A Distributed Alternative to Finite-State-Machine Specifications

PAMELA ZAVE
AT&T Bell Laboratories

A specification technique, formally equivalent to finite-state machines, is offered as an alternative because it is inherently distributed and more comprehensible. When applied to modules whose complexity is dominated by control, the technique guides the analyst to an effective decomposition of complexity, encourages well-structured error handling, and offers an opportunity for parallel computation. When applied to distributed protocols, the technique provides a unique perspective and facilitates automatic detection of some classes of error. These applications are illustrated by a controller for a distributed telephone system and the full-duplex alternating-bit protocol for data communication. Several schemes are presented for executing the resulting specifications.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages, tools*

General Terms: None

Additional Key Words and Phrases: Finite-state machines, executable specifications, distributed decomposition, protocols, switching systems, Jackson System Development method

1. INTRODUCTION

An "operational specification" of a software system is an executable representation whose structure has been chosen to reflect the problem to be solved by the system rather than the resource and performance constraints that might shape an implementation [18]. In other words, it is designed for human comprehension rather than machine efficiency. Operational specifications are validated in the course of a dialogue between the systems analyst and the customer; tests, demonstrations, and static reviews furnish the information that is used to decide whether the specified system meets the customer's needs. The structure of the operational specification may be adopted as the structure of the implementation, if its performance level and resource consumption are acceptable, or it may be transformed into a functionally equivalent structure whose resource/performance trade-offs are more appropriate.

Finite-state machines are widely used as operational specifications of both entire systems and of particular modules within them. They provide a formal representation of how an input message determines a set of output messages, on

Author's address: AT&T Bell Laboratories, Murray Hill, NJ 07974.

the basis of limited information about the history of previous inputs. This is not a complete model of computation, but it is an extremely useful one (e.g., [3, 4, 9, 14]). It is easy to extend a finite-state machine informally so as to include data and data-processing attributes that are not representable within the machine itself (e.g., [2]). Finite-state machines are efficiently executable and amenable to static analysis.

The primary problem with finite-state machines as operational specifications is that they are difficult for human readers to comprehend. When the application is complex, the corresponding machine has a large number of states, few of which have any intrinsic meaning to people, because states usually represent time-dependent combinations of features or processes that are best understood independently. A finite-state machine is also centralized, although distributed solutions to common problems have now assumed considerable theoretical and practical interest.

The natural response to these difficulties is to decompose a single, complex finite-state machine into a set of interacting ones, but this is by no means easy. It is seldom clear how to decompose, and attempts at decomposition are frequently thwarted by persistent interdependencies or race conditions introduced by the decomposition. It is not even simple to define a communication mechanism among machines, as the semantics of SDL illustrate ([2], [13]).

This paper proposes an alternative to finite-state machines for operational specification. It is intrinsically distributed, and designed to provide more comprehensible formal specifications of solutions to complex problems. Section 2 presents the technique in terms of a substantial example: a controller module for a distributed telephone system. Here the technique guides the specifier to a good decomposition, encourages well-structured error handling, and offers an opportunity for parallel computation. *The casual reader need go no further.*

Section 3 shows how our technique can be used to specify distributed protocols, such as the full-duplex alternating-bit protocol for data communication, providing a unique perspective on them and facilitating automatic detection of some classes of error. Section 4 explores the formal relationship between our technique and finite-state machines, presents several analysis algorithms, and outlines a set of implementation schemes.

2. THE SPECIFICATION TECHNIQUE AND A BIG EXAMPLE

2.1. Description of the Example

A module of a computer system can be viewed as a process that receives a sequence of input messages, where the sequence imposes an ordering on inputs from diverse sources. There is a fixed, finite set of inputs, although some inputs may be associated with arbitrary data items. The process waits for each input and, upon receiving it, performs a set of actions such as updating internal storage and sending messages to other processes. These actions may be dependent on both the current input and on the history of previous inputs.

The above abstraction is most useful when the complexity of the module is dominated by control. A controller for an individual telephone in a distributed telephone system is an example of such a module. It communicates with a handset, a network capable of delivering addressed messages between controllers, timers, and auxiliary processes to set up calls and provide other features. Specifically:

(1) Plain Old Telephone Service (POTS): Telephone numbers are single digits. Once a voice connection has been established, it can be disconnected symmetrically by either party, after which either party must hang up before placing another call.

(2) Nonexistent Phones: If a nonworking number is dialed, the caller will hear a recorded message giving this information.

(3) Call Forwarding: Calls may be forwarded to another number. To set up a forwarding state, the user presses a call-forwarding button on the handset (a call-forwarding light begins to flash), picks up the receiver (a special call-forwarding-setup tone is heard), dials a number, and hangs up (the call-forwarding light shines steadily, indicating that call forwarding is in effect). Call forwarding can be cancelled at any time by pressing the call-forwarding button again (the light goes out).

(4) Automatic Callback: While listening to a busy signal, a caller may press an automatic-callback button on the handset. This causes both phones to be monitored, so that if both are free simultaneously the caller's handset will ring with a sound different from normal ringing. If the caller answers this ring, it is as if the number of the target phone had just been dialed. The feature is cancelled if 40 minutes elapse, if the button is pressed again at any time, or if the distinctive ringing is initiated (with any outcome). If this feature is activated, an automatic-callback light on the handset is lit.

Several other telephone features, such as multiple-digit and variable-length numbers, asymmetric disconnect, call add-on, billing, and request/release of talking circuits, have also been specified using our technique. This simpler version of the example has been chosen for purposes of exposition.

2.2 Sequence Diagrams

A set of legal sequences of inputs is specified graphically using a *sequence diagram* (see Figures 1 and 2). A *sequence diagram* is a tree whose root node is labeled with the name of the set of sequences and whose interior nodes are labeled with the names of sets of subsequences. The successors of a node define the sequences in its set in terms of sequence, alternative, and iteration primitives. Leaf nodes denote the inputs themselves. An input may have an *alias*, the name of which appears above the bar dividing the node (the significance of aliases will be explained in Section 2.4).

Except for aliases, sequence diagrams have been borrowed whole from the Jackson Structured Programming (JSP) and Jackson System Development (JSD) methods, which were the inspirations for much of this work (see [7] and [8]).

Figure 3 shows a fragment of a sequence diagram for the telephone controller. The reader may have noticed that sequence diagrams are equivalent to regular expressions (see Section 4.1), but are far more readable. One reason is the graphic display of expression structure; another is the naming of subsequences by means of labels on interior nodes (imagine Figure 3 with only the leaf nodes labeled).

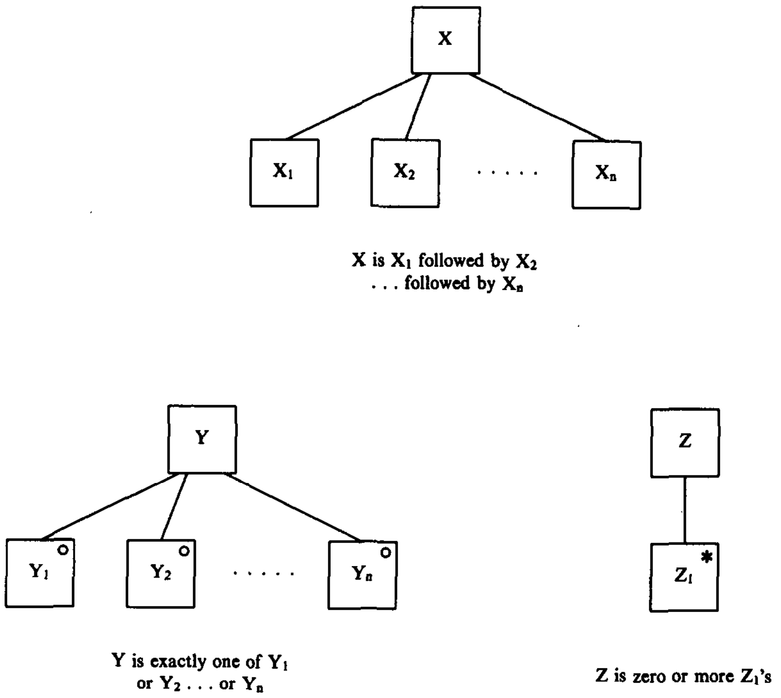

X
X_1
X_2
X_n
X is X_1 followed by X_2 . . . followed by X_n
Y
Y_1 o
Y_2 o
Y_n o
Y is exactly one of Y_1 or Y_2 . . . or Y_n
Z
Z_1 *
Z is zero or more Z_1's

Fig. 1. A graphic notation for defining sets of sequences.

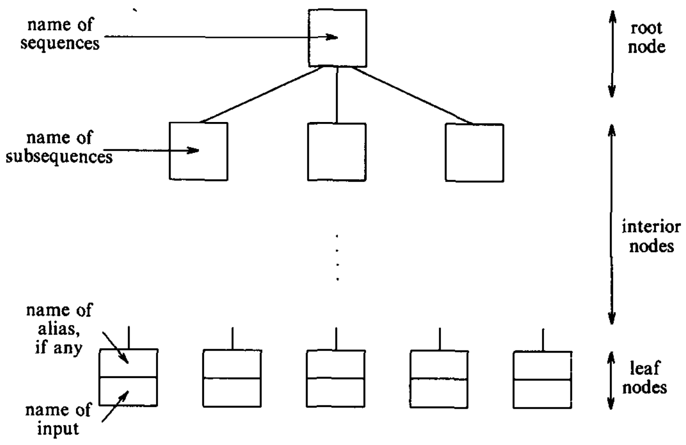


Fig. 2. The components of a sequence diagram.

Labeling also provides a convenient[1] "subroutine" facility: "finish-call," for instance, may be used elsewhere without definition because this diagram defines it.

[1] In some cases more than convenience is at stake. The use of labeled subexpressions reduces the total size of the specification, and this reduction could make the difference between feasibility and infeasibility.

