Ideal Connection Paths in DFC

Pamela Zave
AT&T Laboratories—Research
Florham Park, New Jersey, USA
pamela@research.att.com

2 November 2003

1 Introduction

DFC [1, 3] is a architecture for the description of telecommunication services. Recent experience with using DFC to build real telecommunication services has revealed the need for an improvement to the DFC routing algorithm, namely the introduction of *reverse routing*. This paper begins with the motivation for reverse routing.

This is a significant change with subtle ramifications. The purpose of this paper is to ensure that DFC routing remains on a sound footing. First, there is a formal specification of the new routing algorithm.

The paper also includes proofs of many valuable properties of the new routing algorithm. These properties are defined in terms of an abstraction called an *ideal connection path.*

The presentation herein assumes familiarity with DFC.

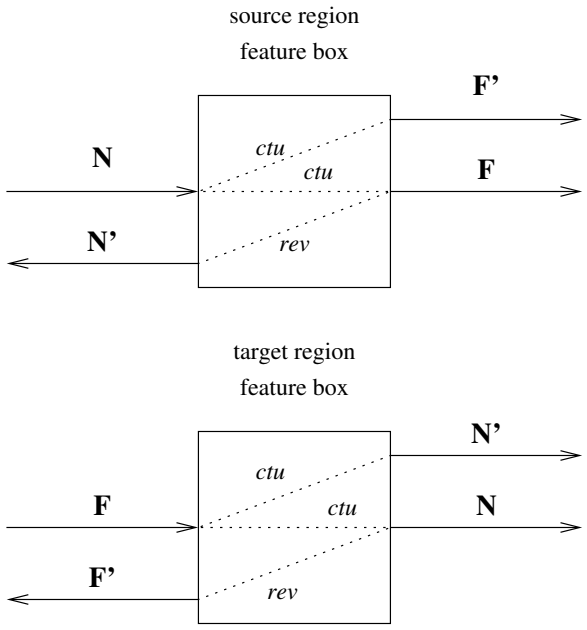

Figure 1: Motivation for the *reverse* method.

2 Motivation for reverse routing

Figure 1 illustrates the need for reverse routing. At the top of the figure is a feature box of type t in a source zone of address a. Its incoming call **N** is the means by which it is connected to its own subscriber, or the *near party.* Behaving typically, it first applied *continue* to the setup of **N**, and used the resulting setup signal to place an outgoing call **F**. **F** is the means by which this box is connected to a *far party.*

Sometime during its lifetime, the box may have to re-establish, replace, or augment one of these connections. To replace **F** by **F'**, it can simply do another *continue.* On the other hand, there is no correct way to replace **N**. Any possible use of *new* or *continue* will produce some awkwardness or anomaly, either in the addresses of the setup signal or in the feature boxes included in the usage.

What kind of feature box performs such an operation? Typically it is a box such as Call Waiting or Mid-Call Move, which is subscribed to in both the source and target regions. This is the crucial clue. When the box replaces **N** by **N'** it is behaving like a target-region feature box, even though it was originally routed to in the source region. Reverse routing allows it to behave properly as a target-region feature box. The box applies *reverse* to the setup signal of **F**, and uses the resulting setup signal to place **N'**. The new setup signal is in the target region. Unless the box programmer has specified address translation, its source and target addresses are the reverse of the source and target addresses in the setup signal of **F**, and it is routed to the next box type after t in the target zone of a.

As the bottom of Figure 1 shows, the situation of a target-region feature box is symmetric. Its original incoming call **F** comes from the far party, and its original outgoing call **N** goes to the near party. To replace **N** by **N'**, it can simply do another *continue*. The only correct way to replace **F**, on the other hand, is to apply *reverse* to the setup signal of **N**, and to place a call as if it were a source-region feature box.

3 A formal model of routing in DFC

3.1 Notation and some basic sets

The notation used in this paper is an older version of Alloy [2], as explained here.

A `domain` is a basic set. A `fixed` domain has fixed membership. All domains are disjoint. A list of sets can be declared to `partition` another set, or to be `disjoint` (but not exhaustive) subsets of another set.

New sets can be formed using `+` for set union and `-` for set difference. The Boolean set operators are `in` for containment, `=` for equality, and `!=` for inequality. Every set has a distinguished subset `emptySet` with no members.

If `X` is a set, then `XSeq` is the set of all finite sequences with members in `X`. Every set `XSeq` has a distinguished subset `emptySeq`; it contains one member of `XSeq`, which is the sequence having no elements. The boolean operator `el` is used for sequence membership.

A variable `V` is typed in a declaration of the form `V: T`, the type `T` is simply a set, and the value of a variable is a subset of its type. The value of a variable can be constrained further by using one of the multiplicity markings `+` (one or more), `?` (one or zero), or `!` (exactly one) to indicate the size of the subset. The keyword `fixed` means that the value is constant.

A binary relation `R` is typed in a declaration of the form `R: S -> T`, where `S` and `T` are sets. The general form `R: S` *m* `-> T` *n* includes multiplicity markings *m* and *n*. This constrains `R` to map each element of `S` to *n* elements of `T`, and to map *m* elements of `S` to each element of `T`. `R: S -> static T` means that `R` always maps a particular element of `S` to the same subset of `T`.

If `S` is a set and `R` is a relation `R: S -> T`, then `S.R` is the relational image of `S` under `R`. In other words, it is the union of all the sets obtained by applying `R` to individuals in `S`.

Assertions are expressed in standard logic, with quantifiers `all`, `some`, binary operators `&&`, `||`, `==>`, `<=>`, and unary operator `!`. Quantification produces singleton subsets rather than individuals. For example, the assertion `all x: X | x in X` is true, which means that in the formula `x in X`, each `x` is a singleton subset of `X` rather than an individual in `X`.

The modified quantifier `some new x: X` creates a new individual in a mutable domain `X`. When an operation is being specified, a prime marks the value of some variable after the operation. All variable values not explicitly specified as changed by the operation are the same after the operation.

Three of the basic sets are enumerated by their partitions.

```
domain { Region, ZoneTag, Orient }

partition srcRegn, trgRegn: fixed Region !
partition whole, suffix: fixed ZoneTag !
partition orig, near, far: fixed Orient !
```

3.2 Entities and attributes

The basic set `BoxType` is considered fixed. It has the distinguished singleton subset `IB`, containing the type of an interface box. This simplifies full DFC, in which there are error boxes and different types of interface box.

```
domain { fixed BoxType }
IB: fixed BoxType !
```

The basic set `Box` is considered fixed, because its dynamic properties are not significant here. Each box has two static attributes and one dynamic attribute.

```
domain { fixed Box }
boxType: Box -> static BoxType !
boxAddr: Box -> static Addr !
ports: Box -> Port
```

The set `ports` contains all the ports that currently belong to the box. The `boxAddr` attribute is a simplification of full DFC; in full DFC an interface box might be associated with more than one address.

