Proceedings of the ICALP2002 satellite workshop "Formal Methods and Component Interaction", Malaga, July 2002.

A Call Abstraction for Component Coordination

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

Michael Jackson
AT&T Laboratories—Research
London, UK
jacksonma@acm.org

24 June 2002

Abstract

DFC is a component architecture for telecommunication services. Although the call protocol through which DFC components interact is intrinsically simple, using it to program useful components is quite complex. We diagnose the problems and propose a solution, which takes the form of a call abstraction embodied in a high-level, domain-specific programming language. The abstraction not only encourages correct programming, but also makes it possible to prove that components have important behavioral properties.

1 The DFC architecture

Telecommunication services are network services whose primary purpose is real-time communication among people. Although telecommunication services have a 125-year history, only since the 1960s have they been under software control. Software control engendered an explosion of features, which led to the explosion of complexity known as the feature-interaction problem [4, 7, 8, 9, 14].

The Distributed Feature Composition (DFC) architecture [12, 13, 19] was developed to address the feature-interaction problem. It is a component-based software architecture, intended for the description and development of telecommunication services. It was designed to offer generality, feature modularity, structured feature composition, and independence from network resources.

DFC is a domain-specific adaptation of the pipes-and-filters architectural style [15]. In this style, there is a fixed graph of pipes (edges) and filters (nodes). Each pipe is a unidirectional stream of data in a common format. Each filter component is a concurrent process with a completely private state, that consumes input streams and produces output streams. Because each filter is a context-independent module, filters can easily be added, deleted, or changed.

DFC has filters called *feature boxes*. In place of pipes it has *internal calls*. Internal calls are point-to-point connections obeying a fixed protocol and allowing transmission of both signals and media in both directions. The destination of each connection is determined when the connection is set up, so the graph of feature boxes and internal calls is dynamic and self-configuring rather than fixed.

Despite these differences, DFC offers the same style of modularity as pipes and filters. Feature boxes exhibit *transparency*, which means that their presence is unobservable when they have nothing to do. They also have *autonomy*, which means that they have enough power to carry out their functions without external help. Feature boxes can place, receive, or tear down internal calls. They can generate, absorb, or propagate signals traveling on the signaling channels of internal calls. They can also process or transmit media such as voice or video.

Although much work remains to be done on DFC, the results so far are promising:

- There is no known telecommunication feature, including mobile and multimedia features, that cannot be described in DFC. This is strong evidence that DFC meets the generality goal.
- Many features have been designed (and some implemented) taking advantage of DFC's feature modularity, for example [17, 20].
- DFC's structured feature composition is beginning to lead to theorems and analysis algorithms concerning important feature interactions [16, 18].
- There is an IP implementation of DFC with many advantageous properties based on DFC's resource independence [1, 2].

2 The DFC protocol and its challenges

This paper concerns the call-level signaling performed by DFC feature boxes: placing calls, receiving calls, tearing down calls, and exchanging status information on the signaling channels of calls. Although feature boxes have other behaviors—they manipulate both data and media as well—call-level signaling is the first and foremost behavior of a box program. It is the framework in which data and media operations are placed.

The protocol for DFC internal calls is relatively simple. An internal call is placed by a feature box or an *interface box*, which is a persistent module representing a telecommunication device. To place a call, a box sends a `setup` signal from a port. The `setup` signal goes to a DFC router which decides which feature or interface box should receive the call, based on various fields of the signal. The router sends the signal to a distinguished *box port* of the receiving box.

When the receiving box accepts the call, it chooses a regular port for it, and sends an `upack` signal from that receiving port back to the sending port. This creates a port-to-port signaling channel.

After sending or receiving the `upack`, a box can use the signaling channel of the call to send any status signals that it likes. This signaling channel is two-way. In either direction it is FIFO and has unbounded capacity.

Either port can send a `teardown` signal to end the call; it is acknowledged by a `downack` signal from the port that receives it. (If `teardown` signals from both ports cross in transit, then both ports generate `downack` signals.) After sending or receiving a `teardown` signal at a port, a box is not allowed to send any subsequent status signals from that port.

Despite this relative simplicity, we have found that, in practice, call-level signaling is difficult to program and difficult to reason about. In the context of distributed components that interact asynchronously with other components and are intended to coordinate with them to produce meaningful global behavior, even a simple protocol becomes very complex.

We have diagnosed the programming challenges and developed an abstraction of DFC internal calls intended to make box programming easier. The abstraction is embodied in a high-level, domain-specific programming language (named "Boxtalk") for programming DFC feature boxes. The semantics of the call abstraction is written in terms of plain finite-state machines, to which the control skeleton of programs in Boxtalk can be compiled.

The challenges of DFC box programming fall into two groups. The first group, discussed in Section 3, contains complexities that emerge when trying to make boxes perform useful functions in the component context. The second group, discussed in Section 5, consists of properties that every box should be guaranteed to have. Box programs must be organized so that each property is either guaranteed automatically, or can be checked efficiently.

Each subsection of Sections 3 and 5 first presents the topic issues, then explains aspects of the call abstraction motivated by those issues. Between those two sections is a more formal interlude. For programming examples, we use two familiar features from the Public Switched Telephone Network: Call Waiting and Sequenced Credit-Card Calling.

3 Programming difficulties

3.1 Ports are too low-level

In programming a feature box, it is inconvenient to refer to a call in which the box participates by means of a port name. This makes it impossible to refer to a call that has been torn down, because the port may already be allocated to a different call. More importantly, the role of a call in a program can change, while its port cannot change.

The most familiar example is the Call Waiting (CW) feature. A CW feature box can be engaged in a maximum of three calls, one leading to its subscriber, and two leading to far parties. For the programmer, the most important fact about the two far-party calls is that one of them is connected to the subscriber call, while the other one is waiting ("on hold"). It is awkward to refer to the two calls by their port names, as these have nothing to do with which call is active and which is waiting.

