On the Practical Need for Abstraction Relations to Verify Abstract Data Type Representations

Murali Sitaraman, *Member*, *IEEE Computer Society*, Bruce W. Weide, *Member*, *IEEE*, and William F. Ogden, *Member*, *IEEE Computer Society*

Abstract—The typical correspondence between a concrete representation and an abstract conceptual value of an abstract data type (ADT) variable (object) is a many-to-one function. For example, many different pointer aggregates give rise to exactly the same binary tree. The theoretical possibility that this correspondence generally should be relational has long been recognized. By using a nontrivial ADT for handling an optimization problem, we show why the need for generalizing from functions to relations arises naturally in practice. Making this generalization is among the steps essential for enhancing the practical applicability of formal reasoning methods to industrial-strength software systems.

Index Terms—Abstract data type, abstraction function, abstraction mapping, abstraction relation, data abstraction, formal specification, greedy algorithm, program verification, nondeterminism, optimization problem, relation.

1 Introduction

The need to separate the specifications and implementations of abstract data types is widely recognized. To keep a specification purely conceptual and unbiased with respect to its many alternative implementations, the behavioral explanation should employ an implementation-neutral abstract model rather than any particular representation model. The formal verification that a given implementation does meet this conceptual specification then involves a correspondence mapping, traditionally called an *abstraction function*, between the model used in the implementation (the concrete or representation model) and the model used in the specification (the abstract or conceptual model) [10].

For some ADT specifications and implementations, the natural connection between concrete and abstract models turns out to be relational, not functional. That is, in some cases a particular concrete value may represent any of several abstract values; see Fig. 1.

The theoretical importance of abstraction relations has long been recognized. Precluding their expression results in modular verification systems which are incomplete in the technical sense that there are implementations that are correct with respect to their specifications, but which cannot be proved to be so using only abstraction functions. Moreover, insisting upon using an abstraction function even when it is technically possible may increase verification complexity to the point where it effectively thwarts modular reasoning about correctness. And it is crucial for tractability and reuse that the verification of an ADT's implementation code should be modular. This means that the proof of correctness should rely only on the given specification of behavior to be implemented and on given specifications of lower-level components that are used in the code. The correctness argument should be independent of the implementations of the lower-level components and independent of other parts of the system that use the code being verified [7], [23].

Here, we formally establish the requirement for supporting abstraction relations by exhibiting a nontrivial ADT for a practical optimization problem, where not just the value of—but the outright need for—an abstraction relation naturally arises. The nature of the example argues that formal reasoning systems must be able to generalize to handle abstraction relations if they are to be applied with confidence to new and nontrivial data abstractions.

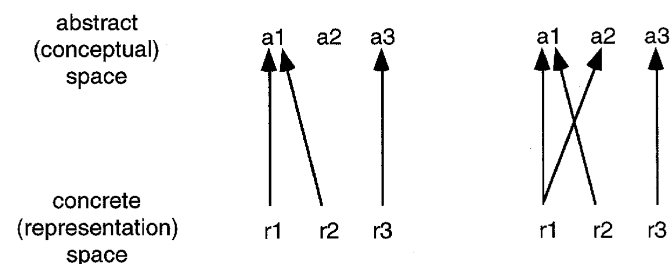

Fig. 1. Abstraction function (left) and abstraction relation (right).

1.1 Prior Work on Abstraction Relations

Previous work involving modular verification of ADTs with model-based specifications leaves the practical role of abstraction relations unsettled. Leavens notes the value of "simulation relations" (essentially abstraction relations) in defining behavioral subtyping [13]. Jones [11, p. 219] and Schoett [19] independently observe that, technically, ab-

- *M. Sitaraman is with the Department of Statistics and Computer Science, West Virginia University, Morgantown, WV 26506. E-mail: murali@cs.wvu.edu.*
- *B.W. Weide and W.F. Ogden are with the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210. E-mail: {weide, ogden}@cis.ohio-state.edu.*

Manuscript received Sept. 18, 1995; revised Feb. 14, 1997.
Recommended for acceptance by J.D. Gannon.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S95823.

0098-5589/97/$10.00 © 1997 IEEE

straction relations might be needed in some cases to verify implementations of ADTs whose specifications are "biased," or "not fully abstract". Schoett's work is based on the assumption that nonfully abstract specifications might arise in practice. But Jones notes that [11, p. 182], "If different abstract values correspond to one concrete value, it is intuitively obvious that such values could have been merged in the abstraction. So, in the situation where the objects used in the specification were abstract enough, the many-to-one situation would not arise."

There also has been some work on abstraction relations in the context of algebraic specifications. For example, to show the theoretical need for abstraction relations, Nipkow describes a construction involving algebras of nondeterministic data types [18]. The relationship between this work and the practical need for abstraction relations to verify implementations of model-based specifications is unclear, however. So while there are subtle differences in the positions taken by different authors on the topic, the comments of Liskov and Wing in their corrigendum to an earlier paper [14] probably best characterize the common belief among software engineers who use formal methods: Abstraction relations are occasionally helpful and might even be technically necessary in some cases; however, [15, p. 4] "for most practical purposes, abstraction functions are adequate (compared to relations)."

1.2 Contributions

We show that abstraction relations are practically important for software specification and modular verification. Technically, abstraction relations are necessary in order to avoid incompleteness. Practically, they are necessary in order to deal with new and nontrivial ADTs such as those resulting from modern software component design techniques.

We use a sample specification based on the technique of "recasting" algorithms as data abstractions [24]. This software component, if it is to lend support to the claim for practical significance of abstraction relations, must have three properties:

1) *Realism.* The specification must not be artificial and conceived just for showing the need, i.e., it must be of a sort that is actually likely to arise in practical systems. Otherwise, the fact that a reasoning system based solely on abstraction functions cannot handle it would have little practical import. Our sample specification captures solutions to a practical optimization problem and serves as an exemplar for a larger class of similar components.
2) *Quality.* The sample specification must be well-designed. In particular, it must be fully abstract; i.e., every two different conceptual values of the abstract data type being defined must be computationally distinguishable [11], [12], [25]. Otherwise, the relational nature of the correspondence mapping could merely arise from the sloppiness of the conceptual specification. Our sample component is a well-designed, fully abstract specification.
3) *Provable resistance to verification with abstraction functions.* There must be an actual proof that shows why no abstraction function can be found to verify that a correct implementation satisfies the sample specification. Our sample component comes with a correct and practical realization that we prove cannot be verified using any abstraction function (but which can be verified using an abstraction relation).

2 Inherently Relational Behavior Specifications

Optimization problems are a general category of problems in which relational specifications arise naturally. In many such problems, it is easy to find multiple solutions which satisfy the constraints yet which all evaluate to the same objective function value. The specification for software to solve such a problem is inherently relational because it should allow an implementation to produce *any* optimum solution. The natural correspondence between such implementations and specifications tends to be relational (even though a functional correspondence might exist in some cases).

2.1 A Realistic Software Component Example

As a sample relational problem specification we use the `Spanning_Forest_Machine_Template` from our recent paper on "recasting" algorithms as objects [24]. This specification exhibits the relational behavior we seek because it requires that *some* minimum spanning forest (MSF) of a given graph must be found; there might be ties and any best answer is acceptable. For a fully connected graph an MSF is also a minimum spanning tree (MST). For a general unconnected graph, an MSF is a union of edges of MSTs for each of the connected components [4].