The basic set `Setup` is a set of setup signals, or simply "setups." The set is considered fixed, as its dynamic properties are not significant. Each setup has the following static attributes:

```
domain { fixed Setup }

regn: Setup -> static Region !
src: Setup -> static Addr !
trg: Setup -> static Addr !
route:
  Setup -> static (ZoneTag + BoxTypeSeq) !
placing: Setup -> static BoxType ?
```

For simplicity, this specification omits the `dld` field of setups in full DFC.

The basic set `Call` is dynamic, and the operations that create and destroy calls will be specified. There is also a basic set `Port` whose members are parts of calls. Therefore the operations that create and destroy calls also create and destroy ports. Each call has the following static attributes:

```
domain { Call, Port }

outPort: Call -> static Port !
inPort: Call -> static Port !
```

Each port has the following static attribute:

```
setup: Port -> static Setup !
```

The `outPort` of a call is also known as its "caller port," and belongs to the box that places the call. The `inPort` of a call is also known as its "callee port," and belongs to the box that receives the call.

The basic set `Addr` contains addresses, and is considered fixed. `Addr` has the distinguished singleton subset `noAddr`, containing the null address. Each address has two static attributes:

```
domain { fixed Addr }
noAddr: fixed Addr !

srcZone: Addr -> static BoxTypeSeq !
trgZone: Addr -> static BoxTypeSeq !
```

Each zone is a sequence containing all the box types to which the address subscribes in the region, in an order compatible with the `precedes` partial order for the region.

3.3 Reversible box types

The set `reversible` is a special set of box types. If an address subscribes to a box type in `reversible` in either region, it *must* subscribe to it in both regions.

```
reversible: fixed BoxType

all addr: Addr | all bt: reversible |
bt el addr.srcZone <=> bt el addr.trgZone
```

In each region, the `precedes` partial order must be a total order on reversible box types. Furthermore, it must have the property that target precedence is the exact opposite of source precedence: if reversible box `b1` precedes reversible box `b2` in the source region, then `b2` precedes `b1` in the target region.

Any box type can be in `reversible` if the designer so chooses. However, either of two circumstances makes it mandatory that the box type be in `reversible`:

- The box behavior includes a `revcall` operation.
- The box is bound, and it *can* be subscribed to in both regions.

3.4 The routing algorithm

The specification of the DFC routing algorithm takes the form of a parameterized predicate:

```
DFCRoutingAlg(out, inn: Setup !;
              bt: BoxType !, addr: Addr !)
```

The parameters `out, inn` represent the setup signals of the caller and callee ports, respectively. The parameters `bt, addr` represent the type and address, respectively, of the box to which the callee port belongs.

The routing algorithm as specified here is a composition of three steps. (Full DFC has four routing steps; the omitted step concerns the `dld` field.) It is specified using the two local variables `st1, st2: Setup !` to represent the setup signal after the completion of the first two steps. So the four setup signals `out, st1, st2, inn` are actually implemented as one setup signal whose fields change as it goes through the routing process.

Step 1 expands a `ZoneTag` into a whole or partial zone, if necessary. The local variable `st1` holds the result of this step.

```
STEP 1

out.route = suffix ==>
  out.placing in reversible

st1.regn = out.regn
st1.src = out.src
st1.trg = out.trg
st1.placing = emptySet

out.regn = srcRegn && out.route = whole
  ==> st1.route = out.src.srcZone
out.regn = trgRegn && out.route == whole
  ==> st1.route = out.trg.trgZone
out.regn = srcRegn && out.route = suffix
  ==> st1.route =
  out.src.srcZone.suffixAfter[out.placing]
out.regn = trgRegn && out.route = suffix
  ==> st1.route =
  out.trg.trgZone.suffixAfter[out.placing]
out.route in BoxTypeSeq
  ==> st1.route = out.route
```

In this specification, `suffixAfter[marker]`, where `marker` is a singleton set, is a special attribute of any sequence `sequence`. If `marker el sequence`, then

its value is the subsequence coming after the sequence member `marker`. If `marker !el sequence`, then its value is `sequence`. `suffixAfter` is only applied to zones, which are sequences with no duplicates, so we do not need to specify its value in the presence of duplicates.

Step 2 follows Step 1. If the source region is exhausted, it advances the region to the target region. The local variable `st2` holds the result of this step.

```
STEP 2

st2.src = st1.src
st2.trg = st1.trg
st2.placing = emptySet

st1.regn = srcRegn && st1.route = emptySeq
  ==> st2.regn = trgRegn &&
      st2.route = st1.trg.trgZone
st1.regn = trgRegn || st1.route != emptySeq
  ==> st2.regn = st1.regn &&
      st2.route = st1.route
```

Step 3 follows Step 2. Because it is the final step, the result of it is held by the parameter `inn`.

To avoid the issue of routing errors, the specification of Step 3 assumes that every address has an interface box. If the address does not map to a real interface box, it maps to an interface box that has the same behavior as an error box in full DFC.

```
STEP 3

inn.regn = st2.regn
inn.src = st2.src
inn.trg = st2.trg
inn.placing = emptySet

st2.route = emptySeq ==>
  inn.route = emptySeq &&
  bt = IB && addr = st2.trg
st2.route != emptySeq ==>
  inn.route = st2.route.tail &&
  bt = st2.route.head &&
( (inn.regn = srcRegn && addr = inn.src) ||
  (inn.regn = trgRegn && addr = inn.trg)  )
```

The sequence attributes `head` and `tail` have their usual meanings.

Finally, it is possible to summarize certain properties of the DFC routing algorithm as a whole.

```
DFC ROUTING ALGORITHM SUMMARY

inn.src = out.src
inn.trg = out.trg
inn.placing = emptySet

out.regn = srcRegn ==>
  inn.regn = srcRegn || inn.regn = trgRegn
out.regn = trgRegn ==> inn.regn = trgRegn

st2.route = emptySeq ==> st2.regn = trgRegn

inn.regn = srcRegn ==>
  inn.route in inn.src.srcZone.suffixesOf
inn.regn = trgRegn ==>
  inn.route in inn.trg.trgZone.suffixesOf

inn.regn = srcRegn ==>
  bt el inn.src.srcZone
inn.regn = trgRegn ==>
  bt el inn.trg.trgZone || bt = IB

inn.regn = srcRegn ==> addr = inn.src
inn.regn = trgRegn ==> addr = inn.trg
```

Here `suffixesOf` is an attribute of every set of sequences. It is the set of all sequences that are suffixes of elements of the original set of sequences.

