Modularity in Distributed Feature Composition

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

26 January 2009

Abstract

Distributed Feature Composition (DFC) is a modular architecture for building telecommunication services. It has been implemented and used to build two industrial-scale Voice-over-IP services, as well as many smaller prototype and demonstration services. With all this experience it is possible to assess how and how well DFC modularity works.

1 Introduction

Distributed Feature Composition (DFC) is a modular architecture for building telecommunication services. Michael Jackson and I got the idea for DFC in an "aha!" moment in December 1996. We spent 1997 and 1998 working out the details, and published the first paper on DFC in 1998 [8]. In 1999 a team[1] began working with us to implement DFC. Since then this team, with new additions[2] and occasionally other AT&T colleagues, has worked continuously on DFC-based technology and applications.

Historically, DFC bridges the Public Switched Telephone Network (PSTN) and the Internet. When we invented DFC, Michael and I had been studying the software problems of the PSTN for some time, and we had no other context in mind. Nevertheless, by the time DFC was ready to implement, Voice-over-Internet-Protocol (VoIP) was the new technology that researchers wanted to work with. DFC proved to be equally applicable to VoIP, and all of the implementations of DFC have been Internet-based.

The focus of this paper is modularity in DFC, which is an adaptation of the pipes-and-filters architectural style to telecommunication applications. This kind of modularity is much less familiar than other kinds of modularity such as object-oriented programming, so the primary purpose of this paper is to explain where, why, and how it works. After 12 years, there is an abundance of experience to draw upon.

DFC was designed to support modular development of *features*; Section 2 explains the significance of this motivation and its history in telecommunications. Section 3 is an overview of pipes-and-filters modularity as realized in DFC.

The benefit of feature modularity comes with the burden of managing interactions among features. This burden is also an opportunity, because each principle for identifying or managing interactions captures important domain knowledge about the organization of features. Section 4 introduces the major categories of feature interaction and how they are managed.

Subsequent sections are based on our experience with implementation of DFC and deployment of services built on our platforms. They evaluate its form of modularity and speculate on its future.

Most of the service examples in this paper come from old-fashioned telephony, because these are simple and easy to discuss. DFC is equally useful, however, for the richer services being built or envisioned today. Contemporary telecommunication services differ from telephony (including mobile telephony) in three ways:

- Rather than being limited to voice (low-fidelity audio), they also support media such as music (high-fidelity audio) and video. Text, images, and other data can also be treated as media. For example, email fits easily into the DFC architecture, as do home networks.
- Telecommunication services used to be limited to and by "black" phones, with their very restricted user interface. Now personal computers with Web browsers are common, as are handheld devices with touch-sensitive screens. These devices make it possible for users to interact conveniently with much more elaborate and data-oriented services.
- Not all telecommunications systems are stand-alone applications. They can also be embedded in applications for multiplayer games, distance learning, collaborative television, networked music performance, and other forms of computer-supported cooperative work

[1] Gregory W. Bond, Eric Cheung, K. Hal Purdy, and J. Christopher Ramming.

[2] Thomas M. Smith and Venkita Subramonian.

and play.

2 Feature-oriented description

DFC was designed to provide feature modularity and to manage the feature-interaction problem, so an explanation of DFC must begin with features.

The behavior of telecommunication software is almost always described in terms of *features*. A *feature* is an optional or incremental unit of functionality. A *feature-oriented description* consists of a base description with additional, optional feature modules.

For example, a traditional informal explanation of telephone service begins with Plain Old Telephone Service (POTS), which has as its primary states idle, dialing, busy, ringing, and talking. This is a base description. The explanation then covers a set of separate features such as Speed Dialing, Call Waiting, and Call Forwarding. Each feature is presented as an addition or exception to POTS, without mentioning or relying on other features.

The modification of POTS by features began in the 1960s, when telephone switches became software-controlled. By the mid-1980s large telephone switches had thousands of features, each described in an informal requirements document. Because there was no feature-oriented programming technique, all of the features had to be implemented in the same piece of software. The size and complexity of this software was making it extremely difficult to add new features and to maintain software reliability.

A *feature interaction* is some way in which a feature or features modify or influence another feature in describing or generating the system's overall behavior. Feature interactions are inevitable in any nontrivial feature-oriented description. The modular nature of the description tends to make interactions (at best) implicit or (at worst) obscure.

For the large telephone switches of the 1980s and later, feature interactions were perceived as a huge problem. It took tremendous skill to predict the interactions implied by multitudes of informal feature descriptions, and arduous labor to specify the desired behavior in all cases. In the implementation, which was not decomposable along feature boundaries, feature interactions were a major source of complexity and software defects.

The primary goal for the design of DFC was to find a feature-oriented way to program telecommunication systems, so that features could be implemented independently and yet composed to produce overall system behavior. We also needed a way to predict potential feature interactions, enable the desired ones, and prevent the undesired ones.

3 Pipes-and-filters modularity

3.1 The signaling protocol

Basic telecommunication service is built into the DFC architecture. Each user device is represented by a persistent software module called an *interface box*, which has a network address and the ability to translate signals between the DFC protocol and the native protocol of the device.

When one interface box calls another, the DFC protocol forms a connection between them. This connection supports a single two-way, FIFO signaling channel and any number of media channels.

When there are applicable features, telecommunication service is provided by a graph called a *usage*, as shown in Figure 1. The nodes of a usage include *feature boxes* as well as interface boxes. Each feature box is a concurrent software process that implements a separate feature.

The edges of a usage are *internal calls*, each of which is a connection made with the DFC protocol. This means that each feature box is a signaling and media endpoint for the internal calls that it participates in. The term *internal call* is used to distinguish an edge in the graph from the informal, end-to-end meaning of "call" in telecommunications.

In the DFC protocol, an internal call begins when one box sends a *setup* signal to another box. The box acknowledges it by sending an *upack* signal back, thus establishing the connection and its signaling channel. Subsequently the signaling channel can be used to open and close media channels. It can also be used for commands and status signals involved in feature control.

Each setup signal carries a source address and a target address. In simple cases these are the addresses of the interface boxes on the two ends of the usage. However, as we shall see, many features manipulate these addresses.

The two most important status signals are *avail* and *unavail*. The *avail* signal travels from the callee or receiving end of the call to the caller or placing end. It indicates that the entity identified by the target address is available for communication. Its dual is *unavail*, which indicates that the targeted entity is not available. Either box participating in a call can tear it down at any time by sending a *teardown* signal, acknowledged by a *downack*.

