Feature Interaction Analysis Using Z*

Michael Butler
Broadcom Éireann Research Ltd
Kestrel House
Clanwilliam Place
Dublin 2
Ireland.

12th October 1993

Contents

*This work was part of the CEC RACE project 2017 SCORE. Part of it also appears in *Report on Methods and Tools for Service Creation (First Version) Part II: Methods and Tools for Service Interaction Analysis.* R2017/SCO/WP2/DS/L/018/b2, January 1994.

1 Executive Summary

Feature interactions arise at different phases in the service creation process. Interactions may appear in the specifications that describe services from the users viewpoint. Equally, specifications that appear to have no interactions from the user viewpoint may interact at the implementation level because they are designed in such a way that they share some network resources. We have examined the use of the formal notation Z in analysing interactions in user-viewpoint service specifications.

Specifying Services

We take the view that a service is provided by a system, and such a system is modelled in Z by a set of state variables and a set of operations representing possible transitions for those state variables. All of these are described using set theory and predicate logic. A user is provided with a service by having a view of the system state, and by engaging in system operations that result in state changes.

Properties and Proofs

We can reason about a Z specification to check certain formal properties that it may enjoy. An important property of a system operation is its *pre-condition*. This is a condition (predicate) describing when an operation is guaranteed to result in a valid post-condition. We take the view that an operation may only be invoked when its pre-condition is satisfied.

For predicates P and Q, we write $P \vdash Q$ when it can be shown that Q is a logical consequence of P. We can check that given a pre-condition P, an operation S results in a post-condition Q, by showing

$$P \land S \quad \vdash \quad Q'.$$

(Here, Q is primed so that it becomes a condition on the state variables after invocation of S.)

An *invariant* is another important property of a system specification. An invariant describes "healthiness" conditions that the state variables should satisfy. It is important to ensure that all operations preserve invariants, and this is checked for operation S and operation I by showing

$$I \land S \quad \vdash \quad I'.$$

Objects and Inheritance

An approach to specifying services and supplementary features in Z has been developed and is described here. This approach was developed by attempting to add some IN-type features such as follow-me-diversion, call-diversion-on-busy, and call-screening to a Z specification of a basic connection service. The approach involves, firstly, specifying the basic service, then regarding the features as extensions of that specification. The extension mechanism used is based on Object-Z, an object-oriented version of Z. The basic service, consisting of some state variables and some operations, is regarded as an object. A feature is then specified as another object that includes the basic service object. An object that includes another object inherits all the state variables and operations of the included object. The new object may also have variables and operations of its own, and may extend the definitions of operations that it inherits. Multiple inheritance allows us to describe a system containing more than one supplementary feature.

An operation of an object may be declared to be internal. This means its invocation is outside the control of its environment (i.e., users). An internal operation is invoked internally by a system whenever its precondition is satisfied. For example, the call-diversion feature may include an internal operation that changes the destination address of call requests according to a call-diversion table.

Analysing Interactions

Along with an approach to describing services and features in Z, an approach to analysing interactions between service and features has also been developed and is described here. Various ways in which specifications of services and features can interact are detailed. This allows the interactions to be identified. Of course, not all identified interactions are necessarily undesirable. Often the interaction between a feature and a service is required for that feature to work. In other cases, we may identify an interaction that has undesirable consequences. We can use formal properties such as pre-conditions, invariants, etc, to help establish whether an identified interaction is desirable or not.

Three general forms of interaction that result when specifying services and features using our approach are described:

- interactions between a basic service and its features,
- interactions between different features,
- interactions between instances of the same feature.

Interactions between a basic service and its features result from the inheritance mechanism used for extending services. One way in which a basic service and a feature may interact is through their respective invariants. These should be compatible and not contradictory. The operations of the feature should not violate the invariants of the basic service, and vice versa. The specification approach allows a feature to change state variables of the basic service. Any operation of a feature that updates a variable of the basic service should be analysed to see that it preserves or establishes some desirable property. For example, in the follow-me-diversion feature, an operation that diverts the destination address of a call request should not result in that request being lost, or in the calling address being changed. Another way in which a feature interacts with a basic service results from the way in which the feature may extend an operation that it inherits from the feature. It may extend an inherited operation in such a way that its pre-condition is strengthened or alternatively that its pre-condition is weakened. Strengthening the pre-condition of an operation may result in it being blocked under certain conditions and we should check that such blocking is desirable. Weakening the pre-condition of an operation means that it may be invoked under more conditions than previously.

Multiple inheritance can be used to combine features. For example, we can specify a system that inherits the follow-me-diversion and call-screening features. Similar interactions to those just mentioned apply between the features and these may be analysed in a similar manner. Because of the way that an inherited operation can be extended, the order in which multiple features are inherited is important. For example, the pre-condition of a basic service operation may be strengthened by one feature and weakened by another. Care must be taken in this case since strengthening and weakening are not commutative. Another interaction between features that should be checked arises from the use of internal operations. Because internal actions are outside the control of an objects environment, the respective internal operations of two features can be invoked in a non-deterministic order and the consequence of this non-determinism should be checked. For example, if follow-me-diversion and call-diversion-on-busy are combined, then different orderings of the diversion operations associated with these features may produce different results. If necessary, an explicit mechanism can be added that controls the order in which diversions occur.

The nature of some features is such that there may be interaction between instances of the feature. For example, we may define the call-diversion feature such that transitive diversions are possible, i.e., if A is diverted to B and B is diverted to C, then incoming calls to A are diverted to C. In that case, we need to ensure that there are no loops in the transitive diversions, otherwise undesirable behaviour may result. With a formal notation like Z, we can impose invariants on the specification that prevent such loops, and then analyse the operations, particularly the operation to update the diversion table, to ensure they preserve these invariants.

Our approach to interaction analysis in Z is not systematic and does not guarantee the absence of undesirable interactions. Nonetheless, it has shown to be useful in helping to tackle the problem of feature interaction. The specification approach used allows us to see quite clearly the relationships within and between services and features. Formal reasoning allows us to analyse some consequences of these relationships. This allows us to make more informed decisions about whether interactions are undesirable and, if so, how they can be tackled. Although the development of the approach was based on the specification of IN-services, it should also be applicable to some other types of services like multi-media services.

2 Introduction

We look at ways in which Z [4] specifications of services and features may be analysed for interactions. Firstly, specification using Z is outlined. We look briefly at proving properties of specifications. Then a mechanism for extending specifications, based on Object-Z [2], is described. A basic specification object is used to describe a service, while the specification extension mechanism is used to add features to the service. We look at three general forms of interaction that result when specifying services and features using this approach:

- interactions between a basic service and its features,
- interactions between different features,
- interactions between instances of the same feature.

3 Specification in Z

A Z schema consists of a name, a declaration of variables, and a predicate:

$$\begin{array}{|l}
\hline \textit{SchemaName} \\
x : X \\
\hline \textit{Predicate} \\
\hline
\end{array}$$

Variable x is declared to be of type X. Note that the declaration part may declare more than one variable. The predicate part is a first-order predicate formula whose free variables are those of the declaration plus any constants.

A system specification in Z consists of some state variables, an initialisation, and a set of operations on the state variables. The state variables will also have some invariants associated with them representing "healthiness conditions" which must always be satisfied. Usually all of these are specified using schemas. For example, the state variables of a counter system may be specified using the following schema:

$$\begin{array}{|l}
\hline \textit{Counter} \\
ctr : \mathbb{N} \\
\hline 0 \leq ctr \leq max \\
\hline
\end{array}$$

Here, the predicate part describes an invariant that must be satisfied by ctr, the state variable of the system.

An initialisation may be specified as follows:

$$\begin{array}{|l}
\hline \textit{InitCounter} \\
\textit{Counter} \\
\hline ctr = 0 \\
\hline
\end{array}$$

An operation is specified in Z with a predicate relating the state before and after the invocation of that operation. For example, an operation to increment the counter may be specified as follows:

$$\begin{array}{|l}
\hline \textit{Increment} \\
\Delta(ctr) \\
\hline ctr < max \\
ctr' = ctr + 1 \\
\hline
\end{array}$$

The declaration $\Delta(ctr)$ means that the value of ctr is changed by the operation. In general, the Δ may refer to a list of variables to be changed known as the Δ-list[1]. In the predicate, the new value of a

[1] The Δ-list notation used here is slightly different to that used in standard Z. In standard Z, all state variables appear with the Δ, and it is explicitly specified which variables should not change. The Δ-list notation used here is based on Object-Z.

variable is primed, while the old value is unprimed. So the above predicate states that the new value of the counter, ctr', is the old value plus one. Any state variables of the system not mentioned in the Δ-list may be refered to in the predicate part, but only in unprimed form, which means they are not to be updated. Note that there is an implicit conjunction (logical-$\wedge$) between successive lines of the predicate part of a schema.

As well as changing the state variables, an operation may also have input and output parameters. Input parameter names are usually suffixed with '?', while output parameter names are suffixed with '!'. For example, the following operation for decrementing the counter has as an input parameter, the amount by which the counter should be decremented:

$$
\begin{array}{|l}
\hline
\textit{Decrement} \\
\Delta(ctr) \\
d? : \mathbb{N} \\
\hline
ctr \geq d? \\
ctr' = ctr - d? \\
\hline
\end{array}
$$

The following operation has an output parameter which is the value of the counter:

$$
\begin{array}{|l}
\hline
\textit{Display} \\
\Delta() \\
c! : \mathbb{N} \\
\hline
c! = ctr \\
\hline
\end{array}
$$

Note that the Δ-list is empty, so that the value of ctr is not updated by this operation.

