Multivalued Possibilities Mappings [*]

Nancy A. Lynch
Laboratory for Computer Science
MIT
Cambridge, MA 02139

Abstract: Abstraction mappings are one of the major tools used to construct correctness proofs for concurrent algorithms. Several examples are given of situations in which it is useful to allow the abstraction mappings to be multivalued. The examples involve algorithm optimization, algorithm distribution, and proofs of time bounds.

Keywords: abstraction mapping, mapping, possibilities mapping, safety property, Alternating Bit Protocol, transaction processing, garbage collection, distributed algorithms, time bounds, history variables

Contents

1 Introduction

2 A Formal Framework

3 Algorithm Optimization
3.1 Alternating Bit Protocol
3.1.1 Problem Statement
3.1.2 Architecture
3.1.3 Alternating Bit Protocol
3.1.4 Redundant Protocol
3.1.5 Possibilities Mapping Proof
3.1.6 Remarks
3.2 Transaction Processing
3.3 Garbage Collection
3.4 Remarks

4 Distribution

5 Proving Time Bounds
5.1 Overview
5.2 Formal Model
5.2.1 Timed Automata
5.2.2 The Automaton *time*(*A*)
5.2.3 Strong Possibilities Mappings

[*] This work was supported by ONR contract N0014-85-K-0168, by NSF contract CCR-8611442, and by DARPA contract N00014-83-K-0125.

5.3 The Algorithm
5.4 The Performance Automaton
5.5 Proof

6 Conclusions

A Proof

1 Introduction

Abstraction mappings are one of the major tools that my colleagues and I use to construct correctness proofs for concurrent (including distributed) algorithms. In this paper, I will try to make one major point about such mappings: that it is useful to allow them to be *multivalued.* That is, often when one maps a "low-level" algorithm L to a "high-level" algorithm H, one would like to allow *several* states of H to correspond to a *single* state of L. I believe that any useful framework for describing abstraction mappings should include the ability to describe multivalued mappings.

I don't know if this point is especially controversial. I have been using multivalued mappings since I started carrying out such proofs in 1981, and the popular notion of *bisimulation* proposed by Milner [20] also permits multiple values (although bisimulation is a stronger notion than I advocate here, since it requires simulation relationships between L and H in both directions). However, work on *history variables*, tracing its roots to [22], takes pains to avoid the use of multivalued mappings by adding extra information to the state of L, and there are also some recent papers (e.g., [13, 12, 1]) that restrict the notion of mapping to be single-valued.

I will describe some situations in which multivalued abstraction mappings are useful. The examples I consider involve

1. algorithm optimization,

2. distribution, and

3. proving time bounds.

I will illustrate the first of these situations in some detail, using one familiar example (the Alternating Bit Protocol) and two less familiar examples, just touch on the second, and spend the remaining time on the third - it's the newest use I have found and possibly the most interesting.

In my work, abstraction mapping seem most useful for proving safety properties; although I have been involved in some work that proves liveness properties using such mappings (e.g., [17, 27]), these efforts are still somewhat ad hoc. Note that timing properties are more like safety properties than like liveness properties; because of this, mappings are useful for proving timing properties as well. In this paper, I will restrict attention to safety and timing properties.

2 A Formal Framework

To be concrete, I will describe the work in terms of I/O automata [17, 18], since that is what I've actually used. The precise choice of model is not very important for most of what I will discuss here (timing proofs excepted); other state machine models would probably do as well. Here, I will review the definition of an I/O automaton and will give the usual notion of mapping, called a *possibilities mapping*, that I use for defining a correspondence between I/O automata.

Recall that an I/O automaton consists of *states*, *start states*, *actions* classified by a *signature* as *output*, *input* and *internal*, and *steps*, which are (state, action, state) triples. So far, that makes them rather ordinary state machines. There is a fifth component that is not normally relevant to my work involving mappings (but that I will use in the timing example): a *partition* of the output and internal actions into classes indicating which are under the control of the same underlying component in the system being modeled by the automaton. Its main purpose is in describing fair executions of the automaton - executions that allow each component fair turns to continue taking steps. For now, I will ignore this partition.

An *extended step* of an automaton describes a state change that can occur as a result of a finite sequence of actions.

The important behavior of an I/O automaton is normally considered to be its interaction with its environment, in the form of its *behaviors*, i.e., its sequences of input and output actions (more precisely, its fair sequences). Problems to be solved by I/O automata are specified as sets of sequences of such actions, and an automaton is said to *solve* a problem if its (fair) behaviors are a subset of the set of problem sequences.

Let L and H be two I/O automata with the same external action signature (same inputs and outputs). Define a *possibilities mapping* from L to H to be a mapping f from $states(L)$ to the power set of $states(H)$ satisfying the following properties.

1. For every start state s_0 of L, there is a start state u_0 of H such that $u_0 \in f(s_0)$.
2. If s' is a reachable state of L, $u' \in f(s')$ is a reachable state of H and (s', π, s) is a step of L, then there is an extended step (u', γ, u) of H such that:
   (a) $\gamma | ext(H) = \pi | ext(L)$, and
   (b) $u \in f(s)$.

The basic theorem about possibilities mappings is:

Theorem 1 *If there is a possibilities mapping from L to H, then all behaviors of L are also behaviors of H.*

This theorem suggests how a possibilities mapping can be used in proving safety properties (defined here to be nonempty, prefix-closed, limit-closed properties of external action sequences) for an automaton L. For example, a safety property P might be specified as the set of behaviors of an automaton H. Then a possibilities mapping from L to H shows that the behaviors of L all satisfy P. For another example, it might be possible to show that the behaviors of an automaton H all satisfy a safety property P; then a possibilities mapping from L to H shows that the behaviors of L all satisfy P.

Concurrent systems are modeled by compositions of I/O automata, as defined in [17, 18]. In order to be composed, automata must be *strongly compatible*; this means that no action can be an output of more than one component, that input actions of one component are not shared by any other component, and that no action is shared by infinitely many components. The result of such a composition is another I/O automaton.

3 Algorithm Optimization

An important use of a possibilities mapping is to decompose the correctness proof for an "optimized" algorithm L using an "unoptimized" variation H as an intermediate stage. Typically, H would be a simple and redundant algorithm that is easy to understand because it maintains a lot of intuitively meaningful information. The algorithm L would be less redundant, more efficient, and correspondingly more difficult to understand. The behavior of L would be very similar to that of H, but would be determined on the basis of less information. A good way to correspond the two algorithms is via a multivalued mapping from L to H. The mapping "puts back" the information that is lost in "optimizing" H; since there may not be a unique way to do this, the mapping must be multivalued.

In this section, I give three examples. The first is a version of the well-known Alternating Bit Protocol [4], the second an example from database concurrency control, and the third an example from highly available replicated data management.

3.1 Alternating Bit Protocol

I begin with the Alternating Bit Protocol (ABP), mostly because it is simple and should be familiar from other papers on verification. Although the main interest in this example is normally the liveness properties, here I will only consider safety. The key safety property to be proved is, roughly speaking, that the subsequence of messages delivered is a prefix of the subsequence sent.

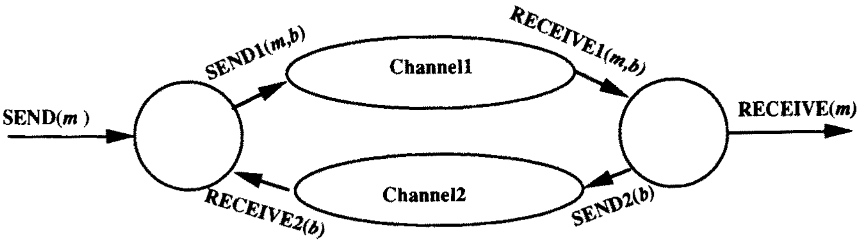
3.1.1 Problem Statement

More specifically, I define correctness at the external boundary of the ABP component (the data link boundary). The input actions are $SEND(m)$, where $m \in M$, the message alphabet. The output actions are $RECEIVE(m), m \in M$. The correctness property P is the set of sequences β of $SEND$ and $RECEIVE$ actions such that in any prefix β' of β, the sequence of messages received in β' is a prefix of the sequence of messages send in β'.

