Features as Service Transformers

Glenn Bruns and Peter Mataga and Ian Sutherland
Software Production Research
Bell Labs, Lucent Technologies

August 24, 2000

Abstract

A primary goal of telecommunications companies is to reduce time to market. An obstacle to this goal in telephone switching equipment is that unwanted interactions between call processing features are difficult to discover, making extensive testing necessary. We present an approach to the specification of call processing features in which a feature is seen as a service transformer. We then define two notions of feature interaction, and illustrate how they can be used for the detection of feature interactions.

1 Motivation

The "feature interaction" problem is a real problem. At Lucent Technologies, extensive system testing is used to help ensure that features will function together properly when combined during development. Improved techniques for avoiding and detecting feature interactions could reduce the need for testing and therefore the time it takes to get new features to market.

Approaches to the feature interaction problem consist of a means of specifying features and a definition of what it means for features to interact. The challenge is to find a specification method and a definition of feature interaction in which: new features can be specified without modifying existing features; a broad range of features can be specified, including their intended interaction with other features; and the class of defined interactions contains most of the bad interactions we wish to locate and few of the good interactions that we intend to occur.

The ability to do these three things at the same time hinges on the way in which feature specifications are composed. The following list summarizes how features are composed in the three main approaches to feature specification:

- In the *logical* approach (e.g., [2, 5, 7]) features are specified as logical formulas and feature composition is logical conjunction. Feature interaction occurs if two features cannot be simultaneously satisfied.
- In the *network specification* approach (e.g., [1]) features are specified as sets of traces of network events and feature composition is set union. Feature interaction occurs if two features sets intersect after certain projection operations are performed.

- In the *operational* approach (e.g., [3]) features are specified as processes and feature composition is some kind of concurrent composition operation. Feature interaction occurs if the composed features fail to satisfy a global property such as deadlock freedom.

The first two approaches, especially the logical approach, allow simple and natural definitions of feature interaction. However, in these approaches it is difficult to define features that restrict the behavior of a POTS service or of other features. For example, suppose we want to specify a *teen line* feature. The teen line feature blocks outgoing calls made during a period of the day unless a PIN is supplied, and so is a restriction of what is allowed by POTS. In the network specification approach, system behavior is obtained by taking the union of feature traces, so one feature cannot prohibit what is provided by another feature. In the logical approach, defining teen line results in a feature interaction with the POTS service, which is undesirable.

In the operational approach it is possible to specify restrictive features. However, this approach suffers from complexity in the definition of feature composition. Furthermore, definitions of feature interaction in this approach are hard to relate to common intuition about feature interaction.

We present a new approach to the feature interaction problem based on the idea of features as service transformers. A service (such as POTS) describes the behavior of a server that responds to input events with one or more output events. Feature composition is the successive transformation of a service by a sequence of features. Two notions of feature interaction are defined. First, two features interact if the order in which they are applied affects system behavior. Second, two features interact if, in some state, an input can generate outputs that "interfere" in some sense. This second notion is related to the idea of feature interaction in the logical approach.

Our approach allows a wide range of features to be specified, including restrictive features like teen line. The notions of feature interaction it uses are simple and natural, and can be used as the basis of algorithms for the automatic detection of feature interactions. Although our approach was motivated by the feature interaction problem, it is general and can be used to specify other kinds of reactive systems.

The paper is organized as follows. After giving the assumptions we make about system architecture and development practices, we define a general specification model. We then define our two notions of feature interaction, and apply our our ideas to the problem used in the upcoming Feature Interaction Detection Contest [6]. We have reformulated the features supplied in the contest as general service transformers, and built a tool to test our ideas.

2 Assumptions

With regard to system architecture, we assume that a call processing system is an asynchronous network. The processing of a call can involve the sending and receiving of messages from other network components. For example, it can involve querying and updating databases in other network components.

With regard to development practice, we assume that new features are developed relative to a base system. We assume that developers of a new feature can view the description of the base system but not the description of other features that are under development. We also assume that developers of a new feature cannot modify the base system description.