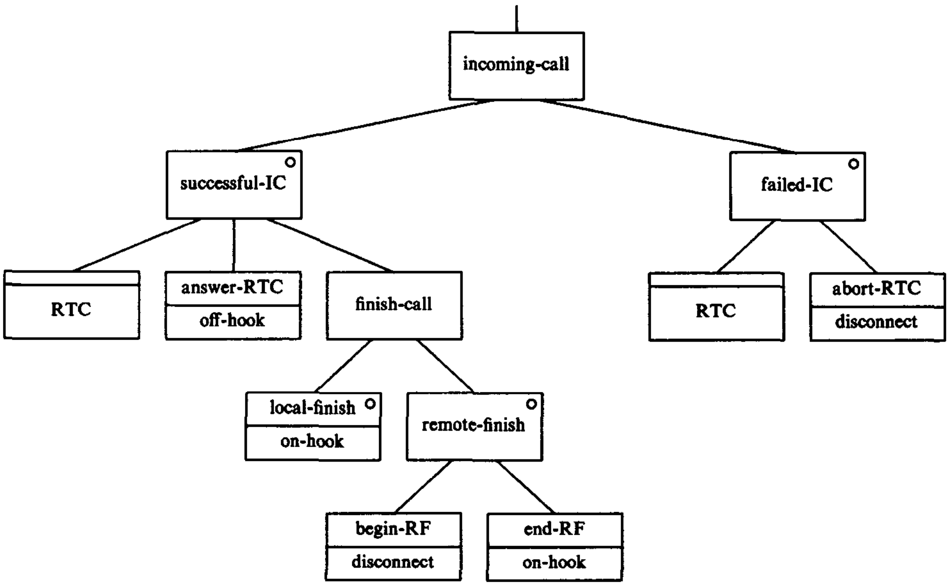


Fig. 3. A fragment of a sequence diagram. RTC (request to connect) is an input from another phone that is trying to call this one.

Another source of the readability of sequence diagrams is that they can (and should be) used to give a static description of the possible input sequences rather than the usual dynamic one. Figure 3, for example, describes a set of incoming calls, as we might do after their occurrence, reviewing the history of events. The fact that when an "RTC" (request to connect) arrives the controller does not yet know whether it is going to belong to a successful or failed call is of no concern from this "retrospective" viewpoint.

Figure 4 is another illustration of static description. An outgoing call is complex because it has many failure points (i.e., where the call can be aborted), but a successful call is a simple sequence of inputs. Figure 4 shows how each different cause of failure can be labeled and treated separately, without affecting the "successful-OC" sequence at all.

2.3 Views

A specification is made up of a set of *views* that describe the constraints on inputs to the module. Each view has a sequence diagram that specifies necessary constraints on a subset of the module's inputs.

The telephone controller is described in terms of four views, of which Figure 5 shows two. Since a controller corresponds to a telephone number, which can be reused, each one has many "lifetimes." The "lifetime-view" specifies mutual constraints on only three inputs (we say that only "install," "RTC", and "remove" are *relevant* to the lifetime view), that is, that installations and removals are paired, and the phone can be called only after it has been installed and before it has been removed.

In addition to its views, a module must have a *filter* that receives the input stream and passes each input on to all views to which it is relevant (Figure 6).

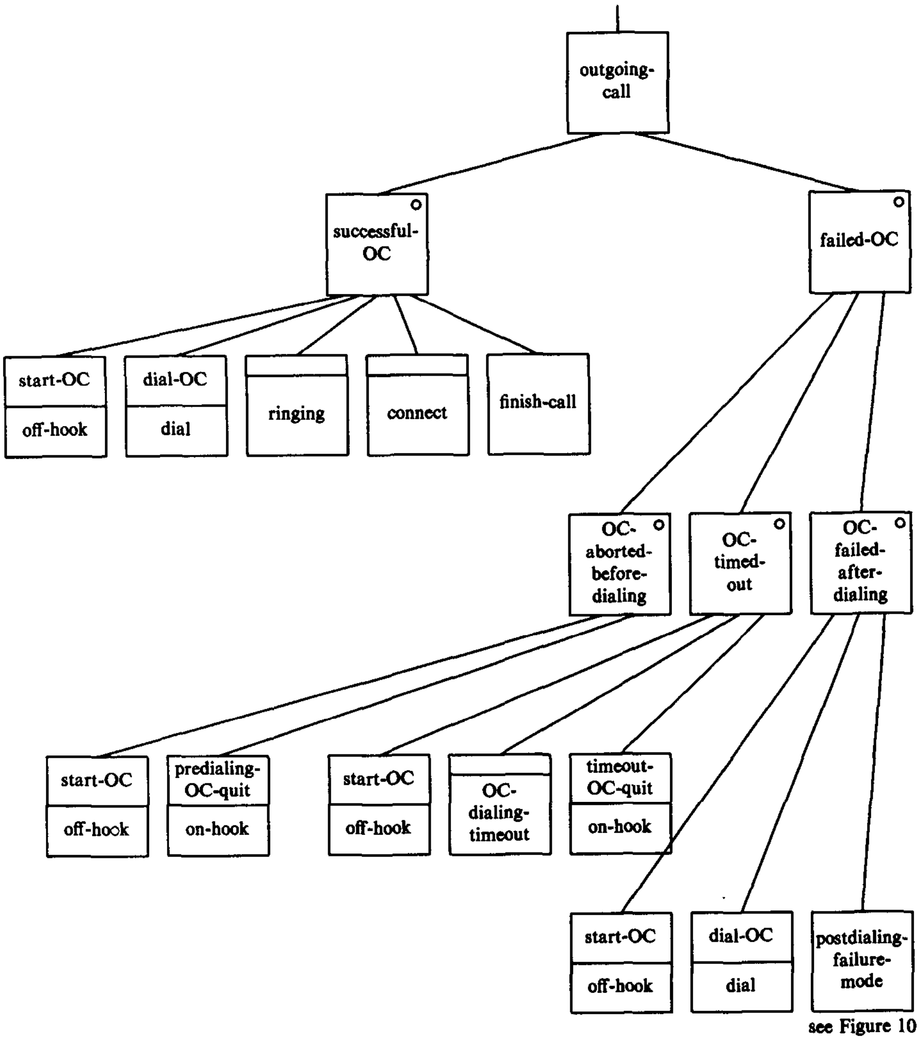


Fig. 4. A sequence diagram showing static case analysis. OC-dialing-timeout comes from a timer one minute after the handset goes off-hook.

An input is acceptable to the module if and only if it is currently acceptable to all views to which it is relevant (i.e., if and only if it satisfies all the sequence constraints to which it is subject).

If an input is not acceptable then the filter does not pass it on to the views, but may perform some *error action*. By knowing which set of views rejected the input, the filter knows the reason for rejection, and therefore which action to take. If an RTC is rejected by the lifetime view, for instance, the phone is inoperative; the filter will forward the RTC to a special pseudophone permanently connected to a tape loop with a recorded message. If the RTC is rejected by the POTS view, the phone is busy and a "busy" is sent back to the origin of the RTC.

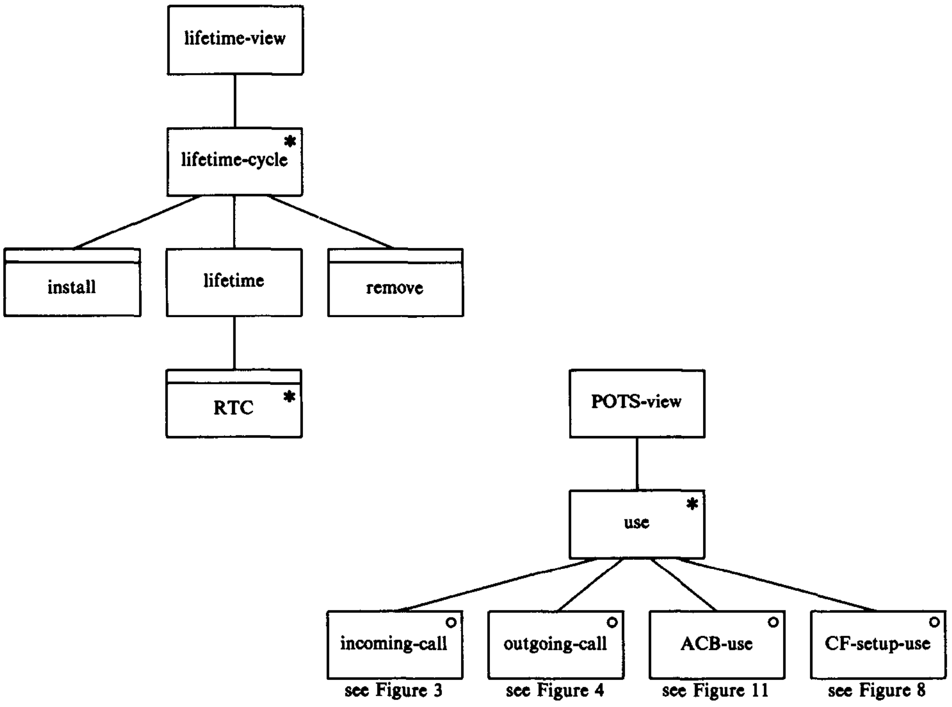


Fig. 5. Two views of the telephone controller. ACB is "automatic callback," CF is "call forwarding."

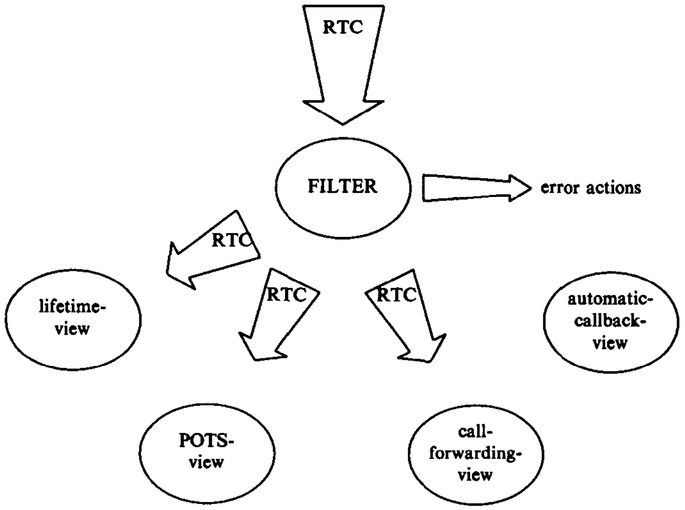


Fig. 6. Submodules of the telephone controller. An RTC is relevant to all views but the automatic-callback view.