Schemas may be combined using logical schema operators to form larger schemas. We make use of schema conjunction (logical-and) and schema disjunction (logical-or). Suppose we have schemas $S1$ and $S2$ as follows:

$$
\begin{array}{|l}
\hline
S1 \\
\Delta(u, v) \\
x, y \\
\hline
P1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
S2 \\
\Delta(v, w) \\
y, z \\
\hline
P2 \\
\hline
\end{array}
$$

Here, u, v, w are state variables, while x, y, z represent possible input and output parameters ($u, \ldots, z$ may represent sets of variables). $S1$ and $S2$ have only v and y in common.

The conjunction of $S1$ and $S2$, written $S1 \wedge S2$, expands to the following:

$$
\begin{array}{|l}
\hline
S1 \wedge S2 \\
\Delta(u, v, w) \\
x, y, z \\
\hline
P1 \wedge P2 \\
\hline
\end{array}
$$

Similarly, the disjunction of $S1$ and $S2$, written $S1 \vee S2$, expands to the following:

$$
\begin{array}{|l}
\hline
S1 \vee S2 \\
\Delta(u, v, w) \\
x, y, z \\
\hline
\quad P1 \wedge w' = w \\
\vee \\
\quad P2 \wedge u' = u \\
\hline
\end{array}
$$

The schema operators provide a useful way of separating specifications into small manageable units that can be combined in a logical fashion.

4 Properties and Proof

An important property of an operation is it's *precondition*. This is a condition on the initial value of the state variables and on the input parameters descibing when the operation will result in a valid new state (i.e., one that satisfies the state invariants) and valid output parameters. An operation of the form

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Operation}} \\ \hline
\Delta(v) \\
x? : X \\
y! : Y \\ \hline
P \\ \hline
\end{array}$$

has a precondition defined as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathrm{pre}\ \mathit{Operation}} \\ \hline
\mathit{Vars} \\
x? : X \\ \hline
(\exists\, v', y! \bullet P) \\ \hline
\end{array}$$

Here *Vars* represents the full set of state variables of the system of which *Operation* is a part. The predicate part of the precondition may be simplified using rules of the predicate calculus. For example, the precondition of the *Decrement* operation is as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathrm{pre}\ \mathit{Decrement}} \\ \hline
\mathit{ctr} : \mathbb{N} \\
d? : \mathbb{N} \\ \hline
\mathit{ctr} \geq d? \\ \hline
\end{array}$$

So the decrement operation may be validly invoked only if $\mathit{ctr} \geq d?$.

Sometimes we may wish to prove more general properties of specifications. For predicates P, Q, a proof of the form

$$P \quad \vdash \quad Q$$

is used to check that Q is a logical consequence of P. As a simple example, we have

$$\mathit{Counter} \wedge \mathit{max} = 100 \quad \vdash \quad \mathit{ctr} \leq 100.$$

Notice that we regard a schema, *Counter*, as a predicate.

A property of operations that may be checked is "given some initial condition P show that the operation results in the postcondition Q". This involves showing:

$$P \wedge \mathit{Operation} \quad \vdash \quad Q'.$$

Here, both P and Q are conditions on the state variables of *Operation*, and Q' represents Q with all variables in the Δ-list of *Operation* primed. A special case of this property involves checking that an operation preserves some invariant I:

$$I \wedge \mathit{Operation} \quad \vdash \quad I'.$$

5 Objects and Inheritance

Object-Z provides an extra level of structuring for Z specifications on top of the schema operators. The notation presented here is a slight variant of Object-Z. An object has its own state variables and operations. An object may include another object in which case it inherits all the state variables and operations of the included object. This inheritance mechanism is illustrated in Figure 1. *Object*1 has some state varibles with invariants, described in the schema *VAR*, an initialisation, and three operations. *Object*2 includes *Object*1. This means that the schemas of *Object*1 are available to *Object*2. The effects of the inclusion are that

BEGIN $Object1$

VAR

$\ldots$

$INIT$

$\ldots$

$Operation1$

$\ldots$

$Operation2$

$\ldots$

$Operation3$

$\ldots$

END $Object1$

BEGIN $Object2$

$Object1$

VAR

$\ldots$

$INIT$

$\ldots$

$\wedge$-extend $Operation1$

$\ldots$

$\vee$-extend $Operation2$

$\ldots$

$Operation4$

$\ldots$

END $Object2$

Figure 1: Objects and Inheritance

- the variables of $Object2$ are the variables of $Object1$ plus the new variables declared in $Object2$,
- the invariants of $Object1$ and $Object2$ are conjoined,
- the initialisations of $Object1$ and $Object2$ are conjoined,
- operations of $Object1$ may be extended using the schema operators; in the case of Figure 1, the new definition of $Operation1$ is the old definition conjoined with the schema '$\wedge$-extend $Operation1$', while the new definition of $Operation2$ is the old definition disjoined with the schema '$\vee$-extend $Operation2$',[2]
- additional operations may be defined for $Object2$ ($Operation4$ in this case).

When discussing and reasoning about specifications, we sometimes wish to state explicitly from which object a schema we are referring to comes. A schema of $Object$ is referred to as $Object.Schema$. For example, $Object1.Operation1$ refers to $Operation1$ of $Object1$, while $Object2.Operation1$ refers to the $\wedge$-extended version of $Operation1$ in $Object2$.

Internal Operations

An operation of an object may be declared to be internal. This means its invocation is outside the control of its environment. An internal operation is invoked internally by a system whenever its precondition is satisfied. We shall see an example of an internal action in the examples that follow.

Examples

Figure 2 contains an object, BC, which is a specification of the basic-connection telecommunications service. For illustrative purposes, some of the variable declarations and operation definitions have been omitted. Two important variables are *free*, representing those phones that are not requesting connections nor involved in any connection, and *request*, recording those connections that have been requested but

[2] In Object-Z, operations may only be extended by an implicit $\wedge$-extend operator; $\vee$-extension is not allowed.

BEGIN BC

$[Phone]$

VAR

$free : \mathbb{P}\, Phone$
$request : Phone \pinj Phone$
$\ldots$

$free \cap \operatorname{dom} request = \{\}$
$\ldots$

INIT

$\ldots$

Dial

$x : Phone$
$\Delta(free, request)$
$y? : Phone$

$x \in free$
$request' = request \cup \{x \mapsto y?\}$
$free' = free \setminus \{x\}$

DialInd

$x : Phone$
$\Delta(request, \ldots)$
$y! : Phone$

$y!\ request\ x$
$x \in free$
$request' = \{y!\} \ndres request$
$\ldots$

DialRespOk

$\ldots$

DialRespBusy

$\ldots$

$$DialResp \ \widehat{=} \ DialRespOk \lor DialRespBusy$$

Clear

$\ldots$

END BC

Figure 2: Basic Connection

not yet established (x *request* y holds if x has requested a connection with y). If a user x is free, then x may invoke the *Dial* operation with input parameter $y!$ representing the phone with which x desires a connection. The effect of the operation is to record the request. A user x can engage in a *DialInd* operation if some phone $y!$ has already requested a connection with x and x is free. Occurrence of the *DialInd* operation is an indication to x that $y!$ is requesting a connection.

Now we wish to describe the call-diversion service. We regard call diversion as an extension of the basic connection service and use the inheritance mechanism to extend the *BC* object of Figure 2. The extended service is described by the object *CD* of Figure 3. *CD* contains two extra variables. The first, *diversion*, represents diversions that have been requested: if x *diversion* y holds, then all calls destined for x will be diverted to y. Diversions may be changed by the operations *OrderCD* and *CancelCD*. When a request is made in the *CD* object, it is firstly checked to see if it should be diverted or not before the call should proceed. The variable *divchecked* records those requests that have been checked for diversion and can proceed. The *DialInd* operation is extended with an extra constraint which says it can only be engaged in when the corresponding request has been already checked for possible diversion. Diversions are processed by the *Divert* operation which checks a request made by phone x to some phone y. The operation is divided into two cases. In the first case, *Divert*0, y has been diverted to another phone z. The effect of the operation in this case is to update *request* so that x is recorded as requesting a call with z instead of y, and x is marked as having been checked for diversion. Now, z can receive a dial indication. In the second case, *Divert*1, y has not been diverted, so x is marked as having been checked for diversion and *request* is not updated. Note that *Divert* is an internal operation which means it is not invoked by the system user, but rather internally by the system.

Figure 4 gives an outline of the specification for the Call Screening feature. This allows a user to set up a blacklist of subscribers so that all incoming calls from a blacklisted subscriber are rejected and instead that calling subscriber gets a busy signal. In the specification, each subscriber is given a screen (blacklist), modelled as a set, to which it may add, remove, or query elements. The *DialInd* operation of *BC* is extended such that a callee will not receive a *DialInd* if the caller is on the callee's blacklist. The *DialRespBusy* operation is extended such that the caller gets a busy signal if the callee is busy or if the caller is on the blacklist of the callee.

6 Interaction Between Basic Service and Features

Figure 2 describes a basic connection service object, while Figure 3 describes how this may be extended to include the call-diversion feature. We look at some of the ways in which *BC* and *CD* interact.

6.1 Invariants

Firstly, we look at the relationships between the invariants of *BC* and *CD*. It is important that these two invariants should not contradict each other, i.e. that

$$BC.INV \wedge CD.INV \quad \neq \quad \mathit{false}.$$

Clearly the (partial) invariants specified in Figures 2 and 3 do not contradict each other.

