# The DFC Manual

Michael Jackson Pamela Zave

November 2003

## Contents

# 1 Introduction (Nov 03)

## 1.1 What is DFC? (Nov 03)

DFC ("Distributed Feature Composition") is an architecture for the description and implementation of telecommunication services. DFC was designed for generality, feature modularity, structured feature composition, and implementation-independence. The principal concepts and initial version of DFC are presented in [5]. The major changes that have been made to the initial version of DFC are explained by [6, 7, 8]. An IP-based implementation of DFC is described in [1].

This manual provides a current description of DFC that can be read sequentially or used for reference. Current research papers related to DFC can be found at [2].

The glossary at the end of this manual is the primary tool for cross-referencing. It defines all the important terms, and lists the chief sections where each is discussed.

## 1.2 Scope of DFC (Nov 03)

DFC describes a telecommunication network whose boundary is marked by *interface boxes*. Outside the boundary are telecommunication devices and other networks. They communicate with the DFC network through various protocols.

Inside the boundary, DFC describes telecommunication features, their logical composition, and their effect on the signaling and media channels that pass through the interface boxes. Inside the boundary, a single protocol is used uniformly. The primary function of the interface boxes is conversion between the inner and outer protocols.

The DFC architecture is not concerned with:

- how the abstract architecture is implemented on a telecommunication network, or how its resources are allocated and optimized;
- how the call-processing it describes is connected to the business processes of billing, marketing, and customer care.

Furthermore, the DFC specification describes a telecommunication network whose configuration (interfaces, addresses, features, subscriptions, and persistent data directly related to them) is static. It is not concerned with the provisioning mechanisms for initializing them or for changing them during the lifetime of the system.

Most parts of the configuration can change during the lifetime of the system; some anomalies may result, but they will be transient. If some part of the configuration is explicitly labeled as *fixed* or *static*, then it really cannot change during the lifetime of the system.

## 1.3 Notation (Nov 03)

Two formal description languages are used in the manual:

- *Alloy* [4] is used to describe data and signal formats, and operations on data and signals; and
- *Promela* [3] is used to describe the protocols by which interface and feature boxes communicate with one another and with DFC routers.

Most formal descriptions are paraphrased in English, so familiarity with these languages is not absolutely necessary for reading this manual. Because Alloy is less widely known than Promela, and because this manual actually uses an older version of it, our Alloy-based syntax is 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 set containment, `=` for set equality, and `!=` for set 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 $S$. The transpose of a relation is denoted by prefixing its name with a tilde.

Assertions are expressed in standard logic, with quantifiers *all, some, sole*, 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 quantifier *sole* denotes that there is at most one singleton subset with the specified property.

# 2 Boxes, box types, and addresses (Nov 03)

## 2.1 Boxes (May 01)

Boxes are either *interface boxes*, *feature boxes*, or *error boxes*.

```
domain { Box }
partition IBox, FBox, EBox: Box
```

An interface box is either a *line interface box*, which provides an interface to an external telecommunication device such as a telephone, or a *trunk interface box*, which provides an interface to customers attached to other telecommunication networks, or a *resource interface box*, which provides an interface to media-processing resources such as announcement players, message recorders, and text-to-voice converters.

```
partition LIBox, TIBox, RIBox: IBox
```

A feature box is either a *free feature box* or a *bound feature box*. Free feature boxes are transient and interchangeable. When a router must incorporate a free feature box into a usage, it creates a new instance of the feature box type. Bound feature boxes are persistent and dedicated to particular addresses. When a router must incorporate a bound feature box into a usage, only the unique box of appropriate type bound to the appropriate address may be chosen.

```
partition FFBox, BFBox: FBox
```

An error box is transient and interchangeable. Its purpose is to handle addressing errors that arise during routing (see Section 6.5). The box simply accepts a call, sends the signal *unknown* upstream, and tears down the call.

## 2.2 Box types (Nov 03)

A *box type* is either an *interface box type, feature box type*, or *error box type*. A *feature box type* is either a *free feature box type* or a *bound feature box type*.

```
domain { BoxType }
EBoxType: BoxType !
partition IBoxType, FBoxType, EBoxType: BoxType
partition FFBoxType, BFBoxType: FBoxType
```

Each box has a permanent type in *BoxType*:

```
boxType: Box -> static BoxType !

all b: IBox | b.boxType in IBoxType
all b: FFBox | b.boxType in FFBoxType
all b: BFBox | b.boxType in BFBoxType
all b: EBox | b.boxType = EBoxType
```

## 2.3 Addresses (Nov 03)

Addresses point to line, trunk, and resource interface boxes. Addresses are also used as mobile subscriber addresses that are not permanently associated with any interface box. An address that is currently mapped to an interface box or is a current mobile subscriber's address is said to be *in use*.

The addresses in use for these purposes belong to four disjoint subsets *LAddress*, *TAddress*, *RAddress* and *MAddress*. Each DFC network's address set also contains a distinguished null value that is never in use and belongs to none of the four subsets.

```
domain { Address }
disjoint LAddress, TAddress, RAddress, MAddress: Address

noAddr: fixed Address !
noAddr in Address - (LAddress + TAddress + RAddress + MAddress)
```

## 2.4 Alphabets (May 01)

Each DFC network uses two symbol alphabets, one for addressing and one for signaling. The addressing alphabet *AAlphabet* is a proper subset of the signaling alphabet *SAlphabet*. This relationship makes it possible to spell out all addresses using signals or sequences of signals, and still have some symbols left over for delimiters.

A member of *AString* is a sequence of symbols from *AAlphabet*. A member of *SString* is a sequence of symbols from *SAlphabet*.

The set *Address* of legal addresses in a DFC network is a subset of *AString* defined by syntactic restrictions.

Although members of *SString* and *Address* are not atomic and are structurally related, the formal description treats them as atomic and unrelated. Like *Address*, *SString* needs a distinguished null value.

```
domain { fixed SString }

noString: fixed SString !
```

## 2.5 Address mappings (Nov 03)

Address mappings are global data used for routing. They are initialized data of a DFC network. There are three address mappings: *LMap*, *TMap*, and *RMap*.

*LAddress* is the set of addresses that uniquely identify line interface boxes. *LMap* is an address mapping from members of *LAddress* to members of *LIBox*. *LMap* is a bijection, or, in other words, it is invertible and onto.

```
LMap: LAddress ! -> LIBox !
```

*TAddress* is the set of addresses known to the DFC network but belonging primarily to other telecommunication networks, and reachable only through

them via trunk interface boxes. Within DFC, *TMap* maps some members of *TAddress* nondeterministically to trunk interface boxes: many members of *TAddress* can be reached via one trunk interface box, and many trunk interface boxes can be used to reach one member of *TAddress*.

```
TMap: TAddress -> TIBox
```

If a trunk interface box is not in the range of *TMap*, then it is supporting only trunks for inbound calls. If a member of *TAddress* is not in the domain of *TMap*, then it is only useful as a source of inbound calls, and the only purpose of making it known to the DFC network is to allow it to subscribe to source-zone feature boxes.

*RAddress* is the set of addresses used by box programmers to reach media-processing resources. A member of *RAddress* names *a type of resource* rather than an individual resource. The type corresponds, in turn, to a fixed programming interface, which is all that a box programmer needs to know. *RMap* maps a member of *RAddress* to some *RIBox* of the appropriate type. It is total. An *RIBox* can have several addresses, if it implements several different programming interfaces.

```
RMap: RAddress + -> RIBox +
```

There is no address mapping for the set *MAddress* of mobile addresses. These addresses have no permanent associations with interface boxes, and thus require no address mapping.

### 2.6 Feature box addresses (Nov 03)

Each feature box has an address. This address is the subscribing address on whose behalf it is created and/or assembled into a usage (see Section 6.5). A bound box type has at most one instance with a particular address.

```
boxAddr: FBox ->
         static (LAddress + TAddress + RAddress + MAddress) !

all t: BFBoxType | all a: Address |
   sole b: Box | b.boxType = t && b.boxAddr = a
```

## 3 Features and subscriptions (Nov 03)

### 3.1 Regions and zones (Nov 03)

A *source region* of a usage is associated with a source address. A feature box is incorporated into a usage in a source region because the source address subscribes to the feature box in the source region, which means that the box is relevant to calls in which that address identifies the caller. A Speed Dialing feature box is always found in a source region.

A *target region* of a usage is associated with a target address. A feature box is incorporated into a usage in a target region because the target address subscribes to the feature box in the target region, which means that the box is relevant to calls in which that address identifies the callee. A Call Forwarding feature box is always found in a target region.

```
domain { fixed Region }
partition srcRegn, trgRegn: fixed Region !
```

In general, a region can consist of several *zones*. Each zone of a source region contains all the feature boxes subscribed to in the source region by a particular source address. There can be more than one zone in the source region because there can be more than one source address in the usage. Each zone of a target region contains all the feature boxes subscribed to in the target region by a particular target address. There can be more than one zone in the target region because there can be more than one target address in the usage.

When used in the most straightforward way, the DFC routing algorithm generates usages with one source region followed by one target region. Real usages can be more complex, because they emerge from the actions of multiple feature boxes employing the routing algorithm on multiple calls, and connecting usage fragments together however they please.

## 3.2 Features and feature box types (Nov 03)

Features are provided by feature boxes.