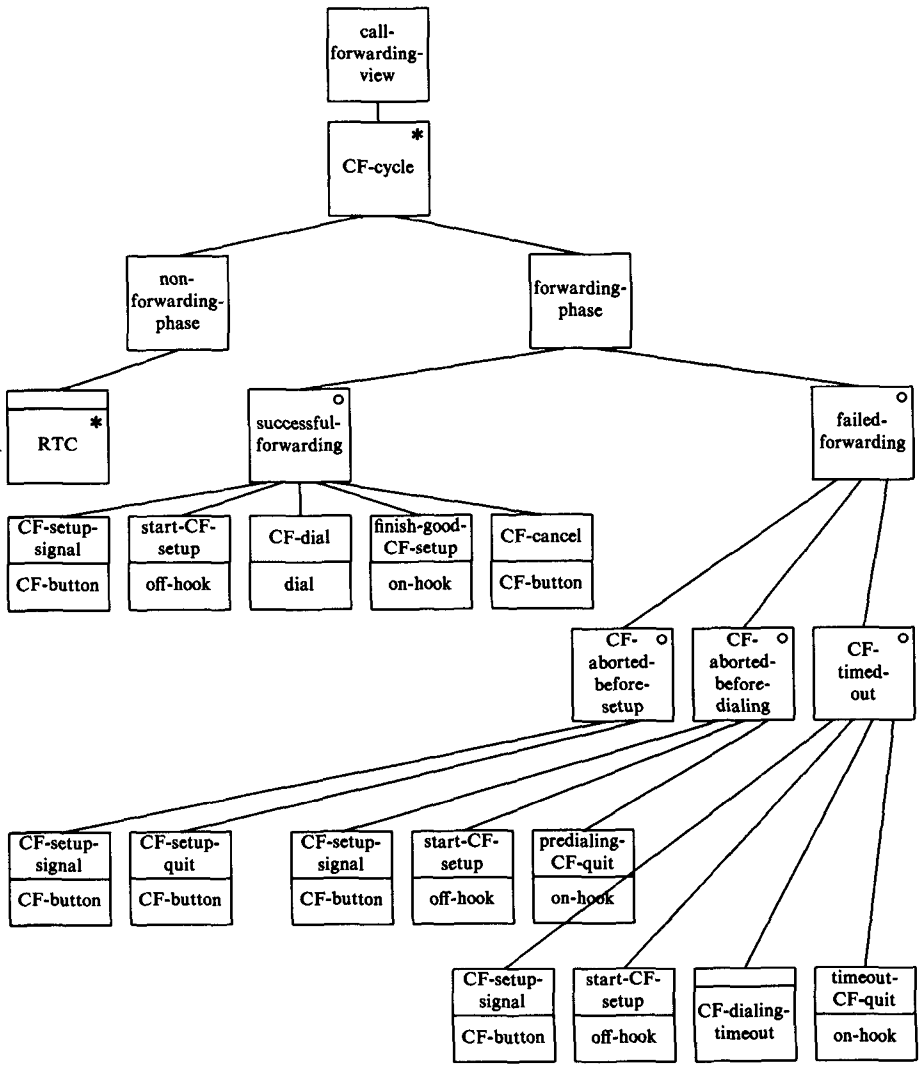


Fig. 7. The call-forwarding view. CF-dialing-timeout comes from a timer one minute after the handset goes off-hook.

Figure 7 shows the call-forwarding view, and Figure 8 the input sequence relevant to the POTS view for setting up call forwarding. The two have many common inputs. Thus, if "CF-button" is received when the call-forwarding view is in the nonforwarding phase (it is acceptable to the call-forwarding view), but when the phone is busy (it is not acceptable to the POTS view), it will not be accepted. In this case there is no error action, and the input is merely ignored.

The RTC is relevant to the call-forwarding view, although it is only acceptable in the nonforwarding phase. If an RTC is rejected here it means that calls are being forwarded, and the appropriate error action (regardless of whether or not

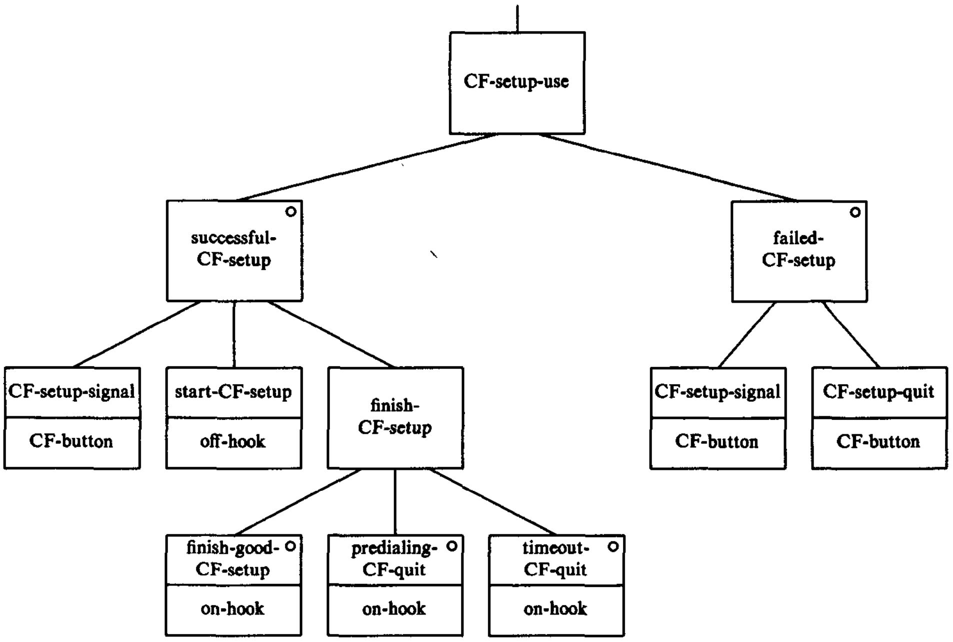


Fig. 8. A part of the POTS view.

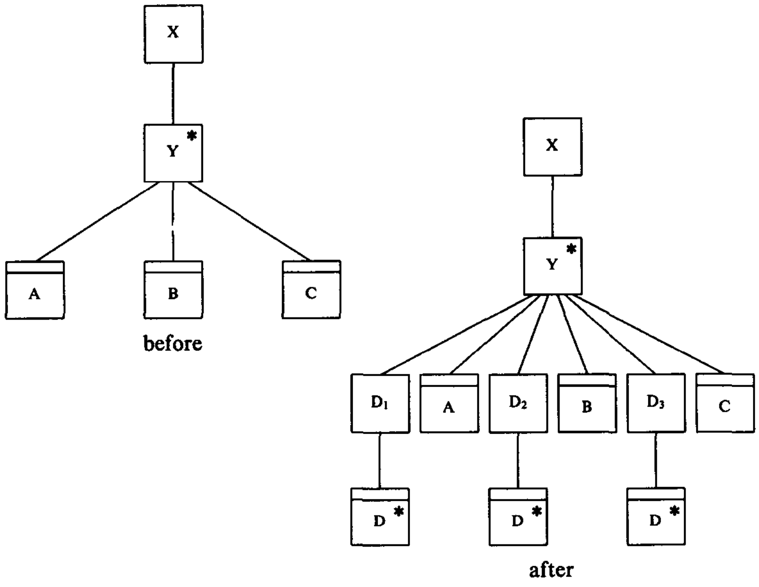


Fig. 9. The ugly consequences of introducing an irrelevant input to a view.

RTC is currently acceptable to the POTS view) is to send the RTC to the forwarding number.

It is an important feature of this specification technique that the decomposition into views is never arbitrary. In most cases the input constraints expressed in two views would be difficult (and unreadable) to express in a single view. Consider, for instance, the relatively simple matter of introducing to a view with only A, B, and C as inputs, an input D that is not really relevant to it (see Figure 9). Since D has no constraints with respect to the other inputs (this being what relevance is all about), it must appear everywhere. Thus, when an analyst tries to express the wrong set of concepts in a particular view, it makes such a mess of the sequence diagrams that he is quickly dissuaded.

One other structuring principle is at work here. No difficulties of the kind just mentioned would arise if the lifetime view were merged with the POTS view, but then the filter could not distinguish between a busy phone and an inoperative one.

2.4 Aliases and Actions

An external input can have any number of internal aliases. An alias corresponds to an interpretation or meaning of an input. For instance, the input "on-hook" has many meanings and aliases: it can represent finishing a conversation, quitting an outgoing call before dialing, completing a call-forwarding setup, and so on.

In the sequence diagrams, leaf nodes show inputs in their lower sections and aliases in their upper sections. Some inputs, such as RTC, are so unambiguous that they have no aliases.

All the previous rules about relevance to views and acceptance by the filter actually refer to aliases rather than inputs (error actions, of course, still refer to inputs). The filter interprets a new input as whichever of its aliases is currently acceptable, and distributes that *alias* to the relevant views. A specification can always be checked (see Section 4.2) to make sure that alias interpretation is unambiguous.

Each time a view receives an alias it performs a (possibly null) set of *actions*. Another way to understand the notion that an alias corresponds to a meaning is the rule that, within a view, an alias must be associated with a unique action set. This guarantees that the view specification will always determine which actions are to be performed. For instance, when the POTS view accepts an RTC (Figure 3), it performs the actions of sending a signal to the handset to begin power ringing, sending the "ringing" signal back to the source phone, and storing the telephone number that is an attribute of the RTC. It does so regardless of whether the RTC will eventually become part of the "successful-IC" or "failed-IC" sequences.

It is always possible to obey this rule. If two instances of one alias in a view are found to require different actions, then two different aliases should be used. If the filter cannot then interpret the input unambiguously, then there is an intrinsic conceptual problem, because the module has no way to decide which of the two actions is appropriate.

Aliasing solves a nasty problem, which we will illustrate with the automatic-callback feature, although it occurs in call forwarding as well. Figures 10 and 11 show parts of the POTS view, while Figure 12 presents the automatic-callback view.

