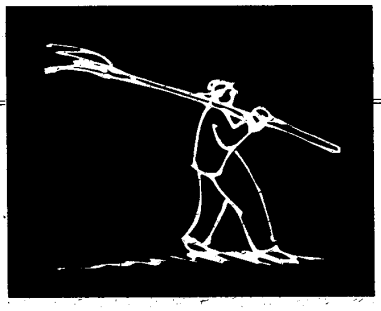


Recasting Algorithms to Encourage Reuse

BRUCE W. WEIDE and WILLIAM F. OGDEN,
Ohio State University
MURALI SITARAMAN, *West Virginia University*

◆ *Instead of viewing algorithms as single large operations, the authors use a machine-oriented view to show how they can be viewed as collections of smaller objects and operations. Their approach promises more flexibility, especially in making performance trade-offs, and encourages black-box reuse. They illustrate it with a sample design of a graph algorithm.*

All large software systems are built from components of some kind. A typical modern software component is a *module*, which usually encapsulates an abstract data type. The data type, in turn, hides the details of both concrete data structures and the algorithms that implement operations to manipulate the abstract data type's variables.

Reusable software components are just modules that have been carefully designed to be useful in several programs, even unanticipated ones.[1] We focus here on two types of flexibility — functional and performance — that make components reusable. We also advocate a systematic black-box style of reuse, in which designers use components without source-code modification. This contrasts to a haphazard opportunistic style in which designers scavenge old code for interesting tidbits to reshape.

We recommend black-box reuse because the real value of reused code lies in its properties, such as correctness with respect to an abstract specification. If you make even small structural or environmental changes, the confidence in these properties tends to evaporate, and with it most of the component's value.

In this article we show how to design an entire category of more flexible black-box reusable software components by applying a general design technique that "recasts" algorithms as objects. To illustrate the technique, we recast a sorting algorithm and a spanning-forest algorithm into objects.

0740-7459/94/$04.00 © 1994 IEEE

RECASTING FOR FLEXIBILITY

Conventional object-oriented design treats application-specific entities as objects and application-specific actions as operations on those objects. Many of these operations change the objects a great deal. Because they are implemented as single operations, they involve algorithms that manipulate complex data structures extensively.

The recasting technique we propose is a refinement of object-oriented design — it turns a single large-effect operation into an object by regarding it as a machine that performs the action. This effectively replaces one operation with an entire module. The module defines an abstract data type — which records the machine state — and several operations — each of which has a smaller effect. One of these smaller effect operations might supply input to the machine, for example; another might return results. This kind of design has greater *functional flexibility* — the component can be readily adapted to provide good solutions to any problem requiring its general services. A design that uses smaller effect operations does two things. First, it provides a finer grain of control. Second, it gives implementers the opportunity to offer more *performance flexibility* — they can substitute alternative implementations of an abstract component by making trade-offs among individual operations. This changes the component's performance characteristics but retains the same functional behavior.

Recasting works for two reasons:

- Component designers can organize data processing along one of two dimensions: The usual *object-structure* dimension relates items according to their explicit representation as data objects using arrays, records, lists, trees, and so on. Our recasting approach adds a *temporal* dimension, which relates items by the time they appear in a program.
- It takes advantage of the widely recognized fact that an abstract behavior specification does not prescribe *how* behavior is to be realized. In fact, module specification hides the knowledge of both how and *when* computations actually take place.

When you design a component to use large-effect operations, you are confining yourself primarily to the object-structure dimension. You miss the opportunity to use the temporal dimension as a data organizer and so preclude some potentially efficient implementations of the desired abstract behavior. Once you realize you can amortize the cost of an algorithm among several operations in the module and retain the same functionality, you gain tremendous flexibility. You can use precomputation, batch computation, deferred computation, and related data-structuring and algorithm-design techniques.[2] This gives various options to applications (like online and real-time systems) that demand that individual operations exhibit certain constrained performance profiles in addition to — or even instead of — optimal performance for an entire operation sequence.

THE MACHINE MODEL LETS DESIGNERS TUNE HOW A COMPONENT PERFORMS, NOT WHAT IT DOES.