3.5 Placing a new call

A call (and its two ports) can be created by execution of a `newcall` operation. Its specification is a parameterized predicate. The parameter `bo` is the box executing the operation, and the parameter `trgArg` is the target address selected by the box program.

```
newcall(bo: Box !, trgArg: Addr !)

ENTITY INTRODUCTION
some bi: Box | some new c: Call |
some new po, pi: Port | some so, si: Setup |
  c.outPort = po
  c.inPort = pi
  bo.ports' = bo.ports + po
  bi.ports' = bi.ports + pi
  po.setup = so
  pi.setup = si

OUTPORT SETUP SIGNAL
so.regn = srcRegn
so.src = bo.boxAddr
so.trg = trgArg
so.placing = emptySet
so.route = whole

ROUTING
DFCRoutingAlg(so,si,bi.boxType,bi.boxAddr)
```

3.6 Continuing a call

A call (and its two ports) can also be created by execution of a `ctucall` operation. The specification parameter `bo` is the box executing the operation, the parameter `ec` is an existing call with a port in `bo`, and the parameters `srcArg, trgArg` are selected by the box program.

```
ctucall(bo: Box !, ec: Call !,
        srcArg, trgArg: Addr !)

PRECONDITION AND ENTITY INTRODUCTION
some p: Port |
  ec.inPort = p && p in bo.ports

ADDITIONAL ENTITY INTRODUCTION
some bi: Box | some new c: Call |
some new po, pi: Port |
some s, so, si: Setup |
  c.outPort = po
  c.inPort = pi
  bo.ports' = bo.ports + po
  bi.ports' = bi.ports + pi
  p.setup = s
  po.setup = so
  pi.setup = si

ADDITIONAL PRECONDITIONS
s.regn = srcRegn && srcArg != noAddr
  ==> srcArg != bo.boxAddr
s.regn = trgRegn && trgArg != noAddr
  ==> trgArg != bo.boxAddr

OUTPORT SETUP SIGNAL
so.regn = s.regn

so.regn = srcRegn && srcArg != noAddr ==>
  so.src = srcArg && so.route = whole
so.regn = srcRegn && srcArg = noAddr ==>
  so.src = bo.boxAddr && so.route = s.route
so.regn = trgRegn && trgArg != noAddr ==>
  so.trg = trgArg && so.route = whole
so.regn = trgRegn && trgArg = noAddr ==>
  so.trg = bo.boxAddr && so.route = s.route

so.regn = srcRegn && trgArg != noAddr
  ==> so.trg = trgArg
so.regn = srcRegn && trgArg = noAddr
  ==> so.trg = s.trg
so.regn = trgRegn && srcArg != noAddr
  ==> so.src = srcArg
so.regn = trgRegn && srcArg = noAddr
  ==> so.src = s.src

so.placing = emptySet

ROUTING
DFCRoutingAlg(so,si,bi.boxType,bi.boxAddr)
```

3.7 Reversing a call

A call (and its two ports) can also be created by execution of a `revcall` operation. The specification parameter `bo` is the box executing the operation, the parameter `ec` is an existing call with a port in `bo`, and the parameters `srcArg, trgArg` are selected by the box program.

```
revcall(bo: Box !, ec:Call !,
        srcArg, trgArg: Addr !)

PRECONDITION AND ENTITY INTRODUCTION
some p: Port |
  ec.outPort = p && p in bo.ports

ADDITIONAL ENTITY INTRODUCTION
some bi: Box | some new c: Call |
some new po, pi: Port |
some s, so, si: Setup |
  c.outPort = po
  c.inPort = pi
  bo.ports' = bo.ports + po
  bi.ports' = bi.ports + pi
  p.setup = s
  po.setup = so
  pi.setup = si

ADDITIONAL PRECONDITIONS
s.regn = trgRegn && srcArg != noAddr
   ==> srcArg != bo.boxAddr
s.regn = srcRegn && trgArg != noAddr
   ==> trgArg != bo.boxAddr

OUTPORT SETUP SIGNAL
s.regn = srcRegn ==> so.regn = trgRegn
s.regn = trgRegn ==> so.regn = srcRegn

so.regn = srcRegn && srcArg != noAddr ==>
  so.src = srcArg && so.route = whole
so.regn = srcRegn && srcArg = noAddr ==>
  so.src = bo.boxAddr && so.route = suffix
so.regn = trgRegn && trgArg != noAddr ==>
  so.trg = trgArg && so.route = whole
so.regn = trgRegn && trgArg = noAddr ==>
  so.trg = bo.boxAddr && so.route = suffix

so.regn = srcRegn && trgArg != noAddr
  ==> so.trg = trgArg
so.regn = srcRegn && trgArg = noAddr
```

```
  ==> so.trg = s.src
so.regn = trgRegn && srcArg != noAddr
  ==> so.src = srcArg
so.regn = trgRegn && srcArg = noAddr
  ==> so.src = s.trg

so.route  = suffix ==>
   so.placing = bo.boxType
so.route != suffix ==> so.placing = emptySet

ROUTING
DFCRoutingAlg(so,si,bi.boxType,bi.boxAddr)
```

3.8 Simple properties

The following properties are trivially provable from the preceding specification of DFC.

- Each port belongs to exactly one box.
- Each port belongs to exactly one call.

4 Ideal links

4.1 What is an ideal link?

An *ideal link* is a connection between two ports of the same box. It is "ideal" in two senses:

- There are restrictions on which ports can be linked. These restrictions do not exist in DFC.
- An ideal link does not correspond directly to signal linkage or media linkage in DFC. It is an abstraction of these. A DFC feature box, behaving ideally, would never create a signal or media link between two ports unless there was an ideal link between them.

The purpose of ideal links is to impose structure that will help us define and prove important properties of DFC usages. It is not possible for all feature boxes to confine themselves to ideal signal and media linkages at all times. However, the exceptions should be few, and they should be scrutinized with extra care to make sure that they are not harmful. The properties based on ideal links will help us understand what "harmful" means.

This theory concerns only connections among telecommunication devices and people. Resources, calls to resources, and intra-box links to calls to resources are not described and not constrained.