The function *providesFeature* indicates which feature box types belong to and are required to implement a feature. Every feature has at least one box type.

```
domain { Feature }

providesFeature: FBoxType + -> Feature !
```

Some feature box types are *reversible*.

```
reversible: FBoxType
```

A feature box type must be reversible if it uses the *reversedSetupSignal* method or if it is bound and it can be subscribed to in both regions. A Call Waiting box is reversible.

## 3.3 Routing choices (Nov 03)

The feature box types that can be routed to in the source region are specified by the set *srcChoice*. The feature box types that can be routed to in the target region are specified by the set *trgChoice*.

```
srcChoice: FBoxType
trgChoice: FBoxType
```

Each reversible box type must be available in both regions. As stated above, if a box type is bound and available in both regions, then it must be in reversible.

```
reversible in srcChoice && reversible in trgChoice

all t: BFBoxType |
(t in srcChoice && t in trgChoice) ==> t in reversible
```

## 3.4 Precedence (Nov 03)

There are two *precedence* relations, which constrain the order of feature boxes within zones.

```
srcPrecedes: srcChoice -> srcChoice
trgPrecedes: trgChoice -> trgChoice
```

Each *precedes* relation is a partial order.

The *precedes* relations on reversible box types are more constrained. First, each relation is a total order on reversible box types. Second, the total order in the target region is the reverse of the total order in the source region.

```
all t1, t2: reversible |
   (t1 in t2.srcPrecedes && t2 in t1.trgPrecedes) ||
   (t2 in t1.srcPrecedes && t1 in t2.trgPrecedes)
```

## 3.5 Subscriptions (Nov 03)

Addresses in use can subscribe to feature box types in both the source and target regions.

```
srcSubscribes: Address -> srcChoice
trgSubscribes: Address -> trgChoice

all a: Address |
    a !in (LAddress + TAddress + RAddress + MAddress)
==> a.srcChoice = emptySet && a.trgChoice = emptySet
```

If an address subscribes to a bound box type, then there must be a box of that type bound to it, and *vice versa*.

```
all a: Address | all t: BFBoxType |
    (t in a.srcSubscribes || t in a.trgSubscribes)
<=> (some b: BFBox | b.boxAddr = a && b.boxType = t)
```

If an address subscribes to a reversible box type in one region, then it must subscribe to it in the other region.

```
all a: Address | all t: reversible |
   t in a.srcSubscribes <=> t in a.trgSubscribes
```

Every address has a *srcZone* and a *trgZone* containing its source and target choices, respectively.

```
srcZone: Address -> BoxTypeSeq !
trgZone: Address -> BoxTypeSeq !

all a: Address | all t: FBoxType |
   (t el a.srcZone <=> t in a.srcChoice)
&& (t el a.trgZone <=> t in a.trgChoice)
```

A *srcZone* or *trgZone* is a sequence with no duplicates. The order of box types in a *srcZone* is consistent with the partial order established by *srcPrecedes*. The order of box types in a *trgZone* is consistent with the partial order established by *trgPrecedes*.

# 4 Signals (Nov 03)

The messages of the DFC protocol are called *signals*.

```
domain { Signal }
```

A *signal* has a *signal type* and some set of named, typed *fields*. Some signal types will be specified as having particular required or optional fields. Any signal can have extra, programmer-defined fields that are not specified by DFC.

# 5 How boxes affect routing (Nov 03)

## 5.1 Basic concepts (May 01)

A *usage* is a dynamic assembly of interface and feature boxes connected by internal calls. To place an internal call, a box prepares a *setup signal* and sends it to a *DFC router*. The DFC router, which is *stateless* in the sense that it maintains no records of usages, applies the routing algorithm to the setup signal, and finds a box to which it routes the internal call. If the box accepts the call, then the usage has grown by one call and one box.

Thus the growth of a usage is affected both by box programs, which prepare setup signals, and the routing algorithm, which routes them to boxes. This section presents what box programs can do to prepare setup signals. Section 6 presents the routing algorithm.

## 5.2 The setup signal (Nov 03)

Each signal with type *setup* has the following fields:

```
setup: Signal

regn: setup -> Region !
```

```
src: setup -> Address !
dld: setup -> SString !
trg: setup -> Address !
route: setup -> (ZoneTag + BoxTypeSeq) !
```

where *ZoneTag* is enumerated:

```
domain { fixed ZoneTag }
partition whole, suffix: fixed ZoneTag !
```

In addition, some setup signals have the following fields:

```
placing: setup -> BoxType ?
outer: setup -> Address ?
```

The meaning of the *regn* field is explained in Section 3.1. The fields *src* and *trg* contain the notional source and target addresses of the usage. In a simple usage, the setup signal of every call might contain the same two addresses in these fields. In more complex situations, however, many feature boxes could change these addresses as a usage unfolds. Similarly, the field *dld* usually contains the string dialed by the user originating the usage, but it, too, can be modified by features.

The field *route* is a record used by the routing algorithm. The field *placing* is the box type of the caller box that sends the setup signal; it is only needed in certain circumstances (see Section 5.6).

The field *outer* is the address of the placing box, as defined in Sections 2.5 and 2.6. It is only needed in certain circumstances (see Sections 5.4, 5.5, 5.6).

## 5.3 Access to setup signals by boxes (Nov 03)

A box program can examine the fields of a received setup signal through a programming interface. The only predefined fields that a box program can examine are *regn, src, dld, trg,* and *outer*. A box program can also examine extra fields that have been added to setup signals by other boxes.

Note that a box can use these fields to determine its own subscriber. If the value of *regn* is *srcRegn,* then the box's subscriber's address is the value of *src*. If the value of *regn* is *trgRegn,* then the box's subscriber's address is the value of *trg*.

To place an internal call, a box must prepare a setup signal for it using one of three methods. These methods are specified in the next three subsections. As the specifications will show, the box program does not have direct control over every field.

Although it is not repeated in the next three subsections, each method allows a box program to add or delete extra fields of a setup signal, or to change the value of an extra field that was provided by another box. If a method prepares a new setup signal from an old setup signal, then all extra fields of the old setup signal that have not been explicitly deleted or changed are copied into the new setup signal.

## 5.4 New setup signals (Nov 03)

The *newSetupSignal* method constructs a completely new setup signal *de novo*: that is, it does not require the presence in the box of a previously received setup signal.

If an interface box places a call, it must use *newSetupSignal*. As a general rule, feature boxes should only use this method when they are setting up calls to resources or when they are truly acting as endpoints.

In the specification of the method, *b* is the box invoking the method, and *s* is the resulting setup signal. The box program's only choices are the arguments *dldArg* and *trgArg*. No *placing* field is needed in *s*.

```
s.regn = srcRegn
b in FBox ==> s.src = b.boxAddr
b in LIBox ==> s.src = b.~LMap
b in TIBox ==> s.src in TAddress
s.dld = dldArg
s.trg = trgArg
s.route = whole
s.outer = s.src
```

Because of various odd situations that arise in the real world, the *src* field cannot be constrained very closely when the placing box is a trunk interface. Resource interface boxes do not place calls.

## 5.5 Continued setup signals (Nov 03)

The *continuedSetupSignal* method creates a new setup signal by modification of a setup signal that has been previously received and stored by the box. This method is never used by an interface box.

*continuedSetupSignal* is by far the most commonly used method. A feature box uses this method to continue and extend a chain of internal calls, all being set up in the same direction.

In the specification of the method, *b* is the box invoking the method, *ss* is the stored setup signal, and *s* is the resulting setup signal. The box program's only choices are the arguments *srcArg, dldArg* and *trgArg*.

In this context, an argument value *noAddr* means "no change". There is a precondition on these arguments, because a box program cannot be allowed to specify a change, yet leave the routing address unchanged.

```
ss.regn = srcRegn && srcArg != noAddr ==> srcArg != b.boxAddr
ss.regn = trgRegn && trgArg != noAddr ==> trgArg != b.boxAddr
```

The specification of the method is as follows:

```
s.regn = ss.regn

s.regn = srcRegn && srcArg != noAddr ==>
```

```
   s.src = srcArg && s.route = whole && s.outer = b.boxAddr
s.regn = srcRegn && srcArg = noAddr ==>
   s.src = b.boxAddr && s.route = ss.route
s.regn = trgRegn && trgArg != noAddr ==>
   s.trg = trgArg && s.route = whole
s.regn = trgRegn && trgArg = noAddr ==>
   s.trg = b.boxAddr && s.route = ss.route

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

dldArg != noString ==> s.dld = dldArg
dldArg  = noString ==> s.dld = ss.dld
```

No *placing* field is needed in *s*. An *outer* field is needed only if the setup is in the source region and the source address is changing. In this case the field can be used for authentication by the feature box that receives the call.

### 5.6 Reversed setup signals (Nov 03)

The *reversedSetupSignal* method creates a new setup signal by modification of a setup signal that has been previously sent, yet is still stored by the box. This method can only be used by a reversible feature box.

A feature box uses the *reversedSetupSignal* method to place an outgoing call that reverses a previous outgoing call. In the modified setup signal, the region has changed, and the source and target roles are reversed. This is needed when a complete end-to-end connection path contains internal calls placed in different directions, which often happens when segments of a path are shared or reused. The various restrictions on reversible box types, and the way that DFC routers handle reversed setup signals, are all designed to ensure an orderly progression of feature boxes in such paths [6].

