Model Checking the Distributed Feature Composition (DFC) Architecture in SPIN

Alma Juarez Dominguez, Wenceslas Godard, and Nancy A. Day

Technical Report CS-2004-40
School of Computer Science
University of Waterloo

April 30, 2004

Abstract

In this report, we describe our effort to represent and model check the Distributed Feature Composition (DFC) architecture in Spin. DFC is an architecture for describing telecommunications services. Spin is a model checker for models described in the language Promela.

NOTE: An extended and updated version of this work can be found in [7].

1 Introduction

Distributed Feature Composition (DFC) [5] is an architecture for coordinating telecommunication features based on a pipe-and-filter model. A call from a caller to a callee is handled by each feature in a sequential manner. The architecture is a distributed system with each feature running as a process and communicating with its neighbours using signals passed along communication channels. DFC supports modular development, where modules that are independently implemented can be composed. Our goal in this work was to create a model of DFC to (1) learn how DFC works, and (2) explore options for verifying properties of DFC configurations.

We chose to model DFC and its features in Promela, the modelling language of the Spin model checker [4]. Promela was chosen because it is suitable for describing distributed, concurrent and evolving systems, and the underlying process coordination and communication mechanism in Spin matches that of DFC: processes are interleaved and they communicate across channels using signals. Promela's channels can be queues, or we can assume communication is instantaneous using rendezvous (channels of size 0).

In the following sections, we describe our model of DFC and of DFC features. Note that our model represents our understanding of DFC based on reading the literature (early papers describing the architecture [5, 13], DFC modifications [12, 11], and in particular the DFC Manual [6]), and does not constitute an official description of DFC. The DFC manual contains Promela descriptions for the DFC caller and callee port protocol behaviour. In our model, we create feature processes dynamically as needed. Section 2 is a brief description of DFC. In Section 3, we provide some background on Promela and Spin to help readers understand our model. We give an overview of the basic communication of DFC as represented in Promela in Section 4, and then describe the caller and callee processes, and several features in Section 6. We show how our model can be used to check properties of DFC configurations in Section 7, and then discuss issues regarding using Spin to model check our model of DFC in Section 8. Finally, we briefly overview related work in Section 9 and conclude in Section 10. The complete Promela model can be found in Appendix A.

2 Distributed Feature Composition (DFC)

In DFC, a **usage** is a dynamic assembly of boxes connected by internal calls as illustrated in Figure 1. **Boxes** are either interface boxes (*e.g.,* caller, callee) or features subscribed to by users (*e.g.,* FB1 could be call forwarding). **Interface boxes** are connected to physical devices to communicate to users (*e.g.,* telephones) or to other networks that use different protocols to exchange information. **Internal calls** are communication channels between **ports** of two different boxes (represented with dark circles in the figure), transmitting signals between boxes in first in–first out (FIFO) order and following the DFC call protocol. There are three phases to the interaction between caller and callee: setup, communication, and teardown.

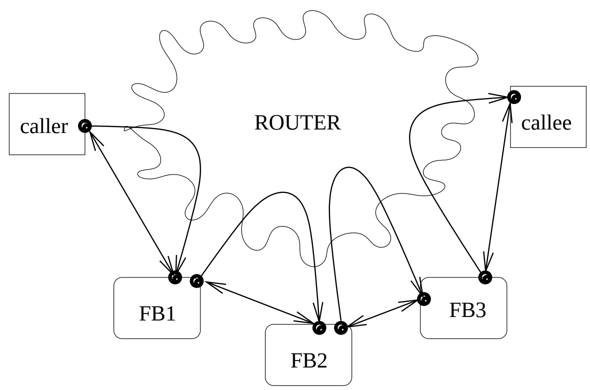

Figure 1: Overview of DFC

Each internal call is setup in a triangular and piecewise manner: a message with a *setup* signal from a box first goes to the router, then the router determines the next box in the usage and sends it a setup message, and finally, a signalling channel is created between the two boxes at either end of the internal call. The router determines the next box in the sequence based on the caller and callee's **subscriptions** and **feature precedence** (the order in which features can occur in a route). When a box receives a setup message from the router, it sends an *upack* message to the calling box along the channel connecting the two boxes. The setup phase continues with the second box sending a setup message to the router to be forwarded to the next box in the usage.

Once a usage from a caller to a callee has been setup, the call proceeds to its communication phase, and then later to its teardown phase. We will provide more details on the signals used by DFC when we describe our PROMELA model of DFC. Usages can be branching structures and involve multiple callees and callers.

Usages are divided into regions. The **source region** involves all the feature boxes subscribed to by the customer who is the caller. The **target region** involves all the feature boxes subscribed to by the customer being called. A usage can have one or more source and target regions.

A **free feature box** is one for which a new instance is generated every time the feature is to be included in a usage. Examples of free features boxes are call forwarding, and call blocking. A **bound feature box** is dedicated to a particular address, and even if it is already used in a usage, when a new usage is created, the same feature box is made part of the new usage. An example of a bound feature box is call waiting because when a customer is involved in a call, all new incoming calls are routed through the same call waiting feature box.

A **feature interaction** occurs when the presence of a box in a usage changes the behaviour of the other boxes in the usage. For example, an upstream box can absorb a signal to which a later box in the usage might react. Another example of a feature interaction is that a box might write or modify fields of the setup message in address translation.

3 SPIN

The SPIN model checker checks descriptions written in PROMELA. SPIN is an explicit-state model checker, and has highly optimized state space representation and reduction techniques for checking LTL properties. SPIN also includes simulation facilities and a graphical interface (XSPIN).

3.1 PROMELA

In PROMELA, the keyword `proctype` begins the definition of a new process. Processes are declared globally. The definition of a process contains the list of formal parameters (if any), and its body starts with declarations of local variables (if any). If the keyword `active` precedes `proctype`, then the process initially runs; otherwise, it is possible to run an instance of a process dynamically at any point in execution using the `run` operator followed by the name of the process (and parameters as needed). There cannot be more than 255 processes running at the same time.

The execution of processes is interleaved. If more than one process can execute, only one of them will be non-deterministically chosen to execute.

Points in the execution of a process can have labels, which can be viewed as state names. Final states can be designated using labels that begin with `end`.

PROMELA 's basic data types include `bit, bool, byte, short, int` and `unsigned`. Symbolic values are created using an `mtype` declaration. These symbols are implicitly associated with natural numbers, so they can be used as array indices. Global variables can be declared to be shared by all processes.

The most common statements used in a process are if-statements and do-loops. The syntax of an **if-statement** is:

```
if
 ::statement
[::statement]*
fi;
```

Each `statement` may start with a *guard* condition, *e.g.,* `x==1`. If no guards are satisfied, then the if statement blocks (*i.e.,* other processes can run) until at least one guard is executable. If more than one guard is satisfied, then one statement is chosen to execute non-deterministically.

The syntax of a **do-loop** is:

```
do
 ::statement
[::statement]*
od;
```

In a do-loop, only one statement in each iteration is executed. The statements may start with a **guard** condition. If more than one guard is satisfied, one statement is chosen to execute non-deterministically. After executing a statement, the process loops back to the beginning of the loop. If no statements are executable, the loop blocks. The loop can be exited using either a `goto` statement that sends the process to a labelled statement or a `break` statement.

Processes communicate via channels. PROMELA supports both asynchronous and synchronous channels. For example, the following is a declaration of a channel `ch1` of capacity 1 that carries messages composed of a pair with the first element of type boolean, and the second element of type byte:

```
chan  ch1 = [1] of {bool,byte};
```

In process A, we could now write: `ch1!true,8` to send the message `(true,8)` on channel `ch1`. Using the statement `ch1?var1,var2`, process B can read the channel and store the received value in two variables var1, and var2 respectively declared as boolean and byte types. Because the channel has capacity 1, if process A sends another message before B retrieves the first message, process A blocks, which might result in the system being in deadlock (no process can take a step). A similar situation can occur if process B tries to read an empty channel.

Rather than using a variable to receive the value of a message passed on a channel, we can force a process to respond only to particular messages. For example, if we declare *setup* as an element of an mtype, and a channel, ch2, as:

```
chan ch2 = [1] of {mtype};
```

then a statement in a process such as ch2?setup is only executed if the message in the channel is a *setup*. Otherwise, the process blocks, without retrieving the message from the channel. The use of the built-in operator eval can lead to similar results. For example, ch2?eval(var3) checks that the value received matches the value stored in a variable var3, otherwise, this statement blocks.

Rendezvous channels are channels of zero capacity. The sending and receiving operations are executed atomically for a rendezvous channel, *i.e.,* no other instruction can be executed in between them. If a process tries to send a message on such a channel when no process is ready to receive the message, the sending process blocks.

The keyword atomic is used to force a sequence of statements to be executed together without any other process interleaving the execution of its statements in between, thereby reducing the number of behaviours in the model. Each statement within an atomic sequence constitutes an individual transition in the underlying state machine, but no other process can interleave its behaviour with transitions in an atomic sequence unless a statement in the sequence blocks. If one of the statements in an atomic group blocks, the group blocks, until the statement can execute. A receiving operation within an atomic sequence may also cause the atomic sequence to block. To avoid confusion about atomic sequences, we adopt the convention that a send or receive statement can only be the last statement of an atomic sequence.

3.2 Simulation and Model Checking

Spin can simulate and model check Promela models. In simulation mode, by default, the non-deterministic choices for which statements execute, and the choices for which processes executes, are resolved randomly; but one can either specify a particular seed value that is used to resolve all the choices (two simulation runs with the same seed value will give exactly the same output), or run the simulation interactively (at each step, the user selects one among all the possible next steps). Non-interactive simulation continues until no more processes can execute.

Spin provides a graphical user interface called XSPIN, which can illustrate a trace of the model's execution generated in simulation or as a counterexample for model checking. The trace is presented as a message sequence chart (MSC) showing the relative ordering of message communication between processes.

In model checking, Spin can check LTL properties using never claims, which are used to specify system behaviour that should never occur, and are given to Spin as the negation of the property to check. Spin can also check application-independent properties such as absence of deadlock. Deadlock means there are invalid end states, *i.e.,* non-final states from which there are no next states.

4 Modelling DFC in Promela

In this section, we describe our model of DFC in Promela. Each line interface and feature box is modelled by one or more Promela processes. The router is also represented as a process for each internal call. Instances of boxes in a usage are *dynamically* created by the router as they are needed in the usage. Instances of the router process are also dynamically created by boxes when needed.

Signals in DFC allow the boxes to communicate and carry out the tasks of the setup and teardown phases. These signals travel on the signalling channels. Our model uses the following notation for DFC protocol signals:

- *setup* creates an internal call via the router.
- *upack* is used to acknowledge a *setup* message.
- *avail* is used to convey that the user is available. In a simple usage, receipt of this signal would generate a ring tone at the caller.
- *unavail* is used to convey that the user is not available. In a simple usage, receipt of this signal would generate a busy tone at the caller.
- *teardown* is used to destroy a usage.
- *downnack* is used to acknowledge a *teardown* message.

All features use the above signals, whose syntax is given next. Specific features may respond to or send additional types of signals. These signals are declared in PROMELA using an `mtype`:

```
mtype {setup, upack, teardown, downack, avail, unavail}
```

For a model with N users, we declare `user0, user1... user(N-1)` as symbols using mtype. These symbolic values are used in the source field in setup messages and as actual parameters to processes to indicate the source and target of the usage. This information is also used to index global arrays representing subscriptions and busy status of users (*e.g.,* `subs_CF[user0 - 1]` is a boolean value that indicates if `user0` subscribes to call forwarding).

Internal calls (between feature boxes, callers, and callees, not with the router processes) are bidirectional, but in SPIN they are modelled using two channels, one for each direction of communication.

```
typedef Com_chan  {
  chan A = [3] of {mtype}
  chan B = [3] of {mtype}
}
```

Because the input channel of one process is the output channel of the other process, we choose the generic names `A` and `B` for the channel and each process must be aware of which part is the input and output for each internal call. The sizes chosen for these channels will be explained later.

From now on, we use the term communication path[1] for a DFC internal call. The model includes a set of communication paths as an array of constant size M. A communication path is dynamically assigned during the creation of a usage by providing an array index to the created process. A boolean array indicates if the corresponding communication path is assigned or free:

```
Com_chan chan_array [M];
bool channel_busy [M];
```

An inline function `channel_busy` is used to return an index of a free communication path. Each time an internal call is made, a counter is incremented and the next value is provided as the communication path to use. This path identifier is sent in the last field of the *setup* message. An **assert** statement in the inline function forces SPIN to report an error if more than M paths are needed. The number of bits needed to hold a path identifier is set by the constant L such that $2^L > M$.

[1] To avoid confusion with the term "call" in the context of a user requesting the creation of a usage, we use the term "communication path" to describe an internal call.

Our model includes data on the active status of a user represented as a globally declared boolean array:

```
bool busy[N];
```

This array indicates whether a user (caller or callee) is currently busy or idle. The caller and callee processes control this status data.

Additional global declarations include:

- Subscription information – boolean arrays indicating whether users subscribe to the implemented features, (*e.g.,* call forwarding uses `subs_CF`). Feature precedence is hard-coded into the router processes in our model (Section 5).
- Call forwarding information – an array (`CF_info`) that contains the numbers to which users wants their incoming calls forwarded.