3 Specification Model

Here we present a semantic model for service specification. To keep the focus on the main ideas of our approach, we do not define a notation. The notation used in the examples below and in the appendices is provisional, and used only for expository purposes. In developing the model, we have attempted to improve on the model underlying the Chisel specifications given in [6]. Chisel uses the network specification approach. A single feature is defined as a modification of POTS, and a system is specified as the union of the traces of the features. We generalize the Chisel notion of feature as POTS transformer to feature as service transformer. This approach allows POTS to be successively refined by the application of features. Also, the specification of [6] does not specify some important information, such as how variables are updated. Instead, this information is given informally in commentary accompanying the specification. Our formal model captures this information.

The main elements of our model are *services* and *features*. Intuitively, a service is what is provided by a server in a system organized as clients and servers. A service waits for an input, and then reacts to it by producing outputs. We define a service using local variables, concrete events, abstract events, and reactions. Concrete events are the input and output events of a service. When an input event is received by a service, the service interprets it as an *abstract event* depending on the state of the service. For example, a concrete `offhook` event may be interpreted as an abstract `newcall` event or an abstract `answer` event. A *reaction* of a service defines the variable updates and output events that should occur for a given abstract event.

A feature is specified as a service transformer. It transforms the service by defining new abstract events and new responses to abstract events. To define a service we begin with a *base service* and successively transform it through the application of features. Each feature refers only to events and variables of the base service and of itself. However, a base service that has been transformed by the application of features can serve as a new base for a later round of feature development.

In this paper we model the call processing for outgoing calls from one phone as a single service. To obtain a complete system we *compose* services by running them in parallel, letting them share concrete input events. In the following sections we provide a detailed description of services, features, and service composition.

3.1 Service

A *service* is defined as a sequence of declarations and statements. The declarations and statements of a service fall into six categories: *concrete event declarations*, *local variable declarations*, *abstract event declarations*, *process declarations*, *local variable update statements*, and *reaction statements*. In the following description of the various declarations and statements, we assume an arbitrary set of values, including at least the values *true* and *false*. An example service definition, for a POTS service, is given in Appendix A.

A *concrete event declaration* consists of an event name and a sequence of value sets that specifies the sets over which the parameters of the event range. The range of the i^{th} parameter is given by the i^{th} value set. An example from Appendix A is

```
offhook(A)
```

where `offhook` is the event name, the event has a single parameter, and A denotes the set $\{0, 1, 2\}$ of user identifiers that the parameter ranges over.

A *local variable declaration* consists of a variable name and an expression that specifies the initial value of the variable. An example from Appendix A is

```
bool busy(A) := (false,false,false)
```

This declaration says *busy* is a boolean variable indexed by elements of set A, and that $busy(i)$ is initially *false* for every index value i.

An *abstract event declaration* consists of an abstract event name, a concrete input event name, and a sequence of arguments of the concrete event. Informally, the declaration states that the named abstract event is considered to have occurred if the specified concrete event occurs and if the specified arguments match the arguments of the concrete event. Each specified argument can be either: a value expression, in which case the value of the expression must equal the actual argument, or a *variable* : *set* pair, in which case the value of the actual argument must belong to the specified set, and the specified variable name becomes bound to the value of the actual argument. An example from Appendix A is

```
o_newcall = offhook(o:A)
```

This statement says that an abstract event `o_newcall` is a concrete event `offhook` with index value in set A. If an `offhook` event is interpreted as an `o_newcall` event, variable o (for "originator") becomes bound to the index of the `offhook` event.

A *process declaration* consists of a process state name and a sequence of abstract event names. Informally, a process state is a kind of "point of control" for the service. A process declaration gives the name of a process state and shows the abstract events that can be input in that state. An example from Appendix A is

```
P1: o_newcall
```

This statement says that "P1" is the name of a process state, and that in process state P1 the service is only prepared to handle abstract event `o_newcall`.

A *local variable update statement* consists of five components: (1) a concrete event name, (2) an identifier for each parameter of the event, (3) a boolean expression built up from local variables and relational operators, (4) a local variable name, and (5) an expression of the same type as the local variable. Informally, the statement says that the variable should be assigned the value of the expression when the event occurs and the condition holds. In evaluating update statements, the identifiers specified as event parameters are bound to the values of the event indices. An example from Appendix A is

```
offhook(x) and not busy(x) => busy(x) := true
```

This statement says that the value of $busy(x)$ should be set to *true* when an `offhook` event with index x occurs and $busy(x)$ is false.