A well-designed DFC feature box has the properties of *transparency*, *autonomy*, and *context-independence*. *Transparency* means that when its feature is not active, it is unobservable by other boxes in the graph. It is acting as an identity element, merely relaying signals from one port to another. *Autonomy* means that when it needs to perform some function, it does so without help from other boxes. A DFC feature box can act autonomously because it sits in a signaling path between user devices, where it can observe all the signals that travel between them. Be-

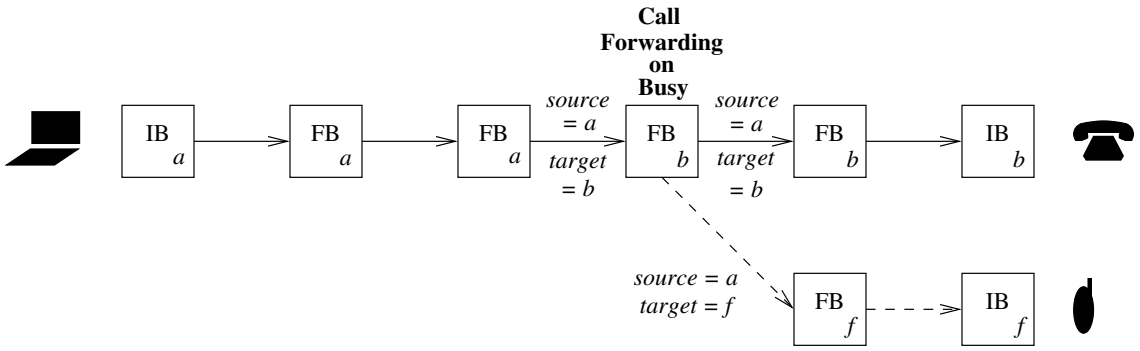

Figure 1: Example of a usage, with interface boxes (IB) and feature boxes (FB). Internal calls are represented by arrows to show the direction in which they are set up.

cause it is a protocol endpoint, it can absorb or generate any signals that it needs to. *Context-independence* means that it does not rely on the presence of other features, or contain any knowledge of them. A DFC feature box does not know what is at the other end of the internal calls it is participating in.

These properties are illustrated in a simple way by Call Forwarding on Busy (CFB), which is the type of one of the feature boxes shown in Figure 1. Initially the box behaves transparently, receiving an incoming internal call, and continuing the chain by placing an outgoing internal call with the same setup information. If it receives *avail* from downstream (its outgoing call), then its function is not necessary, and it stays transparent during its entire lifetime. The usage containing this box (the graph connected by solid arrows in Figure 1) persists while the parties are communicating. When they are no longer communicating and the usage is no longer needed, a *teardown* from either end propagates through the usage, destroying internal calls and terminating box programs as it travels.

If the device interface box with address *b* receives a setup signal when the device is already busy, it will generate the status signal *unavail*. If a CFB box receives *unavail* instead of *avail* from downstream, it takes autonomous action. It tears down its old outgoing call, so that the subgraph between CFB and IB(*b*) disappears. It places a new outgoing call with a setup signal containing the forwarding address *f* as its target. This creates the dashed subgraph extending to IB(*f*).

The CFB box is context-independent because the unavail signal that triggers it might have been generated by the user device, or by any feature box between CFB and the device. This point will be illustrated further in Section 4.1.

3.2 The routing algorithm

In Shaw and Garlan's characterization of a pipes-and-filters architecture [11], the graph of pipes (internal calls) and filters (boxes) is pre-configured and static. DFC is more complex because each usage is assembled dynamically and evolves over time.

The mechanism for assembling usages is the *DFC routing algorithm*, executed by a *DFC router*. A DFC router has a different purpose from IP routers. The purpose of an IP router is to find the destination of a message, while the purpose of a DFC router is to insert feature boxes in the path of a setup signal (message).

Each time a box sends a setup signal, that signal goes to a DFC router that chooses a box to receive it, and forwards the *setup* to the receiving box. Then the receiving box sends an upack signal directly to the sending box, and a direct connection is formed between them.

Every continuous routing chain from one interface box to another contains a *source region* and a *target region*. The source region comes first; it contains feature boxes working on behalf of the source address in its role as caller. The target region contains feature boxes working on behalf of the target address in its role as callee. Each address *subscribes* to some (possibly empty) set of feature box types in each region. In the "solid line" subgraph of Figure 1, there are two feature boxes in the source region subscribed to by address *a*, and two feature boxes in the target region (including CFB) subscribed to by address *b*.

If the CFB box is triggered to take action, its second outgoing call is routed to a box on behalf of the *forwarding address f*, which is the target address in the new setup signal. No additional boxes are routed to on behalf of the original target address *b*. The same thing can happen in the source region, if a box changes the source address when placing a call that continues the chain. Because of this mechanism of *address translation*, a routing chain can have multiple *source zones* in its source region, and mul-

tiple *target zones* in its target region. Each zone contains the feature boxes added to the chain on behalf of a particular address.

Within a zone, the routing algorithm orders the feature boxes by *precedence*. The source and target *precedence relations* are partial orders on feature box types.

Feature box types fall into two categories: *free* and *bound*. When a DFC router is working on a setup signal and selects a free box type as its destination, the router creates a new feature box (program object) as an instance of its type. Thus each free feature box is a transient, anonymous instantiation of its type.

Bound feature boxes are completely different. For each address subscribing to a bound feature box type (in either region), there is a single, persistent instance of that box type. When a router is working on a setup signal and selects a bound box type as its destination, the *setup* goes to the bound box identified with the address on whose behalf it is required. The use and significance of bound boxes are illustrated in Section 4.2.

A simple routing chain from interface box to interface box begins when the calling interface box creates a setup signal with the *new* method and sends it to a DFC router. To continue the chain, a feature box takes a setup signal it has received and applies the *continue* method to it. The *continue* method returns a suitable setup signal, which the box then sends to a DFC router. The *continue* method gives the box the option to change (translate) the source or target address of the *setup*. For example, a CFB box may invoke *continue* twice. The first time there is no address translation, so both addresses of its first outgoing call are the same as the addresses of its incoming call. The second time it exercises its option to change the target address to *f*.

Like signaling in DFC, routing in DFC supports transparency, autonomy, and context-independence. By using the continue method and no address translation, a feature box can continue a routing chain transparently. A feature box has some autonomy because it can affect routing by address translation or its choice of routing method. (Further uses of the *new* method are discussed in Section 4.4, and there is a third method *reverse* not covered here.) A feature box has context-independence because it never uses or sees the names of other feature box types.

Usage-dependent routing history is carried in setup signals and manipulated by routing methods and DFC routers only. It can be encrypted to enforce the context-independence of feature boxes. DFC routers need subscription and precedence data, but are stateless with respect to individual usages.

3.3 Other

Media and media control are discussed briefly in Section 4.4. The only other aspect of the DFC architecture is *operational data*, which is persistent data used by features. For example, the CFB box gets its forwarding address by retrieving it from operational data. Boxes can write operational data as well as read it. Operational data is usually partitioned by address and feature, so that a box can only access data belonging to its subscriber and feature.

As a result of transparency, autonomy, and context-independence, DFC features can be programmed independently. A particular feature can be present or absent in a usage without requiring changes in other features. Similarly, features can be added to or removed from the system without changing other features. These characteristics are the essence of modularity in DFC.

4 Management of feature interactions

DFC features are supposed to interact through the specified mechanisms of the architecture, and in no other way. By constraining how features can interact, the architecture makes it possible to identify and manage feature interactions in an organized fashion.

Once a class of feature interaction is identified, it is necessary to decide which members of the class are desirable or undesirable. Domain knowledge and experience are the best guides during this task.

Once the potential interactions in a feature set are predicted and evaluated, it is necessary to make adjustments to enable the good ones and prevent the bad ones. The preferred mechanism for managing feature interactions is to make adjustments in the precedence relations.