Next, we describe how the three phases of a DFC usage are represented in our PROMELA model. In the setup phase the usage is created. In the communication phase, data is passed between caller and callee. In the teardown phase, the usage is destroyed.

The constants M (number of channels), N (number of users), and L (the number of bits needed to represent channel indices) are assigned at the beginning of our PROMELA code, shown in Appendix A.

4.1 Setup Phase

Setup messages from a line interface are sent to the router. The function of the router is to receive *setup* messages and to propagate them to the next box of the usage, which is determined based on the feature subscriptions of the caller and callee, and also by the feature precedence.

A setup message in our model has five fields:

- the type of the message (always *setup*)
- the identity of the originating caller of the usage
- the dialled number
- the identity of the destination callee of the usage.
- the identity of the communication path to be used for the internal call between the boxes.

To represent the dialled number, we use the constants `user0, user1,` etc. A dialled number field containing `user0` thus means that the caller has dialled the number of `user0`.

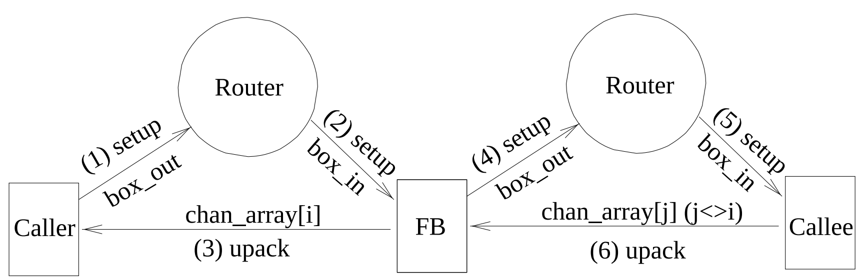

Figure 2: Setup Phase

As illustrated in Figure 2, the setup phase proceeds piecewise:

1. A caller sends a *setup* to a router ("1" in the figure) on channel `box_out`.
2. The router uses the subscription information and feature precedence to determine the next box in the usage and forwards the *setup* to that box ("2" in the figure) on channel `box_in`.
3. The receiving box sends an *upack* message directly back to the caller ("3" in the figure) on the communication path chosen for the internal call (`chan_array`[i]).
4. The box also forwards the setup to another router to continue the creation of the usage if it is not the final box in the usage ("4" in the figure).

For our model, we assume that the routing is carried out correctly. Therefore, we model the communication to and from the router in an atomic sequence using rendezvous channels for `box_out` and `box_in`. No other process can execute during this atomic sequence of actions (processes communicating through rendezvous channels never block) and once completed, the router has reached the end of its code. Therefore, there is never a need to have two router processes in existence at the same time, which means the same channels, `box_out` and `box_in`, can be used for all communication with the routers. This method also reduces the state space of the model.

When a box receives a *setup*, it sets a local variable to the value of the communication path identifier provided in the *setup* message. This variable is used to direct communication on the path between the feature and its neighbour.

4.2 Communication Phase

In the communication phase, data is exchanged between the caller and callee directly (*e.g.,* on the channels labelled `chan_array` in Figure 2). The communication that occurs on the signalling channel connecting the caller and callee of an internal call may be delayed, *i.e.,* a message may not be read immediately after it was sent because of the interleaved execution of processes. Therefore, these channels are not 0-capacity. DFC assumes that messages are read in the same order as they were sent on a particular channel, and Promela 's channels have this behaviour. Messages exchanged during the communication phase have one field for the type of the signal. Possible protocol signals travelling on these channels are *upack, avail, unavail, teardown* and *downnack*.

The signals used to describe the media channels opening or closing (*open*, *close*) are not included in our model, because we are focusing on the protocol signals and properties of the call protocol communication.

Promela cannot handle infinite size channels, therefore we have to choose an appropriate capacity for the communication channels. If boxes behave correctly (*i.e.,* they all follow the protocol), then the maximum sequence of protocol signals that can be written on a channel in an atomic sequence of statements is 3: *upack*, *unavail*, *teardown* indicating that the callee box is busy[2]. If a process tries to write to a channel that is full, the process blocks. Using Spin, we can check for deadlock to ensure that no process will ever block. If no deadlock is possible, then the chosen size of the communication channels is sufficient.

In DFC, data is handled on a channel separate from the signalling channel. Because we are focusing on the protocol signals that travel on the signalling channel, we abstract away the details of the data communication and do not represent the data channel.

4.3 Teardown Phase

In the teardown phase, the usage is destroyed. Similar to the setup phase, the teardown phase is piecewise: a *teardown* is acknowledged by sending a *downack* back to the box that sent the *teardown*, and then propagating the *teardown* to the next box in the usage. Features can alter this behaviour. For example, if a call waiting box receives a *teardown* from the user which is waiting, it will terminate that branch of the usage by sending

[2] An example of where this atomic sequence of actions occurs in in the callee_router_watch process in Figure 7.

a *downack* message, but it does not propagate the *teardown* to the subscriber because the subscriber is involved in another call.

5 Routers

In DFC, the router is responsible for creating or locating the next feature box in the usage and forwarding the setup message to it. The next box in the usage is determined by subscription information and feature precedence. To simplify our model, we assume a static feature precedence in the source region as transparent (FTB), originating call screening (OCS), call waiting (CW), and of the target region as call forwarding (CF), call waiting (CW), as shown in Figure 3.

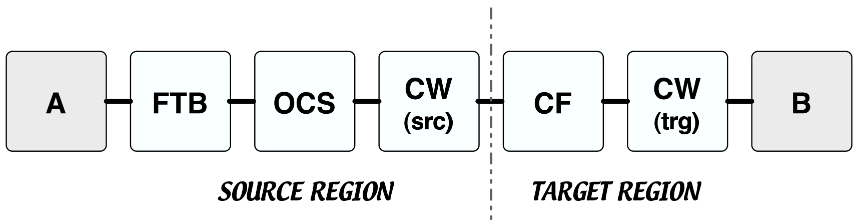

Figure 3: Static Feature Precedence

We considered having only one persistent router process that would handle all setup requests. However, in this case the body of the router would be long because it would have to handle all the possible subscription configurations. Instead, by dynamically creating a router for each part of the setup process, we can customize the router based on feature precedence order.

For each kind of feature box, there is a corresponding router that is dynamically created by the feature box before the feature box sends the router process a setup message. In order to create the next feature box of the usage, a router process checks the subscription information of the caller (and of the callee when the usage gets in the target zone) accordingly to the precedence order. For example, in our model, the router for the caller first checks if the caller subscribes to the free transparent feature, and runs dynamically an instance of a free transparent feature process:

```
if
  ::subs_T[thisindex]->
      atomic {
        run FTB(user[thisindex],caller[thisindex]);
        boxport_in!setup,origS,numberS,destS,i
      }
  ::else -> (...)
fi
```

The router processes for subsequent features in the precedence order do not check whether the free transparent feature should be created. The closer a usage is to its target, the shorter the body the router process. We model these customized routers with processes whose names that start with "router" (*e.g.,* `routeruser`, `routerOCS`).

6 Interface and Feature Boxes

In this section, we describe two interface boxes (a caller and callee), the router processes, and the feature boxes that we have included in our model. Caller, callee and call waiting processes persist, whereas the routers and the rest of the feature box processes are created as appropriate for usages.

We show state machine diagrams to describe the processes. The state names corresponds to statement labels in the PROMELA code found in Appendix A. The state machine diagrams mainly show the sending and receiving of protocol signals, and omit other process details for clarity. Sequences of statements that are atomic in the code are shown on one transition. For communication paths with channels, we omit the channel names (`A` and `B`) and show only the path name because the action (send or receive) disambiguates which channel would be used. Final states are shown with double circles.

6.1 Caller

Figure 4 shows the port and channel labels used by a caller, and Figure 5 shows the state machine corresponding to a caller process. A caller process first sets the `busy` bit of the corresponding user, then picks a destination (using non-deterministic choice), creates an appropriate router (see Section 5.3 for more details on the different routers) and sends a *setup* message (as described in Section 4.1) to this router along the `box_out` channel.

The caller then waits for an *upack* message from the feature box connected to it in the usage. The *upack* may be followed by an *avail* if the callee is available, or an *unavail* and then *teardown* if the callee is not available (in this case the caller sends a *downack* and then terminates). If the callee is available, the caller enters the communication phase, and then either the callee or caller can send a *teardown* message. Because *teardown* messages can cross paths, the caller must be prepared to receive a *teardown* even if it has already sent one. The communication phase could last arbitrarily long; this possibility is modelled by making the linked state a valid end state (`end_linked`).

In DFC, once a call is complete, the process should return to its initial state so that a second call could be initiated. To reduce the state space, we are currently working with a model where each caller can make only one call.

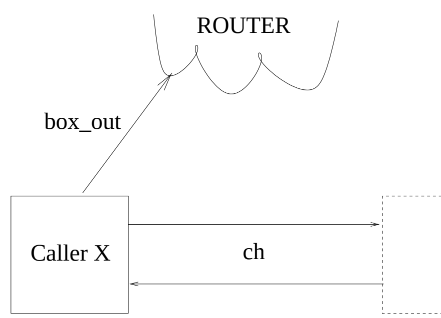

Figure 4: Channels and Ports for Caller X

6.2 Callee

A callee is divided into two processes: one for normal processing (called the callee process), and one that handles *setup* signals from a router through `box_in` channel (called the callee_router_watch process). These are separated because the callee may be busy and unable to respond to a *setup* message from the router. Figure 6 shows the channels associated with these two processes.

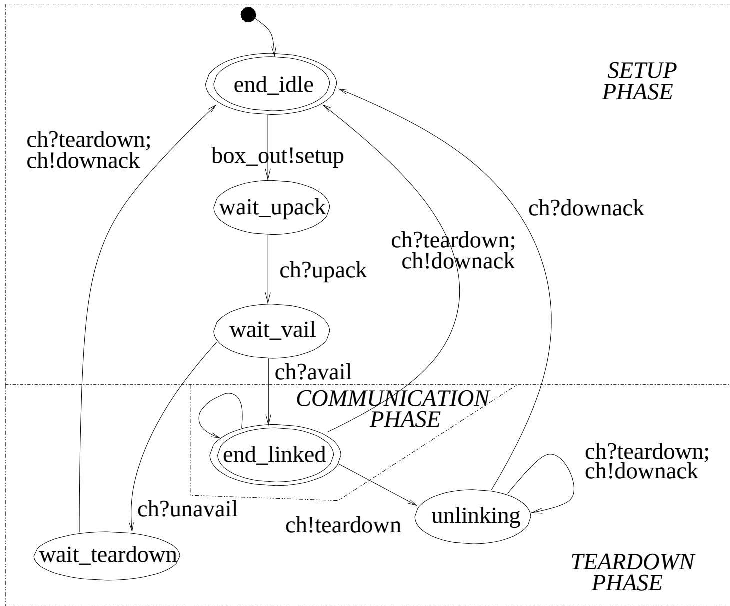

Figure 5: Caller Process State Machine

Figure 7 shows the state machine of the callee_router_watch and callee processes. When it receives a *setup* message from a router, the router_watch process checks if the user associated with the callee is currently busy using the value in the boolean array `busy`. If the callee is busy, the callee_router_watch process sends the sequence *upack*, *unavail*, *teardown* on the `ch` communication path, and waits for a *downack*. If the callee is not busy, the callee_router_watch process sends the callee the necessary information (originating port and channel identifier) to the callee process through a 0-capacity channel called `callee_intern`. The callee then sends the sequence *upack*, *avail* upstream, and moves into its communcation phase (the state `connected`). The rest of the callee's state machine is similar to the caller's.