There is a set of ideal links. This set is dynamic, and the operations that create and destroy links will be specified. Each link has two static attributes.

```
domain { Link }

nearPort: Link -> static Port !
farPort: Link -> static Port !
```

4.2 Port orientation

For convenience, we add a redundant port attribute. Now each port has the following additional static attribute:

```
orient: Port -> static Orient !
```

The `orient` attribute of a port is determined when the port is created, as part of the creation of the call. If the call is created by `newcall`, then:

```
po.orient = orig
```

The following must be added to the definitions of `ctucall` and `revcall`. It defines the `orient` attribute of the `outPort` of the created call.

```
po.setup.regn = srcRegn ==> po.orient = far
po.setup.regn = trgRegn ==> po.orient = near
```

The following must be added to the definitions of `newcall`, `ctucall` and `revcall`. It defines the `orient` attribute of the `inPort` of the created call.

```
pi.setup.regn = srcRegn ==> pi.orient = near
pi.setup.regn = trgRegn ==> pi.orient = far
```

4.3 Linking and unlinking

A link is created by execution of a `link` operation. The parameters are the two ports to be linked. The preconditions on port orientation ensure that these ports are distinct. Another precondition ensures that the two ports are not already linked.

```
link(pn, pf: Port !)

PRECONDITIONS
some b: Box |
  pn in b.ports && pf in b.ports

pn.orient = near
pf.orient = far

! (some l: Link |
  l.nearPort = pn && l.farPort = pf)

POSTCONDITION
some new l: Link |
  l.nearPort = pn && l.farPort = pf
```

As an obvious consequence of the preconditions, a port with `orient = orig` cannot be linked. This formalizes the rule that a feature box should only use a `newcall` operation when it is acting as an agent of its subscriber, and is truly initiating a call as if it were the subscriber. Interface boxes always create calls

using `newcall`. The linkages created by the interface box between its calls and its telecommunication device are not part of this formal model.

A call and its two ports can be destroyed by execution of a `tdcall` operation. If either of these ports is participating in a link, the operation also destroys the link.

A link is also destroyed by execution of an `unlink` operation, which has no other effects.

5 Ideal connection paths

An *ideal connection path* (or just *path*) is a set of calls that are connected together, end-to-end and contiguously, by ideal links.

Because a path is a set, it can have subpaths that are themselves paths. A *maximal path* is a path that is not a subpath of another path.

5.1 Path properties: Orientation

The following table summarizes the possible orientations of the two ends of a call, based on the specification of the DFC routing algorithm and on the definition of the `orient` attribute. The `regn` column for each port `p` gives the value of `p.setup.regn`.

outPort		inPort	
regn	orient	regn	orient
srcRegn	far + orig	srcRegn	near
srcRegn	far + orig	trgRegn	far
trgRegn	near	trgRegn	far

A call `c` with `c.outPort.setup.regn = srcRegn` and `c.inPort.setup.regn = trgRegn` is a *midpoint call.*

Theorem 1: Each path has at most one midpoint call.

Proof Theorem 1: A midpoint call `c` has `c.outPort.orient = far` or `c.outPort.orient = orig`, and `c.inPort.orient = far`. If either of its ports is linked, it is the `farPort` of the link.

The `nearPort` of the link is a port `pn` with `pn.orient = near`. From the table, the other end of its call is a port `pf` with `pf.orient = orig` or `pf.orient = far`. If `pf` is linked, then `pf.orient = far`, and `pf` is the `farPort` of the link.

Because this pattern continues recursively, a path segment extending from a midpoint call can only contain calls with one port having the orientation `near`. Such a call cannot be a midpoint call. □

The proof of Theorem 1 shows us that a path has the structure depicted in Figure 2. A *full path* is a path containing a midpoint call. A *half path* is a path containing no midpoint call. A midpoint call divides a full path into two half paths, either of which might be empty.

Since a path is a set of calls only, we need a way to talk about boxes as well. The *boxes connected by a path* is the smallest set of boxes, each of which contains a port in the path.

Note that any half path has *outward* and *inward* directions, based on the position of the midpoint call. Because these directions can be determined from port orientations alone, they are unambiguous even in a half path that is not a subpath of a full path, and therefore not associated with any midpoint call.

Figure 2 shows one half path whose outer port has `orient = orig`, and one half path whose outer port has `orient = far`. This is for illustration only. A full path could have both outer ports with `orient = orig`, or both outer ports with `orient = far`. The fact that there *are* two outer ports is proved as follows.

Theorem 2: A full path is not a cycle.

Proof Theorem 2: Consider a full path depicted as in Figure 2. For the path to be a cycle, both outer ports in the diagram would have to have `orient = far`, and both outer boxes in the diagram would have to be connected by a call `c` with `c.outPort.orient = near` and `c.inPort.orient = near`. The DFC routing algorithm does not allow such a call. □

Note that a half path can be a cycle. Cyclic half paths must be avoided by additional constraints.

Theorem 3: If a path connects two interface boxes, it is a full path.

Proof Theorem 3: From the model in Section 3, a port `pf` on an interface box has `pf.orient = orig` or `pf.orient = far`. If two such ports are connected by a path with a single call, from the table, that call is a midpoint call, and its path is a full path.

If two ports on interface boxes are connected by a longer path, then at least one of those ports is not part of a midpoint call. From the table, the other end of the call is a port `pn` with `pn.orient = near`. It is linked to a port with `orient = far`.

Because this pattern continues recursively, a path connecting two interface boxes must contain a call with neither port having `orient = near`. This is a midpoint call. □