The concept for `Spanning_Forest_Machine_Template` defines a type `Spanning_Forest_Machine` (a variable of which type we henceforth call a "machine" for brevity) and suitable operations. A typical client repeatedly calls operation `Insert` to add the edges of the graph for which an MSF is to be found (one at a time) into a machine; calls `Change_To_Extraction_Phase` to change the machine to extraction phase, and finally makes multiple calls to `Extract` to remove, one at a time, the edges of one of the (possibly many) MSFs of that graph. Operation `Insert` requires that the machine be in the insertion phase at the time of the call, whereas `Change_To_Extraction_Phase` and `Extract` operations require that the machine be in the extraction phase. `Is_In_Insertion_Phase` tests whether a machine is in insertion phase. `Size` returns the number of MSF edges in the graph and is restricted to be called only when in the extraction phase (for purposes of simplicity in this paper).

The concept described informally above, and specified formally in Fig. 2, is quite different from one providing a single procedure that finds an MSF of a graph. Our component prescribes what computation needs to take place, but *not when.* Viewed through its abstract interface, the component does not reveal to a user when (i.e., in which operation or operations) an MSF is actually being computed. The design gives the implementer freedom both in how and in when to do computations, and the attendant performance flexibility of various kinds of cost amortization, which is part of the rationale for the recasting technique illustrated by this interface [24]. This observation reinforces an important principle

```
concept Spanning_Forest_Machine_Template

   context
      global context
         facility Standard_Boolean_Facility
         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 finite 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 is modeled by (
                 edges: GRAPH
                 insertion_phase: boolean
              )
         exemplar m
         constraint IS_MSF (m.edges, m.edges)
         initialization ensures
            m = (empty_set, true)

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

      operation Insert (
              alters      m: Spanning_Forest_Machine
              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
           )
         requires
           m.insertion_phase
```

```
        ensures
          m = (#m.edges, false)

    operation Extract (
            alters        m: Spanning_Forest_Machine
            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 - {(v1, v2, w)}, false)

    operation Size (
            preserves  m: Spanning_Forest_Machine
          ): Integer
        requires
          not m.insertion_phase
        ensures
          Size = |m.edges|

    operation Is_In_Insertion_Phase (
            preserves  m: Spanning_Forest_Machine
          ): Boolean
        ensures
          Is_In_Insertion_Phase = m.insertion_phase

end Spanning_Forest_Machine_Template
```

Fig. 2. Specification of `Spanning_Forest_Machine_Template`.

for implementers of model-based specifications: They must always distinguish abstract models from concrete representations and must not let the abstract view bias how or when to manipulate the concrete representation.

2.2 Recasting and Abstraction Relations

We might have used any of a number of recasting examples in this paper. When optimization algorithms, such as those for finding MSFs as well as others such as those for finding single-source shortest paths, are recast as data abstractions, abstraction relations arise naturally in verifying some of their implementations. To see this general need for an entire class of situations similar to the one used in our sample, consider the relational specification of any graph optimization problem where the output is specified to be any one of many possible optimum values. Assume that the specification delineates two distinct phases as in the case of `Spanning_Forest_Machine_Template`: an insertion phase in which edges of a graph can be inserted and an extraction phase in which an optimum answer (say, a set of edges) can be extracted one at a time.

A straightforward model of the ADT defined by the above specification might be an ordered triple: a boolean *phase* that indicates the phase of machine *m*, an *input* set of edges that captures the graph edges inserted into *m*, and an *output* set of edges that defines an optimum solution. Initially, *phase* indicates insertion phase, and *input* and *output* are empty sets. The specification of the `Insert` operation changes only *input* as it adds a new edge. The postcondition of `Change_To_Extraction_Phase` is relational and dictates merely that *output* should become an optimum solution for *input*. The `Extract` operation is specified to return one of the remaining edges of *output*. In this specification, then, it appears that a solution is computed in "batch" fashion when `Change_To_Extraction_Phase` is called.

But other implementations might be possible and reasonable. Consider an amortized cost implementation that accumulates graph edges during the insertion phase but does no special computation in the `Insert` or `Change_To_Extraction_Phase` operations; it computes and returns each edge of an optimum solution only incrementally whenever an `Extract` operation is called. For example, this is how any "greedy" algorithm might be naturally amortized. In the extraction phase, the natural correspondence between the internal representation and the abstract model is relational. It is of the general form:

*IS_AN_OPTIMUM_SOLUTION (m.output, S(m.**rep**))*

where *S* is a function from the specific representation of *m* to the mathematical set of edges not yet processed. While there might exist abstraction functions for some implementations such as outlined here, since abstraction relations introduce no significant additional complexity to verification and may actually simplify the conditions—as argued later in this paper—a practical formal system should facilitate the use of abstraction relations in cases like this where they are natural.

The `Spanning_Forest_Machine_Template` can be specified in ways other than the one outlined above [22]. But regardless of how the concept is specified, abstrac-

tion relations arise because the specification needs to capture and allow only MSFs of the input graphs, whereas some implementations might not compute an MSF when the specification suggests. Such situations are typical when the recasting technique is employed.

3 The Practical Need for Abstraction Relations

Section 2 addressed the realistic nature of our sample component. Now we introduce its formal specification; demonstrate its quality in the sense that this specification is fully abstract and, therefore, not defective in the sense discussed in Section 1.1 [11]; and finally prove that there are practical and correct implementations of this component that cannot be verified using any abstraction function.

3.1 A Formal Specification of `Spanning_Forest_Machine_Template`

Fig. 2 is a reproduction of the `Spanning_Forest_Machine_Template` specification from [24] as expressed in the model-based specification language RESOLVE [21].[1] The specification language does not affect the issues discussed in this paper. Any model-based formal specification language [26] would suffice.

To specify the behavior of the operations described informally in Section 2, we model a value of type `Spanning_Forest_Machine` as an ordered pair: a weighted graph *edges* which is treated as a finite set of positively-weighted edges, and a boolean flag *insertion_phase* which is true iff the machine is in the insertion phase. The specification defines and uses a mathematical predicate *IS_MSF*(*msf, g*) which is true iff the graph *msf* is a minimum spanning forest of the graph *g*. The details of this definition are elided in Fig. 2 but they are straightforward.

From the specification it *appears* that a machine in insertion phase retains only an MSF for the edges inserted so far—not the entire set of edges inserted so far—thus giving an external observer the impression that an MSF is kept incrementally all along. But, as noted earlier, because a client of the component cannot see the representation, an implementation actually might keep all the inserted edges until `Change_To_Extraction_Phase` is called and then batch-process them to weed out nonMSF edges; or it might use an amortized cost implementation.