SEND(*m*)
data link
RECEIVE(*m*)

3.1.2 Architecture

The architecture for an implementation consists of a *sender* automaton, a *receiver* automaton, and two FIFO physical channels, *channel1* and *channel2*. Channel1 has input actions $SEND1(m, b)$ and output actions $RECEIVE1(m, b)$, where $m \in M$ and b is a Boolean. Channel2 has input actions $SEND2(b)$ and output actions $RECEIVE2(b)$, where b is a Boolean. The system is modeled by the composition of these automata, with all actions except $SEND(m)$ and $RECEIVE(m)$ hidden.



The channels are fairly ordinary FIFO queues, except that the effect of a $SEND1$ or $SEND2$ action might or might not be to put the data at the end of the queue. (The effect of a $RECEIVE1$ or $RECEIVE2$ is always to remove it, however.) More specifically, consider channel1. Its state is a finite queue of pairs (m, b), where $m \in M$ and b is a Boolean. Initially, the queue is empty.

$SEND1(m, b), m \in M, b$ a Boolean
Effect:

```
        Either add $(m, b)$ to the queue or do nothing.

$RECEIVE1(m, b), m \in M, b$ a Boolean
Precondition:
        $(m, b)$ is first on the queue.
Effect:
        Remove first element from queue.
```

3.1.3 Alternating Bit Protocol

The ABP uses the following sender. It has inputs $SEND(m), m \in M$ and $RECEIVE2(b), b$ a Boolean, and outputs $SEND1(m, b), m \in M, b$ a Boolean. Its state consists of the following components: QS, (for "sender's queue"), which holds a finite sequence of elements of M, initially empty, and FS (for "sender's flag"), a Boolean, initially 1. The actions are:

```
$SEND(m), m \in M$
Effect:
        Add $m$ to end of $QS$.

$SEND1(m, b), m \in M, b$ a Boolean
Precondition:
        $m$ is first on $QS$.
        $b = FS$
Effect:
        None.

$RECEIVE2(b), b$ a Boolean
Effect:
        if $b = FS$ then
                [remove first element (if any) from $QS$;
                $FS := FS + 1 \, mod \, 2$]
```

The corresponding receiver has inputs $RECEIVE1(m, b), m \in M, b$ a Boolean, and outputs $RECEIVE(m), m \in M$ and $SEND2(b), b$ a Boolean. Its state consists of the following components. QR (for "receiver's queue"), which holds a finite sequence of elements of M, initially empty, and FR (for "receiver's flag"), a Boolean, initially 0. The actions are:

```
$RECEIVE(m), m \in M$
Precondition:
        $m$ is first on $QR$.
Effect:
        Remove first element from $QR$.

$RECEIVE1(m, b), m \in M, b$ a Boolean
Effect:
        if $b \neq FR$ then
                [add $m$ to end of $QR$;
                $FR := FR + 1 \, mod \, 2$]

$SEND2(b), b$ a Boolean
Precondition:
        $b = FR$
Effect:
        None.
```

3.1.4 Redundant Protocol

To prove the correctness of this protocol, I describe a redundant but much easier to understand variant of the protocol. In this variant, both the sender and receiver keep sequences of messages forever; furthermore, they tag the messages with positive integer sequence numbers and send them with those sequence numbers. The sender continues to send the same message just until it receives an acknowledgement with that message's tag; then it goes on to the next message in sequence. The receiver, on the other hand, keeps acknowledging the last message it has received, just until it gets the next message. It should be easy to prove that this works, using invariant assertions. Then the ABP can be proved to correspond to this protocol via a possibilities mapping, and so is correct as well.

More specifically, the redundant algorithm uses a slight modification of the channels used by the ABP - the only modification is that integer tags, rather than Boolean tags, are used. The redundant algorithm also has the same actions as the ABP (except that tag parameters are now positive integers). Its sender's state consists of the following components. SS (for "sender's sequence"), which holds an array of $(M \cup \perp)$ (where $\perp$ is a special "undefined" indicator, which is not an element of M), indexed by the positive integers, initially identically equal to $\perp$, IS (for "sender's integer"), a positive integer, initially 1, and LS (for "last message sent"), a nonnegative integer, initially 0.

The actions are:

```
$SEND(m), m \in M$
Effect:
    $LS := LS + 1$
    $SS(LS) := m$

$SEND1(m, i), m \in M, i$ a positive integer
Precondition:
    $SS(i) = m.$
    $i = IS$
Effect:
    None.

$RECEIVE2(i), i$ a positive integer
Effect:
    if $i = IS$ then
        $IS := IS + 1$
```

The corresponding receiver has a state consisting of the following components. SR (for "receiver's sequence"), which holds an array of $(M \cup \perp)$, indexed by the positive integers, initially identically equal to $\perp$, IR (for "receiver's integer"), an integer, initially 0, and LR (for "last message received"), an integer, initially 0. The actions are:

```
$RECEIVE(m), m \in M$
Precondition:
    $m = SR(LR + 1)$
Effect:
    $LR := LR + 1.$

$RECEIVE1(m, i), m \in M, i$ a positive integer
Effect:
    if $i = IR + 1$ then
        [$SR(i) := m;$
        $IR := IR + 1$]
```

```
$SEND2(i), i$ a positive integer
Precondition:
    $i = IR$
Effect:
    None.
```

It should be very easy (if I have not made any stupid mistakes) to show that the resulting algorithm correctly delivers messages, i.e., that the messages received are a subsequence of those sent. The actual ABP is somewhat harder to understand because it does not keep all this information explicitly; it removes redundancies. For example, it does not keep the complete sequences forever, but removes elements after they are no longer needed. More interestingly, it does not tag the messages in the channels and on the remaining queues with the integer indices, but only with bits.

For later use, I note here some basic invariants about the behavior of this redundant algorithm. (Call this algorithm H.)

Lemma 2 *The following statements are true about every reachable state of H.*

1. *Consider the sequence consisting of the indices in channel2, followed by IR, followed by the indices in channel1, followed by IS. The indices in this sequence are nondecreasing; furthermore, the difference between the first and last index in this sequence is at most 1.*

2. *If $IS = IR$, then $LS \geq IS$.*

3.1.5 Possibilities Mapping Proof

Now let L denote the ABP. We will show that L is correct by demonstrating a possibilities mapping from L to H. Note that such a mapping needs to be multivalued - it must augment the partial information contained in each of the two queues by filling in all earlier messages, and must fill in the integer values of tags only working from bits.

In particular, we say that a state u of H is in $f(s)$ for state s of L provided that the following conditions hold.

1. $s.QS$ is exactly the sequence of values of $u.SS$ corresponding to indices in the closed interval $[u.IS, u.LS]$.

2. $s.FS = u.IS mod\, 2$.

3. $s.QR$ is exactly the sequence of values of $u.SR$ corresponding to indices in the closed interval $[u.LR + 1, u.IR]$.

4. $s.FR = u.IR\, mod\, 2$.

5. Channel1 has the same number of messages in s and u. Moreover, for any j, if (m, i) is the j^{th} message in channel1 in u, then $(m, i\, mod\, 2)$ is the j^{th} message in channel1 in s.

6. Channel2 has the same number of messages in s and u. Moreover, for any j, if i is the j^{th} message in channel2 in u, then $i\, mod\, 2$ is the j^{th} message in channel2 in s.

Theorem 3 *f above is a possibilities mapping.*

3.1.6 Remarks

Consider the structure of the possibilities mapping f of this example. In going from H to L, unnecessary entries are garbage-collected, and integer tags are condensed to their low-order bits. The multiple values of the mapping f essentially "replace" this information. In this example, the correspondence between L and H can be described in terms of a mapping in the opposite direction - a (single-valued) projection from the state of H to that of L that removes information. Then f maps a state s of L to the set of states of H whose projections are equal to s. While this formulation suffices to describe many interesting examples, it does not always work, as will be seen in some of the subsequent examples in this paper.

Halpern and Zuck [10] outline a way of organizing the proof of the ABP that is similar to the organization I have described; their proofs are presented somewhat differently, however, using a formal theory of knowledge.