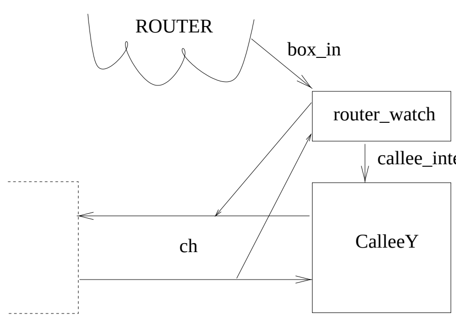

Figure 6: Channels and Ports for Callee Y

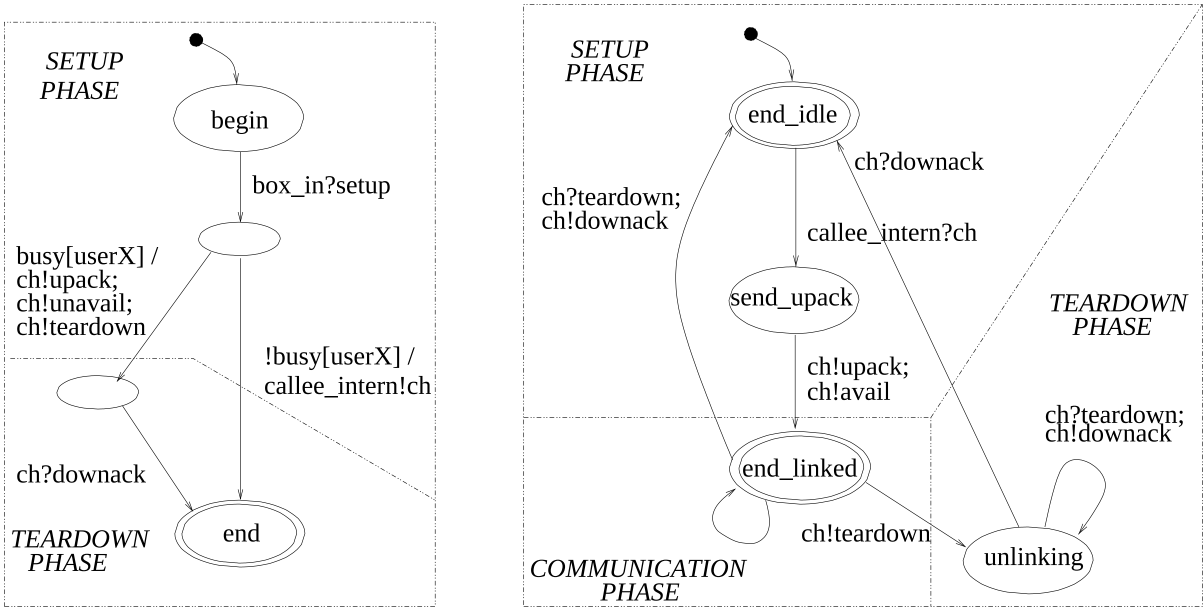

Figure 7: Callee_router_watch (left) and Callee (right) State Machines

6.3 Free Transparent Feature Box (FTB)

After the setup phase, which performs the proper initialization of the box, a free transparent feature box with two ports engaged in calls behaves *transparently*, *i.e.,* any signal received at one port is sent on to the other port. The box behaves as a buffer (which always forwards the signal to its neighbour in its next step, so the buffer never stores signals), remaining in the communication phase until the arrival of a *teardown* signal, which initiates the teardown phase. The channels and ports for a free transparent feature box are shown in Figure 8, and its state machine is shown in Figure 9. Because FTB is a free feature box, there is no transition that returns it to its initial state.

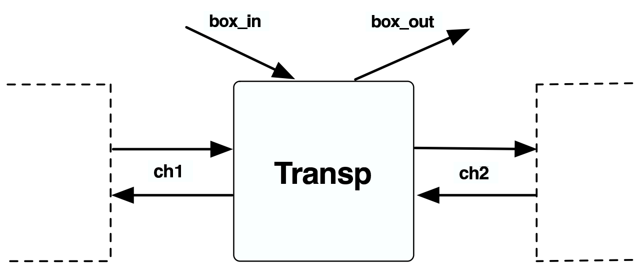

Figure 8: Channels for a Free Transparent Feature Box

6.4 Call Forwarding (CF)

The call forwarding feature is similar to the free transparent feature box except that it changes the field "dialed number" of the *setup* message if the subscriber wants to be reached on another device, which forces

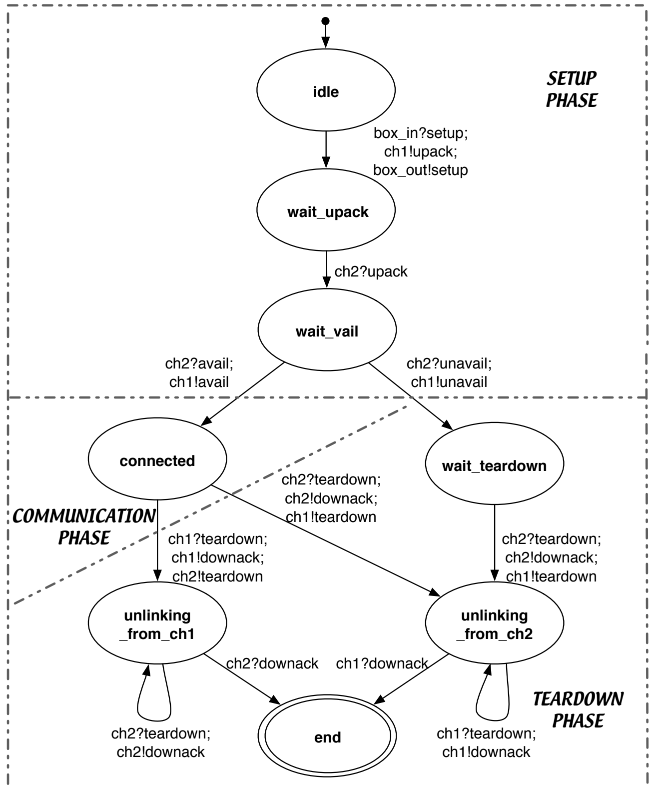

Figure 9: State Machine for Free Transparent Feature Box

the usage to take a different route. Call forwarding is a free feature box. This box communicates with exactly the same channels as the transparent box (Figure 8).

6.5 Originating Call Screening (OCS)

Originating call screening is a free feature box, and Figure 10 shows the state machine for an originating call screening feature. This feature is similar to the free transparent feature box except that when it receives a *setup* it sends the corresponding *upack*, and then either sends an *unavail* followed by *teardown* if the target number is forbidden by the subscriber or it continues the setup phase. Rather than modelling the blocking information, we model these two options as a non-deterministic choice. This box communicates with exactly the same channels as the transparent box (Figure 8).

6.6 Call Waiting (CW)

This feature enables the subscriber to switch between two different correspondents, by issuing an additional *switch* signal. Call waiting makes use of another additional signal, *waitsignal*, which is used to convey to a caller that the subscriber is already engaged in a usage but might switch, as allowed by the call waiting feature. None of the additional signals described are implemented in our Promela model. We abstract away the details of the callee issuing a switch signal by using a boolean variable `talk`, which change non-deterministically whenever two correspondents are connected to the subscriber. If the subscriber hangs up, then the call waiting box rings back the subscriber.

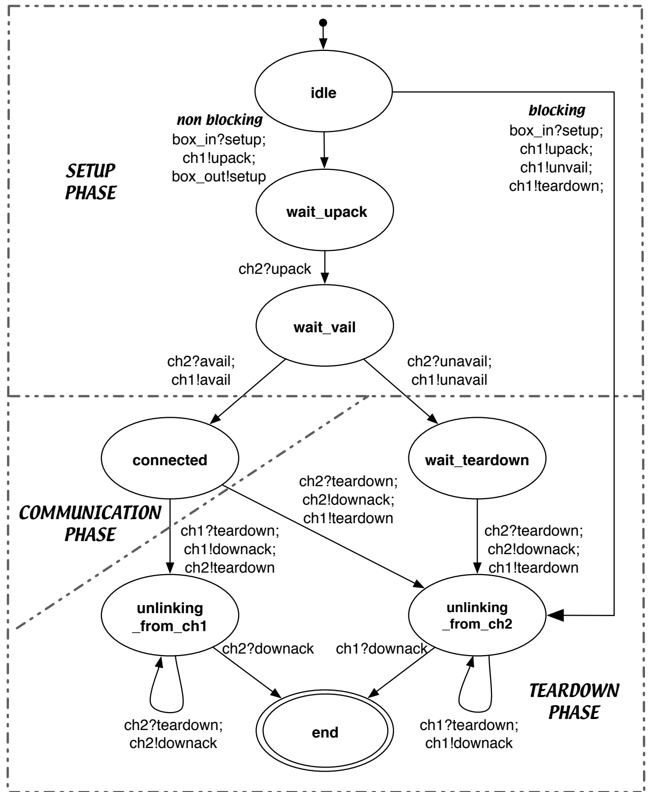

Figure 10: Originating Call Screening State Machine

Figure 11, shows the channels of the call waiting box. The `subsc` communication path links the call waiting box with its subscriber, while `first` and `second` link the call waiting box with the first and second correspondent respectively. The reason why the communication paths identified by `first` can be reversed is because there can be a caller (calling the subscriber) or a callee (called by the subscriber) connected to it. In our model, the role of the boolean variable *calling* is basically to know whether the subscriber is calling (source scenario) or is being called (target scenario).

The call waiting feature is modeled as two processes in a similar manner to how the callee is handled. Figure 12 shows the state machine of the CW_router_watch. The `CW_router_watch` process receives *setup* messages from the router and is connected to the call waiting process using a zero delay channel called `CW_intern`. The array `connCW` keeps track of the number of correspondents connected to the subscriber. If `connCW==2`, then the call waiting box cannot accept a new call and the `CW_router_watch` process responds with an *upack* message, followed by an *unavail*, and a *teardown*. It then waits for the *downack* before terminating. Figure 13 shows the state machine of the call waiting box. The left-hand side manages the source scenario, and it is denoted by **src** in the state names. The right-hand side manages the target scenario, and it is denoted by **trg** in the state names. Either scenarios differentiate whether the subscriber is connected to the first or the second correspondent, denoted by **first** and **second** respectively. The state machine is symmetric, except for the direction of the channels in the communication path `first` and the processing of the call back for the subscriber, which always leaves the call waiting process in the target scenario.

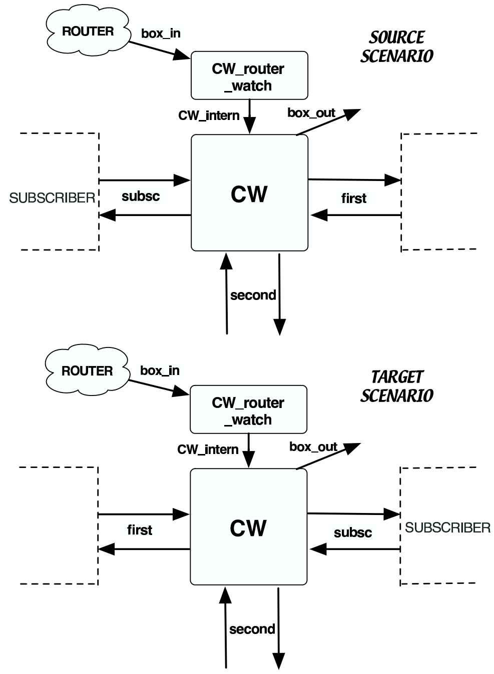

Figure 11: Channels for a Call Waiting box

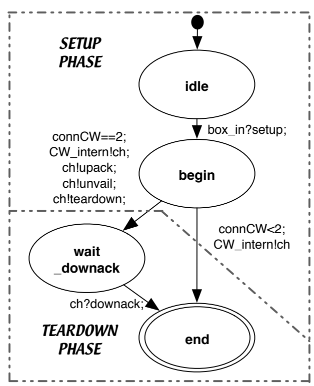

Figure 12: CW_router_watch State Machine

7 Model Checking DFC Properties

Our long-term goal is to find correctness properties of the DFC architecture concentrating on the call protocol, *i.e.,* we define what is good behaviour in the system, rather than detecting feature interactions. In this section, we show how our Promela model can be used to investigate DFC properties for small configurations of DFC. The DFC call protocol followed by the feature boxes describes the manipulation of signals traveling in the calls.

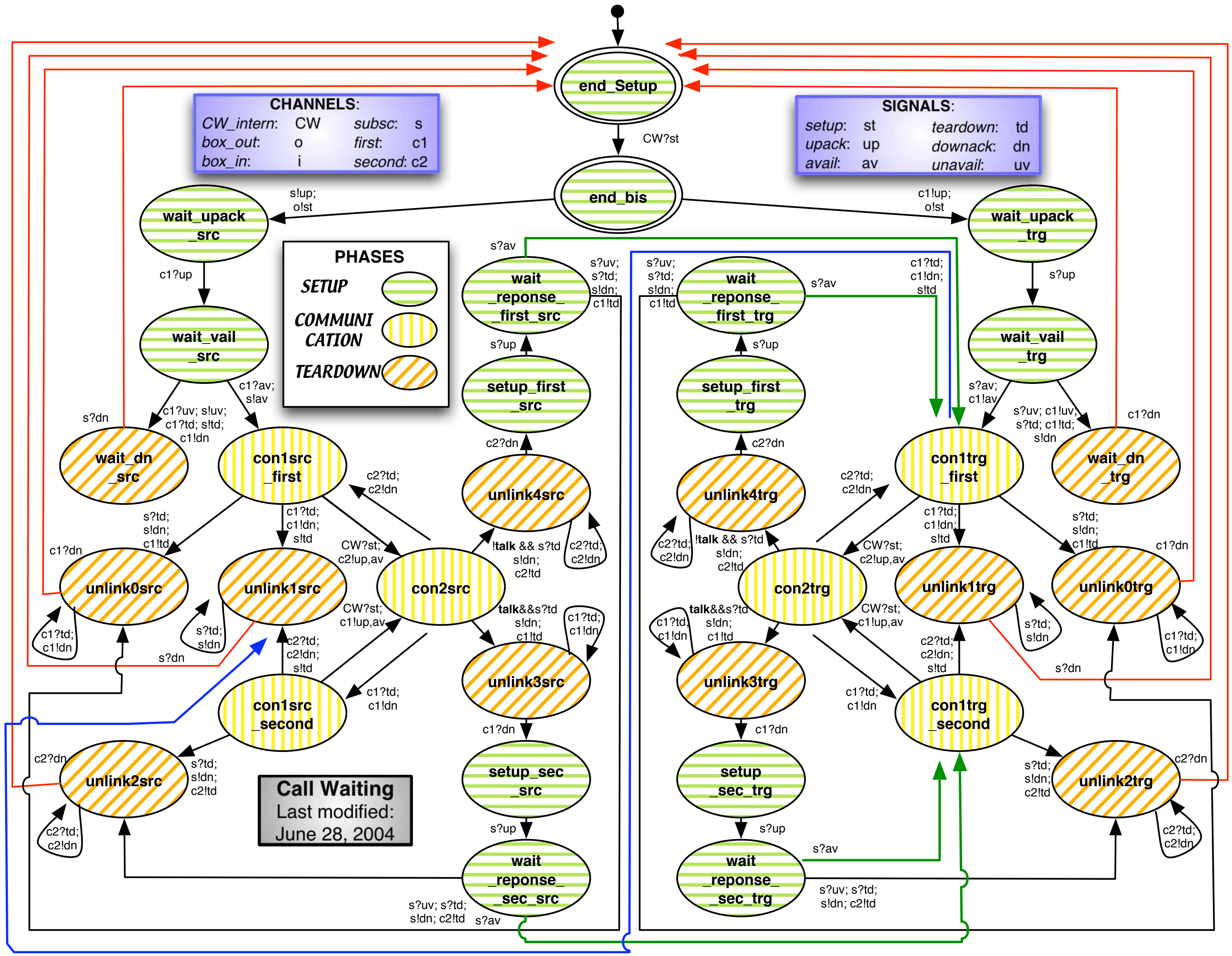

Figure 13: Call Waiting State Machine

To state the call protocol properties in usages between a caller and a callee, we name the communication path connecting a caller with a box as `alpha`, and the communication path connecting a box with a callee as `beta`, as shown in Figure 14.