The specification in Fig. 2 raises an important question: Does it really allow an implementation to produce during the extraction phase *any* MSF of the inserted edges, or does the specified incremental nature of the `Insert` operation rule out some possible MSFs? It turns out that the specification is not restrictive, a fact that follows directly from a lemma from graph theory about MSF properties:

$$\forall G_1, G_2{:}\,GRAPH,\ e{:}\ EDGE$$
$$(IS_MSF(G_1, G_2 \cup \{e\}) \Rightarrow$$
$$\exists G_3{:}\ GRAPH(IS_MSF(G_3, G_2) \wedge IS_MSF(G_1, G_3 \cup \{e\})))$$

1. A summary of RESOLVE specificaiton notations essential for understanding this paper is given in Appendix A. There are a few minor changes in this specification from the one in [24] to reflect current RESOLVE syntax. The one substantive change is that the `Size` operation here has a precondition; it cannot be called during the insertion phase (but there is no reason to do so in any case). This change permits a simplified presentation in Section 3 but does not materially affect any of the issues we raise.

A proof of the lemma involves standard arguments from graph theory, where a case analysis based on whether e is in G_1 yields a construction for G_3. The proof of correctness of a batch-style implementation of `Spanning_Forest_Machine_Template` (see Appendix B) explains the relevance of this lemma and the conclusion that it demonstrates why the specification in Fig. 2 is not restrictive.

We conclude this section by noting that the specification in Fig. 2 is fully abstract, i.e., it is both "observable" and "controllable" [25]. To be observable (hence fully abstract), the specification must make it possible to distinguish every two abstract model values through the provided operations [11], [12]. We specify that a machine keeps only MSF edges at all times; clearly any two different MSFs can be distinguished by a sequence of calls to `Extract` operations. To be controllable, the specification must make it possible to construct every abstract model value. This also is permitted because any particular MSF can be constructed through an appropriate sequence of calls to `Insert` with just that MSF's edges.

3.2 A Class of Implementations that Need Abstraction Relations

To prove the practical need for abstraction relations, it remains to show the existence of a valid and practical implementation of this specification that cannot be proved correct using any abstraction function, but which can be verified using an abstraction relation. The argument is organized as follows. First we characterize a set of valid "nonmonotonic, deterministic, batch-style" implementations, any of whose members could serve as this unverifiable implementation. Next we show why these implementations cannot be proved correct using *any* abstraction function. In the last subsection, we explain how abstraction relations can be used to verify these implementations in a modular proof system [7].

In this discussion, it is important to note that the specificity of the particular class of implementations to be considered arises only because we seek to show the resistance to verification using abstraction functions, with minimal "hand waving." Other than this there is nothing special about the class of implementations considered. Amortized cost implementations, for example, would have served equally well.

3.2.1 Deterministic Batch-Style Implementations

Let ***B*** be the class of valid deterministic batch-style implementations of `Spanning_Forest_Machine_Template`. The implementations in ***B*** are, first, ***deterministic***: the outputs computed by each operation are entirely determined by its inputs. Two abstract operations (`Insert` and `Extract`) have behavioral specifications that are relational, but their implementations may have deterministic functional behavior. In order to be valid, an implementation need only exhibit a behavior pattern that is consistent with the specified relation; it is not necessary for the implementation actually or even potentially to give different results when run multiple times with the same inputs. We restrict our attention to deterministic implementations both because deterministic behavior for an

implementation is a typical situation in practice, and because this determinism simplifies the proof that abstraction relations are required for verification.

The implementations in ***B*** also are *batch-style*. This means they just store all the inserted graph edges while a machine is in insertion phase, deferring computation of a minimum spanning forest to the start of the extraction phase. So initially, the edge collection representing a machine state is empty.[2] The `Insert` operation adds a new edge to it. `Change_To_Extraction_Phase` computes a minimum spanning forest of the edge collection (e.g., using Kruskal's algorithm) and stores only the resulting MSF edges back into the edge collection. The `Extract` operation simply removes and returns any edge from the edge collection. The `Size` operation returns the number of edges in that collection, and `Change_To_Insertion_Phase` empties it.

Why are such batch-style implementations correct, i.e., why should we consider them to behave as specified? This question about correctness has to do with the timing of events: Is it possible for a client of `Spanning_Forest_Machine_Template` to detect that a batch-style implementation is being used, as opposed to an "eager beaver" implementation that seems natural from the specification? The behavior of any implementation of any abstraction can be detected only through the "observer" operations provided in its interface, in this case `Extract` and `Size`. It is clear that the `Size` operation as described above produces the specified result because its precondition limits it to being called during extraction phase, where the representation used in a batch-style implementation contains precisely the same edges as in the conceptual view. `Extract` as described above also works as advertised because, before it can be called, `Change_To_Extraction_Phase` has computed an MSF of the graph that was input during the insertion phase. The apparent discrepancy between the specification and a batch-style implementation with respect to *when* computation of an MSF occurs (seemingly incrementally during the insertion phase according to the specification, but actually in `Change_To_Extraction_Phase` in the implementation) simply cannot be detected by a client from functional behavior alone. So a batch-style implementation is as good as any other from this perspective.

3.2.2 Monotonic Deterministic Batch-Style Implementations

A deterministic batch-style implementation I that can be verified with an abstraction function exhibits an interesting property we term *monotonicity*, denoted $Mono(I)$. Consider the client code labeled `Find_MSF` which takes, as input, a sequence of edges $E_n = \langle e_1, e_2, ..., e_n \rangle$ and produces as output a set of edges $F_n = \{f_1, f_2, ..., f_k\}$. (The output order is irrelevant, so we view the output edges simply as a set, not a sequence.)

```
Find_MSF:
   if not Is_In_Insertion_Phase (m) then
      Change_To_Insertion_Phase (m)
   end if
   for i in 1..n loop
      let (v1, v2, w) = e_i
      Insert (m, v1, v2, w)
   end loop
   Change_To_Extraction_Phase (m)
   k = Size (m)
   for i in 1..k loop
      Extract (m, v1, v2, w)
      let f_i = (v1, v2, w)
   end loop
```

Given any $I \in \boldsymbol{B}$ as the underlying implementation of `Spanning_Forest_Machine_Template`, suppose we run `Find_MSF` on $E_n = \langle e_1, e_2, ..., e_n \rangle$, producing output F_n; and we run it on $E_{n+1} = \langle e_1, e_2, ..., e_n, e_{n+1} \rangle$, producing output F_{n+1}. Then we define:

$$Mono(I) \Leftrightarrow \forall E_{n+1}(IS_MSF(F_{n+1}, F_n \cup \{e_{n+1}\}))$$

That is, a deterministic batch-style implementation I is monotonic iff the output of `Find_MSF` using I, on any extension of any original input sequence E_n, is an MSF of the same extension of the original sequence's MSF F_n.

Using this property, we define the set of monotonic batch-style implementations:

$$\boldsymbol{M} = \{I \mid I \in \boldsymbol{B} \wedge Mono(I)\}$$

3.2.3 Sample Execution of a Nonmonotonic Deterministic Batch-Style Implementation

In Section 3.3, we will see that deterministic batch-style implementations which can be verified using abstraction functions are monotonic, so the implementations in $\boldsymbol{B} - \boldsymbol{M}$ interest us. The obvious question is whether there *are* any such implementations and, if so, whether they have practical significance. Fig. 3 helps us answer both questions affirmatively by showing the behavior of `Find_MSF` on an example for an implementation $I \in \boldsymbol{B} - \boldsymbol{M}$. In the figure, heavy lines depict possible minimum spanning forests of graphs; thin lines depict other edges inserted so far ("redundant" edges); and program states are identified by letters on the left.