3.2 Transaction Processing

With Michael Merritt, Bill Weihl, Alan Fekete and Jim Aspnes [9, 2], I have done some work on describing and proving the correctness of locking- and timestamp-based algorithms for database concurrency control and recovery. Some of this work uses multivalued possibilities mappings in a way that is similar to their use for the ABP. That is, the proofs first show correctness of a simple and inefficient protocol that maintains a lot of extra information, and then shows that some particular protocols of interest implement the inefficient protocol in the formal sense of possibilities mappings.

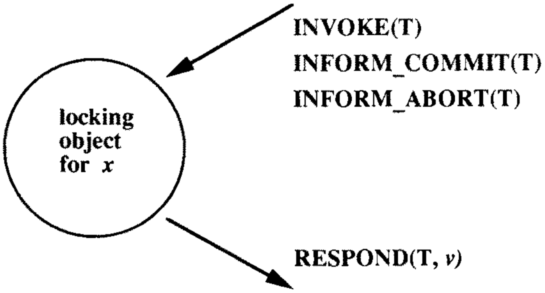
In this work, the advantage we gain from the mapping strategy is not only the decomposition of the proofs of particular algorithms; we also gain an advantage in generality. The high-level protocol is designed to work for arbitrary data types. The same high-level protocol can be used to prove the correctness of many specific low-level protocols that work (in more efficient ways) for particular data types such as read-write objects. (Halpern and Zuck [10] use mappings informally to get a similar generality for protocols related to the ABP.)

Here, I will just describe what we do for locking; our treatment of timestamps is similar. We develop a locking algorithm for nested transactions; in this model, transactions can have subtransactions, and subtransactions can have further subtransactions, and so on until the leaves of the transaction structure, which actually access data objects. The transaction nesting structure is a forest; we augment it with a dummy "root" transaction representing the "outside world", so that it becomes a tree. Transactions can commit (relative to their parents) or abort, and correctness is defined in terms of serializability among each group of siblings.

Our high-level algorithm allows objects of arbitrary data type. We describe this algorithm using a separate program (automaton) for each data object. The automaton for an object x does all the processing involving x. It receives invocations of accesses to x and decides on the appropriate responses to make. It maintains locks for x, together with any other necessary information such as temporary versions. It receives information about the commit and abort of transactions, in order to help it decide on the appropriate responses (and in order to help it decide when it can discard information and how to manipulate locks).

The complete high-level algorithm can be described as the composition of these object automata with other automata, e.g., automata for transactions and a message system automaton. In [9], we prove the correctness of this composition, with a fairly complicated proof. However, once we have proved this correctness for our high-level algorithm, we have a much easier job for some data-type-specific variants, since we can use possibilities mappings. For example, one very popular kind of locking algorithm is read-write locking. (In [9], we actually handle the slightly different case of read-update locking rather than read-write locking; the difference is that write accesses are constrained only to write the object with a predefined value, whereas update accesses can make arbitrary changes, depending on the object's prior value.) We can describe read-write locking as a similar composition, but with different object automata; in particular, we can use a read-write object automaton for each object x instead of a arbitrary data type object automaton for x. The read-write object automaton for x maintains less information than the corresponding arbitrary data type object automaton, but it can be shown to implement the former in terms of a possibilities mapping.

To be specific, the interface of an object automaton for x consists of input actions $INVOKE(T)$, $INFORM\_COMMIT(T)$, and $INFORM\_ABORT(T)$, and output action $RESPOND(T,v)$. Invocations and responses are for particular accesses to x (T locates the access within the transaction nesting structure); informs are for arbitrary transactions.



Let H denote the arbitrary data type automaton for x. It maintains "intentions lists", which are sequences of operations (i.e., (access,return value) pairs), for each transaction in the entire nesting structure, initially empty everywhere. The intentions list for T describes all the operations that are known to have occurred at descendants of T, and have committed up to T but not to its parent. It operates as follows. (Note: This is an informal paraphrase of the code in [9].)

```
INVOKE(T)
Effect:
    Record the invocation.

INFORM_COMMIT(T)
Effect:
    intentions(parent(T)) := intentions(parent(T))intentions(T)
    intentions(T) := empty

INFORM_ABORT(T)
Effect:
    intentions(U) := empty for all descendants U of T

RESPOND(T,v)
Precondition:
    T has been invoked and not yet responded to.
    (T,v) commutes with every (T',v') in intentions(U), where U is not an ancestor of T.
    total(T)(T,v) is a correct behavior of the underlying serial data object for x.
Effect:
    intentions(T) := intentions(T) (T,v)
```

Here, operations are said to "commute", roughly speaking, provided that in any situation in which both can be performed, they can be performed in sequence, in either order, and the result is the same in both cases.) Also, total(T) is defined to be the result of concatenating all the intentions lists for ancestors of T, in order from the root down. As I said earlier, our algorithm based on this object has a somewhat complicated proof.

Note that H maintains a good deal of explicit history information in its intentions lists. Now suppose that the underlying serial data object is a read-write object. In this case, we can improve the efficiency of this algorithm by maintaining more condensed, specially-tailored data structures in place of the intentions lists. In particular, we design a read-write object automaton L that keeps sets of read-lockholders and write-lockholders, plus a version of the underlying serial object for each write-lockholder. Initially, the root holds a write-lock, with the start state of the serial object as the associated version. The steps of L are as follows.

```
$INVOKE(T)$
Effect:
    Record the invocation.

$INFORM\_COMMIT(T)$
Effect:
    if T is a read-lockholder, then read-lockholders := read-lockholders $\cup\{parent(T)\} - \{T\}$
    if T is a write-lockholder, then
        [version(parent(T)) := version(T);
        write-lockholders := write-lockholders $\cup\{parent(T)\} - \{T\}$]

$INFORM\_ABORT(T)$
Effect:
    Remove all locks for descendants of T.

$RESPOND(T,v), T$ a read
Precondition:
    T has been invoked and not yet responded to.
    All write-lockholders are ancestors of T.
    v is the version associated with the least ancestor of T that is a write-lockholder.
Effect:
    read-lockholders := read-lockholders $\cup\{T\}$.

$RESPOND(T,v), T$ a write
Precondition:
    T has been invoked and not yet responded to.
    All read-lockholders and write-lockholders are ancestors of T.
    v = "nil"
Effect:
    write-lockholders := write-lockholders $\cup\{T\}$
    version(T) := v
```

The correctness of the algorithm using L follows from that of the algorithm using H once we demonstrate a possibilities mapping f from L to H. The mapping says the following (paraphrased): $u \in f(s)$ exactly if

1. u and s record that the same set of transactions has been invoked.

2. u and s record that the same set of transactions has been responded to.

3. s.read-lockholders is exactly the set of transaction names T such that u.intentions(T) contains a read access.

4. s.write-lockholders is exactly the set of transaction names T such that u.intentions(T) contains a write access (together with the root).

5. For every T, evaluating total(T) in u results in the value version(T'), where T' is the least ancestor of T in write-lockholders.

Although a read-write serial object is a special case of an arbitrary data type serial object, note that the read-write object automaton L is not really a special case of the arbitrary data type object automaton: the data structures are different, and f expresses a nontrivial correspondence between the different structures. However, the behaviors of the two objects correspond very closely, as shown by the fact that there is a possibilities mapping between them. Note that the mapping f is multivalued, since the summary version and lock-holder information maintained by the read-write object automaton does not (in general) allow a unique reconstruction of the intentions list information in the arbitrary data type object automaton.

For this example, as for the ABP, the possibilities mapping can be described as the inverse of a projection mapping states of H to states of L, but here that seems like a bit of an accident. For, the read-update objects described in [9] have a similar description and proof, but the mapping used there can associate more than one state of L to a state of H. (This is because the serial object state produced by a sequence of operations might not be uniquely determined.)

Although we have not worked this out, it should be possible to describe optimized variants of our high-level algorithm for other specific data types besides read-write objects and read-update objects. I expect that such optimizations should also be verifiable using possibilities mappings to our high-level objects.

Our treatment of timestamp-based concurrency control algorithms in [2] is analogous to our treatment of locking. Namely, we first present an algorithm for arbitrary data types (based on that of Herlihy [11], but extended to nested transactions); we present this using an automaton for each object. Then we present the specially-tailored algorithm of Reed [23] for read-write objects; correctness of this algorithm is proved using possibilities mappings to the algorithm for arbitrary data types.