To illustrate this process, the following subsections introduce the five major classes of feature interaction in DFC. Each subsection attempts to provide a little insight into the nature of the interactions. The subsections also provide some additional explanation of DFC.

4.1 Activation interactions

It is possible for one feature to activate a function of another feature, or to ensure that it will not be activated. Four features illustrate some of that ways that this can happen: Call Blocking, Record Voice Mail, Quiet Time, and Parallel Ringing. All of these features are subscribed to in the target region.

First, I will describe briefly what these features do and how they interact, assuming that they are ordered by precedence as listed above and shown in Figure 2. Then

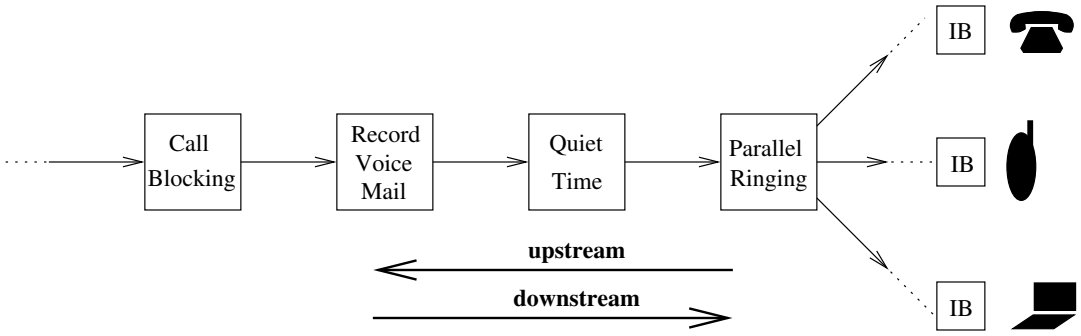

Figure 2: Four features in the target zone of a subscriber.

I will show how this precedence relation could be derived from the features themselves.

An instance of Call Blocking (CB) receives a setup signal targeted to its subscriber, and checks whether the source address is on the subscriber's blocking list. If so, it sends *unavail* upstream, because this subscriber is not available to this caller. CB does not continue the routing chain, but rather tears down its incoming call and terminates.[3]

If CB does not block its incoming call, then it continues the routing chain transparently by placing an outgoing call. Its behavior is transparent from that point on.

Next in the chain is Record Voice Mail (RVM). RVM is initially transparent, merely placing an outgoing call.

Next in the chain is Quiet Time (QT). If QT is currently disabled by the subscriber, it is permanently transparent and merely places an outgoing call to continue the chain. If it is currently enabled, on receiving a setup signal, it employs a media resource (see Section 4.4) to initiate an interactive voice-response (IVR) dialog with the caller. The media resource (IVR server) announces that the subscriber wishes to be undisturbed, and prompts to ask the caller if the call is urgent. If the call is not urgent, then QT sends *unavail* upstream and terminates, because the subscriber is not available for casual calls. If the call is urgent, then QT places an outgoing call and is transparent from that point on.

If QT sends *unavail* upstream, this signal reaches RVM and triggers it. RVM employs an IVR server to prompt for and record a voice message from the caller to the subscriber. Before doing this, RVM sends an avail signal upstream. Thus RVM has the effect of turning failure (*unavail*) to success (*avail*). From a philosophical viewpoint, this means that recording voice mail is considered to be (almost) as good as talking to a person. From a practical viewpoint, sending *avail* prevents features upstream from behaving as if the call is still ringing and unanswered.

Whether QT is disabled or the user's need is urgent, if QT places an outgoing call, that call is routed to Parallel Ringing (PR). PR places concurrent outgoing calls to a list of addresses supplied by the subscriber, for example the addresses of the subscriber's mobile phone, home phone, and work phone. This is the last box in the target zone of the subscriber, because each outgoing call has a different target address.

Note that an interface box to a phone or similar user device will send an avail signal upstream when the user answers the phone. If PR receives an avail signal from one of its downstream branches, it tears down the other branches and forwards the avail signal upstream. If it receives *unavail* from all branches or times out, it tears down all the branches, sends *unavail* upstream, and terminates. The unavail signal will pass transparently through QT and will trigger RVM to record a message, just as if the usage reached a phone and the phone was busy.

This completes the brief description of the four features. Note that each feature is described strictly in terms of its own concerns. Its function and observable effects make perfect sense if it is the only feature that the target address subscribes to.

The "activation" class of interaction among these features is based on the following feature properties, which are easy to extract from feature programs in the form of finite-state machines [12]:

- If a feature receives an incoming call and does not place an outgoing call, it *cancels* all features with later precedence, because they will not even appear in the usage for this subscriber.
- Some functions of some features *are triggered* by receiving *avail* or *unavail* from downstream. For example, *avail* triggers PR to tear down other branches, and *unavail* triggers RVM to record.
- Some of these features *generate* unavail signals upstream (CB, QT, PR) and one *generates* an avail signal upstream after receiving an unavail signal from down-

[3]Because the signaling channel is FIFO, *unavail* will arrive before *teardown*, so that boxes upstream will know why the call is being torn down. Most unavail signals are followed immediately by *teardown*.

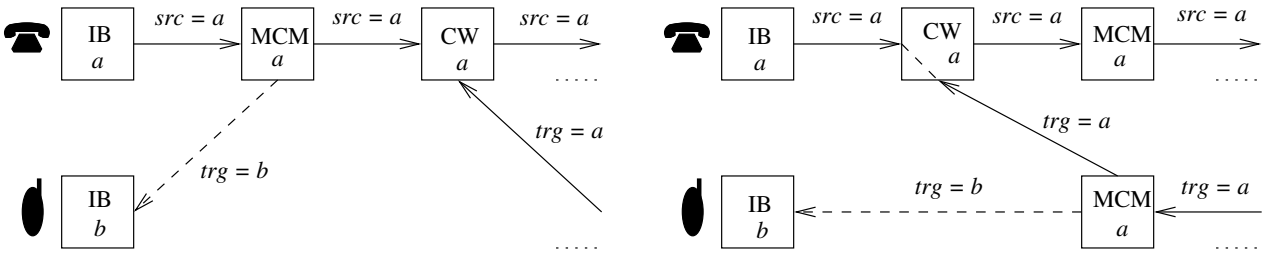

Figure 3: Two compositions of Call Waiting and Mid-Call Move.

stream (RVM).

By evaluating the potential interactions caused by canceling, triggering, and generating, we come to the following conclusions:

- CB should be first. If it blocks, it cancels all subsequent features, which is desirable. If it blocks it generates *unavail*; it would be undesirable for that signal to trigger RVM, which would happen if RVM preceded CB.
- RVM should precede both Both QT and PR. Both of the latter generate *unavail* and it is desirable for these signals to trigger RVM, which can only happen if RVM is upstream of them.
- QT should precede PR. If QT is enabled and the call is not urgent, no phones should ring. So it is desirable for QT to cancel PR in this case.

This reasoning provides a total precedence order on the four features that enables all desirable activation interactions and prevents all undesirable ones. The same kind of analysis can be applied to other shared signals.