Regarding the preservation of invariants by operations, we need to check that the new and extended operations of *CD* preserve the invariants of *BC*, as well as its own invariants. *Divert* is a new operation in *CD*, and it can be shown to preserve $BC.INV$:

$$\mathit{free} \cap \operatorname{dom} \mathit{request} = \{\} \ \wedge \ \mathit{Divert} \quad \vdash \quad \mathit{free}' \cap \operatorname{dom} \mathit{request}' = \{\}.$$

If an operation preserves some invariant, then trivially its $\wedge$-extension will preserve that invariant, i.e.,

$$I \wedge \mathit{Operation} \ \vdash \ I' \quad \text{implies} \quad I \wedge (\mathit{Operation} \wedge \mathit{OperationExtension}) \ \vdash \ I'.$$

Since $BC.DialInd$ preserves $BC.INV$, $CD.DialInd$ also preserves $BC.INV$. The same is not true when we $\vee$-extend an operation. In that case, the $\vee$-extension has to be checked to ensure it preserves the old invariant, and we shall see an example of this later.

We should also check that the operations of the basic service preserve invariants of the new feature. In the case of *CD*, it is easy to see that $BC.Dial$ preserves $\mathit{divchecked} \subseteq \operatorname{dom} \mathit{request}$. The old version of the *DialInd* operation does not preserve $\mathit{divchecked} \subseteq \operatorname{dom} \mathit{request}$ since it removes x from $\operatorname{dom} \mathit{request}$ without changing *divchecked*. The extended version of *DialInd* also removes x from *divchecked*, however, so it preserves the invariant.

BEGIN *CD*

BC

VAR

$diversion : Phone \nrightarrow Phone$
$divchecked : \mathbb{P}\, Phone$

$\ldots$
$divchecked \subseteq \mathrm{dom}\, request$

INIT

$\ldots$

OrderCD

$x : Phone$
$\Delta(diversion)$
$y? : Phone$

$x \in free$
$diversion' = diversion \oplus \{x \mapsto y?\}$

CancelCD

$x : Phone$
$\Delta(diversion)$

$x \in free$
$diversion' = \{x\} \lhd diversion$

$\wedge$-extend *DialInd*

$x : Phone$
$\Delta(divchecked)$
$y! : Phone$

$y! \in divchecked$
$divchecked' = divchecked \setminus \{y!\}$

$\wedge$-extend *DialRespBusy*

$\ldots$

*Divert*0

$x : Phone$
$\Delta(request, divchecked)$

$\exists\, y, z : Phone \bullet$
$x\ request\ y$
$x \notin divchecked$
$y\ diversion\ z$
$request' = request \oplus \{x \mapsto z\}$
$divchecked' = divchecked \cup \{x\}$

*Divert*1

$x : Phone$
$\Delta(divchecked)$

$\exists\, y : Phone \bullet$
$x\ request\ y$
$x \notin divchecked$
$y \notin \mathrm{dom}\, diversion$
$divchecked' = divchecked \cup \{x\}$

Internal $Divert \;\widehat{=}\; Divert0 \vee Divert1$

END *CD*

Figure 3: Call Diversion

BEGIN CS

BC

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{VAR}} \\
\hline
screen : Phone \pfun (\mathbb{P}\, Phone) \\
\hline
\operatorname{dom} screen = subscribers \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\wedge\text{-extend } \textit{DialInd}} \\
\hline
x : Phone \\
\Delta() \\
y! : Phone \\
\hline
y! \notin screen(x) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\vee\text{-extend } \textit{DialRespBusy}} \\
\hline
x : Phone \\
\Delta(request, free) \\
y! : Phone \\
r! : RespType \\
\hline
x \ request \ y! \\
x \in screen(y!) \\
r! = busy \\
\ldots \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddScreen}} \\
\hline
x : Phone \\
\Delta(screen) \\
y? : Phone \\
\hline
screen' = \\
\quad screen \oplus \{\ x \mapsto (screen(x) \cup \{y?\})\ \} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RemoveScreen}} \\
\hline
x : Phone \\
\Delta(screen) \\
y? : Phone \\
\hline
screen' = \\
\quad screen \oplus \{\ x \mapsto (screen(x) \setminus \{y?\})\ \} \\
\hline
\end{array}$$

END CS

Figure 4: Call Screening

6.2 Updating Variables

The operations of a feature may update variables of objects it inherits. This will impact the behaviour of the service; so, we need to analyse the way in which inherited variables are updated by a feature, to determine whether this safe or not. For example, the *request* variable of *BC* is updated by the *Divert* operation of *CD* resulting in calls being diverted. It can be shown that

$$Divert \wedge x \ request \ y \wedge y \ diversion \ z \quad \vdash \quad x \ request' \ z.$$

So the *Divert* operation interferes with the basic connection service and changes its behaviour since now z will receive a *DialInd* instead of y. However, this interference is precisely what we expect from the call-diversion feature and is not harmful. It would be more serious if it could be shown that *Divert* did not preserve $x \in \operatorname{dom} request$, since this would mean that *Divert* was causing connection requests to be lost.

6.3 Precondition Strengthening

When an operation is $\wedge$-extended, its precondition may be strenghtened which reduces the set of states in which it may be invoked. This leads to the operation being blocked which may not be desirable. This occurs, for example, with our specification of *CallDiversion*: the precondition of *BC.DialInd* is

pre *BC.DialInd*

$$\begin{array}{l} x : Phone \\ BC.VAR \\ \hline (\exists\, y : Phone \bullet y \ request \ x \\ \quad x \in free \ ) \end{array}$$

while the extended *DialInd* is stronger:

pre *CD.DialInd*

$$\begin{array}{l} x : Phone \\ CD.VAR \\ \hline (\exists\, y : Phone \bullet y \ request \ x \\ \quad x \in free \\ \quad y \in divchecked \ ) \end{array}$$

This means that *CD.DialInd* is blocked if

$$y \notin divchecked.$$

Of course, this is desirable since we need to prevent a callee getting a *DialInd* that should be diverted. But we don't want to block the diverted callee getting a *DialInd*, and the *Divert* operation ensures this doesn't happen since it can be shown that

$$\begin{pmatrix} Divert[x/y] \\ y \in \operatorname{dom} request \\ y \notin divchecked \end{pmatrix} \vdash \begin{pmatrix} y \in \operatorname{dom} request' \\ y \in divchecked' \end{pmatrix}.$$

Thus, we see that the strengthening of the precondition of *DialInd* is not harmful.

6.4 Precondition Weakening

When an operation is $\vee$-extended, its precondition may be weakened which enlarges the set of states in which it may be invoked. We must ensure that this enlargement of the precondition is desirable. For example, the precondition of the *BC.DialRespBusy* operation is

$$
\begin{array}{|l}
\hline
\text{pre } BC.DialRespBusy \\
x : Phone \\
BC.VAR \\
\hline
(\exists y : Phone \bullet \\
\quad x \ request \ y \\
\quad y \notin free \ ) \\
\hline
\end{array}
$$

while when we extend Basic Connection to add Call Screening, the precondition of $CS.DialRespBusy$ is:

$$
\begin{array}{|l}
\hline
\text{pre } CS.DialRespBusy \\
x : Phone \\
CS.VAR \\
\hline
(\exists y : Phone \bullet \\
\quad x \ request \ y \\
\quad y \notin free \ ) \\
\vee \\
(\exists y : Phone \bullet \\
\quad x \ request \ y \\
\quad x \in screen(y) \ ) \\
\hline
\end{array}
$$

Again we can see that this is precisely what we intended of the extension of *DialResp*, so weakening of the operation does not lead to unexpected behaviour in this case.

7 Interaction Between Different Features

7.1 Multiple Inheritance

The inheritance mechanism for objects may be regarded as an operator that transforms a basic object into a richer extended object by enlarging its state space and extending and adding to its operations. If $Object1$ is a basic object and $Object2$ inherits $Object1$, then we write

$$Object1 \oplus Object2$$

for the transformation of $Object1$ according to the definitions of $Object2$. In this way, we can easily combine a number of features with a basic service. For example, we write

$$(BC \oplus CD) \oplus CS$$

for the object that results from extending the Basic-Connection object firstly according to the definitions of the Call-Diversion object, and then according to the definitions of the Call-Screening object.

7.2 Order of Inheritance

The order in which features are inherited is important since our inheritance operator is not commutative, i.e., it is not always the case that

$$(Object1 \oplus Object2) \oplus Object3 \quad = \quad (Object1 \oplus Object3) \oplus Object2.$$

This lack of commutativity arises because of our use of both conjunction ($\wedge$-extension) and disjunction ($\vee$-extension) for extending operations: it is not the case that

$$(Op1 \wedge Op2) \vee Op3 \quad = \quad (Op1 \vee Op3) \wedge Op2.$$

So when one feature $\wedge$-extends an operation and another $\vee$-extends that operation, then we have to be careful about the order in which these features are combined. For example, the Call-Diversion feature $\wedge$-extends the *DialRespBusy* operation while the Call-Screening feature $\vee$-extends the *DialRespBusy* operation, so that

$$(BC \oplus CD) \oplus CS \quad \neq \quad (BC \oplus CS) \oplus CD.$$

$(BC \oplus CD) \oplus CS$ and $(BC \oplus CS) \oplus CD$ differ in there resultant *DialRespBusy* operation. Using schema expansion we can calculate the resultant *DialRespBusy* operation in both cases and then determine which ordering we require.

7.3 Nondeterminism of Internal Operations

The execution of internal operations is not controlled by the environment of services and features and we make no assumption about the order in which internal operations are exectued. This means that when we combine features that have internal operations, the nondeterministic interleaving of their respective internal actions may lead to nondeterministic behaviour.