3.3 Garbage Collection

With Paul Leach, Liza Martin and Joe Pato at Apollo Computer, I have made use of multivalued possibilities mappings to design and prove correctness of an algorithm for replicated data management. Again, the use involves decomposing the algorithm using a higher-level and less efficient algorithm. I'll just sketch the ideas very roughly and informally here.

The setting we consider involves a replicated data management algorithm in which updates to data objects can be issued at arbitrary nodes. We assume a timestamp mechanism that totally orders all updates produced anywhere in the system. Here I assume for simplicity that all the updates are overwrites. In this setting, nodes exchange information about all the updates that have been generated, so (if the network stays connected), all nodes eventually find out about all updates. (Other transactions, which I will not discuss here, read the data produced by this algorithm and take actions based on it.) We assume that the network is dynamic, i.e., that nodes can be added to and removed from the system during execution. The setting is similar to those considered in [6, 24, 8]. In order to determine whether an incoming update should supersede an already-known update for the same object, a node must maintain some timestamp information for known updates. Because it would be inefficient to keep the complete history of known updates, nodes summarize this history information in a "checkpoint state" that contains summarized values (with associated timestamps) for all objects. But because of the way nodes exchange information about updates, they also maintain some incremental information; the data maintained by each node is thus a combination of a checkpoint state and a log of recent updates. The complete algorithm can be proved correct by standard techniques (basically, the safety properties to be proved say that each node is as up-to-date about the updates originating at each other node as it thinks it is).

Now, the actual system has another complication - we would like to *garbage collect* information about objects whose latest update is an "overwrite(x,nil)", i.e., a "delete(x)". It would be nice not to have to record this update forever (with its associated timestamp). But it is necessary to record it for a while, in order to correctly determine its timestamp ordering with respect to incoming updates of x. We need a criterion that tells us when we may garbage collect such information without affecting the behavior of the algorithm. It is quite nontrivial to determine such a rule, especially in the case we consider, where nodes can be added or removed during computation; e.g., one must ensure that updates issued by newly-added nodes can never get ordered incorrectly with respect to the garbage collected updates.

We have designed an algorithm, L, that includes a local criterion that says when it is safe for a node to garbage collect a delete update. The final algorithm appears to be fairly complicated. It turns out that the best way to understand it is by means of a possibilities mapping from L to the original non-garbage collected algorithm, H. Starting with a state s of L, this possibilities mapping obtains corresponding states of H by adding in information about the missing updates in all possible ways that are consistent with the current remaining state. Of course, there may be many ways to add in such information; thus, the mapping is multivalued. With this correspondence, the correctness proof for the algorithm with garbage collection seems fairly straightforward (although it seemed to us to be quite difficult otherwise).

Note that unlike the two previous examples, this example uses a correspondence that is not expressible as a projection from H to L - here, several L states could also be related to a single H state. That is, given a state of the non-garbage collected algorithm, it is possible to choose the information to garbage collect in many different ways. (Choices of updates to garbage collect are made locally at individual nodes, and asynchronously with respect to the choices made at other nodes.) Thus, in this case, the correspondence is multivalued in both directions.

3.4 Remarks

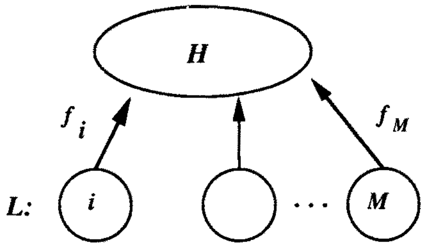
The idea of decomposing algorithms using unoptimized but simpler variants and possibilities mappings seems to be a very generally useful technique. It is useful for algorithms that perform explicit garbage collection, and also for algorithms such as the ABP, that simply omit unused portions of the simpler information. I think that this idea can be pushed much further in the area of distributed algorithms; many clever and complicated algorithms should have decompositions using simpler variants containing extra information. For example, I wonder whether the many complicated algorithms for implementing atomic registers (e.g., [25, 15]) can be verified in this way. It seems to me that at least Bloom's special-case algorithm [7] should have a nice proof in terms of integer tags rather than bits; perhaps a similar strategy will work for other atomic register algorithms.

Note that all the proofs I have given in this section could be recast in terms of history variables added to the low-level algorithms and single-valued rather than multivalued mappings. Thus, although the proofs in terms of multivalued mappings seem more natural to me, there is no theorem that says that multivalued mappings are necessary.

4 Distribution

Another way that multivalued mappings arise is in describing algorithm *distribution* rather than optimization. In the setting I have in mind, a centralized algorithm H (one not explicitly decomposed into nodes and a message system) is first shown to solve the problem of interest. A related distributed algorithm (one that is described explicitly as the composition of a number of node automata and one or several automata representing the message system) L is then given; we want to show that L is correct by showing that it implements H (that is, that all its behaviors are behaviors of H).

The basic strategy is again to define a mapping f from states of L to sets of states of H. However, in this case it may be helpful to define f in terms of a collection of "component mappings" f_i, one for each component of L. Thus, each node i has an abstraction mapping that maps each of its states to a set of states of H, and likewise the message system M, (or each separate message channel M, if there are several) has a mapping f_M from its states to a set of states of H.



These component mappings have an interesting interpretation - e.g., the mapping f_i for node i describes, in terms of i's state, the "possible states" of the centralized algorithm H, as far as i can tell. Thus, in a sense, this mapping can be thought of as giving the "local knowledge" that i has, of the state of the centralized algorithm.

Under certain conditions ([19, 14]) these component mappings can be "composed" to yield a possibilities mapping from each entire state of L to a set of states of H, representing the "possible states" of H, as far as *any* of the components can tell. Formally, the value of $f(s)$, for a state s of L, is exactly the *intersection* of the values of $f_i(s_i)$, where s_i is component i's state in s. That is, the states that are possible for all the components are just those that are in the intersection of the sets that are possible for all the individual components. In other words, the intersection of the local knowledge of all the components (including the message system) is the global knowledge of the system.

Note that the mapping f might or might not be multivalued, but the individual f_i almost certainly will be. This is because in a typical distributed system, no individual node knows everything about the global state.

This decomposition can sometimes be used to simplify algorithm proofs (at least, it has worked in one substantial case I have tried). This case again arises in transaction processing. In [19], I describe a locking algorithm similar to the read-write locking algorithm I described earlier in this paper, by first giving a centralized description. The algorithm H keeps global information such as the sets of transactions that have been *created*, *committed* and *aborted*, plus certain "version mappings" that keep versions of various objects on behalf of various transactions. In the distributed algorithm, each node keeps part of this information: it knows *some* of the created, committed and aborted transactions, and some of the versions. The mapping f_i for a node i just adds in unspecified other transactions and versions to these sets, in addition to the ones locally known. I prove correctness of H directly, then combine the f_i as described above to get a possibilities mapping f and prove correctness of L using this mapping.

More work is needed to determine how generally useful this proof structure is.

5 Proving Time Bounds

My final example arises in my very recent work (joint with Hagit Attiya) on timing-based algorithms. The idea is to use multivalued mappings for reasoning about upper and lower bounds on time for such algorithms. Although this work is still preliminary, I think that its use of multivalued mappings is quite interesting.

5.1 Overview

So far, abstraction mappings (and assertional reasoning in general) have been used primarily to prove correctness properties of sequential algorithms and synchronous and asynchronous concurrent algorithms. It would also be nice to use these techniques to prove properties of concurrent algorithms whose operation depends on time, e.g., that have a clock that ticks at an approximately predictable rate. Also, the kinds of properties usually proved using mappings are "ordinary" safety properties; it would also be nice to use similar methods for proving timing properties (upper and lower bounds on time) for algorithms that have timing assumptions.

Here, I show how abstraction mappings can be used to prove timing properties of timing-dependent concurrent algorithms. I'll focus on a trivial example, an algorithm consisting of two concurrently-operating components, which we call a *clock* and a *manager*. The clock ticks at an approximately known rate. The manager monitors the clock ticks, and after a certain number have occurred, it issues a *GRANT* (of a resource). It then continues counting ticks; whenever sufficiently many have occurred since the previous *GRANT* event, the manager issues another *GRANT*. We wish to give a careful proof of upper and lower bounds on the amount of time prior to the first *GRANT* event and in between each successive pair of *GRANT* events.