SAMPLE DESIGN PROBLEM

To explore the nature and benefits of recasting single large-effect operations as objects, we present a traditional design problem and show how the usual design is flawed from the viewpoint of reusability. The design problem is to implement part of a circuit-layout tool: *Given an output terminal and a set of input terminals to which it must connect, determine how the terminals should be wired together in a net that minimizes total wire length.*

The key to attacking this problem is an abstract mathematical model. In this case, we can reuse well-developed ideas from the algorithms community: The required layout is a *minimum spanning tree* of an edge-weighted graph — a subset of the graph's edges that connect its vertices with a minimum total weight. The vertices are the terminals to be connected, and the edges are weighted by the lengths of wire required to connect corresponding terminal pairs.[2]

Whether you use a traditional functional design approach modified to embrace information-hiding principles[3] or a conventional object-oriented design approach, a typical solution might be

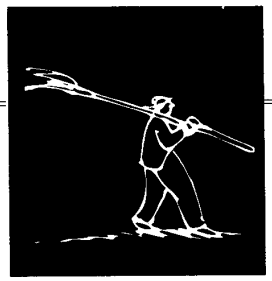
1. *Find the abstractions to be encapsulated in modules, identify their operations, and specify interface behavior.* Here you encapsulate the graph abstraction in a module and identify both operations sufficient for constructing a circuit model and implementing an operation, `Find_MST`, to compute the graph's minimum spanning tree.

2. *Implement the graph module and its associated operations.* You might use any of the many textbook graph representations.[2,4]

3. *Write a client program that uses the graph module and* `Find_MST` *to*

- *construct a graph* g *that models the portion of the circuit for which a net* t *is to be selected and*
- *find a subgraph* t *of* g *that is a minimum spanning tree of* g.

The graph module should be reusable in this and in other applications — if you carefully design it to be reused and not just to support finding minimum spanning trees. However, this design is fundamentally flawed from the standpoint of reusability and maintainability. The `Find_MST` operation is a single large-effect operation. It can be recast to make the task of finding a minimum spanning tree a separate object, offering the programmer who may want to reuse it a design that is more flexible.



Maintenance change. To illustrate, what happens when the users of this layout tool request "minor" changes after it is in the field? For example, suppose the total required wire length cannot exceed a certain bound or the output terminal's electrical features must be adjusted to handle a heavy load. This in turn might re-quire changing some terminal locations, until the net's total wire length is within the required bound. At that point, you can use the original net-selection operation to finish the job. Thus, you must now add the subtask

♦ *Determine if the total wire length of a net exceeds a given bound.*

This operation must be invoked repeatedly, with different graphs and bounds before a net can be selected; something the original code did in one invocation of the net-selection operation.

You can easily solve the bounds-checking problem by adding a step to the third part of the original solution:

♦ *Determine if the total edge weight of* t *exceeds the given bound.*

Unfortunately, this change causes users to complain of poor performance for some nets — and you find that changing the graph module or the `Find_MST` operation does not significantly improve the situation. How can you tune performance?

There is no easy solution to this problem because the decision to design `Find_MST` as a single operation has limited its functional and performance flexibility. Consequently, you must break into the `Find_MST` code to tune performance — eschewing black-box reuse and all its advantages.

Sorting algorithm. Suppose for the moment you are satisfied with the original net-selection program design. You might continue by refining the implementation of the `Find_MST` operation, which eventually should lead to something like textbook code.[2,4] Here's what might happen along the way.

CONVENTIONAL DESIGN FORCES DEVELOPERS TO 'PEEK UNDER THE COVERS' TO TUNE A COMPONENT.

First, you must choose a method for finding a graph's minimum spanning tree. The one we describe here is Kruskal's algorithm,[2, 4] a greedy algorithm that combines smaller trees joining subsets of vertices. This set of trees is called a *spanning forest.* To build it, the algorithm starts with an empty set of edges, T (a spanning forest in which each vertex is connected only to itself), looks at the edges of the graph, E, in nondecreasing order of edge weight, and adds a candidate edge to T if that edge does not form a cycle with those already in T. If the original graph is connected by E then T eventually contains a single tree, which is a minimum spanning tree. Otherwise, T eventually contains a minimum spanning forest of the original graph.

Because Kruskal's algorithm examines the edges in nondecreasing order of edge weight, it might be best to look at the problem in terms of sorting:

Given a list of items and some ordering relation on them, organize the list into nondecreasing order (where "smallest" describes the first item).