To make it possible to refer to calls in a more abstract way, each call is given an internal identifier at the time it is placed or received by the box. This identifier is unique within the lifetime of the box, and is permanently associated with the call. A box program can declare any number of *call variables*. The value of each call variable is either a call identifier or a distinguished value `noCall` (all call variables are initialized to `noCall`). Box programs refer to calls by means of call variables.

To separate programming from port allocation completely, it is necessary to assume that a box can have an unlimited number of ports. This means that the implementation can never fail to find an available port when one is needed for a call. This assumption is easy to satisfy in a software implementation of feature boxes.

Four Boxtalk statements change the values of call variables:

- `rcv(c)` receives a new call if a `setup` signal arrives for it. The identifier of the new call is assigned to call variable `c`.
- `new(c)` and `ctu(c0,c)` place new calls. The identifier of the new call is assigned to call variable `c`.[1]
- An assignment statement such as `c1,c2 = c2,-` moves call identifiers from one variable to another. In this assignment, `c2` gets the value `noCall`. The variables on the right-hand side must be a subset of the variables on the left-hand side, and no variable can appear on the right-hand side more than once.

These statements preserve the important invariant that no two call variables ever point to the same call. If two call variables have the same value, their value must be `noCall`.

Two sets of call identifiers figure prominently in the semantics of the language. The set `portAllocated` contains the identifiers of all calls currently allocated to ports; these are the calls that currently exist according to the DFC protocol. The set `known` is the union of the values of the call variables, minus `noCall`. A call in `known` is accessible to the program, because the program has a call variable with which to refer to it.

These sets often overlap, but they are, in fact, independent. A call can be in `known` and not in `portAllocated`, if it has been torn down but its call variable has not yet been re-assigned to another call. A call can be in `portAllocated` and not in `known`, if its call variable has been re-assigned before the call is torn down (see Section 3.2).

The fields of a `setup` signal contain important information about the call initiated by it. Each `setup` signal received or sent by a box is automatically saved so that the program can refer to it at any time. For example, the value of the `src` field of the `setup` signal that initiated the call in call variable `c` is `c.stp.src`. If a call remains in `known` after it has been torn down, its `setup` signal remains accessible.

Figure 1 is a Boxtalk program for a Call Waiting feature box. It omits media processing, which is not considered in this paper. Otherwise it is complete except for some declarations and definitions which are given in an accompanying textual part. This separation allows the graphical part of Boxtalk to emphasize the all-important call-level signaling, without too much clutter from data manipulation.

[1] Roughly speaking, the difference between `new(c)` and `ctu(c0,c)` is that `ctu` continues a pipeline extended to this box by call `c0`, while `new` begins a new pipeline. The details concern the DFC routing algorithm, and are outside the scope of this paper.

Graphically, a program is a finite-state machine with four types of state:

- The *initial state* is a small black circle.
- A *transient state* is a larger clear circle.
- A *stable state* is a rectangle.
- A *termination state* is a heavy bar.[2]

The semantic differences between these types of state will be explained in Section 4.

The program has three call variables. `s` refers to a call connecting the box to its subscriber. The subscriber controls the function of the box. `a` and `w` both refer to calls connecting the box to far parties; at any time, `a` is the *active* call, which is connected to the subscriber, and `w` is the *waiting* call.

The user interface of this feature consists of two status signals `cw_indicator` and `switch`. When the box is in the `transparent` state, which means that the subscriber is connected to a far party in the normal way, it can receive a third call in the call variable `w`. The box sends `cw_indicator` to its subscriber to indicate that a call is now waiting. Note that this signal carries as a field the source field in the `setup` signal of the waiting call; this identifies the new caller to the subscriber. Whenever the subscriber wishes to switch his attention from the currently active call to the waiting call, he sends the `switch` signal.

This program relies heavily on assignments to call variables. First, in the initial state, the box receives a call in `s`. Because of how a Call Waiting box is chosen by DFC routers, this call may be either from or to the subscriber. The predicate `s_from_subscriber` (defined textually) is true if the call is actually from the subscriber. If so, it is in the right variable. The statement `ctu(s,a)` continues the pipeline extended to this box by the incoming call, by placing an outgoing call.

If `s_from_afar`, on the other hand, this call does not belong in `s`. It is re-assigned to `a`, and the pipeline is continued from `a` to `s`.

The most important assignment comes in the `call_waiting` state, when the subscriber signals a switch. The values of `a` and `w` are swapped, so that the call formerly waiting is now connected to the subscriber, and the call formerly active is on hold.

If `a` is torn down in the `call_waiting` state, the box enters a state in which there are two remaining calls, not connected to each other. A `switch` signal from the subscriber will cause these calls to be connected; it forces another re-assignment to keep the active far-party call in `a`.

The fourth assignment implements a little-known behavior of Call Waiting. Suppose the box is in

[2] CW does not have an explicit termination state.

