Verification of DFC Call Protocol Correctness Criteria

by

Alma L. Juarez Dominguez

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2005

I hereby declare that I am the sole author of this thesis.
I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Alma L. Juarez Dominguez

I further authorize the University of Waterloo to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Alma L. Juarez Dominguez

Abstract

Distributed Feature Composition (DFC) is an architecture developed by Jackson and Zave at AT&T to describe and implement telecommunication services. DFC supports modular development, where features that are independently implemented can be composed. The main goal of our work is to create a definition of DFC compliance for features with respect to the signalling call protocol using a set of linear temporal logic (LTL) properties and to verify that our proposed protocol properties hold for segments of DFC with n features communicating through unbounded queues. Our main contribution is a compositional reasoning method that decomposes the problem into two parts: (1) Model check features individually in an abstract environment to verify that they are DFC compliant with respect to the call protocol, and (2) Use the properties of the individual features in an inductive proof to conclude that the call protocol properties hold over segments of the DFC architecture.

Acknowledgements

I would like to thank my supervisor, Professor Nancy A. Day, for her guidance, support, inspiration, and all the amount of time she spent in my thesis work. Nancy, you are a great example to follow!

Many thanks to Professor Joanne M. Atlee, and Professor Daniel M. Berry, for taking the time to read my thesis. All their comments helped to improve my thesis.

I want to express my gratitude to Pamela Zave for her valuable feedback, suggestions and enjoyable discussions along the way.

Funding for this work was provided by the Consejo Nacional de Ciencia y Tecnología (CONACYT) of México, Dirección General de Relaciones Internacionales de la Secretaría de Educacin Pública of México, Communications and Information Technology Ontario (CITO) of Canada, and the University of Waterloo.

Thanks also go to the members of the WatForm lab for providing me with valuable discussions and interesting conversations. A special recognition to Wenceslas Godard for his work on early stages of the creation of the Promela model.

Thanks to all my friends in Puebla and in Waterloo for helping me to be a better person and making my life enjoyable. In particular, I would like to recognize the endless support of my friends Angelica Alvarez, Elodie Fourquet, Joyce Nieuwesteeg, Arturo Santillan, and Theodoro Koulis. Angie, our friendship has always been a blessing for me. Elodie, thanks for being there when I need it, in the good and the bad times. Joyce, your tuition not only helped me to achieve this goal, but also enriched my life. Arturo, thanks for getting always to the point, no matter if that hurt me at the beginning. Theo, thanks for always provide me with an amusing and interesting conversation. Your patience, support and friendship have made my life special. And for all the members of #beer, my sincere appreciation for letting me be part of the club and remind me that I am not alone in this crusade.

Finally, I would like to thank my parents, Alma Domínguez and Rafael Juárez, for their outstanding support, for encouraging me to pursue my dreams, but above all for their unconditional love. You have been not only excellent parents, but also the best friends I could have ever found. Thanks go to my siblings, Karen and Rafael, for showing me how to laugh and live my life intensely.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An often quoted remark by Dijkstra is that "Program testing can be used to show the presence of bugs, but never their absence" [5]. To ensure that a system is not only free of bugs, but also meets its requirements without unexpected behaviours, we need the tools of formal methods. Formal analysis techniques perform exhaustive analysis, so they are of special interest to verify correctness requirements for distributed systems, such as telephony systems, which involve concurrently executing processes.

In this thesis, we present a method to verify aspects of the call protocol for Jackson and Zave's Distributed Feature Composition (DFC) [12] architecture, which is used by AT&T in its telecommunication services. In this chapter, we provide an overview of our method, list the contributions of this thesis, and discuss related work. Finally, we give a brief description of the organization of this thesis.

1.1 Thesis Description

DFC is an architecture for coordinating telephony features based on a pipe-and-filter model, developed by Jackson and Zave at AT&T [12]. A *feature* box is a function for the users of a system, performed on top of basic services. An example of a feature in the telephony domain is call waiting. The architecture is a distributed system in which each feature runs as a process and communicates with its neighbours using signals passed along communication channels, following the call protocol. A user subscribes to features, which are placed in an order based on precedence information.

A *feature interaction* occurs when the execution of one feature interferes with the work or modifies the behaviour of another feature, although not all feature interactions are bad. The innovation in DFC is that by placing a structure on the way the features interact, some undesirable feature interactions can be avoided, and others can be predicted because the features respond to signals in the order of the usage. The DFC rules on how features interact are strong enough to guarantee that basic call functionality is preserved, since the usage is created and can be torn down, but flexible enough to allow individual feature behaviour. Much research has been done on trying to detect and avoid feature interactions in telecommunication services (*e.g.,* [20, 21, 8, 7]).

In this work, we seek to identify correctness criteria for DFC features concentrating on the call protocol, *i.e.,* we define what is good behaviour in the system, rather than detecting feature interactions. We use *liveness* properties to express that eventually "something good must happen" during a system execution, a notion first introduced by Lamport [16]. To describe properties, we use temporal logic, which has proved to be useful for describing the behaviour of concurrent systems, since temporal logics let us describe the order in which events occur without using time explicitly [14]. Our correctness properties are expressed in linear temporal logic (LTL).

In DFC, a *usage* is a dynamic assembly of *boxes* connected by channels. A usage describes the response to a request for a telecommunication service at a certain time. A box is either an interface box, *e.g.,* Caller or Callee, or a feature subscribed to by users, *e.g.,* call forwarding. An interface box provides an interface to a physical device to communicate to users or to another network. A feature box is either free or bound. A *free* feature box is one for which a new instance of it is generated every time the feature is to be included in a usage, *e.g.,* call forwarding. A *bound* feature box is dedicated to a particular address, and even if it is already in use within a usage, the same feature box is made part of the new usage, *e.g.,* call waiting.

A feature box has the authority to influence the routing process, leading to linear or branching configurations, and changing the number of processes involved in a usage, depending on the signals received. Therefore, we propose the following characterization of the feature boxes that affect routing: A *transparent* box does not affect routing, not absorbing or modifying signals, but passing them along; A *user agent* box can affect routing, by requesting the creation of a usage or by responding to such request. Using this categorization, we describe properties of DFC *segments*, each of which consists of a linear arrangement of boxes from user agent to user agent without a user agent in between. Our problem consists of verifying that our proposed call protocol properties hold for segments with n feature boxes, each following the call protocol, and each communicating through unbounded queues.

We first check our proposed call protocol properties on fixed DFC segments, each having a fixed number of Callers, Callees, and feature boxes. We chose the model checker Spin [9] to develop a model of DFC, because the tool has explicit support for queues and process interleaving, matching DFC behaviour. *Model checking* is an automatic technique to verify the correctness of finite state concurrent systems by exhaustively exploring all behaviours of the system to check if a property holds or not.

Unlike past work modeling DFC in Spin [22], the environment we developed to model check the DFC architecture creates usages dynamically, which allows the model to closely match DFC's behaviour, using the subscription and precedence information to construct a usage. This work allows us to debug our proposed properties. However, even if this verification effort is appropriate for specific instances of DFC usages, we cannot conclude that all configurations with any number of features will satisfy our correctness properties.

Since DFC consists of a set of independently executing processes, we would like to be able to use compositional reasoning to verify the call protocol properties. Usually, compositional reasoning suffers from dependency problems between components. To prove properties of one component, we must assume properties of another component and vice versa, which leads to proof rules for assume-guarantee style reasoning in which care must be taken to avoid circular arguments (*e.g.,* [17]). The semi-regularity of DFC features alleviates this problem: the environment of every feature consists of other features boxes or an interface box such as a Caller or Callee process. By the term semi-regularity we mean that the DFC architecture requires all features to react to certain signals in the same way, while leaving other signals to be used in feature-specific behaviours. If we can capture the most general behaviour of any possible neighbouring box in an abstract environment, we can (1) verify each box individually in the abstract environment, and then (2) show that every box conforms to the behaviour of the abstract environment. This separation of concerns makes it possible to create a compositional method for verifying properties of usages to accompany the existing modularity of the DFC architecture.

We develop a compositional reasoning method to check that our proposed protocol properties hold for any segment with n boxes following the call protocol, and communicating through unbounded queues. Our approach decomposes the problem into two parts: (1) verifying that individual boxes behave as intended, being DFC compliant with respect to the call protocol, and (2) using induction over n boxes, combining box properties with abstract models of the queues to conclude the call protocol properties.

The verification of individual boxes in Spin involves three parts. First, we use model checking to verify DFC-compliance properties of individual boxes, described in LTL. We use an unrestricted environment that can send and receive any call protocol signal at any time. We employ synchronous communication, *i.e.,* the channel capacity is zero and cannot store messages, so when a send statement on a channel is executed, the corresponding receive operation must be executed next. The use of synchronous communication has the advantage of providing a reduction of the state space, abstracting away the behaviour of the channels and neighbouring features in this step.

Second, we use model checking to verify that a box interacting with an environment of neighbouring boxes receives only the signals it is expecting and sends only the signals expected by the environment, because synchronous composition implicitly assumes that the box receives and sends signals only when they are expected. We describe these expected I/O properties in LTL as invariants. Rather than consider the box in all possible environments (*i.e.,* all possible boxes as neighbours), to model the neighbouring boxes we have created an abstract environment that captures the most general DFC port behaviour. We apply asynchronous communication, *i.e.,* the channel capacity is at least one, storing messages that do not need to be read immediately, which allows interleaving.

Third, we check for port compliance, which means that every port of any box conforms to the most general DFC port behaviour used in the abstract environment. Port compliance checking is done using language containment. By proving that the use of abstract environments in the verification of the expected I/O property is a sound abstraction, it is sufficient to verify the individual boxes with an abstract environment.

We use induction to show that if all boxes satisfy their required individual behaviours and the queues behave correctly, then the segment properties are satisfied. We mechanized the proof using the HOL theorem prover [6]. *Theorem proving* is the approach of using a computer to support deductive reasoning. Only properties are used to describe the behaviour of boxes, so we do not need to represent any complete description of box models in HOL. The LTL properties for boxes and queues are stated in HOL and induction is applied over natural numbers, which allow us to represent the size of the segment. The inductive reasoning step needs to be completed only once.

Our thesis statement is:

> **A correctness criteria for the call protocol for any DFC segment can be described using linear temporal logic. A compositional reasoning approach, combining model checking, theorem proving and language containment, makes it possible to reason about DFC boxes individually in an abstract environment, and then to deduce the call protocol properties about any DFC segment with n boxes by induction.**

1.2 Contributions

The main contributions of this thesis are:

- A proposed definition of DFC compliance with respect to the call protocol, using a set of LTL properties.
- An environment for model checking fixed configurations of DFC box models in the model checker Spin. The creation of the configuration is done dynamically, following closely the DFC architecture.
- The detection of surprising feature behaviours during the verification of fixed configurations. The behaviours discovered were not specified in the feature's descriptions given in the literature, *e.g.,* scenarios for the call waiting subscriber when several parties are calling and hanging up.
- A compositional reasoning approach for DFC that consists of four steps:
  - Verify, using model checking, that each box reacts to the call signals in agreement with the DFC architecture constraints, using an unrestricted environment and synchronous communication.
  - Verify, using model checking, that each box receives and sends only the call signals it expects, using an abstract environment that captures the most general DFC port protocol behaviour and asynchronous communication.
  - Prove that every port of any box conforms to the most general DFC port behaviour used in the abstract environment. We proved the language containment relationships between a partial order of abstract port models so that the most specific abstract port model could be used for checking the port compliance of a box.
  - Prove by induction over the length of a segment that the call protocol properties proposed about the DFC architecture hold.

1.3 Related Work

In this section, we briefly overview other efforts to verify DFC-related artifacts.

An interesting motivation for the application of formal methods to DFC was described by Zave [23]. Zave explores feature oriented and architectural system descriptions for DFC and explains the need to use formal methods to support these descriptions. More recently, she introduced a feature-oriented specification technique that follows the structural approach to detecting feature interactions by relying on properties of individual feature programs [24]. We follow a similar method, but looking for requirements of good behaviour instead of detecting feature interactions. Our work checks that individual boxes are DFC compliant, and then we apply induction to confirm our call protocol properties.

Zave provided also a formal description of the service layer of a telecommunication system, organized according to the DFC architecture, using PROMELA and Z [22]. 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 closely match DFC's behaviour. As in Zave's work, the first step of our verification approach is to check for the absence of deadlock in the model, but we check also our call protocol properties for segments and provide a compositional approach for checking the properties on segments of length n.

A Java implementation of DFC in an IP setting, called ECLIPSE, was developed at AT&T Labs, and the Mocha model checker [1] was used to verify the communication protocols [3]. Individual ECLIPSE feature box code was translated to the modeling language framework of Mocha automatically. Similar to our work, the verification consists of combining a box with standardized environmental peer entities of caller, callee and dual ports. At AT&T, a check for deadlock using synchronous communication between the feature and its environment was performed. The example described in the paper involves the analysis of a free transparent feature box, and there is no discussion of analyzing bound feature boxes. We extend this work by checking liveness properties, as well as taking the step of showing that all box behaviour is contained within an abstract model, which captures the most general DFC port behaviour. The dual port peer in the ECLIPSE work reflects the combined behaviour of a caller and callee ports, as described in the DFC manual. However, a dual port does not take into account busy processing, which is part of our abstract models. We present a partial order among these abstract models. Finally, we also state box and segment properties and use inductive reasoning to conclude the segment properties.

Zave and Jackson have created a domain-specific programming language called Boxtalk to simplify the expression of DFC feature behaviour [30]. Boxtalk encourages correct programming of coordinating components and provides the foundation to prove behavioural equivalence. This work lists a set of well-formedness properties that every box should follow to produce meaningful global behaviour in a program.

Other abstractions, leading to theorems and analysis concerning feature interactions have also been introduced. Zave defined a set of constraints on feature behaviour called "ideal address translation" [26]. Adherence to the constraints results in provable properties such as preservation of anonymity. We are not dealing directly with the problem of address translation, so the results are not directly related to our work. Zave presents also a formal specification and properties of the new routing algorithm [25]. These properties are defined in terms of an abstraction called "ideal connection paths". However in our work we are not modeling the behaviour of the router, so we do not use the properties of this abstraction.

There is also significant work in searching for feature interactions in telephony systems (*e.g.,* [2]). In the context of DFC, Zimmer and Atlee have been studying categorizing features for resolving feature interactions [31].

The combination of Spin and Hol has been used to analyze routing protocols [15] where Hol was used to justify abstractions to a finite number of processes.

1.4 Thesis Organization

In Chapter 2, we provide background on the Distributed Feature Composition (DFC) architecture, the Spin model checker and the Hol theorem prover. In Chapter 3, we describe our model of DFC in Promela. Our proposed correctness criteria described in LTL as call protocol properties is presented in Chapter 4, and the process of model checking these properties on fixed configurations is explained in Chapter 5. We discuss the compositional reasoning approach in Chapter 6, and conclude with a summary, discussion of limitations and future work in Chapter 7. We provide the DFC models in Spin as well as language containment proofs and a glossary of terms in the Appendices.

Chapter 2

Background

In this chapter, we provide the background needed to understand the rest of the thesis. First, we give a brief overview of the Distributed Feature Composition (DFC) architecture. Next, we describe the main characteristics of the Spin model checker and the Hol theorem prover, which are used to prove our claims.

2.1 DFC Overview

DFC is an architecture developed by Jackson and Zave at AT&T to describe and implement telecommunication services [12]. Our description of DFC is based on the material found in the DFC Manual [13], DFC modifications [29, 28], and early papers describing the architecture [12, 23, 30]. The architecture coordinates telephony features based on a pipe-and-filter model. A *feature* is a function for the users of a system, performed on top of basic services. An example of a feature in the telephony domain is call waiting. DFC's main characteristic is modularity, where modules are independently implemented components that can be structurally composed.

In DFC, a *usage* describes the response to a request for a telecommunication service at certain time. A usage can be seen as a dynamically assembled graph composed of boxes and internal calls, which are unbounded queues communicating with boxes. This graph can change over time, in response to environmental events. An example is given in Figure 2.1.

To understand and reason about usages, the graph is divided into regions. A *source region* involves all the feature boxes subscribed to by the calling customer (at the source address). A *target region* involves all the feature boxes subscribed to by the customer

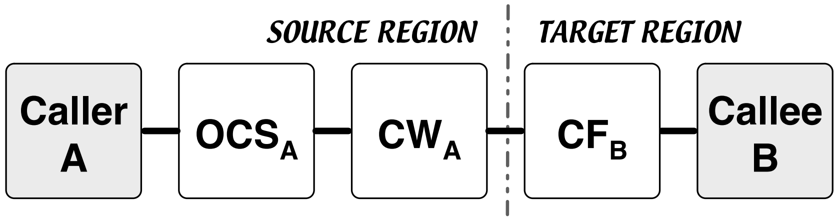

Figure 2.1: Example of a Usage

being called (at the target address). Figure 2.1 shows an example of a usage created when user A calls user B. User A subscribes to originating call screening (OCS) and call waiting (CW) as source features and user B subscribes to call forwarding (CF) as a target feature. If someone else calls A while A is talking to B, that user gets connected to the CW feature of A. The components, as well as the way they interact to respond to a customer telecommunication request, will be explained in the following subsections.

2.1.1 Signals

A *signal* is a message of the DFC call protocol. It has a signal type and a set of named, typed fields. Optional fields can be programmer-defined fields that are included to contain feature-specific information. The primary types of signals used in the call protocol are:

- `setup`: To request the setup of a telecommunication service, creating an internal call via the router.
- `upack`: To acknowledge the receipt of a `setup` signal.
- `teardown`: To request the end of an existing usage.
- `downack`: To acknowledge the receipt of a `teardown` signal.

A `setup` signal requires the following fields:

- *regn*: source region, or target region.
- *src*: the sender's address.
- *dld*: the dialed number.
- *trg*: the destination address.
- *route*: record used by the routing algorithm.

Status signals are used to provide information about the state of the current usage. The status signals in the call protocol are:

- `avail`: To indicate that the target address is available and the usage is deemed to be successful, establishing a voice channel between Caller and Callee.
- `unavail`: To indicate the target address is not available, stimulating a busy tone if none of the feature boxes in the usage offers a "busy treatment", *e.g.,* call forwarding on busy or voice mail.
- `unknown`: To indicate that the target address does not map to any interface box, stimulating an error tone.
- `none`: To cancel the effect of any of the other three previous signals on a user interface, means "no outcome yet".

2.1.2 Boxes

A *box* is a process that performs either interface or feature functions. *Interface boxes* (*e.g.,* Caller or Callee) provide an interface to physical devices to communicate to users or to other networks. *Feature boxes* are either free or bound. A *free* feature box is one for which a new instance is generated every time the feature is to be included in a usage. An example of a free feature box is call forwarding, which is not persistent and gets created upon request. Free feature boxes normally have two ports, so we call them *two-way* feature boxes. A *bound* feature box is dedicated to a particular address, and even if it is already in use within an existing usage, the same feature box is made part of the new usage. An example of a bound feature box is call waiting, which is a persistent process, and can be part of the source or target region of a usage. If a call waiting box is involved in a usage, and the subscriber is being called, the routing process should go through the call waiting box that is already in use. Bound feature boxes normally have three or more ports. We call boxes with three ports *three-way* feature boxes.

The scope of DFC is delimited by interface boxes, which are connected to physical devices to communicate among networks that use different protocols to exchange information. However, inside this boundary, the components are arranged in a pipe-and-filter style and use only the DFC call protocol to handle signals passing through the composed features boxes.

2.1.3 Calls

The interaction that a box has with a communication channel connected to another box is called a *port*. An internal *call* is an communication channel between ports of two different boxes, transmitting signals between boxes in first in–first out (FIFO) order and following the DFC call protocol. The end of the channel connected to the box that initiates a call by sending a `setup` is called a *caller port*, and the port on the other end of the communication channel is called a *callee port*.

2.1.4 Global Data

In this section, we describe the information that is available to all components in order to generate a usage. The way this information is used to build usages is explained in the next section.

Subscriptions: A telephony system provides a set of features. Each customer chooses which features to subscribe to. These choices are captured in the subscription data as a mapping from addresses to feature boxes. In DFC, there are two kinds of subscriptions: `srcSubscribes`, which is an association between an address and the feature boxes in its source region, and `trgSubscribes`, which is an association between an address and the feature boxes in its target region. Some feature boxes can only be placed either in the source or target region.

Precedences: A relation that constrains the order in which feature boxes are placed in the source and target region. This relation is a partial order, thus it helps restrict the number of feature interactions that can occur in a usage.

Operational data: Data relations that can be read and written by feature boxes. Operational data is meant to provide information needed by a particular feature, or by a certain customer, *e.g.,* retrieving the subscriber's forwarding address when a feature requires it.

2.1.5 Router

The *router* helps in the generation of the usage, setting up the communication channels between boxes. This process is done by receiving, modifying and sending `setup` signals. The router selects the next box that will be part of the usage by the application of the routing algorithm to the setup signal, taking into account subscription information as well as feature precedences. For each channel to be created, a `setup` signal from a box is sent to any DFC router, which builds the whole route, *i.e.,* the routing list of all the boxes to be part of the usage. Because usages can be forked and joined, some steps of the routing algorithm repeat more than once while constructing the routing list.

The DFC routing algorithm [13] is as follows:

Step1 Extract the target address from the dialed string.

Step2 Expand the `route` field of the `setup` signal to contain the sequence of feature boxes for the corresponding region (source or target) that are expected to be next in the usage.

Step3 Advance to the next region once the sequence of feature boxes in the current region is exhausted. Thus repeat Step 2 and 3 until all regions to be part of the usage are in the route field.

Step4 Choose a feature box whose box type matches the head of the `route`, and send the modified `setup` signal to it. If the `route` is empty, route to an interface box corresponding to the target address.

When a router must incorporate a free feature box into a usage, it creates a new instance of the feature box. When a router must incorporate a bound feature box into a usage, only the unique box of appropriate type bound to the appropriate address may be chosen. The router always finds a box to route the `setup` signal, because if needed, it chooses a box whose only function is to handle errors.

2.1.6 Usages

The components of the DFC architecture are boxes, signals, calls, the router and the global data. A usage can be seen as a dynamically assembled graph composed of boxes and calls. Usages can be branching structures and involve multiple Callees and Callers.

The DFC architecture works as following. There are three phases to the interaction between Caller and Callee: setup, communication, and teardown. In the *setup phase*, as shown in Figure 2.2, each internal call is set up 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` signal, and finally, a channel is created between the two box ports at either end of the call. The router determines the next box in the sequence based on the Caller and Callee's subscriptions and precedence information. When a box receives a `setup` signal from the router, it sends an `upack` signal to the calling box along the channel connecting the two boxes. The setup phase continues with the second box sending a `setup` signal to the router to be forwarded to the next box in the usage. Once a usage from a Caller to a Callee has been set up, the call proceeds to its communication phase, and later to its teardown phase.

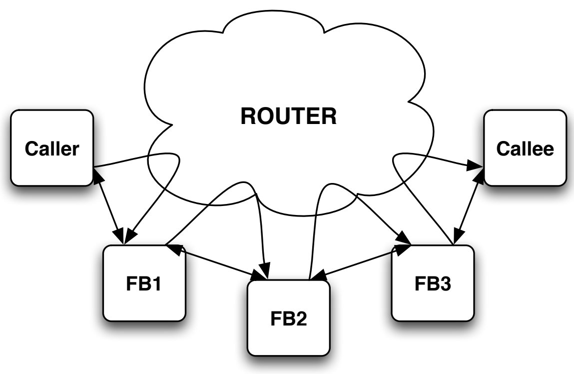

Figure 2.2: Overview of Setup Phase

In the *communication phase*, data is exchanged between the Caller and Callee directly.

In the *teardown phase*, the usage is destroyed. Similar to the setup phase, the teardown phase is performed in a piecewise manner: 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.

2.2 Spin

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). The Spin model checker checks descriptions written in the modelling language Promela. First, we briefly describe Promela and then provide an overview of Spin's verification capabilities. References for our descriptions are the Spin model checker reference manual [9] and the references online [10].

2.2.1 Promela

A model in Promela is a set of processes that communicate via channels. The execution of processes is interleaved. If more than one process can execute, only one of them will be non-deterministically chosen to execute.

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 starting 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.

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.

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`.

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, which sends the process to a labeled statement, or a `break` statement, which jumps to the end of the innermost `do` loop.

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, respectively. 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, *i.e.,* 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 to only 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 executed only 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. In a `do` or `if` statement, these can be used as guards.

A *rendezvous* channel is a channel 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 of 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.

2.2.2 Simulation and Model Checking

SPIN can simulate and model check PROMELA models. SPIN provides a command-line interface as well as a graphical user interface called XSPIN, which can illustrate a trace of the model's execution generated in simulation or as a counterexample from model checking. The trace is presented as a message sequence chart showing the relative ordering of message communication between processes.

In the *simulation mode*, by default, the non-deterministic choices for which statements execute, and the choices for which process 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.

In the *model checking mode*, SPIN can check different types of correctness claims:

- *Absence of Deadlock*: Check for non-existence of invalid end states (non-final states from which there are no next states).
- *State properties*: Claims about the reachability of states.
- *Path properties*: Define valid sequences of events on message channels.

State properties are checked with `never` claims, which are used to specify system behaviour that should never occur. The most simple kind of state property is a system *invariant* that should hold in every system state. The logic used for `never` claims in SPIN is LTL. LTL lets us specify the behaviour of a reactive system, where the executions are sequences of states. The semantics of LTL is defined over infinite executions. LTL is built up from proposition variables, the usual logic connectives $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (if), the quantifiers $\forall$ (forall), $\exists$ (exists), and temporal operators. The temporal operators supported in SPIN and their meaning are described in Table 2.1.

We briefly discuss the theory behind `never` claim verification. The basis is the theory of ω-automata, where the acceptance conditions cover infinite executions [19]. The kind of acceptance condition that SPIN considers was introduced by J.R. Büchi, therefore called Büchi acceptance. It is common to refer to automata with Büchi acceptance conditions simply is *Büchi Automata*. Every PROMELA process is a Büchi automata, which communicates with other processes via message channels. The global behaviour of the system is the asynchronous interleaving product of the PROMELA processes. To perform verification, SPIN takes the negation of the LTL property to be checked, converts

Operator	**Symbol in Spin**	**Meaning**
$\bigcirc$	`X`	Next
$\Diamond$	`<>`	Eventually (future)
$\Box$	`[]`	Always (henceforth)
$\bigcup$	`U`	Strong Until

Table 2.1: Linear Temporal Logic Operators in SPIN

it into a Büchi automaton [18] and computes the synchronous product of the claim and the automaton representing the global behaviour of the system, as illustrated in Figure 2.3. The model checking problem is reduced to checking if the intersection of the languages of a property automaton and a system automaton is empty. Property automata have *accepting* states, which are final states that are visited infinitely often. SPIN's verification procedure, based on a depth-first graph traversal method, terminates when an acceptance cycle is found (counterexample) or when the complete intersection of the product has been computed. An acceptance cycle means that an accepting state was reached from the initial state, and is reachable infinitely often.

An example of the verification procedure is the following. The property to verify is $\Diamond\neg a$. To specify a property `prop1` that should hold, one needs to translate the negation of this property into a `never` claim, using " `spin -f '!(prop1)'` ". SPIN converts the negated formula into the Büchi automata corresponding to the LTL formula, $\Box a$ in our example, represented by the following `never` claim:

```
never {    /* !(<> ! a) */
accept_init:
        if
        :: ((a)) -> goto accept_init
        fi;
}
```

Figure 2.4 shows a model in which the property holds, while Figure 2.5 presents a model in which the property does not hold. The set of atomic propositions of the model is $\{a, b\}$. We think of this system as having the states $\mathbb{P}\{a, b\} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\} = \{o_\emptyset, o_A, o_B, o_{AB}\}$. These system states are the alphabet of the automaton[1]. Sequences of system states (labeling the transitions in Figures 2.4 and 2.5) are the *language* of the

[1]For this explanation, we distinguish between "system states" and "states of the automaton".

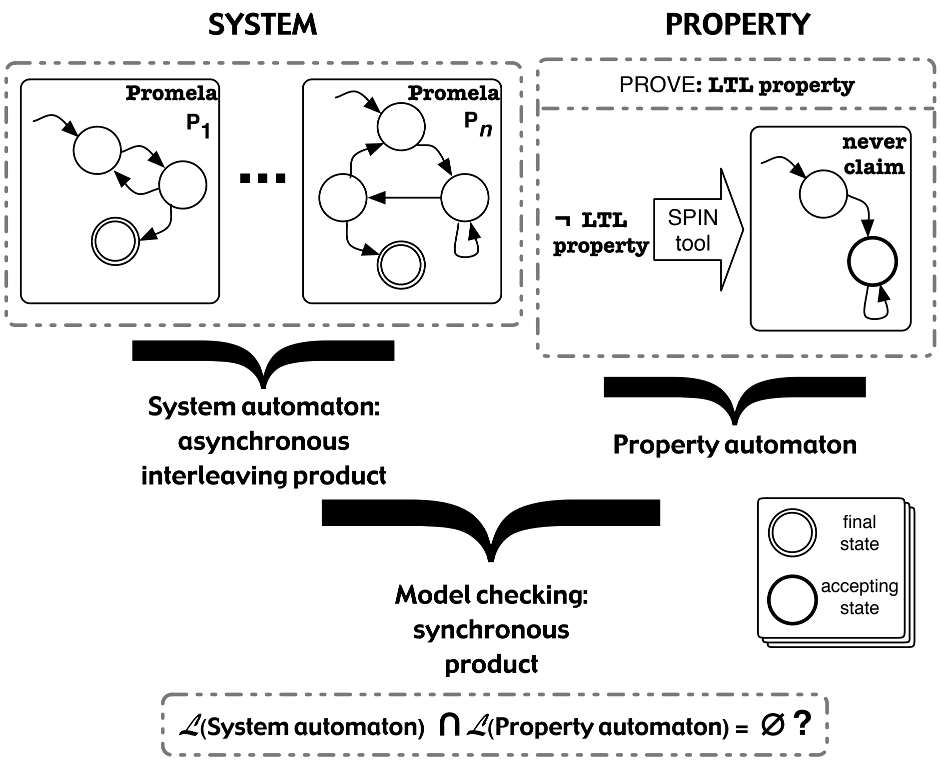

Figure 2.3: Model Checking Problem in SPIN

automaton. For example, the system automaton for Figure 2.4 recognizes the language $O_A(O_B)^\omega$, *i.e.,* an a followed by zero or more b's. The property has only one state, which is an accepting state. Thus, when taking the synchronous product between the property and the system, all states become accepting states. When the property holds, there are no cycles that contain an accepting state, so the intersection of the languages is *empty*, *i.e.,* a cannot repeat infinitely often. However, when the property does not hold, a cycle that contains an accepting state is found, so the intersection of the languages is $(O_A)^\omega$ in Figure 2.5, *i.e.,* a repeats infinitely often, which was the condition that the `never` claim was checking should not occur.