In order to state and prove such results, we need to extend the I/O automaton model to incorporate time in the assumptions and in the conditions to be proved. Fortunately, this has been done for us: Modugno, Merritt and Tuttle [21] define a suitable extension call the *timed automaton* model. In that model, an algorithm with timing assumptions is described as an *I/O automaton* together with a *boundmap* (a construct used to give a formal description of the timing assumptions). This automaton and boundmap generate a set of *timed executions* and a corresponding set of *timed behaviors*. We use timed automata to define the basic assumptions about the underlying system, to describe the algorithm, and to carry out a correctness proof.

In order to carry out an assertional proof about time, we need to reformulate some of the definitions of [21] so that information about time is explicitly included in the algorithm's state. In order to include *assumptions* about time in the state, we use the construction given in [3], of an automaton $time(A)$ for a given timed automaton A. The automaton $time(A)$ is an ordinary I/O automaton (not a timed automaton) whose state includes predictive information describing the first and last times at which various basic events can next occur; this information is derived from the given boundmap. The I/O automaton $time(A)$ is related to the original timed automaton A in that a certain subset of the behaviors of $time(A)$ is exactly equal to the set of timed behaviors of A.

We also require a formal way of describing the timing *requirements* to be proved for our algorithm. In order to do this, we augment A to another I/O automaton B which we call the *performance machine* this time building in predictive information about the first and last times at which certain events of interest (e.g., *GRANT* events) can next occur. Then the problem of showing that the given algorithm satisfies the timing requirements is reduced to that of showing that any behavior of the automaton $time(A)$ is also a behavior of B. *We do this by exhibiting a mapping from $time(A)$ to B.* This mapping turns out to be multivalued; in fact, it is in the form of a set of inequalities!

In the remainder of this section, I give more details.

5.2 Formal Model

Here I describe timed automata and the construction of $time(A)$.

5.2.1 Timed Automata

Recall that an I/O automaton consists of *actions*, *states*, *start states*, *steps*, and a fifth component that is a *partition* of the locally controlled (output and internal) actions into equivalence classes. The last is generally used for fairness and liveness, and so I have not used it so far in this paper, but I will need it now. The partition groups actions together that are to be thought of as under the control of the same underlying process.

In [21], the I/O automaton model is augmented to include timing properties as follows. A *timed automaton* is an I/O automaton with an additional component called a *boundmap*. The boundmap associates a closed interval of the nonnegative reals (possibly including infinity, but where the lower bound is not infinity and the upper bound is not 0) with each class in the automaton's partition. This interval represents the range of possible lengths of time between successive times when the given class gets a chance to perform an action. Let $b_\ell(C)$ and $b_u(C)$ denote the lower and upper bounds, respectively, assigned by the boundmap b to class C.

Now I describe how a timed automaton executes. A *timed sequence* is a sequence of alternating states and (action,time) pairs; the times are required to be nondecreasing, and if the sequence is infinite then the times are also required to be unbounded. Such a sequence is said to be a *timed execution* of a timed automaton A provided that the result of removing the time components is an execution of the ordinary I/O automaton underlying A, and the following conditions hold, for each class C of the partition of A and every i.

1. Suppose $b_u(C) \neq \infty$. If some action in C is enabled in a_i and either $i = 0$ or no action in C is enabled in a_{i-1} or π_i is in C, then there exists $j > i$ with $t(j) \leq t(i) + b_u(C)$ such that either π_j is in C or no action of C is enabled in a_j.

2. If some action in C is enabled in a_i and either $i = 0$ or no action in C is enabled in a_{i-1} or π_i is in C, then there does not exist $j > i$ with $t(j) < t(i) + b_\ell(C)$ and π_j in C.

The first condition says that, starting from when an action in C occurs or first gets enabled, within time $b_u(C)$ either some action in C occurs or there is a point at which no such action is enabled. The second condition says that, again starting from when an action in C occurs or first gets enabled, no action in C can occur before time $b_\ell(C)$ has elapsed.

Definitions for composition of timed automata to yield another timed automaton are given in [21]. We model real-time systems as compositions of timed automata.

5.2.2 The Automaton *time*(A)

Given any timed automaton A with boundmap b, we now show how to define the corresponding ordinary I/O automaton *time*(A). This new automaton has the timing restrictions of A built into its state, in the form of predictions about when the next event in each class will occur.

The automaton *time*(A) has actions of the form (π, t), where π is an action of A and t is a nonnegative real number. Each of its states consists of a state of A, augmented with a time called *Ctime* and, for each class C of the partition, two times, *Ftime*(C) and *Ltime*(C). *Ctime*, (the "current time") represents the time of the last preceding event, initially 0. The *Ftime*(C) and *Ltime*(C) components represent, respectively, the first and last times at which an action in class C is scheduled to be performed (assuming it stays enabled). (We use record notation to denote the various components of the state of *time*(A); for instance, *s.automaton_state* denotes the state of A included in state s of *time*(A).) More precisely, each initial state of *time*(A) consists of an initial state s of A, plus $Ctime = 0$, plus values of $Ftime(C)$ and $Ltime(C)$ with the following properties. If there is an action in C enabled in s, then $s.Ftime(C) = s.Ctime + b_\ell(C)$ and $Ltime(C) = s.Ctime + b_u(C)$. Otherwise, $Ftime(C) = 0$ and $Ltime(C) = \infty$.

Others have proposed building timing information into the state (e.g., [26]); our work differs in the particular choice of information to use - predictive information, giving upper and lower bounds for each automaton class.

The following definitions capture formally what it means for the given timing assumptions to be respected by *time*(A). If (π, t) is an action of *time*(A), then $(s', (\pi, t), s)$ is a step of *time*(A) exactly if the following conditions hold.

1. $(s'.automaton\_state, \pi, s.automaton\_state)$ is a step of A.

2. $s'.Ctime \leq t = s.Ctime$.

3. If π is a locally controlled action of A in class C, then
   (a) $s'.Ftime \leq t \leq s'.Ltime.$
   (b) if some action in C is enabled in $s.automaton\_state$, then $s.Ftime(C) = t + b_\ell(C)$ and $s.Ltime(C) = t + b_u(C)$, and
   (c) if no action in C is enabled in $s.automaton\_state$, then $s.Ftime(C) = 0$ and $s.Ltime(C) = \infty$.

4. For all classes D such that π is not in class D,
   (a) $t \leq s'.Ltime(D)$,
   (b) if some action in D is enabled in $s.automaton\_state$ and some action in D is enabled in $s'.automaton\_state$ then $s.Ftime(D) = s'.Ftime(D)$ and $s.Ltime(D) = s'.Ltime(D)$, and
   (c) if some action in D is enabled in $s.automaton\_state$ and no action in D is enabled in $s'.automaton\_state$ then $s.Ftime(D) = t + b_\ell(D)$ and $s.Ltime(D) = t + b_u(D)$, and
   (d) if no action in D is enabled in $s.automaton\_state$, then $s.Ftime(D) = 0$ and $s.Ltime(D) = \infty$.

Property 3 describes the conditions on the particular class C (if any) containing the action π - basically, that the time for the new action should be in the appropriate interval for the class. New scheduled times are also set for C, in case an action in C is enabled after this step. Property 4 describes conditions involving each other class D. The most interesting is property 4(a), which ensures that the action in C does not occur if D has an action that must be scheduled first.

Now I state how the behaviors of $time(A)$ are related to the timed behaviors of A. Define the *complete executions* of $time(A)$ to be those executions α of $time(A)$ that satisfy one of the following conditions.

1. α is infinite and the time components of the actions in α are unbounded, or

2. α is finite and no locally controlled action of $time(A)$ is enabled in the final state of α.

The *complete schedules* and *complete behaviors* of $time(A)$ are defined to be the schedules and behaviors, respectively, of complete executions of $time(A)$.

