

# Expressiveness Issues in Compositional Performance Reasoning

Bruce W. Weide and William F. Ogden  
Dept. of Computer and Information Science  
The Ohio State University  
2015 Neil Ave.  
Columbus, OH 43210 USA  
+1 614 292 1517  
{weide,ogden}@cis.ohio-state.edu

Murali Sitaraman  
Dept. of Computer Science  
Clemson University  
419 Edwards Hall  
Clemson, SC 29634 USA  
+1 864 656 6738  
murali@cs.clemson.edu

## ABSTRACT

Compositional reasoning about any behavioral property of a system depends, first, on the ability to *express* that property for both individual components and systems constructed from them. Expressiveness problems arise when considering compositional reasoning about *performance* in the presence of complex user-defined types (as opposed to simpler built-in types). There are interesting implications not just for compositional reasoning but for language design and for formal specification.

## Categories and Subject Descriptors

D.2.1 [Requirements/Specifications]: Languages

D.2.4 [Software/Program Verification]: Formal methods

D.3.3 [Language Constructs and Features]: Abstract data types

## General Terms

Performance, Design, Languages, Verification

## Keywords

abstract data type, analysis of algorithms, component-based software, component, execution time, expressiveness, formal specification, model-based specification, performance analysis, performance specification, specification

## 1. BACKGROUND

Programming languages and software engineering methodology have reached the point where commercial software technologies support component-based design. However, this support exists only in the sense that components are *syntactic* modules: systems can be readily constructed from components that are designed and implemented by other parties, components can be compiled

separately, and they can be composed even at run time in some situations. A problem with current component-based software engineering practice is that these components generally are not *behavioral* modules. That is, sound reasoning about the behavior of a component-based system (including reasoning about a larger component built from other components) cannot be accomplished by *compositional* or *modular reasoning* [14]. This entails reasoning separately about the behaviors of the components—once and for all, and in isolation—and then composing the summarized results of such one-time reasoning without again consulting the internal details of previously-analyzed components.

Compositional reasoning about system behavior inherently requires that each component must have a second description in addition to the obvious first description (i.e., that component's implementation code). This second description is the cover story that summarizes the relevant behavior—a *specification* of the aspects of the component's behavior about which reasoning is to be done. In compositional reasoning, a specification substitutes for the component's implementation code during reasoning about the behavior of client code.

From a practical standpoint, a specification of the relevant aspects of component behavior is important because the source code for a component is often unavailable to its clients. Without such a specification to stand in for a component's implementation code, clients could not hope to reason (compositionally or otherwise) about system behavior. And from a technical standpoint, a specification of the relevant aspects of component behavior is indispensable because this is the description against which the component's implementation code is to be evaluated. Whether that code implements its purported specification *is* the reasoning problem at the next lower level of the component hierarchy.

### 1.1 Reasoning About Functional Behavior

Compositional reasoning about functional behavior apparently can be achieved through either of two approaches. One is pure functional programming. The main problem with this approach is that its naive adoption sacrifices performance on the altar of compositional reasoning about functionality. Hence, many researchers are trying to fix the performance problems with pure functional programming, exploring a variety of sophisticated implementation advances that they hope can eventually make pure functional programs more competitive with imperative programs in terms of efficiency (see, among many others, [2, 17]).

The second approach to achieving compositional reasoning about functional behavior is to use an imperative programming style in ways that remove the above-mentioned impediments to sound compositional reasoning. Our previous research has followed this path. One result of this work is the RESOLVE framework, a prescriptive approach to constructing component-based software systems that offer both the generally favorable performance of the imperative style [11] and the capability for compositional reasoning about functional behavior [13] and certain aspects of performance behavior [12, 14]. We have established that these principles are portable to languages such as C++, and that they can be used to build non-trivial systems, including commercial software products, with observable benefits for quality and maintainability [5].

This paper focuses on compositional reasoning about performance in an imperative setting. The computational model is a standard sequential computer. The problem is much more complex in the presence of concurrency and distribution issues, and the sequential situation is a special case that must be handled in any event. Hence, it is appropriate to start there.

## 1.2 Reasoning About Performance

The performance issue is not merely about whether component-based software systems can be “efficient” in a general sense. It is about whether it is possible to reason compositionally about how efficient they are.

This paper focuses on the specification of performance behavior, not on how to do compositional reasoning using such a specification. Prior work has explained how it is possible to use the kinds of “performance profile” specifications discussed in the next section to reason compositionally about execution time in the case of ordinary sequential programs [14].