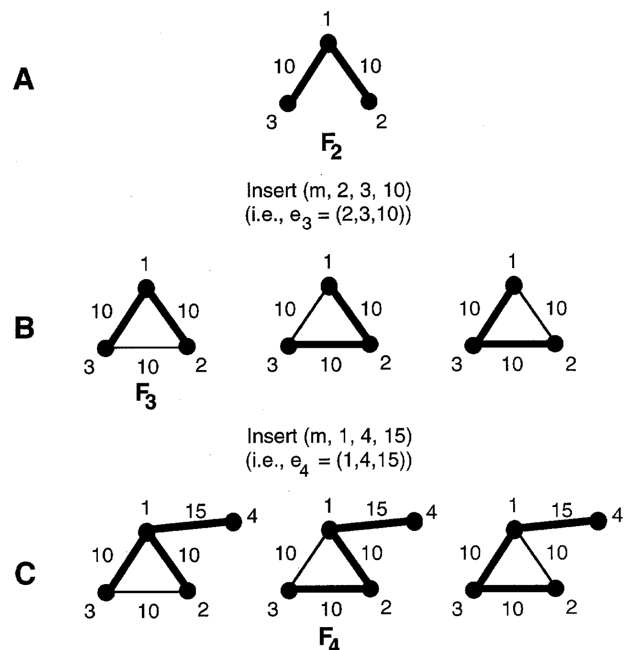

Fig. 3. Sample execution of `Find_MSF` for an implementation $I \in B - M$.

2. The representation also includes a flag indicating the machine's phase, which is handled in the obvious way and therefore is not discussed further.

Starting with an empty machine *m* in insertion phase, we first insert edges (1, 2, 10) and (1, 3, 10), in either order (i.e., E_2 = <(1, 2, 10), (1, 3, 10)> or E_2 = <(1, 3, 10), (1, 2, 10)>). At this point, state A, there is only one possible minimum spanning forest of the edges inserted so far. Calling `Change_To_Extraction_Phase` and then `Extract` until the machine is empty produces F_2 = {(1, 2, 10), (1, 3, 10)}. That is, `Find_MSF` on input E_2 outputs F_2.

Instead of calling `Change_To_Extraction_Phase` after state A, suppose we insert edge e_3 = (2, 3, 10). If the insertion phase ends at state B, `Find_MSF` returns one of three possible MSFs. Suppose (without loss of generality) it is the leftmost one in state B, so `Find_MSF` on input E_3 produces F_3 = {(1, 2, 10), (1, 3, 10)}. The input sequence so far is *not* a witness to the nonmonotonicity of *I* because F_3 *is* an MSF of $F_2 \cup \{e_3\}$.

However, suppose we continue in state B to insert one more edge e_4 = (1, 4, 15). If the insertion phase ends at state C, `Find_MSF` again returns one of three possible MSFs. But now suppose it is the middle one in state C, so `Find_MSF` on input E_4 produces F_4 = {(1, 2, 10), (2, 3, 10), (1, 4, 15)}. This input sequence *is* a witness to the nonmonotonicity of *I* because F_4 is *not* an MSF of $F_3 \cup \{e_4\}$. (In the figure, note that F_3 and F_4 include only the heavy edges.)

Are there really valid batch-style implementations of `Spanning_Forest_Machine_Template` that behave as in Fig. 3? While the answer to this question would be true even if there were only pathological programs that behave this way, the notable feature of the present example is that there is a large and natural class of implementations that serve as exemplars. For instance, implementations that do not keep the edges in input order during the insertion phase, and those that use typical published code for Kruskal's algorithm [4] and are based on sorting algorithms which are not necessarily stable (e.g., quicksort or heapsort), are all examples of actual implementations in ***B*** – ***M***.

3.3 Inadequacy of Abstraction Functions

We wish to prove that if $I \in \boldsymbol{B}$ (call this proposition *p*) and $I \notin \boldsymbol{M}$ (proposition *q*), then *I* cannot be verified using an abstraction function (proposition *r*). Notice that:

$$\begin{aligned}(p \wedge q) \Rightarrow r &\equiv \neg r \Rightarrow \neg(p \wedge q)\\ &\equiv \neg r \Rightarrow (\neg p \wedge \neg q)\\ &\equiv (p \wedge \neg r) \Rightarrow \neg q\end{aligned}$$

So, we begin by assuming $I \in \boldsymbol{B}$ (i.e., proposition *p*) and that *I* can be verified using an abstraction function (i.e., $\neg r$). We show this implies $I \in \boldsymbol{M}$ (i.e., $\neg q$).

Let *A* be the abstraction function used in the assumed proof of *I*. It maps a representation of a `Spanning_Forest_Machine` (call it ***m.rep***) to the corresponding conceptual value *m.edges*.[3] Now observe the detailed operation of `Find_MSF` by considering the trajectory of ***m.rep*** as `Find_MSF` executes with arbitrary input sequence E_n = <e_1, e_2, ..., e_n>, as illustrated in Fig. 4 (top trajectory). There, $\boldsymbol{m.rep_i}$ denotes the representation state immediately after the call that inserts e_i into *m*; $\boldsymbol{m.rep_{i'}}$ the representation state immediately after the call that extracts f_i from *m*; and $\boldsymbol{m.rep_0}$ ($\boldsymbol{m.rep_{0'}}$) the state immediately before the first `Insert` (`Extract`) operation.

3. We ignore the remainder of the correspondence, which trivally maps a flag indicating the machine's phase to the other component of the conceptual value, `m.insertion_phase`.

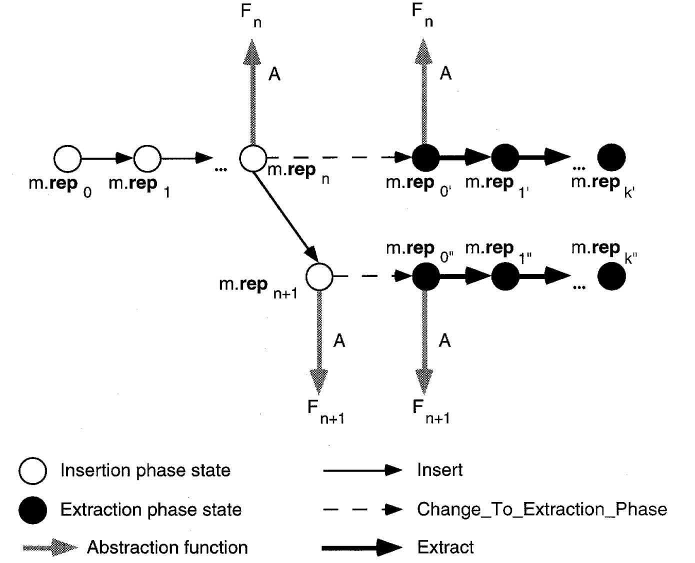

Fig. 4. Behavior of `Find_MSF` for E_{n-1} and E_n.