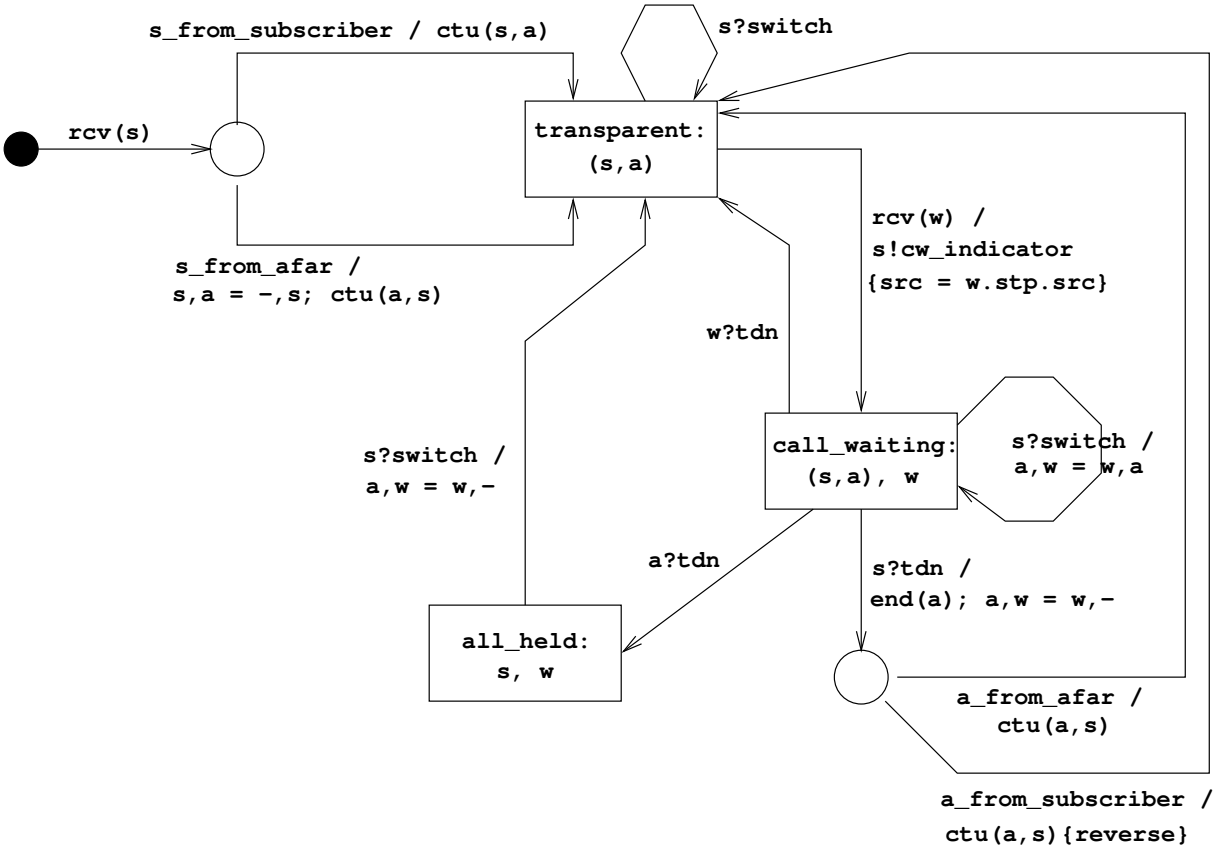

Figure 1: A Call Waiting feature box.

the `call_waiting` state, and the subscriber hangs up (`s?tdn`, short for `teardown`). Naturally this causes the box to tear down `a`.

The call in `w` is still waiting. Rather than tear it down also, the box calls the subscriber back! Before doing so, it does another re-assignment. There is a conditional branch after the assignment statement because formation of the `setup` signal used to place the call depends on whether the call now in `a` was originally an incoming or outgoing call of this box.

3.2 Protocol programming is difficult

A distributed protocol must be asynchronous. But the resultant nondeterministic delays, handshakes, and race conditions add considerable complexity to programs. In Boxtalk we address this problem by making call setup and teardown appear to be atomic, although they are not.

It is relatively easy to make execution of `rcv(c)` an atomic operation. If a `setup` signal arrives at the box in a state in which `rcv(c)` is enabled, the implementation updates internal data structures and sends `upack` as part of the execution of the statement. Note that if a `setup` signal arrives at a box when no `rcv` statement is enabled, then the signal is dismissed automatically and implicitly (see Section 3.3).

Execution of the statement `c?tdn` means that the call in `c` has been torn down from its other end. It is also easy to make execution of this statement an atomic operation, as the implementation simply sends the `downack` signal as part of it.

The statement `end(c)` is used to initiate teardown of the call in `c`. Execution of this statement usually begins with sending a `teardown` signal. The teardown phase in the ending box is not complete, however, until the box has first received every status signal sent from the far port before the far port received the `teardown` signal and stopped sending, and second has received the final `downack` signal. The late-arriving status signals are thrown away.

This "cleanup" of a port's input queue can take an unbounded amount of time, which means it cannot be part of the atomic execution of an `end(c)` statement.

Rather, it takes place implicitly and in the background, after execution of the `end(c)` statement has completed; the post-processing of an `end(c)` statement is asynchronous with respect to the control flow of the program.

If the programmer re-uses call variable `c` before the cleanup has finished, then the call being cleaned up will be in `portAllocated` and not in `known`.

A termination state (as seen in Figure 3) is the end of program execution, but it is not a final state of the underlying finite-state machine representing the semantics of the program. In a termination state, there may still be calls in `portAllocated` whose cleanup is not yet complete. When `portAllocated` becomes empty, there is an implicit transition from a termination state to a true final state.

The statements `new(c)` and `ctu(c0,c)` that place calls are the hardest to make atomic. Not only is there an unbounded wait for an acknowledgment, but also the box program may have an immediate need to send signals through the call.

Execution of the atomic language statement `new(c)` or `ctu(c0,c)` ends as soon as the `setup` signal is sent. It does not include the wait for the `upack`, even though the DFC protocol makes it impossible for a placing port to send signals on a call until it has received this acknowledgment. If a subsequent program statement sends a signal on this call, then the outgoing signal is queued internally, and sent as soon as the `upack` signal is received.

For example, the program may execute the statement sequence `ctu(c0,c); c!s1; c!s2;` *other*, where *other* does not affect call `c`. Although this is invisible to the programmer, *other* may actually be executed before `c!s1; c!s2`, if `c?upack` does not occur until after *other* is complete. The apparent and actual sequences are equivalent, because signals `s1` and `s2` can take arbitrarily long to reach the other end of call `c`.

Thus the semantics of `end(c)`, `new(c)`, and `ctu(c0,c)` statements all require implicit post-processing that is asynchronous with respect to the control flow of the program. It is mildly complex, which we consider to be a major advantage of Boxtalk; if this complexity were not built into the semantics of the language, then each box programmer would have to program it for himself. The semantics of Boxtalk need only be verified and implemented once.