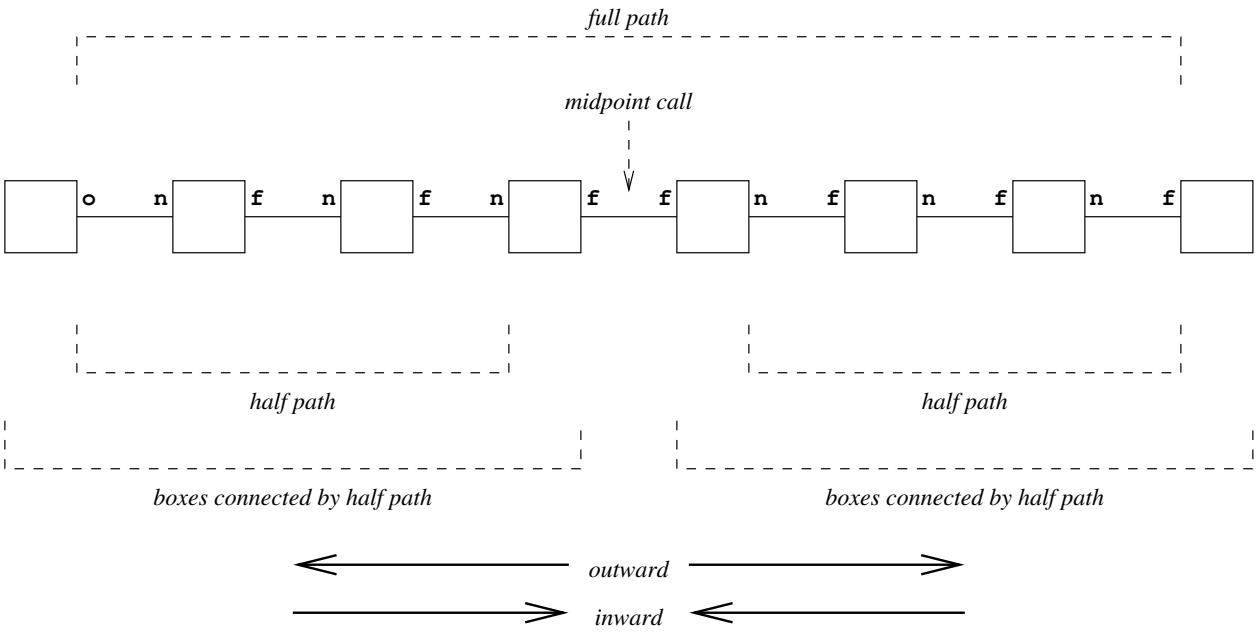

Figure 2: The anatomy of a full path.

5.2 Path properties: Zones

A *zone of address* `a` is a nonempty set of feature boxes with address `a`, connected by a half path containing no port with `orient = orig`. A *maximal zone of* `a` is a zone of `a` that is not a subzone of another zone of `a`. If the size of a zone is one box, then the size of its connecting path is zero.

Because of reverse routing, the structure of a zone can be quite complex. The purpose of this section is to expose and elucidate the structure of a zone.

Lemma 1: Let `h` be a zone of `a: Addr !` Let the path connecting `h` contain only a single call `c`. Let `pn` be its port with `pn.orient = near` and let `pf` be its port with `pf.orient = far`. Let `bn` and `bf` be the boxes of `pn` and `bf`, respectively, with types `tn` and `tf`, respectively. Then either `<tn,tf>` is a subsequence of `a.trgZone`, or `<tf,tn>` is a subsequence of `a.srcZone`.

Proof Lemma 1: Whichever box placed `c` derived its setup signal, through `ctucall` and possibly `revcall`, from the setup `in` of a call received by the box. Let `t` be the type of the placing box. From the routing algorithm,

```
in.regn = srcRegn
  ==> a = in.src &&
      <t,in.route> in a.srcZone.suffixesOf
in.regn = trgRegn
  ==> a = in.trg &&
      <t,in.route> in a.trgZone.suffixesOf
```

Let `ctu` be the outgoing setup signal resulting from applying `ctucall` to `in`. From the definition of `ctucall`,

```
ctu.regn = srcRegn ==>
   (ctu.src = a &&
   <t,ctu.route> in a.srcZone.suffixesOf)
|| (ctu.src != a && ctu.route = whole)

ctu.regn = trgRegn ==>
   (ctu.trg = a &&
   <t,ctu.route> in a.trgZone.suffixesOf)
|| (ctu.trg != a && ctu.route = whole)
```

Let `rev` be the outgoing setup signal resulting from applying `revcall` in the placing box. From the definition of `revcall`,

```
rev.regn = srcRegn ==>
   (rev.src = a && rev.route = suffix)
|| (rev.src != a && rev.route = whole)

rev.regn = trgRegn ==>
   (rev.trg = a && rev.route = suffix)
|| (rev.trg != a && rev.route = whole)
```

From the table, there are two cases.

Case 1: `pn = c.outPort, pn.setup.regn = trgRegn, pf = c.inPort, pf.setup.regn = trgRegn`. Let `out` be `pn.setup`.

From the preceding,

```
out.trg = a ==>
  out.route = suffix ||
  <tn,out.route> in a.trgZone.suffixesOf
```

After Step 1 of routing,

```
st1.trg = a ==>
  st1.route = a.trgZone.suffixAfter[tn]
```

Step 2 of routing does nothing. In the Routing Phase, since `tf != IB`, `tf = st1.route.head`. So `<tn,tf>` is a subsequence of `a.trgZone`.

Case 2: `pf = c.outPort, pf.setup.regn = srcRegn, pn = c.inPort, pn.setup.regn = srcRegn`. Let `out` be `pf.setup`.

From the preceding,

```
out.src = a ==>
  out.route = suffix ||
  <tf,out.route> in a.srcZone.suffixesOf
```

After Step 1 of routing,

```
st1.src = a ==>
  st1.route = a.srcZone.suffixAfter[tf]
```

Step 2 of routing does nothing (if it did, `c` would be a midpoint call, and we would have `pn.orient = far`). This tells us that `st1.route != emptySeq`. In the Routing Phase, `tn = st1.route.head`. So `<tf,tn>` is a subsequence of `a.srcZone`. □

Lemma 2: Let `h` be a zone of `a: Addr !` Let `h` contain only three boxes `bx, by,` and `bz`, listed in order from innermost to outermost. Let the types of the boxes be `tx, ty,` and `tz`, respectively, where `ty` is not a member of `reversible`. Then either `<tx,ty,tz>` is a subsequence of `a.trgZone`, or `<tz,ty,tx>` is a subsequence of `a.srcZone`.

Proof Lemma 2: Because `ty` is not reversible, either `by` is a free box, or it is a bound box that can be subscribed to in only one region. In either case, it receives incoming calls in only one region.

Because `ty` is not reversible, it cannot perform `revcall`, and any call it places must be placed via `ctucall`. Therefore any call it places must be in the same region as the one in which it receives incoming calls.

Combining this with two applications of Lemma 1, either `<tx,ty,tz>` is a subsequence of `a.trgZone`, or `<tz,ty,tx>` is a subsequence of `a.srcZone`. □

For every `a: Addr`, let `a.reversibles` be the projection of `a.trgZone` onto the sequence of its reversible box types. `a.reversibles` contains all the reversible box types subscribed to by `a`, in order from innermost to outermost.

Theorem 4: Let `h` be a zone of `a: Addr !` Ordering the box types in `h` from innermost to outermost, the sequence of reversible box types in `h` is a subsequence of `a.reversibles`.

Proof Theorem 4: Unless `h` contains at least two reversible boxes, the theorem is trivially true.

Let `bx` and `by` be two reversible boxes in `h` that are not separated by any reversible boxes. Let `bx` be connected to `by` through a port in `bx` with `orient = near`, so that `by` is connected to `bx` through a port in `by` with `orient = far`. Let their box types be `tx` and `ty`, respectively. There are two cases.