(CDB) feature that is only diverts call requests an internal operation of *CD* The effect of executing

Suppose we combine *Object*1 and *Object*2 with internal operations *Op*1 and *Op*2 respectively. One way of analysing the nondeterministic ordering between *Op*1 and *Op*2 is to calculate explicitly the sequential compositions "*Op*1; *Op*2 " and "*Op*2; *Op*1" and then compare them. Suppose we have operations *Op*1 and *Op*2 as follows:

$$\begin{array}{|l}
\hline
Op1 \\
x : X \\
\Delta(u, v) \\
\hline
P1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
Op2 \\
x : X \\
\Delta(v, w) \\
\hline
P2 \\
\hline
\end{array}$$

Then their sequential composition is defined as follows:

$$\begin{array}{|l}
\hline
Op1;\ Op2 \\
x : X \\
\Delta(u, v, w) \\
\hline
(\exists\, v'' \bullet P1[v' \backslash v''] \ \land\ P2[u, v \backslash u', v''] \ ) \\
\hline
\end{array}$$

(The final state of *Op*1 is equated with the initial state of *Op*2.)

As an example, suppose we combine the call-diversion feature with a call-diversion-on-busy feature. Like the call-diversion feature, the call-diversion-on-busy feature has an internal action, *BusyDivert*, that diverts call requests if the called party is busy and has invoked the feature. We may calculate the sequential compositions "*Divert*; *BusyDivert*" and "*BusyDivert*; *Divert*". Now suppose the following condition P holds for phones x, y, y', y'', z', z'':

$$P \ \ \widehat{=} \ \ \left(\begin{array}{l} x \ request \ y \ \land \\ y \ divert \ y' \ \land \ y' \ busydivert \ z' \ \land \\ y \ busydivert \ y'' \ \land \ y'' \ divert \ z'' \end{array} \right)$$

That is, the callee, y, has both call-divert and call-divert-on-busy invoked to two different phones. Given P holds, the outcome of "*Divert*; *BusyDivert*" is shown to be different to the outcome of "*BusyDivert*; *Divert*":

$$\begin{array}{l} Divert;\ BusyDivert \ \land\ P \ \ \vdash \ \ x \ request' \ z' \\ BusyDivert;\ Divert \ \land\ P \ \ \vdash \ \ x \ request' \ z'' \end{array}$$

That is, the combined internal divert operations can nondeterministically result in x being diverted to z' or to z''.

Of course, the suitation whereby y has both call-diversion and call-diversion-on-busy invoked is somewhat unusual. If we elaborated the specifications such that only one type of call-diversion can be invoked by a subscriber, then suitation P could never arise, and the combined features wouldn't cause nondeterministic outcomes. Alternatively, we could allow more than one type of diversion to be invoked by a subscriber (e.g., a subscriber may like to invoke diversion based both on time of day and on originating address) and resolve the nondeterminism by specifying in what order the divert operations should be applied.

8 Interaction Within Features

Sometimes the nature of some features is such that there may be interactions between instances of that feature. For example we may define a Call-Forwarding-Unconditional (CFU) feature in which transitive diversions are possible, i.e., if A is diverted to B, and B is to C, then all incoming calls to A are diverted to C. In this case, two instances of call diversion are being combined: that from A to B and that from B to C. Suppose *forward* is a function that maps phones to the phones to which they are forwarded:

$$forward : Phone \nrightarrow Phone.$$

For two subscribers, x and y, we say that y is an *endpoint* of x if there is a transitive forwarding path from x to y, and y is not itself forwarded to another phone:

$$x \ endpoint \ y \quad \Leftrightarrow \quad x \ forward^* \ y \ \wedge \ y \notin \mathrm{dom}\, forward$$

(Here, $forward^*$ represents the reflexive, transitive closure of *forward*.)

If we use *endpoint* to specify how call requests are diverted, then we need to ensure that each phone has one, and only one, endpoint, i.e., that *endpoint* is a total function. It turns out that if we can show that there are no loops in the forwarding table, i.e., *forward* satisfies:

$$(\forall x : Phone \bullet \neg (x \ forward^+ x)),$$

then endpoint is a total function. Proof of this is presented in section A.

We assume the no-looping condition is an invariant of the CFU feature which must be preserved by all operations of the feature. The *OrderCFU* operation updates the forwarding table, so that all calls to x are diverted to $y?$:

$$\begin{array}{|l}
\textit{Update} \\
\hline
x : Phone \\
\Delta(forward) \\
y? : Phone \\
\hline
forward' = forward \oplus \{x \mapsto y?\} \\
\hline
\end{array}$$

It can be shown that if we add the precondition

$$\neg (y? \ forward^+ \ x)$$

to the *OrderCFU* operation, then the operation can be shown to preserve the no-looping invariant, thus preserving the soundness of *endpoint*.

Regarding the other features described here, it is easy to see that there is no interaction between instances of the feature. This is because the effect they have on any one call request depends only on one instance of the information used to cause that effect. For example, with the *CD* feature, the effect that the divert has on a request

$$x \ request \ y$$

depends only on what y is mapped to by the *diversion* function, and is independent of what any other phone is mapped to by *diversion*.

9 Concluding

Our approach to interaction analysis in Z is not systematic and does not guarantee the absence of undesirable interactions. Nonetheless, it has shown to be useful in helping to tackle the problem of feature interaction. The specification approach used allows us to see quite clearly the relationships within and between services and features. Formal reasoning allows us to analyse some consequences of these relationships. This allows us to make more informed decisions about whether interactions are undesirable and, if so, how they can be tackled. Although the development of the approach was based on the specification of IN-services, it should also be applicable to some other types of services like multi-media services.

A On the Soundness of Call Forwarding Unconditional

The Call-Forwarding-Unconditional (CFU) service allows a subscriber to have all incoming calls to his or her telephone forwarded to another telephone instead. In our version of call forwarding, transitive forwarding is provided; for example, if subscriber A has incoming calls forwarded to B, B has incoming calls forwarded to C, and C is not forwarded, then all incoming calls to A are forwarded to C rather than B. For any subscriber x, y is said to be an *endpoint* of x if x is transitively forwarded to y and y is not forwarded; in the example cited, C is an endpoint of A.

Let S be the set of all subscribers. The information required to provide call forwarding is described by the following schema:

Forwarding

$$forward : S \twoheadrightarrow\!\!\!\!\!\!\shortmid\;\; S$$
$$endpoint : S \leftrightarrow S$$

$$(\forall x : S \bullet \neg (x\ forward^{+}\ x))$$
$$endpoint = \{\ x \mapsto y \mid x\ forward^{*}\ y \land y \notin \operatorname{dom} forward\ \}$$

For relation R on S, we write xRy for $(x \mapsto y) \in R$. For subscribers x and y, $x\ forward\ y$ means that subscriber x has requested that all its incoming calls be forwarded to y; since *forward* is a partial function, x can be forwarded to at most one subscriber. Note also that *forward* is a finite function ($S \twoheadrightarrow\!\!\!\!\!\!\shortmid\;\; S$). We write R^i for i copies of R composed sequentially and R^0 for the identity function on S. The transitive closure of R, written R^+, is defined as $(\bigcup i \geq 1 \bullet R^i)$, while the reflexive transitive closure of R, written R^*, is defined as $R^0 \cup R^+$. For subscribers x and y, $x\ endpoint\ y$ means that y is an endpoint of x. The first invariant of *Forwarding* says that no subscriber should be transitively forwarded to itself. Information about connections, etc, is ignored here. The formulation of *endpoint* is taken from [3].

It is required that each subscriber have one, and only one, endpoint, i.e., *endpoint* should be a total function from S to S. This means that if A calls B, then A may simply be connected to $endpoint(B)$. In this paper, we show that the constraints (invariant) of the schema *Forwarding* are sufficient to ensure that *endpoint* is a total function from S to S. We also show that the initial condition for *Forwarding* satisfies the invariant, and that the operations to set up and cancel forwarding of calls both preserve the invariant.

A.1 Total Function

Firstly, we show that the relation *endpoint* is a partial function.

Definition 1 *Relation F is a partial function if* $\quad xFy \land xFz \ \Rightarrow\ y = z.$

Lemma 1 *If F is a partial function, then F^i is a partial function, for any $i > 0$.*

Proof by induction:
Case 0 : holds, since F^0 is the identity function.
Case i+1 :

$$\begin{array}{lll} & xF^{i+1}y \ \land\ xF^{i+1}z & \\ \Rightarrow & (\exists\, y', z' \bullet xF^{i}y' \ \land\ y'Fy \ \land\ xF^{i}z' \ \land\ z'Fz) & \textit{definition of } F^i \\ \Rightarrow & (\exists\, y', z' \bullet y' = z' \ \land\ y'Fy \ \land\ z'Fz) & \textit{induction hypothesis} \\ \Rightarrow & y = z & F \textit{ partial function.} \end{array}$$

□

Lemma 2 *For partial function F,* $\quad xF^{i}y \ \land\ y \notin \operatorname{dom} F \ \land\ j > i \ \Rightarrow\ x \notin \operatorname{dom} F^{j}.$

Proof by contradiction:

$$\begin{array}{lll} & x \in \operatorname{dom} F^{j} \ \land\ xF^{i}y \ \land\ y \notin \operatorname{dom} F & \\ \Rightarrow & (\exists\, z \bullet xF^{i}z \ \land\ z \in \operatorname{dom} F) \ \land\ xF^{i}y \ \land\ y \notin \operatorname{dom} F & \textit{since } j > i \\ \Rightarrow & (\exists\, z \bullet z = y \ \land\ z \in \operatorname{dom} F \ \land\ y \notin \operatorname{dom} F) & F^i \textit{ partial function (Lemma 1)} \\ \Rightarrow & \textit{false}. & \end{array}$$