In the semantics of a Boxtalk program, the call with the identifier in `c` is considered *active* after execution of a `rcv(c)`, `new(c)`, or `ctu(c0,c)` statement, and before execution of an `end(c)` or `c?tdn` statement.

Active calls are the important ones, because they are the only ones that a box programmer can manipulate. They are so important that a programmer must label every stable state with the call variables pointing to all the active calls. (This information is redundant, as explained in Section 5.2.)

All Boxtalk statements that operate on calls have preconditions based on whether the relevant calls are active:

- The precondition of `c!status`, `c?status`, `end(c)`, and `c?tdn` is that `c` points to an active call. Otherwise the statement is undefined, violates the DFC protocol, or both.
- The precondition of `rcv(c)`, `new(c)`, and `ctu(c0,c)` is that `c` does not point to an active call. Otherwise the invariant that every active call must be the value of some call variable is violated.
- Because of the same invariant, in an assignment such as `c1,c2 = c2,-`, any variable such as `c1` that does not appear on the right-hand side must not point to an active call.

The set `active` containing the identifiers of all active calls is a subset of both `portAllocated` and `known`. It is not, however, the intersection of these two sets. A call in its cleanup phase, whose call variable has not yet been re-assigned a different value, is in both `portAllocated` and `known`, and is not in `active`.

3.3 There is a need for end-to-end reasoning

Imagine that there were no feature boxes, so that each DFC internal call is placed by an interface box and received by an interface box. If the target address were invalid or if the target interface box were busy, the router could report the problem directly to the placing box. If the placing box received an `upack` signal, it would know that the call had successfully reached another interface box.

In a component architecture, on the other hand, the situation is very different, as illustrated by Figure 2. This figure is a message-sequence chart showing the detailed signaling among two interface boxes and two feature boxes.

Figure 2 shows that each internal call is completed before the pipeline is continued. It is necessary to organize signaling this way in a component architecture. If each feature box did not acknowledge an incoming `setup` signal immediately, but rather waited to receive an outcome from the target interface box, then all feature boxes would be frozen until the pipeline reached an endpoint. None of their calls would be set up, and none of the boxes could send

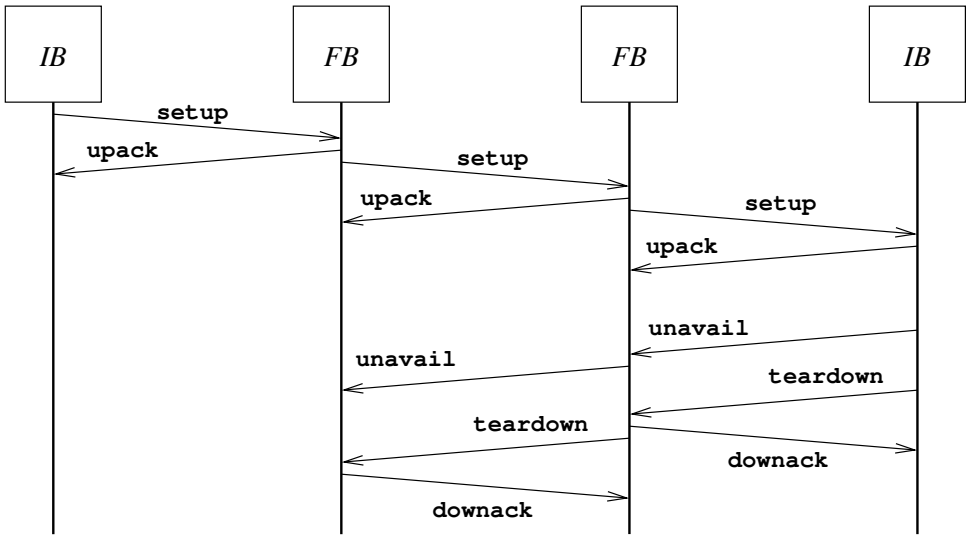

Figure 2: The piecewise nature of signaling in a component architecture.

or receive any signals. They would have very limited autonomy and usefulness as components.

Precisely because call setup is piecewise, however, it does not provide any end-to-end information. The DFC protocol has three built-in status signals intended to provide end-to-end information about the outcome of a communication attempt. The three signals are `unknown`, `unavail`, and `avail`. Together they cover the three outcomes mentioned in the first paragraph of this section.

In Figure 2, the target device is unavailable. The first feature box is triggered by the `unavail` signal; it will offer some "busy treatment" to the caller such as forwarding, voice mail, or automated retry. It absorbs both the `unavail` signal and the teardown from downstream, because if it propagated them upstream, the caller would probably hang up.

Clearly a successful box programmer must think about both the end-to-end and piecewise consequences of his program. Put another way, the programmer must think both about implementing the desired function of the box, and about making the box transparent when it is not performing a specific function. Piecewise and functional thinking tends to come more naturally, so Boxtalk helps with end-to-end and transparent programming as much as possible.

The first and most important language feature of this kind is *signal linkage*. In a stable state, two active calls are signal-linked if their call variables are combined in a parenthesized pair. If two active calls are signal-linked, then the default handling of any status signal that arrives from either call is to forward it to the other call.

For example, in the `transparent` state of CW, the two active calls are signal-linked. If a box forwards all status signals between two signal-linked calls, the box is unobservable.

The default handling of signals established by signal linkage can be over-ridden by explicit transitions. The CW box, for example, never forwards a `switch` signal from the subscriber, because the signal is only meaningful as a subscriber command to this box. Each of the three stable states has a transition triggered by a `switch` signal from the subscriber. In the `transparent` state it is ignored because there is nothing to switch; in the other two states it has an important effect.