Finally, a *reaction statement* consists of six components: (1) an abstract event name, (2) a boolean expression on local variables and relational expressions, (3) a process state name, (4) a symbol *op* from the set {`+`, `-`, `:=`}, (5) a set of (concrete) output event names, and (6) a sequence of assignment statements *variable*`:=`*expr*. Informally, the statement defines a service's response to an input event. If the abstract event occurs and the condition holds, then a new output event set is computed, the assignment statements are performed in sequence, and a new process state is reached. The symbol *op* indicates whether the output events of this reaction statement are to be added to the current output event set (`+`), removed from the current output event set (`-`), or to be taken as the value of the output event set (`:=`). In defining services it is often the case that the condition is simply *true* and there are no assignments. An example from Appendix A is

```
o_newcall and not busy(o) => P2 / {dialtone(o)}
```

This statement says that if abstract event `o_newcall` occurs, and variable $busy(o)$ is false, then evolve to process state P2 and set the output event set to {`dialtone`(o)}. Here, and throughout the appendices, the absence of an *op* symbol indicates that `:=` is intended.

In defining a service we must also specify the initial process state. A service can then be written in the form

$$(CD, VD, US, AD, PD, RS, p_0)$$

where CD is a sequence of concrete event declarations, VD is a sequence of local variable declarations, US is a sequence of variable update statements, AD is a sequence of abstract event declarations, PD is a sequence of process declarations, RS is a sequence of reaction statements, and p_0 is the initial process state. For the POTS service of Appendix A the initial process state is P1.

The *state* of a service is a pair (p, E), where p is the name of the current process state and E is an *environment* that maps variables to values. The initial state (p_0, E_0) of a service is built of the specified initial state p_0 and the initial environment E_0.

We define the semantics of a service by defining a transition relation between service states. Each transition between service states is labelled with a concrete input and a set of concrete output events. We write $(S, p, E) \stackrel{c/C}{\rightarrow} (S, p', E')$ for a transition of service S from state (p, E) to state (p', E') on concrete input event c and concrete output event set C. Also, we write $(S, p, E) \rightarrow^* (S, p', E')$ if there exists a sequence of transitions from (S, p, E) to (S, p', E').

For lack of space we present only an informal overview of the semantics of a service. Suppose we have a service in state (p, E) and a concrete event c arrives. The output event set C and new service state (p', E') are computed in three steps:

Compute abstract event Consider each abstract event declaration in turn. Suppose that the current declaration *decl* is for abstract event a. If input event c satisfies *decl*, and the declaration for process state p shows that p is prepared to handle a, then let a be the computed abstract event. Also, update the environment to reflect any bindings produced by *decl*. Once an abstract action is found, proceed to the next step. If no abstract action is found, skip to the third step.

Compute output event set Initialize the set of output events to the empty set. Then consider each reaction statement in turn. If the abstract action of a reaction statement is a, and its condition is true in the current environment, then execute the statement by updating the output event set, by updating the environment to reflect the assignments of the statement, and by updating the process state if indicated by the statement. The output events of the statement are either added to, removed from, or used to replace the output event set, depending on the *op* of the statement.

Update local variables Consider each variable update statement in turn. If the event of the update is c, and the condition of the update holds, then update E as defined by the update statement. Let E' be the updated value of E, let p' be the possibly updated process state, and let C be the output event set computed in step 2.

Notice the difference between the way that concrete input events are interpreted and the way that the effect of abstract events is computed. In interpreting concrete input events, processing stops once an applicable abstract event declaration is found. In computing the effect of an abstract event, all reaction statements are examined.

Our semantics is unusual in that it allows a *set* of output events to occur in a single transition. The intuition behind the output set is that an implementation of the service may produce the outputs one at a time, but the order in which the outputs are produced is unimportant. The existence of an output event set will simplify a notion of feature interaction presented later.

3.2 Features and their Application

A feature definition differs from a service definition only in that it lacks concrete event declarations and an initial process state. Thus, in a feature definition one can define new abstract events, variables, etc.

Formally, a *feature* has the following form:

$$(VD, US, AD, PD, RS).$$

A feature f is a feature *of* service S if f refers only to concrete events, variables, and abstract events of S. Appendices B, C, D, and E are example feature definitions.