For example, consider a Mid-Call Move feature box routed to in the source region. Until triggered, it simply receives an incoming internal call and continues it transparently. When it is triggered by a signal from the subscriber, its function is to place a new outgoing internal call to a telephone the subscriber would like to be using. Once the subscriber has answered the new telephone and hung up the original telephone, the move is complete. Now the end-to-end path between the subscriber and the far party has a reused segment and a new segment, set up in different directions and connected in the Mid-Call Move box.

In the specification of the method, *b* is the box invoking the method, *ss* is the stored setup signal, and *s* is the resulting setup signal. The box program's only choices are the arguments *srcArg, dldArg* and *trgArg*.

In this context, an argument value *noAddr* means "no change". There is a precondition on these arguments, because a box program cannot be allowed to

specify a change, yet leave the routing address unchanged. All together, the method has the following preconditions.

```
b.boxType in reversible

ss.regn = trgRegn && srcArg != noAddr ==> srcArg != b.boxAddr
ss.regn = srcRegn && trgArg != noAddr ==> trgArg != b.boxAddr
```

The specification of the method is as follows:

```
ss.regn = srcRegn ==> s.regn = trgRegn
ss.regn = trgRegn ==> s.regn = srcRegn

s.regn = srcRegn && srcArg != noAddr ==>
   s.src = srcArg && s.route = whole && s.outer = b.boxAddr
s.regn = srcRegn && srcArg = noAddr ==>
   s.src = b.boxAddr && s.route = suffix &&
   s.placing = b.boxType
s.regn = trgRegn && trgArg != noAddr ==>
   s.trg = trgArg && s.route = whole
s.regn = trgRegn && trgArg = noAddr ==>
   s.trg = b.boxAddr && s.route = suffix &&
   s.placing = b.boxType

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

dldArg != noString ==> s.dld = dldArg
dldArg = noString ==> s.dld = ss.dld
```

The *placing* field is needed only if the value of *route* is *suffix*. An *outer* field is needed only if the resulting setup is in the source region and the source address is changing. In this case the field can be used for authentication by the feature box that receives the call.

# 6 The routing algorithm (Nov 03)

## 6.1 Basic concepts (Nov 03)

For each internal call, the setup signal created by an interface or feature box is sent to any DFC router. DFC routers are interchangeable.

The router modifies the setup signal and chooses a callee box to which the call is routed. The router always succeeds in choosing a callee box: if necessary it chooses a box whose sole function is to diagnose such failures as an unobtainable address.

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

```
DFCRoutingAlgorithm(out, inn: setup !, box: Box !)
```

The parameters *out* and *inn* represent the setup signals sent by the caller port and received by the callee port, respectively. The parameter *box* is the callee box.

The operation of DFC routers on a setup signal consists of four steps, specified in the next four subsections. Intermediate results produced by the first three steps are specified using local setup variables *st1, st2, st3*. So the five setup signals *out, st1, st2, st3, inn* are actually implemented as one setup signal whose fields change as it goes through the routing process.

## 6.2 Step 1: Extract target (Nov 03)

The user who initiates a usage can produce a dialed string. This string might indicate a target address, but that address might be encoded in the string along with other information. If so, it must be extracted from the string.

The partial function *embeddedAddress* finds the address embedded in a string, if there is any.

```
embeddedAddress: SString -> Address ?
```

This function is used in Step 1 to supply a target address, if no target address is already present. The specification of Step 1 is:

```
st1.regn = out.regn
st1.src = out.src
st1.dld = out.dld
st1.route = out.route
st1.placing = out.placing
st1.outer = out.outer

(out.trg != noAddr || dld.embeddedTarget  = emptySet)
   ==> st1.trg = out.trg
(out.trg =  noAddr && dld.embeddedTarget != emptySet)
   ==> st1.trg = dld.embeddedTarget
```

## 6.3 Step 2: Expand zone (Nov 03)

Step 2 expands a *ZoneTag* into a whole or partial zone, which is the sequence of box types that are expected to be next in the usage. The local variable *st1* holds the input to this step. The local variable *st2* holds the output from this step. The specification of Step 2 is:

```
st2.regn = st1.regn
st2.src = st1.src
```

```
st2.dld = st1.dld
st2.trg = st1.trg
st2.outer = st1.outer
st2.placing = emptySet

st1.route in BoxTypeSeq ==> st2.route = st1.route
st1.regn = srcRegn && st1.route = whole
   ==> st2.route = st1.src.srcZone
st1.regn = trgRegn && st1.route == whole
   ==> st2.route = st1.trg.trgZone
st1.regn = srcRegn && st1.route = suffix
   ==> st2.route = st1.src.srcZone.suffixAfter[st1.placing]
st1.regn = trgRegn && st1.route = suffix
   ==> st2.route = st1.trg.trgZone.suffixAfter[st1.placing]
```

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.

Recall from Section 5 that *route* is *suffix* only when the setup signal has been created in a reversible box by the *reversedSetupSignal* method. The placing box type appears in both zones of the address relevant to *regn*. The result of taking the suffix of the zone is exactly what would have been in *st1.route*, if this setup signal had been created by the *continuedSetupSignal* method.

### 6.4 Step 3: Advance region (Nov 03)

If the source region is exhausted, Step 3 advances the region from source to target. The local variable *st2* holds the input to this step. The local variable *st3* holds the output from this step. The specification of Step 3 is:

```
st3.src = st2.src
st3.dld = st2.dld
st3.trg = st2.trg
st3.placing = emptySet

st2.regn = srcRegn && st2.route = emptySeq
   ==> st3.regn = trgRegn && st3.route = st2.trg.trgZone &&
       st3.outer = emptySet
st2.regn = trgRegn || st2.route != emptySeq
   ==> st3.regn = st2.regn && st3.route = st2.route
       st3.outer = st2.outer
```

If the region is advanced, then *route* becomes the target zone of the target address. The *outer* field is removed because its information should not propagate across the region boundary [6].

## 6.5 Step 4: Choose callee box (Nov 03)

The fourth and final step of routing is to choose a callee box and send the modified setup signal to it. The final setup signal is held by the parameter *inn*, while the callee box is held by the parameter *box*. The specification also uses local variables *bt*, which is the type of the callee box, and *addr*, which is the address of the callee box.

¿From the specification of Step 3 we can deduce:

```
st3.route = emptySeq ==> st3.regn = trgRegn
```

This is useful in understanding the specification of Step 4 (the sequence attributes *head* and *tail* have their usual meanings):

```
inn.regn = st3.regn
inn.src = st3.src
inn.dld = st3.dld
inn.trg = st3.trg
inn.placing = emptySet
inn.outer = st3.outer

st3.regn = srcRegn ==>
   inn.route = st3.route.tail && bt = st3.route.head &&
   addr = st3.src
st3.regn = trgRegn && st3.route != emptySeq ==>
   inn.route = st3.route.tail && bt = st3.route.head &&
   addr = st3.trg
st3.route = emptySeq ==>
   inn.route = emptySeq && bt in IBoxType && addr = st3.trg

bt in FBoxType ==>
   box in FBox && box.boxType = bt && box.boxAddr = addr

bt in IBoxType && addr in LAddress ==> box = addr.LMap
bt in IBoxType && addr in TAddress ==>
   (addr.TMap != emptySet ==> box in addr.TMap) &&
   (addr.TMap  = emptySet ==> box in EBox)
bt in IBoxType && addr in RAddress ==> box in addr.RMap
bt in IBoxType && addr in MAddress ==> box in EBox
bt in IBoxType &&
   addr in Address - (LAddress + TAddress + RAddress + MAddress)
   ==> box in EBox
```

If there is a nonempty route, this step routes to a feature box whose type matches the box type at the head of the route. In the source region, the address of this box is the source address; in the target region, the address of this box is the target address. If the box type is bound, the box is uniquely determined.

If the route is empty, this step tries to route to an interface box mapped to by the target address. If this is not possible, it routes to an error box instead.

## 6.6 Properties of DFC routing (Nov 03)

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

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

inn.src = out.src
inn.dld = out.dld
inn.trg = st1.trg

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

inn.placing = emptySet
inn.regn = trgRegn ==> inn.outer = emptySet

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

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.

An *ideal connection path* is a path of internal calls, linked together in slightly restricted ways (see below) by feature boxes. Here are some of the theorems that have been proved about ideal connection paths [6], stated briefly and informally:

- A path has at most one call in which Step 3 advances the region (this is a *midpoint call*).
- A path that connects two interface boxes has a midpoint call.
- A path containing a midpoint call has no cycles.
- In any zone (contiguous feature boxes of one address), unless the zone is truncated by address translation (address translation occurs when a box continues or reverses a setup signal, changing the source or target address as it does so), ordering the boxes from innermost to outermost (from closest to the midpoint to closest to an interface box), the sequence of reversible box types is fixed.

Note that a zone of an ideal connection path can contain internal calls set up in alternating directions, so the last theorem is a strong statement about the orderliness of DFC routing.

An ideal connection path cannot include calls to resources, and cannot include some intra-box links created by conferencing. For example, suppose that

user $a$ has end-to-end paths to users $b$ and $c$. First $a$ is switching between the two paths, and then $a$ conferences them together. The end-to-end path between $a$ and $b$ is ideal, and the end-to-end path between $a$ and $c$ is ideal, but the end-to-end path between $b$ and $c$ created by conferencing is not ideal.