After all edges are inserted, there is a call to `Change_To_Extraction_Phase`. Because *I* is a batch-style implementation we expect that $\boldsymbol{m.rep_{0'}} \neq \boldsymbol{m.rep_n}$. However, by the assumption that *I* can be verified we know from the postcondition of `Change_To_Extraction_Phase` that *m.edges* does not change; thus:

$$A(m.\mathbf{rep}_{0'}) = A(m.\mathbf{rep}_n) \tag{1a}$$

By the same assumption, we know that each subsequent call to `Extract` removes one edge from $A(\boldsymbol{m.rep_{0'}})$. This means the loop in `Find_MSF` produces F_n as its output; so:

$$F_n = A(m.\mathbf{rep}_{0'}) \tag{1b}$$

The trajectory of ***m.rep*** as `Find_MSF` executes with input sequence E_{n+1} = <e_1, e_2, ..., e_n, e_{n+1}> is similar. Because *I* is deterministic, the representation state follows precisely the same trajectory as before through insertion of edge e_n. But this time we continue by inserting e_{n+1}, changing the new representation state to $\boldsymbol{m.rep_{n+1}}$ (bottom trajectory). Subsequent representation states may be different than for the first input sequence, so we mark them with double primes (") in place of single primes ('). But by the same arguments as above we conclude:

$$A(m.\mathbf{rep}_{0''}) = A(m.\mathbf{rep}_{n+1}) \tag{2a}$$

$$F_{n+1} = A(m.\mathbf{rep}_{0''}) \tag{2b}$$

By assumption, the `Insert` operation also can be verified, so it works correctly when we insert the edge e_{n+1} (the diagonal arrow in Fig. 4). From the postcondition of `Insert` with appropriate substitutions for that invocation, we therefore know:

$$IS_MSF(A(m.\mathbf{rep}_{n+1}), A(m.\mathbf{rep}_n) \cup \{e_{n+1}\}) \tag{3}$$

Now substituting in (3) from (1a), (1b) and (2a), (2b), we deduce:

$$IS_MSF(F_{n+1}, F_n \cup \{e_{n+1}\}) \quad (4)$$

But if (4) holds then $I \in \boldsymbol{M}$. We conclude, then, that every valid deterministic batch-style implementation of `Spanning_Forest_Machine` that can be proved using an abstraction function is monotonic. Yet we know there are valid and practical deterministic batch-style implementations that are nonmonotonic. A proof system that relies solely on abstraction functions, therefore, cannot be used to verify the correctness of any member of this entire class of correct and practical implementations of `Spanning_Forest_Machine_Template`.

3.4 Verification Using Abstraction Relations

The implementations discussed above have a natural and simple abstraction relation—the abstract value *m.edges* is any one of the MSFs of the graph stored in the internal representation ***m.rep***. This relation, stated below formally using the predicate *IS_MSF*, is sufficient to prove the correctness of the implementations:

$$IS_MSF(m.edges, S(m.\boldsymbol{rep}))$$

where S is a function from the specific representation of machine m to the mathematical set of edges it contains. Details of the correctness proof are provided in Appendix B.

The MSF example shows that abstraction relations are essential for proving correctness of some implementations of nontrivial relational specifications. Such situations should be expected to arise in industrial-strength software systems. It is also possible that abstraction relations might be used—even though with effort they could be avoided in some cases—where they can simplify verification conditions and can be easier to understand than abstraction functions.

We note that a relational programming language semantics ultimately cannot be avoided if specifications are allowed to be relational—which they must be in order to permit specification of behavior such as that desired for `Spanning_Forest_Machine_Template` [7], [17]. Given that a relational semantics is essential, abstraction relations between concrete and abstract values do not increase verification complexity. For example, it is much easier to define the relation *IS_MSF* than the specific function computed by any given implementation, which depends on intricate algorithmic details involved in finding a particular MSF and which are inessential in the proof. Using an abstraction relation considerably simplifies the verification conditions for each of the `Spanning_Forest_Machine_Template` operations because the correspondence mapping is used separately in the verification of each operation [6], [7].

4 Discussion

The literature on verifying ADTs includes at least two conjectures that abstraction relations, even if technically required in some circumstances, are probably unnecessary in practice—at least where specifications are well-designed. When optimization problems are specified as procedures (for example, a simple operation for finding an MSF), this conjecture might be true. That situation demands relational specification of behavior and relational semantics, but not necessarily abstraction relations. The abstraction relation problem arises here because an optimization problem with possible ties has been captured not as a single procedure, but by recasting it as an ADT. In light of the advantages of the recasting approach [24], the abstraction relation issue assumes additional practical significance.

Formal systems that handle abstraction relations have been discussed (e.g., [9]) and some formal methods tool kits support them (e.g., Cogito [2]). But historically, abstraction functions have been so important and they are so entrenched that the generalization to abstraction relations tends to be resisted in some quarters. So we now examine (not necessarily published) attempts we have seen to avoid abstraction relations, and their ramifications.

The first approach is to prohibit the specification of relational behavior of operations. This would be undesirable when, as in this case, the natural intended behavior is inherently relational. Refusing to admit this possibility would leave a class of useful abstractions that could not be specified or that could not be easily reused in building other component implementations [7], [11], [13]. Moreover, specifying functional behavior for the MSF problem would rule out interesting classes of implementations and would make the specification much harder to understand—that specification would have to single out precisely which MSF must be produced, even in case of ties.

A second approach is to augment ***m.rep*** with an adjunct (auxiliary) variable, say *m.**rep**.abs*, which simply mirrors the abstract value. This would give a trivial abstraction function: $m = m.\boldsymbol{rep}.abs$, and it would introduce a representation convention (invariant) relating *m.**rep**.abs* to the rest of ***m.rep*** (i.e., the original concrete representation). Notice that the availability of this approach does not refute our proof in Section 3 because the adjunct code required to update *m.**rep**.abs* would be nondeterministic, and any implementation written like this would not be in ***B***. Nonetheless it might be argued that any implementation in ***B*** *could* be transformed in this way in order to avoid abstraction relations.

Even if valid, this suggestion would have little practical import because following it would just move the expression of the required relation from the correspondence to the representation convention. Correctness proofs would not be simplified at all. But the bigger problem is that a batch implementation of `Spanning_Forest_Machine_Template` that employed this device would be incorrect. The adjunct code to update *m.**rep**.abs* in Insert would have to select a particular MSF prematurely (albeit nondeterministically), and subsequent `Extract` operations could not be proved to return precisely the edges of the selected MSF.

A third approach involves changing the specifications. This has been considered both in the ADT framework [11] and in related work [5] on concurrent processes involving I/O automata and sequences of actions. Lynch documents the practical utility of "multivalued possibilities mappings" (the I/O automata counterpart of abstraction relations) [16]. However, Abadi and Lam-

port show that specifications can be transformed to avoid multivalued mappings, under certain conditions; "refinement mappings" (the counterpart of abstraction functions) always exist [1]. Abadi and Lamport introduce techniques to avoid abstraction relations which, when adapted to the ADT framework, require changing the specifications of some of the components involved in the proof. Changes to these specifications are ruled out by the modularity requirements we place on ADT correctness proofs.