Path properties are checked with `trace` assertions, which express correctness requirements about sequences of operations that processes can perform on message channels. The `trace` assertion defines an automaton that monitors the system execution, specifying the order in which a sending or receiving operation must be performed. If an operation cannot be matched by any transition of the `trace` automaton, the verifier reports an error. The following is an example of a `trace` assertion [9]:

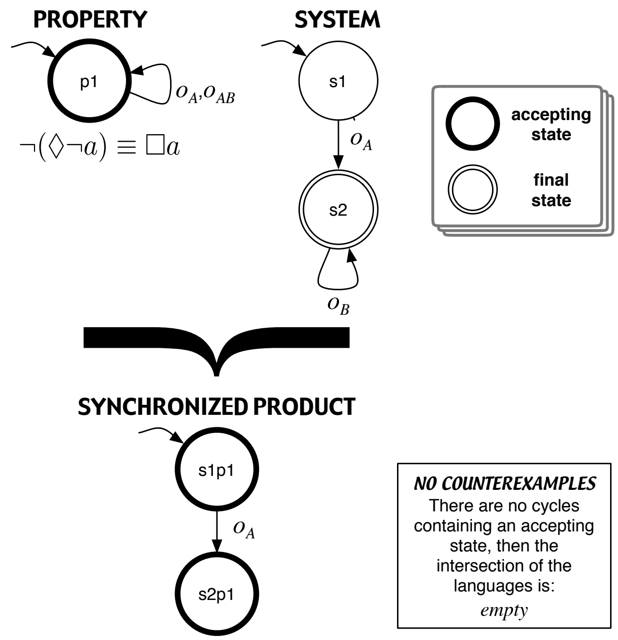

Figure 2.4: Model where property $\Diamond\neg a$ holds

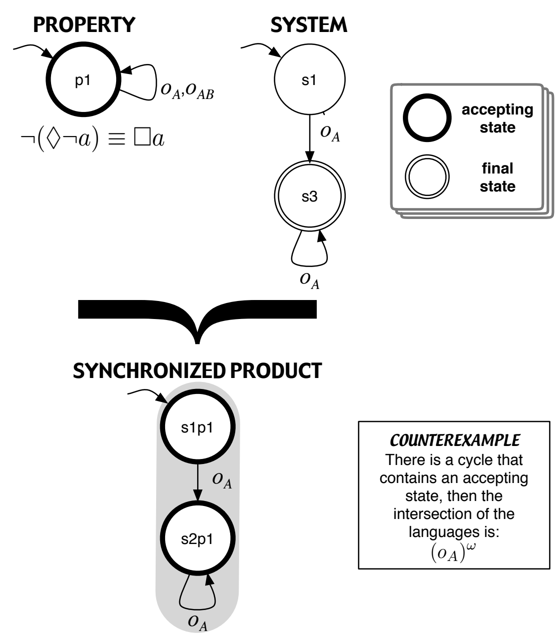

Figure 2.5: Model where property $\Diamond\neg a$ fails

```
trace {
    do
    :: q1!a; q2?b
    od }
```

The example specifies: first, that send operations on channel `q1` alternate with receive operations on channel `q2`, and second, that send messages on channel `q1` must be of type `a`, and that receive messages on channel `q2` must be of type `b`.

To use `trace` assertions, channels must be simple variables and no access to channel array indexes is allowed, which limits their use. Also, there is no clear relationship between the semantics of `trace` assertions and LTL properties, which is a drawback since we wish to declare the correct behaviour DFC using LTL properties. Therefore, we did not use trace assertions in our verification effort.

2.3 Hol

The HOL system is an interactive theorem prover for higher order logic, which allows quantification over predicates and functions. HOL offers not only a rich initial environment of pre-defined tools and pre-proved theorems, but also the possibility of implementing application-specific tools and theories. The tool uses the eager functional meta-language ML for constructing proofs.

A formal proof is a sequence, whose elements are either axioms or follow from previous elements of the sequence by a rule of inference. A theorem is the last element of a proof. The key idea of the HOL system, due to Robin Milner, is that theorems are represented by an abstract type `thm` whose only pre-defined values are axioms, and whose only allowed operations are rules of inference. Therefore, the only way to construct theorems in HOL is to apply rules of inference to axioms or existing theorems, which means that the consistency of the logic is always preserved. HOL has efficient mechanisms to process *forward* and *backward* proofs plus several automated decision procedures that take care of the tedious details of many proofs.

2.4 Summary

In this chapter we described the Distributed Feature Composition (DFC) architecture. We also explained the main characteristics of the SPIN model checker and the HOL theorem prover, which are the main tools we use to prove our claims.

Chapter 3

Modelling DFC

In this chapter we describe our model of the DFC architecture in PROMELA, the modelling language of the SPIN model checker. The feature boxes that have been modelled are call forwarding (CF), originating call screening (OCS), call waiting (CW) and free transparent feature (FTF). We also modelled the interface boxes as Caller and Callee processes. The complete model can be found in Appendix A, which reflects the final specifications after debugging the models. Early versions of the model were created in collaboration with Wenceslas Godard.

3.1 Overview

DFC follows a pipe-and-filter architectural design, where filters are feature boxes, and pipes are internal calls. Each line interface and feature box is modelled as one PROMELA process. We represent the router in a distributed form, having one router process created for each internal call. From now on, we use the term communication path[1] for a DFC internal call.

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

[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.

services. In a static model, all processes (the Caller, Callee and router) and each channel (two for each communication path and two for to and from the router) exist permanently. In a larger configuration with n communication paths, 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 its corresponding channels. For example, for k users, where one user subscribes to the free feature box originating call screening, we need to have $k - 1$ originating call screening processes – one for each potential Callee 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 additionally ports that don't exist in the feature boxes.

Another alternative is to check only one configuration and Caller-Callee combination in a static model as is done by Calder and Miller [4]. 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.

In a dynamic model, instances of boxes in a usage are *dynamically* created by the router as they are needed. Distributed instances of the router process are also dynamically created by boxes when needed. We use an array of channels to declare a set of channels, dynamically allocated by the router, to be used for the communication paths. Each router process provides to the process an array index to make use of the channel. During usage creation, only one router process is running at a time, so we require only two zero capacity channels for all the routing operations. In the dynamic model, we require only $(2n)+2$ channels for n communication paths, since each communication path is composed of two channels.

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 eventually reached in a path of the execution where process identifiers are not released. We abstract the model to a finite state space, by allowing the Caller to call only once or twice. Results of the verification runs with this restriction are shown in Chapter 5 (page 53).

3.2 Signals and Global Data

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 a communication path via the router.
- *upack* is used to acknowledge a *setup* signal.
- *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* signal.

All features use the above signals, whose syntax is given next. 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* signals 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).

Communication paths (between feature boxes, Callers, and Callees, not those to/from 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 {upack, teardown, downack, avail, unavail}
  chan B = [3] of {upack, teardown, downack, avail, unavail}
}
```

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 channels and each process must be aware of which part is the input and output for each communication path. The sizes chosen for these channels will be explained later.

The model includes a set of communication paths as an array of constant size M. A communication path is dynamically allocated 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 a communication path is created, a counter is incremented and the next value is provided as the communication path identifier to use. This communication path identifier is sent in the last field of the *setup* signal. 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 communication path identifier is set by the constant L such that $2^L > M$.

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. Feature precedence is hard-coded into the router processes in our model (Section 3.3.4).

Call forwarding information – An array that contains the numbers to which users want their incoming calls to be 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. Every box in the usage has processing steps for these stages.

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.

3.3 DFC Usage Phases

3.3.1 Setup Phase

Setup signals from a line interface are sent to the router. The function of the router is to receive *setup* signals 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* signal in our model has five fields in the following order:

1. Type of the signal (always `setup`)
2. Identity of the originating Caller of the usage
3. Dialled number
4. Identity of the destination Callee of the usage.
5. Identifier for the communication path between the boxes.

When a box receives a *setup*, it sets a local variable to the value of the communication path identifier provided in the *setup* signal.

To represent the dialled number, we use the constants `user0, user1,` etc. For example, a dialled number field containing `user0` means that the Caller has dialled the number of `user0`.

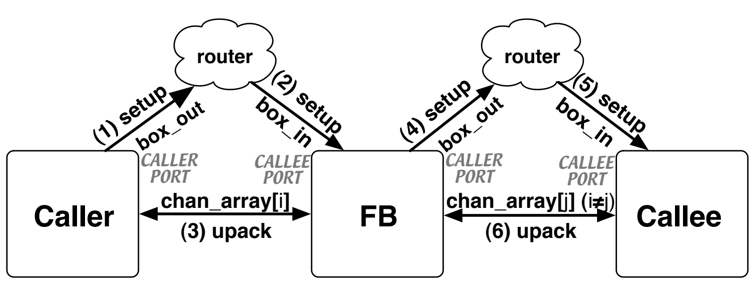

Figure 3.1: Setup Phase

As illustrated in Figure 3.1, the setup phase proceeds piecewise:

- A Caller sends a *setup* to a router (**(1)** in the figure) on channel `box_out`, including the identifier, i, of the communication path to be used.
- 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`, including the identifier, i, of the communication path to be used.
- The receiving box sends an *upack* signal directly back to the Caller (**(3)** in the figure) on the communication path chosen (`chan_array`[i]).
- 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 and have one router for the creation of each communication path between boxes. 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 communications with the routers. This method also reduces the state space of the model.

Spin channels are global variables, so any process can read from them. Persistent processes such as Callee and the bound feature box call waiting always exist and are always trying to read signals from global variables such as the channel `box_in`, even if these signals are not directed to them. To avoid race conditions, where multiple processes want to read `box_in`, for each bound box we introduce a unique channel that the router uses to communicate to a bound box. We need special zero-capacity channels, which we call `box_in_`*process*, to capture the fact that a router is about to send a setup signal to a Callee or call waiting process through `box_in`, and not to another box in the process of usage creation. We continue to also use the `box_in` channel for consistency, so all boxes communicate *setup* signals to and from the routers using channels `box_out` and `box_in`. For a bound feature box, the router first sends the communication path identifier on this `box_in_`*process* channel (which only one box will respond to), then sends a *setup* signal on `box_in`, all within an atomic sequence. In the next chapters, we refer only to `box_in`.

3.3.2 Communication Phase

In the communication phase, data is exchanged between the Caller and Callee directly (*e.g.,* on the communication paths labelled `chan_array` in Figure 3.1). The communication that occurs on the signalling communication path connecting the Caller and Callee may be delayed, *i.e.,* a signal 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 signals 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*.

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 in our model is three: *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 separate channel 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. The signals used to describe the media channels opening or closing (*open*, *close*) are also not included in our model.

3.3.3 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 that is waiting, it will terminate that branch of the usage by sending a *downack* message, but it does not propagate the *teardown* to the subscriber because the subscriber is involved in another call.

[2] An example of where this atomic sequence of actions occurs is in the Callee process in Figure 3.6.

3.3.4 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 the static feature precedence of the source region is free transparent feature (FTF), originating call screening (OCS), call waiting (CW), and of the target region is call forwarding (CF), call waiting (CW), as shown in Figure 3.2.

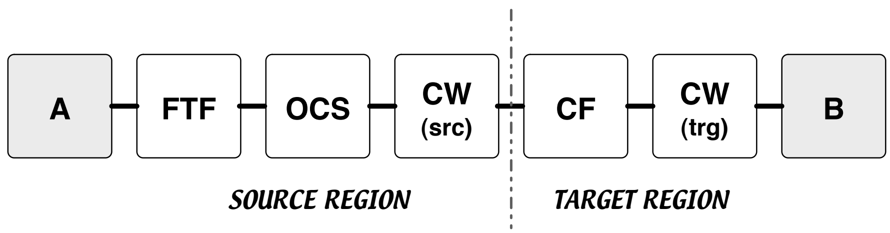

Figure 3.2: Static Feature Precedence

For each kind of feature box, there is a corresponding router that is dynamically created by the feature box before this box sends a setup message to the router process. We abstract away the `route` field of the *setup* signal, so 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) according to the precedence order. For example, in our model, the router for the Caller first checks if the Caller subscribes for free transparent feature, and runs dynamically an instance of an originating call screening process since it is a free feature box:

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

If the user subscribes to a bound feature box, the corresponding information is sent on channels `box_in_`*process* and `box_in`, but no dynamic creation of processes is needed because bound feature boxes are persistent processes created at initialization.

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

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 each router based on a feature's precedence order.

Another modelling issue that we encountered was with regard to how to capture subscription information used by the router. Each router encodes the precedence order of the features, but checks whether a user is subscribed to features using Boolean arrays for each feature (*e.g.,* `subs_OCB[userX]`). 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, which makes it easier to verify different configurations.

3.4 Interface and Feature Boxes

In this section, we describe two interface boxes (a Caller and a Callee), the router processes, and the feature boxes that we have included in our model. The Caller, Callee and call waiting processes persist, whereas the routers and the rest of the feature box processes are created dynamically 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 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[3]. For the communication paths, we omit the channel names (`A` and `B`) and show only the communication path identifier because the action (send or receive) disambiguates which channel would be used. Final states are shown with double circles.

[3]Atomic sequence of statements are used while model checking fixed DFC configurations to reduce the state space, but not when verifying individual feature boxes, during which an atomic sequence is used only to enclose statements that represent the same action.

3.4.1 Caller

Figure 3.3 shows the port and channel labels used by a Caller, and Figure 3.4 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 3.3.4 for more details on the different routers), and sends a *setup* message (as described in Section 3.3.1) to this router along the `box_out` channel.

The Caller then waits for an *upack* message from the neighbour 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 can be initiated.

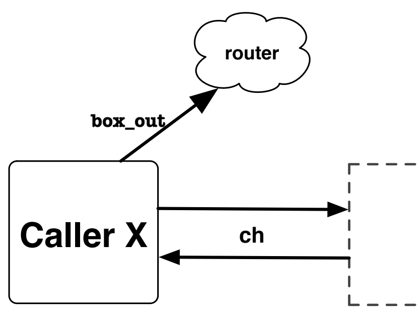

Figure 3.3: Channels and Ports for Caller X

3.4.2 Callee

Figure 3.5 shows the channel and port labels associated with the Callee process, and Figure 3.6 shows the state machine of the Callee process. A Callee is a bound box that can receive more than one *setup* signal, and is a persistent process. Thus, we use a zero-capacity channel `box_in_Callee` (**(1)** in Figure 3.5) to indicate that a router chooses this Callee as the next box in the usage.

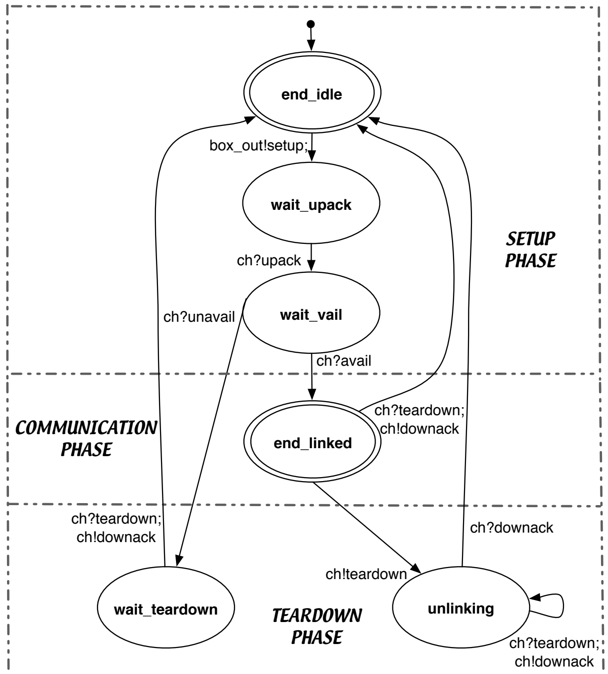

Figure 3.4: Caller Process State Machine

When a Callee process receives the first *setup*, the processing is as follows: When the Callee process is in its end_idle state, the router sends the communication path identifier through `box_in_Callee`, and it is stored in the local variable ch[4]. Next, the Callee process receives a *setup* message from the router through `box_in`, sending the corresponding *upack* upstream on the communication path ch and checking if the user associated with the Callee is currently busy using the value in the boolean array `busy`. If the Callee is not busy, it sends an *avail* message upstream on the ch communication path, and moves into its communication phase (the state end_linked). If the Callee is busy, the sequence *unavail*, *teardown* is sent on the ch communication path, and the Callee process waits for the corresponding *downack*. The communication phase could last arbitrarily long; this possibility is modelled by making the linked state a valid end state (`end_linked`).

Since the Callee process can receive more than one *setup*, "busy processing" is implemented as follows: At any state other than the end_idle, the router may send to the Callee process the communication path identifier of the user trying to reach the Callee using the

[4]A field must be sent on the channel `box_in_Callee`, so the communication path identifier is chosen. When receiving from channel `box_in`, we check with an operator `eval` that this identifier is part of the fields of the *setup* signal, which gives us extra confidence that the *setup* signal is received by the bound process that was meant.

channel `box_in_Callee`. The identifier is stored in an extra local variable called `busy_ch`. Since the user is busy handling another call, the Callee process sends the sequence *upack*, *unavail*, *teardown* on the communication path `busy_ch`, and waits for the corresponding *downack*, so the process go back to its normal processing. Busy processing is shown by the loops labeled $\mathcal{U}$ in Figure 3.6.

As in the Caller process, the Callee returns to its initial state so it is always ready to handle another call.

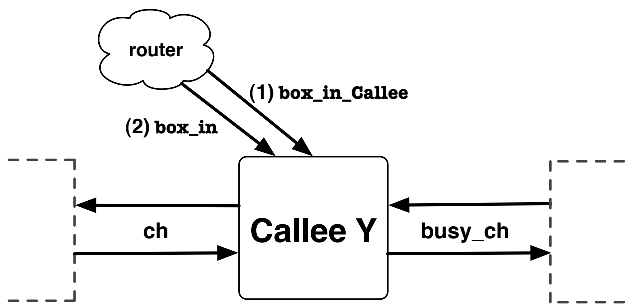