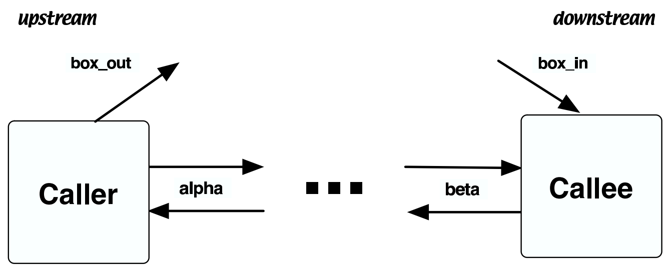

Figure 14: Channels in a Configuration between Caller and Callee

Because `never` claims in SPIN cannot refer directly to channels in an array such as as `chan_array`, we follow the approach of Holzmann [4], and add global variables to check the value of the last signal sent or received. We use these global variables to define our segment properties in LTL, which are translated to `never` claims to perform verification. The global variables `last_out` and `last_in` capture the value of signals traveling in the channels `box_out` and `box_in` respectively. We also used the variables `last_out_B` and `last_in_A` to capture the behaviour of the signals traveling through the communication path `alpha`, as well as the variables `last_out_A` and `last_in_B` to capture the behaviour of the signals traveling through the communication path `beta`. All the variables are initialized by default to zero, and reset whenever a transition is taken that reads or writes to the communication paths. For example, when a *teardown* signal is received from the communication path `beta`, the variable `last_in_B` is updated to have the value *teardown*:

```
chan_array[i].B?teardown;
last_in_B=teardown;
```

When sending a signal, the global variable is updated after the signal is transmitted. For example, when a *downack* signal is sent through the communication path `alpha`, the variable `last_out_B` is updated to have the value *downack*:

```
chan_array[i].B!downack;
last_out_B=downack;
```

Additionally, we use macros to define the boolean expressions used in the temporal properties. For example, below we define `g` as a boolean macro indicating that the last message sent through the communication path `beta` was a *teardown*, and the macro `l` indicates that the last message received from the communication path `alpha` was a *teardown*. Then to check that a *teardown* issued by a callee will eventually reach the caller, we check $\Box(\mathtt{g} \Rightarrow \Diamond \mathtt{l})$.

```
#define g last_out_B==teardown
#define l last_in_A==teardown
```

For clarity, we write our properties in terms of the channels and the communication paths.

In our model, by setting the subscription information of users, we can create different usages. We checked that the following call protocol properties hold in simple configurations, having one or more feature boxes delimited by a caller and a callee:

a. A *setup* signal created by caller A should eventually reach callee B.
$\Box(\mathtt{box\_out!setup} \Rightarrow \Diamond \mathtt{box\_in?setup})$

b. A *teardown* signal created by callee B should eventually reach caller A.
$\Box(\texttt{alpha!teardown} \Rightarrow \Diamond\texttt{beta?teardown})$

c. A *teardown* signal created by callee B should eventually reach caller A.
$\Box(\texttt{beta!teardown} \Rightarrow \Diamond\texttt{alpha?teardown})$

d. An *avail* signal created by callee B should eventually reach caller A.
$\Box(\texttt{beta!avail} \Rightarrow \Diamond\texttt{alpha?avail})$

e. An *unavail* signal created by callee B should eventually reach caller A.
$\Box(\texttt{beta!unavail} \Rightarrow \Diamond\texttt{alpha?unavail})$

When callers are allowed to call multiple times, the model has an infinite state space. We abstract the model to a finite state space, by only allowing the caller to call once or twice. We have not yet analyzed whether this is a sound abstraction for the properties we are checking. With this abstraction, for these properties, on small configurations, the state vector size, and the length of the explored path during verification is relatively small.

The verifications were run on a PC with 1.4GHz Xeon CPU and 8GB in RAM. There are different parameters that can be set in Spin to make the verification effort more efficient when checking more complex configurations.

The Spin options used at compile time are:

`-DCOLLAPSE` reduces the memory requirements, exploiting a hierarchical indexing method to compress the state vector sizes by up to 80% to 90%.

`-DMEMLIM=N` changes the memory limit to N, which by default is 128Mb, but we used 4Gb, which is the maximum that a single machine process access.

The Spin options used at run time are:

`-mN` sets the maximum depth search to N, which by default is 1000.

In our model, we can set the value of certain global variables to perform verification on different configurations. We can change the value of `N` (maximum number of users), `times` (number of calls a caller can make), and `subs_TFB, subs_OCS, subs_CF, subs_CW` (subscription information). The values selected for the global variables, subscription information, as well as the results obtained from the model checking verification runs are shown in Table 1. We present the maximum execution statistics for the largest verification of a property in terms of number states, memory used and time, instead of listing all the segment properties statistics per configuration. If the subscription column shows TFB, OCS, or CW(src), it means that in the configurations checked all the users have these feature boxes in their source region. If the subscription column shows CF, or CW(trg), it means that in the configurations checked all the users have these feature boxes in their target region. For the call waiting feature, we checked a simple configuration shown in Figure 15.

In all the configurations we verified, we checked also for the absence of deadlock, but in the configuration for call waiting we ran out of memory without finishing the verification. For the call waiting box, we found a previously unknown race condition while checking for deadlock in the configuration of Figure 15. The situation happens when the call waiting box is processing the call back for a person on hold after the subscriber hangs up. The normal situation is shown with a message sequence chart in Figure 16, where the subscriber and the person on hold get connected. The race condition situation is shown with a message sequence chart in Figure 17, where the subscriber has not read yet the *downack* signal sent by the call waiting box. The subscriber seems unavailable, so the call back processing fails. We handle the situation of the processing back failure by tearing down the connections for the subscriber and the person on hold, leaving the call waiting box ready to be used again.

Configuration	Number of callers	Number of callees	Times caller can call	Subscription	Number of states	State vector (bytes)	Total depth reached	Memory used (Mbytes)	Time (min:sec)
1	1	6	1	TFB	816	556	172	1.71	0:00.04
2	1	6	2	TFB	88395	588	294	7.14	0:04.48
3	1	6	1	TFB,OCS	3289	572	201	2.02	0:00.15
4	1	6	2	TFB,OCS	1.37726e+6	620	353	88.24	0:01.23
5	1	6	1	TFB,CF	2707	572	202	1.92	0:00.12
6	1	6	2	TFB,CF	1.0337e+6	620	355	61.72	0:01.58
7	1	6	1	TFB,OCS,CF	8871	588	232	2.43	0:00.41
8	1	6	2	TFB,OCS,CF	1.06627e+7	652	418	680.218	0:22.59

Table 1: Execution Statistics for Property Checking

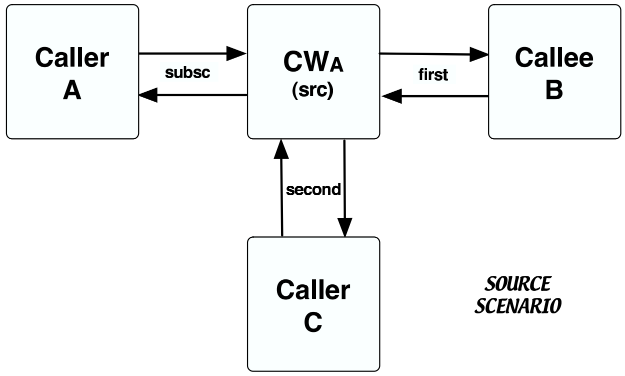

Figure 15: Call Waiting Feature Box in a Simple Configuration

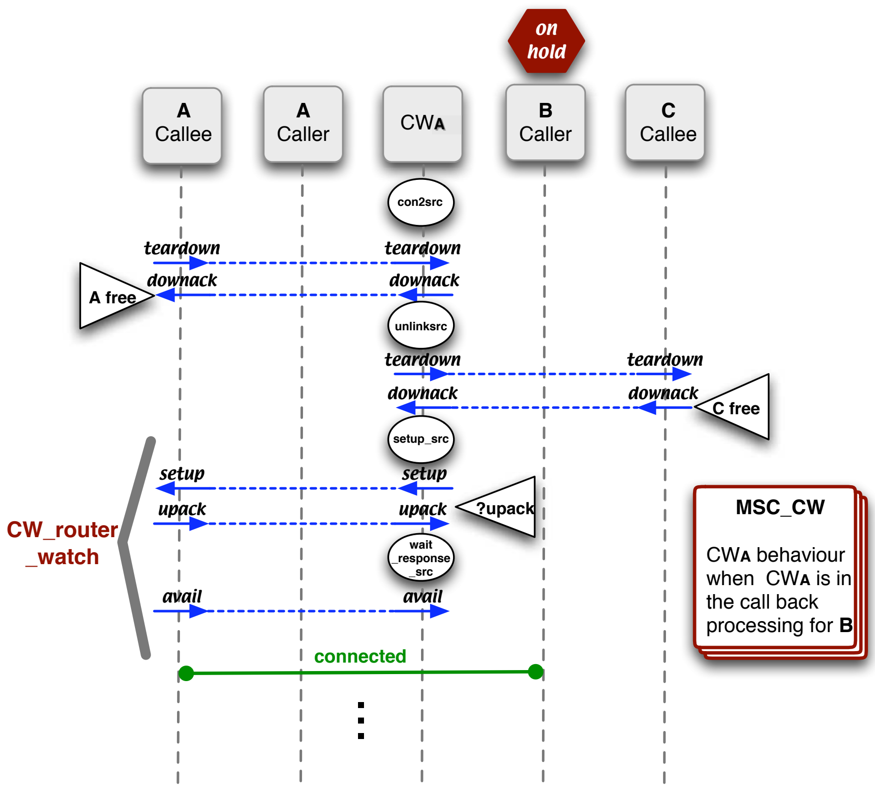

Figure 16: Normal situation in Call Back Processing

Checking fixed configurations (*i.e.*, fixed subscription information) does not allow us to conclude that the properties hold of all DFC configurations. We are exploring a compositional reasoning approach whereby we can model check feature properties on features individually, and then use theorem proving to conclude that

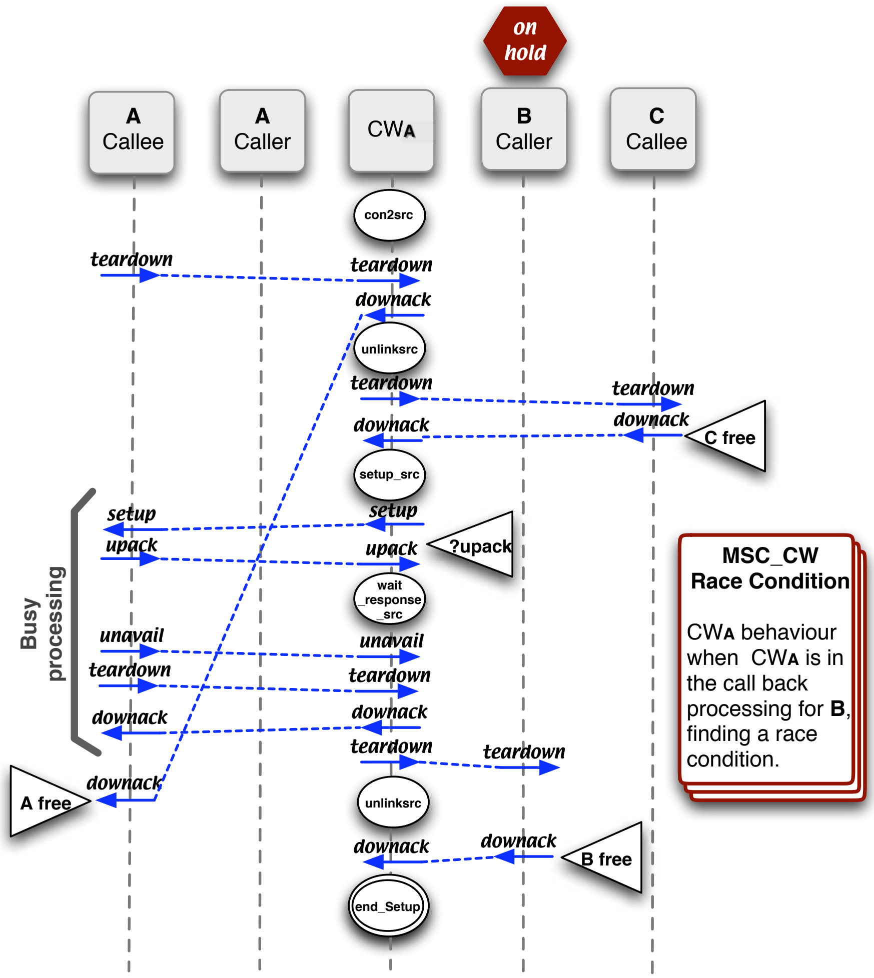

Figure 17: Race Condition in Call Back Processing

call protocol properties hold in all DFC configurations.

