974;
E-mail: murali@cs.clemson.edu.

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. A number of recent efforts have addressed the challenge of modular reasoning of functional behavior for object oriented, component based software [[CCW00], [EHO94], [Hey95], [Lea91], [Lei95], [MuP00], and [SAK00]].

While correct functional behavior is critical to any software system, ultimately, a trustworthy system must be predictable in terms of its performance as well [[CCW00], [IEE92], [Jon99], and [Smi90]]. The importance of predictable performance in the context of parameterized components and component-based systems is receiving increasing attention in the software engineering literature [[JM00], [Mey03], [Sch01], [Smi90], and [WMW03]]. Time and space aspects of performance are the focus of this paper. A program that carries out the right job but takes longer than available time 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. Measurement during execution (e.g., using run-time monitoring) is a common approach for analyzing performance of large-scale systems. The objective of this paper is static analysis (and hence, a priori prediction). In particular, the focus is on *modular* or *compositional performance reasoning* [SKK01]: To reason about the (performance) behavior of a system using the behavioral (and performance) specifications of the components of the system, without a need to examine or re-analyze the implementations of reused components. Solving the compositional performance reasoning problem is important for the following reasons:

- Realistic software systems will be inevitably composed from components, and therefore, it is essential to be able to reason about the behavior of component-based systems;
- Software systems fail routinely for performance reasons, and therefore, it is essential to design systems with predictable performance; and
- The reasoning approach should be compositional to scale up.

The rest of this section examines related work in performance analysis and highlights the contributions of the present paper.

1.1 Related Work

Much of the previous work on static performance analysis has concentrated on timing constraints in a real-time context. Of these, the work of Hayes and Utting in [HaU01] is most closely related to the present paper. Using a sequential context, they have presented a comprehensive refinement calculus for time analysis. This work is based on the foundations for real-time specification and refinement by Mahony in [Mah92] and by Utting and Fidge in [UtF96]. Among the key contributions of this work is in explaining that accurate timing analysis will necessarily rely on assertions from functional correctness. For this reason, they base their timing calculus on the calculus for functional behavior given in [Mor94]. The overall objective here is to refine specifications into a machine-dependent real-time programming language. To

accomplish this goal they use the deadline directive. The time expressions given in this paper take the role of deadlines in their work. A significant secondary contribution of this work is noting that loops need to annotated with deadline directives that are proved and used similar in spirit to loop invariants for functional behavior; the deadline estimates for loops in [GHF98] correspond to the elapsed time estimates for loops given in this paper. The deadline directives place demands on the compiler as well. Others have discussed techniques for getting more exact time bounds by including processor timing issues [[[LBJ95], LiG98]]. These ideas are complementary to the framework discussed in this paper because we make it possible to parameterize time expressions to account for the performance of support systems such as compilers and machines.

A precursor to the work of Hayes and Utting is Hooman's work on real-time analysis [Hoo91]. Hooman's emphasis is on real-time compositionality in a concurrency setting. Hooman does not introduce deadlines nor does he allow for variability in timing of operations such as object assignments. His loop specification does not include the equivalent of an elapsed time assertion either. Distributed system timing analysis is the focus of the work by Lynch and Vaandrager in [LyV95]. Earlier, Alan Shaw introduced a formal system for time verification for constructs in higher-level languages in [Sha89]. This work did not address modularity issues.

Our work differs from the previous work in timing analysis in its emphasis on data abstraction and modular verification involving non-trivial, generic objects. The complexity of our specification and verification system stems from the need to handle generalized performance expressions for operations that are parameterized by both input and output values of arbitrary objects (among other factors, such as situations and support systems). This generalization is essential to establish tight bounds, instead of settling for maximums among alternatives. Furthermore, by providing proof rules for performance analysis that show the direct dependence of performance estimates on abstract object values and assertions from functionality specification and verification, our proof system makes the necessary connections explicit.

While there has been significant research in time analysis, space analysis has received relatively modest attention. Real-time and embedded systems need to be concerned with space utilization both because critical systems should not run out of storage and because space constraints often dictate timing constraints. Schmidt and Zimmerman have discussed an additive calculus for both time and space in [ScZ94] and [Sch01]. They propose an operational approach that allows accounting for lower-level machine details. This work is among the first to generalize performance analysis for non-trivial objects, though parameterization and value-based analysis complexities are not considered. Compositionality was not among the initial objectives either.

Hehner has presented an approach for both specification and verification of space constraints [Heh99]. He describes a formal proof system for time and (maximum and average) space analysis. Hehner captures the distinctions necessary to handle time and space, because space utilization increases and decreases, unlike time. A key result of Hehner's work is in illustrating using a simple, yet complete example that the process can be automated. Our proof system is intended for automation as well. Both systems can handle dynamic memory management. In verification of recursive procedures and loops, we expect time and space clauses to be supplied by a programmer for automation, though the need for the clauses is neither shown nor obvious in

the (simpler) recursive procedures given in Hehner's paper. Our calculus is more involved because we address data abstraction and modular reasoning about space usage, even before the implementations are available, using situational assumptions. The parametric nature of space for non-trivial objects and their operations (as in the case of time analysis) is the other necessary complicating factor in our work. An early precursor to this work is Mary Shaw's paper on performance specification and verification (assuming simplified expressions) [Sha79], where she has introduced (ghost) variables for performance analysis.

We have discussed a system for order-of-magnitude descriptions of space and time behavior [SiW94] and more detailed specifications [Red99] elsewhere. Our goal in this paper is to build upon related work and illustrate how to prove performance correctness for component-based systems in a modular fashion.

1.2 Contributions

The paper makes two key contributions. The first of these concerns specification of *performance contracts*. The contracts include expressions for time and space. A key sub-contribution is identification of a suitable set of coordinates necessary to express the contracts in a modular fashion and thus be applicable for a component-based setting. The second contribution is a modular system for performance verification using both the contracts of functional behavior and performance contracts. The paper includes proof rules for procedure calls and loop constructs. A key sub-contribution here is in handling the necessary dependency of performance contracts of functional behavioral specifications. This dependency arises in handling non-trivial objects, where performance estimates cannot be metricized (i.e., presented in terms of sizes of objects), and instead need to rely on values from an analysis of functional correctness. Section 2 discusses specification of performance contracts. Section 3 discusses modular verification. Section 4 contains a summary.

2. SPECIFICATION OF PERFORMANCE CONTRACTS

While the focus of this paper is on performance reasoning, it is essential to understand the complexities in performance specification to motivate aspects of the reasoning process. Ideally the performance specification for an implementation will be expressed in conceptual terms that are understandable to users without a complete knowledge of its internal details. This section provides a summary of factors that complicate modular performance analysis using simple examples.

