This page intentionally left blank.

Formal Description of Telecommunication Services in Promela and Z

Pamela Zave
AT&T Laboratories—Research
Shannon Laboratory
180 Park Avenue, Room D205
Florham Park, New Jersey 07932, USA
pamela@research.att.com

Abstract. This paper shows how an engineer could write a full formal description of the service layer of a telecommunication system, organized according to the Distributed Feature Composition virtual architecture. Descriptions in Promela and Z can be composed using a joint semantics based on the transition-axiom method. The described system can be reasoned about in several ways, including use of tools developed for the individual languages.

1. The Distributed Feature Composition virtual architecture

Distributed Feature Composition (DFC) is a new architecture for the description of telecommunication services. One of its primary design goals was feature modularity. The other of its primary design goals was abstraction away from most implementation detail (hence the term "virtual"). As it appears to achieve these goals to a useful degree, it provides a good foundation for the application of formal methods to telecommunications.

DFC was developed by Michael Jackson and myself. A full definition of the architecture, along with motivations, intuitive explanations, and examples, can be found elsewhere [6,13]. We are currently exploring various extensions, analysis/verification techniques, and implementation strategies.

The goal of this paper is to provide a means by which an engineer can write a full formal description of the "service layer" [12] of a particular telecommunication system (excluding "business processes" such as billing, provisioning, marketing, and customer care), and apply formal reasoning to it. Because such a description will be organized according to the DFC architecture, it will have virtual components as shown in Figure 1.

In Figure 1 the double rectangles are repositories of global data, to which access is restricted by the architecture. Some data repositories span the system boundary because they are given their initial values by the environment, not by the system.

Squares in Figure 1 are DFC *boxes*, and can be thought of as concurrent processes with local state and *ports* (ports are represented by black circles). The virtual network establishes featureless voice *calls* between ports. When a call is established between two ports, those ports can communicate by means of a signaling channel in each direction and a voice channel in each direction. External lines and trunks also carry voice and messages in both directions, and are the means by which telecommunication services are delivered to telephones and other telecommunication systems, respectively.

Each box is either a line interface, a trunk interface, or the implementation of a particular feature. When a box attempts to place a new call, its request goes from the box's port to the *router* in the virtual network. The router determines a box destination for the

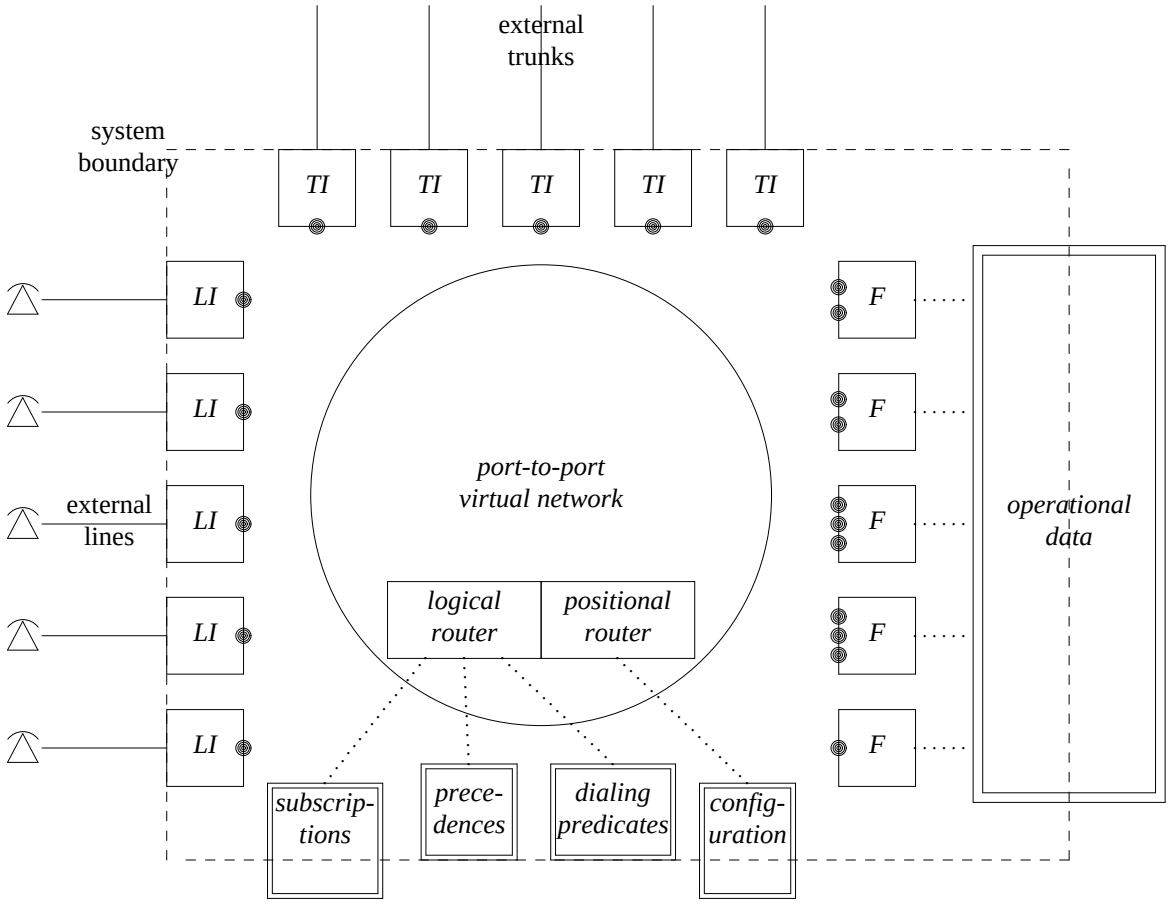

Figure 1. Components of the DFC architecture.

call based on feature-related criteria and also on the configuration of the system's environment. The destination box may accept or reject the call; if the box accepts the call it must name one of its own idle ports as the call's second endpoint.

The primary obstacle to reaching our goal is the fact that no single specification language is convenient enough to cover all aspects of Figure 1 on a large scale.

Jackson and I have used Promela [3], a protocol description language, very successfully to describe the protocols of the virtual network [6] and the control structures of boxes [15]. It has been used subsequently to describe the overall configuration of the virtual network. Being designed for model checking, however, Promela has minimal capabilities for representing system state. It is hopelessly inadequate for describing and manipulating the global data, which is richly structured relational information that expands and contracts in size.

Jackson and I have also used Z [11] very successfully to describe the routing data [6]. Z has been used subsequently to describe the routing algorithm and sample operational data. Yet we are extremely reluctant to undertake the arduous task of describing protocols and control-intensive boxes in Z, in which each control state must be named and manipulated explicitly. In a control-oriented language such as Promela, on the other hand, many convenient control abstractions are available, and most control states can be implicit.

In the rest of this paper I show how the goal can be achieved using Promela and Z together, with each language being employed to describe those aspects of the system for which it is best suited. Sections 2 and 3 define the composition technique and its semantics. Section 4 gives examples of the use of this technique in DFC descriptions. Section 5 discusses reasoning about descriptions in this form, while Section 6 gives reasoning examples. A discussion of related work can be found in Section 7.

2. Formal foundation

The formal foundation of this work is Lamport's transition-axiom method [9]. At the heart of this method is a set of functions on the states of a prospective system implementation. A system specification gives an initial value for each state function.

A system specification also has a set of actions. Each action is characterized by an enabling predicate on the values of state functions and by a rule constraining how the values of state functions change when the action occurs.