# 7 Operational data (Nov 03)

*Operational data* is persistent data that can be read and written by feature boxes. Operational data is partitioned into relations; *OpRelation* is the set of all such relations. Operational data is categorized in two ways:

- If a relation is in *featurePartitioned*, it supports a particular feature.
- If a relation is in *customerPartitioned*, each of its tuples describes and belongs to a particular customer.

All operational data relations must be either feature-partitioned or customer-partitioned, and many are both.

```
domain { OpRelation }

featurePartitioned: OpRelation
customerPartitioned: OpRelation

OpRelation = featurePartitioned + customerPartitioned

supportsFeature: featurePartitioned -> Feature !
```

Access to operational data by feature boxes is restricted in two ways. The first restriction is that a feature box $b$ can only access a data relation $d$ if the data is not feature-partitioned, or if it is feature-partitioned and $b$ is a box of the correct feature:

```
(d in OpRelation - featurePartitioned) ||
(b.boxType.providesFeature = d.supportsFeature)
```

The second restriction is based on a provisioned ownership relationship between customers and addresses. An address need not be owned by a customer. If it is owned, however, it can only be owned by one customer. Each customer must own at least one address.

```
domain { Customer }

owner: (LAddress + TAddress + MAddress) + -> Customer ?
```

If $d$ *in customerPartitioned*, then the tuples of $d$ belonging to customer $c$ can be accessed by a feature box $b$ only if:

```
b.boxAddr = a && a.owner = c
```

# 8 The call protocol (Nov 03)

## 8.1 Basic concepts (Nov 03)

Boxes use the call protocol to set up and tear down internal calls.

The primary signals types used in the call protocol are:

```
mtype = { setup,upack,
          other,
          teardown,downack }
```

The placeholding signal type *other* represents additional signal types that will be introduced later. Some are part of the call protocol, and some are part of the media channel protocol.

Every signal has a *tunnel* field. The value of a tunnel field is a non-negative integer. Signals referring specifically to media channel $n$ have $n$ in their tunnel fields. Signals referring to the call as a whole have zero in their tunnel fields.

This version of the call protocol is written in Promela, for a reference implementation based on packet-switched networks. A reference implementation more suitable for circuit-switched networks can be found in [5].

In this Promela program there is a queue, containing incoming signals, for every location that can receive signals. These locations are either ports on boxes, boxes themselves, or DFC routers. The queues are arranged in an array *to*, with indices of type *byte* ranging from zero to *locsize - 1*. Queue indices are henceforth simply referred to as *queues*.

```
chan to[locsize] = [chansize] of {byte,mtype,byte}
```

In the Promela program, each signal has three fields:

- The first holds the queue of the location from which it was originally sent;
- the second holds the signal type;
- the third holds the tunnel.

## 8.2 Router protocol (Mar 01)

A DFC router does nothing but receive setup signals from calling ports and forward them to appropriate boxes. In the following program for a router, the argument *thisrouter* supplies the router's queue. The placeholder *box* represents the router's current choice of box.

```
proctype router(byte thisrouter)
{ byte from;
  end_begin:  do
              :: to[thisrouter]?from,setup,0;
                 to[box]!from,setup,0
              od
}
```

A DFC network can have any number of routers. In subsequent programs, *router* represents the queue of any router.

## 8.3 Caller port protocol (Dec 02)

Next we present a program for a port that attempts to place a call. The argument *thisport* supplies the port's own queue.

```
proctype caller_port
         (byte thisport)
{ byte farport,tunnel;
end_idle:   to[router]!thisport,setup,0;
            to[thisport]?farport,upack,0;
linked:     do
            :: to[farport]!thisport,other,tunnel
            :: to[thisport]?eval(farport),other,tunnel
            :: to[farport]!thisport,teardown,0; goto unlinking
            :: to[thisport]?eval(farport),teardown,0;
               to[farport]!thisport,downack,0; goto end_idle
            od;
unlinking:  do
            :: to[thisport]?eval(farport),other,tunnel
            :: to[thisport]?eval(farport),teardown,0;
               to[farport]!thisport,downack,0
            :: to[thisport]?eval(farport),downack,0; goto end_idle
            od
}
```

The caller port first sends its setup signal to a router, which has the only queue (besides its own) that it knows. When the response of type *upack* comes back, the call is set up. The queue of origin of the *upack* signal is bound to the local variable *farport*. The difference between the two uses of the variable *farport* in the statements:

```
to[thisport]?farport,upack,0;
to[thisport]?eval(farport),other,tunnel
```

is that the first receive action binds *farport* to the value in the first field of the signal received, while the second receive is only executable if the first field of the signal received matches the current value of *farport*.

For the rest of the call, there is a signaling channel between the two participating ports *farport* and *thisport*. Any sequence of *other* signals can be sent on it, in either direction.

Either port can initiate the teardown sequence by sending a *teardown* signal. The port then enters the *unlinking* state, in which it reads leftover signals until it finally receives a *downack* signal from the other port, which must be the final signal it will receive in this call.

The recipient of the *teardown* signal must send a *downack* signal to acknowledge it. Equally important, once it receives a *teardown*, it must send no other signals in this call except the *downack*.

Both ports can send *teardown* signals at about the same time, so that the signals cross in transmission. A port that receives a *teardown* after sending its own simply acknowledges it and then continues to wait for the acknowledgment of its own.

## 8.4 Callee port protocol (Dec 02)

Each box has a distinguished port *thisbox*, which receives all *setup* signals for the box. The box responds to each *setup* signal by allocating to it an idle callee port, and handing it off to that port. The callee port completes the protocol for that call.

This program gives the protocol for a callee port. The argument *thisport* supplies the queue of the callee port.

```
proctype callee_port
         (byte thisport)
{ byte farport,tunnel;
end_idle:   "handoff from box port";
            to[farport]!thisport,upack,0
linked:     do
            :: to[farport]!thisport,other,tunnel
            :: to[thisport]?eval(farport),other,tunnel
            :: to[farport]!thisport,teardown,0; goto unlinking
            :: to[thisport]?eval(farport),teardown,0;
               to[farport]!thisport,downack,0; goto end_idle
            od;
unlinking:  do
            :: to[thisport]?eval(farport),other,tunnel
            :: to[thisport]?eval(farport),teardown,0;
               to[farport]!thisport,downack,0
            :: to[thisport]?eval(farport),downack,0; goto end_idle
            od
}
```

Once the callee port has sent the *upack*, the call is established and is symmetric. ¿From their *linked* states onwards, the programs for caller and callee ports are identical.

## 8.5 Chains of related calls (Nov 03)

Internal calls are usually set up and torn down in chain reactions. This section concerns the treatment of internal calls that are related to each other by being parts of the same chain. The primary goal of chaining several internal calls is to make them look like one call to an external observer.

The first obligation of a box with respect to two chained calls is *piecewise setup*: when a box accepts a *setup* signal, it should send the corresponding *upack* signal, thus completing call setup, before it attempts to set up the next call in the chain.

It is permissible to add extra fields to *setup* signals. Such fields can propagate status information along chains of related calls, carrying information that might arrive too late if it came in the form of separate status signals. The second obligation of a box with respect to two chained calls is to propagate extra fields from the incoming *setup* signal to the outgoing *setup* signal. This obligation is satisfied automatically by the *continuedSetupSignal* and *reversedSetupSignal* methods. Furthermore, the programming interface described in Section 5.3 gives feature boxes access to extra status fields.

The third obligation of a box with respect to chained calls is *piecewise teardown*: when a box receives a *teardown* signal, it should send the corresponding *downack* signal, thus completing call teardown, before it tears down the next call in the chain.

Like *setup* signals, *teardown* signals can conceivably propagate useful status information along a chain of related calls. Clearly such information cannot be sent later. The fourth obligation of a box with respect to two chained calls is to propagate extra fields from the incoming *teardown* signal to the outgoing *teardown* signal.

There is no point to adding extra fields to *upack* or *downack* signals, so it is not allowed. The futility follows directly from piecewise setup and teardown, which ensure that the same acknowledgment signal cannot be propagated along a chain of calls.

## 8.6 Call-level status signals (Jan 03)

Once a call has been set up, its signaling channel can be used to send *status signals* in either direction. Signals referring to the call as a whole, rather than to a media channel of it, have zero in their tunnel fields.

At the call level, there are four built-in status signals. Although each signal is meaningful within the call in which it is observed, it is often generated by an interface or feature box further down a chain of setup-related calls, and merely propagated through the chain.

- The signal *unknown* indicates that the target address of the call does not map to any interface box. The signal is usually generated by an error box.
- The signal *avail* indicates that the target of the call is available and the call is deemed to be successful. The signal is often generated by an interface box.
- The signal *unavail* indicates that the target of the call is not available. The signal is often generated by an interface box.
- The signal *none* cancels the effect of any of the three previous signals on a user interface. The signal is only generated by feature boxes.

Typically these signals travel from the callee port to the caller port of an internal call. However, the various actions of feature boxes can undermine these expectations. As a result, there are no built-in constraints on how many of these signals there are in a usage, or where they travel within the usage.