2.1 An Example

The specification of a generic Queue data abstraction is given in the Appendix. In the specification, a Queue is modeled as a mathematical string of entries of some arbitrary type. To manipulate objects of type Queue, the data abstraction provides primary operations Dequeue, Enqueue, and Length, among others.

The specification of a secondary **operation**[4] to copy (and append) one Queue to another is given in **Figure 1**. The operation **updates** the Queue *P* as specified in the **ensures** clause. In the **ensures** clause, #P stands for the incoming values of Queue P and "∘" denotes string concatenation. The operation **restores** the second queue Q, and this means that the conjunction Q = #Q is implicit in the ensures clause. (Specifications may also have **requires** clauses or preconditions, though the current specification does not include one.)

```
Operation Copy_Q_to(updates P: Queue; restores Q: Queue);
    ensures P = #P ∘ Q;
```

Figure 1: Functionality Specifications of a Queue Operation[5]

2.2 A Simple Performance Contract

An Implementation

To discuss performance specification issues, it is necessary to consider details of a procedure to implement the Copy_Q_to operation. Making a copy of Q involves making a copy of each of its entries. Accordingly, we will import a procedure Copy_E_to. The procedure for Copy_Q_to is then straightforward. Following the while statement, we have used a constant-time swap statement to transfer the value from the local queue R to the parameter Q.

```
Procedure Copy_Q_to(updates P: Queue; restores Q: Queue);
    Var R: Queue;
    Var E, E_Copy: Entry;

    While Length(Q) ≠ 0 do
            Dequeue( E, Q );
            Copy_E_to(  E_Copy, E );
            Enqueue ( E, P );
            Enqueue( E_Copy, R );
    end;
    Q :=: R;
end Copy_Q_to;
```

Figure 2: An Implementation to Copy a Queue[6]

[4] Without the loss of generality, we have used the RESOLVE [SiW94] specification and implementation notation to illustrate the issues throughout this paper. The same performance questions raised here need to be addressed in generic component development in any language, such as C++ or Java, with or without formal specifications. Similarly, the exact formal notation does not affect the issues either.

[5] The context for the specification, such as the mathematical modeling of Queues, is implicit in this figure. The context is given explicitly in **Figure 10** in section 3.

[6] The context for the implementation, such as the specification of the operations Copy_Q_to and Copy_E_to, is implicit in this figure. The context is given explicitly in **Figure 11** in section 3.

Assumptions and Guarantees

In general, we can assume that a procedure that implements an operation P(**updates** x: T) running on a specific input *x* (i.e., x: T ⊆ Dom) should complete its computation within a time $\mathbf{Dur}_P(x)$. This duration $\mathbf{Dur}_P(x)$ is clearly going to be a function of *x* in most cases, so formally we define:

$\mathbf{Dur}_P$: Dom→ℝ, where Dom is the mathematical domain associated with the type T.

For example, the Copy_Q_to procedure takes two queues as parameters, and hence the input domain is Queue × Queue, the mathematical domain meaning of which is Str(Entry) × Str(Entry), since queues are modeled abstractly as mathematical strings in the specification of Queue_Template given in the Appendix.[7]

Now we can add syntax for duration specifications and in a typical situation we would expect Copy_Q_to heading to read like:

Procedure Copy_Q_to(**updates** P: Queue; **restores** Q: Queue);
  duration $C_1 \cdot |\#Q| + C_2$;

Here C_2 accounts for the general setup activities and C_1 accounts for the cost for copying each entry from Q to P. Unfortunately, this formula is too simplistic because the expression is based upon a *hidden* assumption that all the operations on queues have constant duration estimates that are independent of their arguments. Hidden assumptions are dangerous because there could be grossly inefficient implementations of queues in which the duration of say Dequeue(E,Q) is proportional to the length of Q. So a duration estimate for Copy_Q_to or any other procedure must be attached to a particular situation specification making assumptions explicit or must be parameterized appropriately to be quite general.

The performance specification problem takes on a considerable dimension of complexity when reusing a parameterized data abstraction whose implementation has not been selected yet [[SKK01], [SiW94]]. This is because the performance of the new component needs to be parameterized to allow any implementation of the reused data abstraction to be chosen.

Given the potential complexity of parameterized performance expressions, a *performance contract* defines one or more hypothetical situations, and then expresses performance specifications only for those situations. Each situation is a formal statement of performance requirements of other participating components. If the hypothesized situation is valid in a client application, then it is appropriate to reason about that application using the simplified performance expressions.. When a postulated situation does not hold in a context, then its performance guarantees do not apply.

[7] In this paper, we assume that the durations depend on only the input domain. This assumption is implicit in the specification of the normal duration situation. This assumption is not valid, in general. Procedures for operations with relational specifications might take different durations depending on which of the several outputs are produced. The duration function domains then would be a cross product of input and output domains.

Duration Specification for a Simple Situation

To achieve both modularity of performance description and simplicity of expression, we need to anticipate the forms of supporting performance specifications. Here, we define a simple situation formally.

Duration Situation Simple:

$\exists$ C_D, C_E, C_L, C_{CE}, C_{EI}, $C_{:=:}$, $C_=$, C_+: $R^{>0}$ ∍ C_D = LUB($\mathbf{Dur}_{Dequeue}$[Entry×Queue]) **and** C_E = LUB($\mathbf{Dur}_{Enqueue}$[··· **and** C_{CE} = LUB($\mathbf{Dur}_{Copy_Entry}$[Entry×Entry]) **and** …

The **duration** for Copy_Q_to (in addition to its parameters) depends on the durations of the operations it calls, such as basic Queue operations, operations on type Entry, and Integer operations. In the simple situation, we assume all these calls take a constant time. Stated formally, we have fixed durations C_{EI} and C_{CE}, respectively, to bound the durations to initialize and copy an entry. C_E, C_D, and C_L are used to bound the durations of Queue operations Dequeue, Enqueue, and Length. $C_{:=:}$ denotes the time for the swap statement and $C_=$ is the bound for the duration of Integer comparison operation. The values of these constants will depend upon the details of the supporting hardware and software.

To simplify our duration expression for the iterative Copy_Q_to procedure, it is convenient to define two other constants. The first of these denotes the **duration** of each iteration. The second constant denotes the duration for actions outside the iteration:

Definition $C_1 = C_D + 2 \cdot C_E + C_L + C_{CE} + C_=$;
Definition $C_2 = 2 \cdot C_{EI} + C_= + C_{:=:} + \mathbf{Dur}_{Call}(2)$;

Given these definitions, the specification of Copy_Q_to procedure is given in Figure 3 for the simple situation. The **duration** expression specifies an upper bound on execution time. Naturally the expression depends on the parameters to the procedure and the durations of other procedures that are called. In the expression, notice that we have used Q (the result value of parameter Q) instead of #Q (the input value), because Q is restored by the operation.