8 Modelling Issues

In this section, we discuss some of the choices we made in creating our Promela model. We describe the difference between dynamic and static models, and also alternatives for how subscription information can be handled.

In creating our Promela model for DFC, we had to choose whether to create processes dynamically or to have all processes created at initialization and exist throughout the system's execution. We call the second option the static model. Dynamic creation of processes best matches the way DFC actually works, but means that we are not guaranteed to have a finite state space.

To understand the difference between the dynamic and static models, consider a simple configuration where a caller calls a callee, and both do not subscribe to any services. In a static model all processes (the caller, callee and router) and each channel (two for each internal call and two for to and from the router) exist permanently. In a longer configuration with n internal calls, we would require $4n$ channels. If we want it to be possible for a caller to call multiple callees, we need multiple instances of each free feature box with channels. For k users, where one user subscribes to call forwarding, we need to have $k-1$ call forwarding processes – one for each potential caller with all the corresponding channels. For bound feature boxes and the caller and callee, in order to have all the possible connections statically created, we would have to create additional ports that do not exist in the feature boxes.

In a dynamic model, the caller chooses a channel and creates the router before sending it a setup message, and the router creates the callee before propagating the setup. Because we consider these operations to be atomic, we only require two zero capacity channels for all the routing operations. In the dynamic model, we require only $n+2$ channels for n internal calls.

Another alternative is to check only one configuration and caller-callee combination in a static model as is done by Calder and Miller [3]. They use Perl scripts to generate particular combinations of features for a model of email features in PROMELA. Each sender of email can send multiple messages and the state space is still finite.

We chose to create processes dynamically in our PROMELA model because (1) it was easier to write; (2) we can cover more options for configurations in a single model; and (3) it more closely matches the behaviour of DFC. However, in a dynamic model, if a caller can make multiple calls, we have an infinite state space because process identifiers of created processes cannot be reused. Even though SPIN allows a dead process identifier to be reused, there is always at least one execution of the model in which process identifiers cannot be reused because of the interleaved behaviour: a process that has executed its last instruction still has to be chosen to execute again before it actually releases its process identifier. Therefore, the maximum number of processes (255) is always eventually reached in a path of the execution where process identifiers are not released.

To avoid an infinite state space, we considered using the read-only variable `_nr_pr`, which returns the number of active processes at the time it is referenced. Using this variable, we could test whether this number was higher than some constant K before allowing a new call to proceed. This method forces the caller to block until dead processes have released their pids. However, it is not clear what is an appropriate value for K to capture all possible behaviours of the model without this limitation. We chose to restrict the caller to making only one or two calls to reduce the state space.

As was discussed in Section 7, we model checked some small configurations varying the number of callers, callees, and number of times a caller could make a call. An interesting avenue for further exploration is symmetry reductions that would allow us to conclude that restricting the caller to making a finite number of calls with a certain number of callers and callees is a sound abstraction.

Another modelling issue that we encountered was with regard to how to capture subscription information used by the router. As described in Section 6, the routers encode the precedence order of the features, but check whether a user is subscribed to features using Boolean arrays for each feature (*e.g.,* `subs_OCB`). We experimented with encoding fixed subscription information in the router, and using Perl scripts to generate simplified routers for particular configurations. However, the gain in memory used, state space vector size and depth reached were not significant, and we continued with our original approach of representing the subscription information using arrays.

9 Related Work

In this section, we briefly overview other efforts to verify DFC-related artifacts. Zave provided a formal description of the service layer of a telecommunication system, organized according to the DFC [8] architecture, using PROMELA and Z. The routing algorithm as well as the routing data were described in Z, and the DFC protocols were described in PROMELA. SPIN was used to check that the protocols of the virtual network never deadlock. Zave used a static model, whereas in our model checking of fixed DFC configurations, we dynamically create the box processes, which allows the model to match closely DFC's behaviour. As in

Zave's work, the first step of our verification approach was to check for absence of deadlock, but we also check our call protocol properties.

A Java implementation of DFC in an IP setting is called ECLIPSE, developed at AT&T Labs, and described in [2]. To simplify feature development, the ECLIPSE Statecharts notation was developed, which is a customized version of the UML Statecharts description language. They perform model checking, using the Mocha model checker tool [1], and translate an individual ECLIPSE feature box code to the modeling language framework of Mocha to verify that the feature combined with standardized environmental peer entities follow the DFC protocols. At AT&T, a check for deadlock using synchronous communication between the feature and its environment was performed. We extended this work by checking liveness properties.

An abstraction for DFC internal calls is embodied in a high-level domain specific programming language called Boxtalk [13], which is the next generation of feature-creation tools after ECLIPSE. Boxtalk encourages correct programming of coordinating components and provides a foundation to prove behavioural equivalence in components. One difference between our model and Boxtalk models of features is that in Boxtalk processing of the completion of a call is asynchronous with respect to the control flow of a program, *i.e.,* the model can process the end of the call interleaved with the beginning of a new call. In our model, a call must be fully torn down before a new call can be setup.

In [10], a set of constraints on feature behaviour called "Ideal address translation" are defined. This set has provable properties, is modular and supports extensibility and component coordination. We are not dealing directly with the problem of address translation, so the results are not directly related to our work.

Finally, in [9] there is a description of a feature-oriented specification technique that follows the structural approach to detect feature interactions relying on properties of individual feature programs. Our approach, instead, looks for assertions stating the correct behaviour of the system.

10 Conclusions

We have described our model of DFC and some of its features in PROMELA. Additionally, we have discussed our preliminary effort at model checking DFC configurations. We also described several abstractions, which need to be investigated further to determine their soundness. We plan to use this model as a platform for investigating call protocol properties of DFC.

11 Acknowledgments

We thank members of the DFC study group at the University of Waterloo for discussions on DFC.

References

[1] R. Alur, T. A. Henzinger, F. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In *Computer-Aided Verification*, volume 1427 of *Lecture Notes In Computer Science*, pages 521–525. Springer, 1998.

[2] G. Bond, F. Ivancić, N. Klarlund, and R. Trefler. ECLIPSE Feature Logic Analysis. *2nd International IP-Tel*, 2001.

[3] M. Calder and A. Miller. Generalising Feature Interactions in Email. In D. Amyot and L. Logrippo, editors, *Feature Interactions in Telecommunications and Software Systems VII*, pages 187–204. IOS Press, 2003.

[4] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley.

[5] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering XXIV(10)*, pages 831–847, 1998.

[6] M. Jackson and P. Zave. *The DFC Manual.* AT&T Labs, November 2003.

[7] A. L. Juarez Dominguez. Verification of DFC call protocol correctness criteria. Master's thesis, School of Computer Science, University of Waterloo. May, 2005.

[8] P. Zave. Formal description of telecommunication services in Promela and Z. In M. Broy and R. Steinbrüggen, editors, *Proceedings of the 19th International NATO Summer School: Calculational System Design*, pages 395–420, 1999.

[9] P. Zave. An experiment in Feature Engineering. In A. McIver and C. Morgan, editors, *Monographs In Computer Science, Programming Methodology*, pages 353–377. Springer-Verlag, 2003.

[10] P. Zave. Address Translation in Telecommunication Features. *ACM Transactions on Software Engineering and Methodology*, 13(1):1–36, January 2004.

[11] P. Zave and M. Jackson. DFC modifications II: Protocol extensions. Technical report, AT&T Laboratories–Research, November 1999.

[12] P. Zave and M. Jackson. DFC modifications I: Routing extensions. Technical report, AT&T Laboratories–Research, May 2000.

[13] P. Zave and M. Jackson. A call Abstraction for Component Coordination. In *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming: Workshop on Formal Methods and Component Interaction*, June 2002.

A Promela Model

A.1 Global declarations

A.1.1 Global variables

```
    /* Max number of channels at the same time */
#define M 16
    /* Number of users in the system. This number is related
    to the mtype definitions for users, ranging from user0 to
    userN-1. If this number changes, mtype definitions should
    be added. */
#define N 6
    /* Number of bits needed to identify a channel with (2^L)>M*/
#define L 4

    /* mtype definitions are associated to natural numbers, being
    the last one declared mapped to 1, the second last mapped to 2,
    and so on. Then users must be last in mtype definitions,
    declared in reverse order. */
    /* We tried to use integers to represent users. However,
    mtypes for users is needed to avoid confusion between the
    numbers representing users and the numbers associated to the
    mtype definition of signals */
mtype{
    /* mtypes definitions to represent the signals:*/
 setup, upack, teardown, downack, avail, unavail, waitsignal,
    /* mtypes definitions to represent the users:*/
 noone, user5, user4, user3, user2, user1, user0}

    /* The subscription of users to Transparent (src FB)*/
bool subs_T[N];
    /* The subscription of users to Call Blocking (src FB)*/
bool subs_OCS[N];
    /* The subscription of users to Call Forwarding (trg FB)*/
bool subs_CF [N];
    /* The subscription of users to Call Waiting (src,trg FB)*/
bool subs_CW [N];


    /* The forwarding information.
    NOTE: currently we don't check for loops */
mtype CF_info [N];

    /* Whether a user is busy or idle */
bool busy[N];

    /* Saturation of callwaiting boxes */
byte connCW[N];
```

```
    /* The set of Intern channels for the Callwaiting box*/
typedef CW_intern{
    chan intern = [1] of {mtype,mtype, mtype, mtype, byte}
}
CW_intern CW_channel [N];

    /* The set of Intern channels for the Callee*/
typedef Callee_intern{
    chan intern = [0] of {byte}
}
Callee_intern Callee_channel [N];


    /* The channel for the switch which is shared by all
    boxes because of the atomicity of the switches and
    the 0-capacity buffers*/
chan box_out = [0] of  {mtype,mtype, mtype, mtype, byte};
chan box_in = [0] of {mtype,mtype, mtype, mtype, byte};

    /* The two one-direction channels used for passing
    messages. Both of them are capacity 3 because the
    maximum sequence of signals in our model is
    upack,unavail,teardown */
typedef Com_chan  {
    chan A = [3] of {mtype}
    chan B = [3] of {mtype}
          }

    /* The set of Channels*/
Com_chan chan_array [M];

    /* The array to keep track of whether a channel is allocated or not*/
bool channel_busy [M];

    /* To find the first idle Com_chan. Channels can be reused */
inline C_assign(iden)
{
 iden = 0;
 do
    ::(channel_busy[iden] && !(iden==(M-1)))  -> iden++;
    ::(channel_busy[iden] &&  (iden==(M-1)))  -> assert (false); break
    ::!channel_busy[iden] -> channel_busy[iden] = true; break
 od;
}

    /* For verification */
byte last_out;
byte last_out_B;
byte last_out_A;
```

```
byte last_in;
byte last_in_A;
byte last_in_B;
```

A.1.2 Macros used in temporal properties

```
#define a last_out==setup
#define b last_in==setup
#define c last_out_B==teardown
#define d last_out_B==downack
#define e last_out_A==upack
#define f last_out_A==downack
#define g last_out_A==teardown
#define h last_out_A==avail
#define i last_out_A==unavail
#define j last_in_A==unavail
#define k last_in_A==upack
#define l last_in_A==teardown
#define m last_in_A==avail
#define n last_in_A==downack
#define o last_in_B==teardown
#define p last_in_B==downack
```

A.1.3 Initializations

```
init{

    CF_info[0]=noone;
    CF_info[1]=user2;
    CF_info[2]=user4;
        CF_info[3]=noone;
    CF_info[4]=user2;
    CF_info[5]=user3;

        subs_T[0]=false;
    subs_T[1]=false;
    subs_T[2]=false;
    subs_T[3]=false;
    subs_T[4]=false;
    subs_T[5]=false;

    subs_CF[0]=false;
    subs_CF[1]=false;
    subs_CF[2]=false;
    subs_CF[3]=false;
    subs_CF[4]=false;
    subs_CF[5]=false;
```

```
    subs_OCS[0] = false;
    subs_OCS[1] = false;
    subs_OCS[2] = false;
    subs_OCS[3] = false;
    subs_OCS[4] = false;
    subs_OCS[5] = false;

    subs_CW[0] = true;
    subs_CW[1] = false;
    subs_CW[2] = false;
    subs_CW[3] = false;
    subs_CW[4] = false;
    subs_CW[5] = false;

end:
    atomic{
        byte usr=1;
        do           /* Initialize Caller processes */
        ::(usr<N) -> run Caller(usr); usr=usr+1;
        ::else -> break;
        od;
        usr=1;
    do           /* Initialize Callee processes */
        ::(usr<=N) -> run Callee(usr); usr=usr+1;
        ::else -> break;
        od;
        usr=1;
        do           /* Initialize CW processes */
        ::(usr<=N) -> run CW(usr); usr=usr+1;
        ::else -> break;
        od;
    }
}
```