In a stable state with active call `c` and no explicit transition on `c?tdn`, receipt of a `teardown` signal from call `c` automatically and implicitly terminates the entire box program, which includes ending all other active calls. In the CW program, for instance, there are no explicit teardown transitions from the `transparent` state—if either party hangs up, its interface box begins a chain reaction of teardowns that should soon remove the entire graph of boxes and internal calls. In the `call_waiting` state, on the other hand, there is an explicit transition on teardown of each active call. Since there are three active calls, any single one of them is dispensable.

To help illustrate the remaining aspects of the call abstraction, we introduce a program for a Sequenced Credit-Card Calling (SCCC) feature box (Figure 3).

This feature enables a caller to make a sequence of credit-card calls after a single entry of account information.

On receiving a call in `r`, the program begins by placing a call to a resource capable of implementing an interactive voice-response dialogue with the caller. The {`credit_query`} suffix on the `new` statement indicates that various argument fields are provided in the textual part of the program; these fields specify that the target of the call is a server with a "credit query" program.

In the `credit_query` state the resource reached through the call in `v` is prompting the caller for an account number and collecting the digits of that number. When it has collected a complete number and checked a database for its status, the resource first announces the result of the credit check to the caller, and second sends a signal `result` with the result. The feature box does a conditional branch on the content of this signal. If the account is a bad one, the box program terminates now. If the account is a good one, the feature box places an outgoing call, continuing the pipeline of the call in `r`.

At this point the box becomes transparent. However, if the caller chooses to disconnect from the callee by sending a special `disconnect` signal rather than by hanging up the telephone, the feature box will connect the caller to a resource that prompts for and collects a new telephone number. If this dialogue is successful, the box will place a new outgoing call on the same account.

Once this program is past the credit check, the only event that causes termination is receipt of a `teardown` signal from the call in `r`.

We now return to the topic of outcome status signals. Typically the interface box of a caller translates an outcome signal that it receives into some status indicator observable by the caller. In the PSTN, for instance, `unavail` stimulates a busy tone, `unknown` stimulates an error tone, and `avail` establishes the voice channel all the way from the network to the device.

If there were no features, an outgoing call placed by an interface box would have exactly one outcome. In the presence of features the situation quickly becomes more complicated. Imagine, for example, a Call Forwarding on No Answer (CFNA) feature that is triggered and forwards to a telephone that is busy. First, the caller's interface box receives `avail` and hears ringback.[3] Then the CFNA feature is triggered by a timeout, ends its current outgoing call, and places another outgoing call to the forwarding number. The outcome of this call is `unavail`, which stimulates a busy tone when it reaches the interface box of the caller.

Examination of many features is absolutely conclusive: a calling interface box can receive any sequence whatsoever of outcome signals. Its behavior toward the caller is determined simply by the most recent one.

This means that outcome signals are idempotent. It also means that we need a fourth outcome signal `none`, signifying "no outcome yet." The need for this will become clear from the SCCC example below.

If a feature manipulates calls in any nontrivial way, it is difficult for the feature programmer to handle outcome signals correctly and consistently. For this reason, we have built correct outcome processing into Boxtalk.

Consider a box with an incoming call and an outgoing call. The basic rule of outcome processing is that whenever the two calls are signal-linked, the outcome most recently sent to the incoming call should be the most recent outcome of the outgoing call.

If the two calls become signal-linked as soon as the outgoing call is placed, and stay signal-linked until they are torn down, then signal linkage alone is sufficient to enforce the rule. Whenever an outcome arrives from the outgoing call, it will automatically be forwarded to the incoming call.

To handle more complex cases, the implementation must store the most recent outcome of each active outgoing call (the outcome is initialized to `none`). Whenever an incoming call becomes newly signal-linked to an outgoing call, the implementation automatically sends the most recent outcome to the incoming call.

For example, suppose that the SCCC program is in the `transparent` state, and that the outcome of `e` is `unavail`. The caller connected to this box through `r` is hearing busy tone, and sends a `disconnect` signal so that he can try another number. He expects the busy tone to cease immediately, even though there may be a delay in connecting to the resource. This favorable behavior will be achieved, because as soon as `r` becomes signal-linked to `v`, the box will send `r` the current outcome of `v`, which is `none`. The `none` signal will silence the busy tone, and will later be followed by `avail` from the resource.

If the target of `e` is unavailable, it is very likely that downstream calls will be torn down, and this box will make a transition to the `abandoned` state. Note that this transition will not cause any outcome signal to be sent automatically—they are sent on the *acquisition* of a signal linkage, not on the *removal* of one. This

[3] Ringback and alerting tones are phenomena of media-level signaling rather than call-level signaling, so they are not discussed in this paper.