Procedure Copy_Q_to(**updates** P: Queue; **restores** Q: Queue);
duration Simple: $C_1 \cdot |\#Q| + C_2$;

Figure 3: Duration Specification for a Simple Situation

2.3 "Value-Based" Performance Specifications

While the performance expression in the first example is based on the sizes of queues (i.e., "metricized") as in classical algorithms analysis, this is not adequate in general. The second example clarifies that the space-time behavior of a component will need to be expressed in terms of the *values* of objects involved, *not* merely their *sizes.* For complex generic objects, such as queues containing arbitrary entries, if the performance estimates do not account for the arbitrary nature of the entry type, then the estimates are of little value.

To provide estimates for copying a queue containing a complex type such as trees, we need to account for the particular trees in the queue, not merely the number of trees. This is because a few large trees may take much longer to copy or destroy, and may occupy much more storage capacity than a queue containing several empty trees. More importantly, for generic components, we need to provide performance expressions independent of (and even before we know) the type of parametric objects. To account for the variability in duration to copy entries, we define a "normal" situation.

Duration Specification for a "Normal" Situation

In the normal situation, we do not make the assumption that it takes a constant time to copy entries, though we assume Queue operations such as Dequeue, Enqueue, and Length still take a constant time.

Duration Situation Normal: $\exists$ C_D, C_E, C_L, C_{EI}, $C_{:=}$, $C_{=}$, C_{+}: $R^{>0}$ ∍

C_D =LUB($\mathbf{Dur}_{\text{Dequeue}}$[Entry×Queue]) **and** C_E = LUB($\mathbf{Dur}_{\text{Enqueue}}$[···

In the duration specification in **Figure 4**, we have used a mathematical definition *Copying_Dur* for the duration to copy all entries in a string. This definition is based on the time to copy an entry which is dependent on the specific entry value. In the definition *Occurs_Ct* denotes the number of occurrences of a particular entry in a given string. The constants C_3 and C_4 are defined in terms of the duration bounds given for the Normal situation.

Def Copying_Dur(S: Str(Entry)): $\mathbb{R}^{>0} = \left(\sum_{\text{E:Entry}} \text{Occurs_Ct}(E,S) \cdot \mathbf{Dur}_{\text{Copy_Entry(E)}} \right)$;

Procedure Copy_Q_to(**updates** P: Queue; **restores** Q: Queue);
duration Normal: Copying_Dur(#Q) + $C_3 \cdot |\#Q| + C_4$;

Figure 4: A More Appropriate Duration Specification

It is important to note that there is nothing special about the particular example in this paper that requires specifications in terms of values. In fact, in most non-trivial examples of searching and sorting, an exact time (and space) analysis depends on the exact values of objects. For example, in a "simple" situation, an insertion sort procedure is specified to have a quadratic bound in terms of the size of the container. In actuality, insertion sort may take only a linear time for certain input arrangements, far less than the quadratic upper bounds. A more "normal" situation, for this example, will capture the dependency on the exact value of the container structure that is being sorted. (The case for searching is similar.) An additional dimension of complexity may be introduced if the time to compare different elements is not a constant, but depends on the values of the elements that are compared. This may be the case if non-trivial strings are compared. Our framework allows the definition of a "more complicated" situation where this level of generality is needed.

It is clear that a trade-off is to be made between generality and complexity of performance expressions. There is no a priori way to predict which one is better for component-based

systems. Therefore, the performance specification language must be sufficiently expressive to specify performance for various situations and generality.

2.4 Specification of Storage Space Usage

The second part of the performance specification contract for a procedure gives a storage bound. Just as we defined situations for duration, we need to do so for storage, which we call displacement. Note that displacement introduces some considerations not present when describing durations. It is not just the calling of procedures that requires space, but each object itself takes up some space that must be accounted for in the specifications. We use keyword **Disp** to represent the displacement for a particular object. It is not necessary to distinguish whether the allocations are on run-time stacks or heaps, unless there are individual limits on their sizes.

Of course, there is some base line of displacement taken by the system itself that remains constant throughout the execution of a particular program. What our specifications indicate is the amount of space required above that baseline. As a program executes, displacement above the baseline fluctuates as storage is taken up when variables are declared and returned when blocks are exited. In order to make it possible to keep our reasoning local, we define for each procedure the amount of storage required for it to be able to execute; this includes the storage for all variables a procedure depends on such as its parameters and global variables, as well as local variables. We term the amount of displacement that may be manipulated to carry out the procedure as **manip_disp** (or **Mnp_Disp**) for manipulation displacement,.

In a situation we call *normal*, we assume that the displacement for a particular object depends on its component parts. When procedures are known to take only a small fixed space independent of the values of their parameters, we say that the amount of space is nominal and use **Is_Nominal** to indicate this. For the current example, we need to account for the fact that each queue takes space according to the number and type of its entries. Since we are dealing with a generic queue, we cannot know what the entry type is in advance, and we need to express this dependence using our **Disp** function in terms of the generic type. In writing the displacement specifications for the Queue procedure, it is useful to define a mathematical definition that allows us to talk about the number of times a particular entry, *E*, appears in a string of entries (which is the abstract mathematical model of a queue).

$$\textbf{Def}\ \text{Cnts_Disp(S: Str(Entry))}: \mathbb{N} = \Big(\sum_{\text{E:Entry}} \text{Occurs_Ct(E,S)} \cdot \textbf{Disp}\text{(E)} \Big);$$

Using this definition and the keywords defined previously, we provide a normal displacement situation for our queue example:

Displacement Situation Normal: $\exists\ D_{FQD}, D_{FQED}, D_{EID}: \mathbb{N} \ni \forall$ Q: Queue,
$\textbf{Disp}(Q) = D_{FQD} + D_{FQED} \cdot |Q| + \text{Cnts_Disp}(\ Q\ )$ **and**
$D_{EID} = \textbf{Disp}_{\text{Entry.Init_Val}}$ **and** $\forall$ E: Entry, $\textbf{Disp}(E) \geq D_{EID}$ **and**
Is_Nominal(**Mnp_Disp**$_{\text{Dequeue}}$(E, Q)) **and Is_Nominal**(**Mnp_Disp**$_{\text{Enqueue}}$(E, Q)) **and**
Is_Nominal(**Mnp_Disp**$_{\text{Is_Length}}$(Q));