Case 1: `bx` and `by` are adjacent in `h`. Then from Lemma 1, either `<tx,ty>` is a subsequence of `a.trgZone`, or `<ty,tx>` is a subsequence of `a.srcZone`. Either way, because of the relationship between the source and target total orders on reversible boxes, `<tx,ty>` is a subsequence of `a.reversibles`.

Case 2: `bx` and `by` are not adjacent in `h`, but are separated by some number of non-reversible boxes. From repeated applications of Lemma 2, either `<tx,...,ty>` is a subsequence of `a.trgZone`, or `<ty,...,tx>` is a subsequence of `a.srcZone`, where the ellipses represent sequences of non-reversible box types. Either way, because of the relationship between the source and target total orders on reversible boxes, and because the box types in the ellipses are not in `a.reversibles`, `<tx,ty>` is a subsequence of `a.reversibles`. □

Lemma 3: Let `h` be a maximal zone of `a: Addr !` Let `bi` be the innermost box of `h`, with type `ti`. Let the path connecting `h` be linked in `bi` to an incoming call `c` placed by box `bo`. Then `ti = a.trgZone.head`.

Proof Lemma 3: From Figure 2, `c.inPort.orient = far`. From the table, `c.inPort.setup.regn = trgRegn`.

There are two reasons why `bo` might not be in `h`. `c` might be a midpoint call, in which case it is easy to see from the routing algorithm that `ti = a.trgZone.head`.

Alternatively, `bi` might be a feature box with an address `a'` distinct from `a`. Because `c` is not a midpoint call, we know that:

```
c.outPort.setup.regn = trgRegn
```

From the routing algorithm and the operations for placing calls:

```
c.outPort.setup.trg = a
c.outPort.setup.route = whole
```

From this and the routing algorithm, `ti = a.trgZone.head`. □

Lemma 4: Let `h` be a maximal zone of `a: Addr !` Let `bi` be the outermost box of `h`, with type `ti`. Let the path connecting `h` be linked in `bi` to an incoming call `c` placed by box `bo`. Then `ti = a.srcZone.head`.

Proof Lemma 4: From Figure 2, `c.inPort.orient = near`. From the table, `c.inPort.setup.regn = srcRegn`.

There are two reasons why `bo` might not be in `h`. If `c` has `outPort.orient = orig`, then `h` cannot be extended to `bo`. In this case it is easy to see from the `newcall` operation and routing algorithm that `ti = a.srcZone.head`.

Alternatively, `bo` might be a feature box with an address `a'` distinct from `a`. From the routing algorithm and the operations for placing calls:

```
c.outPort.setup.regn = srcRegn
c.outPort.setup.src = a
c.outPort.setup.route = whole
```

From this and the routing algorithm, `ti = a.srcZone.head`. □

A *source truncating box type* is the type of a box with the possible behavior of placing a call with `outPort.setup.regn = srcRegn` and `outPort.setup.src != a`, where `a` is the box address. In a path, the call would end the source zone of `a` whether the zone contained all of the subscribed boxes or not. A *target truncating box type* is the type of a box with the possible behavior of placing a call with `outPort.setup.regn = trgRegn` and `outPort.setup.trg != a`, where `a` is the box address. In a path, the call would end the target zone of `a` whether the zone contained all of the subscribed boxes or not.

Lemma 5: Let `h` be a maximal zone of `a: Addr !` Let `bo` be the innermost box of `h`, with type `to`. Let the path connecting `h` be linked in `bo` to an outgoing call `c` received by box `bi`. Then `to` is either a source truncating box type, or the last element of `a.srcZone`.

Proof Lemma 5: From Figure 2, `c.outPort.orient = far`. From the table, `c.outPort.setup.regn = srcRegn`.

Case 1: `c.outPort.setup.route = whole`, which from the call operations, can only occur if `c.outPort.setup.src != a`. Either `c` is a midpoint call, or `bi.boxAddr != a`. Because `c.outPort.setup.src != a`, `to` is a source truncating box type.

Case 2: After Step 1 of routing, `st1.route = a.srcZone.suffixAfter[to]`. If this were not an empty sequence, `c` would be routed to a feature box of `a`, which is a contradiction. So this is an empty sequence, which can only occur if `to` is the last element of `a.srcZone`. □

Lemma 6: Let `h` be a maximal zone of `a: Addr !` Let `bo` be the outermost box of `h`, with type `to`. Let the path connecting `h` be linked in `bo` to an outgoing call `c` received by box `bi`. Then `to` is either a target truncating box type, or the last element of `a.trgZone`.

Proof Lemma 6: From Figure 2, `c.outPort.orient = near`. From the table, `c.outPort.setup.regn = trgRegn`.

Case 1: `c.outPort.setup.route = whole`, which from the call operations, can only occur if `c.outPort.setup.trg != a`. Either `bi.boxType = IB`, or `bi.boxAddr != a`. Because `c.outPort.setup.trg != a`, `to` is a target truncating box type.

Case 2: After Step 1 of routing, `st1.route = a.trgZone.suffixAfter[to]`. If this were not an empty sequence, `c` would be routed to a feature box of `a`, which is a contradiction. So this is an empty sequence, which can only occur if `to` is the last element of `a.trgZone`. □

Theorem 5: Let `h` be a maximal zone of `a: Addr !` Let `tr` be a reversible box type subscribed to by `a`. If `a` subscribes to any source truncating box type, `tr` does not succeed that box type in `a.srcZone`. If `a` subscribes to any target truncating box type, `tr` does not succeed that box type in `a.trgZone`. Then `h` contains a box of type `tr`.

Proof Theorem 5:

Innermost box: Let `bj` be the innermost box of `h`. Moving outward, let `bk` be the innermost reversible box of `h` or the outermost box of `h`, whichever comes first.

If Lemma 3 applies to `bi`, then from Lemma 2, `<tj,..,tk>` is an initial subsequence of `a.trgZone`.

If bk is not reversible, then from Lemma 6, tk is either a target truncating box type, or the last element of a.trgZone. This contradicts the assumptions of the theorem, so bk is reversible, and tk = tr or tk precedes tr in a.reversibles.

If Lemma 5 applies to bj, then from Lemma 2, <tk,..,tj> is a final subsequence of a.srcZone, or a subsequence ending with a source truncating box type. If bk is not reversible, then from Lemma 4, tk is a.srcZone.head. This contradicts the assumptions of the theorem, so bk is reversible, and tk = tr or tk precedes tr in a.reversibles.

Outermost box: Let bl be the outermost box of h. Moving inward, let bm be the outermost reversible box of h or the innermost box of h, whichever comes first.