This is all that is needed for the specification of safety properties. An implementation satisfies a safety specification if and only if there exist functions on its state space that conform to the specification, in the sense that their initial values agree with the specification's initial values, and they change only in accordance with the specification's actions.

Liveness properties are added to a safety specification by the addition of formulas in temporal logic. The temporal operators operate on predicates over the values of state functions, and specify how these values must eventually change. Since no notion of fairness is built in, the temporal formulas must be satisfied by all safe schedules.

The safety part of the transition-axiom method is not a specification language, but is rather a means of associating a semantics with a specification language. The particular advantage of this style of semantics, according to Lamport, is that it answers the fundamental question of what it means for an implementation to satisfy a specification.

It is straightforward to give a transition-axiom semantics for Promela. The state functions of a Promela specification consist of each global variable, the local variables and parameters of each process, and the control pointer of each process (message channels are global variables). Variables that are not initialized explicitly receive initial values by default.

Each statement in a Promela specification, except for the unique process-initialization statement, defines an action. For an action to be enabled, all of the following must hold: (1) the control pointer of some process points to the statement, (2) if the statement is a predicate, it is true in the name scope of the same process, (3) if the statement is a channel write, the channel is not full, and (4) if the statement is a channel read, the channel has a message in it that satisfies the statement's constraints as evaluated in the name scope of the process. Execution of a predicate action simply changes the control pointer of the process. Execution of any other action changes the control pointer of the process, and also updates the values of other state functions in the obvious way.

The treatment of liveness properties in Promela is a perfect match to Lamport's approach. There are formulas in linear-time temporal logic for specifying liveness properties, and there are no built-in fairness constraints on process scheduling.

Z has a well-known operational interpretation, although this interpretation is not part of its set-theoretic formal semantics. It is also straightforward to give a transition-axiom semantics for this operational view of Z. The state functions of a Z specification are its variables. The initial values of state functions are given by *Init* schemas. Actions are defined by operation schemas. The enabling predicate of an action is the computed precondition of the operation schema—the condition that ensures that all invariants will still hold after the operation occurs. Z expresses safety properties only.

The transition-axiom method distinguishes actions performed by the system from actions performed by the environment. In both Promela and Z, this distinction can only be made informally. This distinction is extremely important, but it is not discussed further here because the composition of Promela and Z does not introduce any new issues.

3. Language composition

This section concerns how two descriptions, one in Promela and one in Z, are coordinated so that together they describe a telecommunication system. Section 5 will cover much of the same ground, but with a different emphasis: how the two descriptions can be understood and reasoned about separately.

There are three coordination mechanisms. For each I give syntax in Promela and Z, and semantics in the transition-axiom style, augmenting the separate semantics for Promela

and Z in Section 2.

3.1. Shared state functions

A state function in Promela and a state function in Z can be *shared.* Before going into the composition semantics, let us consider the syntax of sharing, i.e., how an engineer specifies which state functions are shared. Quite simply, state functions in Promela and Z are shared if they have the same name. This rule is easily applied to Z because all state functions (variables) in Z have names.

Name matching seems harder to apply to Promela because naming is less universal. State functions that are control pointers do not have explicit names. Names of local variables and parameters can only be understood in the context of their processes.

These gaps are not really a problem, however, because nameless state functions are never shared with Z. Control pointers are not shared because introducing that much control information into Z would defeat the purpose of using both languages. Local variables that need to be shared are turned into global variables. Parameters are shared in an indirect way, as explained in Section 3.3.

Two state functions are shared when both Promela and Z need access to the same underlying state information. The values of the two functions must be guaranteed by the specifier to maintain a given invariant relationship. Thus sharing is a hint to the implementer that both state functions can be supported by one state component in the implementation. From this perspective, one of the strengths of the transition-axiom method is that it allows multiple state functions to be implemented by the same state component.

Since the type system of Promela is strictly less expressive than the type system of Z, invariants between shared state functions can usually be written in Z (see Section 4.2 for the exception). Sometimes the invariant is a projection of Z values onto Promela values, specifically because of the weakness of Promela in representing state.

There are two ways for the specifier to guarantee that the invariant between shared state functions is maintained. It can be maintained "manually" by updating both state functions in tandem (Section 3.2 explains how to synchronize the updates), and by proving that the synchronized updates maintain the invariant when all the relevant enabling predicates are true.

The easiest way for the specifier to achieve an invariant relationship, however, is to update the state function in one language only, and to assume that the non-updated state function gets its values "automatically" from the updated state function, through the mediation of the invariant. Thus the non-updating description is using the shared state function in a "read-only" fashion.

Despite the many advantages of automatic update, it cannot be used when the updating description is in Promela and the invariant between shared state functions is a projection. Obviously, under these circumstances, a value produced by Promela cannot be translated by the invariant to a unique value in Z.

Table 1 summarizes the modes in which state functions can be shared between Promela and Z. An **A** entry in the table indicates that the read-only description is being updated automatically. An **X** entry in the table indicates an impossible combination. A constant state function is understood to be "written" only at its initialization. If a language requires that the value of an automatically updated state function that is declared in the language be initialized, then the initial value must have the invariant relationship with the initial value in the updating language.

	read/write in Promela, read in Z	read/write in Promela, read/write in Z	read in Promela read/write in Z
invariant is one-to-one	**A**		**A**
invariant is a Z-to-Promela projection	**X**		**A**

Table 1. A classification of shared state functions.

3.2. Shared actions

An action in Promela and an action in Z can be *shared.* Before going into the composition semantics, let us consider the syntax of sharing. As with state functions, actions in Promela and Z are shared if they have the same name.

This rule is easily applied to Z because all actions (operation schemas) in Z have names. Actions (statements) in Promela are usually not named, but they can be given names when needed by means of the macro facility. For example, if a use of the statement `skip` needs an action name, we can write

```
#define  Route_Call  skip
```

and use the macro name instead of the statement at the desired place in the Promela code.

Two actions are shared when they need to be synchronized in Promela and Z. Synchronization means that the one action occurs when and only when the other action occurs. In the transition-axiom semantics, shared actions are composed into one action whose enabling predicate is the conjunction of the Promela and Z enabling predicates, and whose state-update rule is the conjunction of the Promela and Z update rules.

Table 2 gives a classification of shared actions, based primarily on whether they have nontrivial enabling predicates in either language. An extremely important point—not emphasized in Table 2—is that a Promela action is never enabled unless a control pointer points to it (this is the significance of the asterisk on "always enabled" Promela actions). Indeed, the principal reason that Z actions are shared with Promela actions is to put the Z actions under Promela control. The marked places in Table 2 are the ones that are used in the DFC description.

		Promela			
		always enabled*		potentially disabled	
		`skip`	assignment	predicate	channel read/write
Z	always enabled	data update			
	potentially disabled	data query			synchronized update

Table 2. A classification of shared actions.

Z data is updated under Promela control by sharing a named `skip` action in Promela with an updating action in Z. As for querying Z data, consider the shared action Q, defined as `skip` in Promela and as a predicate *P* on the current state in Z. In Z, *P* is both the computed precondition and the enabling predicate of operation *Q*. If its value is false, then the statement `Q` in Promela cannot be executed. The effect in Promela is exactly the same as if statement `Q` had been a false predicate on some Promela variables, in which case `Q` would also be non-executable. The only difference between the two cases is whether the relevant state information is represented in Z or Promela, a distinction we are trying to ignore for these immediate purposes.