In our normal situation, we assume that the **Disp** specification for a (generic) Queue object is of a general form that is based on three pieces: D_{FQD}, a fixed displacement, such as might be necessary to store the header information for a queue; $D_{FQED} \cdot |Q|$, displacement that is necessary to link queue entries and is dependent only on the length of the queue; and Cnts_Disp(Q), the summative displacement for the actual entries in the queue. We assume that the space required to create a local variable of type Entry (which may be non-trivial) is denoted by D_{EID} for Entry initialization displacement. This is necessary because at least one local variable of type Entry will need to be declared in the Copy_Q_to procedure. We also assume that the space necessary for Entry initialization process is less than the displacement for any entry; without this assumption, we will need to account for the complexity that the task of copying entries might take up more space than the actual space required for the copied entries. Finally, we assume that the manipulated displacements of the procedures Dequeue, Enqueue, and Length that are invoked in Copy_Q_to are nominal, i.e., they take a small fixed space independent of the values of their parameters. We expect the underlying system to have a sufficiently large, yet a fixed space reserved to handle typical procedure invocation overheads.

Figure 5 contains the entire duration and space performance specification in the normal situation for the Copy_Q_to procedure.

> **Duration Situation** Normal: $\exists\ C_D, C_E, C_L, C_{CE}, C_{EI}, C_{EI}, C_{:=}, C_{=}, C_{+}: R^{>0}\ \ni$
>
> C_D =LUB($\mathbf{Dur}_{Dequeue}$[Entry×Queue]) **and** C_E = LUB($\mathbf{Dur}_{Enqueue}$[···
>
> **Def** Copying_Dur(S: Str(Entry)): $\mathbb{R}^{>0} = (\sum_{E:Entry} \text{Occurs_Ct}(E,S) \cdot \mathbf{Dur}_{Copy_Entry(E)});$
>
> **Def** $C_3 = C_D + 2 \cdot C_E + C_L + C_{\neq}$;
> **Def** $C_4 = 2 \cdot C_{EI} + C_{=} + C_{:=:} + \mathbf{Dur}_{Call}(2)$;
>
> **Def** Cnts_Disp(S: Str(Entry)): $\mathbb{N} = (\sum_{E:Entry} \text{Occurs_Ct}(E,S) \cdot \mathbf{Disp}(E));$
>
> **Displacement Situation** Normal: $\exists\ D_{FQD}, D_{FQED}, D_{EID}: \mathbb{N} \ni \forall$ Q: Queue,
> $\mathbf{Disp}(Q) = D_{FQD} + D_{FQED} \cdot |Q|$ + Cnts_Disp(Q) **and**
> $D_{EID} = \mathbf{Disp}_{Entry.Init_Val}$ **and** $\forall$ E: Entry, $\mathbf{Disp}(E) \geq D_{EID}$ **and**
> **Is_Nominal**($\mathbf{Mnp_Disp}_{Dequeue}$(E, Q)) **and Is_Nominal**($\mathbf{Mnp_Disp}_{Enqueue}$(E, Q))
> **and Is_Nominal**($\mathbf{Mnp_Disp}_{Is_Length}$(Q));
>
> **Procedure** Copy_Q_to(**updates** P: Queue; **restores** Q: Queue);
> **duration** Normal: Copying_Dur(#Q) + $C_3 \cdot |\#Q| + C_4$;
> **manip_disp** Normal: Cnts_Disp (#P) + 2 · Cnts_Disp (#Q)
> $+ (|\#P| + 2 \cdot |\#Q|) \cdot D_{FQED} + 3 \cdot D_{FQD} + 2 \cdot D_{EID}$;

Figure 5: A Performance Specification Contract

The example makes it clear that the capacity requirement of a procedure depends on the capacity required for its local variables as well as capacity needed for any changes the procedures makes to its parametric objects. This is the reason why we have used the term "manipulated

displacement" instead of, for example, "transitional" displacement. In other words, while some of the space used by a procedure may be transitional, the rest of it might impact the storage availability after the return of the procedure.

2.5. Other Complexities in Performance Analysis

This section discusses other issues that complicate performance specification. Since these issues are orthogonal to the topic of this paper, we list them here only for reasons of completeness.

In the examples given here, we have been able to express resource usage estimates strictly in terms of the abstract values of objects, using the abstract models from the functionality specification. Notice that in the examples, the estimates are given based on the abstract "string" values of incoming queues. However, the level of abstraction that is suitable for describing functional behavior may not have sufficient structure for expressing performance estimates precisely. In general, the abstract models may need to be enriched with implementation-specific models. For example, representations of objects may involve hidden internal storage structures for improved execution time efficiency, structures that should not and will not be visible in a functional specification of the object behavior. But the expressions for storage usage for objects and execution time for operations will depend on these internal structures. To facilitate this possibility, in general, abstract models used in behavioral specifications need non-trivial extensions, depending on the precision required in performance expressions [WOS03].

A significant dimension of complexity is added to the performance specification problem when hidden internal structures are shared among a collection of objects [[EHO94], [Red99]]. Such sharing is routinely used in practical software to amortize execution time and space usage costs across operations. In the simpler scenario, sharing is limited to being among homogeneous objects using a large pre-allocated structure. In the more general case, global pointers are used to enable sharing across heterogeneous objects. In the presence of such sharing, accounting must be done carefully to avoid potential double-counting problems, as in a case when a single storage structure (using aliases) is used to represent multiple abstract values. Precise time and storage usage expressions in these cases depend on having suitable abstractions for pointer handling as well as sophisticated global models with sufficiently rich structure for performance analysis.

It is clear that the overall performance of a software system critically depends on support systems such as memory managers and optimizing compilers. The functional and performance behaviors of support systems must be captured because they form the foundations for reasoning about any other system. For predictability, it is important to have a memory manager that is predictable in terms of allocation/deallocation times as well as storage usage. While these issues are quite important, an analytical approach to performance analysis is ultimately necessary to take advantage of expected research advances in predictability of supporting systems.

3. MODULAR VERIFICATION

Section 2 introduced ideas and terminology necessary for specifying performance constraints for generic, object oriented software. It is particularly important to note that our goal is not only to specify performance constraints, but to provide a proof system that allows us to reason about

these constraints in a modular fashion. The proof rules in this section are intended to support automated, modular reasoning for generic components. We begin with a simple rule that deals only with functional correctness of procedure calls to get some familiarity with the form of our rules, and then we progress to more complicated rules that include performance.