The timed executions of a timed automaton A are closely related to the complete executions of the corresponding I/O automaton $time(A)$. In particular, what we use is:

Theorem 4 *The set of timed behaviors of A is the same as the set of complete behaviors of $time(A)$.*

This theorem implies that properties of timed behaviors of a timed automaton A can be proved by proving them about the set of complete behaviors of the corresponding I/O automaton $time(A)$. The latter task is more amenable to treatment using assertional techniques, because of the fact that timing information is built into the state of $time(A)$.

We apply the $time(A)$ construction to the timed automaton A modeling the entire system.

5.2.3 Strong Possibilities Mappings

The work in this section requires a slightly strengthened notion of possibiities mapping - one that preserves the correspondence between *all* actions (internal as well as external). Let L and H be automata with the same actions, and let f be a mapping from states of L to sets of states of H. The mapping f is a *strong possibilities mappings from L to H* provided that the following conditions hold:

1. For every start state s_0 of L, there is a start state u_0 of H such that $u_0 \in f(s_0)$.

2. If s' is a reachable state of A, $u' \in f(s')$ is a reachable state of H, and (s', π, s) is a step of L, then there is a step (u', π, u) of H such that $u \in f(s)$.

The difference between this definition and the ordinary definition for possibilities mappings is in the second condition, where the actions are required to correspond exactly. Now recall that the schedules of an automaton include all its actions.

Lemma 5 *If there is a strong possibilities mapping from L to H, then all schedules of L are also schedules of H.*

5.3 The Algorithm

The algorithm consists of two components, a *clock* and a *manager*. The *clock* has only one action, the output *TICK*, which is always enabled, and has no effect on the clock's state. It can be described as the particular one-state automaton with the following steps.

```
TICK
Precondition:
      true
Effect:
      none
```

The boundmap associates the interval $[c_1, c_2]$ with the single class of the partition. This means that successive *TICK* events will occur with intervening times in the given interval.

The manager has input action *TICK*, output action *GRANT* and internal action *ELSE*. The manager waits a particular number k of clock ticks before issuing each *GRANT*, counting from the beginning or from the last preceding *GRANT*. The manager's state has one component: TIMER, holding an integer, initially k.

The manager's algorithm is as follows: (We assume that $k > 0$).

```
TICK
Effect:
      TIMER := TIMER -1

GRANT
Precondition:
      TIMER $\leq 0$
Effect:
      TIMER := $k$

ELSE
Precondition:
      TIMER $> 0$
Effect:
      none
```

Notice that *ELSE* is enabled exactly when *GRANT* is not enabled. The effect of including the *ELSE* action is to ensure that the automaton continues taking steps at its own pace, at approximately regular intervals. Thus, in the situation we are modeling, when the *GRANT* action's precondition becomes satisfied, the action doesn't occur instantly - the action waits until the automaton's next local step occurs. [1]

[1] An alternative situation to model would be an interrupt-driven model in which the action is triggered to occur whenever its precondition becomes true; the action should then occur shortly thereafter; this situation could be modeled by omitting the *ELSE* action. The two automata have slightly different timing properties. In this paper, I only consider the first assumption.

The partition groups the *GRANT* and *ELSE* actions into a single equivalence class, with which the boundmap associates the interval $[0, l]$. We assume that $c_1 > l$. [2] Now we fix L to be the timed automaton which is the composition of the clock and manager.

I now consider the automaton $time(L)$, constructed as described in Section 5.2. In this case, the construction adds the following components to the state of L: *Ctime*, *Ftime*(*TICK*), *Ltime*(*TICK*), *Ftime*(*LOCAL*), and *Ltime*(*LOCAL*). The latter two represent the times for the partition class consisting of *GRANT* and *ELSE*.

Lemma 6 *All complete executions (and therefore all complete schedules) of* $time(L)$ *are infinite.*

This is essentially because the clock keeps ticking forever. This lemma tells us that for this example we do not have to worry about the case where executions are finite - we can assume that we have infinite executions in which (because of the definition of completeness) the timing component is unbounded.

5.4 The Performance Automaton

We wish to show that all the timed behaviors of L satisfy certain upper and lower bounds on the time for the first *GRANT* and the time between consecutive pairs of *GRANT* events. More precisely, we wish to show the following, for any timed behavior γ of B:

1. There are infinitely many *GRANT* events in γ.
2. If t is the time of the first *GRANT* event in γ, then $k \cdot c_1 \leq t \leq k \cdot c_2 + l$.
3. If t_1 and t_2 are the times of any two consecutive *GRANT* events in γ, then

$$k \cdot c_1 - l \leq t_2 - t_1 \leq k \cdot c_2 + l.$$

We let P denote the set of sequences of $(action, time)$ pairs satisfying the above three conditions. By the earlier characterization, Theorem 4, it suffices to show that all complete behaviors of $time(L)$ are in P.

I have already shown how to describe timing assumptions by building time information into the state. Now I show how to give a similar description for the timing properties to be proved. Thus, we specify P in terms of another I/O automaton, which we call the *performance automaton*. Namely, define a new I/O automaton H by augmenting $time(L)$ with two new components: $Ftime(GRANT)$ and $Ltime(GRANT)$. These are designed to represent the first and last times, respectively, that a *GRANT* event might occur. They are maintained as follows.

1. Initially,
   (a) $Ftime(GRANT) = k \cdot c_1$, and
   (b) $Ltime(GRANT) = k \cdot c_2 + l$.
2. For each step $(s', (GRANT, t), s)$ of H,
   (a) $s'.Ftime(GRANT) \leq t \leq s'.Ltime(GRANT)$.
   (b) $s.Ftime(GRANT) = t + k \cdot c_1 - l$ and $s.Ltime(GRANT) = t + k \cdot c_2 + l$.
3. For each step $(s', (\pi, t), s)$ of H, where $\pi = TICK$ or $ELSE$,
   (a) $t \leq s'.Ltime(GRANT)$.
   (b) $s.Ltime(GRANT) = s'.Ltime(GRANT)$ and

[2] Again, a different assumption would change the timing analysis.

(c) $s.Ftime(GRANT) = s'.Ftime(GRANT)$.

In addition, the other components of the state are maintained just as in the definitions of *time*(L).

This automaton simply builds in explicitly the time bounds to be proved. The initial conditions build in the time bounds that are supposed to hold for the first *GRANT*, and condition 2b builds in the time bounds that are supposed to hold for all the subsequent *GRANT* actions. Conditions 2a and 3a ensure that nothing happens strictly after the latest time at which a *GRANT* is supposed to occur. Condition 2a also ensures that the *GRANT* does not occur too soon.

The following lemma gives the relationship we need between the behaviors of H and the condition P. (Note that the behaviors of H and the sequences in P both consist of elements that are pairs, an action of L together with a time.)

Lemma 7 *Let β be an infinite schedule of H in which the time component is unbounded. Then $beh(\beta) \in P$.*

Note that the performance machine H is a somewhat ad hoc description of the particular timing properties to be proved for our particular algorithm. We are currently working on generalizing the treatment of performance machines.

5.5 Proof

Now we sketch how to prove that all timed behaviors of L are in P, as needed. First, we show that all behaviors of *time*(L) are also behaviors of the performance machine H, using a strong possibilities mapping. Namely, we define a mapping f so that a state u of H is in the image set $f(s)$ exactly if the following conditions hold.

1. If $s.TIMER > 0$ then

   (a) $u.Ltime(GRANT) \geq s.Ltime(TICK) + (s.TIMER - 1)c_2 + l$, and

   (b) $u.Ftime(GRANT) \leq s.Ftime(TICK) + (s.TIMER - 1)c_1$.

2. If $s.TIMER = 0$ then

   (a) $u.Ltime(GRANT) \geq s.Ltime(LOCAL)$, and

   (b) $u.Ftime(GRANT) \leq s.Ctime$.

Thus, in this case the mapping takes the form of inequalities giving upper and lower bounds for the time of the next *GRANT* event, in terms of the values of the variables in the state of *time*(L). For example, condition 1a says that (in case the timer is positive), the upper bound that is being proved on the time for the next *GRANT* is any value that is *at least as great* as the latest time for the next *TICK*, plus the number of remaining *TICK* events that will be counted times the maximum time they might take, plus the maximum time for a local step. This makes sense because the quantity on the right-hand side of the inequality is itself an upper bound on the time until the next *GRANT*; if the performance machine designates anything at least as great as this expression as the upper bound to be proved, then it should be possible to prove that the algorithm simulates the performance machine (i.e., that it respects the upper bound described by that machine).