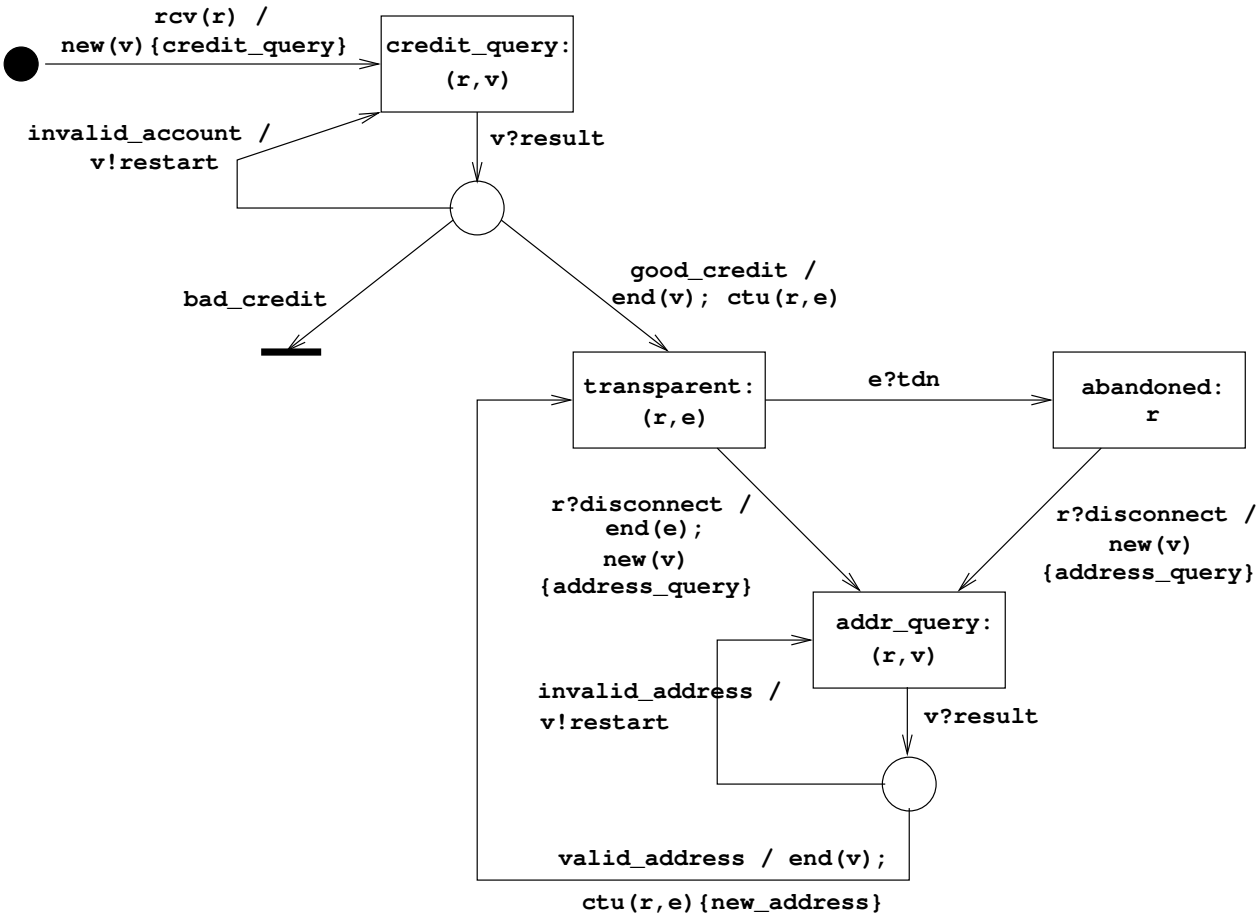

Figure 3: A Sequenced Credit-Card Calling feature box.

is appropriate because the caller should continue to hear the busy tone until he sends a `disconnect` signal or hangs up.

Outcome handling is actually somewhat more complex than this, simply because many graphs have a more complex shape and history than a monodirectional pipeline. The principle in all cases is the same, however, so the simplest case should be sufficient to convey the basic idea.

4 Formal semantics and verification

4.1 Syntax

We begin with a very brief overview of the language syntax. This overview omits many details and irrelevant aspects, including various shorthands definable in terms of the structures presented here.

Syntactically, a program is a connected graph whose nodes represent control states. They are partitioned into four classes, and subject to the following restrictions:

- There is exactly one *initial state,* having no in-transitions.
- There is any number of *transient states.* A transient state has at least one in-transition and one out-transition.
- There is any number of *stable states.* A stable state has at least one in-transition.
- There is any number of *termination states.* A termination state has at least one in-transition and no out-transitions.

The programs attached to this control structure are made up of three kinds of program element:

- An *unconditional statement* is a statement that is always enabled (provided that its precondition is satisfied, which is the subject of Section 5.2), such as `new(c)`, `ctu(c0,c)`, `end(c)`, `c!status`, and

assignments.

- A *conditional statement* is a statement that is only enabled when a particular signal is present in an input queue, such as `rcv(c)` and `c?status`.
- A *condition* is simply a predicate on the state of the box.

A transient state may be labeled with an *unconditional program*, which is a sequence of unconditional statements. An out-transition of a transient state is labeled with a condition followed optionally by an unconditional program. The transition is enabled if the condition is true.

Each stable state must be labeled with the variables of all the calls active in that state. An out-transition of an initial or stable state is labeled with a conditional statement followed optionally by an unconditional program. The transition is enabled if the conditional statement is enabled.

4.2 Semantics

We now present a very brief overview of the language semantics. It is also described in terms of a finite-state machine, with control states that are very similar to the syntactic states. The only difference concerns termination and final states. Regardless of whether the syntax uses one, zero, or many termination states, the semantics has exactly one termination state. The semantics also has a final state, reached from the termination state by an implicit transition. Note that a stable state may lack explicit out-transitions because all of its out-transitions are implicit, including transitions to the termination state on `teardown` signals (Section 3.3).

The actions of the box can be partitioned in two ways, resulting in four categories. An action might be triggered by entrance to a state, or it might be an out-transition of a state. An action might be explicit, based on something written in the program, or it might be implicit, a consequence of the program semantics.

If a transient state is labeled with an unconditional program, there is an explicit action on entrance to that state. The unconditional program is executed.

There may be an implicit action on entrance to a stable state. If there are new signal linkages in the state, outcome signals will be sent automatically.

There may be an implicit action on entrance to a termination state. If there are active calls, they are all ended.

Initial, stable, and termination states are *responsive* states, meaning that the box is paying attention to its input queues. Transient states are not responsive, meaning that the box is not paying attention to its input queues. The significance of responsiveness is discussed further in Section 5.3.

All out-transitions of transient states are explicit. In a transient state, if at least one out-transition is enabled, an enabled out-transition is chosen nondeterministically and executed.