If Lemma 4 applies to bl, then from Lemma 2, <tl,..,tm> is an initial subsequence of a.srcZone. If bm is not reversible, then from Lemma 5, tm is either a source truncating box type, or the last element of a.srcZone. This contradicts the assumptions of the theorem, so bm is reversible, and tm = tr or tr precedes tm in a.reversibles.

If Lemma 6 applies to bl, then from Lemma 2, <tm,..,tl> is a final subsequence of a.trgZone, or a subsequence ending with a target truncating box type. If bm is not reversible, then from Lemma 3, tm is a.trgZone.head. This contradicts the assumptions of the theorem, so bm is reversible, and tm = tr or tr precedes tm in a.reversibles.

Conclusion: There exist reversible boxes bk and bm in h, not necessarily distinct, such that tk = tr or tk precedes tr in a.reversibles, and tm = tr or tr precedes tm in a.reversibles. From Theorem 4, there exists a box of type tr in h. □

6 Uses and examples

As an example of unusual, but nevertheless ideal, connection paths, Figure 3 shows a simple way of doing Click to Dial. The first path is initiated by the Web interface box with address w, which subscribes to C2D in the source zone. The target of the initial call is the subscriber s, who has done the click. The setup signal of the initial call includes an encoding of the intended far-party address t. The first action of the C2D box is simply to continue its incoming call to s.

If the call to s is answered, C2D next tears down its incoming call and reverses its outgoing call, with address translation from w to t. The resultant path is shown at the bottom of the figure. All boxes have links between their two calls. Note that both s and t have target feature boxes.

Ideal connection paths provide a new way of looking at DFC. One lesson of this new way of looking at DFC is that feature boxes should use newcall guardedly. Many previous uses of newcall were actually attempts to simulate revcall.

One legitimate use of newcall by a feature box occurs when the feature is truly acting as an agent of its subscriber, and is making a new call exactly as if it were the subscriber. Since the box "is" the subscriber, and is acting in some sense like an interface box, it need not link the call to any other call. An autoresponse feature for electronic mail uses newcall in this way.

Another legitimate use of newcall by a feature box occurs when the box address represents a scheduled meeting, and the box implements it. The box may place calls using newcall to add some participants.

The only common use of newcall by feature boxes should be to place calls to resource addresses. The call must have a null source address, to avoid invoking source feature boxes. As mentioned in Section 4, the theory of ideal links and ideal connection paths does not concern resource calls.

Any kind of conferencing violates the constraints on ideal connection paths. Figure 4 shows why. The leftmost box is creating a distributed, multimedia virtual device out of several physical devices. It has two near ports linked to each other.

The middle and rightmost box are performing conferencing in a more usual sense. The spontaneous conference is being formed among parties who have called or been called by a person. The scheduled meeting has no person who is distinguished in that way; the address of the box identifies the meeting. Both boxes link far and orig ports to each other.

As these examples show, the existence of conferencing does not undermine the validity of the perspective on DFC usages provided by ideal connection paths. Ideal connection paths may not be the *only* thing going on, but they are the most important thing going on, because they determine which feature boxes are present in each usage. Nevertheless, these examples show that an understanding of ideal connection paths is not sufficient for analyzing feature interactions.

7 Other changes to DFC routing

This work has revealed the need for two other, smaller changes to DFC routing.

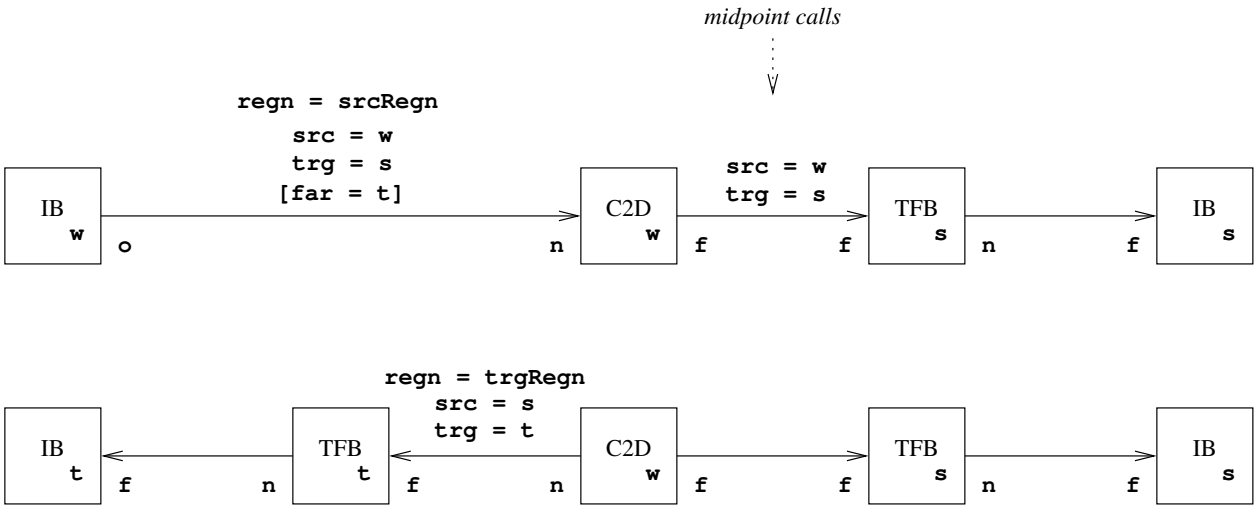

Figure 3: Ideal connection paths containing Click to Dial.

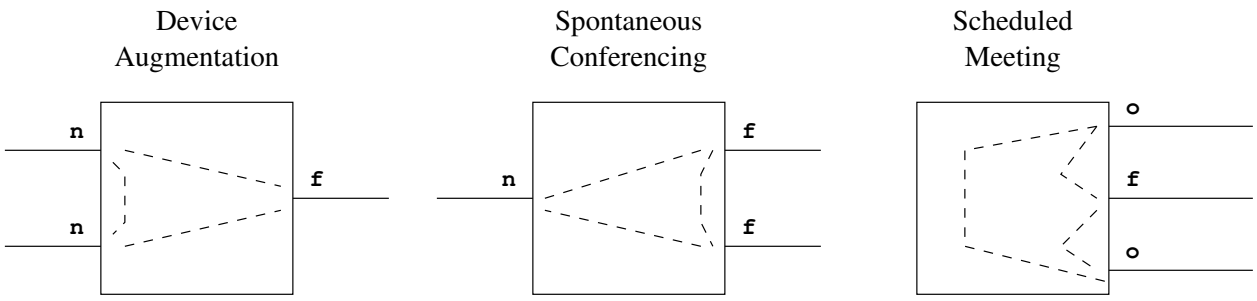

Figure 4: Feature boxes that do some kind of conferencing.