A.2 Caller process

```
proctype Caller(byte me)
{                   /* CALL ONCE */
 byte dest;         /*the non deterministically chosen callee*/
 unsigned ch : L;   /*the channel identifier of the free channel*/

/*  NOTES:                                                               *
*       ch.A is the outgoing receiving channel                           *
*       ch.B is the outgoing sending channel                             */

end_set_up:
    /*Sets up the busy bit*/
```

```
  atomic{
    do
    ::(busy[me-1]==false) -> busy[me-1]= true; break
    ::else -> skip
    od;
    /*find a free channel thanks to a call of the inline function*/
    C_assign(ch);
    /*run a process router_user for "Switch after a user"*/
    run routeruser(me);
    /*and choose non deterministically a calle before sending setup*/
    if
    ::dest = user0
    ::dest = user1
    ::dest = user2
    ::dest = user3
    ::dest = user4
    ::dest = user5
    fi;
    box_out!setup,me,dest,dest,ch;
    last_out=setup;
   }

wait_upack:
  atomic{
    chan_array[ch].A?upack;
    last_in_A=upack };


/*wait the next signal either unavail or avail*/
wait_vail:
    if
    ::chan_array[ch].A?unavail ->
      last_in_A=unavail;
      goto wait_teardown
    ::chan_array[ch].A?avail;
      last_in_A=avail;
      goto end_connected
    fi;

end_connected:
    do
    ::atomic{
       chan_array[ch].B!teardown->
       last_out_B=teardown;
       goto zombie
      }
    ::atomic{
       chan_array[ch].A?teardown->
       last_in_A=teardown;
       chan_array[ch].B!downack;
```

```
       last_out_B=downack;
       busy[me-1]=false;
       goto end
      }
        /* can stay connected forever */
    ::true -> skip
    od;

wait_teardown:
    atomic{
      chan_array[ch].A?teardown;
      last_in_A=teardown;
      chan_array[ch].B!downack;
      last_out_B=downack;
      busy[me-1]=false;
      goto end
    }

zombie:
    do
    ::atomic{
         chan_array[ch].A?downack->
         last_in_A=downack;
         busy[me-1]=false;
         goto end
        }
    ::atomic{
         chan_array[ch].A?teardown->
         last_in_A=teardown;
         chan_array[ch].B!downack;
         last_out_B=downack
        }
    od;
end : skip
}
```

A.3 Callee process

```
proctype callee_router_watch (byte thisnum)
{
 byte scr2;byte scr4;   /*scratch values*/
 unsigned ch:L;         /*the channel identifier of the free channel*/

Set_up:
  atomic{
        /* Shouldn't block because it's rendezvous channel */
    box_in?setup,scr2,eval(thisnum),scr4,ch ->
    last_in=setup;
```

```
  }
    if
    ::(busy[thisnum-1] == false) ->
        busy[thisnum-1]= true;
            /* Shouldn't block because it's rendezvous channel */
        Callee_channel[thisnum-1].intern!ch;

    ::else ->atomic{
        chan_array[ch].A!upack;
        last_out_A=upack;
        chan_array[ch].A!unavail;
        last_out_A=unavail;
        chan_array[ch].A!teardown;
        last_out_A=teardown;
        };
        chan_array[ch].B?downack;
        last_in_B=downack;
    fi;
end: skip
}

proctype Callee (byte me)
{
 byte ch;

/*  NOTES:                                                            *
*       ch.A is the incoming sending channel                          *
*       ch.B is the incoming receiving channel                        */

end_start:
    Callee_channel[me-1].intern?ch;

sendupack:
    atomic{
       chan_array[ch].A!upack;
       last_out_A=upack;
       chan_array[ch].A!avail;
       last_out_A=avail;
       goto end_connected
    }

end_connected:
    do
    ::atomic{
       chan_array[ch].A!teardown;
       last_out_A=teardown;
       goto zombie
        }
    ::atomic{
       chan_array[ch].B?teardown->
```

```
       last_in_B=teardown;
       chan_array[ch].A!downack;
       last_out_A=downack;
       busy[me-1]=false;
       goto end_start
        }
    ::true -> skip
    od;

zombie:
    do
    ::atomic{chan_array[ch].B?downack->
        last_in_B=downack;
        busy[me-1]=false;
        goto end_start
         }
    ::atomic{chan_array[ch].B?teardown->
        last_in_B=teardown;
        chan_array[ch].A!downack;
        last_out_A=downack;
        }
    od;
}
```

A.4 Router processes

A.4.1 router_user process

```
proctype routeruser (byte thisindex)
{
 byte number;
 byte dest;
 unsigned ch: L;

atomic{
    box_out?setup,eval(thisindex),number,dest,ch;
    if
    ::subs_T[thisindex-1] -> atomic {
        run TFB(thisindex);
        box_in!setup,thisindex,number,dest,ch}
    ::else ->
        if
        ::subs_OCS[thisindex-1]-> atomic{
            run OCS(thisindex);
            box_in!setup,thisindex,number,dest,ch}
        ::else ->
            if
            ::subs_CW[thisindex-1] -> atomic{
                run CW_router_watch(thisindex);
```

```
                box_in!setup,thisindex,number,number,ch}
            ::else ->
                if
                ::subs_CF[number-1] ->  atomic{
                    run CF(number);
                    box_in!setup,thisindex,number,number,ch }
                ::else ->
                    if
                    ::subs_CW[number-1] -> atomic{
                        run CW_router_watch(number);
                        box_in!setup,thisindex,number,number,ch}
                    ::else -> atomic{
                        run callee_router_watch(number);
                        box_in!setup,thisindex,number,number,ch}
                    fi;
                fi;
            fi;
        fi;
    fi;
    }
end:skip
}
```

A.4.2 routerT process (Source region)

```
proctype routerT (byte thisindex)
{
 byte number;
 byte dest;
 unsigned ch: L;

atomic{
    box_out?setup,eval(thisindex),number,dest,ch;
    if
    ::subs_OCS[thisindex-1]-> atomic{
        run OCS(thisindex);
        box_in!setup,thisindex,number,dest,ch}
    ::else ->
        if
        ::subs_CW[thisindex-1] -> atomic{
            run CW_router_watch(thisindex);
            box_in!setup,thisindex,number,number,ch}
        ::else ->
            if
            ::subs_CF[number-1] ->  atomic{
                run CF(number);
                box_in!setup,thisindex,number,number,ch }
            ::else ->
                if
                ::subs_CW[number-1] -> atomic{
```

```
                    run CW_router_watch(number);
                    box_in!setup,thisindex,number,number,ch}
                ::else -> atomic{
                    run callee_router_watch(number);
                    box_in!setup,thisindex,number,number,ch}
                fi;
            fi;
        fi;
    fi;
    }
}
```

A.4.3 routerOCS process (Source region)

```
proctype routerOCS (byte thisindex)
{
 byte number;
 byte dest;
 unsigned ch: L;

atomic{
    box_out?setup,eval(thisindex),number,dest,ch;
    if
    ::subs_CW[thisindex-1] -> atomic{
        run CW_router_watch(thisindex);
        box_in!setup,thisindex,number,number,ch}
    ::else ->
        if
        ::subs_CF[number-1] ->  atomic{
            run CF(number);
            box_in!setup,thisindex,number,number,ch }
        ::else ->
            if
            ::subs_CW[number-1] -> atomic{
                run CW_router_watch(number);
                box_in!setup,thisindex,number,number,ch}
            ::else -> atomic{
                run callee_router_watch(number);
                box_in!setup,thisindex,number,number,ch}
            fi;
        fi;
    fi;
    }
}
```

A.4.4 routerCWsrc process (Source region)

```
proctype routerCWsrc (byte thisindex)
{
 byte number;
```

```
 byte dest;
 unsigned ch: L;

atomic{
    box_out?setup,eval(thisindex),number,dest,ch;
    if
    ::subs_CF[number-1] ->  atomic{
        run CF(number);
        box_in!setup,thisindex,number,number,ch }
    ::else ->
        if
        ::subs_CW[number-1] -> atomic{
            run CW_router_watch(number);
            box_in!setup,thisindex,number,number,ch}
        ::else -> atomic{
            run callee_router_watch(number);
            box_in!setup,thisindex,number,number,ch}
        fi;
    fi;
    }
}
```

A.4.5 routerCF process (Target region)

```
proctype routerCF (byte thisindex)
{
 byte orig; byte number; byte dest;
 unsigned ch: L;

atomic{
    box_out?setup,eval(thisindex),number,dest,ch;
    if
    ::subs_CW[number-1] -> atomic{
        run CW_router_watch(number);
        box_in!setup,thisindex,number,number,ch}
    ::else -> atomic{
        run callee_router_watch(number);
        box_in!setup,thisindex,number,number,ch}
    fi;
    }
}
```

A.4.6 routerCWtrg process (Target region)

```
proctype routerCWtrg (byte thisuser)
{
 byte orig; byte number; byte dest;
 unsigned ch: L;

  atomic{
```

```
    box_out?setup,orig,number,dest,ch;
    run callee_router_watch(number);
    box_in!setup,orig,number,number,ch;
  }
}
```

A.5 Feature boxes processes

A.5.1 Free Transparent process

```
proctype TFB(byte thisuser)
{
    byte number;
    byte dest;
    unsigned ch2: L; /*the index of the next communication channel*/
    unsigned ch1: L; /*the index of the previous communication channel*/

/*  NOTES:                                                                *
*       ch1.A is the incoming sending channel                             *
*       ch1.B is the incoming receiving channel                           *
*       ch2.A is the outgoing receiving channel                           *
*       ch2.B is the outgoing sending channel                             */

begin: atomic {
        box_in?setup,eval(thisuser),number,dest,ch1;
        chan_array[ch1].A!upack;
        C_assign(ch2);
        run routerT(thisuser);
        box_out!setup,thisuser,number,dest,ch2
       };

wait_upack:
    chan_array[ch2].A?upack;

wait_vail:
    if
    :: chan_array[ch2].A?unavail ->
        chan_array[ch1].A!unavail;
        goto wait_teardown
    :: chan_array[ch2].A?avail ->
        chan_array[ch1].A!avail;
        goto connected
    fi;

wait_teardown:
    atomic{
        chan_array[ch2].A?teardown ->
        chan_array[ch2].B!downack;
        chan_array[ch1].A!teardown;
        goto unlinking_from_ch2
```

```
    }

connected:
    do
    :: atomic{
            chan_array[ch2].A?teardown ->
            chan_array[ch2].B!downack;
            chan_array[ch1].A!teardown;
            goto unlinking_from_ch2
        }
    :: atomic {
            chan_array[ch1].B?teardown ->
            chan_array[ch1].A!downack;
            chan_array[ch2].B!teardown;
            goto unlinking_from_ch1
        }
    od;

unlinking_from_ch2:
    do
    :: atomic {
        chan_array[ch1].B?downack ->
        goto end
       }
    :: atomic{
        chan_array[ch1].B?teardown ->
        chan_array[ch1].A!downack;
       }
    od;

unlinking_from_ch1:
    do
    :: atomic {
        chan_array[ch2].A?downack ->
        goto end
       }
    :: atomic{
        chan_array[ch2].A?teardown ->
        chan_array[ch2].B!downack;
       }
    od;

end:skip
}
```

A.5.2 Call Blocking process

```
proctype OCS(byte thisuser)
{
 byte number;
```

```
 byte dest;
 unsigned ch1 : L; /*the index of the previous communication channel*/
 unsigned ch2 : L; /*the index of the next communication channel*/

/*  NOTES:                                                             *
*       ch1.A is the incoming sending channel                         *
*       ch1.B is the incoming receiving channel                       *
*       ch2.A is the outgoing receiving channel                       *
*       ch2.B is the outgoing sending channel                         */

begin :
  atomic{
    box_in?setup,eval(thisuser),number,dest,ch1 ->
        chan_array[ch1].A!upack;
    if /*non_deterministic choice*/
        /*blocking*/
    :: chan_array[ch1].A!unavail;
        chan_array[ch1].A!teardown;
        goto unlinking_from_ch2
        /*non blocking */
    :: C_assign(ch2);
        run routerOCS(thisuser);
        box_out!setup,thisuser,number,dest,ch2
    fi
  }

waiting_upack: chan_array[ch2].A?upack;

/*the OCS propagates unavail and teardown but also "downacks" the latter*/
wait_vail:
    if
    ::chan_array[ch2].A?unavail; /*busy tone*/
      chan_array[ch1].A!unavail;
      goto wait_teardown
    ::chan_array[ch2].A?avail; /*ringing tone*/
      chan_array[ch1].A!avail;
      goto connected
    fi;

wait_teardown:
    atomic{
        chan_array[ch2].A?teardown ->
        chan_array[ch2].B!downack;
        chan_array[ch1].A!teardown;
        goto unlinking_from_ch2
    }

connected:
    do
    ::atomic{
```

```
        chan_array[ch2].A?teardown->
        chan_array[ch2].B!downack;
        chan_array[ch1].A!teardown;
        goto unlinking_from_ch2
        }
    ::atomic{
        chan_array[ch1].B?teardown->
        chan_array[ch1].A!downack;
        chan_array[ch2].B!teardown->
        goto unlinking_from_ch1
        }
    od;

unlinking_from_ch2:
    do
    ::atomic{
        chan_array[ch1].B?teardown->
        chan_array[ch1].A!downack
         }
    ::atomic{
        chan_array[ch1].B?downack->
        goto end
         }
    od;
unlinking_from_ch1:
    do
    ::atomic{
        chan_array[ch2].A?teardown->
        chan_array[ch2].B!downack->
         }
    ::atomic{
        chan_array[ch2].A?downack->
        goto end
        }
    od;
end :skip
}
```