The issue of changing specifications of components does raise the question of whether the `Spanning_Forest_Machine_Template` specification could be designed differently to avoid the need for abstraction relations. For example, suppose that the specification is changed to be along the lines discussed in Section 3.4, i.e., conceptually all the edges are kept during the insertion phase, and an MSF is chosen only in `Change_To_Extraction_Phase`. Then for an implementation that mirrors this specification temporally, i.e., for a batch-style one that computes an MSF in `Change_To_Extraction_Phase`, an abstraction function is sufficient for a proof of correctness. However, the amortized-cost implementation in [24], and implementations that defer computation of an MSF to the `Extract` operation such as the one discussed in Section 2.2, still require an abstraction relation. Furthermore, if one must change specifications for the sake of avoiding abstraction relations in correctness proofs—without regard for the impact on understandability to potential component clients [20], [26]—then some of the most important practical benefits of formal specification for software engineering may be lost.

Even if a specification of `Spanning_Forest_Machine_Template` is devised that avoids the need for abstraction relations for that example [22], the completeness and naturalness needs raised in this paper remain. The practical requirement for abstraction relations to handle specifications that are not fully abstract will also continue to exist, because software developers are likely to continue to design and use such concepts. Fully embracing abstraction relations is therefore an essential practical step in broadening the applicability of formal methods beyond simplistic data abstractions.

APPENDIX A – RESOLVE NOTATION

The specifications in this paper are written in RESOLVE, a detailed description of which is available elsewhere [21]. Here we give a brief overview of RESOLVE notation that (along with a general understanding of issues in specification and implementation of abstract data types) should be sufficient to enable a reader to understand the examples in this paper.

A.1 Specification Notation

A RESOLVE **concept** specifies an "abstract template" (generic abstract module) by listing its **context**, which explicitly defines all coupling to the environment and makes all local declarations used in the rest of the specification; and its exported **interface**. The **global context** section identifies fixed coupling of this module to others in a shared component library. The **parametric context** section defines the ways in which a client can provide context, through parameters, when instantiating the generic module. The **local context** section typically introduces special-purpose mathematical notation used in the interface specification. For example, in Fig. 2, *IS_MSF* is a mathematical operation (function). Its **definition** should say formally that *IS_MSF*(*msf*, *g*) is true iff *msf* is a minimum spanning forest of *g*. We have elided this to focus on the more important features of the specification, but *IS_MSF* can be formally defined in a few lines.

The **interface** section explains the concept's exported types and operations. Each program type (family) is explained by referring to its mathematical model. For example, the type `Spanning_Forest_Machine` in Fig. 2 is modeled as an ordered pair consisting of a set of *EDGEs* and a boolean value. The **constraint** clause for a mathematical model (e.g., *EDGE* or the model for `Spanning_Forest_Machine`) says that, of all possible values of the underlying mathematical space, only those satisfying that clause are part of the model space.

Every program type has three implicit operations:

1) The **initialize** operation is invoked only at the beginning of the scope where its argument is declared. It gives the variable an initial value, which is specified in the **initialization ensures** clause of the type specification.
2) The **finalize** operation is invoked only at the end of the scope where its argument is declared, so usually there is no need to specify its abstract effect—because there is none. This operation is generally a hook for the type's implementer to release resources (e.g., memory) used by the representation.
3) The **swap** operation (invoked using the infix :=: operator) exchanges the values of its two arguments [8].

RESOLVE specifications never include preconditions like "*x* has been initialized" because client programs *always* initialize and finalize variables at the beginning and end of scope, respectively. In RESOLVE **initialize** and **finalize** work like C++ constructor and destructor operations, with appropriate calls generated by the compiler.

The effect of each operation is specified using a **requires** clause (precondition) and an **ensures** clause (postcondition). Each of these is an assertion about the values of the mathematical models of the operation's parameters. A missing clause means the assertion is the constant **true**. Mathematically, an operation defines a partial relation on the space of input and output values of the parameters: The **requires** clause tells where the relation is defined, and the **ensures** clause defines it there. Operationally, the contract between operation client and implementer is as follows: If a client calls an operation in a state in which the **requires** clause holds for the actual parameters, then the implementer guarantees that the operation will return in a state in which the **ensures** clause holds; but if the **requires** clause does not hold when the call occurs, then the implementer makes no guarantees whatsoever.

In a **requires** clause a variable stands for its value at the beginning of a call. In an **ensures** clause a variable stands for its value at the end of the call, while a variable with a prefixed # (pronounced "old") stands for the value of that variable at the beginning of the call. Other

mathematical notations depend on the mathematical theories involved in the explanation of behavior. The specification of `Spanning_Forest_Machine_Template` uses finite sets, tuples, integers, and booleans.

Operation specifications are considerably simplified by using abstract parameter modes **alters**, **produces**, **consumes**, and **preserves**. An **alters**-mode parameter potentially is changed by executing the operation; the **ensures** clause says how. A **produces**-mode parameter gets a new value that is defined by the **ensures** clause, which may *not* involve the parameter's old value because it is irrelevant to the operation's effect. A **consumes**-mode parameter gets a new value that is an initial value for its type, but its old value is relevant to the operation's effect. A **preserves**-mode parameter suffers no net change in value between the beginning of the operation and its return, although its value might be changed temporarily while the operation is executing.

A.2 Realization Notation

A RESOLVE **realization** describes a "concrete template" (generic implementation module). A **facility** is an instance of a concept with an associated instance of a realization which implements it, so its declaration involves choosing and fixing the parameters of both a concept and one of that concept's realizations. In operation bodies, the representation of a variable (e.g., *m*) of an exported type is designated as ***m.rep*** so it is clear that this is the representation model's value and not the conceptual model's value.

RESOLVE realization code contains three kinds of assertions. For every loop there is a loop invariant or loop specification; and for every type there is a **convention** assertion that characterizes the subspace of representation configurations that might arise (the representation invariant), and a **correspondence** assertion that explains how to associate such representation configurations with conceptual model values (the abstraction relation).

The **convention** clause of Fig. 5 uses the built-in RESOLVE mathematical function **elements**, which denotes the set of entries in the string of items which is its argument. So, if *str* = <a, b, c, b>, then **elements** (*str*) = {a, b, c}.

APPENDIX B – VERIFICATION OF A BATCH-STYLE IMPLEMENTATION

In this appendix we prove the correctness of the batch-style implementation of `Spanning_Forest_Machine_Template` shown in Fig. 5. Its global context section refers to `Queue_Template`, which is shown in Fig. 6 for completeness.

A proof of correctness [7] of the realization of Fig. 5 starts by factoring out two lemmas that arise during the verification of each individual operation:

C1. For every representation state for which the **convention** clause holds, there is a conceptual state to which the **correspondence** clause relates it.

C2. For every representation state for which the **convention** clause holds, and for every conceptual state related to it by the **correspondence** clause, the **constraint** clause (see Fig. 2) holds for the conceptual state.

In this case, to prove C1 we must prove that there is at least one conceptual `Spanning_Forest_Machine` value for every Machine_Rep value that can arise. This follows from the definition of MSF in graph theory (which we assume is encoded formally in the predicate *IS_MSF*); i.e., every graph has an MSF. To prove C2 we must prove that any MSF of any graph is its own MSF; and again this is a simple lemma in graph theory.