To apply a feature to a service we append the declarations and statements of the feature to those of the service. Let

$$S = (CD, VD, US, AD, PD, RS, p_0)$$

be a service and $f = (VD', US', AD', PD', RS')$ be a feature of S. Furthermore, let $seq_1 \cdot seq_2$ stand for the concatenation of sequences seq_1 and seq_2. The *application* of f to S, written $f(S)$, is the service

$$(CD, VD \cdot VD', US \cdot US', AD' \cdot AD, PD \cdot PD', RS \cdot RS', p_0).$$

Borrowing the notation for function composition, we write $f_1 \circ f_2(S)$ as shorthand for $f_1(f_2(S))$.

Notice that, unlike other declarations, the abstract event declarations of the feature are prepended to the abstract events of the service. Generally, we want declarations and statements of a new feature to take priority over the declarations and statements of the service to which it is applied. In particular, we want a concrete input event to be interpreted as an abstract event of the new feature if possible, so the abstract event declarations of the feature are placed before those of the service. We want an abstract event to be responded to after it has been responded to in the service, so the the response can be modified by the new feature. Therefore reaction statements of the feature are placed after the reaction statements of the service. Because the order of declarations and statements in a service affects the service's semantics, the order of applying features to a service is significant.

3.3 Composing Services

Given multiple services with the same concrete input events, we can compose them by running them in parallel. Consider services $S_1, \ldots, S_n$ with states $(p_1, E_1), \ldots, (p_n, E_n)$. Letting I be the index set $\{1, \ldots, n\}$, we write $\prod_{i \in I}(S_i, p_i, E_i)$ for the composition of the services. We define the transitions of the composed service as follows. If

$$(S_i, p_i, E_I) \stackrel{c_i/C_i}{\rightarrow} (S_i, p'_i, E'_i)$$

for all i in I, then

$$\prod_{i \in I}(S_i, p_i, E_i) \stackrel{c_i/C}{\rightarrow} \prod_{i \in I}(S_i, p_i, E_i)$$

where C is $\bigcup_{i \in I} C_i$. Informally, a concrete event given as input to the composed service is shared by the component services, and the output from the composed service is the union of the outputs from the component services.

In using our specification approach for call processing, a single service could process outgoing calls from a single user. Composing the services of every user gives a complete call processing system.

4 Feature Interaction

We are finally ready to define what we mean by feature interaction. In fact, we define two notions of feature interaction.

4.1 Interactions as Application Order Differences

Our first notion of feature interaction is that features interact if the order in which they are applied to a service affects the behavior of the system that is obtained. To make this notion more precise, we assume a behavioral equivalence $=$ on services. For example, we could use trace equivalence or bisimulation equivalence [8]. In our setting, a service has a trace $(c_1/C_1), \ldots, (c_n/C_n)$ if it can perform a sequence of transitions labelled with the elements of the trace. Two services are then trace equivalent if they have exactly the same set of traces. Given a permutation p of a set $\{f_1, \ldots, f_n\}$ of features, we will write $p(f_1 \circ \cdots \circ f_n)$ as shorthand for $p(f_1) \circ \cdots \circ p(f_n)$.

Let S be a service and let f and g be features for S. We say that f and g *order-interact* in S if there exists a set $\{f_1, \ldots, f_n\}$ of features for S, of which f and g are members, and a permutation p of $\{f_1, \ldots, f_n\}$, such that p maps every element but f and g to itself, and

$$f_1 \circ \cdots \circ f_n(S) \neq p(f_1 \circ \cdots \circ f_n)(S).$$

4.2 Interactions as Service Output Interference

Our second notion of feature interaction is that features interact if a service built from them can reach a state in which it outputs two events that interfere in some way. A simple kind of interference is that two output events can attempt to set a network component to different states. For example, one event might signal that a line busy tone be put on a phone while the other event signals that an audible ringing tone be put on the same phone. Another kind of interference is that two output events can trigger operations on a network component that do not commute. For example, one event might increment the value on a billing record for a customer while the other might zero the value on the record.

We do not attempt to define all possible notions of interference. We simply assume that in the system being specified it is possible to define an interference relation I such that $I(c, c')$ holds if the output events c and c' interfere.