A.5.3 Call Forwarding process

```
proctype CF(byte thisnum)
{
 byte orig;         /*Source of the usage*/
 byte dest;
 unsigned ch2: L; /*the index of the next communication channel*/
 unsigned ch1: L; /*the index of the previous communication channel*/

/*  NOTES:                                                              *
*       ch1.A is the incoming sending channel                           *
*       ch1.B is the incoming receiving channel                         *
```

```
*       ch2.A is the outgoing receiving channel                             *
*       ch2.B is the outgoing sending channel                               */

begin:
  atomic{
    box_in?setup,orig,eval(thisnum),dest,ch1;
    chan_array[ch1].A!upack;
    C_assign(ch2);
    run routerCF(orig);
    if
    ::(CF_info[thisnum-1] == noone)->
        box_out!setup,orig,thisnum,dest,ch2
    ::else ->
        box_out!setup,orig,CF_info[thisnum-1],dest,ch2
    fi
  };

waiting_upack: chan_array[ch2].A?upack;

wait_vail:
    if
    ::chan_array[ch2].A?unavail;
      chan_array[ch1].A!unavail;
      goto wait_teardown
    ::chan_array[ch2].A?avail;
      chan_array[ch1].A!avail;
      goto connected
    fi;

wait_teardown:
    atomic{
        chan_array[ch2].A?teardown ->
        chan_array[ch2].B!downack;
        chan_array[ch1].A!teardown;
        goto unlinking_from_ch2
    }

connected:
    do
    ::atomic{
        chan_array[ch2].A?teardown ->
        chan_array[ch2].B!downack;
        chan_array[ch1].A!teardown;
        goto unlinking_from_ch2
        }
    ::atomic{
        chan_array[ch1].B?teardown->
        chan_array[ch1].A!downack;
        chan_array[ch2].B!teardown;
        goto unlinking_from_ch1
```

```
        }
    od;

unlinking_from_ch2:
    do
    ::atomic{
        chan_array[ch1].B?teardown->
        chan_array[ch1].A!downack
        }
    ::atomic{
        chan_array[ch1].B?downack->
        goto end
        }
    od;

unlinking_from_ch1:
    do
    ::atomic{
        chan_array[ch2].A?teardown->
        chan_array[ch2].B!downack;
        }
    ::atomic{
        chan_array[ch2].A?downack->
        goto end
        }
    od;
end:skip
}
```