□

Theorem 1 *endpoint is a partial function.*

Proof (writing F for *forward*):

$$
\begin{array}{lll}
 & x \ \mathit{endpoint}\ y \ \wedge \ x \ \mathit{endpoint}\ z & \\
\Rightarrow & (\exists\, i \bullet xF^i y \ \wedge \ y \notin \operatorname{dom} F) \ \wedge \ (\exists\, j \bullet xF^j z \ \wedge \ z \notin \operatorname{dom} F) & \textit{definition of endpoint} \\
\Rightarrow & (\exists\, i, j \bullet xF^i y \ \wedge \ y \notin \operatorname{dom} F \ \wedge \ xF^j z \ \wedge \ z \notin \operatorname{dom} F) & \\
\Rightarrow & (\exists\, i, j \bullet xF^i y \ \wedge \ y \notin \operatorname{dom} F \ \wedge \ xF^j y \ \wedge \ i < j & \\
 & \quad \vee\, xF^i y \ \wedge \ xF^j z \ \wedge \ z \notin \operatorname{dom} F \ \wedge \ i > j & \\
 & \quad \vee\, xF^i y \ \wedge \ xF^j z \ \wedge \ i = j) & \\
\Rightarrow & (\exists\, i, j \bullet xF^i y \ \wedge \ xF^j z \ \wedge \ i = j) & \textit{Lemma 2} \\
\Rightarrow & y = z &
\end{array}
$$

□

Next we show that *endpoint* is total.

Definition 2 *Relation R on S is total if* $x \in S \ \Rightarrow \ (\exists\, y : S \bullet xRy)$.

A well-foundedness argument is used to prove that *endpoint* is total. A set WF with irreflevive ordering $<$ is *well-founded* if each non-empty subset of WF has a minimal element under $<$ [1]. Given a well-founded set $(WF, <)$, a predicate ϕ and an expression $E(x)$, the following lemma may be used to establish a predicate ψ:

Lemma 3

$$
\frac{\begin{array}{ll} \text{(i)} & \phi \ \Rightarrow \ E(x) \in WF \\ \text{(ii)} & (\exists\, x \bullet E(x) = e \wedge \phi) \ \Rightarrow \ (\exists\, x \bullet E(x) < e \wedge \phi) \vee \psi \end{array}}{(\exists\, x \bullet \phi) \ \Rightarrow \psi}
\qquad
\left[\begin{array}{l} e \ \textit{not free in}\ \phi \\ x, e \ \textit{not free in}\ \psi \end{array} \right]
$$

Proof: [1] shows that mathematical induction is valid over well-founded sets. Firstly, we show by mathematical induction that $(\exists\, x \bullet E(x) = e \wedge \phi) \ \Rightarrow \ \psi$, each $e \in WF$:
Assume holds for each $e' < e$, then

$$
\begin{array}{lll}
 & (\exists\, x \bullet E(x) = e \wedge \phi) & \\
\Rightarrow & (\exists\, x \bullet E(x) < e \wedge \phi) \vee \psi & \textit{hypothesis}\ \text{(ii)} \\
\Rightarrow & (\exists\, e' < e \bullet (\exists\, x \bullet E(x) = e' \wedge \phi)) \vee \psi & \textit{where}\ e'\ \textit{not free in}\ \psi \\
\Rightarrow & (\exists\, e' < e \bullet \psi) \vee \psi & \textit{induction hypothesis} \\
\Rightarrow & \psi & \textit{since}\ e'\ \textit{not free in}\ \psi.
\end{array}
$$

Now,

$$
\begin{array}{lll}
 & (\exists\, x \bullet \phi) & \\
\Rightarrow & (\exists\, e \in WF \bullet (\exists\, x \bullet E(x) = e \wedge \phi)) & \textit{hypothesis}\ \text{(i)} \\
\Rightarrow & (\exists\, e \in WF \bullet \psi) & \textit{from above} \\
\Rightarrow & \psi & \textit{since}\ e\ \textit{not free in}\ \psi.
\end{array}
$$

□

Let *succ* be a function defined as follows:

$$
\begin{array}{|l}
succ : S \rightarrow (\mathbb{P}\, S) \\
\hline
succ(x) \ = \ \{\ y \mid x \ \mathit{forward}^+ \ y \ \}
\end{array}
$$

For any phone x, $succ(x)$ is the set of transitive successors of x under the forwarding function. Assume that *forward* satisfies the invariant of schema *Forwarding*.

Lemma 4 *For any* $x \in S$, $\ succ(x)$ *is finite.*

Proof: It is easy to see that $succ(x) \subset \operatorname{ran} \mathit{forward}$. Now, since *forward* is finite, $\operatorname{ran} \mathit{forward}$ is finite, and thus $succ(x)$ is finite.
□

Lemma 5 *The set* $\{ \ succ(x) \mid x \in S \ \}$ *with the strict-subset ordering* $\subset$ *is well-founded.*

Proof: A descending chain from $(S, \subset)$ is a (finite or infinite) sequence s_i, $i \geq 0$, from S, such that $s_{i+1} \subset s_i$, each $i \geq 0$. [1] shows that a set is well-founded if, and only if, each descending chain from that set is finite. If $s_i, i \geq 0$, is some infinite descending chain from $\{ \ succ(x) \mid x \in S \ \}$ under $\subset$, then some s_i must be infinite. But this is impossible by Lemma 4.
□

If x is forwarded to y, then x has more successors than y:

Lemma 6 $x \ forward \ y \ \Rightarrow \ succ(x) \supset succ(y)$.

Proof (writing F for *forward*):

$$\begin{array}{rcll}
succ(y) & = & \{ \ z \mid yF^+z \ \} & \textit{definition of succ} \\
& \subset & \{y\} \cup \{ \ z \mid yF^+z \ \} & \textit{since} \ \neg \ yF^+y \\
& \subseteq & \{ \ z \mid xFz \ \} \cup \{ \ z \mid yF^+z \ \} & \textit{since} \ xFy \\
& \subseteq & \{ \ z \mid xFz \ \} \cup \{ \ z \mid xF^+z \ \} & xFy \wedge yF^+z \ \Rightarrow \ xF^+z \\
& = & \{ \ z \mid xF^+z \ \} & xFz \ \Rightarrow \ xF^+z \\
& = & succ(x) & \textit{definition of succ.}
\end{array}$$

□

Theorem 2 *endpoint is total.*

Proof: firstly,

$$\begin{array}{cll}
& (\exists \, y \bullet xF^*y \wedge succ(y) = e) & \\
\Rightarrow & (\exists \, y \bullet xF^*y \wedge succ(y) = e \wedge y \in \mathrm{dom}\, F) & \\
& \vee \, (\exists \, y \bullet xF^*y \wedge y \notin \mathrm{dom}\, F) & \\
\Rightarrow & (\exists \, y, z \bullet xF^*y \wedge yFz \wedge succ(y) = e) & y \in \mathrm{dom}\, F \\
& \vee \, (\exists \, y \bullet xF^*y \wedge y \notin \mathrm{dom}\, F) & \\
\Rightarrow & (\exists \, y, z \bullet xF^*y \wedge yFz \wedge succ(y) = e \wedge succ(z) \subset succ(y)) & \textit{Lemma 6} \\
& \vee \, (\exists \, y \bullet xF^*y \wedge y \notin \mathrm{dom}\, F) & \\
\Rightarrow & (\exists \, z \bullet xF^*z \wedge succ(z) \subset e) & xF^*y \wedge yFz \ \Rightarrow \ xF^*z \\
& \vee \, (\exists \, y \bullet xF^*y \wedge y \notin \mathrm{dom}\, F) &
\end{array}$$

Then

$$\begin{array}{rcll}
x \in S & \Rightarrow & (\exists \, y \bullet xF^*y) & \\
& \Rightarrow & (\exists \, y \bullet xF^*y \wedge y \notin \mathrm{dom}\, F) & \textit{Lemma 3 and above} \\
& \Rightarrow & (\exists \, y \bullet x \ endpoint \ y) & \textit{definition of endpoint}
\end{array}$$

□

A.2 Initialisation

Initially the forwarding table should be empty:

```
┌─ InitForwarding ──────────────────────
│ Forwarding
├──────────
│ forward = {}
└───────────────────────────────────────
```

It is easy to see that *InitForwarding* satisfies the invariant of *Forwarding*: $forward = \{\}$ means that *forward* is trivially finite; It also means that $forward^+ = \{\}$, thus $forward^+$ contains no transitive loops.

A.3 Operations

The operation to update the forwarding table so that all calls to a subscriber a are forwarded to b is defined by the schema *Update*. The operation is only valid if $\neg\, b\ forward^*\ a$ (which implies $a \neq b$):

$$
\begin{array}{|l}
\hline
\textit{Update} \\
\Delta \textit{Forwarding} \\
a, b : S \\
\hline
\neg\ b\ \textit{forward}^*\ a \\
\textit{forward}' = \textit{forward} \oplus \{a \mapsto b\} \\
\hline
\end{array}
$$

Write F for *forward* and let $G \mathrel{\widehat=} F \oplus \{a \mapsto b\}$. ($F \oplus \{a \mapsto b\}$ represents F overridden by $\{a \mapsto b\}$.)