You might call a procedure from the body of `Find_MST`:

```
procedure Sort_List
  (e_list: edge_list)
```

to sort the edges in `e_list` (the edges of the graph) into nondecreasing order of edge weight.

Because Kruskal's algorithm can terminate when it discovers a minimum spanning tree, you might not have to examine many edges. This suggests a variant of sorting in which the problem is to enumerate the k smallest of n items in nondecreasing order. Unfortunately, in this case it is hard to capture this behavior in a single procedure because k is not predictable in advance — you don't know how many edges Kruskal's algorithm will need to examine before it terminates.

So you must separate the (partial) sorting problem into two phases:

1. Construct a data structure containing the set of edges that are to be examined in sorted order.

2. Incrementally deliver one edge at a time to Kruskal's algorithm, on demand, until it needs no more edges to form a minimum spanning tree.

In some textbook implementations of Kruskal's algorithm,[4] this is essentially how things work. Phase 1 consists of creating a heap data structure containing the edges, and phase 2 involves removing edges one at a time from the heap. The heap organization guarantees that the edges come out in nondecreasing order of edge weight.

This design makes it possible to have reasonable overall runtime for `Find_MST` because it lets you precompute a sorted order during phase 1, during the transition between phases 1 and 2, during phase 2, or during any of these, spreading the work around. Most important, it does not need to take as much time to get to phase 2 as it would if the algorithm sorted all the edges. So, if Kruskal's algorithm terminates before examining all the edges, the total time spent on the (partial) sorting can be substantially less than with the single Sort_List operation.

David Parnas' famous KWIC ("keyword in context") example notes the advantage of breaking up sorting into slightly smaller chunks of functionality.[3] However, to our knowledge this basic idea has neither been touted as being as general as it is nor been further developed and systematically applied to the design of reusable components. Twenty years after Parnas' paper, object-oriented component libraries still encapsulate data structures as objects and algorithms as single operations.

RECASTING SORTING

To solve the sorting variation in the `Find_MST` operation and produce a highly reusable software component, you must recast sorting as an object. Our recasting approach is based on a machine-oriented-design paradigm, in which you begin by viewing sorting as a machine that puts things of type `Item` into a sorted order. In this case, `Item` is a graph edge that you want to sort by the usual less-than-or-equal-to order on edge weights. But the module might as well be generic so it can be used with other `Item`s and other orderings.

Sorting machine data type. Imagine a sorting machine that accepts items to be sorted, one at a time, then dispenses items, one at a time, in sorted order. In many applications, you must insert all the Items before extracting the first one. There are two distinct phases: an insertion phase and an extraction phase.

Our encapsulation of a sorting machine into a module exports an abstract data type, `Sorting_Machine_State`, which records a machine state, and six operations. (Here, `m` is of type `Sorting_Machine_State` and `x` is of type `Item`).

- `Change_To_Insertion_Phase(m)`: Prepare `m` for calls to the `Insert` operation. This operation requires that `m` be in the extraction phase at the time of the call.
- `Insert(m,x)`: Insert `x` into `m`. This operation requires that `m` be in the insertion phase at the time of the call.
- `Change_To_Extraction_Phase(m)`: Prepare `m` for calls to the `Extract` operation. This operation requires that m be in the insertion phase at the time of the call.
- `Extract(m,x)`: Extract a smallest (remaining) `Item` from `m`, returning it in `x`. This operation requires that m be in the extraction phase at the time of the call.
- `Size(m)`: Return the number of `Items` in `m`.
- `Is_In_Insertion_Phase(m)`: Test if m is in the insertion phase.

Figure 1 shows the specification for this machine in Resolve.[5-7]

Intuitively, you may think of the collection of items in a sorting machine as a set, but this has two problems: First, sets have no duplicate elements, although you should be able to sort even with duplicate items. Second, sets have no intrinsic order among their elements. Using a multiset or bag (`INVENTORY_FUNCTION` in Figure 1) solves the first problem. You can address the ordering problem by specifying the `Extract` operation so that it selects, from among those items remaining, one that is smallest with respect to the desired ordering.