Let S_{base} be a service, $\{f_1, \ldots, f_n\}$ be a set of features for S_{base}, and $S = f_1 \circ \cdots \circ f_n(S_{base})$. Let (p_0, E_0) be the initial state of S. We say that features $f_1, \ldots, f_n$ *output-interact* if there exist states (p', E') and (p'', E'') of S such that

$$(S, p_0, E_0) \rightarrow^* (S, p', E') \stackrel{c/C}{\rightarrow} (S, p'', E'')$$

and $I(c_1, c_2)$ for some events c_1 and c_2 of C.

Features of a service *interact* if they either order-interact or output-interact.

4.3 Detecting and Avoiding Feature Interactions

Given a base service and a set of feature definitions, we would like to be able to check automatically whether there are interactions between the features. Checking for order-interactions is expensive because it involves checking for equivalence between every two systems in which features were applied in different orders. The following property, if true, would be useful in making it easier to detect order-interactions:

$$f \text{ and } g \text{ order-interact in } S \Leftrightarrow f \circ g(S) \neq g \circ f(S).$$

With this property only one equivalence needs to be checked to determine if features f and g of S interact. The implication definitely holds from right to left. Even if the reverse implication does not hold in general, testing for the right-hand-side property may still detect many interactions.

Output-interactions can also be identified automatically. Given a service and a set of features, we apply the features in some order and then examine the state space of the resulting service. If some state has an output set containing interfering output actions, then an output-interaction exists. Unfortunately, unless we know that order-interactions do not exist, and therefore that the order of feature application is irrelevant, we must check every service that can be obtained through some sequence of feature applications.

4.4 Cheap Detection of Possible Interactions

An alternative to performing an expensive check for actual interactions is to perform a cheap check for possible interactions. One such check involves looking at reaction statements for an abstract event a of a service S in two features f and g of S. If the reaction statement of f extends the output event set, while the reaction statement of g restricts the output event set, then the order in which f and g are applied to S may be important. Similarly, if g extends the output event set, while f sets the value of the output event set, then again their order of application may be important. More generally, if the set operator of either reaction statement is `:=`, or one operator is `+` and the other is `-`, then a possible feature interaction of the first kind is possible. Similar checks can be made for feature interactions of the second kind.

These kinds of checks can not only report that an interaction is possible, but can show the particular reaction statements of other features that may cause problems. In this way the checks help feature developers to focus on the precise areas that may need attention.

5 Example and Tool Support

We are currently applying our specification model to the problem given in the Feature Interaction Detection Contest [6]. The definitions in the appendices show how we have captured the POTS service and features 1-3 and 6 of the contest materials.

We stress that our specifications of POTS and the features do not behave the same as the contest specifications. This was a deliberate decision on our part, based on what we see as a limitation in the network specification approach to feature interaction. As explained in Section 1, in this approach it is impossible to specify a feature that restricts the behavior of POTS or another feature.

We have built a tool (in ML [9]) that captures the semantics of our specification model. It defines data types for each of the declarations and statements of our model, allowing services and features to be coded as ML values. It also contains a function that computes the transitions for a service given its current state and a concrete input action. Other functions use this function to generate traces of output events given a sequence of input events.

We now present an example showing interaction between busy call forwarding and call screening. A situation in which one might suppose these features interact is the following. Suppose Alice would like her phone forwarded to Bob when Alice's line is busy, and that Bob would like calls from Carol to be screened. What should happen if Alice's line is busy when Alice receives a call from Carol? Figures 1 and 2 show two output traces produced by our tool for this scenario. The first is for the service obtained by extending POTS by 'call screening' first and then by 'busy call forwarding'. On each line labelled 'In:' the concrete input event is listed first, with the abstract event, if any, listed afterwards in parenthesis. A following line with label 'Out:' is presented only if an abstract event was found. In the first input of Fig. 1 user 0 goes offhook. Next, user 1 goes offhook, and then user 0 dials user 1. Since user 1 is busy, we see that the phone of user 2 starts ringing and user 0 gets an audible ringing tone. The call from 0 to 2 is not screened. The same sequence of inputs occurs in Fig. 2. However, when user 0 dials user 1 here, an announcement is made to user 0 stating that the call cannot

```
In:     offhook(0)  (o_newcall)
Out:    {dialtone(0)}

In:     offhook(1)  ()

In:     dial(0,1)  (o_dial)
Out:    {start_audible_ringing(0,2),start_ringing(2,0)}

In:     offhook(2)  (t_answer)
Out:    {stop_audible_ringing(0,2),stop_ringing(2,0),
         log_begin(0,2,0),log_begin(2,2,2)}

In:     onhook(0)  (o_end)
Out:    {disconnect(2,0),log_end(0,2),log_end(2,2)}

In:     onhook(1)  ()

In:     onhook(2)  (t_finish)
Out:    {}
```

Figure 1: A Trace for Features 'Call Screening' and 'Call Forwarding'

be made. The call from 0 to 2 has been screened. Since the system behaves differently when features 1 and 6 are applied in opposite order, the two features order-interact.