Other special-purpose status signals can be added as needed. A status signal can carry any set of fields whatsoever.

## 8.7 Properties of the call protocol (Dec 02)

**Embedding** The protocols for box ports and routers can be *embedded* in protocols that send and receive additional signals, provided that their other functions do not interfere with the timely execution of the programs given here. The additional signals might be useful for implementing or optimizing DFC. Caller and callee ports, on the other hand, are specified exhaustively. No component of a DFC network can send them signals, except as described here.

**Absence of deadlock** The call protocol specified here cannot deadlock, as verified by use of the Spin model checker [3]. Note that with the embedding constraint above in force, the pattern matching imposed by the use of *eval(farport)* is unnecessary. Note also that, to enjoy this property, a box program must be faithful to it in three particulars:

1. The box is always able to read the input queue of its box port, except for short intervals of time.
2. Whenever a caller or callee port is busy, the box is always able to read its input queue, except for short intervals of time.
3. Whenever the box reads the input queue of a caller or callee port, it can accept any signal the protocol says can be received in the current port state.

A box that satisfies these three requirements is said to be *input-enabled*.

**Reliable FIFO signaling** The protocol provides reliable signaling between the ports of a call. In other words, the sequence of *other* signals received by the callee port during a call is exactly the sequence sent by the caller port during the call. Also, the sequence of *other* signals received by the caller port during a call is exactly the sequence sent by the callee port during the call. This property also has been verified with the Spin model checker.

The following table summarizes certain properties of the four functional signal types of the call protocol. For each signal type, we show the possible senders and receivers. The lower part of the table gives information about the possible fields of these signals: all have required origin fields, all have tunnel fields equal to zero, only *setup* signals have routing fields, and only some signals can have additional status fields.

| | setup | upack | teardown | downack |
|---|---|---|---|---|
| *senders* | *caller port, router* | *callee port* | *caller port, callee port* | *caller port, callee port* |
| *receivers* | *router, box port* | *caller port* | *caller port, callee port* | *caller port, callee port* |
| internal-Origin | *required* | *required* | *required* | *required* |
| tunnel | 0 | 0 | 0 | 0 |
| *routing fields* | *required* | *forbidden* | *forbidden* | *forbidden* |
| *additional status fields* | *allowed* | *forbidden* | *allowed* | *forbidden* |

Note that when a *setup* signal is being sent from a caller port to a router, its *internalOrigin* field identifies the caller port. When the signal is forwarded by the router to a box port, its *internalOrigin* field still identifies the caller port.

# 9 The media channel protocol (Nov 03)

## 9.1 Basic concepts (Dec 02)

A call can have any number of two-way channels of any medium. This makes it necessary to add a protocol to open and close media channels explicitly, because without it there would be no way to tell which media channels a call is using. Each signal of type *open* carries a field *medium,* indicating the medium of the channel being requested.

The new vocabulary of signals, encompassing the functional signals of both the call and channel protocols, is:

```
mtype = { setup,upack,
          open,oack,onack,ready,
          status,
          close,closeack,
          teardown,downack }
```

The signals mentioned in the call protocol (see Section 8) as having type *other* are now seen to be either functional signals of the media channel protocol or *status* signals. The signals related to each media channel of a call travel through their own tunnel of the signaling channel. Status signals can refer either to a media channel or to the call as a whole.

The channel protocol is nested inside the call protocol, as will be explained. To preserve the familiar structure of the call protocol, and the analogous structure of the channel protocol, we express them as distinct Promela processes. These processes communicate through shared (paired) events, expressed in Promela as reads and writes to buffers of zero capacity.

Since zero-buffered communication between the processes is synchronous, the process decomposition does not affect the ordering of events at each port

of a call in any way. The process decomposition should neither be regarded as part of the protocol, nor as a constraint on the implementation.

## 9.2 The augmented call protocol (Dec 02)

Of the programs in Section 8, only the caller-port and callee-port programs need to change.

The caller-port program has two additional arguments for each possible channel; they are the buffers (declared elsewhere as having zero capacity) through which a caller-port process communicates with the process performing the protocol for that channel.

```
proctype caller_port
         (byte thisport; chan in1,out1,in2,out2,...)
{ byte farport;
  mtype m;
end_idle:   to[router]!thisport,setup,0;
            to[thisport]?farport,upack,0;
            in1!upack; in2!upack; ...;
linked:     do
            :: to[thisport]?eval(farport),status,0
            :: to[thisport]?eval(farport),m,1; in1!m
            :: to[thisport]?eval(farport),m,2; in2!m
            :: ...
            :: to[farport]!thisport,status,0
            :: out1?m; to[farport]!thisport,m,1
            :: out2?m; to[farport]!thisport,m,2
            :: ...
            :: to[farport]!thisport,teardown,0; goto unlinking
            :: to[thisport]?eval(farport),teardown,0;
               to[farport]!thisport,downack,0;
               in1!downack; in2!downack; ...; goto end_idle
            od;
unlinking:  do
            :: to[thisport]?eval(farport),status,0
            :: to[thisport]?eval(farport),m,1; in1!m
            :: to[thisport]?eval(farport),m,2; in2!m
            :: ...
            :: to[thisport]?eval(farport),teardown,0;
               to[farport]!thisport,downack,0
            :: to[thisport]?eval(farport),downack,0;
               in1!downack; in2!downack; ...; goto end_idle
            od
}
```

This program assumes that there is a separate channel process for each possible media channel. In the requesting state, if a caller-port process receives

an *upack* signal through the call protocol, it also sends an *upack* signal to each of these channel processes. This informs them that a call now exists. Note that the signals in these local buffers are of type *mtype*; they have no need for origin queues or tunnels.

In its linked state, the caller-port process can receive signals in channel tunnels. Such a signal is forwarded immediately, through a shared event such as *in1!m*, to the process performing the protocol for the appropriate channel.

In its linked state the process can also receive signals from a channel process, intended to be sent out on the signaling channel of the call. They are received through shared events such as *out1?m*. Such a signal is sent out immediately on the signaling channel of the call, with the appropriate tunnel field.

When a call is completely finished, a caller-port process sends a downack signal to each of its channel processes. This informs them that a call no longer exists.

The callee-port process is augmented in similar ways.

```
proctype callee_port
         (byte thisport; chan in1,out1,in2,out2,...)
{ byte farport;
  mtype m;
end_idle:   "handoff from box port";
            to[farport]!thisport,upack,0;
            in1!upack; in2!upack; ...;
linked:     do
            :: to[thisport]?eval(farport),status,0
            :: to[thisport]?eval(farport),m,1; in1!m
            :: to[thisport]?eval(farport),m,2; in2!m
            :: ...
            :: to[farport]!thisport,status,0
            :: out1?m; to[farport]!thisport,m,1
            :: out2?m; to[farport]!thisport,m,2
            :: ...
            :: to[farport]!thisport,teardown,0; goto unlinking
            :: to[thisport]?eval(farport),teardown,0;
               to[farport]!thisport,downack,0;
               in1!downack; in2!downack; ...; goto end_idle
            od;
unlinking:  do
            :: to[thisport]?eval(farport),status,0
            :: to[thisport]?eval(farport),m,1; in1!m
            :: to[thisport]?eval(farport),m,2; in2!m
            :: ...
            :: to[thisport]?eval(farport),teardown,0;
               to[farport]!thisport,downack,0
            :: to[thisport]?eval(farport),downack,0;
               in1!downack; in2!downack; ...; goto end_idle
```

```
            od
}
```

## 9.3 Channel identifiers (Jun 01)

Within the scope of an internal call, any number of media channels can be opened and closed. If a channel identifier (tunnel value) is reused within a call, it could conceivably cause subtle race conditions in the implementation.

It would also be a problem if the two ports of an internal call both attempted to open a media channel at approximately the same time, using the same channel identifier. Although this problem can be resolved, it causes extra complexity.

Both problems are avoided completely by rules governing a box's choice of channel identifiers.

- No channel identifier can be reused within an internal call.
- An *open* signal sent by a caller port must use an odd channel identifier. An *open* signal sent by a callee port must use an even channel identifier.

## 9.4 The channel protocol (Nov 03)

The following program controls a single channel in a single call. It takes the initiative in opening the channel,

```
proctype active_channel(chan in,out)
{
end_nocall: in?upack;
            if
            :: out!open; goto opening
            :: in?downack; goto end_call
            fi;
opening:    if
            :: in?oack; goto oacked
            :: in?onack; goto end_chan
            :: out!close; goto closing
            :: in?downack; goto end_call
            fi;
oacked:     if
            :: out!ready; goto readying
            :: out!close; goto closing
            :: in?close; goto closed
            :: in?downack; goto end_call
            fi;
readying:   do
            :: in?status
            :: out!status
            :: out!close; goto closing
```

```
            :: in?close; goto closed
            :: in?downack; goto end_call
            od;
closing:    do
            :: in?oack
            :: in?onack
            :: in?status
            :: in?close; goto closing2
            :: in?closeack; goto end_chan
            :: in?downack; goto end_call
            od;
closing2:   if
            :: out!closeack; goto closing3
            :: in?closeack; goto end_chan
            :: in?downack; goto end_call
            fi;
closing3:   if
            :: in?closeack; goto end_chan
            :: in?downack; goto end_call
            fi;
closed:     if
            :: out!closeack; goto end_chan
            :: in?downack; goto end_call
            fi;
end_chan:   in?downack;
end_call:   skip
}
```

Note that call teardown automatically closes an open channel.