The verification is completed by showing that for each operation body, the code implements the associated abstract operation specification. For each operation and for each fixed assignment of values to all the other arguments, we consider four sets of values for each `Spanning_Forest_Machine` argument: initial and final conceptual states, C_i and C_f, respectively; and initial and final representation states, R_i and R_f, respectively. R_i contains those representation states for which: 1) the **convention** clause holds, 2) there exists a conceptual state satisfying the **correspondence** clause and this particular operation's abstract precondition, and 3) every such corresponding conceptual state satisfies this operation's precondition. R_f contains the representation states that can be reached from some representation state in R_i by correct execution of the operation's body. C_i and C_f contain the conceptual states for which the **correspondence** clause holds for some representation state in R_i and R_f, respectively.

Informally stated, we have three kinds of proof obligations; i.e., the verification conditions are of these three forms:

V1. For every $r \in R_i$ and for every trajectory leading from *r* through the operation body, all internal assertions (e.g., loop invariants) hold at the appropriate times, and the preconditions of all called operations hold at the points they are called. (This obligation arises from the need to define R_f, since only if a called operation's precondition holds may we assume that its effect is what we expect from its specified postcondition.)

V2. For every $r \in R_f$ the **convention** clause holds. (This obligation arises from the need to define C_f, since only in this case is it certain that there is some conceptual state to which the **correspondence** clause relates every $r \in R_f$.)

V3. For every $\#r \in R_i$, $r \in R_f$, and $c \in C_f$ for which *r* is reachable from *#r* by some correct execution of the operation body and where the **correspondence** clause relates *c* and *r*, there exists some $\#c \in C_i$ for which the **correspondence** clause relates *#c* and *#r* and where the operation's abstract postcondition holds for *#c* and *c*. (This obligation arises from the need to complete the "commutativity diagram" [7, pp. 303-305].)

There also is a specialized version of such a proof for the **initialize** operation, where we may neither assume that the **convention** clause holds for the initial representation state nor, consequently, that there is any initial conceptual state corresponding to it.

In this case, it is easy to discharge the obligations of the forms V1 and V2 for each operation. There is only

```
realization body Batch for Spanning_Forest_Machine_Template

   context

      global context
         concept Record3_Template
         concept Queue_Template
         concept Record2_Template
         facility Standard_Boolean_Facility

      local context

         type Edge is record
            v1: Integer
            v2: Integer
            e: Integer
         end record

         facility Edge_Queue_Facility is Queue_Template(Edge)
                     realized by Queue_Realization_1

         operation Produce_MSF (
                 alters q: Edge_Queue_Facility.Queue
              )
            ensures
              IS_MSF (elements (q), elements (#q)) and
              |q| = |elements (q)|
         begin
            -- code that batch computes an arbitrary MSF of q
         end Produce_MSF

   interface
      type Spanning_Forest_Machine is represented by record
         graph_edges: Edge_Queue_Facility.Queue
         insertion_flag: Boolean
      end record
         convention
            if not m.rep.insertion_flag
            then IS_MSF (elements (m.rep.graph_edges),
                           elements (m.rep.graph_edges))
         correspondence
            IS_MSF (m.edges, elements (m.rep.graph_edges)) and
            m.insertion_phase = m.rep.insertion_flag

      operation initialize
         begin
            m.rep.insertion_flag := true
         end initialize

      operation Change_To_Insertion_Phase (
              alters      m: Spanning_Forest_Machine
            )
         local context
            variables
              new_rep: Spanning_Forest_Machine
         begin
           m.rep :=: new_rep
           m.rep.insertion_flag := true
         end Change_To_Insertion_Phase

      operation Insert (
              alters      m: Spanning_Forest_Machine
              consumes    v1: Integer
              consumes    v2: Integer
              consumes    w: Integer
           )
         begin
            Enqueue (m.rep.graph_edges, (v1, v2, w))
         end Insert
```

```
    operation Change_To_Extraction_Phase (
            alters        m: Spanning_Forest_Machine
        )
      begin
        Produce_MSF (m.rep.graph_edges)
        m.rep.insertion_flag := false
      end Change_To_Extraction_Phase

    operation Extract (
            alters        m: Spanning_Forest_Machine
            produces      v1: Integer
            produces      v2: Integer
            produces      w: Integer
        )
      begin
        Dequeue (m.rep.graph_edges, (v1, v2, w))
      end Extract

    operation Size (
            preserves     m: Spanning_Forest_Machine
        ): Integer
      begin
         return Length (m.rep.graph_edges)
      end Size

    operation Is_In_Insertion_Phase (
            preserves m: Spanning_Forest_Machine
        ): Boolean
      begin
        return m.rep.insertion_flag
      end Is_In_Insertion_Phase

end Batch
```

Fig. 5. Realization body for a batch-style implementation.

```
concept Queue_Template

  context

    global context
      facility Standard_Integer_Facility

    parametric context
      type Item

  interface

    type Queue is modeled by string of math[Item]
      exemplar q
      initialization ensures
         q = empty_string

    operation Enqueue (
            alters      q: Queue
            consumes    x: Item
        )
      ensures
         q = #q * <#x>

    operation Dequeue (
            alters      q: Queue
            produces    x: Item
        )
      ensures
        #q = <x> * q

    operation Length (
            preserves   q: Queue
```

```
        ): Integer
      ensures
        Length = |q|

  end Queue_Template
```

Fig. 6. Specification of `Queue_Template`.

one called operation (`Dequeue` in the body of `Extract`) that has a nontrivial precondition, and in this case the precondition of `Extract` implies that R_i contains only representation states where *m.**rep**.graph_edges* is not empty. Showing that the **convention** clause holds at the end of each operation body is more tedious because it involves *m.**rep**.insertion_flag* as well as *m.**rep**.graph_edges*, but the details are straightforward.

All the proof obligations of the form V3 are similarly trivial, except for the `Insert` operation. Here, the proof of the only troublesome part of the applicable verification condition follows directly from the graph theory lemma stated in Section 3.

We observe that the verification of this batch implementation answers the question posed in Section 3.1.1: Can the edges obtained from a series of `Extract` operations be *any* MSF of the edges that were inserted into a `Spanning_Forest_Machine`? The body of procedure `Produce_MSF` in Fig. 5 may produce *any* MSF of the edges it is given. The realization in Fig. 5 is correct with no further assumptions about which MSF that must be. So the specification of `Spanning_Forest_Machine_Template` truly places no restriction on which MSF of the inserted edges might be produced.

Acknowledgments

We thank Anish Arora, Steve Edwards, David Fleming, Wayne Heym, Joe Hollingsworth, Chip Klostermeyer, Tim Long, Sethu Sreerama, and Stu Zweben for their insightful comments on drafts of this paper; and Doug Kerr, Nathan Loofbourrow, Spiro Michaylov, and Tom Page for helpful discussions on some key technical points. Comments from the anonymous referees have helped clarify several issues.

Murali Sitaraman was sponsored by the National Science Foundation (NSF) under grant CCR-9204461; Advanced Research Projects Agency (ARPA) under contract numbers DAAH04-96-1-0419 and DAAH04-94-G-0002, both monitored by the U.S. Army Research Office; and the National Aeronautics and Space Administration (NASA) under grant 7629/229/0824.