3.1 Verification of Functional Behavior for Procedure Calls

Suppose that we have an operation with the specification, termed *P_Heading*, as given below:

Operation P(**updates** x: T);
  requires P_Usg_Exp;
  ensures P_Rslt_Exp;

In order to reason about this procedure when it is called, it is necessary that the specifications be available to the proof system. To support both modular reasoning and the automated application of our proof rules, we introduce the idea of a *context*, a collection of statements needed for applying our rules to particular program parts. For a procedure to be called, it is important that the specifications for that procedure be available in the context, $\mathcal{C}$. This happens when the procedure is declared. Of course, there may be other items in the context, since $\mathcal{C}$ contains specifications for everything in the scope of this procedure. **Figure 6** shows a procedure call rule concerned only with verification of functional behavior. The denominator of the rule shows that the procedure call takes place within a particular context, after some code, and followed by a clause (the Outcome_Exp may be viewed as a postcondition).

In the numerator in **Figure 6**, we note that the context includes the P_Heading. When *P* is called with parameter *a*, the rule checks that the **requires** clause, `P_Usg_Exp` with the actual argument *a* replacing the formal argument *x* holds. The second clause checks to see that if the called 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. The second and subsequent conjunctions allow for the fact 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 arguments 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. In the assertions above the line *a* stands for the value of the argument *a* before the procedure call and ?*a* stands for (one of its) values after the call. This is why #*x*, the formal name for the parameter's value before the call, is replaced with *a*, and *x*, the name for the parameter's name for the value after the call, is replaced with ?*a*. The 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 rules are designed to avoid the need to introduce more than two points in the time line, simplifying the process and supporting modular reasoning.

$$\frac{\mathcal{C} \cup \{P_Heading\} \setminus Code;\ \textbf{Confirm}\ P_Usg_Exp[x \rightsquigarrow a] \land \forall\ ?a: T,\ \textbf{if}\ P_Rslt_Exp[\#x \rightsquigarrow a,\ x \rightsquigarrow ?a]\ \textbf{then}\ Outcome_Exp[a \rightsquigarrow ?a];}{\mathcal{C} \cup \{P_Heading\} \setminus Code;\ P(a);\ \textbf{Confirm}\ Outcome_Exp;}$$

Figure 6: A Modular Procedure Call Rule for Verification of Only Functional Behavior

3.2 Verification of Functional Behavior and Timing

We turn now to verification of procedures with timing specifications. Since we support the possibility of multiple implementations for single functionally specified operations, we do not put performance requirements in the operation heading, but rather place them in the procedure that implements the given operation. Note that one can verify functional correctness independent of performance. However, verification of performance correctness is meaningless without functional correctness. Suppose that a procedure (code) to realize the above operation specification has the form shown below:

```
Procedure
    duration Dur_Sitn: Dur_Exp;
    manip_disp Disp_Sitn: M_D_Exp;
            P_body;
end P;
```

We have put the displacement specifications here, but we will first show a rule for time verification and then in the next section deal with all three aspects (functionality, duration, and displacement) together. **Figure 7** shows a rule for verifying both the functionality and timing constraints for a procedure call.

To verify timing, we introduce a verifier keyword **Cum_Dur** that stands for cumulative duration. Its purpose is similar to a ghost or verification variable in other approaches. The value of **Cum_Dur** is a non-negative Real number. At the beginning, the cumulative duration is set to zero. As the program executes, the duration increases as each statement needs a specified amount of time to complete. Suppose that the sequence of statements following the call to *P* take time `Sqnt_Dur_Exp` (for subsequent duration expression). Our objective is to confirm that cumulative **duration** added to this subsequent expression is below the specified duration bound `Dur_Bd_Exp`. To confirm this, we need to take into account the duration expression for the call to *P* (denoted by `Dur_Exp`), and substitute appropriate actual arguments for formals. The P_Heading mentioned in the rule now includes both the operation specifications and the performance specifications from the procedure.

$$\mathcal{C} \cup \{\text{P_Heading}\} \setminus \text{Code; } \textbf{Confirm } \text{P_Usg_Exp}[x \rightsquigarrow a] \land$$

$$\forall\ ?a{:}\ \textbf{M_Exp}(T),\ \textbf{if } \text{P_Rslt_Exp}[\#x \rightsquigarrow a,\ x \rightsquigarrow ?a]\ \textbf{then } \text{Outcome_Exp}[a \rightsquigarrow ?a] \land$$

$$(\textbf{Cum_Dur} + \text{Dur_Exp}[\#x \rightsquigarrow a,\ x \rightsquigarrow ?a] + \text{Sqnt_Dur_Exp}[a \rightsquigarrow ?a] \le \text{Dur_Bd_Exp}[a \rightsquigarrow ?a]);$$

$$\mathcal{C} \cup \{\text{P_Heading}\} \setminus \text{Code; } \textbf{Assume } \text{Dur_Sitn; P(a);}$$

$$\textbf{Confirm } \text{Outcome_Exp} \land (\textbf{Cum_Dur} + \text{Sqnt_Dur_Exp} \le \text{Dur_Bd_Exp});$$

Figure 7: A Modular Procedure Call Rule for Time Verification

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 [HaW91]. The complication that arises when an object is repeated as arguments in a procedure call is the topic of a paper under revision.

3.3. A Modular Procedure Call Rule for Functionality and Performance Verification

We turn now to the call rule which includes functionality and performance, both duration and displacement. 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, depending on memory management, the declaration of new variables will cause sudden, possibly sharp, increases in amount of space needed by the program whereas variables leaving scope will cause a possibly sharp decrease in space. To capture this behavior, we introduce two auxiliary variables in verification of displacement. **Prior_Max_Aug** stands for "prior maximum augmentation" of space. 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. The auxiliary variable **Cur_Aug** represents this additional storage or the current augmentation of space. **Prior_Max_Aug** captures the maximum up to a given state, whereas **Cur_Aug** denotes the additional space usage, beyond fixed storage usage, at a particular state.

The picture in **Figure 8** serves to motivate space-related assertions in the procedure call rule. 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 space required for the code itself is included in this fixed storage. Above the fixed storage the execution of the code requires a fluctuating amount of space.