The fact that there are more *closing* states in the channel protocol than *unlinking* states in the call protocol does not mean that they are fundamentally different. It is simply because the "master" call protocol can have longer uninterruptable sequences of events than the "slave" channel protocol can without causing deadlock.

The following program also controls a single channel in a single call. It receives a channel *open* rather than sending one.

```
proctype passive_channel(chan in,out)
{
end_nocall: in?upack;
            if
            :: in?open; goto opened
            :: in?downack; goto end_call
            fi;
opened:     if
            :: out!oack; goto oacking
```

```
            :: out!onack; goto end_chan
            :: in?close; goto closed
            :: in?downack; goto end_call
            fi;
oacking:    if
            :: in?ready; goto readied
            :: in?close; goto closed
            :: out!close; goto closing
            :: in?downack; goto end_call
            fi;
readied:    do
            :: in?status
            :: out!status
            :: out!close; goto closing
            :: in?close; goto closed
            :: in?downack; goto end_call
            od;
closing:    do
            :: in?status
            :: in?close; goto closing2
            :: in?closeack; goto end_chan
            :: in?downack; goto end_call
            od;
closing2:   if
            :: out!closeack; goto closing3
            :: in?closeack; goto end_chan
            :: in?downack; goto end_call
            fi;
closing3:   if
            :: in?closeack; goto end_chan
            :: in?downack; goto end_call
            fi;
closed:     if
            :: out!closeack; goto end_chan
            :: in?downack; goto end_call
            fi;
end_chan:   if
            :: in?downack; goto end_call
            :: in?close; goto closed
            fi;
end_call:   skip
}
```

## 9.5 Synchronization of signaling and media (Aug 01)

The channel protocol provides a means by which boxes can control the media streams they need, and can learn the status of those media streams. The media streams themselves may be implemented quite separately.

Usually an endpoint of a media stream is an interface box. However, a feature box can also act, either temporarily or permanently, as an endpoint of a media stream. For example, a Call Waiting box must switch voice channels, and must therefore act as an endpoint for all its voice channels.

When a box receives an *open* signal and intends to act as a media endpoint for the requested channel, it must first establish the requested media channel, then respond to the *open* with *oack*. Sending *oack* means that the media channel exists and is ready to transmit in both directions, at least as far as this box is capable of knowing.

However, having sent an *oack*, the box cannot simply begin to transmit media. That media could be lost because it could outrun the *oack* in reaching the other media endpoint.

When a box receives an *oack* signal and intends to act as the media endpoint for that channel, it responds to the *oack* with a *ready* signal. It can then begin to transmit, because it knows the media channel exists end-to-end. When the opposite media endpoint, the one that sent the *oack*, receives *ready*, it also knows that the media channel exists end-to-end, and can also begin to transmit.

A box transparent to a media channel propagates an incoming *open* signal from one call as an outgoing *open* signal in a corresponding call, but does not respond to it with *oack* or *onack* until it has received one of those signals from the outgoing side.

If it receives an *onack* from the outgoing side, it merely propagates this signal on the incoming side. If it receives an *oack* from the outgoing side, on the other hand, it must both link the two channel terminations (see Section 11.2) and propagate this signal on the incoming side.

Thus an *open* signal originates at a true media endpoint, as does the answering *oack* or *onack* signal. We say that media channels are opened *end-to-end*, in contrast to the *piecewise* setup of a chain of related calls.

A media channel must be destroyed before a *closeack* signal, or a *downack* signal subsuming it, is sent. Channels are closed piecewise. This makes sense from a signaling perspective because most channels are closed implicitly by the tearing down of the call within which they exist. It also makes sense from a media stream perspective because an end-to-end media stream can be destroyed by destroying *any* segment of it. This is in contrast to establishment of an end-to-end media stream, which is not complete until *all* segments are established.

## 9.6 Channel-level status signals (Dec 02)

Once a channel has been opened, its signaling tunnel can be used to send status signals in either direction.

There is a long-established convention in telephony that a human user is not alerted by an incoming call until a media channel between his device and another media endpoint is fully ready. The protocol enforces this convention at the channel level, which has four built-in status signals.

- The signal *wait* indicates that an attempt is being made to enable human communication on the requested medium. Typically the signal is generated by an interface box. The attempt might involve initializing a device or device capability, alerting a person to come to a device, or asking a person's permission to communicate on that medium.
- The signal *accept* indicates that communication on the requested medium has been accepted. It is often generated by an interface box, because a person has indicated readiness to communicate. It can also be generated by a feature box, for example so that the box can use the medium as a user interface.
- The signal *reject* indicates that a communication on the requested medium has been rejected. It is often generated by an interface box.
- The signal *none* cancels the effect of any of the three previous signals on a user interface. It is only generated by feature boxes.

As with the built-in status signals of the call protocol, feature boxes can scramble the expected appearance and order of these signals. As a result, there are no built-in constraints on how many of these signals there are in a usage, or where they travel within the usage.

Other special-purpose status signals can be added at the channel level as needed. A status signal can carry any set of fields whatsoever.

## 9.7 Properties of the channel protocol (Dec 02)

The channel protocol does not introduce deadlock, nor does it interfere with reliable signaling.

The following table summarizes fields of the six functional signal types of the channel protocol.

| | `open` | `oack` | `onack` | `ready` | `close` | `closeack` |
|---|---|---|---|---|---|---|
| `internal-Origin` | *required* | *required* | *required* | *required* | *required* | *required* |
| `tunnel` | *natural* | *natural* | *natural* | *natural* | *natural* | *natural* |
| `medium` | *required* | *forbidden* | *forbidden* | *forbidden* | *forbidden* | *forbidden* |
| *additional status fields* | *allowed* | *allowed* | *allowed* | *allowed* | *allowed* | *forbidden* |

# 10 Behavior of line and trunk interface boxes (Jan 03)

## 10.1 Ports on interface boxes (Dec 02)

In all of the Promela programs in this manual, the state of a port is only defined (named) when it is waiting for input. A line interface box can have at most one internal call that is in the *linked* state. If the box has such a call and receives a *setup* signal, it must respond with the sequence *upack; unavail; teardown*. The port on which the new call is received passes directly from the *end_idle* state to the *unlinking* state.

A trunk interface box can have many DFC ports in the *linked* state. ¿From the perspective of a DFC network, these ports are interchangeable and independent.

If a DFC router routes a call to an interface box that is busy, when there is an equally suitable interface box that is not busy, there is no recourse. This problem is especially bad with trunk interfaces because of trunk glare[1]. Glare can have the effect of making a trunk busy *after* it was put to use in the belief that it was idle!

This problem can be eliminated by aggregating interoperation facilities into large trunk interface boxes with many DFC ports. DFC routes to the trunk interface box, which accepts the call on any available port. A large trunk interface box can resolve glare internally and autonomously by changing external channels (trunks) when necessary.

## 10.2 Interface protocols (Jan 03)

An interface box that places an internal call must use the method *newSetupSignal*. If the interface box supplies a dialed string, it need not supply a target address, as the target address will be extracted from the dialed string.

Just as a line interface box interfaces to a device, we can think of a trunk interface box as interfacing to a device, even though that device must be reached through another network.

If an interface box receives a *setup* signal at a time when it has no DFC port in the *linked* state, and it finds that the device it interfaces to is unavailable, it follows the *upack* signal with the sequence *unavail; teardown*.

If a line interface box receives an *open* signal for a medium that its device cannot handle, then it responds with an *onack*. If a line interface box receives an *open* signal for a medium that its device can handle, then it responds with an *oack*.

After receiving a *ready*, if a line interface knows immediately whether communication on that medium is acceptable, for example because the device is

[1] Trunk glare occurs when the two switches at the two ends of a trunk both seize the trunk for an outgoing call, thinking that it is idle. Both switches are aware of the condition, because both receive setup messages when they expected setup acknowledgments. Typically a static priority scheme is used to choose a winner, and the loser retracts its setup.

fully automatic, then the line interface follows *ready* immediately with *accept* or *reject*. If the line interface does not know immediately, which is the most common situation, then the line interface follows *ready* immediately with *wait*. It proceeds with the alerting or whatever else is necessary, and follows with *accept* or *reject* whenever a human responds.

An interface box should send an *avail* signal only when the incoming call is deemed truly successful. In the case of a POTS-like device and interface, it would send *accept; avail* when a user answers the telephone.

## 10.3 User-interface signaling (Dec 02)

An interface box translates between the DFC protocol and its line or trunk protocol to support the user interface of a device. The crudest, and therefore most challenging, user interfaces are those that use the voice medium for signaling.

An interface box with voice signaling reacts to receipt of an *unknown, unavail, wait,* or *reject* signal by generating a tone (or doing whatever is necessary to cause the tone to be generated). It reacts to receipt of an *avail* or *none* signal at the call level by silencing any *unknown* or *unavail* tone. It reacts to receipt of an *accept* or *none* signal on the voice channel by silencing any *wait* or *reject* tone.