Functional flexibility. The `Sorting_Machine_Template` component is functionally more flexible than a single `Sort_List` operation. If you must sort all items in a collection and want a procedure like `Sort_List`, you can layer it on top of `Sorting_Machine_Template`. But if you must find only the *k* smallest items, or remove items until some condition is met, then you can stop after partial sorting.

This design has other advantages. For example, single large-effect operations such as `Sort_List` must operate on a particular data structure (it may be concrete or abstract, but it must be a particular kind in any case). In `Sort_List`, this structure is a list. If a program doesn't happen to have its data in list form, it must translate it into that form. `Sorting_Machine_Template` requires neither the source nor destination of the data to be a particular data structure or even the same kind of structure. For example, if you must get items from an input device and put them into a sorted list, you can easily layer code on `Sorting_Machine_Template` to do this.

IN OUR DESIGN, NEITHER THE DATA'S SOURCE NOR ITS DESTINATION MUST BE A CERTAIN DATA STRUCTURE.

Performance flexibility. The improved performance flexibility of `Sorting_Machine_Template` over `Sort_List` comes from recognizing a key point: The abstract specification of functionality is not a prescription for *how* data structures are represented or *when* sorting actually takes place. Of course you can achieve the specified behavior by representing `Sorting_Machine_State` as a list of items and a Boolean phase flag. And you can implement the `Insert` operation by adding a new `Item` anywhere in the list; the `Change_To_Extraction_Phase` operation by toggling the phase flag; and the `Extract` operation by searching for, removing, and returning the smallest `Item` in the list.

But there are many other implementation strategies with different performance profiles:

- During each call to `Insert`, maintain the list in sorted order.
- During `Change_To_Extraction_Phase`, sort the list explicitly using any sorting algorithm.
- Represent a `Sorting_Machine_State` using a binary search tree or a heap or any other data structure, in each case facing similar choices for what each operation should do to that structure. You can precompute to any extent during each `Insert` operation; batch process during `Change_To_Extraction_Phase`; defer work as long as possible until an `Extract` operation requires it; or amortize the effort among these operations in other ways.

A good choice for a minimum-spanning-tree application is to embed heapsort so that `Change_To_Extraction_Phase` creates a heap but does

not sort the items. The fact that sorting is a two-phase operation makes this implementation possible. And some secondary operations (like finding a smallest `Item` or a *k*th smallest `Item`) run faster with this implementation than with one that sorts everything in `Change_To_Extraction_Phase`.

RECASTING FIND_MST

Returning to the original net-selection problem, note the parallels between our variation of sorting and a minimum-spanning-tree algorithm. Once you realize that you may not have to sort all items, you can profitably recast sorting as an object. The same applies to obtaining edges. If you don't need to obtain all the edges of a

```
concept Sorting_Machine_Template

  context

    global context
      Standard_Boolean_Facility
      Standard_Integer_Facility

    parametric context
      type Item
      math operation ARE_ORDERED (
            x: math[Item]
            y: math[Item]
          ): boolean
      restriction (* ARE_ORDERED is a total
         pre-ordering *)

  local context
     math subtype INVENTORY_FUNCTION is
        function from math[Item] to integer
       exemplar   f
       constraint for all x: math[Item]
         (f(x) >= 0)
     math operation EMPTY_INVENTORY:
        INVENTORY_FUNCTION
       definition for all x: math[Item]
         (EMPTY_INVENTORY (x) = 0)
     math operation IS_FIRST (
           f: INVENTORY_FUNCTION
           x: math[Item]
         )
       definition f(x) > 0  and
         for all y: math[Item] where
          ARE_ORDERED (y, x)  and
          not ARE_ORDERED (x, y)
            (f(y) = 0)
  interface
   type Sorting_Machine_State is modeled by
        (
        count: INVENTORY_FUNCTION
        insertion_phase: boolean
       )
    exemplar  m
    initialization
      ensures m = (EMPTY_INVENTORY, true)

    operation Change_To_Insertion_Phase (
          alters      m:
Sorting_Machine_State
        )
      requires    not m.insertion_phase
      ensures     m = (EMPTY_INVENTORY, true)
    operation Insert (
          alters      m:
Sorting_Machine_State
          consumes    x: Item
        )
      requires    m.insertion_phase
      ensures     differ (m.count, #m.count,
                    {#x})  and
                  m.count(#x) = #m.count(#x)
                      + 1  and
                  m.insertion_phase
    operation Change_To_Extraction_Phase (
          alters      m: Sorting_Machine_State
        )
      requires    m.insertion_phase
      ensures     m = (#m.count, false)
    operation Extract (
          alters      m:
Sorting_Machine_State
          produces    x: Item
        )
      requires    m.count /= EMPTY_INVENTORY
                  and not m.insertion_phase
      ensures     IS_FIRST (#m.count, x)  and
                  differ (m.count, #m.count,
                    {x})  and
                  m.count(x) = #m.count(x)
                    - 1  and
                  not m.insertion_phase
    operation Size (
          preserves   m: Sorting_Machine_State
        ): Integer
      ensures     Size = sum x:math[Item]
                           (m.count(x))
    operation Is_In_Insertion_Phase (
         preserves      m:
Sorting_Machine_State
        ): Boolean
      ensures     Is_In_Insertion_Phase  iff
```