## 1.3 Contributions of This Paper

The major contribution of this paper is in pointing out and explaining some implications of an important technical difficulty encountered when attempting to do compositional reasoning about performance. The problem arises because models that are sufficient for expressing functional behavior may not be sufficient for expressing performance behavior. This observation might seem trivial at first glance. Of course, a functionality specification model is chosen in part to “abstract away” details such as performance that it does not purport to explain! The issue at hand, though, involves the expressiveness of a *language* for writing performance specifications—not simply the claim that more needs to be written down to reason about performance than to reason about functionality.

We begin with an example to illustrate the problem and, in the process, introduce language constructs that address it. Then we discuss some implications of incorporating these or similar features into a language that could support formalized compositional reasoning (i.e., proofs of performance correctness).

## 2. EXPRESSING PERFORMANCE BEHAVIOR

As an example, we consider a component that defines a Map abstract data type. Similar components of this kind are offered

by typical component libraries, such as the C++ Standard Template Library [9] and the Java `java.util` package. For definiteness in subsequent discussion, in Figure 1 we show a simple Map component specification in a dialect of RESOLVE [11]. The point that this example illustrates is not unique to this particular design or specification. It has nothing to do with the simplification that a Map variable (as defined here) is restricted to recording an association involving existing program types Integer (modeled by a mathematical integer with a bound constraint) and Text (modeled by a mathematical string of characters). It has nothing to do with using RESOLVE, since any other model-based specification language [19] (e.g., Larch, VDM, or Z) could be used to write a similar specification.

```

concept Map_Facility

type Map is modeled by
  finite set of (
    id: integer,
    name: string of character
  )
  exemplar m
  constraint
    for all i, j: integer,
      s, t: string of character
      where ((i,s) is in m and
        (j,t) is in m)
      (if i = j then s = t)
  initialization ensures
    |m| = 0

operation Add_To_Map (
  updates m: Map,
  preserves i: Integer,
  preserves t: Text
)
requires
  for all s: string of character
    ((i,s) is not in m)
ensures
  m = #m union {(i,t)}

operation Remove_From_Map (
  updates m: Map,
  preserves i: Integer,
  replaces t: Text
)
requires
  there exists s: string of character
    such that ((i,s) is in m)
ensures
  m = #m - {(i,t)}

operation Remove_Any_From_Map (
  updates m: Map,
  replaces i: Integer,
  replaces t: Text
)
requires
  |m| > 0
ensures
  (i,t) is in #m and
  m = #m - {(i,t)}

```

```

operation Map_Value (
  preserves m: Map,
  preserves i: Integer
): Text
requires
  there exists s: string of character
  such that ((i,s) is in m)
ensures
  (i,Map_Value) is in m

operation Is_Mapped (
  preserves m: Map,
  preserves i: Integer
): Boolean
ensures
  Is_Mapped =
  there exists
  t: string of character
  such that ((i,t) is in m)

operation Size (
  preserves m: Map
): Integer
ensures
  Size = |m|

end Map_Facility

```

**Figure 1: A simple *Map* component**

Compositional (or for that matter, any) reasoning about the functionality of component-based systems relies on the existence of an abstract mathematical model for each data type. Using this model, it is possible to summarize and thereby explain the functionality of any correct implementation of a single abstract interface, such as that in Figure 1. For purposes of describing the functionality of *Map\_Facility*, an appropriate abstract model of a *Map* value is a mathematical set of ordered pairs. This model is “fully abstract”: it reveals to a client exactly what is necessary to explain the functionality of the component—no more, no less. Another way to say essentially the same thing is that this component design makes the abstract state space for the *Map* type both observable and controllable [18]: a client program can reach every point in the abstract state space, and it can distinguish every point in the state space from every other.

Compositional reasoning about performance requires the same sort of foundation: a way to summarize and thereby explain performance information for a single implementation, i.e., to express component performance. How can this be done?

Suppose that execution times are to be described using traditional  $\Theta$ -notation. This choice determines how precise [14] the performance description is—it ignores multiplicative and additive constants, but it captures functional dependencies. The latter are at the heart of the issue we raise, so the simplifications introduced by using  $\Theta$ -notation do not interfere. (Other work [15] has explored whether traditional “big-oh” notations like this might need to be more carefully defined in order to be used in the context of compositional reasoning about performance. That is not the issue here; traditional  $\Theta$ -notation is enough to illustrate the problem.)