Symmetrically, condition 1b says that (in case the timer is positive) the lower bound that is being proved on the time for the next *GRANT* is any value that is *at most as great* as the earliest time for the next *TICK*, plus the number of remaining ticks that will be counted times the minimum time they might take, (plus the minimum time for a local step, which is 0). This makes sense because the quantity on the right-hand side of the inequality is itself a lower bound on the time until the next *GRANT*; if the performance machine designates anything at most as great as this expression as the lower bound to be proved, then it should be possible to prove that the algorithm simulates the performance machine (i.e., that it respects the lower bound described by that machine).

In case the timer is 0, the upper bound that is being proved on the time for the next *GRANT* is any value that is at least as great as the latest time for the next local step. Again, the quantity on the right-hand side of the inequality is an upper bound on the time until the next *GRANT*, so that if the performance machine designates anything at least that large, it should be possible to prove that the algorithm simulates the performance machine. Also for this case, the lower bound that is being proved is any value that is at most as great as the earliest time for the next local step, which is the current time.

This mapping is obviously multivalued, because it is described in terms of inequalities. The inequalities express the fact that *any* sufficiently large number (with respect to the values of the variables in the state of *time*(L)) should be provable as an upper bound for the time for the next *GRANT*, and any sufficiently small number should be provable as a lower bound. [3] [4] We can now show:

Lemma 8 *The mapping f is a strong possibilities mapping.*

Lemmas 5 and 8 yield the following corollary.

Corollary 9 *All schedules of time(L) are schedules of H.*

Now I can put the pieces together.

Theorem 10 *All timed behaviors of L are in P.*

Proof: Let γ be a timed behavior of L. Then by Theorem 4, γ is a complete behavior of *time*(L). Let β be a complete schedule of *time*(L) such that $\gamma = beh(\beta)$. By Lemma 6, β is infinite, and by the definition of completeness for infinite executions, the time components of β are unbounded. Lemma 9 implies that β is also a schedule of H. Since β is an infinite schedule of H in which the time components are unbounded, Lemma 7 implies that $beh(\beta) = \gamma$ is in P. ■

Note that in this case, the possibilities mapping technique yields all the correctness properties we require - including both safety and liveness properties. Certain timing properties are safety properties, e.g., lower bounds, and upper bounds of the form "if time grows sufficiently large, then certain events must occur". These can be proved using possibilities mappings in much the same way as any other safety properties. But when such conditions are combined with the property that all complete executions are infinite and our assumption that the time in infinite timed executions is unbounded (so that "time continues to increase without bound"), they actually imply that the events in question must eventually occur. Thus, liveness properties of the kind that say "certain events must occur" also follow from the mapping technique.

6 Conclusions

In this paper, I have tried to illustrate several situations in which multivalued abstraction mappings are useful in algorithm correctness proofs. Multivalued mappings are useful in cases where one algorithm can be described as an optimized (e.g., garbage collected) version of another algorithm, or where a single high-level algorithm admits several specialized implementations tailored for different

[3] If we simply replaced the inequalities with equations, the resulting mapping would not be a possibilities mapping. For example, suppose that a clock tick occurs within less than the maximum c_2. Then the right-hand side expression in 1a would evaluate after the step to an earlier time than before the step. On the other hand, the corresponding step in the performance machine would *not* change the value of *Ltime*(*GRANT*); the correspondence thus would not be preserved.

[4] It seems possible to use a single-valued mapping for this example by complicating the definition of the performance machine; however, since the performance machine is serving as the problem specification, that does not seem like a good idea.

situations. They are also useful in relating distributed algorithms to centralized variants, and in proving time bounds.

Work remains to be done in exploiting these techniques further. I believe it will be possible to decompose the proofs of many other complicated concurrent algorithms by expressing the algorithms as optimized versions of simpler algorithms, or as special cases of more general algorithms, or as distributed versions of centralized algorithms. It remains to discover such structure and express it in terms of mappings.

The use of mappings for time analysis is new, and should be tried on more (and larger) examples. It remains to see how this technique combines with other methods for time analysis such as methods based on bounded temporal logic [5] or recurrence equations [16]. I hope I have made the point I have tried to make: that multivalued mappings are sufficiently useful that any useful formal framework incorporating abstraction mappings should permit them to be multivalued.

Acknowledgements:

I would like to thank John Leo for working out the algorithm distribution techniques, and for reading and commenting on an earlier verion of the manuscript.

A Proof

Proof: By induction. For the base, let s be the start state of L and u the start state of H. First, $s.QS$ is empty. Also, $[u.IS, u.LS] = [1,0]$, which implies that $s.QS$ is equal to the appropriate (empty) portion of $u.SS$. Second, $s.FS = 1 = u.IS \, mod \, 2$. Third, $s.QR$ is empty, and $[u.LR+1, u.IR] = [1,0]$, which implies that $s.QR$ is equal to the appropriate portion of $u.SR$. Fourth, $s.FR = 0$ and $u.IR = 0$, which is as needed. Fifth and sixth, both channels are empty.

Now show the inductive step. Suppose (s', π, s) is a step of L and $u' \in f(s')$. We consider cases based on π.

1. $\pi = SEND(m)$

   Choose u to be the unique state such that (u', π, u) is a step of H. We must show that $u \in f(s)$. The only condition that is affected by the step is the first; thus, we must show that $s.QS$ is exactly the sequence of values of $u.SS$ corresponding to indices in the closed interval $[u.IS, u.LS]$. But $s.QS = s'.QSm$. Since $u' \in f(s')$, $s'.QS$ is just the sequence of values from $u'.SS$, from indices $u'.IS$ to $u'.LS$. Since the step of H increases LS by 1 and puts m in the new position, we have the needed equation.

2. $\pi = RECEIVE(m)$

   Since π is enabled in s', m is the first value on $s'.QR$. Since $u' \in f(s')$, $m = u'.SR(u'.LR+1)$, which implies that π is enabled in u'. Now choose u to be the unique state such that (u', π, u) is a step of H. All conditions are unaffected except for the third, that $s.QR$ is exactly the sequence of values of $u.SR$ corresponding to indices in the closed interval $[u.LR+1, u.IR]$. Now, $s.QR$ is the same as $s'.QR$ with the first element removed. Since $u' \in f(s')$, we have that $s'.QR$ is just the sequence of values from $u'.SR$, from indices $u'.LR+1$ to $u'.IR$. Since the step of H increases LR by 1, we have the needed equation.

3. $\pi = SEND1(m, b)$

   Since π is enabled in s', $b = s'.FS$ and m is the first element on $s'.QS$. Let i be the integer $u'.IS$. Since $u' \in f(s')$, the first element in $s'.QS$ is the same as the $u'.IS$ entry in $u'.SS$; that is, $u'.SS(i) = m$. It follows that $\bar{\pi} = SEND1(m, i)$ is enabled in u'.

   Now choose u so that $(u', \bar{\pi}, u)$ is a step of H and such that this step puts a message in channel1 exactly if the step (s', π, s) does. We must show that $u \in f(s)$. The only interesting condition is the fifth; that is, we must show that channel1 has the same number of messages in s and u. Moreover, for any j, if (m, k) is the j^{th} message in channel1 in u, then $(m, k \, mod \, 2)$ is the j^{th} message in channel1 in s. The only interesting case is where both steps cause a message to be put into the channel. Then the message value in both cases is m, but the tag is b for algorithm L and i for H. It remains to show that $b = i mod 2$. But $b = s'.FS$ and $i = u'.IS$. Since $u' \in f(s')$, we have $s'.FS = u'.ISmod \, 2$, which implies the result.

4. $\pi = RECEIVE1(m, b)$

   Since π is enabled in s', (m, b) is the first element in channel1 in s'. Since $u' \in f(s')$, (m, i) is the first element in channel1 in u', for some integer i with $b = i mod 2$. Let $\bar{\pi} = RECEIVE1(m, i)$; then $\bar{\pi}$ is enabled in u'. Let u be the unique state such that $(u', \bar{\pi}, u)$ is a step of H. We must show that $u \in f(s)$.