Figure 3.5: Channels and Ports for Callee Y

3.4.3 Free Transparent Feature Box (FTF)

After the setup phase, which performs the proper initialization of the box, a free transparent feature (FTF) box with two ports, connected to communication paths, 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 of a free transparent feature box are shown in Figure 3.7, and its state machine is shown in Figure 3.8. Because FTB is a free feature box, there is no transition that returns it to its initial state.

3.4.4 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 the usage to take a different route. Call

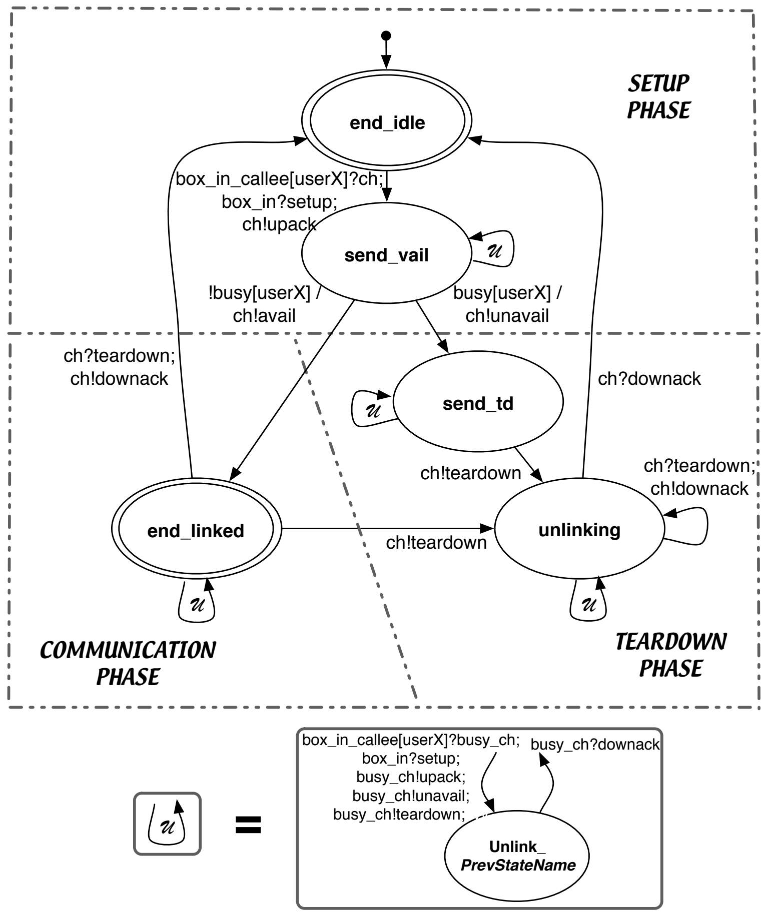

Figure 3.6: Callee Process State Machine

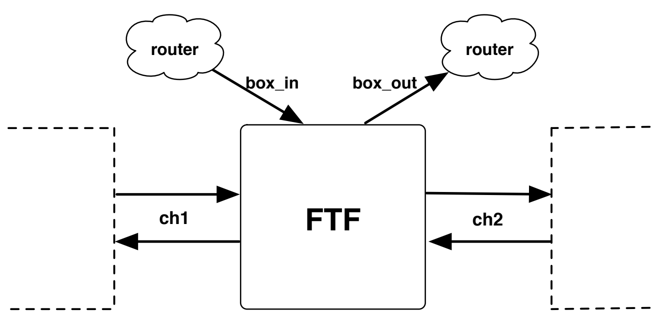

Figure 3.7: Channels for a Free Transparent Feature Box

forwarding is a free feature box. This box communicates with exactly the same channels as the free transparent box (Figure 3.7).

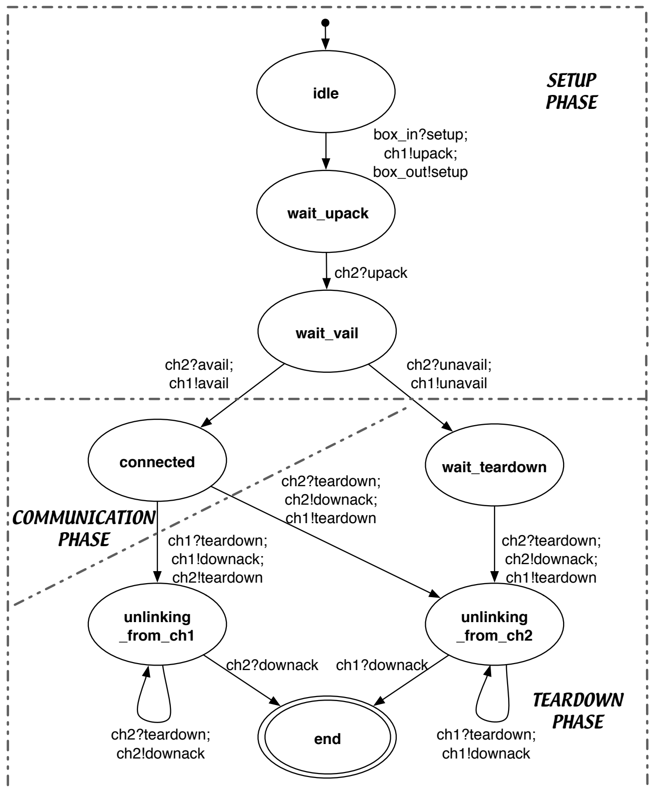

Figure 3.8: Free Transparent Feature Box State Machine

3.4.5 Originating Call Screening (OCS)

Originating call screening is a free feature box, and its state machine is shown in Figure 3.9. 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 3.7).

3.4.6 Call Waiting (CW)

Call waiting is a bound feature box. 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 that the subscriber

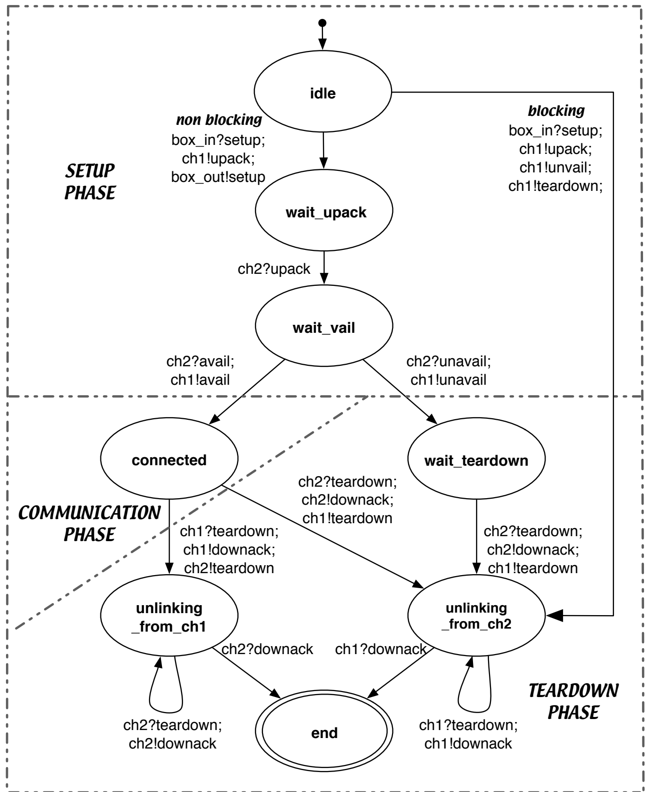

Figure 3.9: Originating Call Screening State Machine

is already engaged in a usage but might switch, as allowed by the call waiting feature. These two additional signals are not implemented in our PROMELA model. We abstract away the details of the Callee issuing a switch signal by using a boolean variable `talk`, which can change non-deterministically whenever two correspondents are connected to the subscriber. The variable `callers` keeps track of the number of Callers connected to the call waiting process. 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 a persistent process, it is ready to read signals at any time, so it uses a zero-capacity channel `box_in_CW` (**(1)** in Figure 3.10) just like Callee.

Figure 3.10, 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 correspondents respectively. Because call waiting is a bound feature, it needs the `busy_ch` communication path, that links the

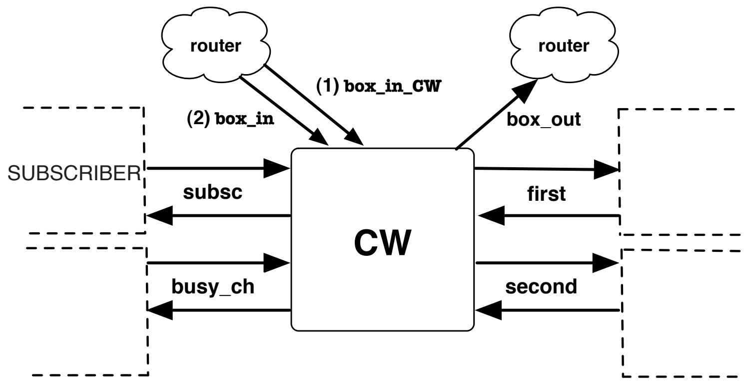

Figure 3.10: Channels for a Call Waiting box

call waiting box with any other correspondant trying to talk to the subscriber. The `busy_ch` communication path is only used for "busy processing", *i.e.,* once the subscriber is already engaged in a call with two parties.

Figure 3.11 shows the state machine for the call waiting box. The SPIN CW process contains 338 states and 445 transitions. SPIN assigns state and transition numbers to all control flow points and statements in the model, thus these numbers differ from the states and transitions shown in Figure 3.11, which combines some transitions for the sake of clarity. The setup, communication and teardown phases are illustrated by shading states with different patterns. The left-hand side manages the source scenario, *i.e.,* the subscriber is calling, so the call waiting box acts as a source feature, which is denoted by src in the state names. The right-hand side manages the target scenario, *i.e.,* the subscriber is being called, so the call waiting box acts as a target feature, which is denoted by trg in the state names. Both scenarios differentiate whether the subscriber is connected to the first or the second correspondent, denoted by first and second in the state names respectively. Up to the con1src_first1 and the con1trg_first1, the behaviour corresponds to the one of a box with two ports. However, call waiting can be connected to two correspondants, and goes to state con2src or con2trg respectively. Then, either of the correspondants can disconnect (leaving the subscriber talking to the other correspondant), or the subscriber can disconnect (leading to the states unlink3src, unlink4src, unlink3trg or unlink4trg for call back processing).

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.

Since the call waiting process is persistent, it can keep receiving *setup* messages, so "busy processing" is implemented as follows: Once the subscriber of call waiting is already communicating with two correspondants, and the router sends to the call waiting process the communication path identifier of the user trying to reach the subscriber, the identifier is stored in a local variable called busy_ch. Then, the call waiting process sends the sequence *upack*, *unavail*, *teardown* on the communication path busy_ch, and waits for the corresponding *downack*, so the call waiting process goes back to its normal processing. Busy processing is shown by the loops labeled $\mathcal{U}$ in Figure 3.11.

3.5 Summary

In this chapter we described our model of DFC in PROMELA. The feature boxes that have been modelled are call forwarding (CF), originating call screening (OCS), call waiting (CW) and free transparent feature (FTF). We also modelled interface boxes as Caller and Callee processes, as well as router processes. We discussed several modelling issues regarding the use of SPIN in the verification effort.

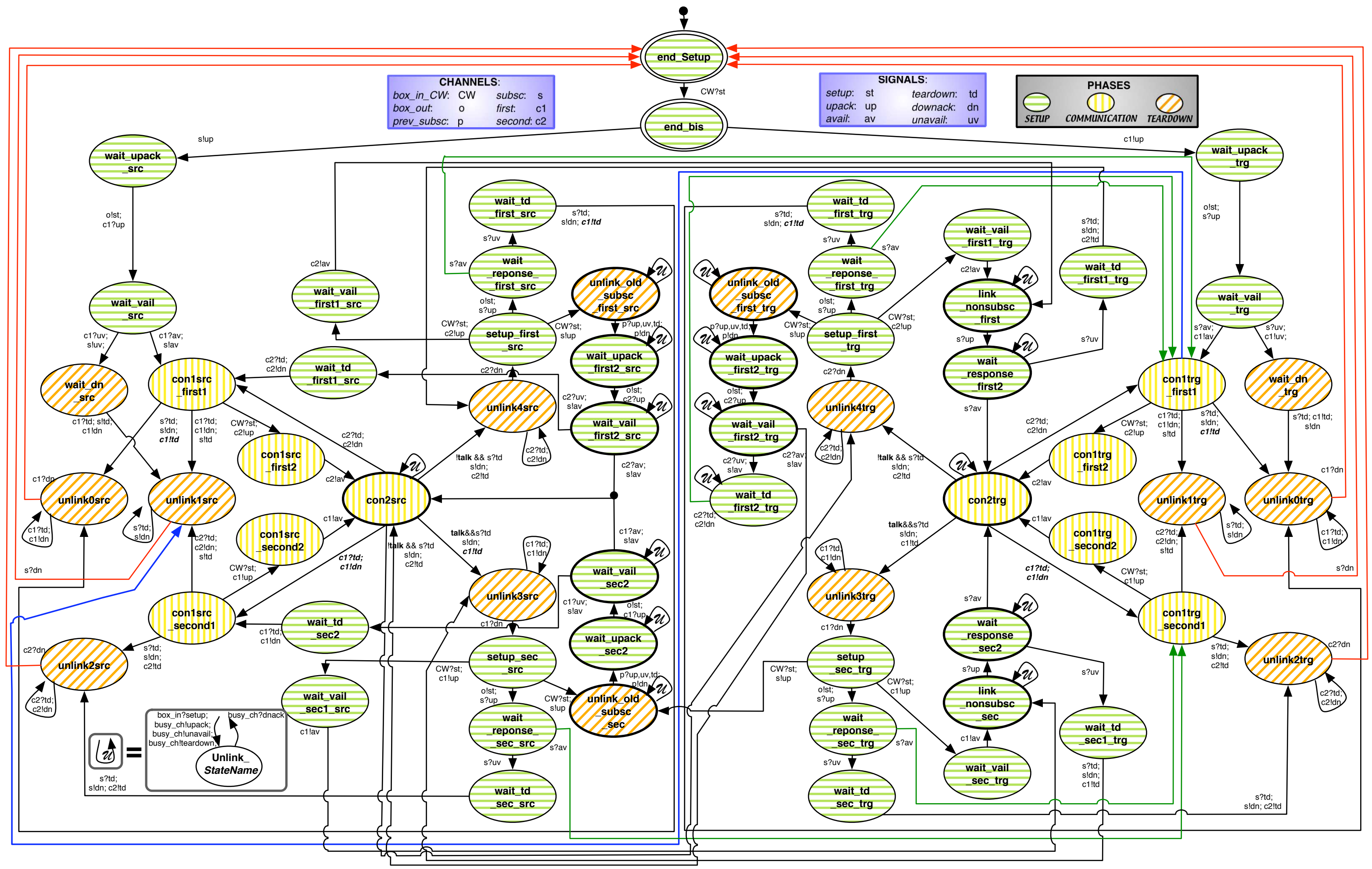

Figure 3.11: Call Waiting State Machine

Chapter 4

Correctness Criteria

To simplify our verification effort, we reason about segments of a DFC usage, based on a categorization of feature boxes by their influence on the routing process. In this chapter, we describe this categorization, and then, we provide a set of LTL properties that define our correctness criteria.

4.1 Categories of Boxes

Feature boxes are independently implemented modules that carry out their functions without external assistance, and they have the power to change the topology of a usage. A feature box can act transparently if the box has no action to perform on the signal received, but it also may have the authority to affect routing, placing, receiving, or tearing down calls. Feature boxes can lead to linear or branching usages, and can also change the number of processes involved in a usage, depending on the signals received. Since the changes in the topologies depend on the features that have the autonomy to influence the routing process, we propose the following characterization of feature boxes:

User agents (UA): Set of interface boxes, plus the feature boxes that can act like a user. A user agent box can request the creation of a usage (*e.g.*, call forwarding on busy), or respond to such request. The response can be positive, accepting the creation of a usage (by the generation of an *avail* signal *e.g.*, voice mail), or negative, rejecting the creation of a usage (by the generation of an *unavail* signal *e.g.*, originating call screening).

***Transparent* (*T*):** Set of all boxes that must forward any call protocol signals that they receive to the next box in the usage, *i.e.,* feature boxes that do not affect the routing. [1]

A further subcategorization of user agents is possible, considering that there are different properties for upstream or downstream user agents. An upstream user agent (*UUA*) is the one requesting the creation of a usage, whereas a downstream user agent (*DUA*) is the one answering or acting on a request.

We denote as a *segment* any part of a usage that starts at a user agent box and ends at a user agent box. More than one *setup* signal is involved in the creation of a branching usage, therefore every *setup* signal generates a segment. The first segment of a branching usage is exactly like the creation of a linear usage. Successive segments would be connected to the previous ones, assembling the usage, as shown in Figure 4.1. The user agents where the segments are connected are called *joins*. The call protocol properties to be presented next are used to verify segments, which may be either linear usages, or segments that compose a branching usage.

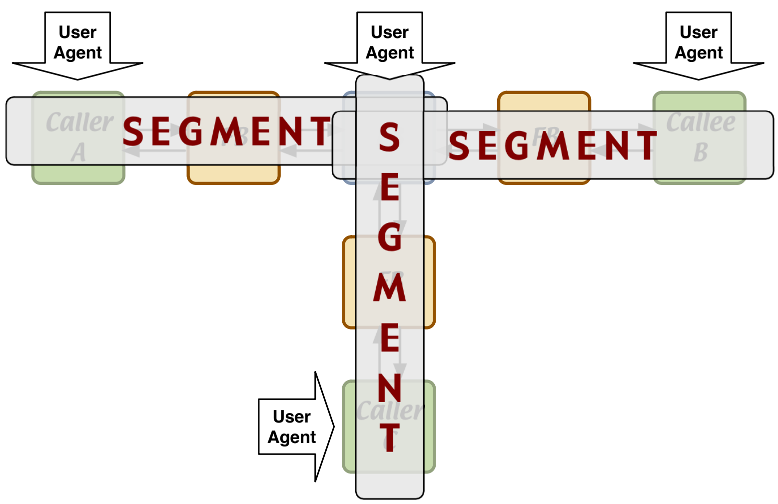

Figure 4.1: Usage composed of Segments

[1] As discussed in the previous chapter, there are specific features called free transparent feature (FTF) and bound transparent feature (BTF) boxes. We use "transparent" as the name for a category of feature boxes, which is inspired by the fact that they act passively (transparently) with respect to the routing process, but this category does not contain only FTF and BTF. The context should make clear which term is being used. We thank Pamela Zave for suggesting this terminology.

4.2 Segment Properties

In this section, we describe the DFC call protocol properties of segments, which define our correctness criteria. The general form of the properties is

$$\Box\,(Send \Rightarrow \Diamond\ Receive).$$

To state the segment properties, we use the names of the channels shown in Figure 4.2. We call the sender of a *setup* signal at one end of the segment an *Upstream User Agent (UUA)*, and the receiving user agent is called a *Downstream User Agent (DUA)*. We also need one feature box (FB i) in the middle of the segment to state the properties.

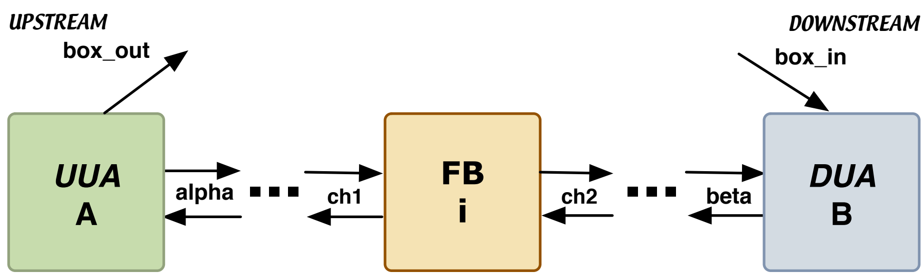

Figure 4.2: Channels in a Segment

The following are the set of segment properties that we initially thought would hold, described in LTL, having one or more feature boxes delimited by a *UUA* and a *DUA*:

***S*.1** A *setup* signal created by *UUA* **A** should eventually reach *DUA* **B**.
$\Box(\texttt{box\_out!setup} \Rightarrow \Diamond\texttt{box\_in?setup})$

***S*.2** A *teardown* signal created by *UUA* **A** should eventually reach *DUA* **B**.
$\Box(\texttt{alpha!teardown} \Rightarrow \Diamond\texttt{beta?teardown})$

***S*.3** A *teardown* signal created by *DUA* **B** should eventually reach *UUA* **A**.
$\Box(\texttt{beta!teardown} \Rightarrow \Diamond\texttt{alpha?teardown})$

***S*.4** An *avail* signal created by *DUA* **B** should eventually reach *UUA* **A**.
$\Box(\texttt{beta!avail} \Rightarrow \Diamond\texttt{alpha?avail})$

***S*.5** An *unavail* signal created by *DUA* **B** should eventually reach *UUA* **A**.
$\Box(\texttt{beta!unavail} \Rightarrow \Diamond\texttt{alpha?unavail})$

In Chapter 5, we describe our effort to verify the segment properties on simple configurations having features boxes delimited by a Caller and a Callee. In the verification process, we found correct behaviours that violate properties ***S*.2** and ***S*.3**.

First, for a configuration of Caller–FTF–Callee, when both Caller **A** and Callee **B** send *teardown* signals to finish the usage, property ***S*.2** fails if the *teardown* signal from Callee **B** is received in FTF before the one from Caller **A**. The *teardown* signal from Caller **A** follows only its first piecewise obligation, since the call to which the signal was supposed to be forwarded is already torn down. The described situation is illustrated with a the message sequence chart (MSC) in Figure 4.3.

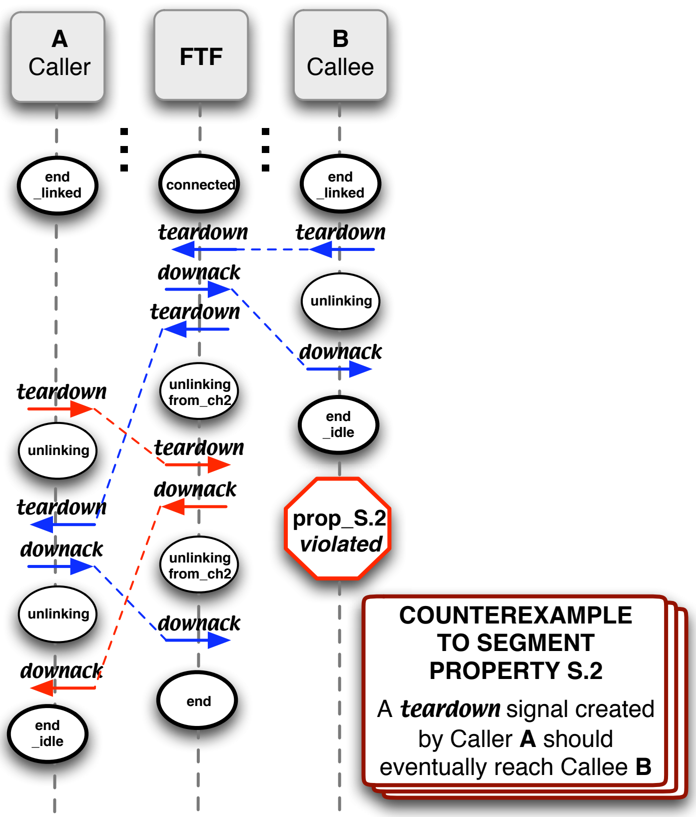

Figure 4.3: Counterexample for Property ***S*.2**

Therefore, we came up with a new formalization for these segment properties, taking into account the consumption of signals by feature boxes:

***S*.2** A *teardown* signal created by *UUA* **A** should eventually reach *DUA* **B** unless *UUA* **A** receives a *teardown* signal first.

$\Box(\mathsf{alpha!teardown} \Rightarrow ((\neg \mathsf{alpha?teardown}) \bigcup (\mathsf{alpha?teardown} \vee \mathsf{beta?teardown})))$

***S*.3** A *teardown* signal created by *DUA* **B** should eventually reach *UUA* **A** unless *DUA* **B** receives a *teardown* signal first.

$\Box(\texttt{beta!teardown} \Rightarrow ((\neg\texttt{beta?teardown}) \bigcup (\texttt{beta?teardown} \vee \texttt{alpha?teardown})))$

However, when a usage has more than two feature boxes, and *teardown* signals are generated by both users, the usage is torn down, but the new ***S*.2** and ***S*.3** properties are violated as illustrated with the MSC in Figure 4.4.

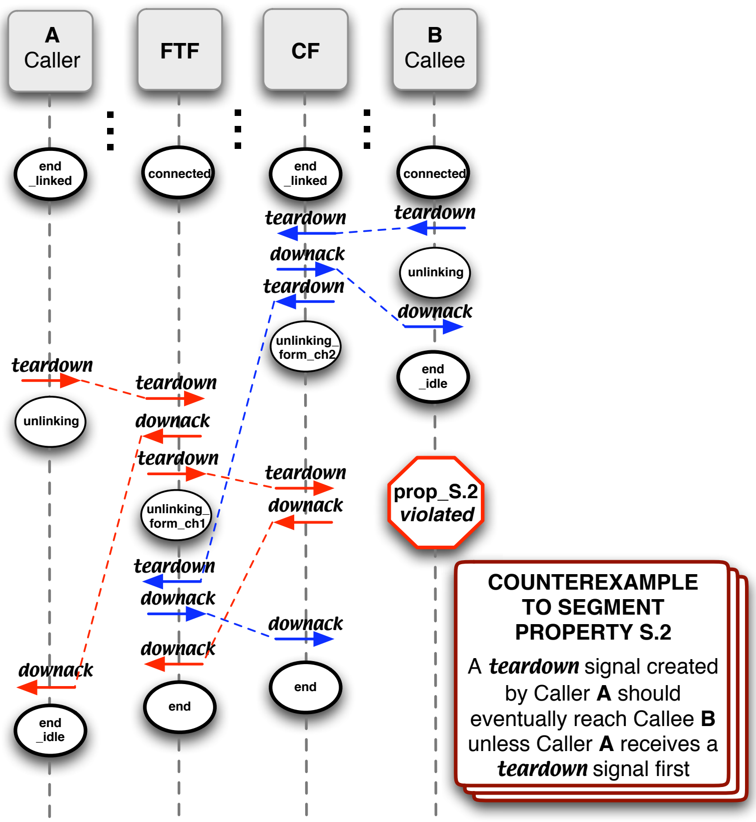

Figure 4.4: Counterexample for Modified Property ***S*.2**

Therefore, we again modified the formalization of the segment properties ***S*.2** and ***S*.3** to express the fact that a *teardown* signal propagates to the end of a segment only if a *teardown* has not been received at some intermediate box. A DFC feature that receives a *teardown* from one side when it has already received a teardown from the other side does not propagate the signal. Otherwise, such messages could cross paths, violating the properties, as shown in the counterexamples. To express this property, we introduce an intermediate box in the segment.

***S*.2** A `teardown` signal created by *UUA* **A** should eventually reach *DUA* **B** if no `teardown` has been received at some intermediate box i.

$\Box(\texttt{box\_out!setup} \Rightarrow$
$\quad((\Box\neg(\exists \mathsf{i}.(\mathsf{i.ch2})?\texttt{teardown})) \Rightarrow$
$\quad\quad(\mathsf{alpha}!\texttt{teardown} \Rightarrow \Diamond(\mathsf{beta}?\texttt{teardown}))))$

***S*.3** A `teardown` signal created by *DUA* **B** should eventually reach *UUA* **A** if no `teardown` has been received at some intermediate box i.

$\Box(\texttt{box\_in?setup} \Rightarrow$
$\quad((\Box\neg(\exists \mathsf{i}.(\mathsf{i.ch1})?\texttt{teardown})) \Rightarrow$
$\quad\quad(\mathsf{beta}!\texttt{teardown} \Rightarrow \Diamond(\mathsf{alpha}?\texttt{teardown}))))$

These properties state that a *teardown* signal generated by an user agent must reach the user agent at the other end of the segment *if* no transparent box in between receives a *teardown* signal in the opposite direction. To set the scope for when the sending of *teardown* signals is relevant, we use `box_out!setup` in the antecedent of properties ***S*.2** and ***S*.3**, which ensures these properties are true for each segment created.

The final set of segment properties, to be checked for segments with any number of feature boxes, is shown in Table 4.1.

$\Box(\texttt{box\_out!setup} \Rightarrow \Diamond\texttt{box\_in?setup})$	***S*.1**
$\Box(\texttt{box\_out!setup} \Rightarrow ((\Box\neg(\exists \mathsf{i}.(\mathsf{i.ch2})?\texttt{teardown})) \Rightarrow (\mathsf{alpha}!\texttt{teardown} \Rightarrow \Diamond(\mathsf{beta}?\texttt{teardown}))))$	***S*.2**
$\Box(\texttt{box\_out!setup} \Rightarrow ((\Box\neg(\exists \mathsf{i}.(\mathsf{i.ch1})?\texttt{teardown})) \Rightarrow (\mathsf{beta}!\texttt{teardown} \Rightarrow \Diamond(\mathsf{alpha}?\texttt{teardown}))))$	***S*.3**
$\Box(\mathsf{beta}!\texttt{avail} \Rightarrow \Diamond\mathsf{alpha}?\texttt{avail})$	***S*.4**
$\Box(\mathsf{beta}!\texttt{unavail} \Rightarrow \Diamond\mathsf{alpha}?\texttt{unavail})$	***S*.5**

Table 4.1: Segment Properties

4.3 Summary

In this chapter, we presented a set of LTL properties describing our correctness criteria, based on a categorization of boxes by their influence on the routing process. In the next chapter, we show how our Promela model can be used to investigate whether our proposed correctness properties hold for DFC usages.

Chapter 5

Model Checking Fixed DFC Configurations

In this chapter, we show how our PROMELA model of DFC, explained in Chapter 3, can be used to check our DFC call protocol correctness properties, described in Chapter 4, for small configurations of DFC. This model checking effort is a useful debugging exercise to see if our proposed properties are correct, but it is not sufficient to conclude that the properties hold for segments with any number of features.

5.1 Model Checking Results

We describe our approach for verifying the segment properties on fixed usages. By "fixed", we mean there is a fixed number of interface and feature boxes that can be part of the usage. In our model, we can set the value of certain global variables and the subscription information of users, to create different fixed usages, and perform verification on them. We can change the maximum number of users, number of calls a Caller can make, and the subscription information.

Because `never` claims in SPIN cannot refer directly to channels in an array such as as `chan_array`, we follow the approach of Holzmann [9], 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 channels and communication paths referred to in this chapter correspond to the ones shown in Figure 4.2 (page 43). The boolean variables `last_rec_setup` and `last_sent_setup` represent the event of the *setup* signal traveling on the channels box_out and box_in respectively. We also use boolean variables last_rec_*signal*_alpha and last_sent_*signal*_alpha, to capture the behaviour of every possible *signal* traveling through the communication path alpha, as well as the boolean variables last_rec_*signal*_beta and last_sent_*signal*_beta, to capture the behaviour of every possible *signal* traveling through the communication path beta. To denote the event of a signal sent or received on a communication path, we change the value of the corresponding variable to true. The event and the change of the boolean variable are enclosed in an atomic sequence since they represent the same action. All the variables are initialized by default to false. For example, when a *teardown* signal is received from the incoming communication path of the call alpha, the variable last_rec_teardown_alpha is updated to true to denote the event:

```
atomic{
    chan_array[alpha].B?teardown;
    last_rec_teardown_alpha=true;
}
```

When sending a signal, the global variable is updated after the signal is transmitted. For example, when a *avail* signal is sent through the outgoing communication path beta, the variable last_sent_avail_beta is updated to true:

```
atomic{
    chan_array[beta].B!avail;
    last_sent_avail_beta=true;
}
```

To capture the meaning of an event, we should set the boolean variable to false immediately after it has been set to true. Placing the update of a boolean variable to true and the reset of this variable to false inside an atomic sequence does not work because the never claim does not register the change of the boolean variable value. If we reset the value to false immediately after the atomic above, we encounter a problem with false negatives due to the interleaving of processes. An example of such a false negative is illustrated in Figure 5.1. The variable last_sent_avail_beta remains true longer then it was expected. The never claim finds an accepting cycle since the boolean variable was not reset to false right after the event, so it keeps checking as if the boolean variable became true more than once. As described earlier (Section 2.2.2), we cannot use trace assertions, so our solution for using model checking to debug fixed configurations is to not reset the boolean variables to false. Using this approach, we are checking only that the segment property passes the first time the antecedent of the property becomes `true` rather than globally. Our compositional reasoning approach, described in Chapter 6, does not encounter this problem

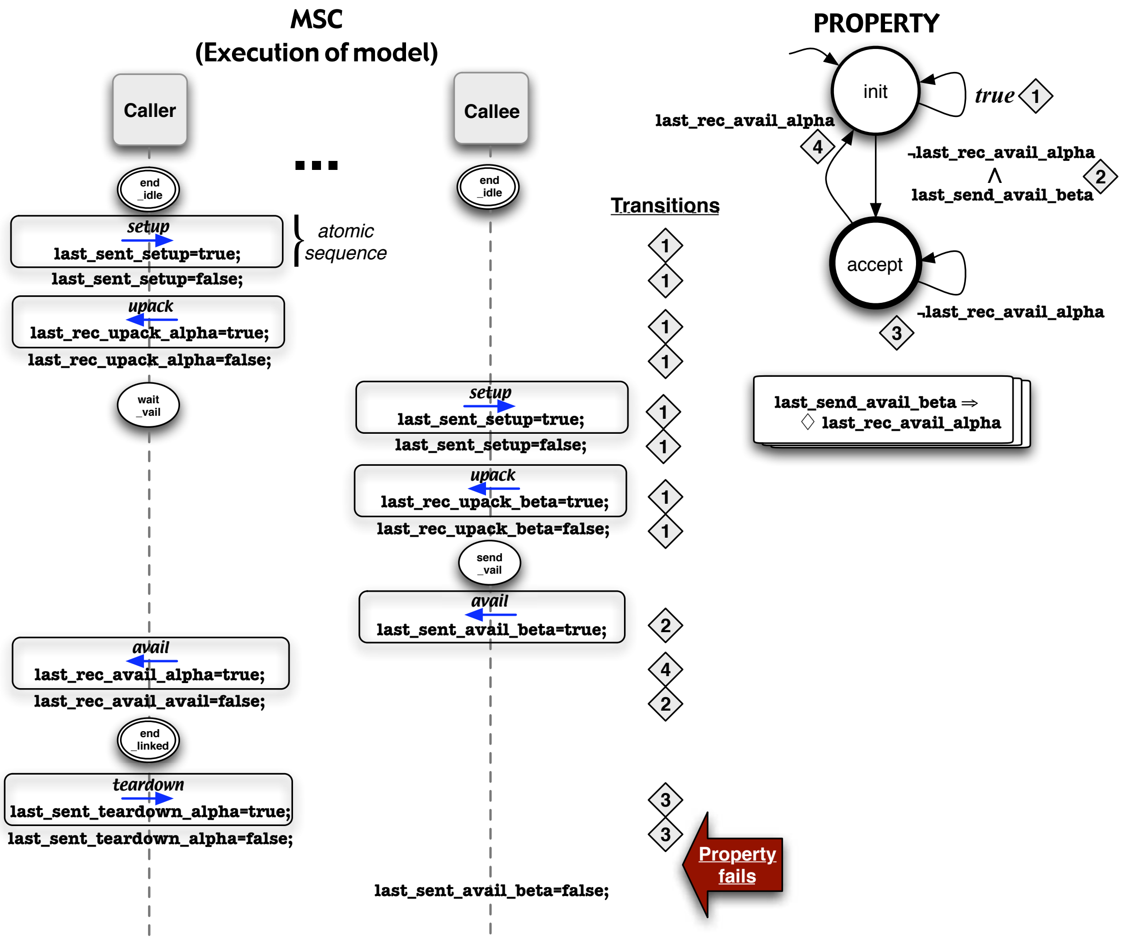

Figure 5.1: False Negative Situation Caused by Interleaving

To check properties ***S*.2** and ***S*.3**, where we need to refer to boxes within the segment, we add to every transparent box boolean variables `rec_teardown_ch2_`*Tbox* and `rec_teardown_ch1_`*Tbox*, where *Tbox* is the name of any box in the transparent category, to represent the event of a *teardown* signal traveling on the box's channels `ch1` and `ch2` respectively. These properties capture the behaviour that a *teardown* signal generated by a user agent must reach the user agent at the other end of the segment *if* no transparent box in between receives a *teardown* signal in the opposite direction. Using these boolean variables, we can formalize

$$\Box(\neg\exists \mathsf{i}.(\mathsf{i.ch1})?\mathtt{teardown})$$

by instantiating the existential quantifier for all possible intermediate boxes in a segment. To avoid confusion with multiple instances of the same box, we check only this property on configurations with one instance of each kind transparent box.

The model checking verification runs were done on a 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.

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 5.1. To reduce the state space, the model of the Caller is restricted to make only one or two calls, avoiding an infinite state system.

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 the name of a feature, *e.g.,* TFB, OCS, or CW(src), it means that in the usage checked, all the users subscribe to this feature box in their source regions. If the subscription column shows the name of a feature, *e.g.,* CF, or CW(trg), it means that in the configurations checked, all the users subscribe to this feature box in their target regions. For the call waiting box, model checking space limits meant we only checked a simple configuration shown in Figure 5.2.

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. In the call waiting box model based on a BoxTalk specification [30], we found previously unknown race conditions within five minutes while checking for deadlock in the configuration of Figure 5.2. 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

Number of Callers	Number of Callees	Times Caller can call	Subscription information	Number of states	State vector (bytes)	Total Depth reached	Memory used (Mbytes)	Time min:sec
1	6	1	TFB	194	552	161	1.716	0:00.020
1	6	2	TFB	1964	568	241	1.921	0:00.095
1	6	1	TFB,OCS	647	568	190	1.716	0:00.049
1	6	2	TFB,OCS	30684	600	304	4.993	0:02.286
1	6	1	TFB,CF	578	568	191	1.716	0:00.044
1	6	2	TFB,CF	23649	600	306	4.685	0:01.233
1	6	1	TFB,OCS,CF	1815	584	221	1.921	0:00.131
1	6	2	TFB,OCS,CF	258456	632	373	28.647	0:23.450
2	1	1	CW	>13799	>532	>50000000	>1802.482	>28:00.366

Table 5.1: Execution Statistics for Segment Property Checking

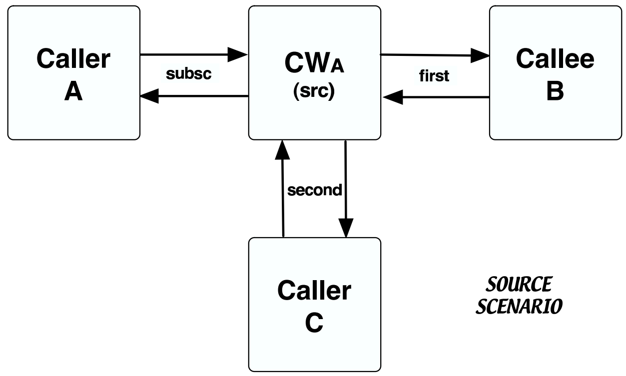

Figure 5.2: Call Waiting Feature Box in a Simple Configuration

is shown with a MSC in Figure 5.3, where the subscriber and the person on hold get connected. Two race condition situations can happen during the time that the CW box is attempting to reach the subscriber in the call back processing: the subscriber may try to make another call, or another user may try to call the subscriber. In either cases, the subscriber seems unavailable to CW, and the call back processing fails. After conferring with Pamela Zave [27], we handled the situations of the call back processing failure as follows. First, when the subscriber tries to make another call, the person on hold is left in the same condition. The subscriber is allowed to reach the user being called, but the subscriber is reminded that there is a person on hold as soon as possible, as illustrated with a MSC in Figure 5.4. Second, when another user tries to call the subscriber, the user trying to reach the subscriber is left on hold. The subscriber is reached by the person that is currently on hold, completing the call back processing, as illustrated with a MSC in Figure 5.5. This behaviour was added to the original specification of call waiting, and the final description is shown in Figure 3.11 (page 40).

Checking fixed segments (i.e., fixed subscription information) does not allow us to conclude that the properties hold of DFC segments with any number of features. In Chapter 6, we explore a compositional reasoning approach whereby we can model check box properties on individual boxes, and then use theorem proving to conclude that segment properties hold in all DFC segments, composed of any number of transparent boxes and delimited by user agents.

5.2 Summary

In this chapter, we explained how to use the DFC Promela model, described in Chapter 3, to check our call protocol correctness properties, described in Chapter 4, on small fixed DFC usages. We provided results from the model checking verification runs, and explained some issues regarding the verification effort. Our models and proposed properties were debugged, but this exercise is not sufficient to conclude that the properties hold for segments with any number of features. Therefore, we propose a compositional reasoning method in the next chapter.

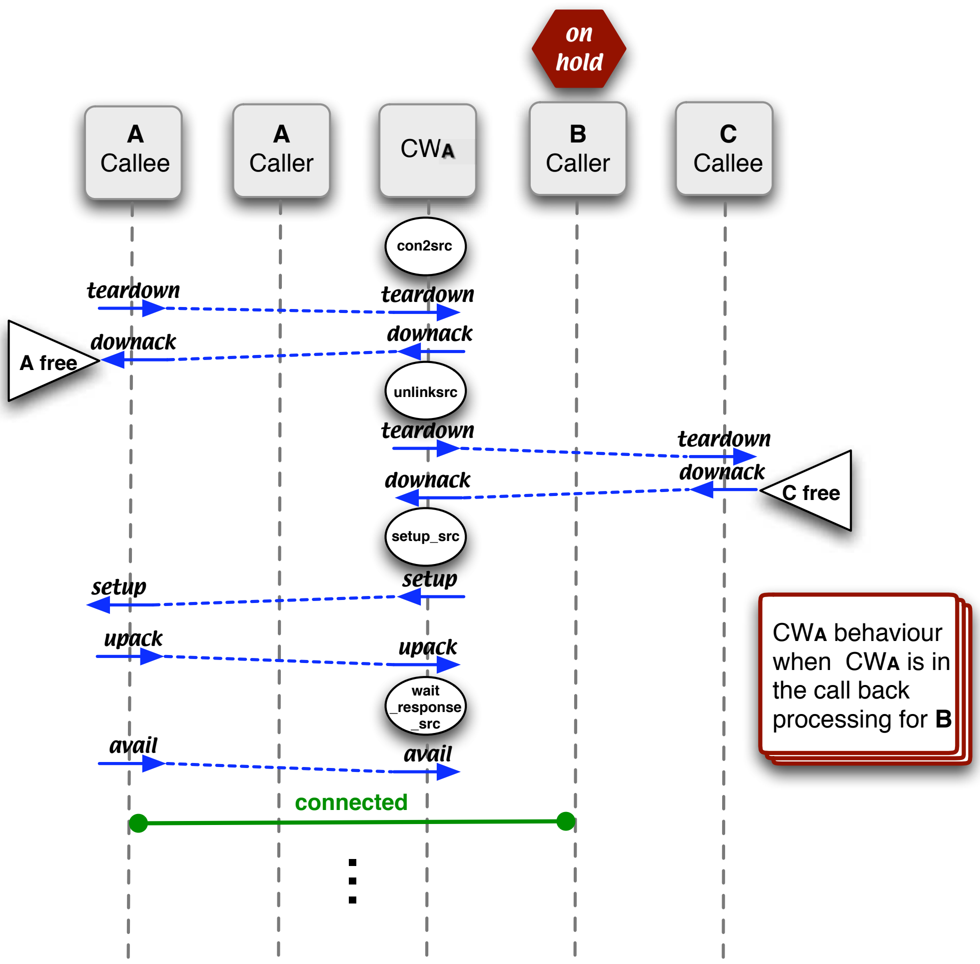

Figure 5.3: Normal Situation in CW Call Back Processing

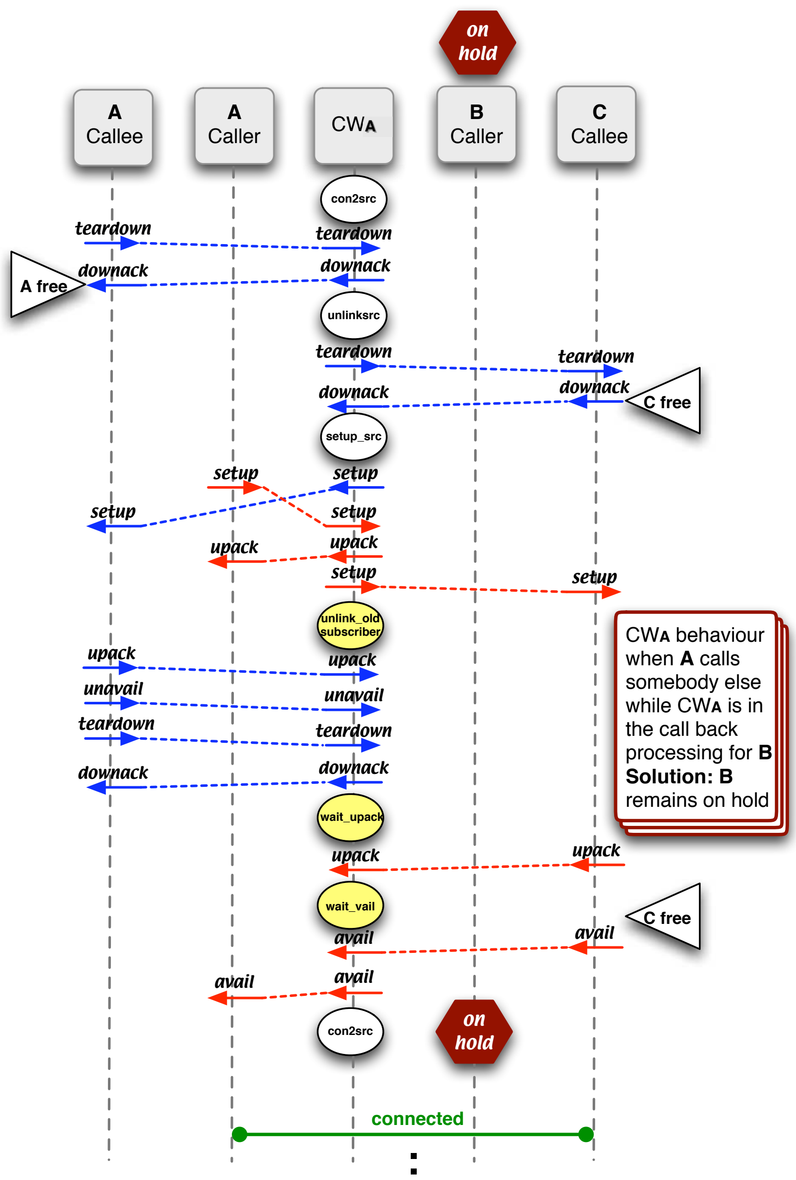

Figure 5.4: Race Condition in CW Call Back Processing for Subscriber

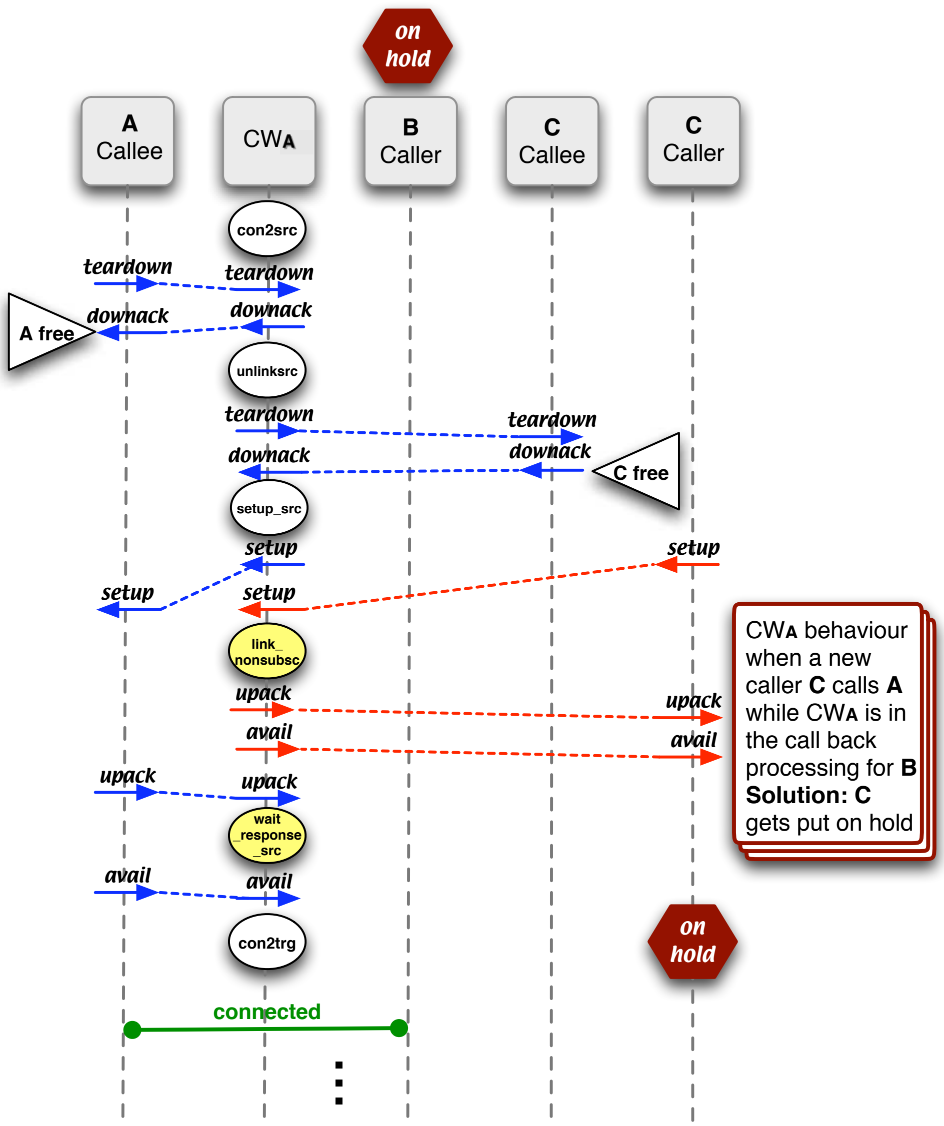

Figure 5.5: Race Condition in CW Call Back Processing for Non-Subscriber

Chapter 6

Compositional Reasoning

In this chapter, we describe our compositional reasoning method to verify that segments with any number of boxes satisfy our correctness properties.

6.1 Overview

Our main goal is to verify that all segments with any number of boxes satisfy our segment properties, which define our correctness criteria. In Chapter 5, we described an approach to check our proposed segment properties on DFC usages with a fixed number of Callers, Callees, and feature boxes. However, while that verification effort is sufficient for verifying specific instances of DFC usages, we cannot conclude from it that the properties will hold for segments of any number of boxes. Therefore, we propose a *compositional reasoning method*, that combines theorem proving and model checking, to prove the segment properties hold in all segments with any number of boxes.

Figure 6.1 shows the decomposition of the top goal into subproblems for a theorem prover and a model checker. To prove the main goal, we reduce the problem to the subproblems of: checking LTL properties of boxes individually (2A), describing the behaviour of the unbounded queues as LTL properties (2B), and then using both sets of LTL properties in an inductive proof that concludes the proposed segment properties. Each subproblem is represented as a component in the second level of the proof tree. The reasoning to use subgoals (2A) and (2B) to conclude the main goal is performed in terms of LTL properties *only*. In the third level, we verify properties for individual boxes by performing model checking on *models* of the boxes. Next, we describe each part of the proof tree presented in Figure 6.1.

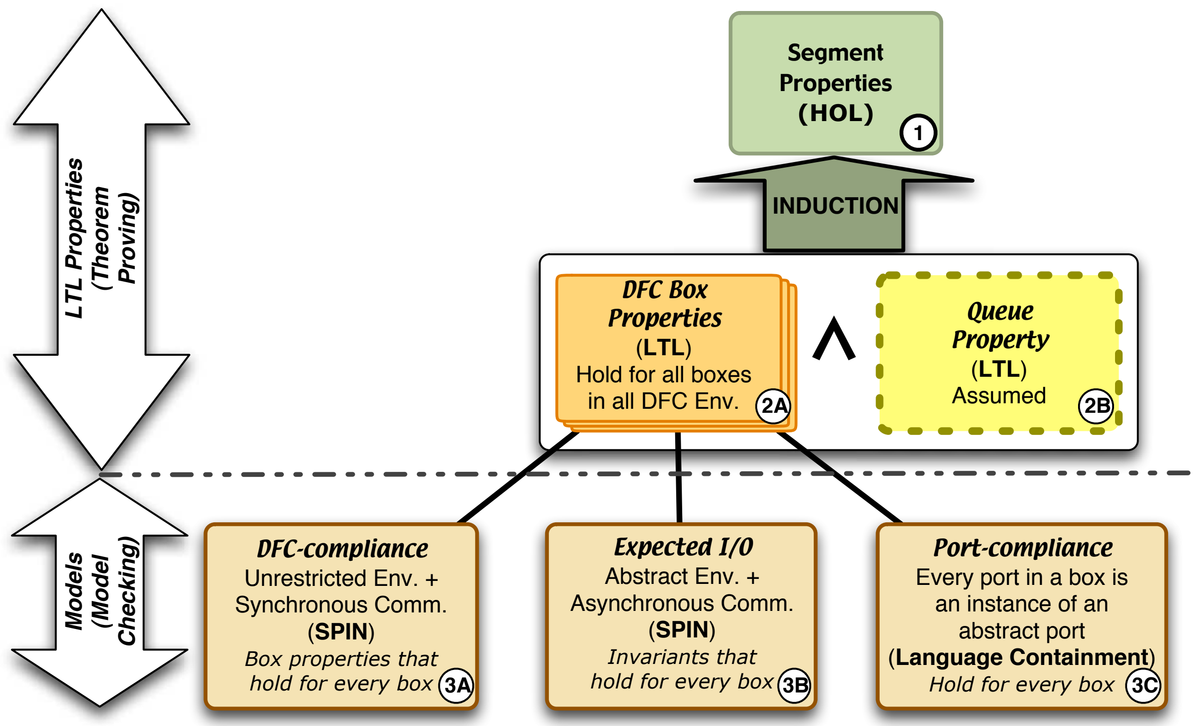

Figure 6.1: Compositional Reasoning Method

We verify the **DFC Box Properties** (2A) to prove that *individual boxes behave as intended when communicating with any DFC environment*. The verification of individual boxes is performed for *Transparent* (*T*), *Upstream User Agent* (*UUA*), and *Downstream User Agent* (*DUA*) box categories, which each have different properties. The verification of boxes in each category follows the methodology described in the third level of the proof tree. The box properties are described in LTL and most have the form "After *receiving* a signal, the box eventually *sends* a signal on another channel".

$$\Box\,(\mathsf{Receive}(\mathtt{signal}) \Rightarrow \Diamond\,\mathsf{Send}(\mathtt{signal}))$$

To represent the behaviour of the unbounded queues connecting boxes in any DFC segment, we assume perfect communication and describe the **Queue Property** (2B) in LTL. The queue property has the common form "After the box *sent* a signal, eventually it is ready to be *received* by the next box[1]".

$$\Box\,(\mathtt{Send}(\mathtt{signal}) \Rightarrow \Diamond\,\mathtt{Receive}(\mathtt{signal}))$$

[1]This does not mean signal is guaranteed to be read. The box properties check for this condition.

To prove that *the segment properties hold for any segment of length n*, we use the box properties (2A) and queue property (2B) to perform induction over the number of boxes communicating through queues (1). Segments are composed of transparent boxes and delimited by *UUA* and *DUA* boxes. The *base case* of the inductive reasoning corresponds to segments composed only of a *UUA* and a *DUA*. In the *inductive step*, we conclude that a segment, composed of a *UUA*, n T boxes, and a *DUA*, satisfies the call protocol properties.

Verifying the DFC Box properties (2A) is decomposed into three proof obligations (3A, 3B and 3C), which must be completed for all individual boxes. In the verification of **DFC-compliance** (3A), the goal is to prove that *each box reacts to the call signals in agreement with the DFC architecture constraints in an unrestricted environment that can send and receive any call protocol signal at any time*. We verify these properties in Spin. For the communication between the unrestricted environment and the individual box, we employ synchronous (rendezvouz) communication, *i.e.,* the channel capacity is zero and it cannot store messages. Therefore, when a send operation is executed, the corresponding receive operation must be executed next. This abstraction captures the communication behaviour of an individual box and its neighbours for any size of queues, and reduces the state space because no queues are modeled. However, synchronous composition implicitly assumes that the box receives only inputs and sends outputs when they are expected.

Verifying that *each box receives only the call signals it expects and sends outputs when they are expected by its neighbours* is done by checking the **Expected I/O Property** (3B) expressed as an invariant in Spin. We annotate the models of boxes with error states, which are reached when a box receives a signal that it was not expecting. The invariant properties have the form $\neg$ ($\Diamond$`Box@error`), thus the model checking reasoning reduces to verifying that the error state is never reached. In contrast with case (3A), in (3B) we use asynchronous (delayed) communication, *i.e.,* the channel can store signals that do not need to be read immediately, which allows interleaving. Rather than combining the box with all possible boxes that could be its neighbours, we create an abstract model of DFC port behaviour that captures the most general behaviour of a DFC port. All boxes are verified for the expected I/O property in an environment consisting of abstract ports. Without this abstraction we would have to verify the invariants on all possible permutations of box models, *i.e.,* every box having a caller port (callee port respectively) must be composed with all the boxes having a callee port (caller port respectively). The verification of the expected I/O property would involve every pairwise composition of caller–callee and callee–caller ports.

As long as the behaviour of every port in any box is contained within the behaviours of the abstract ports, then the use of abstract ports in the environment is a valid abstraction, and there is no need to compose the models of boxes pairwise to verify the expected I/O property. To prove that *every port in any box is an instance of the most general abstract port, i.e.,* **Port-compliance** (3C), we carry out language containment reasoning.

Having completed the proof obligations (3A), (3B) and (3C), we can conclude that each box will satisfy the DFC Box properties (2A) in any DFC environment when communicating with any other box, thus we can proceed to reason using only the LTL properties.

The verification that (2A) $\wedge$ (2B) $\Rightarrow$ (1) needs to be completed only once. Thereafter, using this compositional reasoning approach, to add a new feature to the DFC architecture, and ensure that all segments of length n satisfy the segment properties, there are the following proof obligations for the new feature:

- Verify, using model checking, that an individual box reacts to the call signals in agreement with the DFC architecture constraints, with an unrestricted environment and synchronous communication.

- Verify, using model checking, that an individual box receives and sends only the call signals it expects, with an abstract environment that capture the most general DFC port protocol behaviour, and asynchronous communication.

- Verify that the language recognized by a port in any box is contained in the language recognized by an abstract port.

In the rest of this chapter, we present the box properties for T, UUA and DUA boxes, followed by the model checking approach to verify DFC-compliance on SPIN models of the DFC interface and feature boxes. Then, we explain how to check the expected I/O property on the SPIN models. Next, we describe the proof that every port in a box is an instance of an abstract port. Finally, we explain how to use theorem proving with HOL to perform inductive reasoning, using the box properties and assumptions about the queues to conclude our correctness properties.

6.2 DFC Box Properties

To prove segment properties using a compositional reasoning argument, we first need to verify properties of individual boxes, categorized as T, UUA and DUA boxes depending on the box's influence on the routing process.

The box properties are expressed as LTL formulas, and most have the form "After *receiving* a signal, the box eventually *sends* a signal on another channel".

$$\Box\,(\mathsf{Receive}(\texttt{signal}) \Rightarrow \Diamond\, \mathsf{Send}(\texttt{signal}))$$

The box properties are described in terms of the channels and communication paths shown in Figure 6.2.

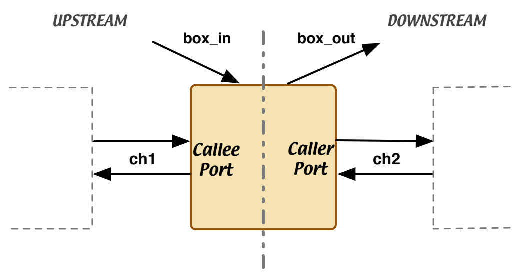

Figure 6.2: Channels and Ports for Box Properties

The channels `box_in` and `box_out` connect the box with the routers sending and receiving *setup* signals respectively. The communication path `ch1` (composed of two one-way communication channels) connects the box's callee-port with a caller port of an adjacent box. A Callee user box has only one callee port, but feature boxes can have either none, one, or more than one callee port. The communication path `ch2` (composed of two one-way communication channels) connects the box's caller port with a callee port of an adjacent box. A Caller user box has only one caller port, but the feature boxes can have either none, one, or more than one caller port.

Next, we list the feature properties organized by category. The DFC Box Properties, formalized in LTL, are listed in Table 6.1 and described in terms of the channels and communication paths shown in Figure 6.2. Most box properties are similar to the segment properties (Table 4.1), but reflect what an individual box must do so that the segment properties will be satisfied. Also because bound boxes participate in more than one segment, these properties must hold for all time. In the following subsections, we explain these properties.

$\Box(\texttt{box\_in?setup} \Rightarrow \Diamond\texttt{ch1!upack})$	***T*.1**
$\Box(\texttt{box\_in?setup} \Rightarrow \Diamond\texttt{box\_out!setup})$	***T*.2**
$\Box(\texttt{ch1?teardown} \Rightarrow \Diamond\texttt{ch1!downack})$	***T*.3**
$\Box(\texttt{ch2?teardown} \Rightarrow \Diamond\texttt{ch2!downack})$	***T*.4**
$\Box(\texttt{box\_in?setup} \Rightarrow ((\neg\texttt{ch2?teardown} \bigcup (\texttt{ch1?teardown} \wedge \Diamond\texttt{ch2!teardown})) \vee (\neg\texttt{ch1?teardown} \bigcup (\texttt{ch2?teardown} \wedge \Diamond\texttt{ch1!teardown})) \vee \Box(\neg\texttt{ch1?teardown} \wedge \neg\texttt{ch2?teardown}))$	***T*.5**
$\Box(\texttt{ch2?avail} \Rightarrow \Diamond\texttt{ch1!avail})$	***T*.6**
$\Box(\texttt{ch2?unavail} \Rightarrow \Diamond\texttt{ch1!unavail})$	***T*.7**
$\Box(\texttt{ch2?teardown} \Rightarrow \Diamond\texttt{ch2!downack})$	***UUA*.1**
$\Box(\texttt{box\_in?setup} \Rightarrow \Diamond\texttt{ch1!upack})$	***DUA*.1**
$\Box(\texttt{ch1?teardown} \Rightarrow \Diamond\texttt{ch1!downack})$	***DUA*.2**
$\Box(\mathsf{ch1!upack} \Rightarrow ((\neg\mathsf{ch1?teardown}) \bigcup (\mathsf{ch1?teardown} \vee \mathsf{ch1!avail} \vee \mathsf{ch1!unavail}))$	***DUA*.3**
$\Box(\mathsf{ch1!upack} \Rightarrow \Diamond(\mathsf{ch1!avail} \vee \mathsf{ch1!unavail}))$	***DUA*.4**
$\Box(\mathsf{ch1!unavail} \Rightarrow ((\neg\mathsf{ch1?teardown}) \bigcup (\mathsf{ch1?teardown} \vee \mathsf{ch1!teardown}))$	***DUA*.5**
$\Box(\mathsf{ch1!unavail} \Rightarrow \Diamond(\mathsf{ch1!teardown}))$	***DUA*.6**

Table 6.1: DFC Box Properties

6.2.1 Transparent Box Properties

For every signal received by a T box, there is a corresponding action taken. We give a brief explanation of the events occurring in the channels connected to the T box.

Both obligations of the *piecewise setup* are followed by these boxes. After receiving a *setup* signal on the port connected to the channel `box_in`, two actions must be performed. First, the box should send the corresponding *upack* signal upstream on the port connected to the communication path `ch1`, as stated in Property ***T*.1**. Second, the box should send a modified[2] *setup* signal downstream on the port connected to the channel `box_out`, as

[2]DFC makes a distinction between a new and a modified *setup* signal, since the former corresponds

stated in Property ***T*.2**.

Both obligations of the *piecewise teardown* are followed by these boxes. However, there is a distinction if the *teardown* signal is generated *upstream* or *downstream*, since the events take place on different channels. First, if the *teardown* signal is generated *upstream*, *i.e.,* on communication path ch1, the corresponding *downack* signal should be sent upstream, *i.e.,* after receiving a *teardown* signal on the port connected to the communication path ch1, the box should send the corresponding *downack* signal on the port connected to the communication path ch1, as stated in Property ***T*.3**. Property ***T*.4** is symmetric to Property ***T*.3** for the case where the *teardown* signal is generated *downstream*, so the corresponding *downack* signal must be send on the port connected to the communication path ch2. Second, since a T box can receive *teardown* signals from either neighbour, only the first one is forwarded. In the formalization of the Property ***T*.5**, after the box receives a *setup*, three situations could occur: (1) if the *teardown* signal received on the port connected to the communication path ch1 is the first to arrive at the box, then the *teardown* signal is forwarded onto the communication path ch2; (2) if the *teardown* signal received on the port connected to the communication path ch2 is the first to arrive at the box, then the *teardown* signal is forwarded onto the communication path ch1; or (3) no *teardown* signal is received on either port.

Finally, a T box propagates *avail* and *unavail* signals upstream. After receiving an *avail* or an *unavail* signal on the port connected to the communication path ch2, it should be sent upstream on the port connected to the communication path ch1 to its adjacent box, as stated in Property ***T*.6** and Property ***T*.7** for *avail* and *unavail* signals respectively.

6.2.2 Upstream User Agent Box Properties

For *UUA* boxes, we give a brief explanation of the events occurring in the channels connected to the *UUA* box. There are no properties of *piecewise setup*, since these boxes are sending a new *setup* signal to request the creation of a usage. In addition, there are no required actions on the reception of an *avail* or *unavail* signals, which may be absorbed by the *UUA* box.

There is an action required on the reception of a *teardown* signal, but only the first obligation of the *piecewise teardown* must be satisfied: If the *teardown* signal is gener-

to the request of the creation of a usage, whereas the later is only a modification of the route field. However, our model abstracts away the route field of the *setup* signal.

ated downstream, the corresponding *downack* signal should be sent downstream, *i.e.,* after receiving a *teardown* signal on the port connected to the communication path `ch2`, the box should send the corresponding *downack* signal on the port connected to the communication path `ch2`, as stated in Property ***UUA*.1**. However, the *UUA* box may not propagate the *teardown* signal received, since there are no more boxes of the segment currently being torn down connected to the *UUA* box.

6.2.3 Downstream User Agent Box Properties

For *DUA* boxes, we give a brief explanation of the events occurring in the channels connected to the *DUA* box. For *piecewise setup*, only the first obligation must be followed: After receiving a *setup* signal on the port connected to the channel `box_in`, the box should send the corresponding *upack* signal upstream on the port connected to the communication path `ch1`, as stated in Property ***DUA*.1**. However, *DUA* boxes do not need to propagate the *setup* signal since they are answering the request for the creation of a usage.

On the reception of a *teardown* signal, only the first obligation of the *piecewise teardown* must be satisfied: If the *teardown* signal is generated upstream, the corresponding *downack* signal should be sent upstream, *i.e.,* after receiving a *teardown* signal on the port connected to the communication path `ch1`, the box should send the corresponding *downack* signal on the port connected to the communication path `ch1`, as stated in Property ***DUA*.2**. But the *DUA* box does not need to propagate the *teardown* signal received, since there are no more boxes of the segment currently being torn down connected to the *DUA* box.

For *avail* and *unavail* signals, we verify properties of the form "After *sending* a signal, the box eventually *sends* a subsequent signal on the same channel".

$$\Box\ (\mathtt{Send(signal)} \Rightarrow \Diamond\ \mathtt{Send(signal)})$$

First, after sending an *upack* signal on the port connected to the communication path `ch1`, the box should send an *avail* or an *unavail* signal upstream on the port connected to the communication path `ch1`, if the box has not received a *teardown* signal from upstream yet, as stated in Property ***DUA*.3**. For our DFC model, a stronger version of this property can be stated because the box does not read input from the port connected to the communication path `ch1` prior to sending the *avail* or *unavail* after the *upack* signal on the port connected to the communication path `ch1`, as stated in Property ***DUA*.4**.

Second, after sending an *unavail* signal on the port connected to the communication path `ch1`, the box sends a *teardown* signal upstream on the port connected to the communication path `ch1`, if the box has not received a *teardown* signal from upstream yet, as stated in Property ***DUA.5***. For our DFC models, a stronger version of this property can be stated because the box does not read input from the port connected to the communication path `ch1` prior to sending the *teardown* after the *unavail* signal on the port connected to the communication path `ch1`, as stated in Property ***DUA.6***.

6.3 DFC Compliance

To verify that individual boxes behave as intended, we prove that each box reacts to all signals in agreement with the DFC call protocol architecture constraints, described in the previous section as box properties, and formalized in LTL. We check these properties using model checking in Spin. We decided to use `atomic` sequences only when needed, *i.e.,* conditions in a process must be changed avoiding interleaving, or when a set of statements are intended to represent the same action to maximize the possible interleavings. We place each box in an unrestricted environment that can receive and send non-deterministically *any* of the signals through all channels shown in Figure 6.3[3], using synchronous (rendezvous) communication. Next we explain the environments used for the Caller, Callee, Two-way and Three-way feature boxes.

A *Caller* has only a caller port, so communication path `ch2`, and channel `box_out` are used. A *Callee* has two callee ports, so two communication paths `ch1` (for regular communication) and `busy_ch` (for "busy" processing in bound persistent boxes), and the channel `box_in` are used[4]. For *Two-way Feature* boxes (*i.e.,* boxes with two ports), communication paths `ch1` and `ch2` (for regular communication), as well as channels `box_in` and `box_out` are used. For *Three-way Feature* boxes (*i.e.,* boxes with three ports), communication paths `subsc`, `ch1`, `ch2` (for regular communication), and `busy_ch` (for handling the box's response to another user that tries to call, where it simply tears down the call), as well as channels `box_in` and `box_out` are used.

Dynamic allocation of channels is not necessary because only one box is verified at a time. Figure 6.4 shows part of the Spin process for the unrestricted environment, called `env`. The unrestricted environment has no life on its own, *i.e.,* all it provides is

[3]All kinds of boxes are shown in the figure, but only one box is verified at a time.

[4]For presentation purposes, we leave out the details of the `box_in`_*process* channels needed for bound boxes, as described on page 28, but these exist in our model.

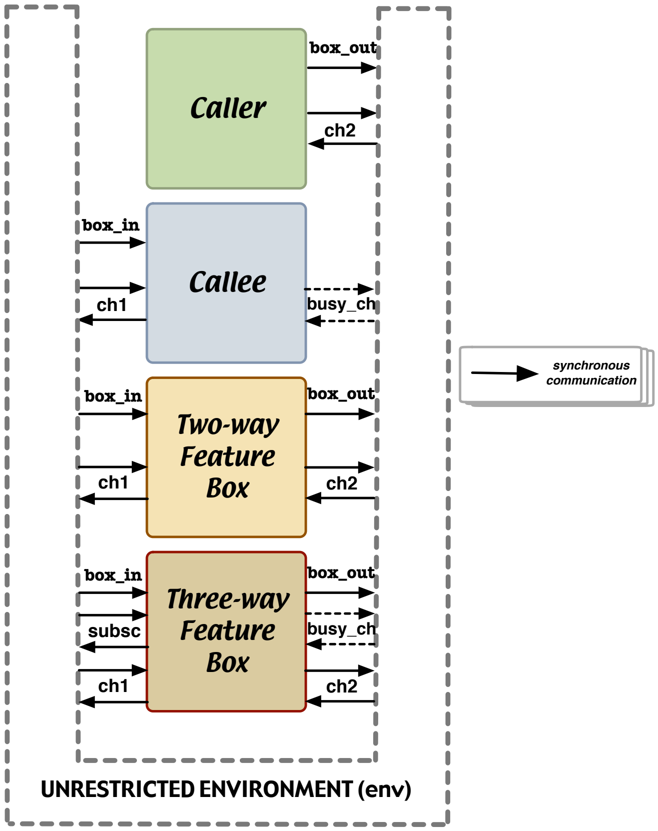

Figure 6.3: Unrestricted Environment to Verify DFC Box Properties