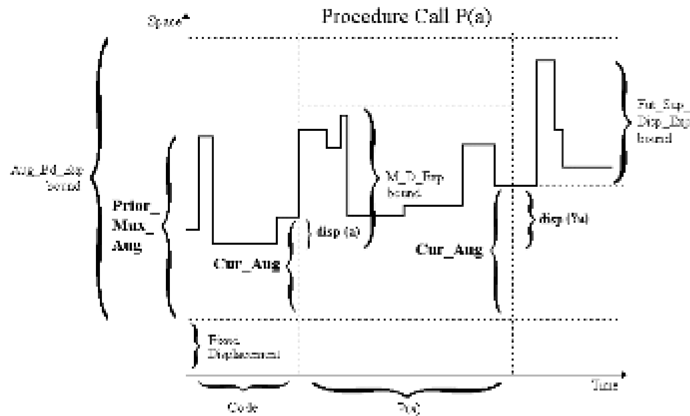

Figure 8: An Illustration of Space Calculus

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. As the execution proceeds, **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 to be considered correct in terms of performance. As the picture shows, the `M_D_Exp` is an expression involving only local variables and parameters. 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. Though performance analysis depends on functional behavioral analysis, it can be kept modularized in the following sense. If a new component replaces another component that provides the same observable functional behavior in a larger system, then it is possible to re-analyze only the performance behavior of the modified system without re-analyzing its functionality.

At the point `P(a)` is called, 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` describes a bound on the storage used by the remaining code, i.e., code following the current statement under consideration. Given below is the procedure call rule:

$\mathcal{C} \cup$ {P_Heading} Code; **Confirm** P_Usg_Exp[x $\rightsquigarrow$ a] $\wedge$

$\forall$?a: **M_Exp**(T), **if** P_Rslt_Exp[#x $\rightsquigarrow$ a, x $\rightsquigarrow$?a] **then** Outcome_Exp[a $\rightsquigarrow$?a] $\wedge$

Cum_Dur + Dur_Exp[#x $\rightsquigarrow$ a, x $\rightsquigarrow$?a] + Sqnt_Dur_Exp[a $\rightsquigarrow$?a] $\leq$ Dur_Bd_Exp[a $\rightsquigarrow$?a] $\wedge$
Max(**Prior_Max_Aug**, **Cur_Aug** – **Disp**(a) +

Max(M_D_Exp[#x $\rightsquigarrow$ a, x $\rightsquigarrow$?a],   **Disp**(?a) + Fut_Sup_Disp_Exp[a $\rightsquigarrow$?a]))

$\leq$ Aug _Bd_Exp[a $\rightsquigarrow$?a];

$\mathcal{C} \cup$ {P_Heading} \Code; **Assume** Dur_Sitn $\wedge$ Disp_Sitn; P(a); **Confirm** Outcome_Exp $\wedge$
(**Cum_Dur** + Sqnt_Dur_Exp $\leq$ Dur_Bd_Exp) $\wedge$
Max(**Prior_Max_Aug**, **Cur_Aug** + Fut_Sup_Disp_Exp) $\leq$ Aug_Bd_Exp;

Figure 9: A Modular Procedure Call Rule for Verification of Functionality and Performance Correctness

3.4. Modular Procedure Body Verification

In this subsection, we consider modular verification of a generic procedure such as Copy_Q_to, introduced in Section 2.1. **Figure 10** makes explicit the context for the specification given in **Figure 1**. The specification of Copy_Q needs to be written within the context of Queue_Template as an additional operation or *enhancement*, as shown in **Figure 10**. An **enhancement** automatically carries with it all aspects of the concept which it is enhancing and then adds some additional operation(s).

Enhancement Copying_Capability **for** Queue_Template
    Operation Copy_Q_to(**updates** P: Queue; **restores** Q: Queue);
        ensures P = #P $\circ$ Q;
end Copying_Capability;

Figure 10: An Enhancement Operation for Queue_Template

Unlike the primary operations in the Queue_Template specification given in the Appendix, such as such as *Enqueue*, *Dequeue*, *Swap_First_Entry*, *Length*, and *Clear*, Copy_Q_to is secondary. In other words, it can be realized using a combination of primary operations. In fact, an enhancement may have many implementations. Discussion of an iterative implementation in **Figure 2** has been the focus of this paper. The context for the realization in **Figure 2** is given in **Figure 11**. Details of the performance expressions are identical to those given in **Figure 5** and are omitted here for brevity. The Easy realization of Copying_Capability imports a procedure to copy the entries in the queue. This is necessary because the type of the entries may be arbitrary. The code and the performance estimates are given in terms of arbitrary generic type of entries.

```
Realization  Easy_Realization(
                    Operation Copy_E_to(replaces Copy: Entry; restores Orig: Entry );
                                   ensures Copy = Orig;  );
                for Queue_Template.Copying_Capability;
        Duration Situation Normal:  …
        Displacement Situation Normal: …
    Procedure Copy_Q_to(updates P: Queue; restores Q: Queue);
        duration Normal: Copying_Dur(#Q) + C3·|#Q | + C4;
        manip_disp Normal: Cnts_Disp (#P) + 2 · Cnts_Disp (#Q)
                + ( |#P| + 2 · |#Q| )· D_FQED + 3 · D_FQD + 2 · D_EID;
        (* procedure body as given in Figure 2 *)
    end Copy_Q_to;
end Easy_Realization;
```

Figure 11: A Realization for the Copying Capability Enhancement

Figure 12 contains a proof rule for verifying that a procedure body meets its behavioral and performance contract. In the rule Example_Realization_Heading refers to the information necessary to use Example_Realization (e.g., realization parameters and performance estimates).

```
C ∪ {Example_Enhancement} ∪ {Example_Realization_Heading}\
        Assume P_Usg_Exp ∧ Dur_Sitn ∧ Cum_Dur = 0.0 ∧
        Disp_Sitn ∧  Prior_Max_Aug = Disp(x) ∧ Cur_Aug = Disp(x);
                P_Body;
        Confirm P_Rslt_Exp ∧  Cum_Dur + 0.0 ≤ Dur_Exp ∧
                Max( Prior_Max_Aug, Cur_Aug) ≤ M_D_Exp;
______________________________________________________________
C\      Enhancement Example_Enhancement for Example_Concept;
                Operation P (  x : T );
                        Requires P_Usg_Exp;
                        Ensures P_Rslt_Exp;
        end Example_Enhancement;
        Realization Example_Realiz(Oper O(  ); req O_Req_Exp; ens O_Rslt_Exp; )
                        for Example_Concept.Example_Enhancement;
                …
                Duration Situation Dur_Sitn = …;
                Displacement Situation Disp_Sitn = …;
                Procedure P (  x  );
                        duration Dur_Sitn: Dur_Exp;
                        manip_disp Disp_Sitn: M_D_Exp;
                        P_Body;
                end P;
        end Example_Realiz;
```

Figure 12: A Functionality and Performance Contract Verification Rule

In the rule in **Figure 12**, within the context $\mathcal{C}$, we **assume** that the **requires** clause is true at the beginning and we need to **confirm** that the **ensures** clause is true at the end of the procedure body. For time verification, we need to show **Cum_Dur** accumulates to a value no greater than the duration bound, assuming the given duration situation. For local space verification, we set the **Prior_Max_Aug** and the **Cur_Aug** both to be the amount of space required by variables before the procedure is called, such as parameter x at the beginning. **Disp**(x) denotes the displacement of the procedure's parametric object, and it depends on the value of x as noted earlier. After the body, the rule checks that the max over the stated values is within the specified bound.