It is important to note that `Q` must be a guard in a guarded command with other

executable alternatives. If `Q` is not a guard, or if it is a guard in a statement with no other true guards, then execution of `Q` is mandatory—Promela has no other choice. This would be an inconsistency introduced by language composition.

Finally, when shared state functions are updated in both languages, the updates must be synchronized by sharing the update actions. Synchronized updates are listed under "potentially disabled" by both languages, because in both languages the same preconditions (such as that an empty channel cannot be read) apply.

3.3. Arguments to Z operations

Descriptions in Z are open in the limited sense that Z operations can have input arguments, the values of which are not provided by Z, and can produce output arguments, the values of which are not used by Z. When a Z operation occurs under Promela control, it can often be convenient to pass information between the shared Promela action and Z through the operation's arguments. Since the type system of Promela is strictly less expressive than the type system of Z, Promela-supplied values always make sense in Z, and Z-supplied values must be limited to those that make sense in Promela.

Promela constraints make the syntax a bit awkward. I suggest the following syntax for naming the Promela action and specifying the actual arguments:

```
#define  Port_Send_to_Switch__p__m  ...
```

Here the actual arguments to the Z operation are `p` and `m`; a double underscore seems to be the only marker available for delimiting arguments. In the Z operation *Port_Send_to_Switch*, actual arguments are matched to formal arguments according to ordering.

For the composed descriptions to be consistent, the type of a formal argument in Z must be the same as the type of the corresponding actual argument in Promela.

4. Examples of language composition

The following examples indicate all of the ways in which Promela and Z must be coordinated to describe DFC systems. They also cover all of the difficulties in detail.

4.1. Voice processing

Voice processing concerns what users of the telecommunication system hear and say. Within boxes of the DFC architecture, plain transmission can be augmented by conferencing, broadcasting, wiretapping, playing tones, playing recordings, making recordings, and monitoring for recognizable sounds such as touch tones. All the important issues can be illustrated, however, without tones or recordings.

Figure 2 shows a dynamically assembled configuration of lines, line interfaces, feature boxes, and calls, referred to as a *usage* in the DFC architecture. Lines *A* and *B* are connected to customers Alice and Bob. Line *S* is connected to a service representative of a company, and line *C* is connected to a coach who is training the service representative.

The usage arrived at the state shown in Figure 2 as follows. Alice called Bob, then placed a conferenced call to the company's toll-free number; the feature box on the left is an implementation of Three-Way Calling, the feature used by Alice to make the conference. The toll-free call was routed to the service representative by way of the coaching feature box, which automatically called a coach for help. The coach can listen to the entire conversation between the customers and the service representative, and also talk to the service representative without being heard by the customers.

Our goal in this subsection is to describe what can be heard by the user on each line. This description depends on two kinds of information: what is happening to voice signals inside the boxes, and what virtual calls are established outside the boxes.

The state of virtual calls is captured completely by the Promela state function `ex_con` (for *external connection*), declared and initialized as follows:

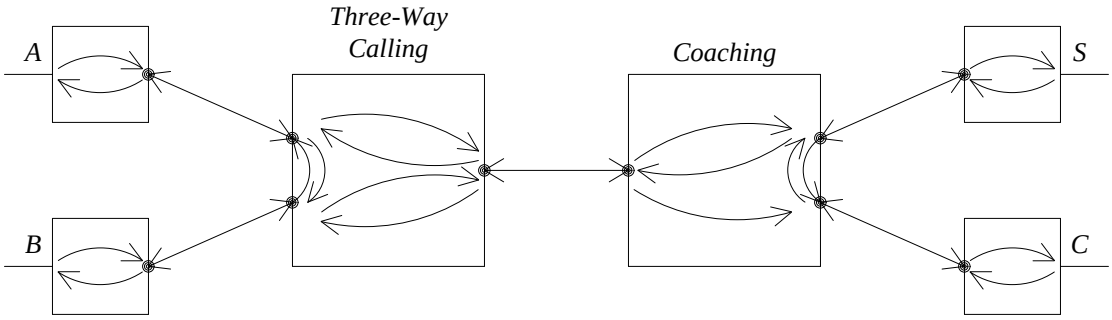

Figure 2. A snapshot of a usage. Small boxes are line interfaces, while large boxes are feature boxes. Double-headed arrows represent virtual calls, while single-headed arcs represent voice transmission in one direction only.

```
#define  Psize  ...

byte ex_con[Psize] = Psize
```

`Psize` is a constant state function giving the number of ports in the system, and ports are identified in Promela by natural numbers from 0 to `Psize-1`. Array `ex_con` of natural numbers is indexed from 0 to `Psize-1`. `ex_con[p] = q` if and only if there is an established call between ports `p` and `q`, in which case also `ex_con[q] = p`. If port `p` is not currently engaged in an established call, then `ex_con[p] = Psize`. Initially all ports are idle, and all array values are set to `Psize`. Thereafter the value of `ex_con` is maintained by the `switch` process in the Promela description of the virtual network.

The Z description needs access to the value of this state function. This effect can be achieved by declaring a shared state function in Z that gets its values automatically from the Promela state function. In Z its declaration is:

$$ex\_con: port \pfun port$$

with no initial value in Z. To write the invariant for this shared state function, we also need to share the constant state function

$$Psize: \mathbf{N}$$

with an invariant of identity. Then there is a bijection mapping ports to their identifiers in Promela:

$$p\_to\_id: port \rightarrowtail 0..Psize\text{-}1$$

Finally, the invariant for *ex_con*, which determines a unique Z value for each Promela value and vice-versa, is:

$$\forall p, q: port \bullet$$
$$(ex\_con_Z(p) = q) \Leftrightarrow \exists j, k: \mathbf{N} \bullet (p\_to\_id(p) = j \land p\_to\_id(q) = k \land ex\_con_P(j) = k)$$

Note that the Z expression $ex\_con_P(j)=k$ is treating the Promela array `ex_con` as a function, which seems perfectly reasonable in this context.

Voice processing inside the boxes is represented only in Z, by the value of the relation *in_con*. The domain and range of this relation are defined elsewhere as:

$$port, line, trunk: \mathbf{P}\ Appendage$$

$$appendage == port \cup line \cup trunk$$

The relation itself is described:

Internal_Connection $\hat{=}$
[
Configuration ;
in_con: appendage $\leftrightarrow$ *appendage*
|
$\forall$*v,w: appendage* $\bullet$
(v,w) $\in$ *in_con* $\rightarrow$ $\exists$ *b:box* $\bullet$ *attached_to(v,b)* $\wedge$ *attached_to(w,b)*
]

where the constraint says that *in_con* only connects appendages of the same box (this makes sense because each appendage belongs uniquely to a box).

If *(v,w)* is in *in_con*, then the voice signal into the box at appendage *v* flows out of the box at appendage *w*; this is pictured as a directed arc in Figure 2. If more than one arc is directed to an appendage, then the voice signal leaving the box at the appendage is the mixture (normalized sum) of the signals from all the arc sources. Initially there are no internal connections.

Many possible operations can be defined to update *in_con* as needed. Here is a simple example, an operation that makes a two-way internal connection between two ports.