It is important to note that this is one of many possible examples of target-zone feature sets. The functions of these features could be bundled into features differently. Changes in feature purpose and bundling of functions could result in different decisions about how the features should interact. With very little extra programming, it is possible to generate a wide range of possible and desirable behaviors.

4.2 Multi-party interactions

A free feature box has exactly one incoming call; it cannot have more than one because each incoming call is routed to a fresh instance of the box type. In contrast, if an address subscribes to a bound feature box type, then *all* calls routed to that feature for that address go to the same box. Thus bound boxes make it possible for two separate usages to join into one usage graph.

Bound boxes usually implement multi-party features, such as Call Waiting (CW). A subscriber usually subscribes to CW in both source and target regions, because its function is desired whether the subscriber's phone is busy in a caller role or busy in a callee role.

CW is initially transparent. Its function is triggered when it receives an incoming call for the subscriber when CW is already supporting (transparently) a connection between the subscriber and some far party. At this time it sends an *alerting* signal to the new call as if the subscriber's phone were ringing, and sends a signal to the subscriber that a call is waiting. On the subscriber's command, it will switch the subscriber back and forth between talking to the old party and talking to the new party. If CW receives another incoming call while it already has one call waiting, it will refuse it by generating *unavail* and then *teardown*.

Another multi-party feature much appreciated by our users is Mid-Call Move (MCM). MCM allows a subscriber to move from one device to another during a conversation. For example, a subscriber can be talking on a home phone, realize that it is time to leave the house, and move the call to a mobile phone without interrupting the conversation. On receiving a command from the user, MCM places an outgoing internal call to the new device. When the new device rings, the subscriber answers it and hangs up the other phone. Like CW, a subscriber usually subscribes to MCM in both regions, so that he can use the feature regardless of who initiated the conversation.

A typical multi-party interaction is illustrated by Figure 3. The device with address *a* subscribes to both CW and MCM in both regions. In the left usage, MCM precedes CW in the source region and succeeds it in the target region, so MCM is always closer to the device than CW. If the subscriber triggers MCM when CW has a call waiting, both calls to far parties are carried along to the new connection with the phone having address *b*.

In the right usage of Figure 3, the precedence orderings of CW and MCM have been reversed in both regions. Note that MCM is implemented by free boxes, because the function of MCM does not require joins. This is why we see two instances of MCM, both on behalf of *a*.

The right usage is troublesome when both features are active. The typical default behavior of a feature is to forward unrecognized signals transparently, so that it does not interfere unnecessarily with the functions of other features or user devices. If CW forwards unrecognized sig-

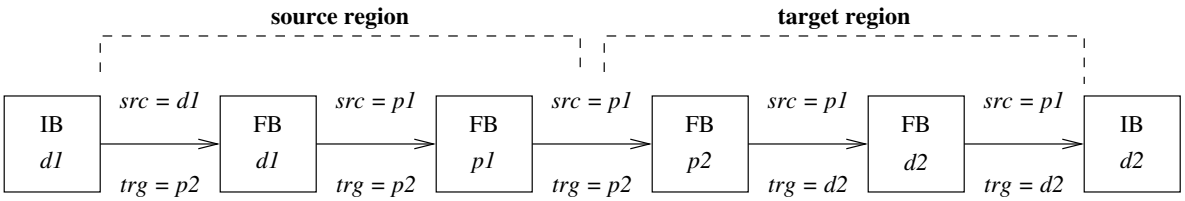

Figure 4: Proper address translation with device addresses *d1, d2* and personal addresses *p1, p2*. Each zone has one feature box.

nals from the subscriber to the far party currently selected, as shown by the dashed line in CW, then a move command goes only to the lower instance of MCM. After the subscriber hangs up phone *a* and the lower MCM tears down its call to *a*, the waiting call will be lost. If CW forwards unrecognized signals from the subscriber to both far parties, then a move command goes to both instances of MCM. The resulting behavior (as both MCM boxes try to connect to *b*) will be time-dependent and probably undesirable.

DFC modularity is especially valuable when it comes to multi-party features. In Figure 3, each of the CW and MCM boxes controls at most three internal calls, which is not difficult to program. If we added Three-Way Calling (TWC) to the feature set, CW and MCM boxes would remain the same, and each TWC box would also control three internal calls.

If these features were programmed monolithically, however, the implementation of CW and MCM would have to control four internal calls simultaneously, and with TWC active there could be six. The complexity of a monolithic program rises very rapidly with the number of simultaneous calls to be controlled. This programming problem is particularly acute when multi-party features are added incrementally.

Multi-party features introduce other issues as well. Here are three in addition to the previously mentioned question of replicating or selectively forwarding signals to multiple far parties:

- An internal call linking a feature box to its subscriber may have to multiplex signals from multiple far parties.
- Signaling paths between devices may contain irregular patterns. For example, on the left side of Figure 3, the path between device *b* and the lower far party has two outgoing internal calls joined at MCM, and two incoming internal calls joined at CW.
- Different devices may have different user interfaces. A feature box may interact with multiple or changing devices, and therefore have multiple or changing user interfaces.

Modularity in DFC draws attention to these issues, and they can be handled in DFC with relative ease [16]. Without the kind of help DFC offers, such issues are often overlooked or mishandled.

4.3 Interactions caused by address translation

When a feature box in the target region of an address *t1* uses the option in the *continue* routing method to change the target address to *t2*, it has an important effect on assembly of the usage. There will be no subsequent boxes on behalf of *t1*, even if some of *t1*'s features have not been included yet. Instead, there will be boxes on behalf of *t2*. The same thing happens when a box in the source region translates the source address.

Address translation is a powerful mechanism. It performs many functions and solves many problems. It can also cause problems, in the form of interactions between features of different zones in the same region. Precedence does not help to manage these interactions, because the relative order of zones in a region is determined strictly by address translation. Precedence can only affect the order of features within a zone.

A feature interaction in this category occurs when a user with address *p* has Parallel Ringing (PR) as described in Section 4.1, configured to ring several devices including his mobile phone at address *d*. At the same time, the user's mobile phone has Unconditional Call Forwarding (UCF), which he sets to forward all calls to the address *p* that subscribes to PR. Since UCF and PR are both implemented by free feature boxes, the initial *setup* for a call to one of these two addresses (*d* or *p*) will create a usage in which a new instance of PR(*p*) translates the target address to *d*, creating a new instance of UCF(*d*) that translates the target address to *p*, creating a new instance of PR(*p*) that translates the target address to *d*, and so on until the system runs out of resources. However strange it might seem, many users will do this if given the chance.