The automatic-callback button is used to initiate the feature. Since this can be done only when the caller is hearing the busy signal, the "ACB-button" input has mutual constraints with inputs concerning outgoing calls, and the ACB-

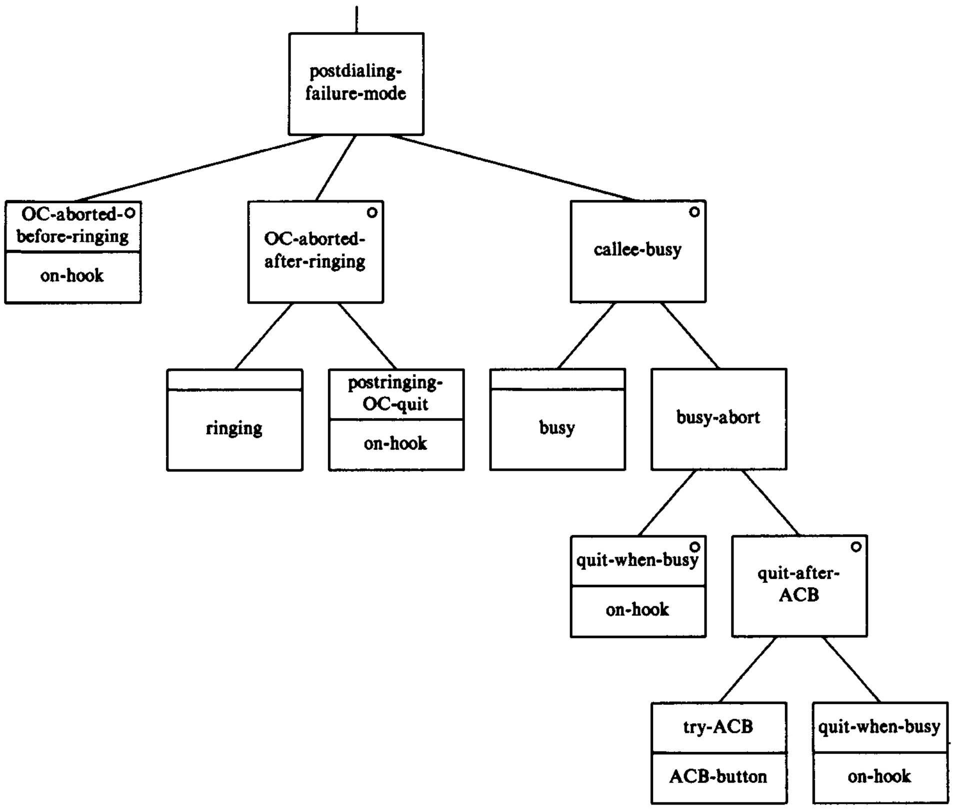


Fig. 10. A part of the POTS view.

button must be relevant to the POTS view. The problem is that the button is also used to cancel the feature,[2] but this can happen at *any* time with respect to the POTS view. Without aliasing we would have to sprinkle the POTS view with ACB-button inputs, showing that they could appear between any other two inputs. With aliasing, only the "try-ACB" alias of ACB-button is relevant to the POTS view, while both aliases are relevant to the automatic-callback view. Note that the sequence in "ACB-cycle" prevents any ambiguity in interpreting the ACB-button input.

When try-ACB is accepted the automatic-callback light is lit, a timer is initiated, and an independent ACB-function process for this phone is activated. The ACB-cycle may end because 40 minutes elapse or the feature is cancelled. In the meantime the function process will monitor both phones to see if they are idle. Our local phone is monitored by sending it "QRTC" (query RTC) inputs. These inputs are never accepted because they are relevant to no view, but the filter responds to the function process on the basis of whether or not it could

[2] We could have evaded the problem by using two buttons, but this requires more hardware and makes it easier for the user to make mistakes.

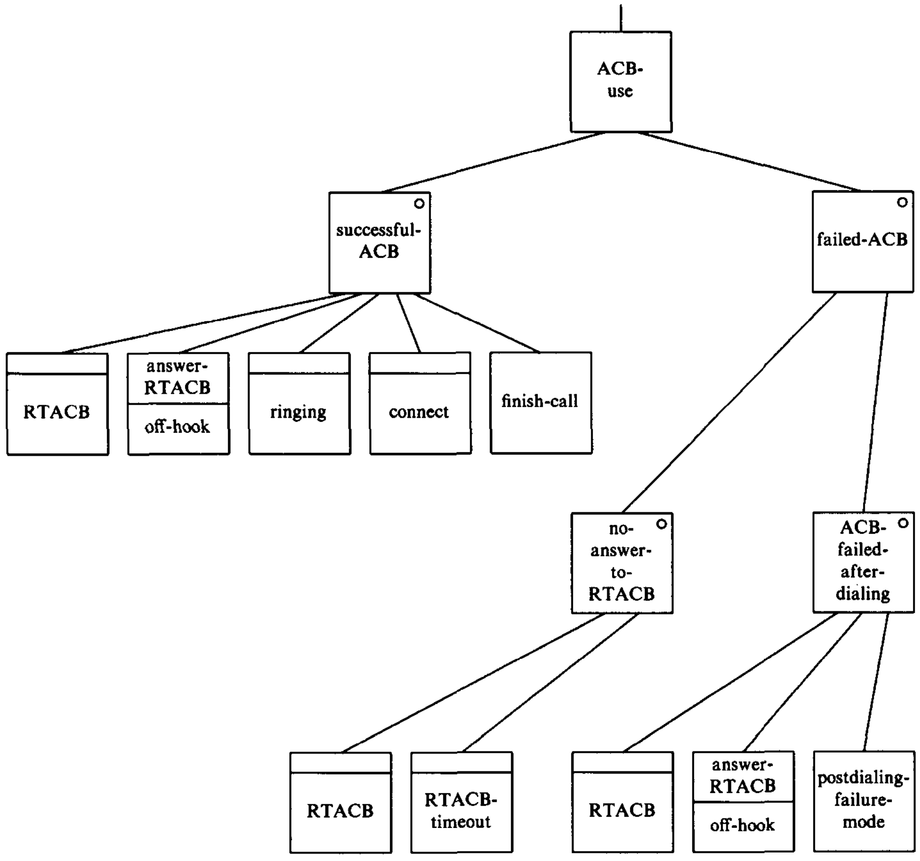


Fig. 11. A part of the POTS view. RTACB is "request to automatic callback." RTACB-timeout comes from a timer one minute after the RTACB.

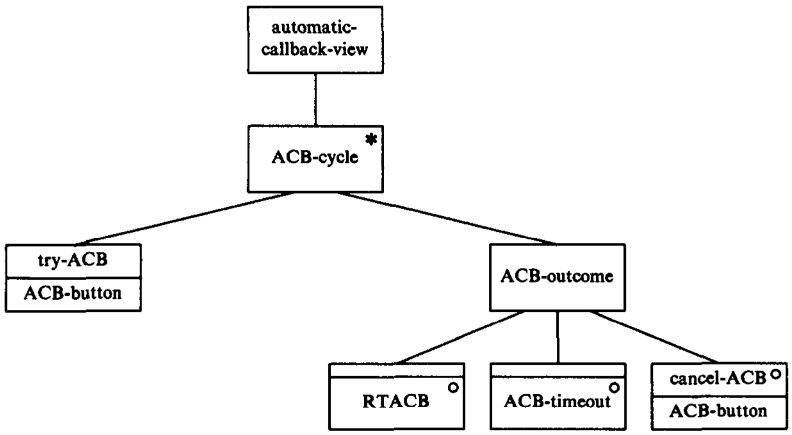


Fig. 12. The automatic-callback view. ACB-timeout comes from a timer 40 minutes after try-ACB.

have accepted an RTC. If the function process finds both phones idle, it sends an "RTACB" to the controller; if the phone is answered, a call is automatically initiated.

Note that when an alias is relevant to multiple views, its acceptance activates the action sets in the multiple views simultaneously. This provides an opportunity for parallel computation.

2.5 Data

Data arrives at the telephone controller in the form of data items associated with inputs. With this control scheme it is easy to handle the data in a distributed manner. The filter saves whichever of the data items it needs in local variables, and the views (which receive the data items as well as the interpreted inputs from the filter) do the same.

Occasionally there is a problem in recognizing which data items are needed, but the aliasing mechanism solves it. For instance, the filter sometimes needs to save dialed numbers (when they are for call forwarding) and sometimes not (when they are for ordinary outgoing calls). But the filter can identify the situation once it has interpreted the "dial" input as either its "CF-dial" or its "OC-dial" alias.

Finally, the filter may have to add new data items to the aliases before passing them on to the accepting views. Consider the situation in which a user dials a call, but the busy or ringing signal sent back in reply to the RTC is delayed so long in transit that the user hangs up and makes another call. When the reply finally arrives it might mistakenly be interpreted as referring to the later call. The same could happen with any other intercontroller signal.

We solve this problem by giving each call a unique identifier which must be associated with any intercontroller signal. The filter rejects and ignores any input whose identifier does not match that of the current call. Identifiers are chosen by the filter at the right moment (which it knows from alias interpretation) and passed on to views as data items associated with the crucial alias, so that both filter and views are synchronized with respect to identifiers. The same mechanism can also be used to get rid of outdated timer signals.

2.6 Summary of the Specification Technique

The appendix completes the example, including all details except those given in the sequence diagrams. Note that the set of inputs, the set of aliases associated with each input, and the relevance relation on aliases and views can be extracted automatically from the sequence diagrams.

Figure 13 summarizes the features of the specification technique. Those on the left can be formalized straightforwardly, and correspond to the role of the finite-state machine in a specification based on that technique. Those on the right have many different formal representations, and are best defined in a way that is compatible with the specification of the rest of the system; they correspond to the informal extensions commonly found in finite-state-machine specifications.