This feature interaction was discovered during the process of looking at sample scenarios, not through an automatic search by our tool.

6 Discussion

Much remains to be done in developing our specification model, our tool, and our examples. With regard to our specification model our main aim is to simplify it and to see how it works on a wider range of examples. For clarity, it might be more useful to define the semantics of our notation by translation to a value-passing process algebra. We have not done this so far because a simple translation seems difficult with the features in our language. For example, the sharing of events by component services in a composed service would be difficult to express in a notation like CCS [8] that supports only two-party synchronization. However, we could attempt to make small extensions to a simple process algebra and then define the semantics by translation.

We also want to remove limitations of our specification model. For example, in our current model the outputs of a service to an input event must always belong to a single set, and hence are imagined to occur in any order. Sometimes we need to specify that in a response to an input a service produces one output after another output. For example, in the call screening feature the `start_audible_ringing` output is supposed to occur before the `announce` output.

With regard to our tool, we need to extend it from a simulator to a tool that

```
In:    offhook(0)  (o_newcall)
Out:   {dialtone(0)}

In:    offhook(1)  ()

In:    dial(0,1)  (o_dial)
Out:   {start_audible_ringing(0,2),announce(0)}

In:    offhook(2)  ()

In:    onhook(0)  (o_finish)
Out:   {}

In:    onhook(1)  ()

In:    onhook(2)  ()
```

Figure 2: A Trace for Features 'Call Forwarding' and 'Call Screening'

can perform state-space search. This is a straightforward project, as the construction of such tools is well understood (see for example [4]). For the purpose of detecting instances of our first notion of feature interaction, we would also like to extend the tool so that it performs equivalence checking, which should also be straightforward.

We would also like to see how checking for feature interactions can be simplified. We would like to find properties, such as the one in Section 4.3, that will make detection of feature interactions less expensive. Similarly, we would like to find properties by which possible interactions can be found cheaply.

Finally, once our tool is extended, we want to see how widely it can be applied. At a minimum, we plan to apply it to the full set of features given in [6]. Of course, our primary goal is to see how we can help in discovering feature interactions in call processing platforms in Lucent. Toward this end we are discussing our ideas with local call processing experts.

References

[1] Alfred V. Aho, Sean Gallagher, Nancy D. Griffeth, Cynthia R. Schell, and Deborah F. Swayne. Sculptor with Chisel: Defining requirements for communications services, 1997. Submitted for publication.

[2] Johan Blom, Bengt Jonsson, and Lars Kempe. Using temporal logic for modular specification of telephone services. In W. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, 1994.

[3] Kong Eng Cheng. Towards a formal model for incremental service specification and interaction management support. In W. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, 1994.

[4] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.

[5] Anders Gammelgaard and Jens E. Kristensen. Interaction detection, a logical approach. In W. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, 1994.

[6] Nancy Griffeth, Ralph Blumenthal, Jean-Charles Gregoire, and Tadashi Ohta. Feature interaction detection contest instructions, 1998.

[7] Peter Mataga and Pamela Zave. Multiparadigm specification of an AT&T switching system. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, chapter 16, pages 375–398. Prentice Hall, 1995.

[8] Robin Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[9] Robin Milner and Mads Tofte. *The Definition of Standard ML.* MIT Press, 1990.