The best general approach to managing address translation begins with a recognition of what each address represents [13]. In the example above, *p* represents a *person* and *d* represents one of the telecommunication *devices* that the person uses. Other common kinds of address rep-

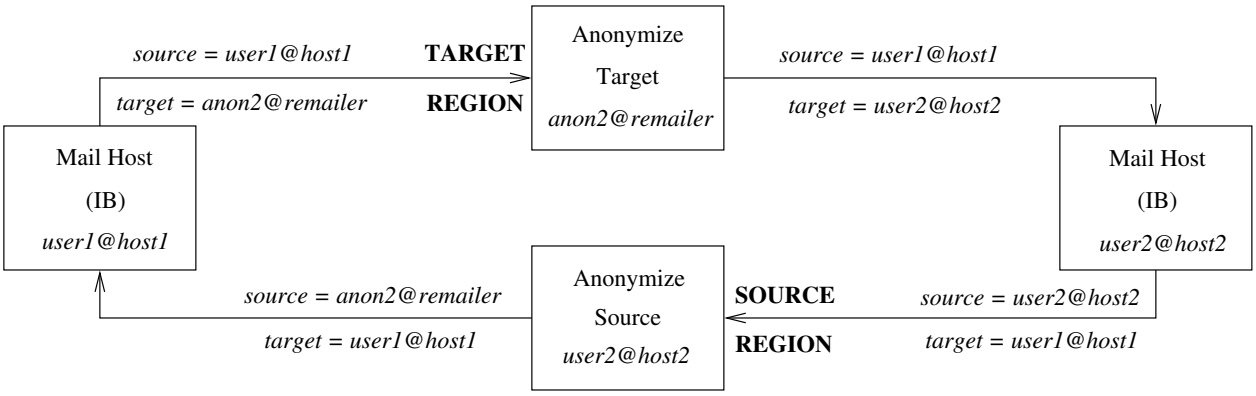

Figure 5: Symmetric feature boxes provide anonymous email.

resent *groups* of people, *roles* that a person plays, or *organizations*.

This recognition is useful all by itself because it emphasizes that an address should subscribe to the features that work on behalf of the entity that the address represents. For example, an address representing a person might subscribe in the source region to a feature that allows the person to translate the source address to an address representing his role as an employee. If the employee exercises this option, his outgoing call would be routed through a source feature that charges the call to his employer.

After categorizing addresses, it is next necessary to impose a partial order on the categories. This order is based on "abstractness" vs. "concreteness" of the thing represented *as a network endpoint*. For example, a device is very concrete, being literally a network endpoint. From this perspective a person is more abstract, being reachable from many network endpoints, and a group of people is more abstract still.

There are three principles, all based on the abstraction hierarchy of addresses, for organizing address translation [13]. The first principle is that source-region features should only translate source addresses to addresses more abstract than their own, and target-region features should only translate target addresses to addresses more concrete than their own. This principle creates the pattern illustrated by Figure 4. It prevents the bad feature interaction above (between UCF and PR) because UCF cannot translate *d* to *p*.

The second principle is symmetry between the source and target regions, which is often required for correct behavior. For example, one of the bad email feature interactions identified by Hall [7] occurs when *user2@host2* is maintaining anonymity in an email conversation with *user1@host1*. *User2* is known to *user1* as *anon2@remailer*; email to this address goes through an anonymous remailer, which forwards it to *user2@host2* while retaining the original source address.

The trouble arises when *user2* goes on vacation. On receiving email, his vacation program automatically replies with "vacation" email, reversing the source and target addresses. Thus *user1@host1* receives email directly from *user2@host2*, whose identity is now revealed.

This problem is due to the lack of a source-region feature box to balance the remailer, which is a target-region feature box. A solution is shown in Figure 5. In this figure the mail hosts are interface boxes, and the vacation program is part of the mail host for *user2@host2*. This address subscribes to Anonymize Source (AS) in the source region. AS has in its operational data the correspondents to whom *user2* wishes to be anonymous. When it receives email for one of these correspondents, it translates the source address to *anon2@remailer*.

With this symmetric solution, all the user has to do to create an anonymous conversation is to put the correspondent address in AS data before sending the first email. The rest is automatic.

For simplicity, Figure 5 is somewhat different from the way that real email works. [13] describes several realistic schemes based on the same underlying principle.

The third principle is that internal addresses can be produced and consumed by feature boxes for their own coordination purposes, provided that an abstraction hierarchy is preserved. This principle can be illustrated by the Answer Confirm (AC) feature.

Parallel Ringing (PR), as seen in Section 4.1 and Figure 2, has a serious vulnerability: if it tries to ring a device configured for immediate voice mail (for example, a mobile phone that is turned off), then voice mail will answer immediately, aborting PR's attempts to reach other phones that might have been answered by people. AC removes this vulnerability by reacting to *avail* from downstream. It connects the answered phone to an IVR server, plays an announcement "This is a call for . . . ," and prompts for

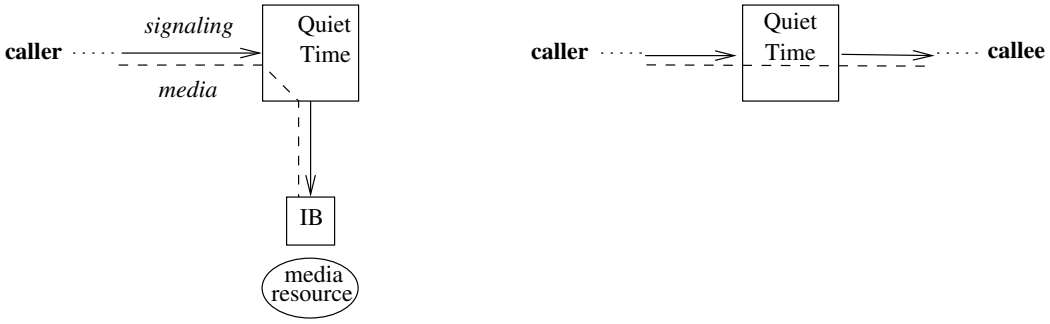

Figure 6: How Quiet Time uses a media resource.

a touch-tone acknowledgment. If the answering entity is voice mail it will fail this test, and AC will send *unavail* upstream instead of *avail*.

AC is a device feature, in the sense that some devices should have it, and all other devices should be unencumbered by it. We cannot expect device addresses to subscribe to it, however, because it acts on behalf of the personal address subscribing to PR. For a personal address *p* subscribing to PR and a device address *d* needing AC when called by PR(*p*), the best solution is to introduce a new address *p2d* that PR calls instead of *d*. The address *p2d* subscribes to AC. When AC receives an incoming call, it places an outgoing call to *d*. The address *p2d* can be described as "internal" because it appears in the usage only between PR and AC, and in the abstraction hierarchy only between *p* and *d*.

4.4 Media-related interactions

In Section 4.1, both Record Voice Mail and Quiet Time (QT) used media resources to implement IVR dialogs with callers. To do this, a feature box places an internal call to a suitable media resource (IVR server), which has a DFC interface box just like any other device. The feature box uses the *new* routing method and an empty source address, because this call should not be routed through any of the feature boxes of the caller or callee. It will be routed through target features of the resource, if any.

The setup signal of the new call carries the identifier of a script that the resource should work from. The script acts as a flowchart combining announcements and prompts to be played to the caller, decisions made by the caller through touch tones, and recording sessions.

When a QT box has established an internal call to a resource, it connects the voice channel to the caller with the voice channel to the resource, so that voice flows between those endpoints in both directions, as shown in Figure 6 (left). If QT places an outgoing call, then QT tears down the call to the resource and connects the voice channel of its incoming call to the voice channel of its outgoing call (figure right).