```
proctype env(byte thisuser)
{
 byte orig1;   /*source subsc channel*/
 byte dest1;   /*dest subsc channel*/
 byte orig2;   /*source first channel*/
 byte dest2;   /*dest first channel*/
 byte orig3;   /*source second channel*/
 byte dest3;   /*dest second channel*/
 byte orig4;   /*source busy channel*/
 byte dest4;   /*dest busy channel*/

 end_idle:
  do
  :: box_in!setup,orig2,dest2,dest2,ch1
  :: box_out?setup,orig1,dest1,dest1,eval(ch1)
  :: chan_array[ch1].A?upack
  :: chan_array[ch1].A?avail
  :: chan_array[ch1].A?unavail
  :: chan_array[ch1].A?downack
  :: chan_array[ch1].A?teardown
  :: chan_array[ch1].B!upack
  :: chan_array[ch1].B!avail
  :: chan_array[ch1].B!unavail
  :: chan_array[ch1].B!downack
  :: chan_array[ch1].B!teardown
  /* Same as 14-25 for ch2 */
  /* Same as 14-25 for subsc */
  /* Same as 14-25 for busy_ch */
  od;

 end: skip
}
```

Figure 6.4: SpinProcess for the `env` Unrestricted Environment

the corresponding handshake associated with all signals that could be sent or received by the box being verified because synchronous communication is used. For boxes with more ports, we need only to add the corresponding sending and receiving transitions, as shown in lines 14–25 in Figure 6.4. In our unrestricted environment, we use synchronous communication, *i.e.,* rendezvous channels between the box and the unrestricted environment processes to abstract away the communication queues, and to avoid deadlock. This abstraction captures the communication behaviour of an individual box and its neighbours for any size of queues, and reduces the state space because no queues are modeled. Rendezvous communication is composed of a sending and a receiving part, but for an external observer, they act as one operation. Once a rendezvous send occurs, only the corresponding receiving part becomes executable. The receiving part of a rendezvous operation becomes executable just after the corresponding sending part was selected. If, in the unrestricted environment, SPIN happens to choose a rendezvous send statement that has no receiving part in the box process, the send statement is discarded by SPIN, selecting a new candidate from the set of executable statements.

We add global boolean variables to the individual box models to capture the behaviour of the event of a signal being sent or received, following the approach for model checking segment properties, described in Section 5.1 (page 49). The boolean variables `last_rec_setup` and `last_sent_setup` are set to true when the *setup* signal is traveling on the channels `box_out` and `box_in` respectively[5]. We use boolean variables `last_rec_`*signal* to `last_sent_`*signal*, to capture the behaviour of every possible *signal* traveling through the communication paths. To denote the event of a signal sent or received on a communication path, we change the value of the corresponding variable to `true`. The event and the change of the boolean variable are enclosed in an atomic sequence, since they represent the same action. All the variables are initialized by default to `false`, and reset (set again to `false`) after a transition is taken that reads or writes to the calls. We do not have the problem of false negatives, described in Chapter 5 because we use synchronous communication with the unrestricted environment. For example, when a *teardown* signal is received on the incoming channel of communication path `ch2`, the variable `last_rec_teardown` is updated to `true` to denote the event, and reset to `false` afterwards[6]:

[5]`setup` signals travel only to and from the router.

[6]This statement in PROMELA is `chan_array[ch2].B?downack;`, however, to make it easier to follow, we do not state explicitly the use of the array of channels `chan_array` or the channel of the communication path that is used to send or receive (`A` or `B`).

```
atomic{
    ch2?teardown;
    last_rec_teardown=true;
}
last_rec_teardown=false;
```

When sending a signal, the global variable is updated before the signal is transmitted, because control of the execution changes from the sender to the receiver with rendezvous channels. For example, when a *downack* signal is sent through the outgoing channel of communication path `ch1`, the variable `last_sent_downack` is updated to `true`, and reset to `false` afterwards:

```
atomic{
    last_sent_downack=true;
    ch1!downack;
}
last_sent_downack=false;
```

We verified the DFC Box properties on the *T*, *UUA*, and *DUA* boxes that we described in our Spin model, and the results obtained from the verifications are shown in Table 6.2. We present the maximum execution statistics for the largest verification of a property in terms of number of states, memory used and time, instead of listing all the box properties statistics per box. The properties that took the longest to verify were the ones related to *teardown* signals.

Box verified	**Number of states**	**State vector (bytes)**	**Depth reached**	**Total memory used (Mbytes)**	**Time min:sec**
Caller	153	480	140	1.778	0:00.015
Callee	67	484	122	1.778	0:00.014
TFB	95	480	137	1.778	0:00.015
OCS	128	480	141	1.778	0:00.016
CF	99	488	145	1.778	0:00.016
CW	295645	500	23938	41.548	0:20.331

Table 6.2: Maximum Execution Statistics for DFC Box Properties Verification

6.4 Port Compliance

Using synchronous communication with an unrestricted environment to verify DFC-compliance implicitly assumes that the box receives inputs and sends outputs only when they are expected. Therefore, we also need to verify the expected I/O property when the box is placed in an environment that follows the DFC call protocol. To avoid the problem of having to verify every box in the environment of every other box, we capture the essential behaviour of a DFC port in an abstract model of a port's behaviour.

The DFC manual presents models of caller port and callee port behaviour [13]. We started using these caller and callee port models for our abstract environment, but found that the ports of the call waiting feature box can switch from being callers to callees and vice versa during the box's execution. This behaviour happens in two situations. The first situation occurs when a user who was *called* by the subscriber (interacting with CW through a caller port) is placed on hold. If this user decides to hang up, it releases the CW's caller port. The port just released can be used if another user *calls* the subscriber, and therefore interacts with CW through a callee port. The second situation occurs when a user *calls* the subscriber so the subscriber interacts with CW through a callee port, and another user tries to reach the subscriber. This user remains on hold. When the subscriber hangs up, the CW feature calls back the subscriber, reminding them that there is a person on hold. The subscriber is now *called* by CW and therefore interacts with CW through a caller port. Because of these situations, we created an abstract model of a port, called a *combo* port, that can switch between these modes. The details of how we verify all features with neighbours whose ports behave as this most general abstract port are provided in the next section. In this section, we describe how to verify that the behaviour of every port of a box is contained within the possible behaviours of the combo port.

The abstract models of caller port, callee port, combo ports and their free and bound instances can be arranged in a partial order based on language containment as shown in Figure 6.5, where the dashed boxes are the abstract models (1-6). The language of the ports is the communication between the port and the channel. Bound ports (ports of bound boxes) have the behaviours of free ports, but after the call is torn down they return to their initial state to await another *setup* signal. The behaviour of both caller ports and callee ports is contained within the behaviour of the combo ports. The combo bound port model, shown in Figure 6.6, behaves like a caller port (or callee port) until it reaches the communication phase (state `CommPhase`), then there is no distinction in behaviour between caller port and callee port. This captures the behaviour of call waiting where the subscriber can call (using a call waiting callee port) or be called (using a call waiting caller port). Bound ports include the behaviour of the `busy_ch` port, which handles the reception of a *setup* signal from the router when the box is already communicating with another box. The `busy_ch` port rejects the request for an additional connection. This behaviour is captured by the looping transitions on states in Figure 6.6 labelled $\mathcal{U}$. The rest of the abstract models (2 to 6 in Figure 6.5) are shown in Appendix B.

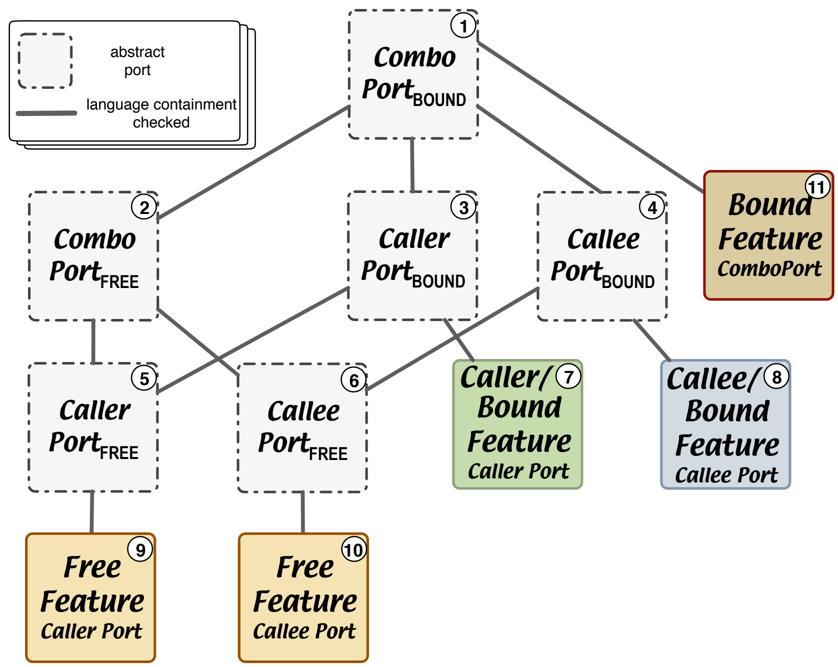

Figure 6.5: Language Containment Relations

Rather than trying to show language containment of a box's port directly with the behaviour of the combo bound port, we rely on the partial order of abstract models, and match the box's ports with the most appropriate element of the abstract model hierarchy. This makes it easier to find the abstraction function needed to show language containment, and also provides a tighter verification of the port's behaviour. In Figure 6.5, the shaded boxes (the leaves of the tree) represent the ports of particular boxes. For example, the caller port of the Caller process (7) is checked against the abstract bound caller port model (3), and the callee port of a free feature box, such as call forwarding, (10) is checked against the free callee port abstract model (6).

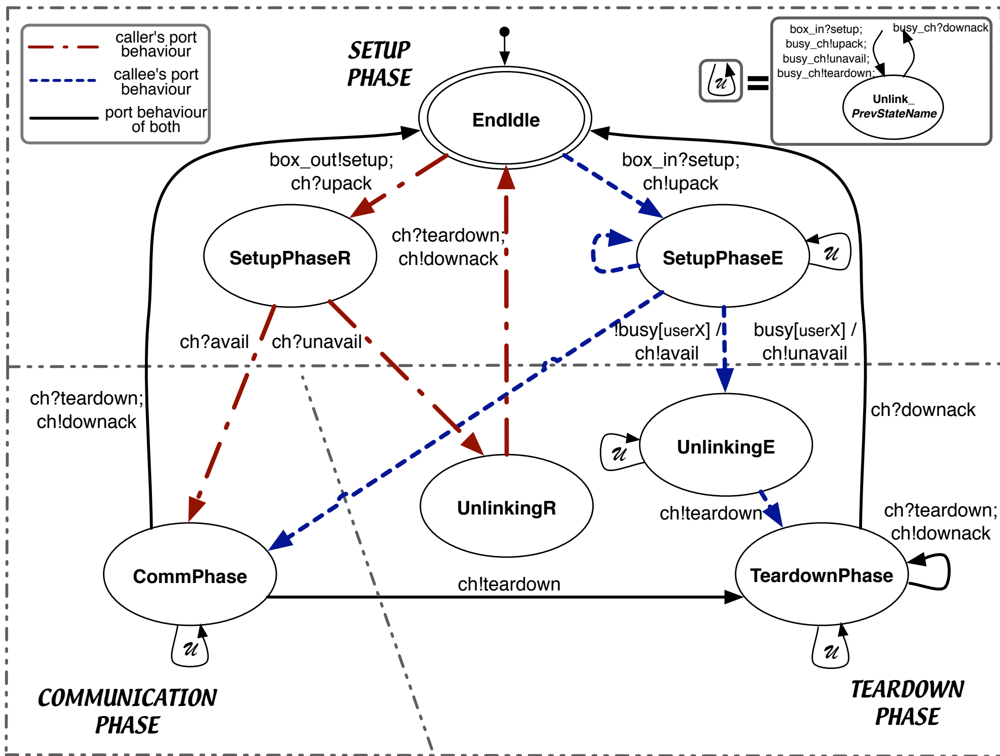

Figure 6.6: State Machine for $\texttt{ComboPort}_{\mathsf{BOUND}}$ Process

We check that the behaviours of a box are contained within the behaviours of an abstract port in two steps. First, we isolate the behaviour of the box to its communication on only one port by replacing all transition triggers based on communication with other ports with a guard of "true" and removing all outputs except those to the port being verified. This is a valid abstraction of the port's behaviour – it does not add or remove any port behaviour. Second, we find an abstraction function, `abs`, matching the states of the box (`concrete`) with the states of an abstract model (`abstract`). Then we show, for every transition in the concrete machine consisting of a source state, trigger, which receives or sends a signal, and destination state:

$$
\begin{aligned}
&\forall\ \mathsf{src}, \mathsf{sig}, \mathsf{dest} \cdot (\mathsf{src}, \mathsf{sig}, \mathsf{dest}) \in \texttt{concrete} \\
&\qquad \Rightarrow (\mathsf{abs}(\mathsf{src}), \mathsf{sig}, \mathsf{abs}(\mathsf{dest})) \in \texttt{abstract}
\end{aligned}
$$

We have written a simple tool in ML that takes a description of the box as a set of transitions, isolates one port's behaviour, and carries out the language containment check stated above. It walks over transitions of a box, uses the abstraction function to compute the abstract states matching the source and destination of the transition, and checks that there is a corresponding transition in the abstract machine. Because we are considering only signal types, and not considering data values, this reasoning can be done using an enumerative search. The creation of the state transition diagram for a box is currently done by hand, but a tool could automatically extract these models from the PROMELA model. Determining the abstraction function is usually straightforward, but can be tedious for a bound box like CW, due to the large number of states and transitions.

We checked the behaviour of the all the boxes we modelled against the appropriate abstract port model. All the ports of call waiting behave as bound combo ports. We also checked the relationship between the abstract models (*e.g.,* that caller port bound is contained within the combo port bound). Using the partial order, we know that as long a box's port behaviour is contained within one of the abstract models, it is contained within the most general abstract model. This part of the verification effort took 5 seconds to check each call waiting port, which is the most complicated and time consuming example.

The abstract port models contain more information than captured by the DFC box properties in LTL (Table 6.1). The LTL properties capture only the essential information about box behaviour needed to prove the segment properties. By isolating this information from the more complicated abstract port descriptions, we simplify the inductive reasoning, which is the final step in our method.

Tables showing the abstraction functions for all the relationships of Figure 6.5 are in Appendix B.

6.5 Expected I/O

We use SPIN to verify the expected I/O property: that each individual box placed in an environment of the most abstract ports (*i.e.,* communicating with combo bound ports) receives only signals it expects and sends only the signals the abstract port models expect. Without abstract port models, we would need to check all possible permutations of box models, *i.e.,* every box having a caller port (callee port respectively) must be composed with all the boxes having a callee port (caller port respectively). The verification of the expected I/O properties would have involved every single pairwise composition of caller–callee and callee–caller ports. The abstract port models for verifying the expected I/O property for individual boxes uses the channels and communication paths shown in Figure 6.7.

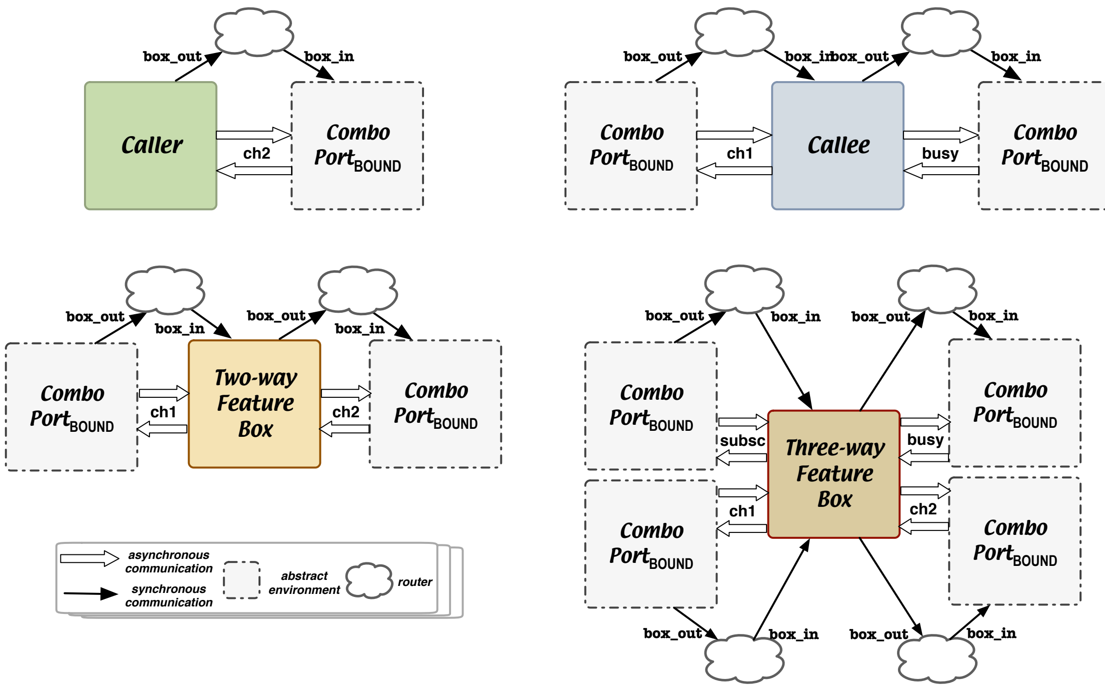

Figure 6.7: Abstract Environment to Verify Expected I/O Property

Boxes in the transparent (T) category can have at most two ports, but user agents may have more. The ports `subsc` (subscriber), `ch1`, and `ch2` are for regular communication, and the `busy_ch` port rejects the request for an additional connection. By using the combo bound port in the environment we capture the behaviour of multiple users communicating with a box as a caller or callee port at different times. The router process is included in the model, but the box is not dynamically created (as we are checking only one box) and the channels used are fixed in advance. As in our verification of fixed DFC configurations, we use synchronous communication for the channels `box_out` and `box_in`. We use asynchronous communication for the rest of the channels with a queue size of one. This forces the maximal amount of interleavings. We checked that the box and its environment are free of deadlock to show that this channel size is sufficient. Having a larger queue size would permit extra behaviours that involve fewer interleavings.

Checking for lack of deadlock in this model is not sufficient to ensure the expected I/O property since the deadlock check looks only for invalid end states[7]. Even if a deadlock check passes, a signal that should have been read by a process that is in a valid end state can remain in the queue. Therefore, we add error states to our interface and feature box processes, as well as to our abstract environment processes, and transitions to error states when the signal received is not one the process was expecting to process. We then describe the expected I/O property using an invariant in LTL, which has the form $\neg$ ($\Diamond$`Box@error`), thus the model checking reasoning reduces to verifying that the error state is never reached.

We verified the expected I/O property for the *T*, *UUA*, and *DUA* boxes that we described in our Spin model. The results obtained from the verifications of invariant properties are shown in Table 6.3.

Box verified	**Number of states**	**State vector (bytes)**	**Depth reached**	**Total memory used (Mbytes)**	**Time min:sec**
Caller	276	492	290	1.778	0:00.014
Callee	48	492	97	1.778	0:00.011
TFB	671	512	189	1.881	0:00.026
OCS	720	512	189	1.881	0:00.028
CF	683	520	191	1.881	0:00.026
CW	165297	544	8158	50.213	0:05.698

Table 6.3: Execution Statistics for Expected I/O Property Verification

In Spin, it is not possible to verify the DFC Box Properties in the environment we use for the expected I/O property (proving both DFC-compliance and expected I/O at one time), since we face the problem of false negatives due to interleaving, as explained in Chapter 5. In addition, the two types of properties are interesting individually and verification of them separately may help isolate errors.

6.6 Inductive Reasoning

The final step in our compositional verification method is to use the DFC box properties stated in Table 6.1 to prove the segment properties, stated in Table 4.1, for all segments of

[7] We recently learned that using XSPIN, there is a parameter that can be specified to make sure that queues are empty when checking for deadlock ("Extra Run-Time Option" `-q`). However, this is not the default behaviour, and not all versions of the tool support this advanced option.

a bounded but unknown length n. To do this proof, we use induction. To gain confidence in the correctness of this proof, we verified it using the HOL theorem proving system.

Having shown that the DFC box properties hold for a box in any DFC environment, we can restrict our reasoning at this point to rely only on the LTL properties. This restriction means we do not have to deal with the details of the models and greatly simplifies the reasoning. A graphical representation of the inductive process is shown in Figure 6.8, where every component represents a formula. Because we reason about *segments*, the base case is only an *UUA* box connected to a *DUA* box communicating through a queue, whereas the inductive step is an *UUA* box connected to n transparent boxes and finalized by a *DUA* box, all communicating through queues. Since we are assuming the correctness of the routing protocol, the box address can be represented using natural numbers in the order they appear in the usage.

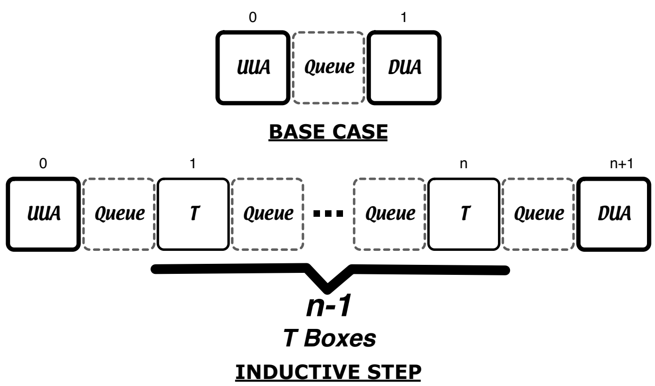

Figure 6.8: Inductive Reasoning

Rather than using an embedding of temporal logic in the logic of the theorem prover, we found it easier to convert the segment properties to their equivalent versions as functions of time. For example, $\Box p$ means that proposition p holds at all points in the future, and is expressed using propositions as functions of time as $\forall m \cdot p(t+m)$, where t the time at which the property is supposed to hold.

We model the sending and receiving of signals at a box as uninterpreted predicates:

Send p1 p2 p3 p4
Rec p1 p2 p3 p4

The parameters in order represent:

(p1) Time when the function takes place (type: num=natural numbers).

(p2) Label describing the originating box (type: `num`=natural numbers).

(p3) Label describing the destination box (type: `num`=natural numbers).

(p4) Signal being sent or received (signal types: `Setup`, `Upack`, `Td`, `Downack`, `Avail`, `Unavail`).

Because our focus is on verifying the behaviour of DFC boxes, we make the assumption that the unbounded queues behave perfectly, meaning that every signal sent is eventually received by the destination box. The unbounded queues property has the form "After the box *sent* a signal, it is held in the queue ready to be eventually *received* by the next box".

$$\Box\ (\mathsf{Send}(\mathtt{signal}) \Rightarrow \Diamond\ \mathsf{Receive}(\mathtt{signal}))$$

For example, the queue property for the *setup* signal, `QueueSetup`, is described using explicit time functions as follows:

$\forall \mathsf{c,t} \cdot (\mathsf{Send}\ \mathtt{t}\ \mathsf{c}\ (\mathsf{c}+1)\ \mathtt{Setup}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec}\ \mathsf{t2}\ \mathsf{c}\ (\mathsf{c}+1)\ \mathtt{Setup}))$

Most of the box properties, expressed as LTL formulas, have the form "After *receiving* a signal, the box reacts *sending* a signal on another channel".

$$\Box\ (\mathtt{Receive}(\mathtt{signal}) \Rightarrow \Diamond\ \mathtt{Send}(\mathtt{signal}))$$

For example, the box property for the *setup* signal, `BoxSetup`, is expressed using explicit time functions as follows:

$\forall \mathsf{c,t} \cdot (\mathsf{Rec}\ \mathtt{t}\ \mathsf{c}\ (\mathsf{c}+1)\ \mathtt{Setup}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Send}\ \mathsf{t2}\ (\mathsf{c}+1)\ (\mathsf{c}+2)\ \mathtt{Setup}))$

Using the DFC properties of boxes, plus the queue property, we prove the segment properties. For the segment properties, we describe first the reasoning for ***S*.1**, ***S*.4**, and ***S*.5**, and then discuss ***S*.2** and ***S*.3**.

Properties ***S*.1**, ***S*.4**, and ***S*.5** have the same form, the only difference is that in property ***S*.1** the *setup* signal goes from an *UUA* to a *DUA*, whereas in properties ***S*.4** and ***S*.5** the signals *avail* and *unavail* travel from a *DUA* to an *UUA*. Since they are symmetric, we need only to focus on one of the properties, say ***S*.1**, to explain the inductive proof. The proofs for properties ***S*.4**, and ***S*.5** are similar, but the induction goes from $n+1$ to 0 instead of from 0 to $n+1$. For example, for segment property ***S*.1** describing the propagation of `setup` signals, we prove,

$$
\begin{array}{l}
\quad /^{*}\mathtt{QueueSetup}^{*}/ \\
\quad \forall \mathsf{c,t} \cdot (\mathsf{Send\ t\ c\ (c{+}1)}\ \mathtt{Setup}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ c\ (c{+}1)}\ \mathtt{Setup}))\ , \\
\quad /^{*}\mathtt{BoxSetup}^{*}/ \\
\quad \forall \mathsf{c,t} \cdot (\mathsf{Rec\ t\ c\ (c{+}1)}\ \mathtt{Setup}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Send\ t2\ (c{+}1)\ (c{+}2)}\ \mathtt{Setup})) \\
\models \\
\quad \forall \mathsf{n,t} \cdot (\mathsf{Send\ t\ 0\ 1}\ \mathtt{Setup}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ n\ (n{+}1)}\ \mathtt{Setup}))
\end{array}
$$

Applying induction over n, we obtain two subgoals to prove, one corresponding to the base case, and one to the inductive step.

BASE CASE: Prove

$(\mathsf{Send\ t\ 0\ 1}\ \mathtt{Setup}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ 0\ 1}\ \mathtt{Setup}))$

We get this directly from the `QueueSetup` property.

INDUCTIVE STEP: Prove

$\forall \mathsf{n,t} \cdot (\mathsf{Send\ t\ 0\ 1}\ \mathtt{Setup}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ (n{+}1)\ (n{+}2)}\ \mathtt{Setup}))$

Proof sketch:

The Induction Hypothesis (*IH*) is

$$\forall \mathsf{t} \cdot (\mathsf{Send\ t\ 0\ 1}\ \mathtt{Setup}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ n\ (n{+}1)}\ \mathtt{Setup}))$$

First, we assume

$$\mathsf{Send\ t\ 0\ 1}\ \mathtt{Setup} \tag{1}$$

From *IH* and (1), we can conclude there is a t3 such that

$$(\mathsf{t3} > \mathsf{t}) \wedge (\mathsf{Rec\ t3\ n\ (n{+}1)}\ \mathtt{Setup}) \tag{2}$$

From `BoxSetup` and (2), we can conclude there is a t4 such that

$$(\mathsf{t4} > \mathsf{t3}) \wedge (\mathsf{Send\ t4\ (n{+}1)\ (n{+}2)}\ \mathtt{Setup}) \tag{3}$$

From `QueueSetup` and (3), we can conclude there is a t5 such that

$$(\mathsf{t5} > \mathsf{t4}) \wedge (\mathsf{Rec\ t5\ (n{+}1)\ (n{+}2)}\ \mathtt{Setup}) \tag{4}$$

Therefore, we can conclude

$$\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \land (\mathsf{Rec\ t2\ (n{+}1)\ (n{+}2)\ \texttt{Setup}}) \tag{5}$$

■

Properties ***S.2***, and ***S.3*** have the same form, the only difference is that in property ***S.2***, the *teardown* signal goes from an *UUA* to a *DUA*, whereas in property ***S.3*** the *teardown* signals travel from a *DUA* to an *UUA*. Then, since they are symmetric, we need only to focus on one of the properties, say ***S.2***, to explain the inductive proof. The proof for property ***S.3*** is similar, but the induction goes form $n+1$ to 0 instead of from 0 to $n+1$.

For segment property ***S.2*** describing the propagation of *teardown* signals, we prove,