Bruce W. Weide and William F. Ogden were sponsored by NSF under grant number CCR-9311702 and by ARPA under contract number F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714.

References

[1] M. Abadi and L. Lamport, “The Existence of Refinement Mappings,” *Theoretical Computing Science,* vol. 82, no. 2, pp. 253–284, May 1991.

[2] A. Bloesch, E. Kazmierczak, P. Kearney, and O. Traynor, “Cogito: A Methodology and System for Formal Software Development,” *Intl. J. Software Eng. and Knowledge Eng.,* vol. 5, no. 4 pp. 599–617, Dec. 1995.

[3] S.A. Cook, “Soundness and Completeness of an Axiom System for Program Verification,” *SIAM J. Computing,* vol. 7, no. 1, pp. 70–90, Feb. 1978.

[4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms.* Cambridge, Mass.: MIT Press, 1990.

[5] H.-D. Ehrich and A. Sernadas, “Algebraic Implementation of Objects over Objects,” *Stepwise Refinement of Distributed Systems—Lecture Notes in Computer Science 430.* New York: Springer-Verlag, pp. 239–266, 1990.

[6] G.W. Ernst, R.J. Hookway, J.A. Menegay, and W.F. Ogden, “Modular Verification of Ada Generics,” *Computer Language,* vol. 16, nos. 3/4, pp. 259–280, 1991.

[7] G.W. Ernst, R.J. Hookway, and W.F. Ogden, “Modular Verification of Data Abstractions with Shared Realizations,” *IEEE Trans. Software Eng.* vol. 20, no. 4, pp. 288–307, Apr. 1994.

[8] D.E. Harms and B.W. Weide, “Copying and Swapping: Influences on the Design of Reusable Software Components,” *IEEE Trans. Software Eng.,* vol. 17, no. 5, pp. 424–435, May 1991.

[9] J. He, C.A.R. Hoare, and J.W. Sanders, “Data Refinement Refined,” *Lecture Notes in Computer Science 213,* B. Robinet and R. Wilhelm, eds., pp. 187–196, Springer-Verlag.

[10] C.A.R. Hoare, “Proof of Correctness of Data Representations,” *Acta Informatica,* vol. 1, no. 1, pp. 271–281, 1972; also in D. Gries, ed., *Programming Methodology: A Collection of Articles by Members of IFIP WG 2.3,* pp. 269–281, New York: Springer-Verlag, 1978.

[11] C.B. Jones, *Systematic Software Development Using VDM,* Hertfordshire, U.K.: Prentice Hall Int’l, 1990.

[12] D. Kapur and S. Mandayam, “Expressiveness of the Operation Set of a Data Abstraction,” *Conf. Record Seventh Ann. ACM Symp. Principles of Programming Languages,* pp. 139–153, ACM, 1980.

[13] G.T. Leavens, “Modular Specification and Verification of Object-Oriented Programs,” *IEEE Software,* vol. 8, no. 4, pp. 72–80, July 1991.

[14] B. Liskov and J.M. Wing, “A New Definition of the Subtype Relation,” *ECOOP 1993—Lecture Notes in Computer Science 707,* pp. 118–141. New York: Springer-Verlag, 1993.

[15] B. Liskov and J.M. Wing, “Corrigenda to ECOOP ‘93 Paper,” *ACM SIGPLAN Notices,* vol. 29, no. 4, p. 4, Apr. 1994.

[16] N. Lynch, “Multivalued Possibilities Mappings,” *Stepwise Refinement of Distributed Systems—Lecture Notes in Computer Science 430,* pp. 519–543, New York: Springer-Verlag, 1990.

[17] G. Nelson, “A Generalization of Dijkstra’s Calculus,” *ACM Trans. on Program Languages and Systems,* vol. 11, no. 4, pp. 517–561, Oct. 1989.

[18] T. Nipkow, “Are Homomorphisms Sufficient for Behavioral Implementations of Deterministic and Nondeterministic Data Types?” *Lecture Notes in Computer Science 247,* F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, eds., pp. 260–271, New York: Springer-Verlag, 1987.

[19] O. Schoett, “Behavioral Correctness of Data Representations,” *Science of Computer Programming,* vol. 14, pp. 43–57, 1990.

[20] M. Sitaraman, L.R. Welch, and D.E. Harms, “On Specification of Reusable Software Components,” *Intl. J. Software Eng. and Knowledge Eng.,* vol. 3, no. 2 , pp. 207–219, June 1993.

[21] “Special Feature: Component-Based Software Using RESOLVE,” M. Sitaraman and B.W. Weide, eds., *ACM SIGSOFT Software Eng. Notes,* vol. 19, no. 4, pp. 21–67, Oct. 1994.

[22] M. Sitaraman, “Impact of Performance Considerations on Formal Specification Design,” *Formal Aspects of Computing,* vol. 8, no. 6, pp. 716–736, 1997.

[23] B.W. Weide and J.E. Hollingsworth, “Scalability of Reuse Technology to Large Systems Requires Local Certifiability,” *Proc. Fifth Ann. Workshop Software Reuse,* Palo Alto, Calif., Oct. 1992.

[24] B.W. Weide, W.F. Ogden, and M. Sitaraman, “Recasting Algorithms to Encourage Reuse,” *IEEE Software,* vol. 11, no. 5 pp. 80–88, Sept. 1994.

[25] B.W. Weide, S.H. Edwards, W.D. Heym, T.J. Long, and W.F. Ogden, "Characterizing Observability and Controllability of Software Components," *Proc. Fourth Int'l Conf. Software Reuse*, M. Sitaraman, ed., pp. 62–71, IEEE, Apr. 1996.

[26] J.M. Wing, "A Specifier's Introduction to Formal Methods," *Computer,* vol. 23, no. 9, pp. 8–24, Sept. 1990.

Murali Sitaraman received his ME in computer science from the Indian Institute of Science, Bangalore, and his PhD in computer and information science from The Ohio State University. He is presently an associate professor of computer science at West Virginia University. Sitaraman is a principal investigator of the RESOLVE research effort and directs the Reusable Software Research Group at WVU (http://www.cs.wvu.edu/~resolve). His interests span theoretical, practical, and educational aspects of software engineering. Sitaraman served as program chair of the *Fourth International Conference on Software Reuse* (Orlando, Florida, April 1996). He is a member of the IEEE Computer Society and ACM.

Bruce W. Weide holds a PhD in computer science from Carnegie Mellon University and a BSEE from the University of Toledo. He is professor of computer and information science at The Ohio State University, where he co-directs 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 practice and in teaching its principles to beginning CS students. He is a member of the IEEE, ACM, and CPSR.

William F. Ogden received his MS and PhD degrees from Stanford University, and his BS degree from the University of Arkansas. He is an associate professor of computer and information science at The Ohio State University and co-director of the Reusable Software Research Group with Bruce Weide and Stu Zweben. Previously, he served on the faculties at Case Western Reserve University and the University of Michigan. His main research interests are in software reuse, software specification, and program verification. He is a member of the IEEE Computer Society and ACM.