A box programmer needs two main primitives to control media channels: a primitive to *link* two media-channel endpoints together within the box, as shown in Figure 6, and a primitive to *hold* a media endpoint within a box, which means keeping the media channel open even though there is no media flow at the moment [15]. Binary links are sufficient even for conferencing, because conferencing applications always connect all the participants individually to a *conference bridge*, as shown in Figure 7. A conference bridge is a media server that mixes all its input channels and sends the mix to all its output channels.

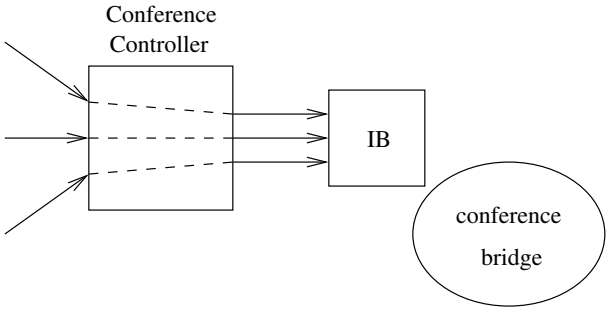

Figure 7: A Conference Controller feature box connects all the participants to a conference bridge.

A typical media-related feature interaction in other architectures is as follows. Alice will participate in a conference call today, and asks the conference server to call her at her personal address when the conference begins. This address subscribes to Find Me (FM), which is like Parallel Ringing except that it tries different device addresses sequentially. Because FM can take some time to find Alice, it first plays an announcement to the caller, "Please wait while we find Alice for you."

When the conference server (playing the role of caller) receives the signals to open a voice channel for the

announcement, it believes that Alice has answered the phone. It prompts for the callee to enter a conference code, times out, prompts again, and then disconnects the call. Alice misses her conference.

DFC avoids feature interactions in this class by recognizing that opening a voice channel and connecting to a person are two different things. Often the former precedes the latter, because of the use of the voice channel for signaling and control purposes. Confusion between these things is avoided by having separate signals for them, with *avail* being the "connected to a person" signal.[4]

A second class of media feature interaction is caused by the fact that feature boxes use the voice channel for IVR dialogs independently. Consequently, there is a danger that two features in a usage might attempt to use the voice channel to the same user at the same time.

The majority of IVR dialogs are triggered when the box receives a setup signal from upstream or an outcome signal (*avail* or *unavail*) from downstream. Contention for the voice channel is easily avoided by a convention that the triggering signal is a token that confers the right to use the voice channel. If a feature box wants to use the voice channel, it must complete its IVR phase before forwarding the token signal [14].

A third class of media feature interaction is illustrated by Figure 8. A, B, and C are phones, while R is an IVR server; interface boxes are omitted. A subscribes to Call Waiting (CW), and is using it to switch between far parties B and C. The figure is a snapshot in which C is selected, so CW is connecting the voice channel to A with the voice channel to C.[5]

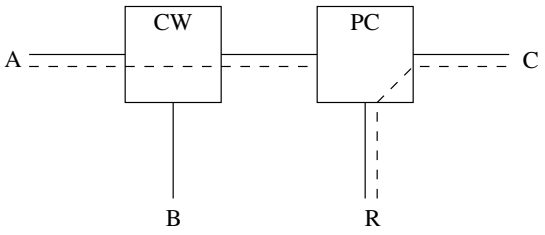

Figure 8: A media feature interaction resolved by proximity in the usage graph.

At the same time, C called A with the help of a prepaid card. The account on the card is exhausted, so the Prepaid Card (PC) feature box has interrupted C's voice connection to the far party, put the far party on hold, and is connecting C to an IVR server through which C can authorize payment. This is a feature interaction because, from the viewpoint of CW, A is talking to a far party. It is only because of the presence of PC that A is on hold.

In DFC the management of this feature interaction is automatic and obvious from the figure: PC has priority over CW in controlling the voice channel to C because PC is closer to C in the usage graph than CW is. CW only has the power to connect A to B, or to leave A on hold in the PC box. Note that these feature interactions cannot be resolved by the token convention because they occur after the last token signal (*avail*) has been sent by one device and received by another.

A fourth class of media feature interaction concerns different media channels of the same call. Bandwidth limitations could constrain the number and type of media channels that can be used simultaneously. There is not enough experience with multimedia features to discuss this issue yet.

4.5 Data interactions

Subscription data need not be static. A user could turn a feature off and on by unsubscribing and subscribing, respectively, though a Web interface. The time of day could also be used to alter subscriptions automatically.

To prevent feature interaction through operational data, this data is usually partitioned by subscriber address and feature, so only one feature box type on behalf of one subscriber can access a particular datum.

It is also possible to partition data less strictly, for example by subscriber address only. A subscriber's address book could safely be accessed by all the feature boxes of that subscriber. One feature might even add an entry, which another feature ultimately uses. This is a very indirect and benign feature interaction.

The partitioning constraint is also softened by the fact that, as an increment of functionality, a feature is sometimes implemented by more than one box type. For example, Anonymize Target and Anonymize Source in Section 4.3 are both required for anonymous email.

The current trend is toward *converged* services that have both telecommunication and Web aspects. Web services are naturally data-intensive, and provide the most convenient and popular user interface to data. For example, all the services we have built have Web user interfaces to DFC subscription and operational data.

We plan to investigate converged services, data modularity, and data feature interactions as aspects of a single research topic. This makes more sense than treating DFC operational data differently from Web-services data, especially in a converged environment.

[4]Being connected to a voice mail recorder is an adequate substitute for a person in most cases (Section 4.1), but is not adequate in the presence of Parallel Ringing (Section 4.3). This is typical of the subtleties of feature interaction.

[5]More precisely, CW is connecting A to the far party reached through the call on its right, as opposed to the far party reached through the call on its bottom.

5 Brief notes on implementation

Although this paper is not much focused on implementation, it makes sense to say a little about how DFC is now implemented. These notes will make some comparisons and evaluations more intelligible.

SIP [10] is the dominant signaling protocol for IP multimedia services. Because telecommunication devices and other network elements use SIP, DFC implementations must interoperate with SIP.

DFC is independent of system architecture, because the feature boxes of a DFC usage can be located anywhere. If two adjacent boxes are on the same host, then the signaling channel of the DFC internal call between them will be implemented with software queues. If two adjacent boxes are on different hosts, then the signaling channel between them will be implemented with a network connection.

Consequently, one possible implementation architecture is to implement all the feature boxes subscribed to by a device address in the software of the device itself [3]. But there are many situations in which this architecture is undesirable or impractical, so that device feature boxes must be implemented in network servers called *application servers*. Even if all devices implement their own features, there are many abstract addresses whose features must be implemented in application servers because there is nowhere else to put them. So most usages will be distributed over telecommunication devices and one or more application servers.

Our first implementation of DFC [2] ran in an application server created just for this purpose. Both subsequent implementations of DFC have run within commercial or open-source SIP application servers, to make use of their performance, reliability, and operational conveniences.

The SIP Servlets API is a standardized way of programming SIP application servers, offering "servlets" as functional modules. In our second DFC implementation (the first one on a SIP Servlet container) the entire DFC runtime environment was packaged in one servlet.