Figure 1. Specification of a sorting-machine concept.

```
concept Spanning_Forest_Machine_Template

 context

  global context

     Standard_Boolean_Facility
     Standard_Integer_Facility

  parametric context

     constant max_vertex: Integer
     restriction  max_vertex > 0

  local context

     math subtype EDGE is (
                  v1: integer
                  v2: integer
                  w: integer
        )
       exemplar   e
       constraint 1 <= e.v1 <= max_vertex and
                  1 <= e.v2 <= max_vertex and
                  e.w > 0
     math subtype GRAPH is set of EDGE

     math operation IS_MSF (
           msf: GRAPH
           g: GRAPH
      ): boolean
       definition (* true iff msf is an
         MSF of g *)
 interface
  type Spanning_Forest_Machine_State
    is modeled by (
        edges: GRAPH
        insertion_phase: boolean
     )
     exemplar     m
     initialization
       ensures    m = (empty_set, true)

 operation Change_To_Insertion_Phase (
       alters     m: Spanning_Forest_Machine_
                     State
     )
     requires   not m.insertion_phase
     ensures    m = (empty_set, true)

operation Insert (
     alters     m:Spanning_Forest_Machine_
                   State
     consumes   v1: Integer
     consumes   v2: Integer
     consumes   w: Integer
    )
  requires    m.insertion_phase  and
              1 <= v1 <= max_vertex  and
              1 <= v2 <= max_vertex  and
              w > 0
  ensures     IS_MSF (m.edges,
                #m.edges union
                  {(#v1, #v2, #w)})  and
              m.insertion_phase

operation Change_To_Extraction_Phase (
     alters        m:Spanning_Forest_
                     Machine_State
    )
  requires     m.insertion_phase
  ensures      m = (#m.edges, false)

operation Extract (
     alters        m:Spanning_Forest_
                     Machine_State
     produces      v1: Integer
     produces      v2: Integer
     produces      w: Integer
    )
  requires      m.edges /= empty_set  and
                not m.insertion_phase
  ensures       (v1, v2, w) is in
                  #m.edges  and
                m = (#m.edges without
                  {(v1, v2, w)}, false)

operation Size (
     preserves     m:Spanning_Forest_
                     Machine_State
    ): Integer
  ensures        Size = m.edges

operation Is_In_Insertion_Phase (
   preserves
m:Spanning_Forest_
                     Machine_State
    ): Boolean
  ensures  Is_In_Insertion_Phase  iff
             m.insertion_phase
end Spanning_Forest_Machine_Template
```

Figure 2. Specification of a spanning-forest-machine concept.

minimum spanning tree, you can profitably recast this operation as an object.

Two phases. Imagine a spanning-forest machine that accepts weighted edges of a graph, one at a time, then dispenses the edges of a minimum spanning forest, one at a time. (We call this a spanning-forest machine, not a spanning-tree machine, because the original graph might not be connected by its edges. In the net-selection application the graph presumably is connected and everything will work fine. But the specification is easier, implementations are essentially the same, and the component is more reusable if the machine can find the minimum spanning forests of unconnected graphs, too.)