Our formal specification technique decomposes the complexity of the problem by guiding the analyst to an appropriate decomposition into views. The views constitute a distribution of both state information and data, and offer an opportunity for parallel computation. The filter provides a clean and effective

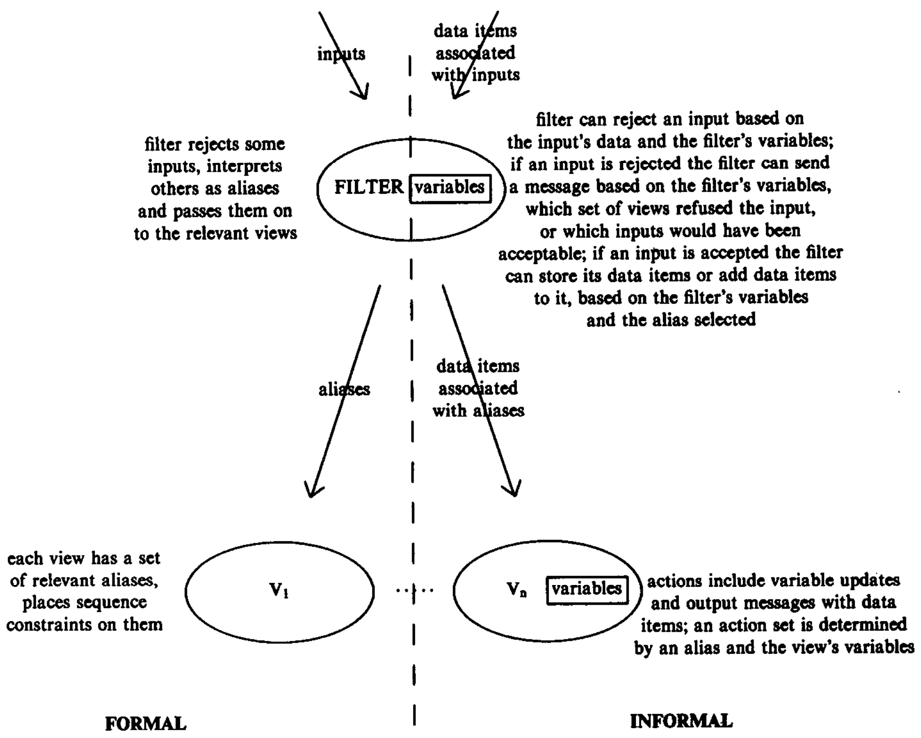

inputs
data items associated with inputs
filter rejects some inputs, interprets others as aliases and passes them on to the relevant views
FILTER
variables
filter can reject an input based on the input's data and the filter's variables; if an input is rejected the filter can send a message based on the filter's variables, which set of views refused the input, or which inputs would have been acceptable; if an input is accepted the filter can store its data items or add data items to it, based on the filter's variables and the alias selected
aliases
data items associated with aliases
each view has a set of relevant aliases, places sequence constraints on them
V1
Vn
variables
actions include variable updates and output messages with data items; an action set is determined by an alias and the view's variables
FORMAL
INFORMAL

Fig. 13. The formal and informal features of the specification technique.

way to handle subtle interdependencies among features (views), without demanding much centralized information. Explicit error handling is encouraged. Comprehensibility is provided by the decomposition into views, hierarchical graphic notation, and static description of sets of sequences.

On the informal side, distribution of the specification does not interfere with the various data manipulations and conditional actions that are needed—whatever the mechanism, there seems to be a rational and convenient place to incorporate it in this organization. Success here indicates that we are dealing with a robust approach to this type of problem.

In JSD [8], sequence diagrams are used to express constraints on inputs, filters are sometimes used to intercept erroneous inputs, and inputs are sometimes referred to by multiple sequence diagrams. Our specification technique is novel in the definition and use of aliases, the organization of multiple views under a common filter (this idea was developed jointly with John Cameron, and is also mentioned in [1]), the distributed-system application to be presented in Section 3, and the algorithms to be presented in Section 4.

3. A NEW LOOK AT PROTOCOLS

Specification and verification of distributed protocols is now a research topic of considerable importance. While axiomatic techniques (e.g., [5]) may offer the most abstract specifications of desired properties and the most concise proofs of them, operational specifications are more easily used and more easily associated

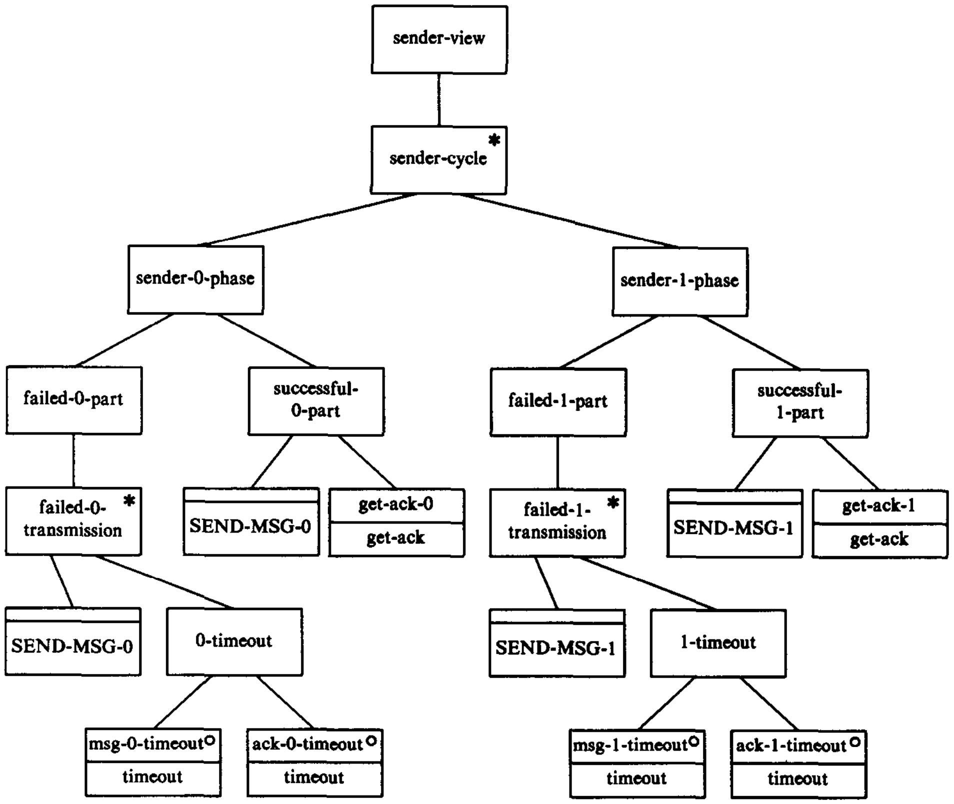


Fig. 14. The sender's view of the alternating-bit protocol.

with particular implementations. For example, a finite-state machine can be used straightforwardly as in [15] to represent and explore the global behavior of a protocol.

Our specification technique is also an alternative to finite-state machines in the specification of distributed protocols. In this context "inputs" are actually *events*, each view represents a distinct location or actor, and the events relevant to a view are those that are observable from the location (or by the actor) represented by the view. This will be illustrated using the full-duplex version of the alternating-bit protocol for data communication.

The protocol can be described in terms of four (again, coincidentally) views: the sender, the receiver, the data channel, and the acknowledgment channel. The sender's view is shown in Figure 14. Some of the events seen by the sender (and therefore relevant to this view) are *caused* or *produced* by the sender, while others are *accepted* or *consumed* by the sender. The former are written in capital letters while the latter are written in lower case; across the views, events and aliases are the same if they are spelled the same.

The sequence diagram in Figure 14 constrains the behavior of the sender with respect to primitive "sends" and "receives"; if local data manipulations were

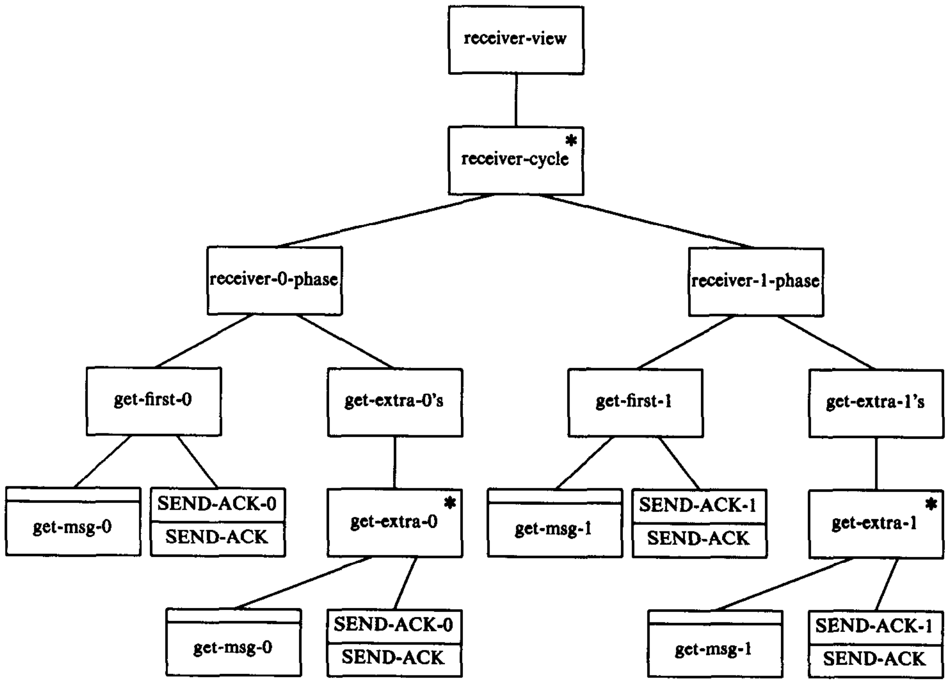


Fig. 15. The receiver's view of the alternating-bit protocol.

added as action sets associated with events or their aliases, it would serve as a complete program for the sender. As in Section 2, aliases and labels on interior nodes serve to structure and interpret the information given, so that the strategy behind the protocol is made clear.

Figures 15, 16, and 17 show the receiver, data channel, and acknowledgment channel views, respectively. In this system[3], each event is produced by one view and consumed by one view. An event can occur only when it is simultaneously acceptable to all views to which it is relevant. When the system is in a state such that several events are acceptable, one of the eligible events is chosen nondeterministically to occur (this models the effects of variable rates in the protocol's physical implementation, conflict resolution, hardware failure, etc.).

This representation can be analyzed algorithmically for common protocol errors. If any sequence of events leads the system to a state in which no event is acceptable to all its relevant views, for instance, the protocol deadlocks. This is equivalent to finite-state machine analysis of the protocol (see Section 4.2).