The first implementation illustrates that the abstract model for a *Map* value that was introduced to explain functionality in

*Map\_Facility* might be adequate to explain the performance of an implementation of *Map\_Facility*.

```

performance profile for
  realization Map_Using_Ordered_List
  of Map_Facility

definition RANK_IN_SET (
  i: integer,
  m: Map
): integer
explicit definition
  |{j: integer, s: string of character
  where ((j,s) is in m and j < i)
  (j)}|

operation Add_To_Map (
  updates m: Map,
  preserves i: Integer,
  preserves t: Text
)
duration
  theta (RANK_IN_SET(i,#m) + |t|)

operation Remove_From_Map (
  updates m: Map,
  preserves i: Integer,
  replaces t: Text
)
duration
  theta (RANK_IN_SET(i,#m) + |#t|)

operation Remove_Any_From_Map (
  updates m: Map,
  replaces i: Integer,
  replaces t: Text
)
duration
  theta (|#t|)

operation Map_Value (
  preserves m: Map,
  preserves i: Integer
): Text
duration
  theta (RANK_IN_SET(i,m) + |Map_Value|)

operation Is_Mapped (
  preserves m: Map,
  preserves i: Integer
): Boolean
duration
  theta (RANK_IN_SET(i,m))

operation Size (
  preserves m: Map
): Integer
duration
  theta (1)

end Map_Using_Ordered_List

```

**Figure 2: Performance profile for one implementation of *Map\_Facility***

We consider an ordered list representation of a Map, where the pairs in the list are kept in increasing order by the value of the first (Integer) component. Figure 2 shows how the execution-time performance of this implementation might be described formally, in a RESOLVE-like notation.

Figure 2 is something that a client of the implementation must be able to understand in order to reason about the performance of a client program, along with the functionality specification of Figure 1. RANK\_IN\_SET is a mathematical definition that is used throughout the performance profile; RANK\_IN\_SET(*i*,*m*) is the number of integers that are first components of pairs in *m* and that are less than *i*. The **duration** clause associated with each operation summarizes how long the operation's implementation takes, based on the values of its incoming (#-prefixed) and outgoing parameters.

Specifically, the execution time of Add\_To\_Map(*m*,*i*,*t*) is  $\Theta(\text{RANK\_IN\_SET}(i, \#m) + |t|)$ , the first term arising from the code that walks down the list to find the position at which to insert the pair (*i*,*t*), and the second term arising from the code that copies the string *t* before inserting it into the list. (Copying the fixed-length Integer *i* takes constant time, and hence does not contribute to the  $\Theta$ -notation expression.) Similar expressions describe the execution times of the other operations. But while only Add\_To\_Map requires copying a Text value, now the two Remove operations require reclaiming the storage for the replaced values of their incoming Text parameters. Remove\_Any\_From\_Map(*m*,*i*,*t*) involves no walk down the list because the first list entry can be removed and returned, and the operation Size(*m*) is also a bit simpler because its execution time is constant if the implementation explicitly keeps and incrementally updates the size of a Map.

The key point of this example implementation is that RANK\_IN\_SET(*i*,*m*), and hence all the durations that are described in terms of it, are expressible even though the model of *m*'s value in the functionality specification is evidently "more abstract" than the model of its representation.

### 3. THE LIMITED EXPRESSIVENESS PROBLEM

A problem in expressing performance arises because, in other cases, the original abstract model for a Map value is not adequate to explain the performance of an implementation of Map\_Facility. For an example, consider an unordered list representation. The execution time of Add\_To\_Map(*m*,*i*,*t*) is proportional to the length of *t*, because it must be copied. But the execution time of Remove\_From\_Map(*m*,*i*,*t*) now depends on the order of the elements in the list that represents *m*—and this cannot be expressed in terms of the abstract set model of *m* because the elements of a mathematical set are unordered.

```

performance profile for
realization Map_Using_Unordered_List
  of Map_Facility

type Map is augmented by
  ordered: string of integer
augmentation invariant
  |m.ordered| = |m| and
  for all i: integer
    (<i> is substring of m.ordered
     iff