```
/*QueueTdU*/
∀c,t · (Send t c (c+1) Td) ⇒ (∃t2 · (t2 > t) ∧ (Rec t2 c (c+1) Td)) ,
/*BoxTdU*/
∀c,t · (Send t c (c+1) Setup) ⇒
  (      (∃t2 · t2 > t ∧ ((Rec t2 c (c+1) Td)
                       ∧ (∃t3 · t3 > t2 ∧ Send t3 (c+1) (c+2) Td))
     ∧ (∀t4 · t < t4 < t2 ⇒ ¬(Rec t4 (c+2) (c+1) Td)))
   ∨   (∃t5 · t5 > t ∧ ((Rec t5 (c+2) (c+1) Td)
                       ∧ (∃t6 · t6 > t5 ∧ Send t6 (c+1) c Td))
     ∧ (∀t7 · t < t7 < t5 ⇒ ¬(Rec t7 c (c+1) Td)))
   ∨    (∀t8 · t8 > t ∧ ¬(Rec t8 c (c+1)Td) ∧ ¬(Rec t8 (c+2) (c+1)Td))
  )
|= ∀n,t' · Send t' 0 1 setup ⇒
   (∀t1 · ¬(∃i · (Rec t1 (i+1) i Td))) ⇒
    (∀t · (t > t' ∧ Send t 0 1 Td) ⇒ (∃t2 · (t2 > t) ∧ (Rec t2 n (n+1) Td)))
```

Applying induction over n, we obtain two subgoals to prove, one corresponding to the base case, and one to the inductive step.

BASE CASE: Prove

$\mathsf{Send\ t'\ 0\ 1\ \mathtt{setup}} \Rightarrow$
$(\forall \mathsf{t1} \cdot \neg(\exists \mathsf{i} \cdot (\mathsf{Rec\ t1\ (i{+}1)\ i}\ \mathtt{Td}))) \Rightarrow$
$(\forall \mathsf{t} \cdot (\mathsf{t} > \mathsf{t'} \wedge \mathsf{Send\ t\ 0\ 1}\ \mathtt{Td}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ 0\ 1}\ \mathtt{Td})))$

We get this directly form the `QueueTdU` property.

INDUCTIVE STEP: Prove

$\mathsf{Send\ t'\ 0\ 1\ \mathtt{setup}} \Rightarrow$
$(\forall \mathsf{t1} \cdot (\neg(\exists \mathsf{i} \cdot (\mathsf{Rec\ t1\ (i{+}1)\ i}\ \mathtt{Td})))) \Rightarrow$
$(\forall \mathsf{t} \cdot (\mathsf{t} > \mathsf{t'} \wedge \mathsf{Send\ t\ 0\ 1}\ \mathtt{Td}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ (n{+}1)\ (n{+}2)}\ \mathtt{Td})))$

Proof sketch:

The Induction Hypothesis (*IH*) is
$\mathsf{Send\ t'\ 0\ 1\ \mathtt{setup}} \Rightarrow$
$(\forall \mathsf{t1} \cdot \neg(\exists \mathsf{i} \cdot (\mathsf{Rec\ t1\ (i{+}1)\ i}\ \mathtt{Td}))) \Rightarrow$
$(\forall \mathsf{t} \cdot (\mathsf{t} > \mathsf{t'} \wedge \mathsf{Send\ t\ 0\ 1}\ \mathtt{Td}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ n\ (n{+}1)}\ \mathtt{Td}))))$

First, we assume

$$\mathsf{Send\ t'\ 0\ 1}\ \mathtt{Setup} \tag{1}$$

and

$$\forall \mathsf{t1} \cdot \neg(\exists \mathsf{i} \cdot (\mathsf{Rec\ t1\ (i{+}1)\ i}\ \mathtt{Td})) \tag{2}$$

and

$$\mathsf{Send\ t\ 0\ 1}\ \mathtt{Td} \tag{3}$$

and from properties ***S*.1**, `BoxSetup` and `QueueSetup`, we can deduce there is a t" such that

$$\mathsf{Send\ t''\ n\ (n{+}1)}\ \mathtt{Setup} \tag{4}$$

From *IH* and (1), we get

$$(\forall \mathsf{t1} \cdot \neg(\exists \mathsf{i} \cdot (\mathsf{Rec\ t1\ (i{+}1)\ i}\ \mathtt{Td}))) \Rightarrow (\forall \mathsf{t} \cdot (\mathsf{t} > \mathsf{t'} \wedge \mathsf{Send\ t\ 0\ 1}\ \mathtt{Td}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ n\ (n{+}1)}\ \mathtt{Td}))) \tag{5}$$

From (2) and (5), we get

$$((\mathsf{t} > \mathsf{t'} \wedge \mathsf{Send\ t\ 0\ 1}\ \mathtt{Td}) \Rightarrow (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ n\ (n{+}1)}\ \mathtt{Td}))) \tag{6}$$

From (3) and (6), we get

$$\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ n\ (n+1)\ \texttt{Td}}) \tag{7}$$

From `BoxTdU` and (4), we can conclude that for all n and t''

$$\begin{array}{ll}
(& (\exists \mathsf{t2} \cdot \mathsf{t2} > \mathsf{t''} \wedge ((\mathsf{Rec\ t2\ n\ (n+1)\ \texttt{Td}}) \\
 & \qquad\qquad \wedge\ (\exists \mathsf{t3} \cdot \mathsf{t3} > \mathsf{t2} \wedge \mathsf{Send\ t3\ (n+1)\ (n+2)\ \texttt{Td}})) \\
 & \wedge (\forall \mathsf{t4} \cdot \mathsf{t''} < \mathsf{t4} < \mathsf{t2} \Rightarrow \neg(\mathsf{Rec\ t4\ (n+2)\ (n+1)\texttt{Td}}))) \\
\vee & (\exists \mathsf{t5} \cdot \mathsf{t5} > \mathsf{t''} \wedge ((\mathsf{Rec\ t5\ (n+2)\ (n+1)\ \texttt{Td}}) \\
 & \qquad\qquad \wedge\ (\exists \mathsf{t6} \cdot \mathsf{t6} > \mathsf{t5} \wedge \mathsf{Send\ t6\ (n+1)\ n\ \texttt{Td}})) \\
 & \wedge (\forall \mathsf{t7} \cdot \mathsf{t''} < \mathsf{t7} < \mathsf{t5} \Rightarrow \neg(\mathsf{Rec\ t7\ n\ (n+1)\texttt{Td}}))) \\
\vee & (\forall \mathsf{t8} \cdot \mathsf{t8} > \mathsf{t''} \wedge \neg(\mathsf{Rec\ t8\ n\ (n+1)\texttt{Td}}) \wedge \neg(\mathsf{Rec\ t8\ (n+2)\ (n+1)\texttt{Td}})) \\
) &
\end{array} \tag{8}$$

From (8), we get three cases, one for each disjunct:

In CASE *(a)*, we assume

$$\begin{array}{ll}
(& (\exists \mathsf{t2} \cdot \mathsf{t2} > \mathsf{t''} \wedge ((\mathsf{Rec\ t2\ n\ (n+1)\ \texttt{Td}}) \\
 & \qquad\qquad \wedge\ (\exists \mathsf{t3} \cdot \mathsf{t3} > \mathsf{t2} \wedge \mathsf{Send\ t3\ (n+1)\ (n+2)\ \texttt{Td}})) \\
 & \wedge\ (\forall \mathsf{t4} \cdot \mathsf{t} < \mathsf{t4} < \mathsf{t2} \Rightarrow \neg(\mathsf{Rec\ t4\ (n+2)\ (n+1)\ \texttt{Td}})) \\
) &
\end{array} \tag{9}$$

In (9), we know there is t3 and t2 such that

$$\mathsf{t3} > \mathsf{t2} \wedge \mathsf{Send\ t3\ (n+1)\ (n+2)\ \texttt{Td}} \tag{10}$$

From `QueueTdU` and (10), we can conclude there is a t4 such that

$$\mathsf{t4} > \mathsf{t3} \wedge \mathsf{Rec\ t4\ (n+1)\ (n+2)\ \texttt{Td}} \tag{11}$$

Therefore in this case, we can conclude,

$$\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ (n+1)\ (n+2)\ \texttt{Td}}) \tag{12}$$

In CASE *(b)*, we assume

$$
\begin{aligned}
(\quad & (\exists \mathsf{t5} \cdot \mathsf{t5} > \mathsf{t''} \wedge ((\mathsf{Rec\ t5\ (n{+}2)\ (n{+}1)}\ \mathtt{Td}) \\
& \qquad\qquad \wedge\ (\exists \mathsf{t6} \cdot \mathsf{t6} > \mathsf{t5} \wedge \mathsf{Send\ t6\ (n{+}1)\ n}\ \mathtt{Td})) \\
& \wedge\ (\forall \mathsf{t7} \cdot \mathsf{t} < \mathsf{t7} < \mathsf{t5} \Rightarrow \neg(\mathsf{Rec\ t7\ n\ (n{+}1)}\mathtt{Td})) \\
) &
\end{aligned}
\tag{13}
$$

In (13), we know there is a t5 such that

$$\mathsf{t5} > \mathsf{t''} \wedge \mathsf{Rec\ t5\ (n{+}2)\ (n{+}1)}\ \mathtt{Td} \tag{14}$$

From (2), we know

$$\neg(\mathsf{Rec\ t5\ (n{+}2)\ (n{+}1)}\ \mathtt{Td}) \tag{15}$$

From the contradiction between (14) and (15), we can conclude this case is not possible.

In CASE *(c)*, we assume

$$\forall \mathsf{t8} \cdot \neg(\mathsf{Rec\ t8\ n\ (n{+}1)}\mathtt{Td}) \wedge \neg(\mathsf{Rec\ t8\ (n{+}2)\ (n{+}1)}\mathtt{Td}) \tag{16}$$

In (7), we know there is a t8 such that

$$\mathsf{t8} > \mathsf{t} \wedge (\mathsf{Rec\ t8\ n\ (n{+}1)}\ \mathtt{Td}) \tag{17}$$

From (16), we can conclude that for all t8

$$\neg(\mathsf{Rec\ t8\ n\ (n{+}1)}\mathtt{Td}) \wedge \neg(\mathsf{Rec\ t8\ (n{+}2)\ (n{+}1)}\mathtt{Td}) \tag{18}$$

From the contradiction between (17) and (18), we can conclude this case is not possible.

Therefore, we can conclude by cases

$$\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ (n{+}1)\ (n{+}2)}\ \mathtt{Td}) \tag{19}$$

The complete proofs were done in the HOL theorem prover. ■

6.7 Summary

In this chapter we described the compositional reasoning method for verifying liveness properties of the DFC call signalling protocol, so we can conclude that segments with any number of boxes satisfy our correctness properties.

Chapter 7

Conclusions

In this chapter, we summarize the main contributions of our work, as well as its limitations. Finally, we discuss ideas to extend and improve our approach as future work.

7.1 Contributions

In this work, we identified and proposed a definition of DFC compliance for features, concentrating on the call protocol, *i.e.,* we verify that the system behaves well, rather than detecting feature interactions. This definition describes our correctness criteria as a set of LTL properties for segments. One of the main contributions in our work is a categorization of boxes by their influence on the routing process, which helps to reason about segments of a DFC usage and is used to express our correctness properties.

We created an environment for model checking fixed DFC segments, which is a sequence of transparent features in a pipeline delimited by user agent features. The creation of the segment was done dynamically, following closely the DFC architecture. This model checking effort was a useful debugging exercise to see if our models of boxes were fully specified and if our segment properties were correct. We detected surprising feature behaviours during the verification of fixed configurations. The behaviours discovered were not specified in the feature's descriptions given in the literature, *e.g.,* scenarios for the call waiting subscriber when several parties are calling and hanging up. After conferring with Pamela Zave, we handled the situation of the call back processing failure, and provided a full and correct specification for the call waiting feature. However, call waiting could not be fully verified for the segment properties in a small configuration of two Callers and one Callee, even when increasing the memory and maximum depth search in Spin to the highest limits allowed by the server. Therefore, this effort is not sufficient to conclude that the properties hold for segments with any number of features.

To verify liveness properties of the DFC call signalling protocol for segments of length n, we developed a *compositional reasoning method*, that combines theorem proving and model checking. Our verification method allows us to reason about each feature individually, considerably reducing the verification effort. Our method consists of four steps:

DFC Compliance: Verify, using model checking, that an individual box reacts to the call signals in agreement with the DFC architecture constraints, using an unrestricted environment and synchronous communication.

Expected I/O: Verify, using model checking, that an individual box receives only the signals it is expecting and sends only the signals expected by the environment, using an abstract environment that captures the most general port protocol behaviour of DFC, and asynchronous communication.

Port Compliance: Prove that the behaviour of every port in the box is within the most general behaviour allowed by DFC. We proved the language containment relationships between a partial order of abstract port models so that the most specific abstract port model could be used for checking the port compliance of a box.

Inductive Reasoning: Prove by induction over the number of boxes communicating through unbounded queues that the segment properties hold for segments of length n. Queues are assumed to provide perfect communication between boxes in any DFC configuration.

This form of compositional reasoning is possible because of the semi-regularity of DFC features. With our method, we achieve a separation of concerns in that features can be verified individually, and a description in terms of only LTL properties is used to reason about segments. The inductive reasoning step needs to be completed only once and is in terms of LTL properties only. Thereafter, using this compositional reasoning approach, to add a new feature to the DFC architecture, and ensure that all segments of length n satisfy the segment properties, each new features must follow the proof obligations described as DFC compliance, expected I/O and port compliance.

7.2 Limitations

In the environment created for model checking fixed DFC segments, only four feature boxes were implemented: free transparent feature (FTF), call forwarding (CF) and originating call screening (OCS) as free feature boxes, as well as call waiting (CW) as a bound feature box.

We abstracted away some feature specific behaviours to simplify the modeling and verification. In OCS, we chose non-deterministically if a user to be called is in the screening list instead of checking the global information. In CW, we abstracted away two signals. We did not implement the *wait* tone, which lets the subscriber know that a new correspondant is on hold. We do not use the *switch* signal, which allows the subscriber to change between correspondants, and instead, we make this a non-deterministic choice. In our model, we also assume that the routing is carried out correctly.

7.3 Future Work

We model checked some small configurations of the DFC architecture, varying the number of Callers, Callees, and number of times a Caller could make a call. To make it possible to model check larger configurations, an interesting avenue for further exploration is symmetry reductions (*e.g.,* [11]) which 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.

Currently, segment properties allow us to reason about the signals traversing the segment in one direction. Next, we plan to tackle round-trip properties, which involve the receipt of acknowledgments for signals that have been sent. For example, one of our properties guarantees that a *setup* signal travels from a *DUA* to an *UUA*. Once the *setup* signal is received, the *DUA* generates an *avail* or *unavail* signal. Then, another property states that *avail* or *unavail* signals travels upstream, reaching the *DUA* that generated the *setup* signal. We also expect there are additional properties of segments, plus many other interesting properties of DFC (*e.g.,* address translation, properties of branching usages) that can be proved using the compositional method presented in this work. We would also like to introduce properties that can be used to identify the category of a feature. For example, a property could describe what it means for a user agent to have the behaviour of absorbing a *setup* signal, or generating an *avail* or *unavail* signal.

We are working on creating an environment in which Boxtalk models can be verified directly using our method, and in which we can begin to explore the application of this method to other pipe-and-filter systems that exhibit a semi-regularity in their behaviour. We also want to use other model checkers, in which events can be modeled more easily, while avoiding the use of global boolean variables.

Lastly, it was not clear what the optimal size of channels should be when asynchronous communication is allowed. This is closely related to the issue of granularity, *i.e.,* how many statements to include in an `atomic` sequence. Coarse granularity implies less interleaving, thus some behaviours would not be checked, but fine granularity can make the verification effort less efficient. To cover all possible interleaving behaviours, we decided to use `atomic` sequences only when needed, *i.e.,* conditions in a process that must be changed without interleaving (*e.g.,* busy bit changing from false to true) or when a set of statements are intended to represent the same action (*e.g.,* the event of sending or receiving on a channel and the change of the boolean variable to represent the event) to maximize interleavings. However, it seems likely that when combined with certain sizes of channels, fewer interleavings would be sufficient.

7.4 Summary

In this work, we proposed a definition of DFC compliance with respect to the call protocol, using a set of LTL properties. To model check fixed configurations of DFC box models, we created an environment in the model checker Spin. We also described a compositional method for verifying liveness properties of the DFC call signalling protocol for segments of length n. To define the segment properties, we categorized boxes by their influence on the routing process. Our method consists of verifying an individual box, and then, in a step that needs only to be completed once and is in terms of LTL properties only, we proved by induction that the segment properties hold for segments of length n. This form of compositional reasoning is possible because of the semi-regularity of DFC features.

Appendix 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 callers in the system. Different from N for
    verification purposes, leading to different configurations.*/
#define Cr 0
        /* Number of callees in the system. Different from N for
    verification purposes, leading to different configurations.*/
#define Ce 0
    /* Number of bits needed to identify a channel with (2^L)>M*/
#define L 4
    /* Number of times Caller can call*/
#define TIMES 1

    /* 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_FTF[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}*/
    chan intern = [1] of {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*/
bool channel_busy [M];

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

    /* To find the first idle Com_chan. Channels can be reused */
inline C_uassign(iden)
{
  atomic {     /*buggy*/ /*comment*/
      do
    ::!channel_busy[iden] -> break
         ::channel_busy[iden] -> channel_busy[iden] = false; break
    od;
```

```
   }             /*buggy*/ /*comment*/
}

    /* For verification */

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

A.1.2 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_FTF[0]=false;
    subs_FTF[1]=false;
    subs_FTF[2]=false;
    subs_FTF[3]=false;
    subs_FTF[4]=false;
    subs_FTF[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] = false;
    subs_CW[1] = false;
    subs_CW[2] = false;
    subs_CW[3] = false;
    subs_CW[4] = false;
    subs_CW[5] = false;

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

A.2 Caller process

```
proctype Caller(byte me)
{
```

```
byte dest;          /*the non deterministically chosen callee*/
unsigned ch : L;      /*the channel identifier of the free channel*/
byte counter=0;     /*count number of times Caller repeats*/

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

condition:
    if
    ::(counter==TIMES) -> goto end
    ::else -> counter= counter+1
    fi;
end_idle:
  atomic{
    do
        /*Sets up the busy bit*/
    ::(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 callee before sending setup*/
    if
    ::dest = user0
    ::dest = user1
    ::dest = user2
    ::dest = user3
    ::dest = user4
    ::dest = user5
    fi;
    box_out!setup,me,dest,dest,ch;
    a=true;
   }

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


/*wait the next signal either unavail or avail*/
```

```
wait_vail:
    if
    ::atomic{
      chan_array[ch].A?unavail ->
      j=true;
      goto wait_teardown
      }
    ::atomic{
      chan_array[ch].A?avail;
      m=true;
      goto end_linked
      }
    fi;

end_linked:    /* can stay connected forever */
    do
    ::atomic{
       chan_array[ch].B!teardown->
       c=true;
       goto unlinking
      }
    ::atomic{
       chan_array[ch].A?teardown->
       l=true;
       chan_array[ch].B!downack;
       d=true;
       busy[me-1]=false;
       goto condition
      }
    od;

wait_teardown:
    atomic{
      chan_array[ch].A?teardown;
      l=true;
      chan_array[ch].B!downack;
      d=true;
      busy[me-1]=false;
      goto condition
    }

unlinking:
    do
    ::atomic{
         chan_array[ch].A?downack->
```

```
         n=true;
         busy[me-1]=false;
         goto condition
        }
    ::atomic{
         chan_array[ch].A?teardown->
         l=true;
         chan_array[ch].B!downack;
         d=true
        }
    od;
end : skip
}
```

A.3 Callee process

```
proctype Callee (byte me)
{
 byte scr2;byte scr4;     /*scratch values*/
 byte ch, ch1;
 byte msg;

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


end_idle:
        /* Shouldn't block because it's rendezvous channel */
    Callee_channel[me-1].intern?ch2;
    atomic{
        box_in?setup,scr2,eval(me),scr4,eval(ch2) ->
        b=true;
        chan_array[ch2].A!upack;
        e=true;
        }
    goto send_vail;

send_vail:
    do
    ::atomic{
        (busy[me-1] == false) ->
        busy[me-1]= true;
```

```
        }
        atomic{
            chan_array[ch2].A!avail;
            h=true;
        }
           goto end_linked;
    ::(busy[me-1] == true) ->
        atomic{
            chan_array[ch2].A!unavail;
            i=true;
        }
        goto send_td
    ::Callee_channel[me-1].intern?ch ->
        atomic{
            box_in?setup,scr2,eval(me),scr4,eval(ch) ->
            a=true;
            chan_array[ch].A!upack;
            k=true;
        }
        atomic{
            chan_array[ch].A!unavail;
            j=true;
        }
        atomic{
            chan_array[ch].A!teardown;
            l=true;
        }
            goto unlink_send_vail
    od;

end_linked: /* can stay connected forever */
    do
    ::atomic{
       chan_array[ch].A!teardown;
       g=true;
       goto unlinking_nonbusy
        }
    ::atomic{
       chan_array[ch].B?teardown->
       o=true;
       chan_array[ch].A!downack;
       f=true;
       busy[me-1]=false;
       goto end_idle
        }
```

```
    ::Callee_channel[me-1].intern?ch1 ->
        atomic{
            box_in?setup,scr2,eval(me),scr4,eval(ch1) ->
            b=true;
            chan_array[ch1].A!upack;
            e=true;
            chan_array[ch1].A!unavail;
            i=true;
            chan_array[ch1].A!teardown;
            g=true;
            goto unlinking2
        }
    od;

unlinking_nonbusy:
    do
    ::atomic{chan_array[ch].B?downack->
        p=true;
        busy[me-1]=false;
        goto end_idle
         }
    ::atomic{chan_array[ch].B?teardown->
        o=true;
        chan_array[ch].A!downack;
        f=true
        }
    ::Callee_channel[me-1].intern?ch1 ->
        atomic{
            box_in?setup,scr2,eval(me),scr4,eval(ch1) ->
            b=true;
            chan_array[ch1].A!upack;
            e=true;
            chan_array[ch1].A!unavail;
            i=true;
            chan_array[ch1].A!teardown;
            g=true;
            goto unlinking3
        }
    od;

unlinking2:
    atomic{
        chan_array[ch1].B?downack->
        p=true;
        goto end_linked
```

```
        }

unlinking3:
    atomic{
        chan_array[ch1].B?downack->
        p=true;
        goto unlinking_nonbusy
        }

unlinking_busy:
    do
    ::atomic{chan_array[ch].B?downack->
        p=true;
        goto end_idle
         }
    ::atomic{chan_array[ch].B?teardown->
        o=true;
        chan_array[ch].A!downack;
        f=true
        }
    od;

end: skip
}
```

A.4 Router processes

A.4.1 routeruser process

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

atomic{
    box_out?setup,eval(thisindex),number,dest,ch;
    if
    ::subs_FTF[thisindex-1] -> atomic {
        run FTF(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{
                CW_channel[thisindex-1].intern!ch;
                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{
                        CW_channel[number-1].intern!ch;
                        box_in!setup,thisindex,number,number,ch}
                    ::else -> atomic{
                        Callee_channel[number-1].intern!ch;
                        box_in!setup,thisindex,number,number,ch}
                    fi;
                fi;
            fi;
        fi;
    fi;
    }
end:skip
}
```

A.4.2 routerFTF process (Source region)

```
proctype routerFTF (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{
            CW_channel[thisindex-1].intern!ch;
            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{
                    CW_channel[number-1].intern!ch;
                    box_in!setup,thisindex,number,number,ch}
                ::else -> atomic{
                    Callee_channel[number-1].intern!ch;
                    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{
        CW_channel[thisindex-1].intern!ch;
        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{
                CW_channel[number-1].intern!ch;
                box_in!setup,thisindex,number,number,ch}
            ::else -> atomic{
                Callee_channel[number-1].intern!ch;
                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{
            CW_channel[number-1].intern!ch;
            box_in!setup,thisindex,number,number,ch}
        ::else -> atomic{
            Callee_channel[number-1].intern!ch;
            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{
        CW_channel[number-1].intern!setup,thisindex,number,dest,ch;*/
        CW_channel[number-1].intern!ch;
        box_in!setup,thisindex,number,number,ch}
    ::else -> atomic{
        Callee_channel[number-1].intern!ch;
        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,eval(thisuser),ch;
    Callee_channel[number-1].intern!ch;
    box_in!setup,orig,number,number,ch;
  }
}
```

A.5 Feature boxes processes

A.5.1 Free Transparent process

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

```
    unsigned ch1: L; /* 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 routerFTF(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 Originating Call Screening process

```
proctype OCS(byte thisuser)
{
 byte number;
 byte dest;
 unsigned ch1 : L; /* index of the previous communication channel*/
 unsigned ch2 : L; /* 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 ->
        b=true;
        chan_array[ch1].A!upack;
        e=true;
    if /*non_deterministic choice*/
        /*blocking*/
    :: chan_array[ch1].A!unavail;
        i=true;
        chan_array[ch1].A!teardown;
        g=true;
        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->
        o=true;
        chan_array[ch1].A!downack;
        f=true
         }
    ::atomic{
        chan_array[ch1].B?downack->
        p=true;
        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; /* index of the next communication channel*/
 unsigned ch1: L; /* 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(byte thisuser)
{
 bool calling;    /* denote subscriber calling or being called */
 bool talk;     /* non-deterministic: first/second party talking */
 byte callers;    /* denote number of callers connected to CW */
 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 extra: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?ch;
      box_in?setup,orig,number,dest,eval(ch);
```

```
    connCW[thisuser-1]=1;
    if  /*no fake id possible*/
    ::orig == thisuser -> /* subscriber is calling */
        orig2=orig; number2=number; dest2=dest;
        calling=true;
        callers=1;
        subsc=ch;
        chan_array[subsc].A!upack;
        goto wait_upack_src;
    ::else ->             /* subscriber is being called */
        orig3=orig; number3=number; dest3=dest;
        calling = false;
        callers=1;
        first=ch;
        chan_array[first].A!upack;
        goto wait_upack_trg;
    fi;
  }

wait_upack_src:
    C_assign(first);
    run routerCWsrc(thisuser);
    box_out!setup,orig,number,dest,first;
    chan_array[first].A?upack;
    goto wait_vail_src;

wait_vail_src:
    if
    :: atomic{
            chan_array[first].A?avail ->
            chan_array[subsc].A!avail;
            goto con1src_first1
        }
    :: atomic{
            chan_array[first].A?unavail ->
            chan_array[subsc].A!unavail;
            goto wait_dn_src
        }
    fi;

wait_dn_src:
    atomic{
        chan_array[first].A?teardown;
        chan_array[subsc].A!teardown;
        chan_array[first].B!downack;
```

```
        goto unlink1src
    }

wait_upack_trg:
    C_assign(subsc);
    run routerCWtrg(thisuser);
    box_out!setup,orig,number,dest,subsc;
    chan_array[subsc].A?upack;
    goto wait_vail_trg;

wait_vail_trg:
    if
    :: atomic{
                chan_array[subsc].A?avail ->
                chan_array[first].A!avail;
                goto con1trg_first1
        }
    :: atomic{
                chan_array[subsc].A?unavail ->
                chan_array[first].A!unavail;
                goto wait_dn_trg
        }
    fi;

wait_dn_trg:
    atomic{
        chan_array[subsc].A?teardown;
        chan_array[first].A!teardown;
        chan_array[subsc].B!downack;
        goto unlink0trg
    }

con1src_first1:
    connCW[thisuser-1]=1;
      do
    ::atomic{
        chan_array[first].A?teardown ->
        chan_array[first].B!downack;
        chan_array[subsc].A!teardown;
        goto unlink1src
        }
    ::atomic{
        chan_array[first].B?teardown ->
        chan_array[first].A!downack;
        chan_array[subsc].A!teardown;
```