Port_Talk $\hat{=}$
[
Configuration ;
Δ *Internal_Connection* ;
b?: 0..Bsize-1 ;
p?, q?: 0..Psize-1
|
(attached_to(p_to_id˜(p?),b_to_id˜(b?)))$\wedge$
(attached_to(p_to_id˜(q?),b_to_id˜(b?)))$\wedge$
(in_con' = in_con $\cup$ *{(p_to_id˜(p?),p_to_id˜(q?))}*)
]

The actual arguments *b?* (the box identifier), *p?*, and *q?* (the connected ports) are supplied by the synchronized Promela action, for example:

```
#define  Two_Talk__b__p__q  skip
```

The types of all of them are subranges of the naturals, which are legitimate in both Promela and Z. The operation precondition requires that ports *p?* and *q?* both be attached to box *b?*.

To describe what can be heard by the user on each line, we need to define a function that maps each box appendage to the set of original (from lines and trunks) incoming voice signals that are currently being mixed to produce the appendage's outgoing voice signal. In Z this function is defined axiomatically as:

[
voice_sources: appendage $\rightarrow$ **P** *(line* $\cup$ *trunk)*
|
voice_sources(v) = **dom**$($*port* $\lhd$ $($*in_con* $\cup$ *ex_con*$)^+$ $\rhd$ *{v})*
]

The transitive closure of *in_con* $\cup$ *ex_con* describes all current one-way voice paths. This relation may have cycles, but they do not matter because of the attenuation produced by mixing. Its definition removes ports because they are intermediate transmission points, not original voice sources. In the state depicted by Figure 2 we have

voice_sources(A) = {B, S}
voice_sources(B) = {A, S}
voice_sources(S) = {A, B, C}

voice_sources(C) = {A, B, S}

The descriptions in this subsection are not particular to any feature set, and would appear in all DFC formalizations.

4.2. Configuration

In DFC each port is attached uniquely and permanently to a box. In Z this aspect of the system configuration is represented by the value of the relation

attached_to: appendage $\twoheadrightarrow$ *box*

when domain-restricted to ports.

Attachment is crucial, global information, and must be shared with Promela. The problem is that, in Promela, there is no constant or variable containing this information. Rather, it is embedded in the process structure as created by the `init` statement. We have already seen that boxes and processes are identified by natural numbers in Promela. As an example of how the process structure is created in Promela, let a system contain a box implementing the Call Forwarding on Busy feature, with box identifier 44, and with attached ports identified as 44 and 45. The Promela process corresponding to the box is created by the clause

```
run CFB_box(44,44,45)
```

in the Promela `init` statement.

Thus the attachment state is both read and written by both Promela and Z, but it is not written in Promela as a value of any type. For this reason, the invariant for the shared state function *attached_to* cannot be written completely in Z—it must have informal parts. A partial invariant might look something like this, where boxes are mapped to natural identifiers by *b_to_id*:

If *b* is a CFB box such that
$\exists$ *p1,p2: port* $\bullet$ $\exists$ *j,k,l:* **N** $\bullet$
p1 $\neq$ *p2* $\wedge$
(attached_to(p1,b) $\wedge$ *attached_to(p2,b)* $\wedge$
b_to_id(b,j) $\wedge$ *p_to_id(p1,k)* $\wedge$ *p_to_id(p2,l)*
then and only then the `init` statement contains the clause
`run CFB_box(j,k,l)`

4.3. Signaling

Signaling is by far the most difficult challenge for language composition, because signaling has different aspects that are best represented in different languages.

Figure 3 shows the signaling channels and message types used for communication, in Promela, between a typical box process and the central `switch` process. Each port uses a pair of channels (one shared and one private) for participating in calls. The box itself also uses a pair of channels (one shared and one private) for accepting or rejecting new calls, and for designating ports for the accepted calls. The protocols used on these channels determine all of the control flow within the `switch` process, and most of the control flow in a typical box process.

The one aspect of signaling that is missing from the Promela description is the various data fields of messages, particularly setup messages. A setup message has several data fields, some of which are lists or sets. Such a nested data structure cannot be represented in Promela, but it can easily be represented in Z. Because the data fields of each type of message are different, the message type is defined as a free type in Z:

$mtype_Z ::=$
$setup_Z \langle\langle DN \times \mathbf{seq}\ DTMF\_char \times DN \times command \times \mathbf{seq}\ box\_zone \times ... \rangle\rangle \mid$
$quickbusy_Z \langle\langle DN \rangle\rangle \mid ... \mid$

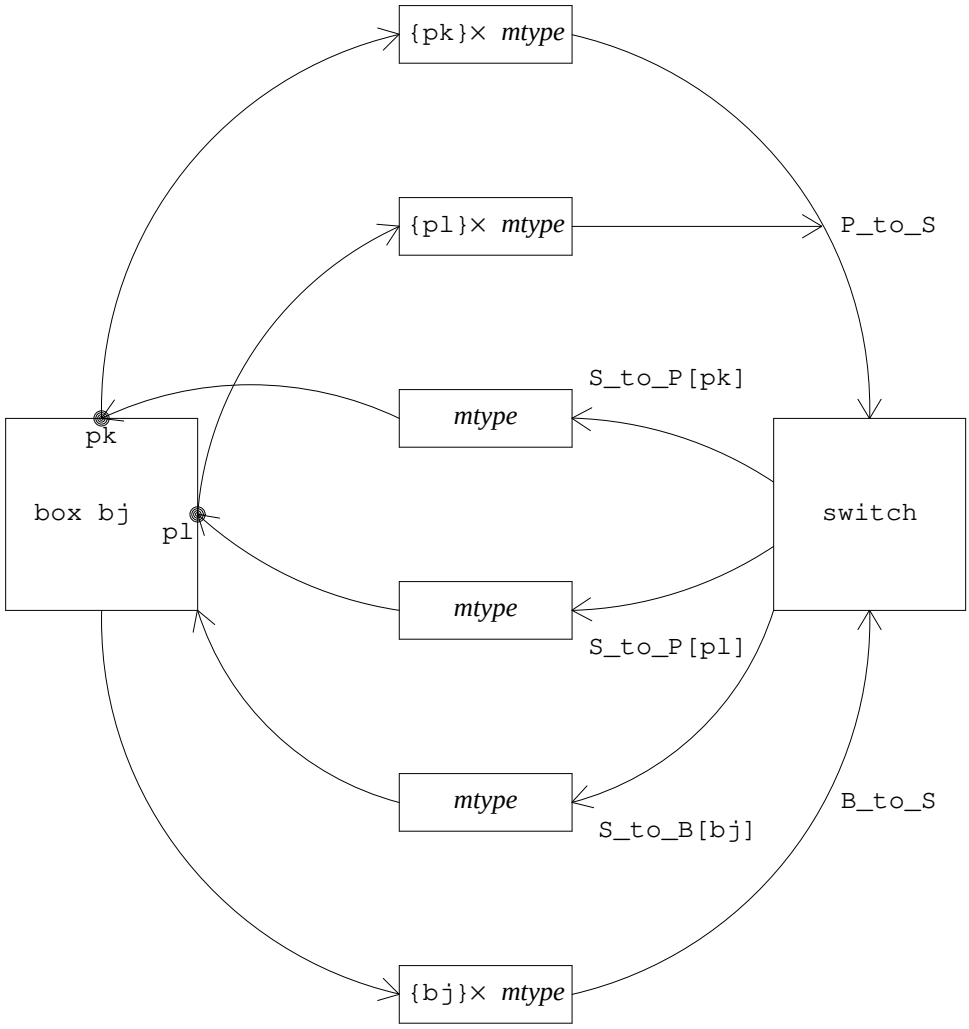

Figure 3. Signaling channels used by a typical box to communicate with the central virtual switch.

downack$_Z$

where a directory number *(DN)* is a restricted sequence of DTMF (touch-tone) characters, a *command* is one of four possibilities, and a *box_zone* is a pair.