Responsive states have implicit transitions as well as the explicit transitions given in the syntax of the program. Like explicit transitions, all implicit transitions are triggered by the existence of a signal in an input queue of the box. There are two reasons for the existence of an implicit transition:

- The implicit transition exists simply so that the programmer will not have to write it explicitly. For example, every responsive state with no explicit transition triggered by a `rcv` statement has an implicit out-transition that replies to a `setup` as the target interface box in Figure 2 does, and returns to the same state.
- The implicit transition is part of the asynchronous post-processing of a `new, ctu,` or `end` statement. For example, all transitions triggered by `upack` and `downack` are implicit transitions.

In a responsive state, if at least one out-transition is enabled, an enabled out-transition is chosen nondeterministically and executed. If no out-transition is enabled, execution halts until one is enabled by the receipt of a new signal.

4.3 Verification

The DFC protocol has been specified in Promela and checked using the Spin model checker [10]. Assuming that all feature boxes are input-enabled (see Section 5.3) and obey the protocol, there will be no deadlock, lost signals, or extraneous signals.

To show that the semantics of Boxtalk is sound, we should verify that all programs satisfy the safety properties of the DFC protocol, and that various invariants on the box state (some of which have been mentioned in Section 3) are preserved. This section describes a plan for performing that verification.

The first step is to deal with implicit actions. If we introduce some pseudo-statements such as `c?upack` and `c?downack`, which cannot actually be used in a Boxtalk program, then all implicit actions can be written explicitly. In other words, a Boxtalk program could be preprocessed to produce an intermediate program with pseudo-statements and without implicit actions of any kind. It is programs in this intermediate form that we shall verify.

The state of a box obviously includes the values of its local variables, sets such as `active` and `portAllocated`, functions such as the one that maps

calls in `portAllocated` to their ports, etc. We have already partially specified the statements and pseudo-statements as operations on this state. We have also specified some of the correctness invariants on this state, all in Alloy [11].

The state of a box could be extended to include the input queue of each port, the protocol state of each port, and the most-recently-sent signals in the output queue of each port. This extension has two purposes: (1) It enables us to specify the signaling behavior of a statement in terms of queue modifications, so that statements and pseudo-statements can be specified completely in terms of operations on the state. (2) It enables us to express protocol compliance in terms of preconditions and postconditions on operations.

We hope that, having extended the box state and completed the specification of operations in this way, we will be able to verify invariant preservation and protocol compliance. The assumed preconditions of each operation will be derived from the context in which it can occur, and from syntactic analysis (see Section 5.2). We also hope to automate some of this verification using the Alloy constraint analyzer.

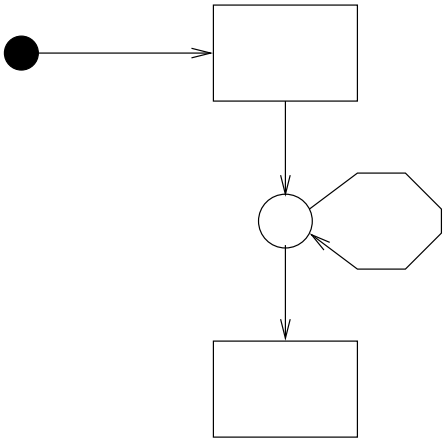

Figure 4: An illegal program graph.

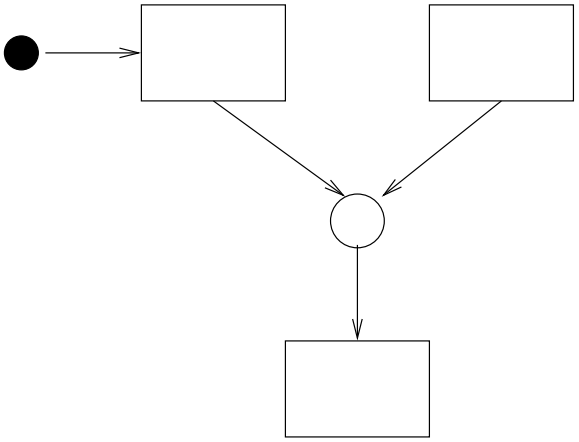

Figure 5: A legal program graph.

5 Necessary properties

5.1 A graph constraint

Both the properties discussed in this section require an additional syntactic restriction on Boxtalk programs. In the graph of a program, each path between two responsive states must be cycle-free.

For example, the graph shown in Figure 4 would not be legal as the basis of a Boxtalk program, because there is a path between the two stable states with a cycle in it.

Note that this restriction does not prevent convergence on transient states. For example, the graph shown in Figure 5 is legal.

5.2 No runtime errors

It is especially important to prevent runtime errors in component architectures. There are more independent programs running cooperatively, and there is less control over where they came from. In telecommunications, one of the major motivations for a component architecture is to allow customer programmability.

As described in Section 3.2, all the Boxtalk statements that operate on calls have preconditions based on whether relevant calls are active. As described in Section 4.3, we intend to prove that if the precondition of a statement is guaranteed at the time that the statement is executed, then the statement will execute successfully, without runtime errors.

Because of the constraint in Section 5.1, there is a finite number of paths that join two responsive states. Each path corresponds to a known sequence of explicit statements. For example, the path from the `credit_query` state of SCCC to the `transparent` state corresponds to the sequence `v?result; end(v); ctu(r,e)`.

Consequently, there is an analysis algorithm that checks the preconditions of all call-related statements. It simply traverses each path between responsive states, keeping track of the set of active calls, and checking preconditions as it goes. For each path traversal, the active set is initialized to the set in the state label at the beginning of the path (the initial

state is assumed to be labeled with the empty set). Once path traversal is complete, the active set must match the state label at the end of the path (unless the path ends at the termination state).

For example, traversal of the path mentioned above would compute active sets as follows:

```
credit_query STATE
                    { r, v }
v?result
                    { r, v }
end(v)
                    { r }
ctu(r,e)
                    { r, e }
transparent STATE
```

Clearly the preconditions of all the statements on the path are satisfied, and the resulting active set matches the label of the `transparent` state.