```

```

  there exists
    s: string of character
    such that ((i,s) is in m))

definition POSITION_IN_LIST (
  i: integer,
  list: string of integer
): integer
implicit definition
  there exists a, b: string of integer
  such that
    (list = a * <i> * b and
     |a| = POSITION_IN_LIST(i,list))

operation Add_To_Map (
  updates m: Map,
  preserves i: Integer,
  preserves t: Text
)
duration
  theta (|t|)

operation Remove_From_Map (
  updates m: Map,
  preserves i: Integer,
  replaces t: Text
)
duration
  theta (POSITION_IN_LIST(i,m.ordered) +
        |t|)

operation Remove_Any_From_Map (
  updates m: Map,
  replaces i: Integer,
  replaces t: Text
)
duration
  theta (#t)

operation Map_Value (
  preserves m: Map,
  preserves i: Integer
): Text
duration
  theta (POSITION_IN_LIST(i,m.ordered) +
        |Map_Value|)

operation Is_Mapped (
  preserves m: Map,
  preserves i: Integer
): Boolean
duration
  theta (POSITION_IN_LIST(i,m.ordered))

operation Size (
  preserves m: Map
): Integer
duration
  theta (1)
end Map_Using_Unordered_List

```

Figure 3: Performance profile for another implementation of Map\_Facility

This requires us to introduce an *augmented data model* in order to explain the performance behavior of the implementation. An augmented data model is an addition to the model used in the functionality specification, which is intimately related to the functionality model but is “less abstract” in some sense. In this case, an appropriate augmentation is a string of integers, with the invariant property that the integers in the string are exactly the first components of the pairs in the set that was introduced to explain functionality. Figure 3 shows how the execution-time performance of this implementation might be expressed formally. Compare this with Figure 2, where no augmentation of the functionality model was required in order to express performance.

Unfortunately, even with the augmented model of a `Map` value in Figure 3, Figures 1 and 3 together still do not supply enough information to the client programmer to support compositional reasoning about performance. Why? The client of `Map_Using_Unordered_List` also must know what effect each of the operation implementations has on `m.ordered`. This effect cannot be deduced from the original functionality specification of Figure 1, which was developed independently of (and presumably before) Figure 3.

## 4. SOME IMPLICATIONS

We believe that the implications of these observations are important to researchers who tackle compositional reasoning about extra-functional properties such as performance. There is little reason to believe that performance is unique among extra-functional properties—unless it is in the sense that it is *easier* to define and to describe than most other such properties, e.g., “security”. The following specific implications are therefore worth noting:

- **Language design** — Compositional reasoning about performance requires a rich language in which to express the performance of an implementation. For example, in addition to having a functionality specification sublanguage such as that illustrated in Figure 1, there must be a performance specification sublanguage such as that illustrated in Figures 2 and 3. The performance specifier must be able to write arbitrary mathematical definitions, invariants, etc., just like the functionality specifier.
- **Mathematical modeling** — In some cases, the mathematical model that is most suitable for explaining the functionality of a component to a client programmer is not adequate to explain its performance. This is true simply because the functionality model might be “too abstract” to permit the performance of an implementation to be expressed. Hence, the performance description language must support the introduction of augmented data models through a mechanism such as that illustrated in Figure 3.
- **Extended functionality specifications** — In cases where an augmented data model is used, the functionality description also needs to be augmented with specifications of what the operations do to the augmented data model. Either the original functionality description language needs to have this capability, or the performance description language needs to have it.
- **Proof rules** — The proof rules of any such enriched specification/programming language must incorporate the

augmentation invariant. As noted above, the requirement for compositional reasoning about performance “bleeds back” into the functionality specification aspect. It therefore impacts proof-of-correctness of functionality by adding additional proof obligations involving the augmented data model, along with proof obligations for the new performance aspect.

One more observation is not supported directly by the examples in this paper, but is supported by other examples.

- **Performance specifications** — In general, even more detailed models are needed to express and reason about performance at higher levels of precision. Still, it is common for the augmented model to remain simpler than the complete representation model. Moreover, for clients of a component who wish to reason about performance, the effort required to understand an augmented model is generally far less than the effort required to understand the complete code of the implementation.

We hope that the component-based software community will take note of these issues as it moves forward in the direction of compositional reasoning about both functionality and extra-functional properties.

## 5. RELATED WORK

The importance of performance considerations in software engineering [1] and in component-based software construction [16] is well accepted. Formal techniques for static reasoning about performance have mostly concentrated on execution-time analysis. While earlier efforts in sequential program execution-time analysis concentrated on reasoning within a module [10], efforts that are more recent also have included timing deadline specifications [3, 7]. Other efforts have been devoted to real-time analysis in a concurrency setting [6, 8]. Hehner was among the first to consider formalization of memory space performance (including dynamic allocation) [4].