There is also a message type in Promela; its values are projections of *mtype* values in Z:

```
mtype = { setup, quickbusy, ..., downack }
```

The counterpart of this, expressed within the Z type system, is:

mtype$_P$ *::=*
setup$_P$ | *quickbusy*$_P$ | ... | *downack*$_P$

For the sake of a future invariant, we need to define a projection function from Z messages to Promela messages. This function can be defined axiomatically as:

[
mtype_proj: *mtype*$_Z \rightarrow$ *mtype*$_P$
|

$$
\begin{array}{ll}
(& \forall\, m: mtype_Z \bullet m \in \textbf{ran}\ setup_Z \Rightarrow mtype\_proj(m) = setup_P \quad)\wedge \\
(& \forall\, m: mtype_Z \bullet m \in \textbf{ran}\ quickbusy_Z \Rightarrow mtype\_proj(m) = quickbusy_P \quad)\wedge \\
(& \ldots \quad)\wedge \\
(& \forall\, m: mtype_Z \bullet m \in \textbf{ran}\ downack_Z \Rightarrow mtype\_proj(m) = downack_P \quad)
\end{array}
$$

It is very important that the Promela description contain channel values, reads, and writes—they are at the heart of the protocols, and process descriptions would be useless without them. At the same time, the value of a channel variable in Promela is only a projection of its true value, which can only be represented in Z. Table 1 shows us that automatic update is impossible in this situation, so it follows that channel variables must be shared state functions that are updated in tandem in both languages.

As an example of how this is done, consider the channel variables `P_to_S` and `S_to_P`. These shared state functions are declared in Promela as:

```
chan P_to_S        = [...] of {byte,mtype}

chan S_to_P[Psize] = [...] of {mtype}
```

`S_to_P` is an array of channels, one read by each port, while all ports write to one channel `P_to_S`. The ellipses concern only maximum sizes. All channels are automatically initialized by Promela semantics to the empty sequence.

In Z the shared state functions are:

P_to_S: **seq** $(0..Psize\text{-}1 \times mtype)$

S_to_P: $0..Psize\text{-}1 \rightarrow$ **seq** $mtype$

The initial value of P_to_S is $\langle\rangle$. The initial value of S_to_P maps every member of its domain to $\langle\rangle$. The invariant between $P\_to\_S_P$ and $P\_to\_S_Z$ is simply:

$$P\_to\_S_P = P\_to\_S\_proj(P\_to\_S_Z)$$

where $P\_to\_S\_proj$ is defined axiomatically as:

$$
\begin{array}{ll}
[& \\
 & P\_to\_S\_proj: \textbf{seq}\ (0..Psize\text{-}1 \times mtype_Z) \rightarrow \textbf{seq}\ (0..Psize\text{-}1 \times mtype_P) \\
| & \\
(& P\_to\_S\_proj\ (\langle\rangle) = \langle\rangle \quad)\wedge \\
(& P\_to\_S\_proj\ (\langle(p,m)\rangle \frown s) = \langle(p,mtype\_proj(m))\rangle \frown P\_to\_S\_proj(s) \quad) \\
] &
\end{array}
$$

The invariant between $S\_to\_P_P$ and $S\_to\_P_Z$ is similar.

For exactly the same reasons that channel-valued variables need to be shared state functions updated in tandem in both languages, message-valued variables also need to be shared state functions updated in tandem. Normally message-valued variables in Promela would be local variables of the various processes. Since local naming introduces problems, however, I use a global array of message variables. The array is declared and initialized in Promela as follows:

```
mtype mvars[Msize] = downack
```

In Z it is declared as:

$mvars: 0..Msize\text{-}1 \rightarrow mtype$

and initialized correspondingly.

The *Msize* distinct message variables are identified by indices from 0 to *Msize-1*. Each Promela process has exclusive use of some number of variables from this collection,

and gets the indices of these variables through formal arguments. For example, if the Call Forwarding on Busy box mentioned above needs two message variables, its program declaration would be:

```
proctype CFB_box(byte b,p1,p2,m1,m2)
```

where formal argument `b` is the box identifier (as before), `p1` and `p2` are the port identifiers (as before), and `m1` and `m2` are the message variable identifiers. The association between boxes and message variables need not be shared with the Z description in any way. It is only necessary that, in the Promela `init` statement, each process is given message identifiers that are not given to any other processes.

Finally we come to the actions that update message and channel variables. In Promela, an action that a box executes to send a message to the switch on behalf of a port, and an action that a box executes to receive a message from the switch on behalf of a port, respectively, could be named as follows:

```
#define  Port_Send_to_Switch__p1__m1  P_to_S!p1,mvars[m1]

#define  Port_Recv_from_Switch__p1__m1
         S_to_P[p1]?mvars[m1]
```

In the corresponding Z operation *Port_Send_to_Switch* with input arguments *p?* and *m?*, the channel update is described:

$$P\_to\_S' = P\_to\_S \frown (p?, mvars(m?))$$

In the operation *Port_Recv_from_Switch* with input arguments *p?* and *m?*, the channel and message-variable updates are described:

$$S\_to\_P' = S\_to\_P \oplus (p?, tail\ S\_to\_P(p?))$$

$$mvars' = mvars \oplus (m?, head\ S\_to\_P(p?))$$

The head message is removed from sequence *S_to_P(p?)*, and it also becomes the new value of *mvars(m?)*, destroying its previous value.

This method of describing signaling is annoyingly redundant, but necessary in this style of language composition. The good news is that the redundant Z description of signaling is infrastructure that can be written once and then used by all features in all DFC systems.

4.4. Accessing operational data

Consider a CFB box that has received a setup message into a variable known locally as `m1`. This box has been routed to because the target of the call subscribes to CFB. However, the target subscriber may have turned the feature off temporarily. Thus the first thing that the box program must do is to ascertain whether or not the CFB feature is currently active for this subscriber.

The activation information is contained in the operational data for the CFB feature. It can be accessed through a Z operation *CFB_active* with two input arguments, the first of type *0..Bsize-1* identifying which box invoked the operation, and the second of type *0..Msize-1* identifying which message variable has the target field to be examined. The body of the Z operation is a predicate that is true if and only if the relevant target field names a subscriber whose CFB feature is currently active. As explained in Section 3.2, this predicate acts as the enabling condition of the Z operation.

In Promela the named action is defined as follows:

```
#define  CFB_active__b__m1  skip
```

and used in a guarded command as follows:

```
if
::  CFB_active__b__m1; ...
:: _CFB_active__b__m1; ...
fi;
```

where the ellipses indicate further actions to be taken if either guard is executable. If the enabling condition of the shared action `CFB_active__b__m1` is false in Z, then the action is not executable in either language.

The semantics of a Promela `if` statement demands that exactly one alternative be executed. So that there will be an executable alternative, `_CFB_active__b__m1` is also defined as `skip` in Promela, and is also a shared action whose Z operation has the complementary enabling condition.

4.5. Routing

As mentioned in Section 1, the router determines a box destination for each internal call. The *logical router* decides which additional feature boxes belong in the usage, and updates a routing list in the call's setup message accordingly. For example, Figure 4 shows a simple usage containing feature boxes Spontaneous Messaging on Busy and Set Call Forwarding on Busy because the source telephone subscribes to them, containing feature boxes Call Forwarding on Busy and Call Forwarding on No Answer because the target telephone subscribes to them, and also containing a Credit Card Calling box because of the exact form of the dialed digits.