Should a spanning-forest machine have two phases? The same factors that influenced the design of Sorting_Machine_Template suggest that it should have. But there is also another reason. Given the way a minimum spanning forest is defined, it doesn't make much sense to ask for the next edge in a graph that is changing as you extract its edges. Edges previously extracted could be made erroneous as new edges are inserted. This additional reason supports the logic behind making this a two-phase machine.

How you define a spanning-forest machine is important. It is best to explain it as an "organizer" machine: The Insert operation promises to

```
realization Kruskal_Amortized
  for Spanning_Forest_Machine_Template
  context
    global context
      ...
    parametric context
      ...
    local context
      type Edge is record
          vertex1: Integer
          vertex2: Integer
          weight: Integer
        end record
      facility Sorting_Machine_Facility is
          Sorting_Machine_Template
            (Edge, EDGES_ARE_ORDERED)
        realized by Heapsort_Embedding (...)
      facility Coalesceable_Equivalence_
        Relation_Facility is
          Coalesceable_Equivalence_
          Relation_Template (max_vertex)
        realized by Disjoint_Set (...)
      type Spanning_Forest_Machine_
      State_Rep is record
          graph_edges: Sorting_Machine_State
          are_connected: Coalesceable_
            Equivalence_Relation
          num_spanning_edges: Integer
      end record
      ...
  interface
    type  Spanning_Forest_Machine_State
        is represented by
          Spanning_Forest_Machine_State_Rep
      convention          (* rep invariant *)
      correspondence      (* representation-
        abstraction relation *)

    operation Change_To_Insertion_Phase (
          alters      m: Spanning_Forest_
                         Machine_State
        )
        new_rep: Spanning_Forest_Machine_
                 State_Rep
      begin
       m.rep :=: new_rep
      end Change_To_Insertion_Phase

    operation Insert (
          alters      m: Spanning_Forest_
                         Machine_State
          consumesv1: Integer
          consumesv2: Integer
          consumesw: Integer
        )
      begin
        if not Are_Equivalent
            (m.rep.are_connected, v1, v2)
          then
            Make_Equivalent
              (m.rep.are_connected, v1, v2)
            m.rep.num_spanning_edges :=
              m.rep.num_spanning_edges + 1
        end if
        Insert (m.rep.graph_edges, (v1, v2, w))
      end Insert

    operation Change_To_Extraction_Phase (
          alters      m: Spanning_Forest_
                         Machine_State
        )
        new_equivalence_relation:
          Coalesceable_Equivalence_Relation
      begin
        Change_To_Extraction_Phase
          (m.rep.graph_edges)
        m.rep.are_connected :=:
          new_equivalence_relation
      end Change_To_Extraction_Phase

    operation Extract (
          alters      m: Spanning_Forest_
                         Machine_State
          produces    v1: Integer
          produces    v2: Integer
          produces    w: Integer
      )
      begin
        loop
          maintaining (* loop invariant *)
          Extract (m.rep.graph_edges,
            (v1,v2,w))
          if not Are_Equivalent
              m.rep.are_connected,v1,v2)
            then
              Make_Equivalent
                (m.rep.are_connected, v1, v2)
              m.rep.num_spanning_edges :=
                m.rep.num_spanning_edges - 1
              exit
          end if
        end loop
      end Extract

      operation Size (
          preserves   m: Spanning_Forest_
                         Machine_State
        ): Integer
      begin
        return m.rep.num_spanning_edges
      end Size

      operation Is_In_Insertion_Phase (
          preserves   m: Spanning_Forest_
                         Machine_State
        ): Boolean
      begin
        return Is_In_Insertion_Phase
                 m.rep.inserting
      end Is_In_Insertion_Phase
  end Spanning_Forest_Machine_Template
```

Figure 3. An amortized-cost implementation of Spanning_Forest_Machine_Template.

keep only edges that are part of a minimum spanning forest, and the Extract operation simply removes and returns some remaining ones, as Figure 2 shows.