Most of this related work assumes that all programming objects have simple built-in types such as `Integer` in our example, or that non-trivial objects can be handled by considering only their sizes. Both of these simplifications hide the underlying expressiveness problem, as our example helps to illustrate. Note that the `Integer` variables (whose sizes are fixed) and the `Text` variables (whose sizes can be expressed in terms of the string model used to explain the functionality of the `Text` type) are not the source of the problem. It is the more complex user-defined `Map` type whose abstract model is perfect for explaining functionality but is sometimes inadequate for expressing performance. Other standard simplifications that are inherited from traditional algorithm analysis—such as the tendency to express performance as a function of a single “problem size” parameter—also make it impossible to express and reason about performance at the level of precision needed for component-based systems that involve non-trivial objects and data abstractions.

## 6. ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation under grants CCR-0081596 and CCR-0113181, and by Lucent Technologies. Any opinions, findings, and conclusions or recommendations expressed in this paper are

those of the author and do not necessarily reflect the views of the National Science Foundation or Lucent.

We are also grateful to the members of the Reusable Software Research Groups at Ohio State and Clemson for their valuable insights, and to the referees for pointing out parts of the paper that were particularly unclear.

## 7. REFERENCES

- [1] Cheng, A. M. K., Clemens, P., and Woodside, M., eds. Special section: Workshop on Software and Performance. *IEEE Trans. on Software Engineering* 26, 11/12 (November/December 2000).
- [2] Gasbichler, M., and Sperber, M. Final shift for call/cc:: direct implementation of shift and reset. In *Proc. 7<sup>th</sup> ACM SIGPLAN International Conference on Functional Programming*, ACM, 2002, 271-282.
- [3] Hayes, I.J., and Utting, M. A sequential real-time refinement calculus. *Acta Informatica* 37, (2001), 385-448.
- [4] Hehner, E.C.R. Formalization of time and space. *Formal Aspects of Computing*, (1999), 6-18.
- [5] Hollingsworth, J.E., Blankenship, L., and Weide, B.W. Experience report: using RESOLVE/C++ for commercial software. In *Proc. ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering*, ACM, 2000, 11-19.
- [6] Hooman, J., *Specification and Compositional Verification of Real-Time Systems*, LNCS 558, Springer-Verlag, 1991.
- [7] Liu, Y. A. and Gomez, G. Automatic accurate time-bound analysis for high-level languages. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LNCS 1474, Springer-Verlag, 1998.
- [8] Lynch, N. and Vaandrager, F. Forward and backward simulations, part II: timing-based systems. *Information and Computation* 121, 2 (1995), 214-233.
- [9] Musser, D.R., Derge, G.J., and Saini, A. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.
- [10] Shaw, A. C. Reasoning about time in higher-level language software. *IEEE Trans. Software Engineering* 15, 7 (1989), 875-889.
- [11] Sitaraman, M., and Weide, B.W. Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes* 19, 4 (1994), 21-67.
- [12] Sitaraman, M. On tight performance specification of object-oriented software components. In *Proc. of the 1994 International Conference on Software Reuse*, Ed. W. Frakes, IEEE Computer Society Press, 1994, 149-157.
- [13] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Pike, S., Heym, W.D., and Hollingsworth, J.E. Reasoning about software-component behavior. In Frakes, W.B., ed., *Software Reuse: Advances in Software Reusability (Proceedings Sixth International Conference on Software Reuse)*, Springer-Verlag LNCS 1844, 2000, 266-283.
- [14] Sitaraman, M. Compositional Performance Reasoning. In *Proc. Fourth ICSE Workshop on Component-Based Software Engineering: Component-Certification and System Prediction*, Toronto, CA, 2001.
- [15] Sitaraman, M., Kulczycki, G., Krone, J., Ogden, W.F., and Reddy, A.L.N. Performance specification of software components. In *Proc. SSR '01: 2001 Symposium on Software Reusability (ACM SIGSOFT Software Engineering Notes 26, 3)*, ACM, 2001, 3-10.
- [16] Szyperski, C., *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.
- [17] Voigtländer, J. Concatenate, reverse and map vanish for free. In *Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, ACM, 2002, 14-25.
- [18] Weide, B.W., Edwards, S.H., Heym, W.D., Long, T.J., and Ogden, W.F. Characterizing observability and controllability of software components. In M. Sitaraman, ed., *Proceedings 4th International Conference on Software Reuse*, IEEE, 1996, 62-71.
- [19] Wing, J.M. A specifier's introduction to formal methods. *IEEE Computer* 29, 9 (1990), 8-24.