A.5.4 Call Waiting process

```
proctype CW_router_watch (byte thisuserindex)
{
 byte thisport;                  /*the previous box's port*/
 byte scr2;byte scr3 ;byte scr4; /*scratch values*/
 unsigned i:L;             /*the channels identifier of the free channel*/


proctype CW_router_watch (byte thisuserindex)
{
 byte scr2;byte scr3 ;byte scr4; /*scratch values*/
 unsigned ch:L;            /*the channel identifier of the free channel*/

begin:
  atomic{
    box_in?setup,scr2,scr3,scr4,ch ->
    if
    ::(connCW[thisuserindex-1]<2) ->
        CW_channel[thisuserindex-1].intern!setup,scr2,scr3,scr4,ch;
        goto end
```

```
    ::(connCW[thisuserindex-1]==2) ->
        chan_array[ch].A!upack;
        chan_array[ch].A!unavail;
        chan_array[ch].A!teardown;
        goto wait_downack
    fi;
  }

wait_downack:
    chan_array[ch].B?downack;
end:skip
}

proctype CW(byte thisuser)
{
 bool calling;  /* denote subscriber calling or being called */
 bool talk;     /* non-deterministic: first/second party talking */
 byte orig;   byte number  ; byte dest;
 byte orig2;  byte number2 ; byte dest2;
 byte orig3;  byte number3 ; byte dest3;
 unsigned ch:L;     /* temporal channel */
 unsigned new:L;        /* temporal channel */
 unsigned first:L;  /* channel connecting CW box with first party */
 unsigned subsc:L;  /* channel connecting CW box with subscriber */
 unsigned second:L; /* channel connecting CW box with second party */

/*  NOTES:                                                                *
*       first channel connecting the CW box with the first party          *
*       subs channel connecting the CW box with the subscriber            *
*       second channel connecting the CW box with the second party        *
*       talk==true => first party talking                                 *
*       talk==false => first party waiting                                *
*       calling==true => subscriber is calling                            *
*       calling==false => subscriber is being called                      *
*
*   1)  waitsignal and switch signals are not implemented.                *
*   2)  NO change of global variables. e.g. busy[thisuser-1] = false;     *
*   3)  switch signal is abstracted and handled by the talk varible,      *
*  which changes only non-deterministically!                              */

endSet_up:
    talk=true;
end_bis:
  atomic{
    CW_channel[thisuser-1].intern?setup,orig,number,dest,ch;
    connCW[thisuser-1]=1;
    if  /*no fake id possible*/
    ::orig == thisuser ->       /* subscriber is calling */
        orig2=orig; number2=number; dest2=dest;
        calling=true;
```

```
        subsc=ch;
        chan_array[subsc].A!upack;
        C_assign(first);
        run routerCWsrc(thisuser);
        box_out!setup,orig,number,dest,first;
        goto wait_upack_src
    ::else ->                   /* subscriber is being called */
        orig3=orig; number3=number; dest3=dest;
        calling = false;
        first=ch;
        chan_array[first].A!upack;
        C_assign(subsc);
        run routerCWtrg(thisuser);
        box_out!setup,orig,number,dest,subsc;
        goto wait_upack_trg
    fi;
  }

wait_upack_src:
    chan_array[first].A?upack;

wait_vail_src:
    if
    ::atomic{
        chan_array[first].A?avail;
        chan_array[subsc].A!avail;
        goto con1src_first
        }
    ::atomic{
        chan_array[first].A?unavail;
        chan_array[subsc].A!unavail;
        }
      atomic{
        chan_array[first].A?teardown;
        chan_array[subsc].A!teardown;
        chan_array[first].B!downack;
        }
        goto wait_dn_src
    fi;

wait_dn_src:
    chan_array[subsc].B?downack;
    connCW[thisuser-1]=0;
    goto end;

wait_upack_trg:
    chan_array[subsc].A?upack;

wait_vail_trg:
    if
```

```
    ::atomic{
        chan_array[subsc].A?avail;
        chan_array[first].A!avail;
        goto con1trg_first
        }
    ::atomic{
        chan_array[subsc].A?unavail;
        chan_array[first].A!unavail;
        }
      atomic{
        chan_array[subsc].A?teardown;
        chan_array[first].A!teardown;
        chan_array[subsc].B!downack;
        }
        goto wait_dn_trg
    fi;

wait_dn_trg:
    chan_array[first].B?downack;
    connCW[thisuser-1]=0;
    goto end;

con1src_first:
    connCW[thisuser-1]=1;
    do
    ::atomic{
        chan_array[subsc].B?teardown;
        chan_array[subsc].A!downack;
        chan_array[first].B!teardown;
        goto unlink0src
        }
    ::atomic{
        chan_array[first].A?teardown ->
        chan_array[first].B!downack;
        chan_array[subsc].A!teardown;
        goto unlink1src
        }
    ::atomic{
        /* Another party gets involved in the usage */
        CW_channel[thisuser-1].intern?setup,orig,number,dest,ch;
        connCW[thisuser-1]=2;
        orig3=orig;
        number3=number;
        dest3=dest;
        second=ch;
        chan_array[second].A!upack;
        chan_array[second].A!avail;
        }
        if
        ::talk = true /*port 2 will be the talking one*/
```

```
        ::talk = false /*port 3 will be the talking one*/
        fi;
        goto con2src
        od;

con1src_second:
    connCW[thisuser-1]=1;
    do
    ::atomic{
        chan_array[subsc].B?teardown;
        chan_array[subsc].A!downack;
        chan_array[second].A!teardown;
        goto unlink2src
        }
    ::atomic{
        chan_array[second].B?teardown;
        chan_array[second].A!downack;
        chan_array[subsc].A!teardown;
        goto unlink1src
    }
    ::atomic{
        /* Another party gets involved in the usage */
        CW_channel[thisuser-1].intern?setup,orig,number,dest,ch;
        connCW[thisuser-1]=2;
        orig2=orig;
        number2=number;
        dest2=dest;
        first=ch;
        chan_array[first].A!upack;
        chan_array[first].A!avail;
        }
        if
        ::talk = true /*port 2 will be the talking one*/
        ::talk = false /*port 3 will be the talking one*/
        fi;
        goto con2src
        od;

unlink0src:
    do
    ::atomic{
        chan_array[first].A?teardown;
        chan_array[first].B!downack
        }
    ::atomic{
        chan_array[first].A?downack;
        connCW[thisuser-1]=0;
        goto end
        }
    od;
```

```
unlink1src:
    do
    ::atomic{
        chan_array[subsc].B?teardown;
        chan_array[subsc].A!downack
        }
    ::atomic{
        chan_array[subsc].B?downack;
        connCW[thisuser-1]=0;
        goto end
        }
    od;

unlink2src:
    do
    ::atomic{
        chan_array[second].B?teardown;
        chan_array[second].A!downack
        }
    ::atomic{
        chan_array[second].B?downack;
        connCW[thisuser-1]=1;
        goto end
        }
    od;


con1trg_first:
    connCW[thisuser-1]=1;
    do
    ::atomic{
        chan_array[subsc].A?teardown;
        chan_array[subsc].B!downack;
        if
        ::calling ->
            chan_array[first].B!teardown;
            goto unlink0src
        ::else ->
            chan_array[first].A!teardown;
            goto unlink0trg
        fi;
        }
    ::atomic{
        chan_array[first].B?teardown;
        chan_array[first].A!downack;
        chan_array[subsc].B!teardown;
        goto unlink1trg
        }
    ::atomic{
```

```
        chan_array[first].A?teardown ->
        chan_array[first].B!downack;
        chan_array[subsc].B!teardown;
        goto unlink1trg
        }
    ::atomic{
        /* Another party gets involved in the usage */
        CW_channel[thisuser-1].intern?setup,orig,number,dest,ch;
        connCW[thisuser-1]=2;
        orig3=orig;
        number3=number;
        dest3=dest;
        second=ch;
        chan_array[second].A!upack;
        chan_array[second].A!avail;
        }
        if
        ::talk = true /*port 2 will be the talking one*/
        ::talk = false /*port 3 will be the talking one*/
        fi;
        goto con2trg
    od;

con1trg_second:
    connCW[thisuser-1]=1;
    do
    ::atomic{
        chan_array[subsc].A?teardown;
        chan_array[subsc].B!downack;
        chan_array[second].A!teardown;
        goto unlink2trg
        }
    ::atomic{
        chan_array[second].B?teardown;
        chan_array[second].A!downack;
        chan_array[subsc].B!teardown;
        goto unlink1trg
        }
    ::atomic{
        /* Another party gets involved in the usage */
        CW_channel[thisuser-1].intern?setup,orig,number,dest,ch;
        connCW[thisuser-1]=2;
        orig2=orig;
        number2=number;
        dest2=dest;
        first=ch;
        chan_array[first].A!upack;
        chan_array[first].A!avail;
        }
        if
```

```
        ::talk = true /*port 2 will be the talking one*/
        ::talk = false /*port 3 will be the talking one*/
        fi;
        goto con2trg
        od;

unlink0trg:
    do
    ::atomic{
        chan_array[first].B?teardown;
        chan_array[first].A!downack
        }
    ::atomic{
        chan_array[first].B?downack;
        connCW[thisuser-1]=0;
        goto end
        }
    od;

unlink1trg:
    do
    ::atomic{
        chan_array[subsc].A?teardown;
        chan_array[subsc].B!downack
        }
    ::atomic{
        chan_array[subsc].A?downack;
        connCW[thisuser-1]=0;
        goto end
        }
    od;

unlink2trg:
    do
    ::atomic{
        chan_array[second].B?teardown;
        chan_array[second].A!downack
        }
    ::atomic{
        chan_array[second].B?downack;
        connCW[thisuser-1]=0;
        goto end
        }
    od;

con2src:
    do
    ::atomic{   /* CASE 1: subscriber sends teardown=>CALL BACK PROCESSING */
        chan_array[subsc].B?teardown->
        chan_array[subsc].A!downack
```

```
        }
        connCW[thisuser-1]=1;
        if
        ::talk->    /**connecting to second party**/
            chan_array[first].B!teardown;
            goto unlink3src
        ::else ->   /**connecting to first party**/
            chan_array[second].A!teardown;
            goto unlink4src
        fi;

setup_sec_src:              /**connecting to second party**/
        atomic{
            run routerCWtrg(thisuser);
            C_assign(ch);
            box_out!setup,orig,number,dest,ch
            }
        do
            /* WAIT_UPACK from subscriber*/
        ::chan_array[ch].A?upack;
            goto wait_response_second_src
        ::CW_channel[thisuser-1].intern?setup,orig,number,dest,new;
            connCW[thisuser-1]=2;
            if
            /* SITUATION 2) Subscriber calls somebody else [1][2] */
            ::orig == thisuser ->
                orig2=orig; number2=number; dest2=dest;
                calling=true;
                subsc=new;
                chan_array[subsc].A!upack;
                C_assign(first);
                run routerCWsrc(thisuser);
                box_out!setup,orig,number,dest,first;
                goto unlink_old_subsc_sec
            /* SITUATION 3) Somebody else calls subscriber [3][4] */
            ::else ->
                orig3=orig; number3=number; dest3=dest;
                calling = false;
                first=new;
                chan_array[first].A!upack;
                chan_array[first].A!avail;
                goto link_nonsubsc_sec
            fi;
        od;

unlink_old_subsc_sec: /* SITUATION 2 */
            atomic{ /* Subscriber as in trg scenario */
                chan_array[ch].A?upack;
                chan_array[ch].A?unavail;
                chan_array[ch].A?teardown;
```

```
                chan_array[ch].B!downack;
                goto wait_upack_sec2
            }

wait_upack_sec2:
            chan_array[first].A?upack;

wait_vail_sec2:
            if
            ::atomic{
                chan_array[first].A?avail;
                chan_array[subsc].A!avail;
                if
                ::talk = true /*port 2 will be the talking one*/
                ::talk = false /*port 3 will be the talking one*/
                fi;
                goto con2src
                }
            ::atomic{
                chan_array[first].A?unavail;
                chan_array[subsc].A!unavail;
                chan_array[first].A?teardown;
                chan_array[first].B!downack;
                }
                goto con1src_second
            fi;

link_nonsubsc_sec: /* SITUATION 3 */
            subsc=ch;
            chan_array[subsc].A?upack;

wait_response_sec2:
            if
            ::atomic{
                chan_array[first].B?avail;
                chan_array[subsc].B!avail;
                if
                ::talk = true /*port 2 will be the talking one*/
                ::talk = false /*port 3 will be the talking one*/
                fi;
                goto con2trg
                }
            ::atomic{
                chan_array[first].B?unavail;
                chan_array[first].B?teardown;
                chan_array[first].A!downack;
                }
                goto con1trg_second
            fi;
```

```
wait_response_second_src: /*Normal situation:connecting to second party*/
        do
        ::chan_array[ch].A?avail;
            subsc=ch;
            goto con1trg_second
        ::chan_array[ch].A?unavail;
            chan_array[ch].A?teardown;
            chan_array[ch].B!downack;
            chan_array[second].A!teardown;
            subsc=ch;
            goto unlink2src
        od;

setup_first_src:            /**connecting to first party**/
        atomic{
            run routerCWtrg(thisuser);
            C_assign(ch);
            if
            ::orig==thisuser ->
                box_out!setup,dest,orig,orig,ch
            ::else ->
                box_out!setup,orig,number,dest,ch
            fi;
            }
        do
            /* WAIT_UPACK from subscriber*/
        ::chan_array[ch].A?upack;
            goto wait_response_first_src

        ::CW_channel[thisuser-1].intern?setup,orig,number,dest,new;
            connCW[thisuser-1]=2;
            if
            /* SITUATION 2) Subscriber calls somebody else [5][6] */
            ::orig == thisuser ->
                orig2=orig; number2=number; dest2=dest;
                calling=true;
                subsc=new;
                chan_array[subsc].A!upack;
                C_assign(second);
                run routerCWsrc(thisuser);
                box_out!setup,orig,number,dest,second;
                goto unlink_old_subsc_first_src
            /* SITUATION 3) Somebody else calls subscriber [7][8] */
            ::else ->
                orig3=orig; number3=number; dest3=dest;
                calling = false;
                second=new;
                chan_array[second].A!upack;
                chan_array[second].A!avail;
                goto link_nonsubsc_first
```

```
            fi;
        od;

unlink_old_subsc_first_src: /* SITUATION 2 */
            atomic{ /* Subscriber as in trg scenario */
                chan_array[ch].A?upack;
                chan_array[ch].A?unavail;
                chan_array[ch].A?teardown;
                chan_array[ch].B!downack;
                goto wait_upack_first_2src
            }

wait_upack_first_2src:
            chan_array[second].B?upack;

wait_vail_first_2src:
            if
            ::atomic{
                chan_array[second].B?avail;
                chan_array[subsc].A!avail;
                if
                ::talk = true /*port 2 will be the talking one*/
                ::talk = false /*port 3 will be the talking one*/
                fi;
                goto con2src
                }
            ::atomic{
                chan_array[second].B?unavail;
                chan_array[subsc].A!unavail;
                chan_array[second].B?teardown;
                chan_array[second].A!downack;
                }
                goto con1src_first
            fi;

link_nonsubsc_first: /* SITUATION 3 */
            subsc=ch;
            chan_array[subsc].A?upack;

wait_response_first2:
            if
            ::atomic{
                chan_array[second].B?avail;
                chan_array[subsc].B!avail;
                if
                ::talk = true /*port 2 will be the talking one*/
                ::talk = false /*port 3 will be the talking one*/
                fi;
                goto con2trg
                }
```

```
            ::atomic{
                chan_array[second].B?unavail;
                chan_array[second].B?teardown;
                chan_array[second].A!downack;
                }
                goto con1trg_first
            fi;

wait_response_first_src: /*Normal situation:connecting to first party*/
        do
        ::chan_array[ch].A?avail;
            subsc=ch;
            goto con1trg_first
        ::chan_array[ch].A?unavail;
            chan_array[ch].A?teardown;
            chan_array[ch].B!downack;
            chan_array[first].B!teardown;
            subsc=ch;
            goto unlink0src
        od;

    ::atomic{   /* CASE 2: first party sends teardown */
        chan_array[first].A?teardown->
        chan_array[first].B!downack;
        goto con1src_second
        }
    ::atomic{   /* CASE 3: second party sends teardown */
        chan_array[second].B?teardown->
        chan_array[second].A!downack;
        connCW[thisuser-1]=1;
        goto con1src_first
        }
    od;

unlink3src:
    do
    ::atomic{
        chan_array[first].A?teardown;
        chan_array[first].B!downack
        }
    ::atomic{
        chan_array[first].A?downack;
        connCW[thisuser-1]=0;
        goto setup_sec_src
        }
    od;

unlink4src:
    do
    ::atomic{
```

```
        chan_array[second].B?teardown;
        chan_array[second].A!downack
        }
    ::atomic{
        chan_array[second].B?downack;
        connCW[thisuser-1]=1;
        goto setup_first_src
        }
    od;

con2trg:
    do
    ::atomic{   /* CASE 1: subscriber sends teardown=>CALL BACK PROCESSING */
        chan_array[subsc].A?teardown->
        chan_array[subsc].B!downack
        }
        connCW[thisuser-1]=1;
        if
        ::talk->        /**connecting to second party**/
            chan_array[first].A!teardown;
            goto unlink3trg
        ::else ->       /**connecting to first party**/
            chan_array[second].A!teardown;
            goto unlink4trg
        fi;

setup_sec_trg:              /**connecting to second party**/
        atomic{
            run routerCWtrg(thisuser);
            C_assign(ch);
            box_out!!setup,orig,number,dest,ch
            }
        do
            /* WAIT_UPACK from subscriber*/
        ::chan_array[ch].A?upack;
            goto wait_response_second_trg

        ::CW_channel[thisuser-1].intern?setup,orig,number,dest,new;
            connCW[thisuser-1]=2;
            if
            /* SITUATION 2) Subscriber calls somebody else [9][10] */
            ::orig == thisuser ->
                orig2=orig; number2=number; dest2=dest;
                calling=true;
                subsc=new;
                chan_array[subsc].A!upack;
                C_assign(first);
                run routerCWsrc(thisuser);
                box_out!setup,orig,number,dest,first;
                goto unlink_old_subsc_sec /* Same as src scenario */
```

```
            /* SITUATION 3) Somebody else calls subscriber [11][12] */
            ::else ->
                orig3=orig; number3=number; dest3=dest;
                calling = false;
                first=new;
                chan_array[first].A!upack;
                chan_array[first].A!avail;
                goto link_nonsubsc_sec  /* Same as src scenario */
            fi;
        od;

        /* SITUATION 2: Same as src scenario */
        /* SITUATION 3: Same as src scenario */

wait_response_second_trg: /*Normal situation:connecting to second party*/
        do
        ::chan_array[ch].A?avail;
            subsc=ch;
            goto con1trg_second
        ::chan_array[ch].A?unavail;
            chan_array[ch].A?teardown;
            chan_array[ch].B!downack;
            chan_array[second].A!teardown;
            subsc=ch;
            goto unlink2trg
        od;

setup_first_trg:            /**connecting to first party**/
        atomic{
            run routerCWtrg(thisuser);
            C_assign(ch);
            if
            ::orig==thisuser ->  /*REVERSE SETUP ??? double !!*/
                box_out!setup,dest,orig,orig,ch
            ::else ->
                box_out!!setup,orig,number,dest,ch
            fi;
            }
        do
            /* WAIT_UPACK from subscriber*/
        ::chan_array[ch].A?upack;
            goto wait_response_first_trg

        ::CW_channel[thisuser-1].intern?setup,orig,number,dest,new;
            connCW[thisuser-1]=2;
            if
            /* SITUATION 2) Subscriber calls somebody else [13][14] */
            ::orig == thisuser ->
                orig2=orig; number2=number; dest2=dest;
                calling=true;
```

```
                subsc=new;
                chan_array[subsc].B!upack;
                C_assign(second);
                run routerCWsrc(thisuser);
                box_out!setup,orig,number,dest,second;
                goto unlink_old_subsc_first_trg
            /* SITUATION 3) Somebody else calls subscriber [15][16] */
            ::else ->
                orig3=orig; number3=number; dest3=dest;
                calling = false;
                second=new;
                chan_array[second].A!upack;
                chan_array[second].A!avail;
                goto link_nonsubsc_first
            fi;
        od;

unlink_old_subsc_first_trg: /* SITUATION 2 */
            atomic{ /* Subscriber as in trg scenario */
                chan_array[ch].A?upack;
                chan_array[ch].A?unavail;
                chan_array[ch].A?teardown;
                chan_array[ch].B!downack;
                goto wait_upack_first_2trg
            }

    /*
     * NOTE: In connecting to first party, channels for first
     *  wouldn't correspond to the src scenario, so we need to
     *  treat subscriber channels as in trg scenario, which
     *  lead us in "con2trg" or "con1trg_first" states.
     */
wait_upack_first_2trg:
            chan_array[second].B?upack;

wait_vail_first_2trg:
            if
            ::atomic{
                chan_array[second].B?avail;
                chan_array[subsc].B!avail;
                if
                ::talk = true /*port 2 will be the talking one*/
                ::talk = false /*port 3 will be the talking one*/
                fi;
                goto con2trg
                }
            ::atomic{
                chan_array[second].B?unavail;
                chan_array[subsc].B!unavail;
                chan_array[second].B?teardown;
```

```
                chan_array[second].A!downack;
                }
                goto con1trg_first
            fi;

        /* SITUATION 3: Same is in src scenario */

wait_response_first_trg: /*Normal situation:connecting to first party*/
        do
        ::chan_array[ch].A?avail;
            subsc=ch;
            goto con1trg_first
        ::chan_array[ch].A?unavail;
            chan_array[ch].A?teardown;
            chan_array[ch].B!downack;
            chan_array[first].A!teardown;
            goto unlink0trg
        od;

    ::atomic{   /* CASE 2: first party sends teardown */
        chan_array[first].B?teardown->
        chan_array[first].A!downack;
        connCW[thisuser-1]=1;
        goto con1trg_second
        }
    ::atomic{   /* CASE 3: second party sends teardown */
        chan_array[second].B?teardown->
        chan_array[second].A!downack;
        connCW[thisuser-1]=1;
        goto con1trg_first
        }
    od;

unlink3trg:
    do
    ::atomic{
        chan_array[first].B?teardown;
        chan_array[first].A!downack
        }
    ::atomic{
        chan_array[first].B?downack;
        connCW[thisuser-1]=0;
        goto setup_sec_trg
        }
    od;

unlink4trg:
    do
    ::atomic{
        chan_array[second].B?teardown;
```

```
        chan_array[second].A!downack
        }
    ::atomic{
        chan_array[second].B?downack;
        connCW[thisuser-1]=0;
        goto setup_first_trg
        }
    od;

end:
    goto endSet_up
}
```