The important aspect of the proof rule is that it is possible to handle the verification of the procedure in a modular fashion using locally available assertions and the limited context of the specifications of reused operations. We have not addressed verification of data abstraction implementations in this paper. While the principles are similar, additional factors such as object displacement expressions (in addition to procedure displacement) need to be verified.

3.5. Verification of Loops

We conclude the discussion on performance verification with a proof rule for loops. First we annotate the loop in the Copy_Q_to procedure from **Figure 11** with assertions necessary for automated verification.

```
Realization Easy_Realization for …
    …
  Procedure Copy_Q_to(updates P: Queue; restores Q: Queue);
    duration Normal: …
    manip_disp Normal: …
    Var R: Queue; Var E, E_Copy: Entry;
    While (Length(Q) ≠ 0)
            affecting P, Q, R, E, E_Copy;
            maintaining (P ∘ Q = #P ∘ #Q) ∧ R ∘ Q = #Q;
            decreasing |Q|;
            elapsed_time Normal: Copying_Dur(R) + C3*|R | + C4 ;
            max_manip_disp Normal: Cnts_Disp (P ∘ Q ∘ R) +
                            | P ∘ Q ∘ R | · D_FQED + 3 · D_FQD + 2 · D_EID;
    do
            Dequeue( E, Q );
            Copy_E_to (E_Copy, E);
            Enqueue ( E, P );
            Enqueue( E_Copy, R);
    end;
    Q :=: R;
  end Copy_Q_to;
```

Figure 13: A Procedure Annotated with Suitable Loop Assertions for Automated Verification

In verifying correctness of loops, loop invariants and progress metrics for establishing termination of loops are necessary. Since these assertions cannot be generated automatically in general, a loop programmer must supply these assertions.

- An **affecting** 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_disp** clause (for each situation assumption) that denotes the maximum space manipulated since the beginning of the loop in any iteration.

The proof system first establishes the correctness of the given assertions, and then employs them in proofs. We give a straightforward total correctness rule for loops in **Figure 14**. The first hypothesis confirms that the invariant is true for the loop. The second one confirms that the invariant is true at the beginning of the loop, and employs it in proving the assertion Q following the loop. In the expressions **M_Exp** is used to denote the mathematical expression that corresponds to the condition B_Exp. In the loop in **Figure 13**, the loop condition is Length(Q) $\neq$ 0, and it corresponds to $|Q| \neq 0$ from the specification of Length operation given in the Appendix.

We introduce ?u to denote the value of the variable u at the end of the loop. Outcome_Exp needs to be confirmed for whatever value u might have after the loop. While the use of the variable can be avoided in this first version of the loop rule, it becomes necessary in recording displacement assertions.

C \ **Assume** **M_Exp**[B_Exp] $\wedge$ Inv $\wedge$ **P_Val** = P_Exp;
    body;
  Confirm Inv $\wedge$ P_Exp < **P_Val**;
C \ Code; **Confirm** Inv **and**
    ($\forall$?u: T, ($\neg$**M_Exp**[B_Exp] $\wedge$ Inv) [u $\rightsquigarrow$?u] $\Rightarrow$ Outcome_Exp[u $\rightsquigarrow$?u]);

C \ Code; **While** (B_Exp) **affecting** u; **maintaining** Inv; **decreasing** P_Exp; **do**
        body
    end;
    Confirm Outcome_Exp;

Figure 14: A Total Correctness Loop Rule

Figure 15 contains the loop rule for functionality and performance verification. The first hypothesis checks that the elapsed time and maximum manipulated displacement assertions are invariants. The second hypothesis is set up in the same way as the procedure call, using ?u to denote the value of the variable u at the end of the loop and u to denote its value at the beginning.

𝒞 \ **Assume** **M_Exp**(B_Exp) ∧ Inv ∧ **P_Val** = P_Exp ∧
Cum_Dur ≤ E_T_Exp ∧
Cur_Aug ≤ **Disp**(u) ∧ **Prior_Max_Aug** ≤ M_D_ Exp;
body;
Confirm Inv ∧ P_Exp < **P_Val** ∧
Cum_Dur + 0.0 ≤ E_T_Exp ∧
Max(**Prior_Max_Aug**, **Cur_Aug** + 0) ≤ M_D_Exp;

𝒞 \ Code; **Confirm** Inv ∧ E_T_Exp ≥ 0.0 ∧ M_D _Exp ≥ 0 ∧
∀ ?u: T, (¬ **M_Exp**(B_Exp) ∧ Inv)[u ⇝ ?u] ⇒
(Outcome_Exp ∧ **Cum_Dur** + E_T_Exp + Sqnt_Dur_Exp ≤ Dur_Bd_Exp)[u ⇝ ?u] ∧
Max(**Prior_Max_Aug**, **Cur_Aug** - Disp(u) +
Max(M_D_Exp[u ⇝ ?u], Fut_Sup_Disp_Exp[u ⇝ ?u] + Disp(?u)))
≤ Aug_Bd_Exp[u ⇝ ?u];

𝒞 \Code; **While** (B_Exp) **affecting** u; **maintaining** Inv; **decreasing** P_Exp;
elapsed_time E_T_Exp; **max_manip_disp** M_D_Exp;
do body **end**;
Confirm Outcome_Exp ∧ **Cum_Dur** + Sqnt_Dur_Exp ≤ Dur_Bd_Exp ∧
Max(**Prior_Max_Aug**, **Cur_Aug** + Fut_ Sup_Disp_Exp) ≤ Aug_Bd_Exp;

Figure 15: A Performance Correctness Loop Rule

The proof rules for other constructs such as if-then-else statements, swap statements, and function assignments are straightforward.

3.6. Verification of Main Programs

While we have concentrated this paper on performance specification and verification of a generic procedure involving non-trivial objects, the overall objective is to verify that a program (or process) works within specified time and space requirements. This is accomplished by verifying that the "main" or calling procedure or process with "total permissible time" as its duration bound and "total memory capacity" as its displacement bound. However, the specification and verification of the calling program will be parameterized by inputs (and outputs) of that program. It is the calling program that will fix the parameters of reusable concepts (e.g., the type Entry) and choose specific implementations for the concepts and their enhancements. Once these choices are made, then in the performance analysis of the calling program, situational assumptions can be checked and simplified performance expressions can be used for the reused components. The verification process discussed here makes it possible to accomplish this goal in a modular fashion – one component at a time.