In our third and current implementation, individual servlets simulate DFC feature boxes. To implement DFC internal calls between boxes in the same SIP Servlet container, we use SIP in a stylized way that approximates the DFC protocol. We have persuaded the community that a DFC-like router is the best way to perform application composition, so that DFC routing is now enabled by all SIP Servlet containers [9], and an open-source DFC router for SIP Servlet containers is freely available.

By far the most difficult part of implementing DFC on the Internet is implementing media flow and control. DFC conceptualizes media streams as passing through feature boxes, because that is the best way to understand them and to specify what behavior is required. This point is illustrated by Figure 8. On the Internet, however, it is necessary to make a distinction between signaling channels and media channels. Signaling channels need to go through application servers; as they are low-bandwidth, this is not a problem. Media channels are high-bandwidth and should take the most direct route between media devices. It is too inefficient to route media packets through one or more application servers, which may be located far from either media device.

The result of this situation is a difficult problem of distributed control. Media flow must be implemented by instructing the media devices to send media packets directly to one another through the Internet. These instructions come from feature boxes like Call Waiting and Prepaid Card, which prescribe different media flows in their different states. These feature boxes do not know about one another. Yet the instructions received by media devices must correctly reflect the *composition* of the states of all relevant feature boxes.

Our first implementation simplified this problem by solving it in a separate, but centralized, component to which all feature boxes report their states [4]. We have since found an efficient, completely distributed solution for DFC [15], and are adapting it for use within SIP [6].

6 Experience and evaluation

6.1 Experience

Our experience with DFC began with design and implementation of several service prototypes, for demonstration and trial use within AT&T. Lessons from the most ambitious of these are captured in [16].

In 2003 were were given the opportunity to develop the advanced features for AT&T's first consumer VoIP service. For the first trial, we specified, implemented, and delivered eleven features two months from the inception of the project. This feature set included such challenging features as Parallel Ringing, Ten-Way Calling, and Mid-Call Move. We also implemented voice mail. System testing and subsequent trials revealed very few bugs in the feature server. In 2004 the service went public, winning two industry awards citing its voice quality and advanced features. By 2005 the service was supporting close to 100,000 customers [1].

This unprecedented speed and quality of development was possible because of separation of concerns. Different people could safely and independently work on different features at the same time. With the DFC architecture providing structure, overlapping tasks could also be performed in parallel. For example, once we had an informal specification of each feature, feature implementation and analysis of feature interactions could proceed at the same time. Feature interactions were managed mostly by precedence and occasionally by small feature changes.

The second major system built with DFC is the teleconferencing service now used internally by AT&T. On a typical workday, the service handles millions of minutes of calls. It was originally designed for our second implementation of DFC, and is now being re-engineered for our third implementation of DFC, with interesting differences between them. As with the consumer VoIP service, there have been very few post-deployment bugs in the feature code [5].

6.2 Failures of modularity

As anyone with software experience will expect, DFC modularity is not perfect. The independence of features is not always as complete as the overview of Section 3 implies. As an example of a typical exception, consider a Call Log (CL) feature that writes a record of each incoming call to its subscriber's operational data. The record should show if the caller talked to the subscriber, recorded a voice message, or neither.

CL must precede Record Voice Mail (RVM) in the subscriber's target zone, because once RVM receives *unavail* from downstream it tears down its downstream call (if not torn down already) and downstream boxes disappear. However, the *avail* and subsequent teardown signals sent upstream by RVM do not tell CL whether the caller left a message or not. The only way to provide the desired interaction between CL and RVM is for RVM to send a special-purpose signal or signal field to CL indicating whether it recorded a message. In this case interaction between these two features must be programmed explicitly into both features.

This example illustrates the problem of status signals, which is the one part of DFC that does not feel "settled." Built-in status signals such as *avail* and *unavail* provide a common language for communication among features. They support modularity because a feature can react to such a signal without knowing whether it came from a user or another feature. More status signals means more modularity. For example, if "message recorded" were part of the built-in signal vocabulary, then CL and RVM would be independent. On the other hand, the more built-in signals there are, the more work it is to program each feature, and the harder it becomes to give each signal a feature-independent meaning.

Arguably the worst failure of DFC modularity concerns treatments (call forwarding, call queueing, interrupt, voice mail) for failure (busy, no answer) when there are multiple zones within a target region. From Section 4.3, we assume that the zones of more abstract addresses precede the zones of more concrete addresses. Because failure signals travel upstream, the most concrete features receive them and act upon them first. Abstract features receive failure signals only when concrete features cannot fix the failures.

Failure treatments, and the situations in which failures arise, are a big subject. Suffice it to say that sometimes the most abstract feature should be triggered first. This behavior can be achieved in various ways (Answer Confirm, shorter timeouts in the more abstract features, explicit cooperation among features at different levels of abstraction), but all of them can be seen as subverting the native mechanisms of the architecture.

Finally, a designer should be able to use any legacy component (with a suitable purpose and interfaces) as a feature box program. This is not always possible, because an unfortunate grouping of functions may include functions with different natural places in the precedence order. The result is that the precedence relation becomes overconstrained, i.e., cyclic. The easy fix is to separate the feature implementation into two box programs.

6.3 Modularity successes

Despite occasional exceptions, DFC modularity has proven to be very successful. All of the experience related in Section 6.1 indicates that it is intuitive and effective from an engineering viewpoint. It supports fast development and quality code.

For evidence of a different kind, consider Hall's study of feature interactions in 10 common email features [7]. Of the 26 undesirable interactions identified by Hall, 14 have something to do with address representation, address translation, or feature application. All 14 of these are diagnosed by, and could be prevented by, the DFC techniques for managing feature interactions caused by address translation [13].

The original purpose of DFC modularity was to support easy development of features as additions or exceptions to a basic telecommunication service. In keeping with common practice, customers could subscribe to features individually, making each one optional. An interesting lesson learned from all our experience is that DFC seems to provide "all-purpose" modularity: it works fairly well regardless of what functions are being decomposed into modules, or why the decomposition is desired. In addition to the expected purpose, we have so far identified many other (somewhat overlapping) purposes served by DFC modularity.

First, a feature can be an addition or exception, not to the basic service, but to another feature. This is illustrated perfectly by the addition of Answer Confirm to solve a problem with Parallel Ringing.

Second, as with other forms of modularity, DFC modularity can insulate a system from the effects of probable change. In our teleconferencing service, we prototyped features that we ended up dropping, because their value was not sufficient to justify their user complexity and re-

source costs. Because they were optional modules, they were trivial to remove. We also used feature boxes to encapsulate uncertainty concerning which vendor's media resources would be used. The content of these boxes most closely resembles a software library [5].

Third, many box programs are re-usable modules. For example, consider a Call Forwarding on Failure program with two parameters: a forwarding address and a type of failure (because unavail signals can have failure types attached). This program can be used to implement a variety of features, including RVM (on any failure, forward to voice mail resource) and Redirect on No Answer. In both deployed services, we re-used several box programs from earlier prototypes.

Fourth, a feature box can easily be inserted into a usage as an adaptor. Adaptors can enhance the re-usability of other box programs [5]. DFC feature boxes used as adaptors are extremely valuable for solving problems in other technologies, because they are powerful and quick to deploy. Integration testing of the consumer VoIP service revealed many integration problems, due to immature vendor-supplied components, inadequate standardization, and innate deficiencies in other technologies such as SIP [1]. We were able to fix many of these problems immediately by building software adaptors. Having such adaptors as separate modules is advantageous because they can be removed easily from the software when technologies and standardization improve.