A more interesting aspect of the representation is the use of aliases to express the interpretations being put on events by the protocol, as "get-ack-0" and "get-ack-1" express whether the sender is interpreting an unlabeled "get-ack" as an acknowledgment to a message labeled 0 or 1, respectively. Since the protocol's

[3] A broadcast protocol, for instance, might have events that were produced by one view and consumed by many.

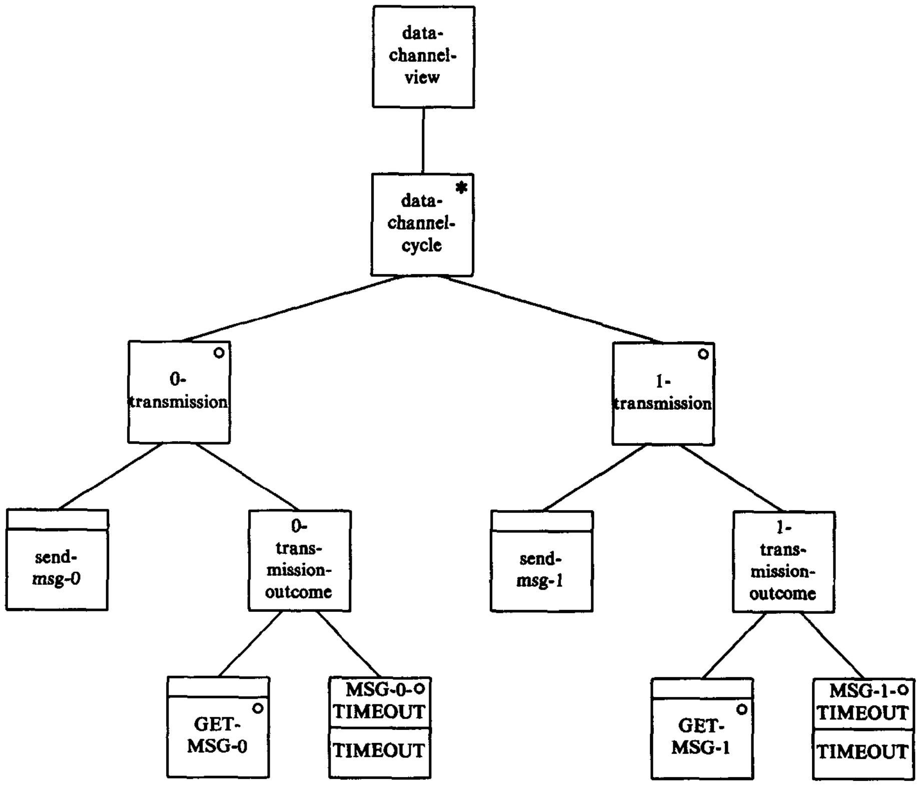


Fig. 16. The data channel's view.

implementation will have no mechanism for processing aliases (i.e., there will be no central filter), the aliases must not be defined in such a way that they affect relevance (i.e., it cannot be the case that one but not all aliases of an event are relevant to a view, but adherence to this rule is quickly determined by scanning the specification). Aliases are useful, however, because many errors will express themselves as conficts in alias interpretation—execution states in which an event is acceptable to all its relevant views, but *not under the same alias.* When there are no possible conflicts in alias interpretation, our confidence that the protocol's strategy is effective and complete can increase.

For instance, the protocol specified so far relies on the assumption that transmission times are bounded, so that a "timeout" never arrives unless a message has in fact been lost. Under this assumption we can model a "timeout" as the output that a channel produces when it has lost a message, as in Figures 16 and 17. If this assumption were relaxed, then timeouts could arrive while the messages to which they refer were still in transit. The only effect this would have on the sequence diagrams would be to remove timeouts from relevance to the channel views, so that "0-transmission" would be as in Figure 18 (analogous changes should be made throughout the channel views). But now the sequence of events

send-msg-0

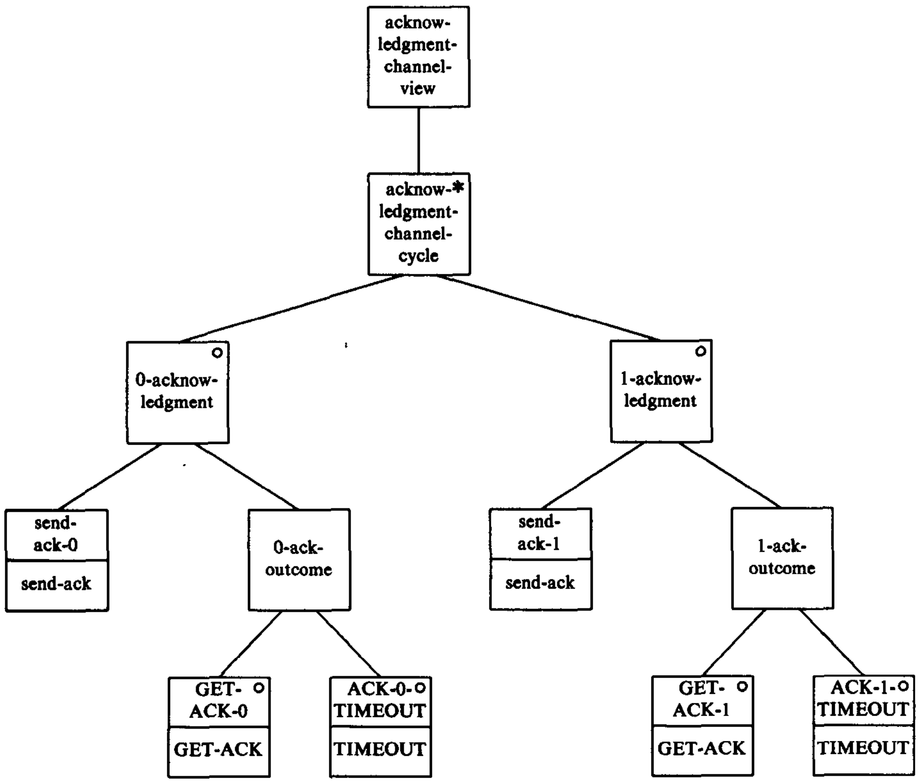


Fig. 17. The acknowledgments channel's view.

get-msg-0
send-ack-0/send-ack
timeout
send-msg-0
get-ack-0/get-ack
get-msg-0
send-ack-0/send-ack
send-msg-1

would bring the system to a state in which the sender is willing to accept get-ack-1/get-ack, but the acknowledgment channel can only accept get-ack-0/get-ack. If the get-ack event is accepted despite the conflict in alias interpretation, the event sequence

send-msg-0
get-msg-0
send-ack-0/send-ack
get-ack-0/get-ack

puts the system into a state in which the protocol is happy, but two consecutive messages have been lost.

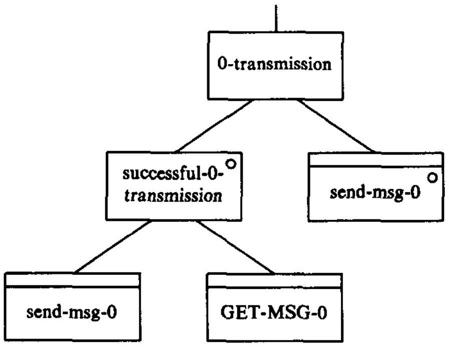


Fig. 18. Transmission without constraints on timeout.

In summary, the distributed nature of our specification technique can be exploited directly to specify distributed protocols. It provides a notation in which the protocol designer's *intentions* are part of the formal representation, and these intentions can be checked automatically for consistency with the actual operation of the protocol.

4. THEORETICAL RESULTS, DECISION PROCEDURES, AND IMPLEMENTATION SCHEMES

4.1 Equivalence with Finite-State Machines

Our specification technique is formally equivalent to finite-state machines, as is easily shown.

First of all, the language of aliases accepted by a specification is regular. Since sequence diagrams are simply hierarchical, graphic[4] forms of regular expression, each view specifies a regular language over the alphabet r of its *relevant* aliases. This will be called a *restricted view language.* To transform a restricted view language into an *extended view language*, expressing the constraints placed by this view on *all* aliases, the alphabet e of aliases *irrelevant* to this view must be taken into account. The extended view language is formed by applying a substitution transformation to the restricted view language that maps any symbol of r onto e^*re^*, and taking the union of the result and e^*. Since regular sets are closed under union and substitution [6], an extended view language is regular. The language of aliases accepted by a specification is the intersection of the extended view languages, which is also regular.

The language of inputs accepted by a specification can be obtained from the language of aliases accepted by application of a homomorphism [6] that maps each alias onto its input. Homomorphisms preserve regularity.

When a centralized module is being specified, an input is accepted by the module if and only if the sequence of inputs accepted so far, with the new input appended, is an initial subsequence of a member of the regular language we have

[4] For comparison, BNF is hierarchical but not graphic.

just defined. When a distributed protocol is being specified, the regular language is *generated* by the system, and captures the range of its global behavior just as an event expression (e.g., [11, 12]) does.