Lemma 7 $xGy \ \Rightarrow\ (x = a \wedge y = b) \vee xFy.$

Proof: follows easily from definition of $\oplus$ operator.
□

Lemma 8 *If* $\neg\, bF^*a$, *then, for each* $i \geq 1$,

$$xG^iy \ \Rightarrow\ xF^+y \vee (xF^*a \wedge bF^*y).$$

Proof by induction:
Case 1 :

$$
\begin{array}{rll}
xG^1y & \Rightarrow\ (x = a \wedge y = b) \vee xFy & \textit{Lemma 7} \\
& \Rightarrow\ (xF^*a \wedge yF^*b) \vee (xF^+y) & \textit{Definition of } F^*, F^+.
\end{array}
$$

Case $i + 1$:

$$
\begin{array}{rll}
& xG^{i+1}y & \\
\Rightarrow & (\exists\, z \bullet xGz \wedge zG^iy) & \\
\Rightarrow & (\exists\, z \bullet xGz \wedge zF^+y) & \\
& \vee (\exists\, z \bullet xGz \wedge zF^*a \wedge bF^*y) & \textit{induction hypothesis} \\
\Rightarrow & (\exists\, z \bullet x = a \wedge z = b \wedge zF^+y) & \\
& \vee (\exists\, z \bullet xFz \wedge zF^+y) & \\
& \vee (\exists\, z \bullet x = a \wedge z = b \wedge zF^*a \wedge bF^*y) & \\
& \vee (\exists\, z \bullet xFz \wedge zF^*a \wedge bF^*y) & \textit{Lemma 7} \\
\Rightarrow & (x = a \wedge bF^+y)\ \vee\ xF^+y\ \vee\ bF^*a\ \vee\ (xF^*a \wedge bF^*y) & \\
\Rightarrow & (xF^*a \wedge bF^*y)\ \vee\ xF^+y\ \vee\ \textit{false}\ \vee\ (xF^*a \wedge bF^*y) & \textit{since } \neg\, bF^*a \\
\Rightarrow & (xF^*a \wedge bF^*y)\ \vee\ xF^+y &
\end{array}
$$

□

Lemma 9 *If* $\neg\, bF^*a$, *then,*

$$xG^+y \ \Rightarrow\ xF^+y \vee (xF^*a \wedge bF^*y).$$

Proof: follows from Lemma 8.
□

Theorem 3 *If* $\neg\, bF^*a$, *then* $\quad \neg\, xF^+x \ \Rightarrow\ \neg\, xG^+x.$

Proof:

$$
\begin{array}{rll}
& xG^+x & \\
\Rightarrow & xF^+x \vee (xF^*a \wedge bF^*x) & \textit{Lemma 9} \\
\Rightarrow & xF^+x \vee (bF^*x \wedge xF^*a) & \\
\Rightarrow & xF^+x \vee bF^*a & \\
\Rightarrow & xF^+x & \textit{since } \neg\, bF^*a
\end{array}
$$

□

Using Theorem 3, it is easy to show that *Update* preserves the invariant of *Forwarding*: the theorem guarantees that if there are no transitive loops in *forward* initially, and there is no path from b to a, then there are no transitive loops in the updated value of *forward*.

The operation to cancel forwarding for a subscriber a is defined by the schema *Cancel*:

$$\begin{array}{|l}
\textit{Cancel} \\
\hline
\Delta \textit{Forwarding} \\
a : S \\
\hline
\textit{forward}' = \{a\} \mathbin{⩤} \textit{forward} \\
\hline
\end{array}$$

Let $H = \{a\} ⩤ F$. ($\{a\} ⩤ F$ restricts the domain of F to those elements not in $\{a\}$.)

Lemma 10 $xHy \;\Rightarrow\; xFy.$

Proof: Follows from definition of the $⩤$ operator.
□

Lemma 11 $xH^iy \;\Rightarrow\; xF^+y$, *for each* $i \geq 0$.

Proof: by induction using Lemma 10.
□

Lemma 12 $xH^+y \;\Rightarrow\; xF^+y.$

Proof: follows from Lemma 11.
□

Theorem 4 $\neg\, xF^+x \;\Rightarrow\; \neg\, xH^+x.$

Proof: follows from Lemma 12.
□

Using Theorem 4, it can easily be shown that *Cancel* preserves the invariant of *Forwarding*: the theorem says that if there are no transitive loops in *forward* before the canceling, then there are no transitive loops afterwards.

B Glossary of Symbols

Logic

$\wedge, \vee$	logical-and, logical-or
$\neg$	negation
$\Rightarrow$	logical implication
$(\exists\, x \bullet P)$	exists an x such that P
$(\forall\, x \bullet P)$	forall x, P holds

Sets

$\{\cdots\}$	set delimiters
$\in, \notin$	set membership, non-membership
$\cup, \cap$	set union, set intersection
$\setminus$	set subtraction
$\mathbb{P}\, S$	powerset of S
$\mathbb{P}_1\, S$	$\mathbb{P}\, S$ excluding the emptyset
$\mathbb{N}$	set of natural numbers
disjoint S	no two sets of S have elements in common
SS partition S	SS is an exact partitioning of the elements of S
$S \subseteq T$	S is a subset of T
$S \times T$	cartesian product of S and T

Relations and Functions

$S \leftrightarrow T$	set of relations from S to T
$S \nrightarrow T$	set of partial functions from S to T
$S \nrightarrow_{\mathrm{i}} T$	set of partial injective functions from S to T
$x \mapsto y$	mapping of element x to element y
$f(x)$	application of function f to element x
f^{-1}	inverse of function f
$\operatorname{dom} f, \operatorname{ran} f$	domain of f, range of f
ID_S	identity function on set S
$f \oplus g$	function f overridden by function g
$S \lhd f$	function f with all elements in S removed from its domain
R^+	transitive closure of R
R^*	reflexive transitive closure of R

C Basic Connection

This is simply the basic voice-telephony service (POTS). A user may have at most one two-way connection with some other user. A caller requests a connection by dialling the callee's number. If the callee accepts the request, then the connection is established. If a callee is busy (i.e., involved in a connection or requesting a connection) then the caller gets a busy signal. The caller gets an error tone if it dials an invalid callee. A connection can be cleared by either the caller or the callee.

BEGIN BC

Actions

$$
\begin{array}{ll}
Dial_x & \textbf{for } x \in Phone \\
DialInd_x & \textbf{for } x \in Phone \\
DialResp_x & \textbf{for } x \in Phone \\
Clear_x & \textbf{for } x \in Phone \\
ClearInd_x & \textbf{for } x \in Phone \\
AddSubscriber & \\
RemoveSubscriber &
\end{array}
$$

$[\ Phone\ ]$

VAR

$$
\begin{array}{l}
subscribers : \mathbb{P}\, Phone \\
free : \mathbb{P}\, Phone \\
request : Phone \pfun Phone \\
response : Phone \pinj Phone \\
connected : Phone \pinj Phone \\
clear\_ind : \mathbb{P}\, Phone \\
\hline
\langle free,\ \mathrm{dom}\, request,\ \mathrm{dom}\, response,\ \mathrm{ran}\, response, \\
\mathrm{dom}\, connected,\ \mathrm{ran}\, connected,\ clear\_ind \rangle \quad \mathrm{partition}\ subscribers
\end{array}
$$

INIT

$$
subscribers = \{\}
$$

Dial

$$
\begin{array}{l}
x : Phone \\
\Delta(free, request) \\
y? : Phone \\
\hline
x \in free \\
request' = request \cup \{x \mapsto y?\} \\
free' = free \setminus \{x\}
\end{array}
$$

DialInd

$$
\begin{array}{l}
x : Phone \\
\Delta(request, response) \\
y! : Phone \\
\hline
y!\ request\ x \\
x \in free \\
request' = \{y!\} \ndres request \\
response' = response \cup \{y! \mapsto x\}
\end{array}
$$

$RespType \mathrel{\widehat{=}} ok \mid busy \mid error$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DialRespOk}} \\
\hline
x : Phone \\
\Delta(response, connected) \\
y! : Phone \\
r! : RespType \\
\hline
x \ response \ y! \\
r! = ok \\
response' = \{x\} \ntriangleleft response \\
connected' = connected \cup \{x \mapsto y!\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DialRespBusy}} \\
\hline
x : Phone \\
\Delta(request, free) \\
y! : Phone \\
r! : RespType \\
\hline
x \ request \ y! \\
y! \in subscribers \\
y! \notin free \\
r! = busy \\
request' = \{x\} \ntriangleleft request \\
free' = free \cup \{x\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DialRespError}} \\
\hline
x : Phone \\
\Delta(request, free) \\
y! : Phone \\
r! : RespType \\
\hline
x \ request \ y! \\
y! \notin subscribers \\
r! = error \\
request' = \{x\} \ntriangleleft request \\
free' = free \cup \{x\} \\
\hline
\end{array}
$$