IV. SUMMARY

The importance of performance considerations in component-based software engineering is well documented. Designers of languages and developers of object-based component libraries have considered alternative efficient implementations providing performance trade-offs, including parameterization for performance. While efficiency is important, performance predictability is essential for high confidence systems. To guarantee predictability, we need an analytical method for performance prediction, i.e., to determine a priori if and when a system will fail due to space/time limits. We have explained how this basic need for predictable (software) engineering can be addressed in a modular fashion, within modern software engineering tenets of abstraction of objects and parameterized components.

Clearly, performance specification and analysis are complicated activities. Bringing these results into practice will require considerable education and sophisticated tools. More importantly, current language and software design techniques for component-based software engineering 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 also gratefully acknowledge financial support from the National Science Foundation under grants CCR-0081596 and CCR-0113181.

APPENDIX

Figure 16 shows the specification of a Queue **concept**. Queue_Template is a specification template parameterized by the type of items to be contained in queues.

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

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

```
Concept Queue_Template ( type Entry );
        uses Std_Integer_Fac, String_Theory;

    Type Family Queue ⊆ Str(Entry);
        exemplar Q;
        initialization ensures Q = Λ;

    Operation Enqueue(clears E: Entry; updates Q: Queue);
        ensures Q = #Q ∘ <#E>;

    Operation Dequeue(replaces R: Entry; updates Q: Queue);
        requires |Q| > 0;
        ensures #Q = <R> ∘ Q;

    Operation Swap_First_Entry(updates E: Entry; updates Q: Queue);
        requires |Q| > 0 ;
        ensures ∃ Rem: Str(Entry) ∋ #Q = <E> ∘ Rem and Q = <#E> ∘ Rem;

    Operation Length(restores Q: Queue): Integer;
        ensures Length = ( |Q| );

    Operation Clear(clears Q: Queue );
end Queue_Template;
```

Figure 16: Specification of a Queue_Template Concept

Using an **exemplar** queue variable Q, the specification tells a client what is true of every queue variable. From the **initialization ensures** clause, a client has the guarantee that whenever a queue variable is declared, it has Λ for its initial value. (Occasionally, the initialization clause may specify a set of possible initial values, instead of a single value.) The practice of providing well-defined initial values, in situations where it is natural is often helpful to avoid a class of routine software errors. The rest of the **concept** provides specifications of other Queue operations.

Each operation is specified by a **requires** clause (precondition), which is an obligation for the caller; and an **ensures** clause (postcondition), which must be guaranteed from a correct implementation. When users violate the requirements, guarantees become void. Operation Enqueue, for example, guarantees that after Enqueue is called, *Q* will be updated to be the incoming value of entry (denoted by *#E*) concatenated with the incoming queue (denoted by *#Q*). 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 **ensures** clauses of operations to be loose to allow maximum flexibility for implementers.

In a similar fashion, the requirement and guarantee clauses on the Dequeue, Length, Swap_Front, and Clear operations specify in string theoretic terms precisely what they will do. The **restores** mode in the specification of Length has the meaning that the parametric queue is

unaffected by calls to these operations. Clear gives its parametric queue Q the initial queue value Λ, and it gets this meaning from the **clears** parameter mode. (The **evaluates** mode allows expressions to be passed as parameters.) The important point here is that by conceiving of queues as strings, we can give a complete and coherent explanation of all of the operations on queues. Absolutely no reference to details of any one implementation possibilities such as arrays, pointers, or linked lists, is needed. Absence of such details simplifies understanding of the concept for users yet provides developmental freedom for implementers of the concept, so long as users and implementers adhere to their obligations and guarantees stated in the specification.

REFERENCES

[CCW00] Special section: Workshop on Software and Performance, Eds., A. M. K. Cheng, P. Clemens, and M. Woodside, *IEEE Trans. on Software Engineering*, November/December 2000.

[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.

[GHF98] Grundon, S., Hayes, I. J., and Fidge, C. F., "Timing Constraint Analysis," *Proceedings 21st Australasian Computer Science Conference*, Singapore, Springer-Verlag, January 1998, 575-586.

[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, pp. 424-435.

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

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

[Hey95] 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.

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

[IEE92] Special issue on Real-Time Specification and Verification, *IEEE Trans. on Software Engineering*, September 1992.

[JM00] *Generic Programming*, eds. M. Jazayeri, R. G. K. Loos, and D. R. Musser, LNCS 1766, Springer, 2000.

[Jon99] Jones, R., Preface, *Procs. of the International Symposium on Memory Management, ACM SIGPLAN Notices 34,* No. 3, March 1999, pp. iv-v.

[LBJ95] Lim, S., Bae, Y. H., Jang, G. T., Rhee, B., Min, S. L., Park, C. Y., Shin, H., Park, K., Moon, S., Kim, C. S., "An Accurate Worst Case Timing Analysis for RISC Processors," *IEEE Trans. Software Eng. 21*, 1995, 593–604.

[Lea91] Leavens, G., "Modular Specification and Verification of Object-Oriented Programs", *IEEE Software,* Vol. 8, No. 4, July 1991, pp. 72-80.

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

[LiG98] 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), September 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.

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

[Mor94] Morgan, C. C., *Programming from Specifications*, Prentice Hall, 1994.

[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.

[Red99] Reddy, A. L. N., *Formalization of Storage Considerations in Software Design,* Ph.D. Dissertation, Dept. CSEE, West Virginia University, Morgantown, WV, 1999.

[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*, LNCS 1844, Springer-Verlag, 2000, 266-283.

[Sch01] Schmidt, H., "Trusted Components – Towards Automated Assembly with Predictable Properties," *Procs. Fourth ICSE Workshop on Component-Based*

Software Engineering: Component-Certification and System Prediction, Toronto, CA, May 2001, 64-69.

[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.

[Sit01] Sitaraman, M., "Compositional Performance Reasoning," *Procs. Fourth ICSE Workshop on Component-Based Software Engineering: Component-Certification and System Prediction*, Toronto, CA, May 2001, 3-10.

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

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

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

[UtF96] Utting, M. and Fidge, C. J., "A Real-Time Refinement Calculus That Changes Only Time," *Procs. 7th BCS/FACS Refinement Workshop*, Springer, July 1996.

[WMW03] Wu, X., McMullan, D., and Woodside, M., "Component-Based Performance Prediction," *Procs. Sixth ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, Portland, OR, May 2003, 13 - 18.

[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.