4.2 Decision Procedures

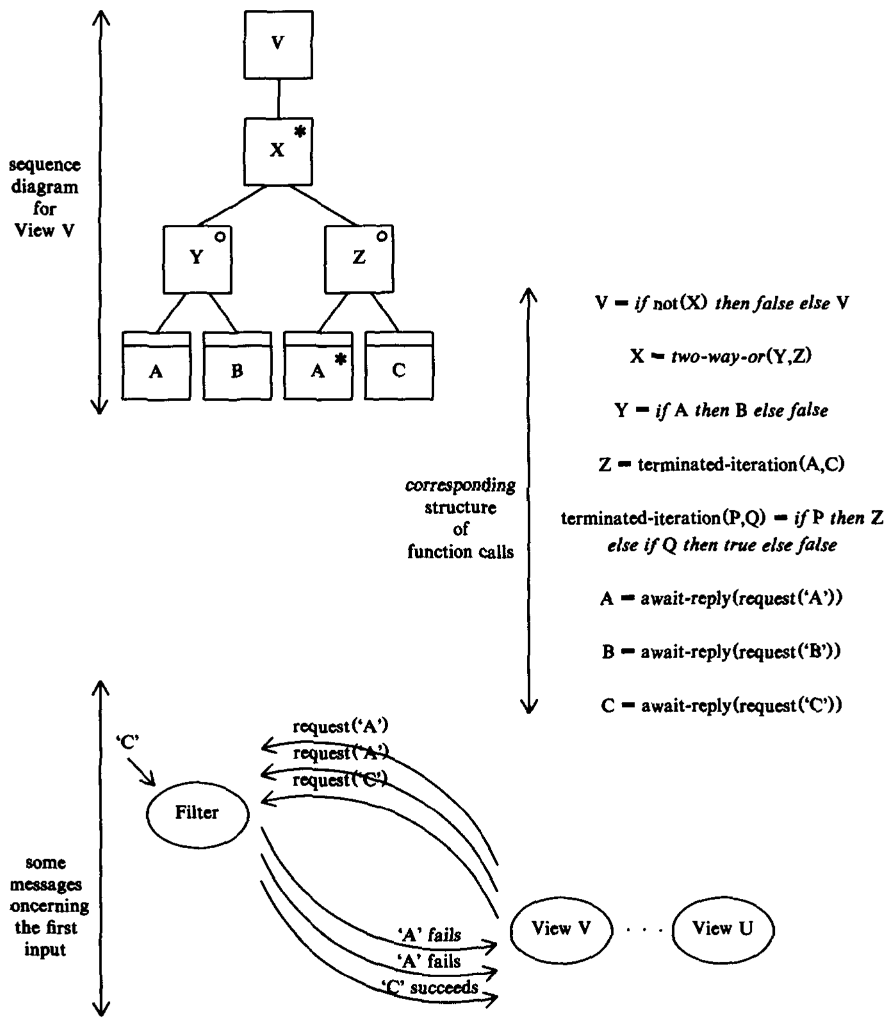
In a well-defined specification, interpretation of aliases is unambiguous and there is no possibility of deadlock (a state in which no inputs or events are acceptable). Either property can be tested by constructing a deterministic finite-state machine that recognizes the same regular language of aliases accepted by the specification (note that the regularity of this language does not depend on unambiguous alias interpretation). Interpretation of aliases is unambiguous if and only if there is no state of the machine in which two or more aliases of the same input cause transitions that can lead to recognizing states. There is no possibility of deadlock if and only if there is no state of the machine in which no alias causes a transition that can lead to a recognizing state.

When the specification technique is being used to describe a centralized module, unambiguous aliases imply that actions are well defined, and freedom from deadlock implies that the module cannot get into a state in which no inputs are legal. If the specification technique is being used to describe a distributed protocol, unambiguous aliases still imply well-defined actions, but freedom from deadlock implies both that some event is always enabled to occur (an input must be acceptable to all views to which it is relevant) and that events are being interpreted properly by all parties (an input must be acceptable to all relevant views *under the same alias*).

4.3 Implementation Schemes

One of the most important properties of our specifications is that they are operational, or executable. They can be implemented in a variety of ways, with the various choices differing in such properties as similarity between the structure of the implementation and the structure of the specification, potential for parallelism, efficiency of sequential processing, and the effort required to "compile" each specification. Different situations suggest different decisions: for a production version on a uniprocessor mainframe, the comprehensibility of the implementation is unimportant, parallelism cannot be exploited, and compilation time can be large, but runtime efficiency is valuable. For a demonstration/validation version, the structure of the specification should be followed as closely as possible (so errors and unforeseen behavior can be traced easily to the specification), compilation time should be short (in anticipation of many compilations), parallelism in the specification should be preserved if it is to be preserved in the final implementation (for more accurate debugging), but runtime efficiency is unimportant.

Specifications based on our technique use parallelism both among and within views (the latter because there are parallel attempts to recognize different subsequences). We present several implementation schemes as roughly ordered by the amount of parallelism they preserve, making a coarse "continuum" of possibilities.



sequence diagram for View V
V
X*
Y°
Z°
A
B
A*
C
corresponding structure of function calls
V = *if* not(X) *then false else* V
X = *two-way-or*(Y,Z)
Y = *if* A *then* B *else false*
Z = terminated-iteration(A,C)
terminated-iteration(P,Q) = *if* P *then* Z *else if* Q *then true else false*
A = await-reply(request('A'))
B = await-reply(request('B'))
C = await-reply(request('C'))
some messages concerning the first input
'C'
Filter
request('A')
request('A')
request('C')
'A' fails
'A' fails
'C' succeeds
View V
View U

Fig. 19. An implementation scheme preserving full parallelism and specification structure.

(1) The implementation closest to the specification is one in which each view (and the filter) is implemented by a separate process. Within each view process, *the tree structure of the sequence diagram is mimicked by a tree structure of* function calls, each function seeking to recognize a sequence in the set denoted by some node in a sequence diagram, and returning a Boolean value indicating whether it succeeded (see Figure 19). This scheme preserves parallelism within views (as well as among them) because multiple functions can be active simultaneously.

A function corresponding to a leaf node in a sequence diagram is seeking to recognize a particular alias. It does so by sending a request for that alias to the filter process. When the filter receives a new input, it matches it against the current batch of requests from all views (which determine the set of aliases

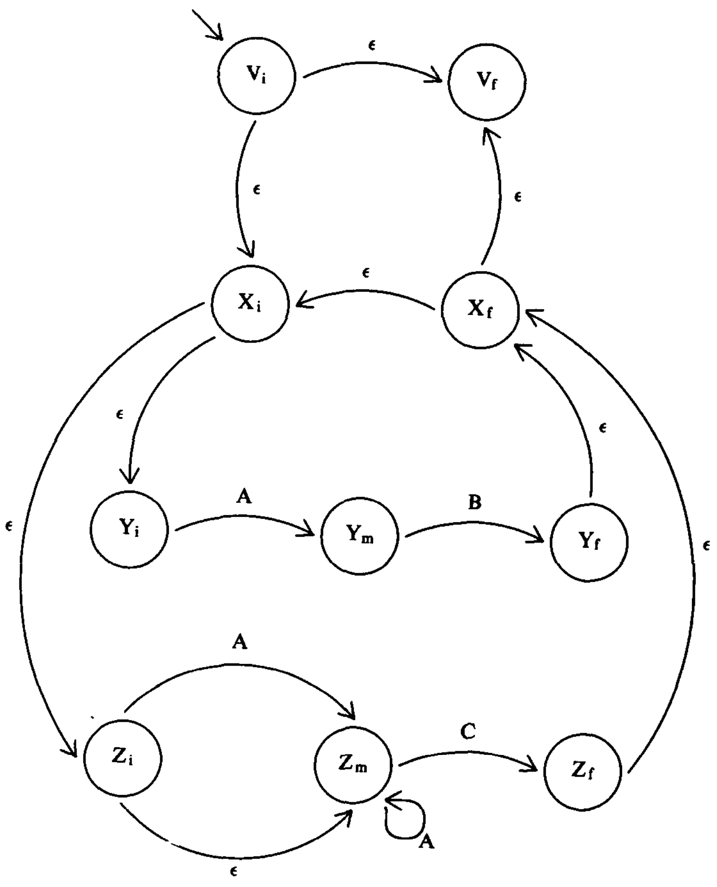


Fig. 20. A nondeterministic finite-state machine for view V. Epsilon transitions can occur at any time (without input), and the machine can be in many states simultaneously. The subscripts *i* and *f* refer to, respectively, initiating and finishing a named subsequence.

acceptable to each view at that time), and thereby interprets the input as some alias. Each request from each relevant view is then sent a "success" or "failure" reply to its attempt to recognize a particular alias.

The present author (in an earlier version of this paper [16]) describes how this scheme can be specified in the executable specification language PAISLey [17], and adds many details not discussed here. Actions can be specified in PAISLey as well. In this case our specification technique acts as a domain-specific front-end for the more general-purpose specification language PAISLey. The PAISLey version is not as readable as the sequence diagrams (this is the benefit of specialization), but it is more complete, and easily traced to the sequence diagrams.

Unfortunately, this is not a general solution, and requires patches in some cases. Compilation into PAISLey is relatively simple, requiring only a tree traversal of each sequence diagram to construct the functions, and one scan of all views to create the filter's tables of inputs, aliases, and relevance.

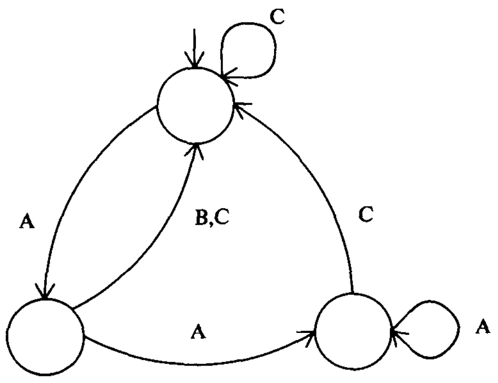


Fig. 21. A deterministic finite-state machine for view V.

(2) A similar implementation leaves the interaction between filter and view processes the same, but replaces the tree of function calls by a nondeterministic finite-state machine, as in Figure 20. The machine is slightly further from the original sequence diagram than the tree of function calls, but it is approximately the same in execution efficiency and compilation time, and requires no special cases.

(3) The next step is to implement each view process as a deterministic finite-state machine, as in Figure 21. Now the machine can be in only one state at a time, so the view process can send to the filter a single request listing the set of aliases it can accept, and receive a reply naming the particular alias. This scheme takes longer to compile than the previous ones and loses a great deal of specification structure, but it gains in runtime efficiency. Note that the potential for parallelism among views is retained and that the parallelism within views (which has been lost) does not provide an opportunity for computational speedup—only one set of actions is taken, no matter how many parallel subexpressions recognize the same alias.

(4) The final implementation reduces the whole specification to a single deterministic[5] finite-state machine. Parallelism and specification structure are gone entirely and compilation is the most expensive, but sequential execution is the most efficient.

The distinctions among these schemes are actually blurrier than indicated so far. The compilation time required to produce a global deterministic finite-state machine is not a disadvantage, for instance, if the machine is already being constructed for purposes of analysis. Even specification structure can be preserved by a global deterministic machine in the form of information relating each state to the original specification's views and pending subexpressions. In practice,

[5] It is also possible to construct a global *non*deterministic finite-state machine, although probably with prohibitive cost and little benefit. To preserve the correspondence between machine structure and specification structure, the machine should be the *product* of the nondeterministic view machines, that is, have a state for each possible combination of view machine states. This would be a large and confusing machine.

the choice may well be swayed by the availability of tools such as LEX [10], but designers should be aware of the options and the criteria by which they can be evaluated.