This analysis need not take implicit actions into account. Each implicit action can be shown to be valid in the state in which it occurs. The only implicit action that changes the set of active calls is the implicit action on entrance to the termination state. By definition, it ends all active calls, and precedes entrance to a state in which there are no active calls.

This algorithm shows that the labels of stable states are redundant (only insofar as they show the active call set—the signal linkages are not redundant). Since every stable state is reachable from the initial state, the algorithm can be used to compute its label. The label is required in Boxtalk simply because it is a valuable form of redundancy.

5.3 Boxes are input-enabled

A box is *input-enabled* if it is guaranteed to read every signal in every input queue in a timely fashion. Input-enabling is extremely important in a component architecture because it prevents deadlock and makes components responsive to their inputs. For example, even if a box's true function is hopelessly compromised by errors or resource failures, it should respond to a `teardown` from its subscriber.

All Boxtalk programs are input-enabled, provided that their implementation makes fair nondeterministic choices among enabled transitions, and provided that they pass one additional check (see below).

The argument concerning input-enabling has two parts: (1) A program is input-enabled in every responsive state. (2) On leaving a responsive state, a program always returns to a responsive state within a bounded interval of time.

The argument that a responsive state is input-enabled is simple: because of all the implicit features of Boxtalk, in every responsive state, there is an out-transition triggered by every possible signal that can be in every possible input queue. For example, although it has not been mentioned previously, consider a status signal `s` in the input queue of call `c`, when the program is in a stable state. If `c` is not currently signal-linked to any other call, and if there is no explicit transition on `c?s`, then an implicit transition reads `s` and throws it away.

The argument that a box returns promptly to a responsive state begins with the restriction that paths between responsive states are acyclic and therefore finite. The execution time of a path will be bounded if the execution time of each statement is bounded, and if there is no possibility of execution blockage somewhere on the path.

Boxtalk statements are restricted to those with bounded execution times. All call-manipulating statements have been designed to have this property. Data-manipulating statements are also bounded, because they do not include loops. (There are some restricted features for set comprehension.)

Once an out-transition from a responsive state has been triggered, the only execution scenario that might halt path execution before it reaches another responsive state is that execution has reached a transient state, and no out-transition from it has a true condition. To prevent this possibility, it is necessary to prove that, for each transient state in a Boxtalk program, the disjunction of the conditions on the out-transitions is *true*.

Thus the only loops in a Boxtalk program are loops that pass through responsive states. We have found this to be an appropriate constraint on DFC feature boxes, although it might not apply to other component architectures or other application domains.

6 Conclusions

There are already some basic tools for facilitating the programming of DFC feature boxes [3]. The forthcoming implementation of Boxtalk will provide the next generation of feature-creation tools.

We have shown that it is a significant challenge to program coordinating components. The programmer must manage asynchronous protocols. Proper coordination requires both piecewise and end-to-end reasoning. Certain component properties must be guaranteed.

At the same time, a component architecture gives us a foundation for deciding which component be-

haviors are equivalent. Behavioral equivalence is a tool we can apply to choose program abstractions and simplify programming.

It seems likely that at least some aspects of our call abstraction are applicable outside DFC. Any component architecture in which components interact asynchronously, for example [5, 6], may have some of the same problems, and be subject to some of the same solutions. And any component architecture whatsoever has a critical need for a way to certify the good behavior of components, regardless of their provenance.

References

[1] Greg Bond, Eric Cheung, Andrew Forrest, Michael Jackson, Hal Purdy, Chris Ramming, and Pamela Zave. DFC as the basis for ECLIPSE, an IP communications software platform. In *Proceedings of the IP Telecom Services Workshop 2000*, pages 19-26. Atlanta, Georgia, September 2000.

[2] Gregory W. Bond, Eric Cheung, K. Hal Purdy, J. Christopher Ramming, and Pamela Zave. An open architecture for next-generation telecommunication services. Submitted for publication.

[3] Gregory W. Bond, Franjo Ivančić, Nils Klarlund, and Richard Trefler. ECLIPSE feature logic analysis. In *Proceedings of the Second IP Telephony Workshop,* pages 49-56. Columbia University, New York, New York, April 2001.

[4] L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems.* IOS Press, Amsterdam, 1994.

[5] Manfred Broy. Compositional refinement of interactive systems. *Journal of the ACM* IVIV(6):850-891, November 1987.

[6] Manfred Broy. Functional specification of time-sensitive communicating systems. *ACM Transactions on Software Engineering and Methodology* II(1):1-46, January 1983.

[7] M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI.*, IOS Press, Amsterdam, 2000.

[8] K. E. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunications Systems III.*, IOS Press, Amsterdam, 1995.

[9] P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interactions in Telecommunication Networks IV.* IOS Press, Amsterdam, 1997.

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

[11] Daniel Jackson, Ian Schechter and Ilya Shlyakhter. Alcoa: the Alloy Constraint Analyzer. In *Proceedings of the International Conference on Software Engineering,* Limerick, Ireland, June 2000.

[12] 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.

[13] Michael Jackson and Pamela Zave. The DFC Manual. AT&T Research Technical Report, August 2001. Available at `http://www.research.att.com/info/pamela`.

[14] K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V.* IOS Press, Amsterdam, 1998.

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

[16] Pamela Zave. Address translation in telecommunication features. Submitted for publication.

[17] Pamela Zave. An architecture for three challenging features. In *Proceedings of the Second IP Telephony Workshop,* pages 176-187. Columbia University, New York, New York, April 2001.

[18] Pamela Zave. An experiment in feature engineering. In *Essays by the Members of the IFIP Working Group on Programming Methodology,* Springer-Verlag, to appear.

[19] Pamela Zave. Formal description of telecommunication services in Promela and Z. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design (Proceedings of the Nineteenth International NATO Summer School)*, pages 395-420. IOS Press, 1999.

[20] Pamela Zave and Michael Jackson. New feature interactions in mobile and multimedia telecommunication services. In M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI,* pages 51-66. IOS Press, 2000.