```
        goto unlink1src
        }
    ::atomic {
        chan_array[subsc].B?teardown ->
        chan_array[subsc].A!downack;
        if
        /* 2 callers talking to subscriber ->
         * channels of first as caller, and trg scenario
         */
        ::(callers>=2) ->
            chan_array[first].A!teardown;
            goto unlink0trg
        /* 1 caller and 1 callee talking to subscriber ->
         * channels of first as callee, and src scenario
         */
        ::else ->
            chan_array[first].B!teardown;
            goto unlink0src
        fi;
        }
      ::atomic{
          CW_channel[thisuser-1].intern?ch;
          box_in?setup,orig,number,dest,eval(ch);
          b=true;
          connCW[thisuser-1]=2;
          callers=callers+1;
          orig3=orig;
          number3=number;
          dest3=dest;
          second=ch;
          chan_array[second].A!upack;
          e=true;
          goto con1src_first2
        }
    od;

con1src_first2:
    atomic{
        chan_array[second].A!avail;
        h=true;
        if
        ::talk = true /*port 2 will be the talking one*/
        ::talk = false /*port 3 will be the talking one*/
        fi;
        goto con2src
```

```
    };

con1src_second1:
    connCW[thisuser-1]=1;
      do
    ::atomic{
        chan_array[second].B?teardown ->
        chan_array[second].A!downack;
        chan_array[subsc].A!teardown;
        goto unlink1src
        }
    ::atomic {
        chan_array[subsc].B?teardown ->
        chan_array[subsc].A!downack;
        chan_array[second].A!teardown;
        goto unlink2src
        }
      ::atomic{
          CW_channel[thisuser-1].intern?ch;
          box_in?setup,orig,number,dest,eval(ch);
          b=true;
          connCW[thisuser-1]=2;
          callers=callers+1;
          orig2=orig;
          number2=number;
          dest2=dest;
          first=ch;
          chan_array[first].A!upack;
          e=true;
          goto con1src_second2
        }
    od;

con1src_second2:
    atomic{
        chan_array[first].A!avail;
        h=true;
        if
        ::talk = true /*port 2 will be the talking one*/
        ::talk = false /*port 3 will be the talking one*/
        fi;
        goto con2src
    };

unlink0src:
```

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

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

unlink2src:
    do
    ::atomic {
        chan_array[second].B?downack ->
        connCW[thisuser-1]=0;
        callers=0;
        goto end
        }
    ::atomic{
        chan_array[second].B?teardown ->
        chan_array[second].A!downack;
        }
    od;

con1trg_first1:
    connCW[thisuser-1]=1;
      do
    ::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 {
        chan_array[subsc].A?teardown ->
        chan_array[subsc].B!downack;
        if
        ::(callers>=1) ->
            chan_array[first].A!teardown;
            goto unlink0trg
        ::else ->
            chan_array[first].B!teardown;
            goto unlink0src
        fi;
        }
      ::atomic{
          CW_channel[thisuser-1].intern?ch;
          connCW[thisuser-1]=2;
          callers=callers+1;
          orig3=orig;
          number3=number;
          dest3=dest;
          second=ch;
          chan_array[second].A!upack;
          e=true;
          goto con1trg_first2
        }
    od;

con1trg_first2:
    atomic{
        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
```

```
        };

con1trg_second1:
    connCW[thisuser-1]=1;
      do
     ::atomic{
        chan_array[second].B?teardown ->
        chan_array[second].A!downack;
        chan_array[subsc].B!teardown;
        goto unlink1trg
        }
     ::atomic {
        chan_array[subsc].A?teardown ->
        chan_array[subsc].B!downack;
        chan_array[second].A!teardown;
        goto unlink2trg
        }
      ::atomic{
          CW_channel[thisuser-1].intern?ch;
          box_in?setup,orig,number,dest,eval(ch);
          b=true;
          connCW[thisuser-1]=2;
          callers=callers+1;
          orig2=orig;
          number2=number;
          dest2=dest;
          first=ch;
          chan_array[first].A!upack;
          e=true;
          goto con1trg_second2
        }
    od;

con1trg_second2:
    atomic{
        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
        };

unlink0trg:
    do
```

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

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

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

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

```
        chan_array[subsc].A!downack;
        };
        connCW[thisuser-1]=1;
        if
        ::talk-> /**connecting to second party, disconnecting first**/
            if
            ::(callers==3) ->
                chan_array[first].A!teardown;
                goto unlink3srcA
            ::else ->
                chan_array[first].B!teardown;
                goto unlink3srcB
            fi;
        ::else -> /**connecting to first party, disconnecting second**/
            chan_array[second].A!teardown;
            if
            ::(callers==3) ->
            /*since subscr=caller instead of callee in =>trg scenario*/
                callers=callers-1;
                goto unlink4trg
            ::else ->
                goto unlink4src
            fi;
        fi;

setup_sec_src:    /**connecting to second party, disconnecting first**/
        atomic{
            run routerCWtrg(thisuser);
            C_assign(ch);
            box_out!setup,orig,number,dest,ch;
            subsc=ch;
            }
        do
            /* WAIT_UPACK from subscriber*/
        ::chan_array[subsc].A?upack;
            goto wait_response_second_src
          ::atomic{
              CW_channel[thisuser-1].intern?new;
            box_in?setup,orig,number,dest,eval(new);
            b=true;
            connCW[thisuser-1]=2;
            }
            if
            /* SITUATION (2) Subscriber calls somebody else [1][2]*/
            ::orig == thisuser ->
```

```
                orig2=orig; number2=number; dest2=dest;
                calling=true;
                callers=callers+1;
                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;
                callers=callers+1;
                first=new;
                chan_array[first].A!upack;
                e=true;
                goto wait_vail_sec1_src
            fi;
        od;

wait_vail_sec1_src:
        atomic{
            chan_array[first].A!avail;
            h=true;
            goto link_nonsubsc_sec
            };

unlink_old_subsc_sec: /* SITUATION 2 */
        if
        ::atomic{    /* Subscriber as in trg scenario */
            chan_array[ch].A?upack;
            k=true;
            chan_array[ch].A?unavail;
            j=true;
            chan_array[ch].A?teardown;
            l=true;
            chan_array[ch].B!downack;
            d=true;
            goto wait_upack_sec2
            }
        ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
```

```
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_unlink_old_subsc_sec
            }
        fi;

dn_unlink_old_subsc_sec:
           atomic{
            chan_array[extra].B?downack;
            p=true;
            goto unlink_old_subsc_sec
            };

wait_upack_sec2:
        if
        ::atomic{
            chan_array[first].A?upack;
            goto wait_vail_sec2
             }
        ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_wait_upack_sec2
            }
        fi;

dn_wait_upack_sec2:
           atomic{
            chan_array[extra].B?downack;
            p=true;
            goto wait_upack_sec2
            };

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;
            j=true;
            chan_array[subsc].A!avail;
            h=true;
            goto wait_td_sec2
             }
        ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_wait_vail_sec2
            }
        fi;

dn_wait_vail_sec2:
           atomic{
            chan_array[extra].B?downack;
            p=true;
            goto wait_vail_sec2
            };

wait_td_sec2:
        atomic{
            chan_array[first].A?teardown;
            l=true;
            chan_array[first].B!downack;
            d=true;
            goto con1src_second1
```

```
             };

link_nonsubsc_sec: /* SITUATION 3 */
        if
        ::atomic{
            chan_array[subsc].A?upack;
            goto wait_response_sec2
             }
         ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_link_nonsubsc_sec
            }
        fi;

dn_link_nonsubsc_sec:
           atomic{
            chan_array[extra].B?downack;
            p=true;
            goto link_nonsubsc_sec;
             };

wait_response_sec2:
        if
        ::atomic{
            chan_array[subsc].A?avail;
            m=true;
            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[subsc].A?unavail;
            j=true;
            goto wait_td_sec1_trg
             }
```

```
        ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_wait_response_sec2
            }
            fi;

dn_wait_response_sec2:
           atomic{
            chan_array[extra].B?downack;
            p=true;
            goto wait_response_sec2
            };

wait_td_sec1_trg:
        atomic{
            chan_array[subsc].A?teardown;
            chan_array[subsc].B!downack;
            chan_array[first].A!teardown;
               goto unlink3srcA
             };

wait_td_sec1_trg_bis:
        atomic{
            chan_array[subsc].A?teardown;
            chan_array[subsc].B!downack;
            chan_array[second].A!teardown;
               goto unlink2trg
             };

wait_response_second_src: /*Normal situation:connecting to second party*/
        do
           ::atomic{
            chan_array[subsc].A?avail;
            m=true;
            goto con1trg_second1
            }
        ::atomic{
```

```
            chan_array[subsc].A?unavail;
            j=true;
            goto wait_td_sec_src
            }
        od;

wait_td_sec_src:
        atomic{
            chan_array[subsc].A?teardown;
            chan_array[subsc].B!downack;
            chan_array[second].A!teardown;
            goto unlink2src
             };

setup_first_src: /**connecting to first party, disconnecting second**/
        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;
            subsc=ch;
            }
        do
        ::atomic{
            chan_array[subsc].A?upack;
            goto wait_response_first_src
            }
          ::atomic{
              CW_channel[thisuser-1].intern?new;
            box_in?setup,orig,number,dest,eval(new);
            b=true;
            connCW[thisuser-1]=2;
            }
            if
            /* SITUATION (2) Subscriber calls somebody else [1][2] */
            ::orig == thisuser ->
                   orig2=orig; number2=number; dest2=dest;
                calling=true;
                callers=callers+1;
                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 [3][4] */
            ::else ->
                orig3=orig; number3=number; dest3=dest;
                calling = false;
                callers=callers+1;
                second=new;
                chan_array[second].A!upack;
                e=true;
                goto wait_vail_first1_src
            fi;
        od;

wait_vail_first1_src:
        atomic{
            chan_array[second].A!avail;
            h=true;
            goto link_nonsubsc_first
            };

unlink_old_subsc_first_src: /* SITUATION 2 */
        if
        ::atomic{
            chan_array[ch].A?upack;
            k=true;
            chan_array[ch].A?unavail;
            j=true;
            chan_array[ch].A?teardown;
            l=true;
            chan_array[ch].B!downack;
            d=true;
            goto wait_upack_first_2src
             }
        ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
```

```
            g=true;
            goto dn_unlink_old_subsc_first_src
            }
        fi;

dn_unlink_old_subsc_first_src:
           atomic{
            chan_array[extra].B?downack;
            p=true;
            goto unlink_old_subsc_first_src;
             };

wait_upack_first_2src:
        if
        ::atomic{
            chan_array[second].A?upack;
            goto wait_vail_first_2src
             }
        ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_wait_upack_first_2src
            }
        fi;

dn_wait_upack_first_2src:
           atomic{
            chan_array[extra].B?downack;
            goto wait_upack_first_2src;
            };

wait_vail_first_2src:
        if
        ::atomic{
            chan_array[second].A?avail;
            chan_array[subsc].A!avail;
            new=first;
            first=second;
```

```
            second=new;
            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].A?unavail;
            j=true;
            chan_array[subsc].A!avail;
            h=true;
            goto wait_td_first1_src
            }
        ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_wait_vail_first_2src
            }
        fi;

dn_wait_vail_first_2src:
           atomic{
            chan_array[extra].B?downack;
            p=true;
            goto wait_vail_first_2src
            };

wait_td_first1_src:
        atomic{
            chan_array[second].A?teardown;
            l=true;
            chan_array[second].B!downack;
            d=true;
            goto con1src_first1
            };

link_nonsubsc_first: /* SITUATION 3 */
```

```
        if
        ::atomic{
            chan_array[subsc].A?upack;
            goto wait_response_first2
            }
        ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_link_nonsubsc_first
            }
        fi;

dn_link_nonsubsc_first:
           atomic{
            chan_array[extra].B?downack;
            p=true;
            goto link_nonsubsc_first
            };

wait_response_first2:
        if
        ::atomic{
            chan_array[subsc].A?avail;
            m=true;
            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[subsc].A?unavail;
            j=true;
            goto wait_td_first1_trg
             }
        ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
```

```
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_wait_response_first2
            }
        fi;

dn_wait_response_first2:
           atomic{
            chan_array[extra].B?downack;
            goto wait_response_first2
            };

wait_td_first1_trg:
        atomic{
            chan_array[subsc].A?teardown;
            chan_array[subsc].B!downack;
            chan_array[second].A!teardown;
            /* Go back to call back processing for second */
               goto unlink4src
               };

wait_td_first1_trg_bis:
        atomic{
            chan_array[subsc].A?teardown;
            chan_array[subsc].B!downack;
            if
            ::(callers>1) ->
                chan_array[first].A!teardown;
                goto unlink0trg
            ::else ->
                chan_array[first].B!teardown;
                goto unlink0src
            fi;
            };

wait_response_first_src: /*Normal situation:connecting to first party*/
        do
           ::atomic{
            chan_array[subsc].A?avail;
            m=true;
```

```
            goto con1trg_first1
            };
        ::atomic{
            chan_array[subsc].A?unavail;
            j=true;
            goto wait_td_first_src
            }
        od;

wait_td_first_src:
        atomic{
            chan_array[subsc].A?teardown;
            chan_array[subsc].B!downack;
            if
            ::(callers>0) ->
                chan_array[first].A!teardown;
                goto unlink0trg
            ::else ->
                chan_array[first].B!teardown;
                goto unlink0src
            fi;
             };

    /* Main do loop: LABEL con2src -> first.A?, fisrt.B! caller2=false*/
    ::atomic{    /* CASE 2: first party sends teardown */
        chan_array[first].A?teardown ->
        o=true;
        chan_array[first].B!downack ->
        f=true;
        /*first=callee, so callers no change*/
        goto con1src_second1
        }
    /* Main do loop: LABEL con2src -> first.B?, fisrt.A! caller2=true*/
    ::atomic{    /* CASE 2: first party sends teardown */
        chan_array[first].B?teardown ->
        o=true;
        chan_array[first].A!downack ->
        f=true;
        callers=callers-1;
        goto con1src_second1
        }
    /* Main do loop: LABEL con2src */
    ::atomic{    /* CASE 3: second party sends teardown */
        chan_array[second].B?teardown ->
        o=true;
```

```
        chan_array[second].A!downack ->
        f=true;
        connCW[thisuser-1]=1;
        callers=callers-1;
        goto con1src_first1
        }
    /* Main do loop: LABEL con2src */
    ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
        CW_channel[thisuser-1].intern?extra;
        box_in?setup,orig,number,dest,eval(extra);
        b=true;
        chan_array[extra].A!upack;
        e=true;
        chan_array[extra].A!unavail;
        i=true;
        chan_array[extra].A!teardown;
        g=true;
        goto dn_extra_setup_src
        }
    /*MAIN con2src*/
    od;

dn_extra_setup_src:
    atomic{
        chan_array[extra].B?downack;
        p=true;
        goto con2src
        };

unlink3srcA:
    do
    ::atomic{
        chan_array[first].B?teardown ->
        chan_array[first].A!downack
        }
    ::atomic {
        chan_array[first].B?downack ->
        connCW[thisuser-1]=1;
        if
        ::(callers==3) ->
            /*disconnecting callers first and subsc*/
            callers=callers-2;
        ::else ->
            /*disconnecting caller first, subsc=callee*/
            callers=callers-1;
```

```
        fi;
        calling=false;
        goto setup_sec_src
        }
     od;

unlink3srcB:
    do
    ::atomic{
        chan_array[first].A?teardown ->
        chan_array[first].B!downack
        }
    ::atomic {
        chan_array[first].A?downack ->
        connCW[thisuser-1]=1;
        callers=callers-1; /*disconnecting caller subsc*/
        calling=false;
        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;
        if
        ::(callers>=2 && calling) ->
            /*disconnecting callers second and subsc*/
            callers=callers-2;
        ::(callers>=2 && !calling) ->
            /*disconnecting callers second and subsc*/
            callers=callers-1;
        ::else ->
            /*disconnecting caller second in call back processing*/
            callers=callers-1;
        fi;
        calling=false;
        goto setup_first_src
        }
    od;
```

```
con2trg:
    do
    /* CASE 1: subscriber sends teardown=>CALL BACK PROCESSING */
    ::atomic{
        chan_array[subsc].A?teardown ->
        chan_array[subsc].B!downack;
        };
        connCW[thisuser-1]=1;
        if
        ::talk->    /**connecting to second party, disconnecting first**/
            if
            ::(callers>=2) ->
                chan_array[first].A!teardown;
                goto unlink3trgA
            ::else ->
                chan_array[first].B!teardown;
                goto unlink3trgB
            fi;
        ::else ->    /**connecting to first party, disconnecting second**/
            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;
            subsc=ch;
            }
        do
            /* WAIT_UPACK from subscriber*/
        ::atomic{
            chan_array[subsc].A?upack;
            goto wait_response_second_trg
            }
          ::atomic{
              CW_channel[thisuser-1].intern?new;
            box_in?setup,orig,number,dest,eval(new);
            b=true;
            connCW[thisuser-1]=2;
            }
            if
            /* SITUATION (2) Subscriber calls somebody else [9][10] */
```

```
            ::orig == thisuser ->
                orig2=orig; number2=number; dest2=dest;
                calling=true;
                callers=callers+1;
                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;
                callers=callers+1;
                first=new;
                chan_array[first].A!upack;
                e=true;
                chan_array[first].A!avail;
                h=true;
                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
           ::atomic{
            chan_array[subsc].A?avail;
            m=true;
            goto con1trg_second1;
            }
        ::atomic{
            chan_array[subsc].A?unavail;
            j=true;
            chan_array[subsc].A?teardown;
            chan_array[subsc].B!downack;
            chan_array[second].A!teardown;
            goto unlink2trg
            }
        od;

setup_first_trg:            /**connecting to first party**/
```

```
atomic{
    run routerCWtrg(thisuser);
    C_assign(ch);
    if
    ::orig==thisuser ->  /*REVERSE SETUP */
        box_out!setup,dest,orig,orig,ch
    ::else ->
        box_out!!setup,orig,number,dest,ch
    fi;
    subsc=ch;
    }
do
    /* WAIT_UPACK from subscriber*/
::atomic{
    chan_array[subsc].A?upack;
    goto wait_response_first_trg
    }
  ::atomic{
      CW_channel[thisuser-1].intern?new;
    box_in?setup,orig,number,dest,eval(new);
    b=true;
    connCW[thisuser-1]=2;
    }
    if
    /* SITUATION (2) Subscriber calls somebody else [13][14]*/
    /* SRC scenario -> Caller channels!! */
    ::orig == thisuser ->
        orig2=orig; number2=number; dest2=dest;
        calling=true;
        callers=callers+1;
        subsc=new;
        chan_array[subsc].A!upack;
        C_assign(second);
        /**/
        new=second;
        second=first;
        first=new;
        /**/
        run routerCWsrc(thisuser);
        /* Due to the exchange second=first*/
        box_out!setup,orig,number,dest,first;
        goto unlink_old_subsc_first_trg
    /* SITUATION (3) Somebody else calls subscriber [15][16]*/
    ::else ->
        orig3=orig; number3=number; dest3=dest;
```

```
                calling = false;
                callers=callers+1;
                second=new;
                chan_array[second].A!upack;
                e=true;
                chan_array[second].A!avail;
                h=true;
                goto link_nonsubsc_first
            fi;
         od;

unlink_old_subsc_first_trg: /* SITUATION 2 */
        if
        ::atomic{    /* Subscriber as in trg scenario */
            chan_array[ch].A?upack;
            k=true;
            chan_array[ch].A?unavail;
            j=true;
            chan_array[ch].A?teardown;
            l=true;
            chan_array[ch].B!downack;
            d=true;
            goto wait_upack_first_2trg
            }
         ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_unlink_old_subsc_first_trg
            }
        fi;

dn_unlink_old_subsc_first_trg:
           atomic{
            chan_array[extra].B?downack;
            p=true;
            goto unlink_old_subsc_first_trg
            };
```

```
    /*
     * 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_first1" states.
     */
wait_upack_first_2trg:
        if
        ::atomic{
            chan_array[first].A?upack;
            goto wait_vail_first_2trg
             }
        ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_wait_upack_first_2trg
            }
        fi;

dn_wait_upack_first_2trg:
           atomic{
            chan_array[extra].B?downack;
            p=true;
            goto wait_upack_first_2trg
            };

wait_vail_first_2trg:
        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;
            j=true;
            chan_array[subsc].A!avail;
            h=true;
            chan_array[first].A?teardown;
            l=true;
            chan_array[first].B!downack;
            d=true;
            goto con1trg_second1
            }
        ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
            CW_channel[thisuser-1].intern?extra;
            box_in?setup,orig,number,dest,eval(extra);
            b=true;
            chan_array[extra].A!upack;
            e=true;
            chan_array[extra].A!unavail;
            i=true;
            chan_array[extra].A!teardown;
            g=true;
            goto dn_wait_vail_first_2trg
            }
         fi;

dn_wait_vail_first_2trg:
           atomic{
            chan_array[extra].B?downack;
            p=true;
            goto wait_vail_first_2trg
            };

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

wait_response_first_trg: /*Normal situation:connecting to first party*/
        do
           ::atomic{
            chan_array[subsc].A?avail;
            m=true;
            goto con1trg_first1;
            }
        ::atomic{
            chan_array[subsc].A?unavail;
            j=true;
            chan_array[subsc].A?teardown;
            chan_array[subsc].B!downack;
```

```
            if
            ::(callers==0) ->
                /* first and subsc are callees*/
                chan_array[first].B!teardown;
                goto unlink0src
            ::else ->
                /* first=caller and subsc=callee*/
                chan_array[first].A!teardown;
                goto unlink0trg
            fi;
            }
        od;

    /* Main do loop: LABEL con2trg */
    ::atomic{    /* CASE 2: first party sends teardown, first=caller  */
        chan_array[first].B?teardown ->
        o=true;
        chan_array[first].A!downack ->
        f=true;
        connCW[thisuser-1]=1;
        callers=callers-1;
        goto con1trg_second1
        }
    /* Main do loop: LABEL con2trg */
    ::atomic{    /* CASE 2: first party sends teardown, first=callee */
        chan_array[first].A?teardown ->
        o=true;
        chan_array[first].B!downack ->
        f=true;
        connCW[thisuser-1]=1;
        callers=callers;
        goto con1trg_second1
        }
    /* Main do loop: LABEL con2trg */
    ::atomic{    /* CASE 3: second party sends teardown */
        chan_array[second].B?teardown ->
        o=true;
        chan_array[second].A!downack ->
        f=true;
        connCW[thisuser-1]=1;
        callers=callers-1;
        goto con1trg_first1
        }
    /* Main do loop: LABEL con2trg */
    ::atomic{    /* CASE: extra setup => connCW[thisuser-1]>2 */
```

```
        CW_channel[thisuser-1].intern?extra;
        box_in?setup,orig,number,dest,eval(extra);
        b=true;
        chan_array[extra].A!upack;
        e=true;
        chan_array[extra].A!unavail;
        i=true;
        chan_array[extra].A!teardown;
        g=true;
        goto dn_extra_setup_trg
        }
    /*MAIN con2src*/
    od;

dn_extra_setup_trg:
    atomic{
        chan_array[extra].B?downack;
        p=true;
        goto con2trg
        };

unlink3trgA:
    do
    ::atomic{
        chan_array[first].B?teardown ->
        chan_array[first].A!downack;
        }
    ::atomic {
        chan_array[first].B?downack ->
        connCW[thisuser-1]=1;
        /*one less, since subsc=caller and first=callee*/
        callers=callers-1;
        goto setup_sec_trg
        }
    od;

unlink3trgB:
    do
    ::atomic{
        chan_array[first].A?teardown ->
        chan_array[first].B!downack;
        }
    ::atomic {
        chan_array[first].A?downack ->
        connCW[thisuser-1]=1;
```

```
        /*the same, since subsc and first=callees*/
        callers=callers;
        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]=1;
        /*one less, since subsc=callee and second=caller*/
        callers=callers-1;
        goto setup_first_trg
        }
    od;

end:
    goto endSet_up
}
```