5. CONCLUSION

A technique has been presented for organizing and decomposing the structure of computations that accept or generate regular sequences of signals. The organized decomposition can be used to subdue the complexity of centralized modules or to analyze the behavior of distributed protocols. There are many possible applications of the technique, especially in the area of communications.

These results were obtained by extending and formalizing some features of the Jackson System Development method. I recommend JSD to anyone who is looking for insightful, imaginative solutions to the problems of system design.

APPENDIX: Remainder of the Telephone-Controller Specification

The views listed under each alias are the ones to which it is relevant. An alias can be relevant to a view even if it causes no actions in that view.

LOCAL VARIABLES

The filter has a local variable F.I (the identifier of the current call), a local variable F.CF (the current call-forwarding number), and a local variable F.ACBI (the identifier of the current automatic-callback instance). The POTS view has a local variable P.N (the number of the phone with which it is conversing) and a local variable P.I (the identifier of the current call).

```
ALIAS                                   INPUT               SOURCE
abort-RTC                               disconnect          another phone
  Data Items: I (the call identifier).
    Filtration: reject if I does not match F.I.
      POTS: send "end-power-ringing" to handset.
(ACB-timeout)                           ACB-timeout         timer
  Data Items: ACBI (the ACB instance identifier).
    Filtration: reject if ACBI does not match F.ACBI.
      ACB: send "ACB-light-off" to handset; send "finish-ACB" to ACB function process.
answer-RTACB                            off-hook            handset
      POTS: send "end-ACB-ringing" to handset; send "(RTC,my.N,P.I)" to P.N.
answer-RTC                              off-hook            handset
      POTS: send "end-power-ringing" to handset; send "(connect,P.I)" to P.N.
begin-RF                                disconnect          another phone
  Data Items: I (the call identifier).
    Filtration: reject if I does not match F.I.
      POTS:
(busy)                                  busy                another phone
  Data Items: I (the call identifier).
    Filtration: reject if I does not match F.I.
      POTS: send "begin-busy-signal" to handset.
cancel-ACB                              ACB-button          handset
      ACB: send "ACB-light-off" to handset; send "finish-ACB" to ACB function process.
CF-cancel                               CF-button           handset
      CF: send "CF-light-off" to handset.
```

CF-dial	dial	handset
Data Items: N (the number dialed).		
Filtration: if accepted store N as F.CF.		
CF: send "end-CF-dialtone" to handset.		
(CF-dialing-timeout)	CF-dialing-timeout	timer
Data Items: I (the call identifier).		
Filtration: reject if I does not match F.I.		
CF: send "begin-howler" to handset.		
CF-setup-quit	CF-button	handset
POTS:		
CF: send "CF-light-off" to handset.		
CF-setup-signal	CF-button	handset
POTS:		
CF: send "CF-light-flashing" to handset.		
(connect)	connect	another phone
Data Items: I (the call identifier).		
Filtration: reject if I does not match F.I.		
POTS: send "end-audible-ringing" to handset.		
dial-OC	dial	handset
Data Items: N (the number dialed).		
POTS: store N as P.N; send "end-dialtone" to handset; send "(RTC,my.N,P.I)" to N.		
end-RF	on-hook	handset
POTS:		
finish-good-CF-setup	on-hook	handset
POTS:		
CF: send "CF-light-on" to handset.		
(install)	install	installer
Life:		
local-finish	on-hook	handset
POTS: send "(disconnect,P.I)" to P.N.		
OC-aborted-before-ringing	on-hook	handset
POTS: send "(disconnect,P.I)" to P.N.		
(OC-dialing-timeout)	OC-dialing-timeout	timer
Data Items: I (the call identifier).		
Filtration: reject if I does not match F.I.		
POTS: send "begin-howler" to handset.		
postringing-OC-quit	on-hook	handset
POTS: send "end-audible-ringing" to handset; send "(disconnect, P.I)" to P.N.		
predialing-CF-quit	on-hook	handset
POTS: send "end-CF-dialtone" to handset.		
CF: send "CF-light-off" to handset.		
predialing-OC-quit	on-hook	handset
POTS: send "end-dialtone" to handset.		
(QRTC)	QRTC	ACB function
Filtration: since this alias belongs to no views, it is never accepted; if an RTC were accepted, then "idle" would be sent to the ACB function process, while if an RTC were rejected, then "busy" would be sent to the ACB function process.		
quit-when-busy	on-hook	handset
POTS: send "end-busy-signal" to handset.		
(remove)	remove	installer
Life:		

(ringing) ringing another phone
Data Items: I (the call identifier).
Filtration: reject if I does not match F.I.
POTS: send "begin-audible-ringing" to handset.

(RTACB) RTACB ACB function
Data Items: N (the number to be called automatically), ACBI (the ACB instance identifier), I (the call identifier, which is added by the filter).
Filtration: reject if ACBI does not match F.ACBI; if rejected by main view send "busy" to ACB function process, else if accepted by main view send "accepted" to ACB function process; if accepted get a new unique identifier and store as F.I, extend alias to "(RTACB,N, ACBI,F.I)".
POTS: store I as P.I, store N as P.N; send "start-ACB-ringing" to handset; request "(RTACB-timeout, ACBI)" in one minute.
ACB: send "ACB-light-off" to handset.

(RTACB-timeout) RTACB-timeout timer
Data Items: ACBI (the ACB instance identifier).
Filtration: reject if ACBI does not match F.ACBI.
POTS: send "end-ACB-ringing" to handset.

(RTC) RTC another phone
Data Items: N (the number of the other phone), I (the call identifier).
Filtration: if rejected by Life, forward to pseudo-phone connected to recorded message, if rejected by CF forward to F.CF, if rejected by Main send "(busy,I)" to N; if accepted store I as F.I.
Life:
POTS: store N as P.N, store I as P.I; send "begin-power-ringing" to handset; send "(ringing,I)" to N.
CF:

start-CF-setup off-hook handset
Data Items: I (the call identifier) is added by the filter.
Filtration: get a new unique identifier and store as F.I, expand alias to "(start-CF-setup,F.I)".
POTS: request "(CF-dialing-timeout,I)" in one minute; send "begin-CF-dialtone" to handset.
CF:

start-OC off-hook handset
Data Items: I (the call identifier) is added by the filter.
Filtration: get a new unique identifier and store as F.I, expand alias to "(start-OC,F.I)".
POTS: store I as P.I; send "begin-dialtone" to handset; request "(OC-dialing-timeout,I)" in one minute.

timeout-CF-quit on-hook handset
POTS: send "end-howler" to handset.
CF: send "CF-light-off" to handset.

timeout-OC-quit on-hook handset
POTS: send "end-howler" to handset.

try-ACB ACB-button handset
Data Items: ACBI (the ACB instance identifier) is added by the filter.
Filtration: get a new unique identifier and store as F.ACBI, expand alias to "(try-ACB,F.ACBI)".
POTS: send "(start-ACB,P.N,ACBI)" to the ACB function process for this phone.
ACB: send "ACB-light-on" to handset; request "(ACB-timeout,ACBI)" in 40 minutes.

ACKNOWLEDGMENTS

This paper could not have been written without John Cameron, who taught me JSD and spent many hours discussing telephones with me. Bill Schell contributed an idea when I needed it most. Nick Maxemchuk proposed the application to protocols, Al Aho and Pierre Wolper made valuable suggestions concerning implementation, Jon Bentley urged me to make numerous changes on the readers' behalf, and Steve Mahaney's advice on regular expressions spared me many fruitless hours.

REFERENCES

1. CAMERON, J.R. *JSP and JSD: The Jackson Approach to Software Development.* IEEE Computer Society, Long Beach, Calif., 1983.
2. CCITT, Functional Specification and Description Language (SDL). In *Recommendations Z.101-Z.104,* Vol. VI, Fascicle VI.7, Geneva, 1981.
3. DAVIS, A.M. The design of a family of application-oriented requirements languages. *Computer 15,* 5 (May 1982), 21-28.
4. DETREVILLE, J.D. Phoan: An intelligent system for distributed control synthesis. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pa., April 1984), 96-103.
5. HAILPERN, B.T., AND OWICKI, S.S. Modular verification of computer communication protocols. *IEEE Trans. Commun. COM-31,* 1 (Jan. 1983), 56-68.
6. HOPCROFT, J.E., AND ULLMAN, J.D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass., 1979, 60-62.
7. JACKSON, M.A. *Principles of Program Design.* Academic Press, New York, 1975.
8. JACKSON, M.A. *System Development.* Prentice-Hall International, Englewood Cliffs, N.J., 1983.
9. JACOB, R.J.K. Executable specifications for a human-computer interface. Naval Research Laboratory, Washington, D.C., Feb. 1983.
10. LESK, M.E. A lexical analyzer generator. CSTR 39, Bell Laboratories, Murray Hill, N.J., 1975.
11. RIDDLE, W.E. An approach to software system behavior description. *Comput. Lang. 4* (1979), 29-47.
12. RIDDLE, W.E. An approach to software system modelling and analysis. *Comput. Lang. 4* (1979), 49-66.
13. ROCKSTROM, A., AND SARACCO, R. SDL—CCITT Specification and Description Language. *IEEE Trans. Commun. COM-30,* 6 (June 1982), 1310-1318.
14. SUNSHINE, C.A. *et al.* Specification and verification of communication protocols in AFFIRM using state transition models. *IEEE Trans. Softw. Eng. SE-8,* 5 (Sept. 1982), 460-489.
15. TANENBAUM, A.S. *Computer Networks.* Prentice-Hall, Englewood Cliffs, N.J., 1981, 177-180.
16. ZAVE, P. A distributed alternative to finite-state-machine specifications. Bell Laboratories Tech. Memo. 82-11384-17, Murray Hill, N.J., Dec. 1982.
17. ZAVE, P. An operational approach to requirements specification for embedded systems. *IEEE Trans. Softw. Eng. SE-8,* 3 (May 1982), 250-269.
18. ZAVE, P. The operational versus the conventional approach to software development. *Commun. ACM 27,* 2 (Feb. 1984), 104-118.

Received December 1982; revised November 1983; accepted September 1984