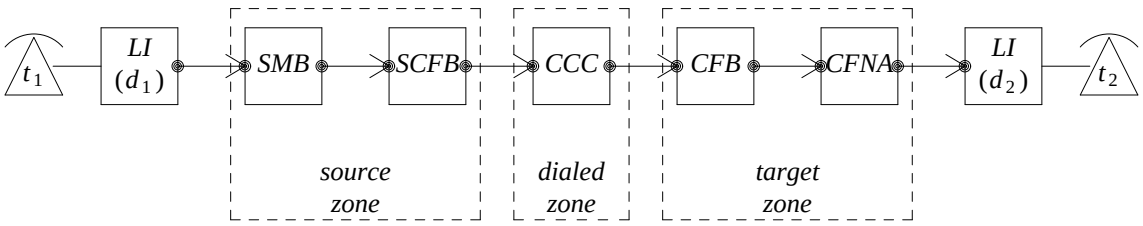

Figure 4. Routing zones in a simple usage.

The *positional router* returns the identifier of a specific destination box. If more feature boxes are still needed in the usage, the destination is typically an interchangeable instance of a feature box of the correct type. If no more feature boxes are needed, the destination is typically an interface box through which the target directory number can be reached.

The router and its routing data are all specified in Z as shown previously [6]. The action of routing a call is a Z operation performed under Promela control, and thus shared with a named `skip` action in Promela.

The identifier of the message variable holding the setup message to be examined and modified is passed to the *Route_Call* operation in Z as an input argument `m`. The identifier of the destination box is passed from the *Route_Call* operation as an output argument `b`. Thus the actual naming of the action in Promela is performed by:

```
#define  Route_Call__m__b  skip
```

It is interesting to note that *Route_Call* updates the Z state function *mvars*, which in turn is shared with the Promela state function `mvars`, which is not automatically updated by virtue of the Z operation. Why is the invariant between the shared state functions maintained? Simply because the change to *mvars* is in the data fields of a setup message, and it does not show up in the projected values found in `mvars`.

5. Language decomposition

This section concerns how a telecommunication system can be reasoned about despite the fact that it is described in two different formal languages, Promela and Z.

5.1. Joint execution

Probably the most common way to reason about a system description is to simulate the system by executing the description. This can be done in an exploratory, opportunistic fashion, or as part of a well-organized program of testing.

Promela descriptions can be executed by Spin [3]. Z is much harder to execute—even impossible in some cases—but there are now multiple tools for executing descriptions written in various subsets of Z [2,5,7,8]. Clearly, to be useful for the purpose of coordinating with Spin, a Z execution tool must operate in interactive mode.

There is no question that Spin and a Z execution tool would both have to be changed to accomplish joint execution. On the other hand, the changes seem to be peripheral rather than central. Also, it may not be necessary to support all the options shown in Tables 1 and 2. The DFC description, for example, does not use automatic updating of Promela state functions.

Thus a Z execution tool needs to be augmented only with automatic update of some state variables. As shown in Table 1, the translation from Promela values to Z values is guaranteed to be one-to-one, with no type incompatibilities.

A Z execution tool also needs to receive operation invocations and input arguments from Promela. Any Z execution tool, however, must be designed for external invocation of operations, since there is no concept of this internal to Z. Thus invocation of Z operations by Promela should be achievable without changing the Z tool itself.

Spin needs to be augmented with the ability to invoke operations in Z, passing input argument values to Z and receiving output argument values from Z. Also, if the invoked operation is not executable in Z because its enabling predicate (computed precondition) is false, then the shared action in Promela must be regarded by Spin as nonexecutable.

5.2. Model checking

Spin is primarily a model checker. From the perspective of decomposition, the prerequisite for model checking is meaningful execution of the Promela description in isolation. Model checking is simply an exhaustive search of the Promela execution space, which is known to be finite.

Working in isolation, Spin has no way to obtain values for output arguments from Z of shared actions. These values must be supplied by some additional Promela code.

Working in isolation, Spin never has an executable action rendered nonexecutable because of a shared action in Z. This is particularly significant for proving liveness properties, because it means that the isolated Promela description can have more possible behaviors than the described system. Thus, the isolated Promela description might satisfy liveness constraints that the actual system fails to satisfy.

As with output arguments, the deficiency in the Promela description must be rectified before model checking. This sounds difficult, but it is not. As Section 6 will show, the use of this composition capability is typically so limited and structured that it is not an impediment to model checking.

5.3. Model enumeration

Nitpick [5] is primarily a tool for model enumeration. This new analysis technique automatically finds counterexamples to asserted properties of specifications in a relational subset of Z. The inability of Nitpick to find a counterexample is not a proof of the property, because the size of models enumerated is limited, but it is powerful evidence that the property holds.

The temporal scope of Nitpick analysis is one operation. This must be the case for any analysis of pure Z, because pure Z has nothing even remotely temporal except the suggestive prime notation, which is usually used to distinguish the pre- and post-states of one operation.

As with model checking, the prerequisite for meaningful model enumeration is that

the Z description makes sense in isolation. Fortunately, this is always true. A Z description separated from its corresponding Promela description is missing only updates to its read-only shared state functions, if any.

Obviously the missing update operations cannot be analyzed using Nitpick, but their absence does not prevent Nitpick from analyzing any other operation. A read-only shared state function is known within Z by its type declaration and by its invariants. Those invariants should be made strong enough to establish the correctness of the operations that depend on them.

5.4. General verification

General verification requires that lemmas about the Promela description and lemmas about the Z description be combined to prove theorems about the description as a whole. This should not be difficult because the semantics of the transition-axiom method, where they meet, is simple and straightforward.

6. Examples of language decomposition

At the architectural level of a DFC description, there are several different kinds of property worth proving. In this section I assume that all descriptions have been statically type-checked.

First of all, there are properties establishing the internal consistency of a single-language description. Logic-based formalisms such as Z can easily express inconsistent assertions. In an operational language such as Promela, inconsistencies take the form of problems that abort execution, such as out-of-bounds array indices or type errors detected at runtime.

Secondly, there are properties establishing consistency between two languages. With the composition technique presented here, consistency fails only (1) if an invariant on shared state functions is not preserved, (2) if Promela passes Z an input argument of the wrong type, (3) if Z passes Promela an output argument of the wrong type, or (4) if a mandatory Promela action is shared with a Z operation having a false enabling predicate.

Finally, there may be requirements or specifications to satisfy. These are properties expressible strictly in terms of the system’s environment or its interface with the environment [16]. For example, a trunk interface must send signals on its trunk only in accordance with the protocol specified for that trunk. For another example, there may be "rules of telephone etiquette" that all features and services should observe.

6.1. Model-checking the protocols

I have used Spin model checking extensively to establish that the `switch` process and protocols of the virtual network never deadlock. This is a form of intra-Promela consistency, since a deadlock is an abnormal termination of a Promela program.

The only difficulty, that of making the Promela description execute meaningfully in isolation, proved minor. To see why, consider the box configuration in Figure 5, containing three line interfaces and a Call Waiting feature box associated with Line 0. The parenthesized numbers are box identifiers, while the numbers near ports are port identifiers. Arrows show the source and target boxes of possible calls.