At the channel level, any time a line interface box is in the *readied* state as defined in Section 9.4, media can flow in both directions, and should be allowed to reach the endpoint device. This is necessary because many feature boxes use media channels for user-interface signaling. Feature behavior and feature interactions must coordinate among tone generation in an interface box and media signaling in various feature boxes.

Often it is desirable to prevent end-to-end media flow, for example until the service provider is sure of being paid for it. Such blockage must be performed by a feature box rather than an interface box, because of the need for media-based user-interface signaling.

Users control features by causing their devices to generate events of some kind. These events are translated by interface boxes into DFC status signals, which may be sent at the call level or the channel level.

Many status signals for feature control are specific to particular devices and/or features. These are valuable because they make it possible to exploit sophisticated devices, and to provide rich user interfaces for complex feature sets.

On the other hand, it is also important to have a set of universal signals that can be generated by any device. Feature boxes controlled by these signals can be used in any context. This purpose is served by the built-in status signal type *dtmf*, which has a *symbol* field indicating one of the twelve DTMF symbols. Status signals of type *dtmf* are always sent in the tunnel of a voice channel.

If a line interface represents a device, such as an analog black telephone, that can only generate these signals in-band, then the line interface must detect them and duplicate the information in the form of out-of-band *dtmf* signals.

# 11 Media processing (Jan 03)

## 11.1 Media (Mar 01)

Each DFC network has a fixed set of *media.* A *medium* is a distinct and global form of communication, as seen from the user's perspective. Any endpoint for a medium can communicate with any other endpoint for that medium.

Although the most commonly discussed media for telecommunications are voice, video, text, images, and high-fidelity sound, a DFC network is not limited to these, nor need it offer all of them.

## 11.2 Links (Mar 01)

There are two kinds of media processing.

Transmission, muting, switching, replicating, and summing [2] of media streams are all easy from the description perspective because they can be specified statically and uniformly, simply by the state of an intra-box connection relation.

Each media channel in an internal call has an identifier and two *channel terminations,* one at each port of the call. A channel termination is identified by a (port,channel-identifier) pair.

The media-processing state of a box is described as a set of *links.* Each link is a unidirectional media connection between two channel terminations; the channel terminations must have distinct ports on the same box. The links whose sink is a channel termination *(p,c)* specify those channel terminations at other ports of the same box whose media streams arriving as input to the box are to be summed and sent as output from the box at the channel termination *(p,c).*

For example, consider a box with ports $p_1$, $p_2$, and $p_3$ at which voice channels $c_1$, $c_2$, and $c_3$ respectively are open. The set of links

$$\{\ ((p_2,c_2),\ (p_1,c_1)),\ ((p_3,c_3),\ (p_1,c_1)),\ ((p_1,c_1),\ (p_2,c_2))\ \}$$

specifies that the output from *($p_1$,$c_1$)* is the sum of the inputs at *($p_2$,$c_2$)* and *($p_3$,$c_3$)*, the output at *($p_2$,$c_2$)* is just the input at *($p_1$,$c_1$)*, and there is no output at *($p_3$,$c_3$)*. Thus *($p_1$,$c_1$)* and *($p_2$,$c_2$)* have two-way voice communication; *($p_1$,$c_1$)* can also hear *($p_3$,$c_3$)*, while *($p_3$,$c_3$)* can hear nothing.

## 11.3 Resources and resource interface boxes (Jan 03)

In contrast to the previously mentioned forms of media processing, recording, playing, mixing, monitoring (pattern recognition), and media conversion are much harder. They require both control ("start recording") and status ("announcement completed") events. They require many control arguments such as announcement identifiers, recognition grammars, and volume levels. Finally, there are many different devices for performing these functions, each with its own programming interface. A box programmer who needs control-intensive

[2] Summing is a simple form of mixing, with no selective volume control, and possibly a limit on the number of channels involved.

media processing must know what kind of device is being used to implement it, and must program his feature box using the programming interface of that device.

A device that performs control-intensive media processing is visible in the DFC architecture as a *resource*, and is joined to a DFC network by means of a *resource interface box*.

*RAddress* is the set of addresses used by box programmers to reach resources. A member of *RAddress* names a *type of resource* rather than an individual resource. The type corresponds, in turn, to a fixed programming interface defined in terms of status signals.

The address mapping *RMap* maps a member of *RAddress* to some *RIBox* of the appropriate type. The mapping from resource addresses to resource interface boxes is many-to-many. A resource address can map to many interface boxes so that routing can choose resources in optimal locations. A resource interface box that implements several programming interfaces can have several resource addresses, one for each programming interface.

Like a trunk interface box, a resource interface box can have one or more (depending on the size and capabilities of the resource itself) DFC ports in the *linked* state.

Resource interface boxes do not place internal calls.

Resource interface boxes use the built-in call-level status signals in the usual way. Whether they use the built-in channel-level status signals depends on how their programming interface is defined.

# Glossary

This glossary defines briefly the main terms used in *The DFC Manual*, and shows for each one the chief sections of the manual where it is explained or discussed.

**AAlphabet** The alphabet of symbols used for addresses. See Section 2.4.

**accept signal** A status signal indicating that communication on a media channel has been accepted. See Section 9.6, Section 10.2, Section 10.3.

**address** The set of syntactically correct identifiers in a DFC network, or a member of that set. See Section 2.3, Section 2.5, Section 3.5, Section 5, Section 6.

**Alloy** A formal language [4] used in this manual for describing data structures and data operations. See Section 1.3.

**AString** A string containing symbols from the address alphabet AAlphabet. See Section 2.4.

**avail signal** A status signal indicating that the target of a call is available and the call is deemed successful. See Section 8.6, Section 10.2, Section 10.3.

**bound feature box (BFBox)** A feature box that is persistent, unique, and permanently associated with a subscribing address. See Section 2.1, Section 2.6, Section 6.5.

**box** Boxes are the primary modules and components in DFC. A box is a concurrent process, and performs either feature functions or interface functions. Boxes are connected by internal calls to form usages.

**box address** The subscribing address on whose behalf a feature box is created and/or assembled into a usage. See Section 2.6, Section 6.5.

**box type** The type of a box corresponds to a box program; once created, a box is an instance of the program corresponding to its type. See Section 2.1, Section 2.2, Section 3, Section 6.5.

**call protocol** The protocol governing the setup, use and teardown of internal calls between boxes. See Section 8.

**callee port protocol** A protocol governing the behavior of a port in an internal call, applicable when the port is placing the call. See Section 8.4, Section 9.2.

**caller port protocol** A protocol governing the behavior of a port in an internal call, applicable when the port is receiving the call. See Section 8.3, Section 9.2.

**chain** Short for "a set of calls set up or torn down in a chain reaction." See Section 8.5.

**channel** Short for "media channel."

**channel identifier** A natural number identifying a media channel and its corresponding tunnel within an internal call. See Section 9.1, Section 9.3.

**channel termination** The termination of a media channel at a port. See Section 11.2.

**close signal** A signal of the media channel protocol that closes a channel. See Section 9.

**closeack signal** A signal of the media protocol that acknowledges closing of a channel. See Section 9.

**continuedSetupSignal** A method used by a box to create a setup signal for an outgoing internal call based on a previously received setup signal of an incoming internal call. See Section 5.5.

**customer** The owner of one or more addresses. See Section 7.

**customer-partitioned data** Operational data that is partitioned by customer. See Section 7.

**device** A physical device such as a telephone, computer, or hardware signal processor that is attached to a DFC network by a line or resource interface box.

**dialed (dld)** A field of a setup signal containing the string "dialed" by the calling user at a device, if any. See Section 5.2, Section 5.4, Section 6.2.

**downack signal** A signal of the call protocol acknowledging a teardown signal. See Section 8.

**dtmf signal** A channel-level status signal that transmits a DTMF symbol. See Section 10.3.

**embeddedAddress** A function used by the router to extract an address from a dialed string. See Section 6.2.

**end-to-end** This adjective refers to a connection path between two interface boxes, or to aspects of the DFC protocol that emphasize the role of interface boxes and de-emphasize the role of feature boxes. See Section 5.6, Section 6.6, Section 9.5.

**error box (EBox)** A box that handles an addressing error. See Section 2.1, Section 6.5.

**feature** An incremental unit of functionality in a DFC network. See Section 3, Section 7.

**feature box (FBox)** A box that provides the full or partial functionality of a feature. See Section 2.1, Section 3.

**feature box type (FBoxType)** These partition the set of all feature boxes. See Section 2.2.

**feature-partitioned data** Operational data that is partitioned by feature. See Section 7.

**free feature box (FFBox)** A feature box of a fungible type, that is, of a type of which any instance may be substituted for any other. See Section 2.1, Section 2.6, Section 6.5.

**input-enabled** A box is input-enabled if it is always (except for short intervals of time) able to read the input queue at each of its ports and to process any input signal permitted by the relevant protocol and port state. See Section 8.7.

**interface box (IBox)** A box that provides an interface to a line, trunk, or resource. See Section 2.1, Section 2.5, Section 10, Section 11.3.

**internal call** A featureless, point-to-point call with one two-way signaling channel and any number of media channels. Internal calls connect boxes to form usages, so they are one of the two major structures in DFC.

**LAddress** The set of addresses that uniquely identify line interface boxes. See Section 2.3, Section 2.5, Section 6.5.

**line interface box (LIBox)** An interface box connecting a line and telecommunication device to a DFC network. See Section 2.1, Section 2.5, Section 6.5, Section 10.