$$DialResp \ \widehat{=} \ DialRespOk \lor DialRespBusy \lor DialRespError$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Clear0}} \\
\hline
x : Phone \\
\Delta(request, free) \\
\hline
x \in \mathrm{dom}\, request \\
request' = \{x\} \ntriangleleft request \\
free' = free \cup \{x\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Clear1}} \\
\hline
x : Phone \\
\Delta(response, clear\_ind, free) \\
\hline
x \in \mathrm{dom}\, response \\
response' = \{x\} \ntriangleleft response \\
clear\_ind' = clear\_ind \cup \{y\} \\
free' = free \cup \{x\} \\
\qquad \textbf{where } y = response(x) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Clear2} \\
x : \textit{Phone} \\
\Delta(\textit{response}, \textit{clear\_ind}, \textit{free}) \\
\hline
x \in \operatorname{ran} \textit{response} \\
\textit{response}' = \{y\} \ntriangleleft \textit{response} \\
\textit{clear\_ind}' = \textit{clear\_ind} \cup \{y\} \\
\textit{free}' = \textit{free} \cup \{x\} \\
\qquad \textbf{where } y = \textit{response}^{-1}(x) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Clear3} \\
x : \textit{Phone} \\
\Delta(\textit{connected}, \textit{clear\_ind}, \textit{free}) \\
\hline
x \in \operatorname{dom} \textit{connected} \\
\textit{connected}' = \{x\} \ntriangleleft \textit{connected} \\
\textit{clear\_ind}' = \textit{clear\_ind} \cup \{y\} \\
\textit{free}' = \textit{free} \cup \{x\} \\
\qquad \textbf{where } y = \textit{connected}(x) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Clear4} \\
x : \textit{Phone} \\
\Delta(\textit{connected}, \textit{clear\_ind}, \textit{free}) \\
\hline
x \in \operatorname{ran} \textit{connected} \\
\textit{connected}' = \{y\} \ntriangleleft \textit{connected} \\
\textit{clear\_ind}' = \textit{clear\_ind} \cup \{y\} \\
\textit{free}' = \textit{free} \cup \{x\} \\
\qquad \textbf{where } y = \textit{connected}^{-1}(x) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Clear5} \\
x : \textit{Phone} \\
\Delta(\textit{clear\_ind}, \textit{free}) \\
\hline
x \in \textit{clear\_ind} \\
\textit{clear\_ind}' = \textit{clear\_ind} \setminus \{x\} \\
\textit{free}' = \textit{free} \cup \{x\} \\
\hline
\end{array}
$$

$$\textit{Clear} \;\widehat{=}\; \textit{Clear0} \vee \textit{Clear1} \vee \ldots \vee \textit{Clear5}$$

$$
\begin{array}{|l}
\hline
\textit{ClearInd} \\
x : \textit{Phone} \\
\Delta(\textit{clear\_ind}, \textit{free}) \\
\hline
x \in \textit{clear\_ind} \\
\textit{clear\_ind}' = \textit{clear\_ind} \setminus \{x\} \\
\textit{free}' = \textit{free} \cup \{x\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{AddSubscriber} \\
\Delta(\textit{subscribers}, \textit{free}) \\
x? : \textit{Phone} \\
\hline
x? \notin \textit{subscribers} \\
\textit{subscribers}' = \textit{subscribers} \cup \{x?\} \\
\textit{free}' = \textit{free} \cup \{x?\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{RemoveSubscriber} \\
\Delta(subscribers, free) \\
x? : Phone \\
\hline
x? \in subscribers \\
x? \in free \\
subscribers' = subscribers \setminus \{x?\} \\
free' = free \setminus \{x?\} \\
\hline
\end{array}
$$

END BC

D Follow-Me Diversion

This feature allows a user to have all its incoming calls diverted to a different phone. A user makes a diversion request from the phone to which it wants all incoming calls diverted. Once user A has made a diversion request from phone B, then all further incoming calls to A are diverted to B until user A cancels the diversion, or requests a diversion from some other phone. A user can only invoke call diversion if it is a subscriber to the Follow-Me-Diversion service. For security purposes, each subscriber to the service has a PIN number which must be used each time a diversion is requested. A subscriber is allowed to change its PIN number and to query where it is currently diverted to.

BEGIN *FMD*

BC

Actions $SubscribeFMD$
$UnsubscribeFMD$
$OrderFMD_x$ **for** $x \in Phone$
$CancelFMD_x$ **for** $x \in Phone$
$QueryFMD_x$ **for** $x \in Phone$
$ChangePIN_x$ **for** $x \in Phone$

Internal Actions $Divert_x$ **for** $x \in Phone$

$[\ PIN\ ]$

VAR

$subscribersFMD : \mathbb{P}\ Phone$
$diversion : Phone \nrightarrow Phone$
$pin : Phone \nrightarrow PIN$
$divchecked : \mathbb{P}\ Phone$

$subscribersFMD \subseteq subscribers$

$\operatorname{dom} diversion \subseteq subscribersFMD$

$subscribersFMD = \operatorname{dom} pin$

$divchecked \subseteq \operatorname{dom} request$

Divert0

$x : Phone$
$\Delta(request, divchecked)$

$(\exists\, y, z : Phone \bullet$
$\quad x \ request \ y$
$\quad x \notin divchecked$
$\quad y \ diversion \ z$
$\quad request' = request \oplus \{x \mapsto z\}$
$\quad divchecked' = divchecked \cup \{x\}\ )$

Divert1

$x : Phone$
$\Delta(divchecked)$

$(\exists\, y : Phone \bullet$
$\quad x \ request \ y$
$\quad x \notin divchecked$
$\quad y \notin \operatorname{dom} diversion$
$\quad divchecked' = divchecked \cup \{x\}\ )$

$Divert \mathrel{\widehat{=}} Divert0 \lor Divert1$

$$
\begin{array}{|l}
\hline
\land\text{-extend } DialInd \\
x : Phone \\
\Delta(divchecked) \\
y! : Phone \\
\hline
y! \in divchecked \\
divchecked' = divchecked \setminus \{y!\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\land\text{-extend } DialRespBusy \\
x : Phone \\
\Delta(divchecked) \\
y! : Phone \\
\hline
x \in divchecked \\
divchecked' = divchecked \setminus \{x\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\land\text{-extend } DialRespError \\
x : Phone \\
\Delta(divchecked) \\
y! : Phone \\
\hline
x \in divchecked \\
divchecked' = divchecked \setminus \{x\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\land\text{-extend } Clear0 \\
x : Phone \\
\Delta(divchecked) \\
\hline
divchecked' = divchecked \setminus \{x\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
SubscribeFMD \\
\Delta(subscribersFMD, pin) \\
x? : Phone \\
p? : PIN \\
\hline
x? \in subscribers \ \land \ x? \in free \\
x? \notin subscribersFMD \\
subscribersFMD' = subscribersFMD \cup \{x?\} \\
pin' = pin \cup \{x? \mapsto p?\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
UnsubscribeFMD \\
\Delta(subscribersFMD, pin, diversion) \\
x? : Phone \\
\hline
x? \in subscribersFMD \ \land \ x? \in free \\
subscribersFMD' = subscribersFMD \setminus \{x?\} \\
pin' = \{x?\} \mathbin{\lhd\!\!\!-} pin \\
diversion' = \{x?\} \mathbin{\lhd\!\!\!-} diversion \\
\hline
\end{array}
$$

$$\begin{array}{l}
\textbf{OrderFMD} \\
\hline
x : Phone \\
\Delta(diversion) \\
y? : Phone \\
p? : PIN \\
\hline
x \in free \\
y? \in subscribersFMD \\
p? = pin(y?) \\
diversion' = diversion \oplus \{y? \mapsto x\}
\end{array}$$

$$\begin{array}{l}
\textbf{CancelFMD} \\
\hline
x : Phone \\
\Delta(diversion) \\
y? : Phone \\
p? : PIN \\
\hline
x \in free \\
y? \in subscribersFMD \\
p? = pin(y?) \\
diversion' = \{y?\} \lhd diversion
\end{array}$$

$$\begin{array}{l}
\textbf{QueryFMD} \\
\hline
x : Phone \\
\Delta() \\
y? : Phone \\
p? : PIN \\
r! : \{diverted, notdiverted\} \\
z! : Phone \\
\hline
x \in free \\
y \in subscribersFMD \\
p? = pin(y?) \\
y? \in \mathrm{dom}\, diversion \Rightarrow \quad r! = diverted \\
\qquad\qquad\qquad\qquad\qquad z! = diversion(y?) \\
y? \notin \mathrm{dom}\, diversion \Rightarrow r! = notdiverted
\end{array}$$

$$\begin{array}{l}
\textbf{ChangePIN} \\
\hline
x : Phone \\
\Delta(pin) \\
y? : Phone \\
p?, q? : PIN \\
\hline
x \in free \\
y? \in subscribersFMD \\
p? = pin(y?) \\
pin' = pin \oplus \{y? \mapsto q?\}
\end{array}$$

$$\begin{array}{l}
\wedge\text{-extend } \textbf{RemoveSubscriber} \\
\hline
\Delta(subscribersFMD, pin, diversion) \\
x? : Phone \\
\hline
subscribersFMD' = subscribersFMD \setminus \{x?\} \\
pin' = \{x?\} \lhd pin \\
diversion' = \{x?\} \lhd diversion
\end{array}$$

END *FMD*

E Call Diversion on Busy

This allows a user to specify a phone to which incoming calls should be diverted whenever the user is busy. A user may only issue a diversion request from its home phone so that PIN numbers are unnecessary. A user may cancel and query diversion requests.