Finally, off-the-shelf servers or other components can be treated as feature boxes and composed with other features. When consumer VoIP migrated to a vendor-supplied voice mail server, we were able to improve its integration with other features significantly by treating it as an idiosyncratic DFC resource. A subscriber can call the server, listen to a message, enter a code, and have the voice mail server call the person who left the message. If that call from the server is routed by DFC as coming from the subscriber, then the ensuing usage contains the subscriber's source-region features such as Ten-Way Calling and Mid-Call Move. If the call from the server is not routed by DFC, the subscriber does not have his normal features available.

6.4 Analysis of feature interactions

The first step in managing feature interactions is to analyze each box program in a feature set to see if it has interaction-prone behaviors such as generating signals (Section 4.1) or translating addresses (Section 4.3). Individual box programs are small, and this should be easy to do [12].

The second step is to calculate all potential feature interactions, based on these behaviors. The third step is to classify each potential feature interaction as desirable or undesirable. The fourth step is to derive from this information, if possible, a set of precedence constraints that enables all the good interactions and prevents all the bad ones.

All steps but the third one are easily automatable, while the third one relies on human knowledge of what goals the features are intended to achieve. The real problem with this straightforward approach, however, is that it generates too many potential feature interactions for a person to pass judgment on. A practical approach must combine heuristics, partial constraints, and dependencies to prune the potential interactions. Then requirements engineers will be able to find and consider the important ones.

A typical usage in the consumer VoIP service had at least 20 different feature boxes [1]. The principles in Section 4 separated concerns well enough to make manual analysis possible for this service, but not ideal because of the scope for human error. Other real feature sets could be much larger because they could have many alternative features from which subscribers can choose. Manual analysis will not be feasible for these larger feature sets.

We have not been able to do much work yet on automated analysis of feature interactions, because of our long journey through the SIP jungle. When we emerge from it, analysis will be high on our list of priorities.

Analysis of feature interactions is an opportunity as well as a burden. The structures, properties, and principles used to manage feature interactions are a precious kind of domain knowledge in their own right. Section 4 provides numerous examples of this.

6.5 Performance

Our current implementation of DFC runs on SIP Servlet containers compliant with the new standard [9]. Both our implementation and the containers are too new to say much about performance, except that it does not seem to be a pressing problem.

In general, we expect all forms of modularity to impose overhead, and can accept that overhead if it is not excessive. Early SIP Servlet containers expected to run exactly one servlet per external call, rather than a chain with many servlets, so their descendants may implement servlets in a way that is too heavyweight for DFC modularity. If this proves true, some targeted optimization of servers will be necessary.

From the system viewpoint, a usage is a graph of devices, application servers, and network connections. In this context there is reason to believe that the DFC protocol is considerably more efficient than SIP [15]. The conclusion is drawn from analyzing message traces rather than measuring real deployments, however, so it cannot be considered definitive.

7 Conclusion

For telecommunication systems, DFC's form of modularity is a clear and proven success. Now our most pressing research problem is to understand converged applications, where the DFC architecture must interoperate with Web services, which have a very different architecture. The challenge is to compose the architectures in such a way that each view has its own appropriate form of modularity.

In practice, DFC will live on as a overlay structure imposed on SIP. In containers compliant with the new standard, SIP servlets can be programmed and invoked just like DFC feature boxes, with the sole difference being the protocol they must use.[6] The early work on Boxtalk, a high-level programming language for DFC feature boxes [17], is now evolving into StratoSIP, a high-level programming language for SIP servlets. StratoSIP restores much of the simplicity of the DFC protocol by making the right abstraction of SIP.

It appears that there are important Internet design issues, and networked applications other than telecommunications, that could benefit from the ideas in DFC. Investigating these relationships is another compelling area of future research.

References

[1] Gregory W. Bond, Eric Cheung, Healfdene H. Goguen, Karrie J. Hanson, Don Henderson, Gerald M. Karam, K. Hal Purdy, Thomas M. Smith, and Pamela Zave. Experience with component-based development of a telecommunication service. In *Proceedings of the Eighth International Symposium on Component-Based Software Engineering*, pages 298–305. Springer-Verlag LNCS 3489, May 2005.

[2] Gregory W. Bond, Eric Cheung, K. Hal Purdy, Pamela Zave, and J. Christopher Ramming. An open architecture for next-generation telecommunication services. *ACM Transactions on Internet Technology*, 4(1):83–123, February 2004.

[3] Eric Cheung. Implementing endpoint services using the SIP Servlet standard. In *Proceedings of the Fifth International Conference on Networking and Services*. IEEE, April 2009.

[4] Eric Cheung, Michael Jackson, and Pamela Zave. Distributed media control for multimedia communications services. In *Proceedings of the 2002 IEEE International Conference on Communications: Symposium on Multimedia and VoIP—Services and Technologies*. IEEE Communications Society, 2002.

[5] Eric Cheung and Thomas M. Smith. Experience with modularity in an advanced teleconferencing service deployment. In *Proceedings of the Thirty-First International Conference on Software Engineering*, 2009. to appear.

[6] Eric Cheung and Pamela Zave. Generalized third-party call control in SIP networks. In *Proceedings of the Second International Conference on Principles, Systems and Applications of IP Telecommunications*. Springer-Verlag LNCS, 2008. To appear.

[7] Robert J. Hall. Feature interactions in electronic mail. In M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI*, pages 67–82. IOS Press, Amsterdam, 2000.

[8] Michael Jackson and Pamela Zave. Distributed Feature Composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.

[9] JSR 289: SIP Servlet API Version 1.1. Java Community Process Final Release, `http:// www.jcp.org/ en/ jsr/ detail?id=289`, 2008.

[10] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. IETF Network Working Group Request for Comments 3261, 2002.

[11] Mary Shaw and David Garlan. *Software Architecture*. Prentice-Hall, 1996.

[12] Pamela Zave. An experiment in feature engineering. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, pages 353–377. Springer-Verlag, 2003.

[13] Pamela Zave. Address translation in telecommunication features. *ACM Transactions on Software Engineering and Methodology*, 13(1):1–36, January 2004.

[14] Pamela Zave. Audio feature interactions in voice-over-IP. In *Proceedings of the First International Conference on Principles, Systems and Applicatons of IP Telecommunications*, pages 67–78. ACM SIGCOMM, 2007.

[15] Pamela Zave and Eric Cheung. Compositional control of IP media. *IEEE Transactions on Software Engineering*, 35(1), January/February 2009.

[16] Pamela Zave, Healfdene H. Goguen, and Thomas M. Smith. Component coordination: A telecommunication case study. *Computer Networks*, 45(5):645–664, August 2004.

[17] Pamela Zave and Michael Jackson. A call abstraction for component coordination. In *Proceedings of the Twenty-ninth International Colloquium on Automata, Languages, and Programming: Workshop on Formal Methods and Component Interaction*. University of Málaga, 2002.

[6] This view is supported by our suite of open-source tools, available at `echarts.org`.