You can use the component in Figure 2 to solve the original net-selection problem with one Spanning_Forest_Machine_State, m:

```
while some edge of g is not
   yet inserted into m do
  let (v1, v2, w) be any edge
   of g not yet inserted
   into m
  Insert (m, v1, v2, w)
end while
Change_To_Extraction_Phase (m)
while Size (m) > 0 do
  Extract (m, v1, v2, w)
  record/report that (v1, v2,
   w) is an edge of t
end while
```

An interesting feature of this code is that you can easily change the second loop to find if a net's total wire length exceeds a given bound:

```
total_weight := 0
while (Size (m) > 0 and
   total_weight <= bound) do
  Extract (m, v1, v2, w)
  total_weight :=
   total_weight + w
end while
exceeds_bound :=
  (total_weight > bound)
```

This change by itself is not particularly easier or harder to make than for the original design. But other ramifications of the new design are significant. Now you can tune performance without "peeking under the covers" into Kruskal's algorithm. All you need to do is select an implementation that amortizes the costs of Spanning_Forest_Machine_Template so that the Insert and Change_To_Extraction_Phase operations don't actually compute a minimum spanning forest — it's almost all done in the Extract operation.

Amortizing costs. Model-based formal specifications do not favor any particular implementation and certainly do not limit you to a single implementation. So, despite this specification's claim that Insert keeps only minimum-spanning-forest edges, you are free to amortize the cost of finding a minimum spanning forest among Insert, Change_To_Extraction_Phase, and Extract in any way that makes sense.

MACHINE-ORIENTED DESIGN IS EASIER FOR CLIENTS TO UNDERSTAND.

For example, you could choose to do all the interesting work during Change_To_Extraction_Phase. To do so, build a graph from the inserted edges, use Kruskal's algorithm to find a minimum spanning forest, and save the spanning-forest edges in a list (for example) from which they can be dispensed during subsequent Extract operations. This gives the same performance as the original solution, and it also means that you pay for finding a minimum spanning forest even if you don't need to extract all the edges — precisely the performance problem raised by the maintenance example.

Instead, your implementation could defer computation until the Extract operation, as Figure 3 shows. Represent a Spanning_Forest_Machine_State (in part) with a Sorting_Machine_State whose Item type is a record that contains two vertices and a weight for a single edge. The EDGES_ARE_ORDERED relation is less-than-or-equal-to on the weight field. Then call the Insert operation on the Sorting_Machine_State to add the new edge and call the Change_To_Extraction_Phase operation on the Sorting_Machine_State to change the phase of the Spanning_Forest_Machine_State. Finally, keep calling the Extract operation on the Sorting_Machine_State to get the smallest remaining edge until you find one that doesn't form a cycle with the previously extracted edges.

Besides its use of amortization, the code in Figure 3 is subtle in another respect. It includes a Size operation, which keeps a count of spanning-forest edges without knowing which edges are involved. This means you do not have to compute the minimum spanning forest when Size is first called, which would have unfortunate performance consequences!

To analyze the performance of Figure 3, let n = the number of edges in m. Insert (m,...) takes O(1)time,and Change_To_Extraction_Phase (m) takes $O(n)$ time. Extract (m,...) might take only O(log n) time if the smallest remaining edge is an edge of a minimum spanning forest. If not, it might take much longer, but not more than $O(n \log n)$ time.

On any given graph, both the original implementation of the Find_MST operation and its implementation layered on top of this realization of Spanning_Forest_Machine_Template use $O(n \log n)$ time in the worst case. In summary, there is no difference in performance from the original net-selection problem. However, our recasting design has a potentially significant performance advantage over the conventional design for the bounds-checking problem.

(Incidentally, Figure 3 also uses a Coalesceable_Equivalence_Relation type to solve the cycle-detection problem, and you would want to use it in Find_MST even if you settled for the original design. There are operations to make two integers equivalent and to test if two integers in a Coalesceable_Equivalence_Relation are equivalent, but space prevents us from showing the formal specification for this component here. Suffice to say that an efficient representation of Coalesceable_Equivalence_Relation uses the textbook disjoint-set data structure with path-compression.[2,4])

Conventional reusable component design techniques — even ones based on object-oriented principles —result in components that encapsulate data structures as objects and algorithms as single operations. Separating data structures and algorithms for this purpose is a false dichotomy. Algorithms can and should be encapsulated as objects, just as data structures are. By following machine-oriented design principles, you can achieve more of the functional and performance flexibility potential of systematic component reuse. You also can make your designs consistent and therefore easier for clients to understand.