   All conditions except for the third, fourth and fifth are unchanged. It is easy to see that the fifth is preserved, since each of π and $\bar{\pi}$ simply removes the first message from channel1.

   Suppose first that $b = s'.FR$. Then the effects of π imply that the receiver state in s is identical to that in s'. Now, since $u' \in f(s')$, $s'.FR = u'.IRmod \, 2$; since $b = i mod 2$, this case must have $i \neq u'.IR + 1$. Then the effects of $\bar{\pi}$ imply that the receiver state in u is identical to that in u'. It is immediate that the third and fourth conditions hold.

So now suppose that $b \neq s'.FR$. The invariant above for H implies that either $i = u'.IR$ or $i = u'.IR + 1$. Since $b = i\,mod\,2$ and (since $u' \in f(s')$) $s'.FR = u'.IRmod\,2$, this case must have $i = u'.IR + 1$. Then $u.IR = u'.IR + 1$ and $s.FR = s'.FR + 1\,mod\,2$, preserving the fourth condition. Also, $u.SR$ is the same as $u'.SR$ except that the entry with index $u.IR$ is set equal to m; moreover, $s.QR$ is the same as $s'.QR$ except that m is added to the end. It follows that the third condition is preserved.

5. $\pi = SEND2(b)$

   Since π is enabled in s', $b = s'.FR$. Let i be the integer $u'.IR$. Let $\bar{\pi} = SEND2(i)$; clearly, $\bar{\pi}$ is enabled in u'.

   Now choose u so that $(u', \bar{\pi}, u)$ is a step of H and such that this step puts a message in channel2 exactly if the step (s', π, s) does. We must show that $u \in f(s)$. The only interesting condition is the sixth; that is, we must show that channel2 has the same number of messages in s and u. Moreover, for any j, if k is the j^{th} message in channel2 in u, then $kmod\,2$ is the j^{th} message in channel2 in s. The only interesting case is where both steps cause a message to be put into the channel. Then the tag is b for algorithm L and i for H. It remains to show that $b = imod\,2$. But $b = s'.FR$ and $i = u'.IR$. Since $u' \in f(s')$, we have $s'.FR = u'.IRmod\,2$, which implies the result.

6. $\pi = RECEIVE2(b)$

   Since π is enabled in s', b is the first element in channel2 in s'. Since $u' \in f(s')$, i is the first element in channel2 in u', for some integer i with $b = imod\,2$. Let $\bar{\pi} = RECEIVE2(i)$; then $\bar{\pi}$ is enabled in u'. Let u be the unique state such that $(u', \bar{\pi}, u)$ is a step of H. We must show that $u \in f(s)$.

   All conditions except for the first, second and sixth are unchanged. It is easy to see that the sixth is preserved, since each of π and $\bar{\pi}$ simply removes the first message from channel2.

   Suppose first that $b \neq s'.FS$. Then the effects of π imply that the sender state in s is identical to that in s'. Now, since $u' \in f(s')$, $s'.FS = u'.IS\,mod\,2$; since $b = i\,mod\,2$, this case must have $i \neq u'.IS$. Then the effects of $\bar{\pi}$ imply that the sender state in u is identical to that in u'. It is immediate that the first and second conditions hold for this situation.

   So now suppose that $b = s'.FS$. The invariant above for H implies that either $i = u'.IS - 1$ or $i = u'.IS$. Since $b = i\,mod\,2$ and (since $u' \in f(s')$) $s'.FS = u'.IS\,mod\,2$, this case must have $i = u'.IS$. Then $u.IS = u'.IS + 1$ and $s.FS = s'.FS + 1\,mod\,2$, preserving the second condition. Also, $u.SS$ is unchanged; moreover, $s.QS$ is the same as same as $s'.QS$ except that the first entry (if any) is removed.

   Now, the invariant for H and the fact that the first entry in channel2 in u' has index $u'.IS$ implies that $u'.IS = u'.IR$. Again by the invariant for H, this implies that $u'.LS \geq u'.IS$. Then the fact that $u' \in f(s')$ implies that $s'.QS$ is nonempty. Therefore, the first entry in $s'.QS$ really is removed by the step. Since $s'.QS$ consists of the entries in $u'.SS$, from indices $u'.IS$ to $u'.LS$, since the first entry in $s'.QS$ is removed to yield $s.QS$ and since $u.IS = u'.IS + 1$, it follows that the first condition is preserved.

■

References

[1] Martin Abadi and Leslie Lamport. The existence of refinement mappings. Digital Equipment Corporation, TR No. 29, August 14,1988.

[2] J. Aspnes, A. Fekete, N. Lynch, M. Merritt, and W. Weihl. A theory of timestamp-based concurrency control for nested transactions. In *Proceedings of 14th International Conference on Very Large Data Bases*, pages 431–444, Los Angeles, CA., August 1988.

[3] H. Attiya and N. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. April 1989. Submitted for publication.

[4] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12, 1969.

[5] Arthur Bernstein and Paul K. Harter, Jr. Proving real-time properties of programs with temporal logic. In *Proceedings of the 8th Annual ACM Symposium on Operating System Principles*, pages 1–11, ACM, 1981.

[6] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: an exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[7] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computing*, pages 249–259, Vancouver, British Columbia, Canada, August 1987. Also, to appear in special issue of *IEEE Transactions On Computers on Parallel and Distributed Algorithms.*

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6^{th} Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, August 1987.

[9] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-based locking for nested transactions. In *Proceedings of 3^{rd} International Workshop on Persistent Object Systems*, pages 113–127, Newcastle, Australia, January 1989. An extended version is available as Technical Memo, MIT/LCS/TM-370, Laboratory for Computer Science, MIT, Cambridge, MA, August 1988.

[10] Joseph Y. Halpern and Lenore D. Zuck. A little knowledge goes a long way: simple knowledge-based derivations and correctness proofs for a family of protocols. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 269–280, August 1987. A revised version appears as *IBM Research Report RJ 5857*, October, 1987.

[11] Maurice Herlihy. Extending multiversion time-stamping protocols to exploit type information. *IEEE Transactions on Computers*, C-36(4):443–448, April 1987.

[12] Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, July 1984.

[13] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[14] J. Leo. Personal Communication.

[15] M. Li and P.M.B. Vitanyi. Tape versus stacks and queue: the lower bounds. *Information and Computation*, 78:56–85, 1988.

[16] N. Lynch and K. Goldman. *Distributed Algorithms.* MIT/LCS/RSS 5, Massachusetts Institute of Technology, Laboratory for Computer Science, 1989. Lecture notes for 6.852.

[17] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, August 1987. Extended version in Technical Report MIT/LCS/TR-387, Lab for Computer Science, Massachusetts Institute of Technology, April 1987.

[18] N. Lynch and M. Tuttle. An introduction to input/output automata. To be published in *Centrum voor Wiskunde en Informatica Quarterly.* Also in Technical Memo, MIT/LCS/TM-373, Lab for Computer Science Massachusettes Institute of Technology, November 1988.

[19] Nancy A. Lynch. Concurrency control for resilient nested transactions. *Advances in Computing Research*, 3:335–373, 1986.

[20] Robin Milner. *A Calculus of Communicating Systems. Lecture Notes in Computer Science 92*, Springer-Verlag, Berlin, 1980.

[21] F. Modugno, M. Merritt, and M. Tuttle. Time constrained automata. November 1988. Unpublished manuscript.

[22] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[23] David P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[24] S.K. Sarin, B.T. Blaustein, and C.W. Kaufman. System architecture for partition-tolerant distributed databases. *IEEE Trans. Comput.*, C-34, December 1985.

[25] R. Schaffer. On the correctness of atomic multi-writer registers. Bachelor's Thesis, June 1988, Massachusetts Institute Technology. Also, Technical Memo MIT/LCS/TM-364.

[26] F. Schneider. Personal Communication.

[27] J.L. Welch, L. Lamport, and N. Lynch. A lattice-structured proof technique applied to a minimum spanning tree algorithm. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 28–43, Toronto, Canada, August 1988. Expanded version in Technical Memo, MIT/LCS/TM-361, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 1988.