BEGIN *CDB*

BC

Actions $OrderCDB_x$ **for** $x \in Phone$
$CancelCDB_x$ **for** $x \in Phone$
$QueryCDB_x$ **for** $x \in Phone$

Internal Actions $BusyDivert_x$ **for** $x \in Phone$

VAR

$busydiversion : Phone \rightarrowtail\!\!\!\!\!\!+ \ Phone$
$busydivchecked : \mathbb{P}\, Phone$

$\mathrm{dom}\, busydiversion \subseteq subscribers$

$busydivchecked \subseteq \mathrm{dom}\, request$

$BusyDivert0$

$x : Phone$
$\Delta(request, busydivchecked)$

$(\exists\, y, z : Phone \bullet$
$\quad x \ request \ y$
$\quad x \notin busydivchecked$
$\quad y \notin free$
$\quad y \ busydiversion \ z$
$\quad request' = request \oplus \{x \mapsto z\}$
$\quad busydivchecked' = busydivchecked \cup \{x\}\ )$

$BusyDivert1$

$x : Phone$
$\Delta(busydivchecked)$

$(\exists\, y : Phone \bullet$
$\quad x \ request \ y$
$\quad x \notin busydivchecked$
$\quad (y \notin \mathrm{dom}\, busydiversion \ \lor \ x \in free)$
$\quad busydivchecked' = busydivchecked \cup \{x\}\ )$

$\wedge$-extend $DialInd$

$x : Phone$
$\Delta(busydivchecked)$
$y! : Phone$

$y! \in busydivchecked$
$busydivchecked' = busydivchecked \setminus \{y!\}$

$\wedge$-extend *DialRespBusy*

$$x : Phone$$
$$\Delta(busydivchecked)$$
$$y! : Phone$$

$$x \in busydivchecked$$
$$busydivchecked' = busydivchecked \setminus \{x\}$$

$\wedge$-extend *DialRespError*

$$x : Phone$$
$$\Delta(busydivchecked)$$
$$y! : Phone$$

$$x \in busydivchecked$$
$$busydivchecked' = busydivchecked \setminus \{x\}$$

$\wedge$-extend *Clear0*

$$x : Phone$$
$$\Delta(busydivchecked)$$

$$busydivchecked' = busydivchecked \setminus \{x\}$$

OrderCDB

$$x : Phone$$
$$\Delta(busydiversion)$$
$$y? : Phone$$

$$x \in free$$
$$busydiversion' = busydiversion \oplus \{x \mapsto y?\}$$

CancelCDB

$$x : Phone$$
$$\Delta(busydiversion)$$

$$x \in free$$
$$busydiversion' = \{x\} \ntriangleleft busydiversion$$

QueryCBD

$$x : Phone$$
$$\Delta()$$
$$r! : \{diverted, notdiverted\}$$
$$z! : Phone$$

$$x \in free$$
$$x \in \mathrm{dom}\, busydiversion \Rightarrow \begin{array}{l} r! = diverted \\ z! = busydiversion(x) \end{array}$$
$$x \notin \mathrm{dom}\, busydiversion \Rightarrow r! = notdiverted$$

$\wedge$-extend *RemoveSubscriber*

$$\Delta(busydiversion)$$
$$x? : Phone$$

$$busydiversion' = \{x?\} \ntriangleleft busydiversion$$

END *CDB*

F Call Screening Terminating

This allows a user to set up a blacklist of subscribers so that all incoming calls from a blacklisted subscriber are rejected and instead that calling subscriber gets a busy signal. A user may add a subscriber to its screening list, remove a subscriber from its list, or check if it has blacklisted a particular subscriber.

BEGIN CS

BC

Actions $AddScreen_x$ **for** $x \in Phone$
$RemoveScreen_x$ **for** $x \in Phone$
$QueryScreen_x$ **for** $x \in Phone$

VAR

$screen : Phone \rightarrow\!\!\!\!\!\!+ (\mathbb{P}\ Phone)$

$\mathrm{dom}\ screen = subscribers$

$\wedge$-extend $DialInd$

$x : Phone$
$\Delta()$
$y! : Phone$

$y! \notin screen(x)$

$\vee$-extend $DialRespBusy$

$x : Phone$
$\Delta(request, free)$
$y! : Phone$
$r! : RespType$

$x\ request\ y!$
$y! \in subscribers$
$x \in screen(y!)$
$r! = busy$
$request' = \{x\} \lhd request$
$free' = free \cup \{x\}$

$AddScreen$

$x : Phone$
$\Delta(screen)$
$y? : Phone$

$x \in free$
$screen' = screen \oplus \{\ x \mapsto (screen(x) \cup \{y?\})\ \}$

$RemoveScreen$

$x : Phone$
$\Delta(screen)$
$y? : Phone$

$x \in free$
$screen' = screen \oplus \{\ x \mapsto (screen(x) \setminus \{y?\})\ \}$

$$
\begin{array}{|l}
\hline
\textit{QueryScreen} \\
x : \mathit{Phone} \\
\Delta() \\
y? : \mathit{Phone} \\
r! : \{\mathit{screened}, \mathit{notscreened}\} \\
\hline
x \in \mathit{free} \\
y? \in \mathit{screen}(x) \;\Rightarrow\; r! = \mathit{screened} \\
y? \notin \mathit{screen}(x) \;\Rightarrow\; r! = \mathit{notscreened} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\wedge\text{-extend } \mathit{AddSubscriber} \\
\Delta(\mathit{screen}) \\
x? : \mathit{Phone} \\
\hline
\mathit{screen}' = \mathit{screen} \cup \{\ x? \mapsto \{\}\ \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\wedge\text{-extend } \mathit{RemoveSubscriber} \\
\Delta(\mathit{screen}) \\
x? : \mathit{Phone} \\
\hline
\mathit{screen}' = \{x?\} \lhd \mathit{screen} \\
\hline
\end{array}
$$

END *CS*

G Call Forwarding Unconditional

This is similar to Follow-Me-Diversion and Call-Diversion-on-Busy, except that the diversion is transitive rather than single-level[3]. This means that if, for example, A is diverted to B, and B is diverted to C, then all incoming calls to A are diverted to C. In the case of Follow-Me-Diversion and Call-Diversion-on-Busy, on the other hand, incoming calls would only be diverted as far as B. A user may only request the Call-Forwarding-Unconditional service from its home phone. Again, a user may cancel and query diversions.

BEGIN *CFU*

BC

Actions $OrderCFU_x$ **for** $x \in Phone$
$CancelCFU_x$ **for** $x \in Phone$
$QueryCFU_x$ **for** $x \in Phone$

Internal Actions $Forward_x$ **for** $x \in Phone$

VAR

$$
\begin{array}{l}
forward : Phone \rightarrowtail\!\!\!\!\!\!\!\!\;\; Phone \\
endpoint : Phone \leftrightarrow Phone \\
forwardchecked : \mathbb{P}\, Phone \\
\hline
forward \subseteq subscribers \times subscribers \\
(\forall\, x : Phone \bullet \neg\,(x\ forward^{+}\ x)) \\
endpoint = \{\ x, y : Phone \mid x\ forward^{*}\ y \wedge y \notin \operatorname{dom} forward\ \} \\
forwardchecked \subseteq \operatorname{dom} request
\end{array}
$$

Forward

$$
\begin{array}{l}
x : Phone \\
\Delta(request, forwardchecked) \\
\hline
(\exists\, y : Phone \bullet \\
\quad x\ request\ y \\
\quad x \notin forwardchecked \\
\quad request' = request \oplus \{\ x \mapsto endpoint(y)\ \} \\
\quad forwardchecked' = forwardchecked \cup \{x\}\ )
\end{array}
$$

$\wedge$-extend *DialInd*

$$
\begin{array}{l}
x : Phone \\
\Delta(forwardchecked) \\
y! : Phone \\
\hline
y! \in forwardchecked \\
forwardchecked' = forwardchecked \setminus \{y!\}
\end{array}
$$

$\wedge$-extend *DialRespBusy*

$$
\begin{array}{l}
x : Phone \\
\Delta(forwardchecked) \\
y! : Phone \\
\hline
x \in forwardchecked \\
forwardvchecked' = forwardchecked \setminus \{x\}
\end{array}
$$

[3] Sometimes the interpretation of Call-Forwarding-Unconditional is such that the diversion is single-level rather than transitive.

$\wedge$-extend *DialRespError*

$$
\begin{array}{l}
x : Phone \\
\Delta(forwardchecked) \\
y! : Phone \\
\hline
x \in forwardchecked \\
forwardchecked' = forwardchecked \setminus \{x\}
\end{array}
$$

$\wedge$-extend *Clear0*

$$
\begin{array}{l}
x : Phone \\
\Delta(forwardchecked) \\
\hline
forwardchecked' = forwardchecked \setminus \{x\}
\end{array}
$$

OrderCFU

$$
\begin{array}{l}
x : Phone \\
\Delta(forward, endpoint) \\
y? : Phone \\
\hline
x \in free \\
y? \in subscribers \\
\neg\,(y?\ forward^*\ x) \\
forward' = forward \oplus \{x \mapsto y?\}
\end{array}
$$

CancelCFU

$$
\begin{array}{l}
x : Phone \\
\Delta(forward, endpoint) \\
\hline
x \in free \\
forward' = \{x\} \lhd forward
\end{array}
$$

QueryCFU

$$
\begin{array}{l}
x : Phone \\
\Delta() \\
r! : \{forwarded, notforwarded\} \\
y!, z! : Phone \\
\hline
x \in free \\
x \in \operatorname{dom} forward \Rightarrow \quad \begin{array}{l} r! = forwarded \\ y! = forward(x) \\ z! = endpoint(x) \end{array} \\
x \notin \operatorname{dom} forward \Rightarrow r! = notforwarded
\end{array}
$$

$\wedge$-extend *RemoveSubscriber*

$$
\begin{array}{l}
\Delta(forward, endpoint) \\
x? : Phone \\
\hline
forward' = \{x?\} \lhd forward
\end{array}
$$

END *CFU*

References

[1] E.W. Dijkstra and C. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[2] Roger Duke, Paul King, Gordon Rose, and Graeme Smith. The Object-Z specification language, version 1. Technical Report 91-1, Department of Computer Science, University of Queensland, Australia, May 1991.

[3] A. Lee and G. Smith. Using formal techniques to detect service interactions. University of Queensland, January 1991.

[4] J.M. Spivey. *The Z Notation.* Prentice Hall, 1989.