In principle, there are no limits in applying this approach. For example, you could specify a "record-high" machine that reports each largest item so far; an "eigenvalue" machine that dispenses eigenvalues of a matrix in increasing order;[8] a "compression" or "encryption" machine that works on a series of items.

There are several points to consider when you recast a single large-effect operation as an object. First, try to develop a simple, fully abstract, clearly explainable mathematical model for the collection of items in the machine.[6,7] Then consider if you can settle for a two-phase machine. You probably should have a two-phase version of every machine in the reusable component library even if you can't see the immediate need for it in a particular application. Often, implementations for two-phase machines are easier and/or potentially more efficient than for multiphase or phase-less machines.

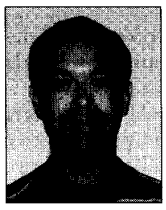
Finally, consider which explanation style you should use to specify the machine's overall behavior by characterizing what it apparently does during the insert, change-to-extraction, and extract operations. What and when your machine does something will depend, in part, on which explanation is most understandable. You might just have to use trial and error before you can judge which is best. But don't worry too much about the initial cost of making these design decisions! If your component is really reusable, the effort you spend on making a good design choice will be amortized over many future uses. ◆

ACKNOWLEDGMENTS

We thank Steve Edwards, Wayne Heym, Joe Hollingsworth, Tim Long, Stu Zweben, and the anonymous *IEEE Software* referees for many helpful suggestions. We also acknowledge the financial support for our research from the National Science Foundation (Weide and Ogden are supported under grants CCR-9111892 and CCR-9311702; Sitaraman is supported under grant CCR-9204461); the Department of Defense's Advanced Research Projects Agency (Weide and Ogden are supported under contract F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order A714; Sitaraman is supported under ARPA contract DAAH04-94-G-0002, monitored by the US Army Research Office); and the National Aeronautics and Space Administration (Sitaraman is supported under grant 7629/229/0824).

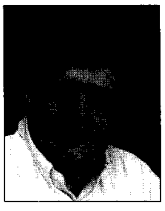
REFERENCES

1. B.W. Weide, W.F. Ogden, and S.H. Zweben, "Reusable Software Components," *Advances in Computers Vol. 33*, M.C. Yovits, ed., Academic Press, New York, 1991, pp. 1-65.
2. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 1990.
3. D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems Into Modules," *Comm. ACM*, Dec. 1972, pp. 1,053-1,058.
4. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Rockville, Md., 1976.
5. M. Sitaraman, L.R. Welch, and D.E. Harms, "On Specification of Reusable Software Components," *Int'l J. of Software Eng. and Knowledge Eng.* June 1993, pp. 207-219.
6. B.W. Weide et al., "Design and Specification of Iterators Using the Swapping Paradigm," *IEEE Trans. Software Eng.*, Aug. 1994.
7. Special feature on "Component-Based Software Using Resolve," *SIGSoft Software Eng. Notes*, Oct. 1994, to appear.
8. E.R. Davidson, "Monster Matrices: Their Eigenvalues and Eigenvectors," *Computers in Physics*, Sept./Oct. 1993, pp. 519-522.



Bruce W. Weide is an associate professor of computer and information science at Ohio State University, and codirector of the Reusable Software Research Group with Bill Ogden and Stu Zweben. His research interests include all aspects of software component engineering, especially in applying RSRG work to Ada and C++ practice.

Weide received a BS in electrical engineering from the University of Toledo and a PhD in computer science from Carnegie Mellon University. He is a member of the IEEE, ACM, and Computer Professionals for Social Responsibility.



William F. Ogden is an associate professor of computer and information science at Ohio State University, and codirector of the Reusable Software Research Group with Bruce Weide and Stu Zweben. His main research interests are in software reuse, software specification, and program verification.

Ogden received a BS in mathematics from the University of Arkansas and an MS and a PhD in mathematics from Stanford University. He is a member of the IEEE Computer Society and ACM.

Murali Sitaraman is on the faculty of statistics and computer science at West Virginia University. His research interests span all areas of software component construction including design, formal specification, and verification.

Sitaraman received an ME in computer science from the Indian Institute of Technology at Bangalore and a PhD in computer and information science from Ohio State University.

Address questions about this article to Weide at Ohio State University, Computer and Information Science Dept., Columbus, OH 43210; weide@cis.ohio-state.edu.