7.1 No more network zone

The network zone is being eliminated from DFC because its usefulness has never been demonstrated in practice, and it complicates the routing considerably. This decision is safe because anything that could have been accomplished with a network zone can still be accomplished.

Let `NB` be a type of box that was written for the the network zone. A box of type `NB` is free; if and when it decides to continue its incoming call, it uses `ctucall`, and does not change the source address.

Theorem 6: For `all a: Addr` let `NB` be the last element of `a.srcZone`, and not an element of `a.trgZone`. Every call placed by a box of type `NB` is a midpoint call. Every midpoint call is placed by a box of type `NB`.

Proof Theorem 6: Every call `c` placed by a box of type `NB` has `c.outPort.setup.regn = srcRegn` and `c.outPort.setup.route = emptySeq`, and is therefore a midpoint call.

For a call `c` to be a midpoint call, `c.outPort.setup` must have the following attributes. First, `regn = srcRegn`. Either `route = whole` and `src.srcZone = emptySeq` (Case 1), `route = suffix` and `src.srcZone.suffixAfter[out.placing] = emptySeq` (Case 2), or `route = emptySeq` (Case 3).

Case 1 cannot occur because no address has an empty source zone. Case 2 cannot occur because the placing box would have to be both reversible and the last element of some address's source zone, which contradicts the assumptions of the theorem. In Case 3 the placing box is the last box of some address's source zone, which means it is a box of type `NB`. □

The significance of Theorem 6 is that, under the conditions of the theorem, boxes of type `NB` appear in usages exactly where they would have appeared if they were network-zone boxes, and DFC routing still

had a network zone. If a designer wishes to simulate a network zone containing a box of type NB, all he has to do is subscribe every address to it in the source zone, and make its precedence last in the source zone.

7.2 A new setup field

Some setups have an additional attribute:

```
outer: Setup -> static Addr ?
```

Its purpose is to carry the previous source address in the source region, if any, for use in address authentication. It is constrained by the architecture so that it is guaranteed correct for this purpose.

Consider a source feature box of address s1 that continues an incoming call, changing the source address to s2. As explained in [4], feature boxes of s2 have a need and a right to determine that they are being invoked legitimately. Voice-based authentication of the caller is one way of doing this. Whenever it is appropriate, by far the simplest alternative is to check that s1 is an address trusted by s2. This is only possible, however, if the feature boxes of s2 have a trustworthy means of obtaining the address of the box that performed the address translation. This is the purpose of the outer attribute.

Privacy is a major concern of [4], and privacy demands that more concrete addresses not be leaked to the feature boxes and owners of more abstract addresses. Since s1 is probably more concrete than s2, why is this not a violation of privacy? First, it makes source translation the exact dual of target translation, in which the feature boxes of an abstract address know the next outer address in the target region. Second, the definition of the outer attribute guarantees that address information cannot travel in it any further than the feature boxes of the next inner address in the source region.

The outer attribute belongs only to some setup signals with regn = srcRegn. The newcall operation is augmented as follows:

```
so.outer = bo.boxAddr
```

The ctucall and revcall operations are augmented as follows:

```
so.regn = srcRegn && srcArg != noAddr
  ==> so.outer = bo.boxAddr
so.regn = trgRegn || srcArg = noAddr
  ==> so.outer = emptySet
```

In Step 2 of the routing algorithm, the two major rules are changed as follows:

```
st1.regn = srcRegn && st1.route = emptySeq
  ==> st2.regn = trgRegn &&
      st2.route = whole &&
      st2.outer = emptySet
st1.regn = trgRegn || st1.route != emptySeq
  ==> st2.regn = st1.regn &&
      st2.route = st1.route &&
      st2.outer = st1.outer
```

All the other steps of the routing algorithm are augmented to copy the outer attribute without change.

Two theorems establish the correctness of the outer attribute.

Theorem 7: Let h be a maximal zone of a: Addr ! Let bi be the outermost box of h, and let the path connecting h be linked in bi to an incoming call c placed by box bo. Then c.inPort.setup.outer = bo.boxAddr.

Proof Theorem 7: From the proof of Lemma 4, c.inPort.setup.regn = srcRegn, and there are two cases. In the first case, c was placed using newcall, so that c.outPort.setup.outer = bo.boxAddr. The only routing step that might alter it is Step 2, but if that occurred we would have c.inPort.setup.regn = trgRegn, which contradicts the assumptions.

In the second case, c was placed using ctucall or revcall, and bo is a feature box with an address distinct from a. From the routing algorithm and the operations for placing calls:

```
c.outPort.setup.regn = srcRegn
c.outPort.setup.outer = bo.boxAddr
```

The only routing step that might alter the outer field is Step 2, but if that occurred we would have c.inPort.setup.regn = trgRegn, which contradicts the assumptions. □

Theorem 8: Let box b be connected by an ideal connection path. Let c be part of the path and an incoming call to b. b does not satisfy the conditions to play the role of bi in Theorem 7. Then c.inPort.setup.outer = emptySet.

Proof Theorem 8: We consider three cases.

Case 1: c was placed using newcall. After Step 1 of routing, st1.route = out.src.srcZone. If the route is empty, the augmented Step 2 will make st2.outer = emptySet, which will be preserved by subsequent routing steps. If the route is not empty, then b will satisfy the conditions to play the role of bi, which contradicts the assumptions.

Case 2: `c` was placed using `ctucall` or `revcall`, and `so.regn = trgRegn || srcArg = noAddr`, so that `c.outPort.setup.outer = emptySet`. No routing step changes an empty `outer` attribute to a nonempty one.

Case 3: `c` was placed using `ctucall` or `revcall`, and `so.regn = srcRegn && srcArg != noAddr`, so that

```
c.outPort.setup.regn = srcRegn
c.outPort.setup.route = whole
c.outPort.setup.src != c.outPort.setup.outer
c.outPort.setup.outer != emptySet
```

After Step 1 of routing, `st1.route = out.src.srcZone`. If the route is empty, the augmented Step 2 will make `st2.outer = emptySet`, which will be preserved by subsequent routing steps. If the route is not empty, then `b` will satisfy the conditions to play the role of `bi`, which contradicts the assumptions. □

References

[1] The DFC Web site:
`http://www.research.att.com/projects/dfc`

[2] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proceedings of the Ninth ACM SIGSOFT International Symposium on the Foundations of Software Engineering and the Eighth European Software Engineering Conference*, pages 62-73. ACM, 2001.

[3] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering* XXIV(10):831-847, October 1998.

[4] Pamela Zave. Address translation in telecommunication features. Submitted for publication, 2003.