**link** A one-way connection between two media channel terminations in one box. See Section 9.5, Section 11.2.

**LMap** The mapping from addresses to line interface boxes. See Section 2.5, Section 6.5.

**MAddress** A mobile address that has no permanent association with any interface box, but can subscribe to features. See Section 2.3, Section 2.5, Section 6.5.

**medium** A class of transmissible data, for example, voice, video, or text. See Section 9, Section 11.

**media channel** A channel by which media may be transmitted in an internal call. See Section 9, Section 11.

**media channel protocol** The protocol governing the opening and closing of media channels. See Section 9.

**mixing** Combining an arbitrary number of voice media streams in specified relative volumes. Functionally more complex than summing. See Section 11.

**mtype** In a Promela specification, the set of all signal types.

**muting** Interrupting the transmission of a media stream, possibly in only one direction, without closing the channel. See Section 11.

**newSetupSignal** A method used by a box to create a new setup signal. See Section 5.4.

**none signal** This signal is unique in being used at both the call and channel levels. It is a built-in status signal that cancels the effect of any previous built-in status signal on a user interface. See Section 8.6, Section 9.6.

**noAddr** A distinguished value of any address field or variable indicating the absence of an address. See Section 2.3.

**noString** A distinguished value of any string field or variable indicating the absence of a string. See Section 2.4.

**oack signal** A signal of the media channel protocol indicating that a request for a media channel has been received by an endpoint, and that the endpoint is capable of that medium. See Section 9, Section 10.2,

**onack signal** A signal of the media channel protocol indicating that a request for a media channel has been received by an endpoint, and that the endpoint is not capable of that medium. See Section 9.

**open signal** A signal of the media channel protocol requesting a new media channel. See Section 9.

**operational data** Persistent data that can be read and written by feature boxes. See Section 7.

**outer** A field of a setup signal indicating a previous source address in a source region, used for authentication. See Section 5, Section 6.4.

**piecewise setup** A discipline of setting up an end-to-end connection by completing the setup of each segment without waiting for completion of the setup of either neighboring segment. See Section 8.5.

**placing** A field of a setup signal specifying the type of the box attempting to place a call with this signal. See Section 5.2, Section 5.6.

**port** A locus within a box for sending and receiving signals, with its own independent signal queue. See Section 8.

**precedence** A constraint on the order in which boxes may be included in a zone of a usage. See Section 3.4, Section 3.5.

**Promela** A formal language [3] used in this manual for describing protocols. See Section 1.3, Section 8.1.

**providesFeature** The mapping from a feature box type to the feature it implements or partially implements. See Section 3.2, Section 7.

**RAddress** The set of identifiers of resource types. See Section 2.3, Section 2.5, Section 6.5.

**ready signal** A signal of the media protocol indicating that an end-to-end media connection has been established. See Section 9, Section 10.2, Section 10.3.

**region** Feature boxes are routed to in either the source region or target region. The source region contains feature boxes subscribed to by a source address in its role as caller; the target region contains feature boxes subscribed to by a target address in its role as callee. Also, the *regn* field of a setup signal indicates which region routing is currently concerned with. See Section 3.1, Section 3.3, Section 3.4, Section 3.5, Section 5.2, Section 6.

**reject signal** A status signal indicating that communication on a media channel has been rejected. See Section 9.6, Section 10.2, Section 10.3.

**resource** A hardware or software device capable of specialized media signal processing, such as recording, playing, and voice recognition. See Section 11.3.

**resource interface box (RIBox)** An interface box connecting a media-processing resource to a DFC network. See Section 2.1, Section 2.5, Section 6.5, Section 11.3.

**reversedSetupSignal** A method used by a box to create a setup signal for a new outgoing internal call based on one previously used to place an outgoing internal call. The created signal continues that call, but in the opposite direction. See Section 5.6.

**reversible box type** A box type that is always subscribed to in both source and target regions. Only reversible boxes can use the *reversedSetupSignal* method. See Section 3.2, Section 3.3, Section 3.4, Section 3.5, Section 5.6, Section 6.3.

**RMap** The mapping from addresses to resource interface boxes. See Section 2.5, Section 6.5, Section 11.3.

**route** The field of a setup signal containing the expected route—a sequence of feature boxes—to be taken in constructing the usage. It can also contain a *ZoneTag*, which is a placeholder used before zone expansion. See Section 5, Section 6.

**router protocol** The protocol governing the behavior of a DFC router in exchanging signals with boxes. See Section 8.2.

**router** A component of a DFC network that sets up internal calls between boxes by modifying and transmitting setup signals. A router is an implementation of the DFC routing algorithm. See Section 6, Section 8.2.

**SAlphabet** The alphabet of symbols used for signaling. See Section 2.4.

**setup signal** A signal of the call protocol requesting the setup of a call. It is sent by a port of one box, via the router, to another box. A usage is created by chains of related setup signals. See Section 5, Section 6, Section 8.

**signal** A message sent or received as part of the DFC protocol. See Section 4, Section 8, Section 9.

**signal queue** A queue of signals waiting to be processed by a box, by a box port, or by a DFC router. See Section 8.

**source (src)** A field of a setup signal containing an address representing the caller. See Section 5, Section 6.

**SString** A sequence of symbols from the signaling alphabet SAlphabet. See Section 2.4.

**status signal** A signal of the call or media channel protocol other than the basic signals for setting up, tearing down, and confirming the call or channel connection.

**subscriber** A customer of a DFC network who subscribes to features at an LAddress or MAddress. See Section 3.5, Section 7.

**subscription** An association between an address and a feature box type. The source subscriptions of an address determine which feature boxes are in its source zones, and its target subscriptions determine which feature boxes are in its target zones. See Section 3.3, Section 3.5, Section 6.3.

**summing** Combining a small number of voice media streams in equal volumes. Functionally simpler than mixing. See Section 11.

**TAddress** The set of addresses that identify lines on another network, accessible through a trunk interface. See Section 2.3, Section 2.5, Section 6.5.

**target (trg)** A field of a setup signal containing an address representing the callee. See Section 5, Section 6.

**teardown signal** A signal of the call protocol that tears down the call. See Section 8.

**trunk interface box (TIBox)** An interface box acting as a proxy for multiple line interfaces on an adjacent network. See Section 2.1, Section 2.5, Section 6.5, Section 10.

**TMap** The mapping from addresses to trunk interface boxes. See Section 2.5, Section 6.5, Section 11.3.

**trunk glare** A condition in which the two switches at the ends of a trunk both sieze the trunk for an outgoing call. See Section 10.

**tunnel** Each signal has a *tunnel* field. If the field is zero, the signal refers to the call as a whole. Otherwise, the tunnel value refers to a media channel of the call, and the signal is part of the media channel protocol for that channel. See Section 8.1, Section 9.1, Section 9.3.

**unavail signal** A status signal indicating that the target of a call is not available. See Section 8.6, Section 10.2, Section 10.3.

**unknown signal** A status signal indicating that the target address of a call does not map to an interface box. See Section 8.6.

**upack signal** A signal sent by a box indicating that a setup signal has been received, a port has been allocated, and the requested internal call has been established. See Section 8.

**usage** A set of boxes and internal calls forming a maximal connected graph. Usages can be identified in a snapshot of a running DFC network, but this identity does not persist over time, because usages can split and merge. All external requests for telecommunication service are satisfied by usages.

**wait signal** A status signal indicating that an endpoint is attempting to enable communication on a medium, usually by alerting. See Section 9.6, Section 10.2, Section 10.3.

**zone** A subsequence of feature boxes in a usage, all of which are subscribed to by the same address. In simple usages, a zone is either a subsequence of a source region (a source zone) or a subsequence of a target region (a target zone). Also, for each address there are *srcZone* and *trgZone* records presenting the box types subscribed to by the address in the two regions, in an order compatible with precedence constraints. Section 3.1, Section 3.3, Section 3.4, Section 3.5, Section 6.

**ZoneTag** An enumerated set containing the values *whole* and *suffix*. When a zone tag is the value of the *route* field of a setup signal, it must be replaced by all or part of a *srcZone* or *trgZone*. See Section 5.2, Section 5.4, Section 5.5, Section 5.6, Section 6.3.

# References


[1] Gregory W. Bond, Eric Cheung, K. Hal Purdy, Pamela Zave, and J. Christopher Ramming. An open architecture for next-generation telecommunication service. *ACM Transactions on Internet Technology* IV(1), February 2004, to appear. 1

[2] The DFC Web site:
`http://www.research.att.com/projects/dfc` 1, 41

[3] Gerard J. Holzmann. Design and validation of protocols: A tutorial. *Computer Networks and ISDN Systems* XXV:981-1017, 1993. 2, 22, 38

[4] 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. 2, 34

[5] 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. 1, 18

[6] Pamela Zave. Ideal connection paths in DFC. AT&T Research Technical Report, 2003, on [2]. 1, 11, 14, 16

[7] Pamela Zave and Michael Jackson. DFC modifications I (Version 2): Routing extensions. AT&T Technical Memorandum HA1640000-000128-4TM, January 2000. 1

[8] Pamela Zave and Michael Jackson. DFC modifications II: Protocol extensions. AT&T Technical Memorandum HA1640000-991119-16TM, November 1999. 1