It takes eight Promela processes to represent this configuration fully: the four boxes, the `switch` process, and a telephone process driving each line interface. The `switch` process manages six ports. From the perspective of the protocols it is a powerful test; from the perspective of model checking it is a big job.

Nevertheless, model-checking this configuration required very little *ad hoc* Promela code. There are no shared actions that are potentially disabled by Z. There is only one output argument from Z, carrying the identifier of the destination box from the routing operation in Z to the invoking `switch` process. In this configuration, as Figure 5 shows, the destination box of a call is almost completely determined by its origin box. Thus, the extra code to enable the `switch` process to do its own routing is trivial.

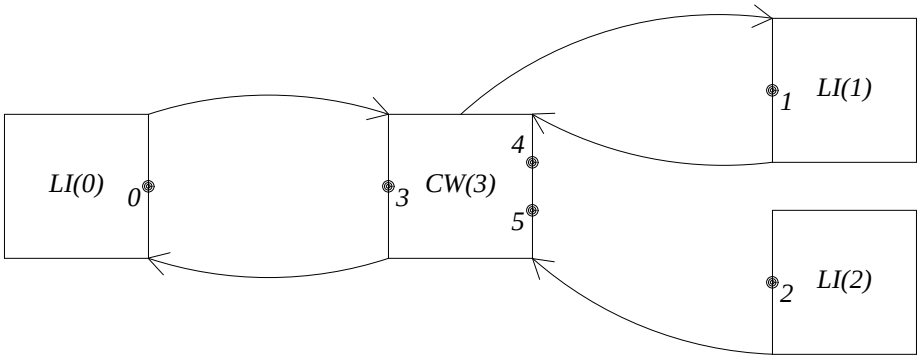

Figure 5. A box configuration for model checking.

6.2. Routing analysis

The routing algorithm and routing data are described entirely in Z, with no shared state functions. Thus the description of routing is completely independent of Promela.

Routing is complex, and there are many theorems one would like to prove about it, particularly that it always succeeds in finding a suitable destination box. This is a matter of inter-language consistency, because if routing fails in this way then the computed precondition of the Z routing operation is false, and the mandatory shared routing action in Promela is blocked.

Using a theorem prover for Z is probably the best reasoning choice, but it is tempting to consider model enumeration as well. Currently Nitpick cannot handle the relations on relations needed for routing, but it would be interesting to investigate whether there are abstractions of routing that can be checked for meaningful properties by model enumeration.

7. Related work

Jackson presents a method for organizing a Z specification into multiple views [4]. Shared state functions and shared actions are essentially the same as his two primary structuring mechanisms. He does not make special use of operation arguments, as they are a means by which a Z description communicates with its environment.

The differences between this composition/decomposition technique and Jackson's arise from the fact that he is working within one language while I am trying to utilize two. He does not need a separate composition semantics (the transition-axiom method) because composition of Z views can be expressed within Z. He does not need to find the common ground between two type systems, nor does he need to consider changes to existing language-based tools.

Fischer compares a number of different methods for combining Z with a process algebra [1]. Although there are some superficial similarities between his methods and the technique presented here, the similarities are misleading because the two things are actually fundamentally different.

All of the methods surveyed by Fischer are identifying some part of a Z description, for example an object in Object-Z, with a process in a process algebra. The algebraic process provides an operational semantics for the Z, answering such questions as in which states can an operation occur and in which states must it occur. Communication among these process/objects is described in two ways. Synchronization and routing are described within the process algebra. Data values communicated, and their types, are expressed in Z. Finally, since process algebras have no explicit states, there are no shared state functions.

I am using a completely different decomposition of "the things that need to be said" into the things expressed in Z and the things expressed in a process-oriented formalism. Z is given a particular operational semantics by assumption, not by explicit description in a

process-oriented formalism. There is no structured relationship between Promela processes and Z fragments. Z input and output are used for communication between Z and Promela, not for communication among different objects described in Z. In effect, my Z description operates as a big process on its own that interacts with the Promela processes at various points. And a Promela process has *two* mechanisms for interacting with the "Z process": shared actions and shared state functions.

Earlier work of ours [17] presents yet a different technique for composing Z descriptions with descriptions in other languages, particularly control-oriented notations such as automata. That earlier technique is more similar to this later work than the work surveyed by Fischer is, but there are still many differences.

Like this later work, that earlier technique has two major mechanisms through which languages can interact. One of them is shared state functions. In the earlier technique each shared state function is written within exactly one language, and read only by all the other languages that share it. This later technique would adopt the same happy simplification, except that the signaling problem discussed in Section 4.3 does not allow it.

In sharing state functions, the earlier technique matches types by requiring a much deeper, more detailed translation of each notation into a common semantics [14]. Thus dealing with types across languages is more general in the earlier technique, accommodating many languages instead of just our specific two, but also requires more work.

The second interaction mechanism in the earlier technique is "event classification," which allows one language to define a context-sensitive class of events that another language can respond to. Event classification could be used to achieve roughly the same kind of Promela-to-Z operation invocation achieved here by shared actions. Shared actions are superior in handling output arguments and disabling of Promela actions by Z far more elegantly. Event classification is superior in avoiding the problem discussed in Section 4.3. As with state functions, that earlier work requires a deeper, more complete translation of each language's syntax into a common semantics [14] than is required by this later work.

Now that the earlier and later techniques have both been exercised on large examples and are thoroughly understood, it may be possible to revise one of them to obtain the advantages of both.

Taking all together the technique presented in this paper, the three approaches discussed above, and new multiprogramming languages such as Seuss [10], there is now quite a variety of wide-spectrum formalisms within which one might plausibly hope to describe and reason about complex systems such as DFC telecommunication systems. What we do not yet have is any indication of which works better, for this or any other application. This is the most important direction for future research on this subject.

Currently there is rapidly growing interest in the subject of combining model checking with theorem proving. It should be noted that the issues of language composition and decomposition discussed in this paper are directly relevant to this pressing subject, because model checking is the natural way to reason about a Promela description, and theorem proving is the natural way to reason about a Z description.

Acknowledgments

I have profited greatly from discussions with Jean-Raymond Abrial, Jorge Cuellar, Cliff Jones, Jay Misra, and Michel Sintzoff on this work.

References

[1] Clemens Fischer. How to combine Z with a process algebra. In *Proceedings of the Eleventh International Conference of Z Users*, Berlin, Germany, September 1998.

[2] M. A. Hewitt. Automated animation of Z using prolog. Department of Computing Project Report, Lancaster University, England, April 1991.

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

[4] Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, IV(4):365-389, October 1995.

[5] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering* XXII(7):484-495, July 1996.

[6] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, to appear, October 1998.

[7] Xiaoping Jia. An approach to animating Z specifications. In *Proceedings of COMPSAC '95*, pages 108-113, 1995.

[8] R. D. Knott and P. J. Krause. The implementation of Z specifications using program transformation systems: The SuZan project. In C. Rattray and R. G. Clark, editors, *The Unified Computation Laboratory*, pages 207-220. Clarendon Press, Oxford, 1992.

[9] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM* XXXII(1):32-45, January 1989.

[10] Jayadev Misra. A discipline of multiprogramming. In this volume.

[11] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International, 1996.

[12] Pamela Zave. 'Calls considered harmful' and other observations: A tutorial on telephony. In Tiziana Margaria et al., eds., *Services and Visualization—Towards User-Friendly Design*, pages 8-27. Springer-Verlag Lecture Notes in Computer Science 1385, 1998.