A POTS Service

```
  int set A = {0,1,2}       (* subscriber id's *)

Concrete input events:

  offhook(A),onhook(A),flash(A),dial(A,A)

Concrete output events:

  start_audible_ringing(A,A),stop_audible_ringing(A,A)
  start_ringing(A,A),stop_ringing(A,A)
  line_busy_tone(A),disconnect(A,A),log_begin(A,A,A),log_end(A,A)

Local variables:

  bool busy(A)              := (true,true,true)
  bool ringing(A,A)         := (false,false,false,
                                false,false,false,
                                false,false,false)
  bool audible_ringing(A,A) := (false,false,false,
                                false,false,false,
                                false,false,false)

Local variable updates:

  offhook(x) and not busy(x)            => busy(x) := true
  start_ringing(x,y) and not busy(x)    => busy(x) := true
  onhook(x) and busy(x)                 => busy(x) := false
  stop_ringing(x,y) and busy(x)         => busy(x) := false
```

```
  start_ringing(x,y) and not ringing(x,y) => ringing(x,y) := true
  stop_ringing(x,y) and ringing(x,y)      => ringing(x,y) := false
  start_audible_ringing(x,y) and not audible_ringing(x,y) =>
        audible_ringing(x,y) := true
  stop_audible_ringing(x,y) and audible_ringing(x,y) =>
        audible_ringing(x,y) := false

Abstract events:

  o_newcall = offhook(o:A)
  o_dial    = dial(o,t:A)
  o_hangup  = onhook(o)
  o_abort   = onhook(o)
  t_answer  = offhook(t)
  o_end     = onhook(o)
  t_end     = onhook(t)
  o_finish  = onhook(o)
  t_finish  = onhook(t)

Processes:

  P1: o_newcall (initial)
  P2: o_dial,o_hangup
  P3: o_abort,t_answer
  P4: o_end,t_end
  P5: t_finish
  P6: o_finish

Reaction statements:

  o_newcall and not busy(o) => P2 / {dialtone(o)}
  o_dial and not busy(t) =>
        P3 / {start_audible_ringing(o,t),start_ringing(t,o)}
  o_dial and busy(t) => P6 / {line_busy_tone(o)}
  t_answer =>
        P4 / {stop_audible_ringing(o,t),stop_ringing(t,o),log_begin(o,t,o)}
  o_end => P5 / {disconnect(t,o),log_end(o,t)}
  t_end => P6 / {disconnect(o,t),log_end(o,t)}
  o_abort  => P1 / {stop_audible_ringing(o,t),stop_ringing(t,o)}
  t_finish => P1
  o_finish => P1
```

B Feature 1: Call Forwarding Busy Line

```
Local variables:

  int  blf(A)     := (2,2,0)
```

```
  bool f1_sub(A)  := (false,false,false)

Reaction statements:

  o_dial and busy(t) and not(busy(blt(t))) and subf1(t) =>
        P3 / {start_ringing(blf(t)),start_audible_ringing(o,blf(t)}
        it := t; t := blf(t)
  o_dial and (not busy(t) or busy(blt(t)) or not subf1(t)) =>
        P3
        it := t
  t_answer and t <> it =>
        P4 / {stop_audible_ringing(o,t),stop_ringing(t,o),
              log_begin(o,it,o),log_begin(it,t,it)}
  t_end and t <> it =>
        P6 / {disconnect(o,t),log_end(o,it),log_end(it,t)}
  o_end and t <> it =>
        P5 / {disconnect(t,o),log_end(o,it),log_end(it,t)}
```

C Feature 2: Calling Number Delivery

```
Local variables:

  bool f2_sub(A)  := (true,false,false)

Reaction statements:

  o_dial and not(busy(t)) and subf2(t) =>
        P3 / + {(display(t,o))}
```

D Feature 3: IN Freephone Billing

```
Local variables:

  f3_sub(A)  := (false,true,false)

Abstract events:

  ot_route = response(AR,t,o,t,t)

Processes:

  P7: ot_route

Response statements:

  o_dial and f3_sub(t) =>
```

```
      P7 / {trigger(IA,t,o,t)}
ot_route and not busy(t) =>
      P3 / {start_audible_ringing(o,t),start_ringing(t,o)}}
ot_route and busy(t) =>
      P6 / {line_busy_tone(o))
t_answer =>
      P4 / - {log_begin(o,t,o)}
           + {log_begin(o,t,t)}
```

E Feature 6: Terminating Call Screening

```
Local variables:

  (* screened(i,j) is true if calls to i from j should be screened *)

  bool f6_sub(A)     := (false,true,true)
  bool screened(A,A) := (false,false,false,
                         false,false,false,
                         false,false,false)

Reaction statements:

  o_dial and screened(t,o) =>
        P6 / {start_audible_ringing(o,t),announce(o,"screened_message")}
```