[13] Pamela Zave and Michael Jackson. A component-based approach to telecommunication software. *IEEE Software* XV(5):70-78, September/October 1998.

[14] Pamela Zave and Michael Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology* II(4):379-411, October 1993.

[15] Pamela Zave and Michael Jackson. The DFC virtual architecture: Scenarios for use and plans for future work. AT&T Research Technical Memorandum, December 1997, `http://www.research.att.com/info/pamela`.

[16] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology* VI(1):1-30, January 1997.

[17] Pamela Zave and Michael Jackson. Where do operations come from? A multiparadigm specification technique. *IEEE Transactions on Software Engineering* XXII(7):508-528, July 1996.

Appendix: A box program for the Call Forwarding on Busy feature

The ultimate purpose of all the infrastructure presented in the body of the paper is to make it possible to describe feature boxes conveniently. Here is an example of how the Promela part of a feature box program built on this infrastructure looks; it is the Call Forwarding on Busy program alluded to several times already. Comments can be found at the end, linked to the program by line numbers.

```
01 #define  CFB_active__b__m1   skip
02 #define _CFB_active__b__m1   skip
03 #define  CFB_forward__b__m1  skip
04 #define  Box_Recv__b__m1   S_to_B[b]?mvars[m1]
05 #define  Box_Send__b__m2   B_to_S!b,mvars[m2],p1
06 proctype CFB_box(byte b,p1,p2,m1,m2)
07 {
08 BGN: Box_Recv__b__m1;
09      m2 = reserve; Box_Send__b__m2;
10      Recv_Up__p1__m2; Send_Down__p2__m1;
11      if
12      ::  CFB_active__b__m1; goto AWT
13      :: _CFB_active__b__m1; goto PWT
14      fi;
15 AWT: Recv_Down__p2__m2;
16      if
17      :: m2 == upack; goto PRE
18      :: m2 == quickbusy; goto FWD
19      fi;
```

```
PRE: do
     :: Recv_Down__p2__m2;
        if
        :: m2 == busy; m2 = teardown; Send_Down__p2__m2;
           goto CLF
        :: m2 == alerting; Send_Up__p1__m2; goto LNK
        :: m2 == teardown; Send_Up__p1__m2; m2 = downack;
           Send_Down__p2__m2; goto CL1
        :: !((m2==busy)||(m2==alerting)||(m2==teardown));
           Send_Up__p1__m2
        fi
     :: Recv_Up__p1__m2;
        if
        :: m2 == teardown; Send_Down__p2__m2; m2 = downack;
           Send_Up__p1__m2; goto CL2
        :: !(m2 == teardown); Send_Down__p2__m2
        fi
     od;
FWD: CFB_forward__b__m1; Send_Down__p2__m1; goto PWT;
CLF: do
     :: Recv_Down__p2__m2;
        if
        :: m2 == teardown; m2 = downack; Send_Down__p2__m2
        :: m2 == downack; goto FWD
        :: !((m2 == teardown)||(m2 == downack))
        fi
     od;
PWT: Recv_Down__p2__m2;
     if
     :: m2 == upack; goto LNK
     :: m2 == quickbusy; m2 = busy; Send_Up__p1__m2;
        m2 = teardown; Send_Up__p1__m2; goto CL1
     fi;
LNK: do
     :: Recv_Down__p2__m2;
        if
        :: m2 == answered; Two_Talk__b__p1__p2;
           Send_Up__p1__m2
        :: m2 == teardown; Send_Up__p1__m2; m2 = downack;
           Send_Down__p2__m2; Two_Untalk__b__p1__p2; goto CL1
        :: !((m2 == teardown)||(m2 == downack));
           Send_Up__p1__m2
        fi
     :: Recv_Up__p1__m2;
        if
        :: m2 == teardown; Send_Down__p2__m2; m2 = downack;
           Send_Up__p1__m2; Two_Untalk__b__p1__p2; goto CL2
        :: !(m2 == teardown); Send_Down__p2__m2
        fi
     od;
CL1: do
     :: Recv_Up__p1__m2;
        if
        :: m2 == teardown; m2 = downack; Send_Up__p1__m2
        :: m2 == downack; goto END
        :: !((m2 == teardown)||(m2 == downack))
        fi
     od;
CL2: do
```

```
79      :: Recv_Down__p2__m2;
80         if
81         :: m2 == teardown; m2 = downack; Send_Down__p2__m2
82         :: m2 == downack; goto END
83         :: !((m2 == teardown)||(m2 == downack))
84         fi
85      od;
86 END: skip
87 }
```

01 See Section 4.4.

03 The Z operation shared with this action replaces the *target* field of the setup message indexed by `m1` with a new target, namely the directory number that the original target is forwarded to.

04 This shared action is very like `Port_Recv_from_Switch__p1__m1` as presented in Section 4.3. The phrase `from_Switch` is redundant, as boxes never communicate directly with any Promela process except the `switch` process.

05 This shared action is very like `Port_Send_to_Switch__p1__m1` as presented in Section 4.3. However, the actual transmitted message contains *another* field: the identifier of the port that the box has selected for receiving this call. This field is completely invisible to Z. So the description as a whole contains *three* message abstractions, one private to each language and one "common denominator" for shared use.

08 This statement receives a setup message. This original setup message is stored in `m1` and stays there, being modified there if forwarding becomes necessary. The message variable indexed by `m2` is used for everything else.

09 See note to Line 05.

10 These shared actions are used by many boxes, and are defined elsewhere for all. In a box with two ports, we refer to the one that receives an incoming call as the *upstream* port, and the one that places an outgoing call as the *downstream* port. Boxes send and receive messages at both ports. Here the box receives an initialization message on the upstream port, and places a downstream call using the same setup message that it received.

12 Having attempted a downstream call, the box is waiting for its outcome. If the subscriber has activated this feature, the box goes to an active waiting state, in which it may do something special. If the subscriber has not activated this feature, the box goes to a passive waiting state, in which its behavior is transparent to all other boxes in the usage. See also the note to Line 47.

15 In a waiting state, a box is waiting to find out whether a call it attempted succeeds or not. An `upack` means it succeeded. A `quickbusy` means it failed because the destination box is busy (has no free ports), in which case this box is definitely going to forward.

20 In the "pre" state it has not yet been determined whether the outgoing call will alert or will suffer from a busy condition. In this state the box is waiting to find out. If the call turns out to suffer from a busy condition, then the box is going to forward, but first it must clean up the remnants of the downstream call. If the call results in alerting a telephone, then no forwarding will be needed, and the job of this box is effectively over.

38 This is where the box forwards. The shared operation is explained in the note to Line 03.

39 This state cleans up a downstream call, then goes to the forwarding state.

47 This line begins the "transparent zone" of the box program (note that once control reaches this line or a lower one, it never again jumps back above this line). In this zone the box is behaving transparently, i.e., it is behaving in such a way that it is unobservable by other boxes in the usage. Like all feature boxes, whenever this box reaches a point where no particular functions will again be required of it, it transfers control to the appropriate point in the transparent zone.

53 In this linked state, there are established calls at both ports. As soon as an `answered` message comes from downstream, the ports are voice-connected internally (see

Section 4.1). Status messages are forwarded in either direction. When either call is torn down, the box acknowledges the teardown, disconnects the internal voice connection, and initiates teardown of the other call.

70 This state presides over the final cleanup of the call at the upstream port. Cleanup is completely routine, and could be handled implicitly by a more application-specific language.