Appendix B

Language Containment Proof

B.1 State Machine for Abstract Models

B.1.1 `ComboPort`$_{\textsf{BOUND}}$

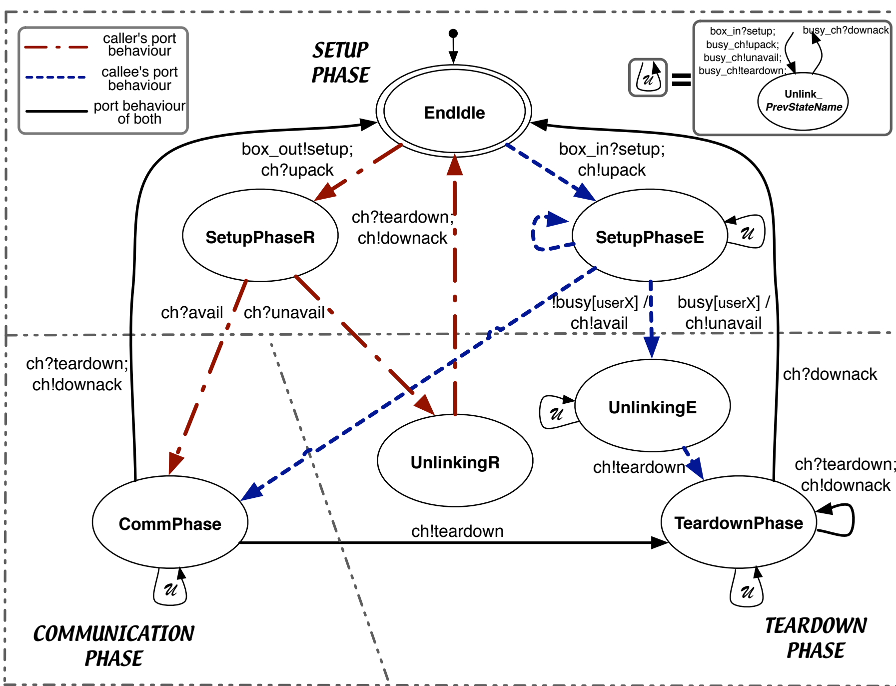

Figure B.1: State Machine for `ComboPort`$_{\textsf{BOUND}}$ Process

B.1.2 ComboPort$_{\mathsf{FREE}}$

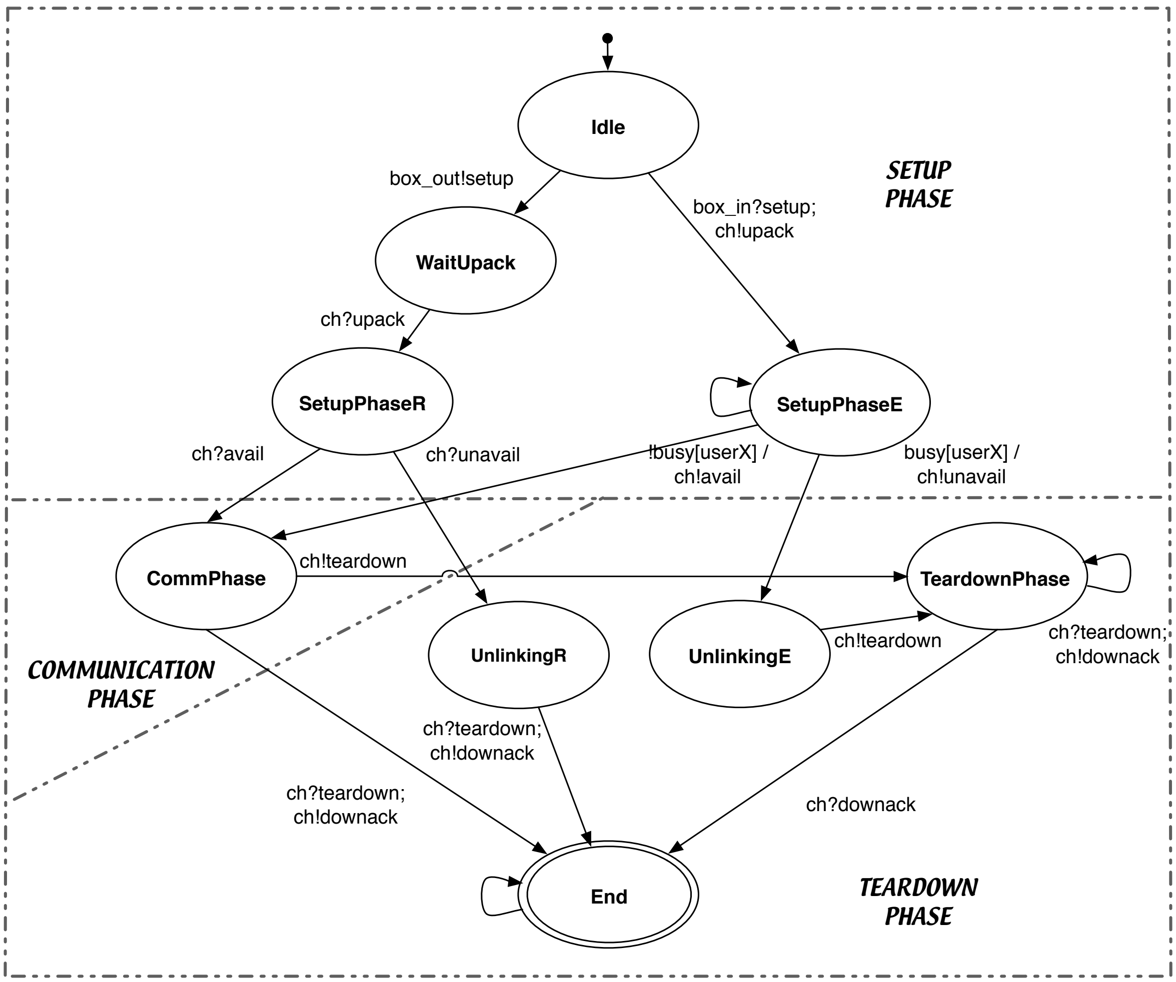

Figure B.2: State Machine for ComboPort$_{\mathsf{FREE}}$ Process

B.1.3 `CallerPort`$_{\text{BOUND}}$

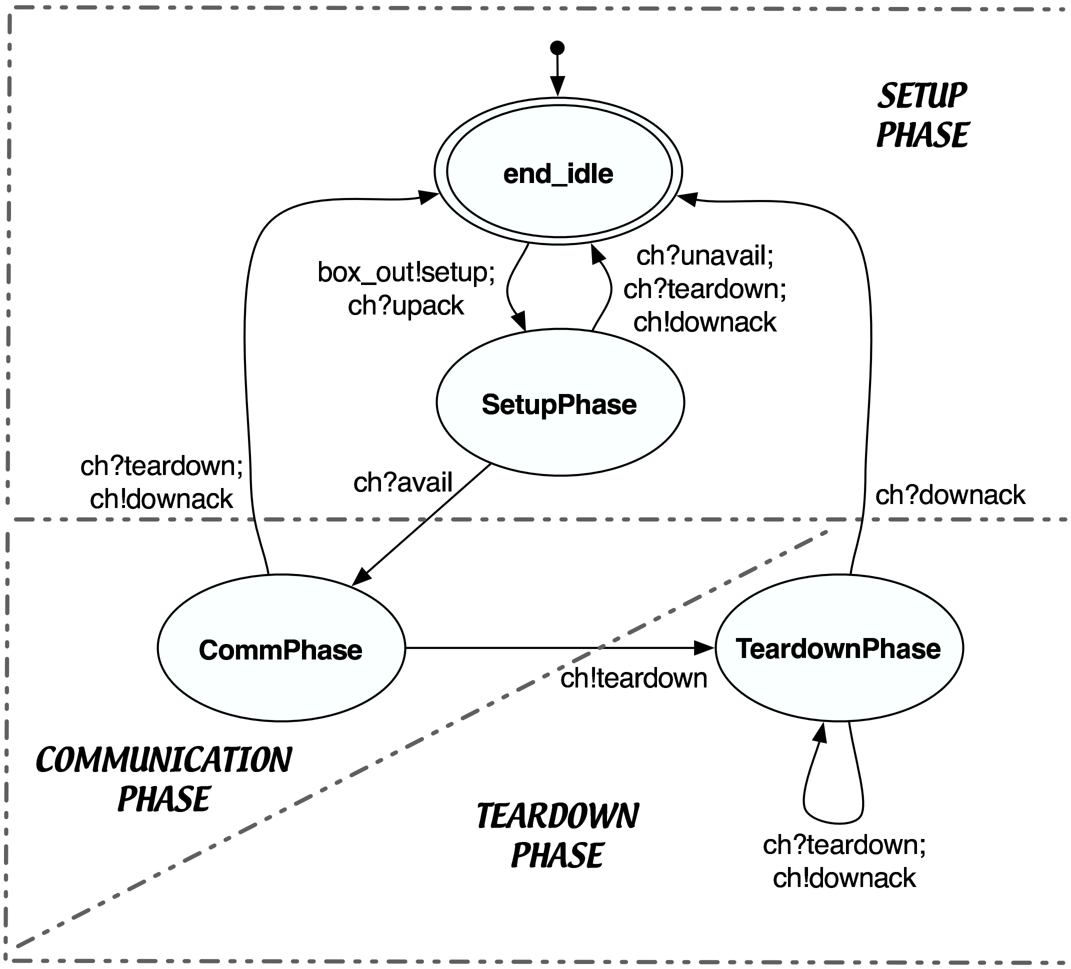

Figure B.3: State Machine for `CallerPort`$_{\text{BOUND}}$ Process

B.1.4 `CalleePort`$_{\text{BOUND}}$

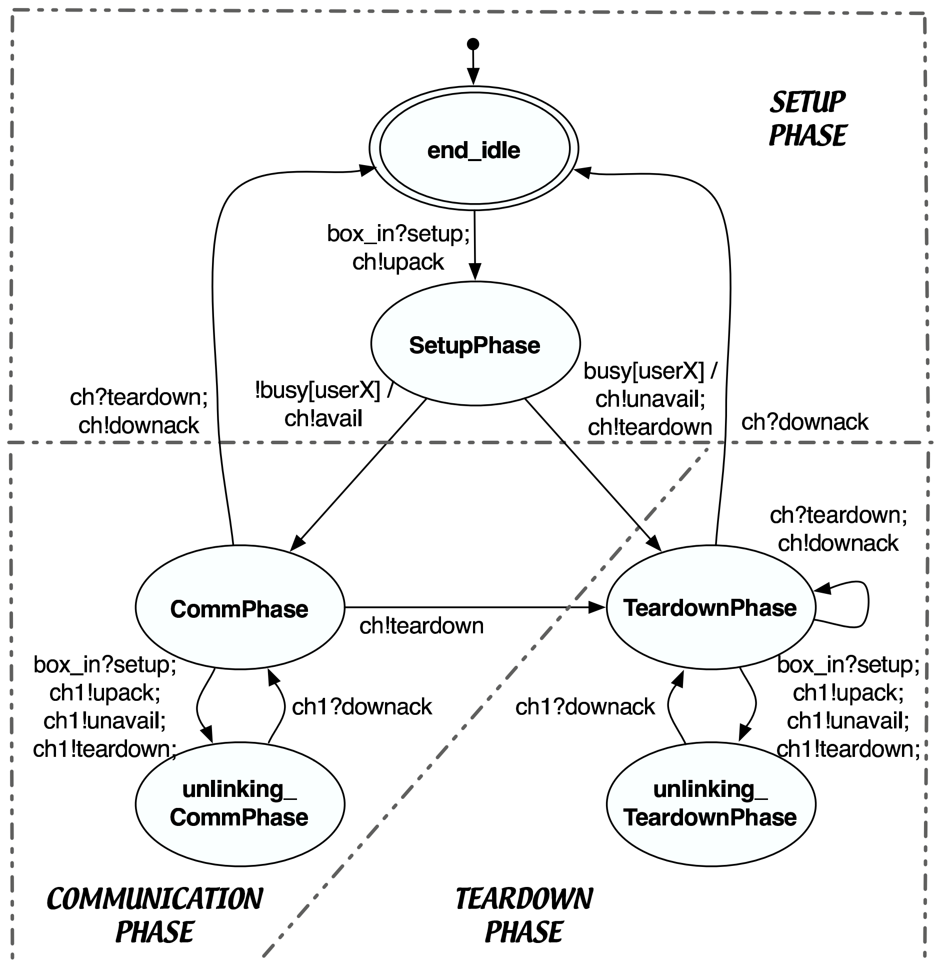

Figure B.4: State Machine for `CalleePort`$_{\text{BOUND}}$ Process

B.1.5 `CallerPort`$_{\mathsf{FREE}}$

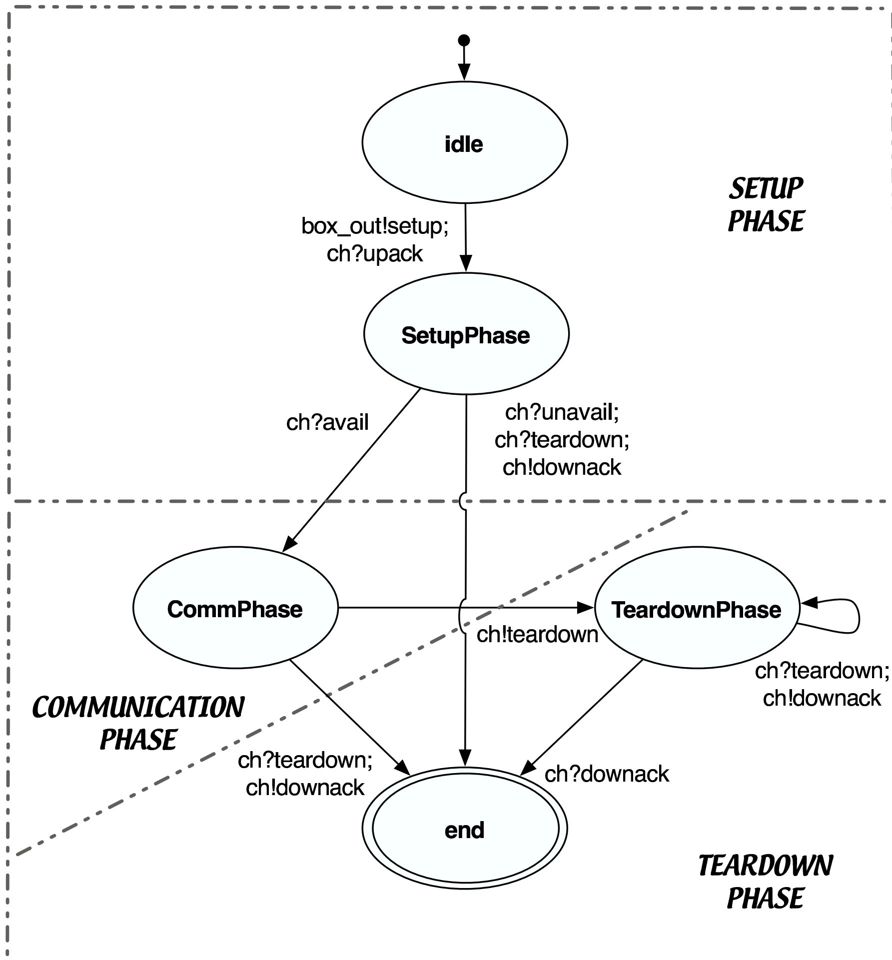

Figure B.5: State Machine for `CallerPort`$_{\mathsf{FREE}}$ Process

B.1.6 CalleePort$_{\mathsf{FREE}}$

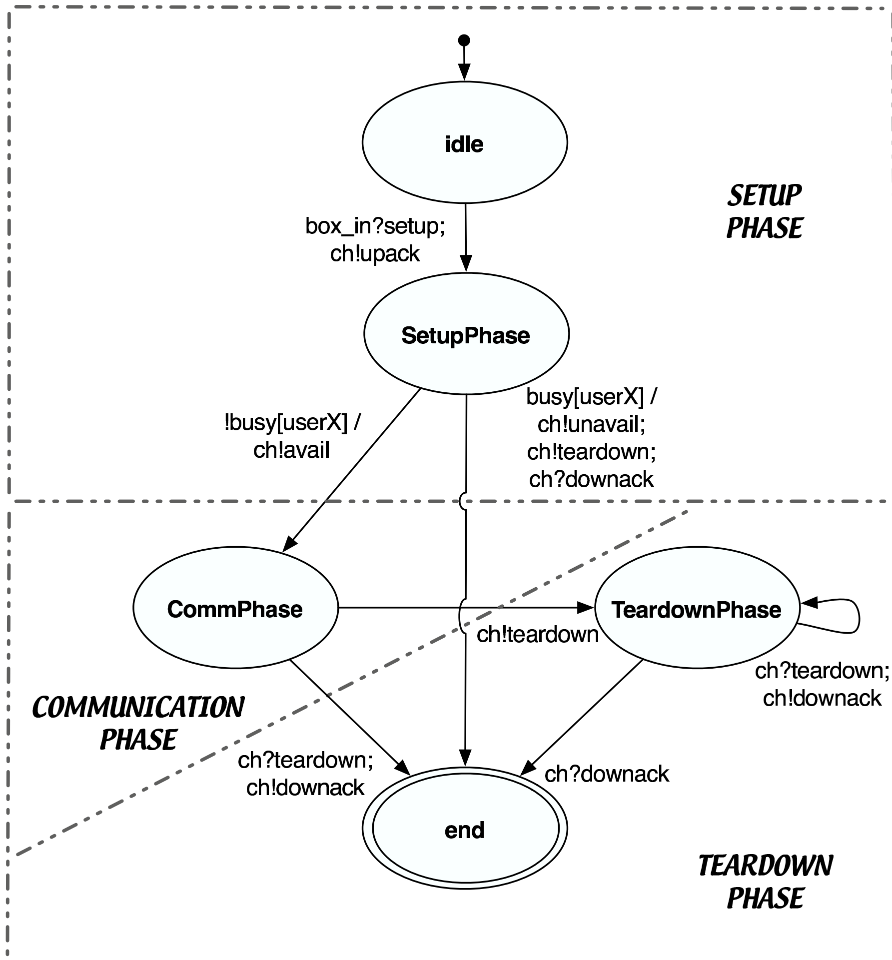

Figure B.6: State Machine for `CallerPort`$_{\mathsf{FREE}}$ Process

B.2 $\mathcal{L}(\texttt{ComboPort})_{\textsf{FREE}}) \subseteq \mathcal{L}(\texttt{ComboPort}_{\textsf{BOUND}})$ (1)-(2)

$$\mathcal{R} = \left\{ \begin{array}{ll} \text{(idle,EndIdle),} & \text{(waitUpack,WaitUpack),} \\ \text{(setupPhaseR,SetupPhaseR),} & \text{(setupPhaseE,SetupPhaseE),} \\ \text{(setupPhaseE,U\_SetupPhaseE),} & \text{(unlinkingR,UnlinkingR),} \\ \text{(unlinkingE,UnlinkingE),} & \text{(unlinkingE,U\_UnlinkingE),} \\ \text{(commPhase,CommPhase),} & \text{(commPhase,U\_CommPhase),} \\ \text{(teardownPhase,TeardownPhase),} & \text{(teardownPhase,U\_TeardownPhase),} \\ \text{(end,EndIdle),} & \end{array} \right\}$$

Source State $\texttt{ComboPort}_{\textsf{FREE}}$ / $\texttt{ComboPort}_{\textsf{BOUND}}$	**Trigger**	**Destination State** $\texttt{ComboPort}_{\textsf{FREE}}$ / $\texttt{ComboPort}_{\textsf{BOUND}}$
idle / EndIdle	box_out ! setup	waitUpack / WaitUpack
idle / EndIdle	box_in ? setup; ch ! upack	setupPhaseE / SetupPhaseE
waitUpack / WaitUpack	ch ? upack	setupPhaseR / SetupPhaseR
setupPhaseR / SetupPhaseR	ch ? avail	commPhase / CommPhase
setupPhaseR / SetupPhaseR	ch ? unavail	unlinkingR / UnlinkingR
setupPhaseE / SetupPhaseE	ch ! avail	commPhase / CommPhase
setupPhaseE / SetupPhaseE	ch ! unavail	unlinkingE / UnlinkingE
setupPhaseE / SetupPhaseE		setupPhaseE / SetupPhaseE
commPhase / CommPhase	ch ? td; ch ! dnack	end / EndIdle
commPhase / CommPhase	ch ! td	teardownPhase / TeardownPhase

Source State `ComboPort`$_{\mathsf{FREE}}$ / `ComboPort`$_{\mathsf{BOUND}}$	**Trigger**	**Destination State** `ComboPort`$_{\mathsf{FREE}}$ / `ComboPort`$_{\mathsf{BOUND}}$
unlinkingR / UnlinkingR	ch ? td; ch ! dnack	end / EndIdle
unlinkingE / UnlinkingE	ch ! td	teardownPhase / TeardownPhase
teardownPhase / TeardownPhase	ch ? td; ch ! dnack	teardownPhase / TeardownPhase
teardownPhase / TeardownPhase	ch ? dnack	end / EndIdle

B.3 $\mathcal{L}(\texttt{CallerPort})_{\textsf{BOUND}}) \subseteq \mathcal{L}(\texttt{ComboPort}_{\textsf{BOUND}})$ (1)-(3)

$$\mathcal{R} = \left\{ \begin{array}{ll} \text{(endIdle,EndIdle)}, & \text{(waitUpack,WaitUpack)}, \\ \text{(setupPhase,SetupPhaseR)}, & \text{(unlinking,UnlinkingR)}, \\ \text{(commPhase,CommPhase)}, & \text{(commPhase,U\_CommPhase)}, \\ \text{(teardownPhase,TeardownPhase)}, & \text{(teardownPhase,U\_TeardownPhase)} \end{array} \right\}$$

Source State $\texttt{CallerPort}_{\textsf{BOUND}}$ / $\texttt{ComboPort}_{\textsf{BOUND}}$	**Trigger**	**Destination State** $\texttt{CallerPort}_{\textsf{BOUND}}$ / $\texttt{ComboPort}_{\textsf{BOUND}}$
endIdle / EndIdle	box_out ! setup	waitUpack / WaitUpack
endIdle / EndIdle		endIdle / EndIdle
waitUpack / WaitUpack	ch ? upack	setupPhase / SetupPhaseR
setupPhase / SetupPhaseR	ch ? avail	commPhase / CommPhase
setupPhase / SetupPhaseR	ch ? unavail	unlinking / UnlinkingR
commPhase / CommPhase	ch ? td; ch ! dnack	endIdle / EndIdle
commPhase / CommPhase	ch ! td	teardownPhase / TeardownPhase
commPhase / CommPhase		commPhase / CommPhase
teardownPhase / TeardownPhase	ch ? td; ch ! dnack	teardownPhase / TeardownPhase
teardownPhase / TeardownPhase	ch ? dnack	endIdle / EndIdle
unlinking / UnlinkingR	ch ? td; ch ! dnack	endIdle / EndIdle

B.4 $\mathcal{L}(\texttt{CalleePort})_{\textsf{BOUND}}) \subseteq \mathcal{L}(\texttt{ComboPort}_{\textsf{BOUND}})$ (1)-(4)

$$\mathcal{R} = \left\{ \begin{array}{ll} \text{(endIdle,EndIdle),} & \text{(setupPhase,SetupPhaseE),} \\ \text{(unlink\_setupPhase,U\_SetupPhaseE),} & \text{(unlinking,UnlinkingE),} \\ \text{(unlink\_unlinking,U\_UnlinkingE),} & \text{(commPhase,CommPhase),} \\ \text{(unlink\_commPhase,U\_CommPhase),} & \text{(teardownPhase,TeardownPhase),} \\ \text{(unlink\_teardownPhase,U\_TeardownPhase),} & \end{array} \right\}$$

Source State `CalleePort`$_{\textsf{BOUND}}$ / `ComboPort`$_{\textsf{BOUND}}$	**Trigger**	**Destination State** `CalleePort`$_{\textsf{BOUND}}$ / `ComboPort`$_{\textsf{BOUND}}$
endIdle / EndIdle	box_in ? setup; ch ! upack	setupPhase / SetupPhaseE
endIdle / EndIdle		endIdle / EndIdle
setupPhase / SetupPhaseE	ch ! avail	commPhase / CommPhase
setupPhase / SetupPhaseE	ch ! unavail	unlinking / UnlinkingE
setupPhase / SetupPhaseE		setupPhase / SetupPhaseE
setupPhase / SetupPhaseE	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	unlink_setupPhase / U_SetupPhaseE
unlink_setupPhase / U_SetupPhaseE	ext ? dnack	setupPhase / SetupPhaseE
unlinking / UnlinkingE	ch ! td	teardownPhase / TeardownPhase
unlinking / UnlinkingE	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	unlink_unlinking / U_UnlinkingE
unlink_unlinking / U_UnlinkingE	ext ? dnack	unlinking / UnlinkingE

Source State `CalleePort`$_{\mathsf{BOUND}}$ / `ComboPort`$_{\mathsf{BOUND}}$	**Trigger**	**Destination State** `CalleePort`$_{\mathsf{BOUND}}$ / `ComboPort`$_{\mathsf{BOUND}}$
commPhase / CommPhase	ch ? td; ch ! dnack	endIdle / EndIdle
commPhase / CommPhase	ch ! td	teardownPhase / TeardownPhase
commPhase / CommPhase		commPhase / CommPhase
commPhase / CommPhase	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	unlink_commPhase / U_CommPhase
unlink_commPhase / U_CommPhase	ext ? dnack	commPhase / CommPhase
teardownPhase / TeardownPhase	ch ? td; ch ! dnack	teardownPhase / TeardownPhase
teardownPhase / TeardownPhase	ch ? dnack	endIdle / EndIdle
teardownPhase / TeardownPhase	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	unlink_teardownPhase / U_TeardownPhase
unlink_teardownPhase / U_TeardownPhase	ext ? dnack	teardownPhase / TeardownPhase

B.5 $\mathcal{L}(\texttt{CallerPort})_{\textsf{FREE}}) \subseteq \mathcal{L}(\texttt{ComboPort}_{\textsf{FREE}})$ (2)-(5)

$$\mathcal{R} = \left\{ \begin{array}{ll} \text{(Idle,Idle)}, & \text{(WaitUpack,WaitUpack)}, \\ \text{(SetupPhase,SetupPhaseR)}, & \text{(CommPhase,CommPhase)}, \\ \text{(TeardownPhase,TeardownPhase)}, & \text{(Unlinking,UnlinkingR)}, \\ \text{(End,End)} & \end{array} \right\}$$

Source State `CallerPort`$_{\textsf{FREE}}$ / `ComboPort`$_{\textsf{FREE}}$	**Trigger**	**Destination State** `CallerPort`$_{\textsf{FREE}}$ / `ComboPort`$_{\textsf{FREE}}$
Idle / Idle	box_out ! setup	WaitUpack / WaitUpack
WaitUpack / WaitUpack	ch ? upack	SetupPhase / SetupPhaseR
SetupPhase / SetupPhaseR	ch ? avail	CommPhase / CommPhase
SetupPhase / SetupPhaseR	ch ? unavail	Unlinking / UnlinkingR
CommPhase / CommPhase	ch ? td; ch ! dnack	End / End
CommPhase / CommPhase	ch ! td	TeardownPhase / TeardownPhase
TeardownPhase / TeardownPhase	ch ? td; ch ! dnack	TeardownPhase / TeardownPhase
TeardownPhase / TeardownPhase	ch ? dnack	End / End
Unlinking / UnlinkingR	ch ? td; ch ! dnack	End / End
End / End		End / End

B.6 $\mathcal{L}(\texttt{CalleePort})_{\textsf{FREE}}) \subseteq \mathcal{L}(\texttt{ComboPort}_{\textsf{FREE}})$ (2)-(6)

$$\mathcal{R} = \left\{ \begin{array}{ll} \text{(Idle,Idle)}, & \text{(SetupPhase,SetupPhaseE)}, \\ \text{(CommPhase,CommPhase)}, & \text{(Unlinking,UnlinkingE)}, \\ \text{(TeardownPhase,TeardownPhase)}, & \text{(End,End)} \end{array} \right\}$$

Source State $\texttt{CalleePort}_{\textsf{FREE}}$ / $\texttt{ComboPort}_{\textsf{FREE}}$	**Trigger**	**Destination State** $\texttt{CalleePort}_{\textsf{FREE}}$ / $\texttt{ComboPort}_{\textsf{FREE}}$
Idle / Idle	box_in ? setup; ch ! upack	SetupPhase / SetupPhaseE
SetupPhase / SetupPhaseE	ch ! avail	CommPhase / CommPhase
SetupPhase / SetupPhaseE	ch ! unavail	Unlinking / UnlinkingE
SetupPhase / SetupPhaseE		SetupPhase / SetupPhaseE
Unlinking / UnlinkingE	ch ! td	TeardownPhase / TeardownPhase
CommPhase / CommPhase	ch ? td; ch ! dnack	End / End
CommPhase / CommPhase	ch ! td	TeardownPhase / TeardownPhase
TeardownPhase / TeardownPhase	ch ? td; ch ! dnack	TeardownPhase / TeardownPhase
TeardownPhase / TeardownPhase	ch ? dnack	End / End
End / End		End / End

B.7 $\mathcal{L}(\texttt{CallerPort})_{\textsf{FREE}}) \subseteq \mathcal{L}(\texttt{CallerPort}_{\textsf{BOUND}})$ (3)-(5)

$$\mathcal{R} = \left\{ \begin{array}{ll} \text{(Idle,EndIdle)}, & \text{(WaitUpack,WaitUpack)}, \\ \text{(SetupPhase,SetupPhase)}, & \text{(CommPhase,CommPhase)}, \\ \text{(TeardownPhase,TeardownPhase)}, & \text{(End,EndIdle)} \end{array} \right\}$$

Source State $\texttt{CallerPort}_{\textsf{FREE}}$ / $\texttt{CallerPort}_{\textsf{BOUND}}$	**Trigger**	**Destination State** $\texttt{CallerPort}_{\textsf{FREE}}$ / $\texttt{CallerPort}_{\textsf{BOUND}}$
Idle / EndIdle	box_out ! setup	WaitUpack / WaitUpack
WaitUpack / WaitUpack	ch ? upack	SetupPhase / SetupPhase
SetupPhase / SetupPhase	ch ? avail	CommPhase / CommPhase
SetupPhase / SetupPhase	ch ? unavail	Unlinking / Unlinking
CommPhase / CommPhase	ch ? td; ch ! dnack	End / EndIdle
CommPhase / CommPhase	ch ! td	TeardownPhase / TeardownPhase
TeardownPhase / TeardownPhase	ch ? td; ch ! dnack	TeardownPhase / TeardownPhase
TeardownPhase / TeardownPhase	ch ? dnack	End / EndIdle
Unlinking / Unlinking	ch ? td; ch ! dnack	End / EndIdle
End / EndIdle		End / EndIdle

B.8 $\mathcal{L}(\texttt{CalleePort})_{\textsf{FREE}}) \subseteq \mathcal{L}(\texttt{CalleePort}_{\textsf{BOUND}})$ (4)-(6)

$$\mathcal{R} = \left\{ \begin{array}{ll} \text{(Idle,EndIdle),} & \text{(SetupPhase,SetupPhase),} \\ \text{(SetupPhase,Unlink\_SetupPhase),} & \text{(CommPhase,CommPhase),} \\ \text{(CommPhase,Unlink\_CommPhase),} & \text{(Unlinking,Unlinking),} \\ \text{(Unlinking,Unlink\_Unlinking),} & \text{(TeardownPhase,TeardownPhase),} \\ \text{(End,EndIdle)} & \end{array} \right\}$$

Source State $\texttt{CalleePort}_{\textsf{FREE}}$ / $\texttt{CalleePort}_{\textsf{BOUND}}$	**Trigger**	**Destination State** $\texttt{CalleePort}_{\textsf{FREE}}$ / $\texttt{CalleePort}_{\textsf{BOUND}}$
Idle / EndIdle	box_in ? setup; ch ! upack	SetupPhase / SetupPhase
SetupPhase / SetupPhase	ch ! avail	CommPhase / CommPhase
SetupPhase / SetupPhase	ch ! unavail	Unlinking / Unlinking
SetupPhase / SetupPhase		SetupPhase / SetupPhase
Unlinking / Unlinking	ch ! td	TeardownPhase / TeardownPhase
CommPhase / CommPhase	ch ? td; ch ! dnack	End / EndIdle
CommPhase / CommPhase	ch ! td	TeardownPhase / TeardownPhase
TeardownPhase / TeardownPhase	ch ? td; ch ! dnack	TeardownPhase / TeardownPhase
TeardownPhase / TeardownPhase	ch ? dnack	End / EndIdle
End / EndIdle		End / EndIdle

B.9 $\mathcal{L}(\mathtt{FTF}_{\mathsf{CallerPort}}) \subseteq \mathcal{L}(\mathtt{CallerPort}_{\mathsf{FREE}})$ (5)-(9)

$$\mathcal{R} = \left\{ \begin{array}{ll} \text{(idle,Idle)}, & \text{(wait\_upack,WaitUpack)}, \\ \text{(wait\_vail,SetupPhase)}, & \text{(connected,CommPhase)}, \\ \text{(unlinking\_from\_ch1,TeardownPhase)} & \text{(wait\_teardown,Unlinking)}, \\ \text{(unlinking\_from\_ch2,End)}, & \text{(end,End)} \end{array} \right\}$$

Source State $\mathtt{FTF}_{\mathsf{CallerPort}}$ / $\mathtt{CallerPort}_{\mathsf{FREE}}$	**Trigger**	**Destination State** $\mathtt{FTF}_{\mathsf{CallerPort}}$ / $\mathtt{CallerPort}_{\mathsf{FREE}}$
idle / Idle	box_out ! setup	wait_upack / WaitUpack
wait_upack / WaitUpack	ch2 ? upack	wait_vail / SetupPhase
wait_vail / SetupPhase	ch2 ? avail	connected / CommPhase
wait_vail / SetupPhase	ch2 ? unavail	wait_teardown / Unlinking
wait_teardown / Unlinking	ch2 ? td; ch2 ! dnack	unlinking_from_ch2 / End
connected / CommPhase	ch2 ! td	unlinking_from_ch1 / TeardownPhase
connected / CommPhase	ch2 ? td; ch2 ! dnack	unlinking_from_ch2 / End
unlinking_from_ch1 / TeardownPhase	ch2 ? td; ch2 ! dnack	unlinking_from_ch1 / TeardownPhase
unlinking_from_ch1 / TeardownPhase	ch2 ? dnack	end / End
unlinking_from_ch2 / End		unlinking_from_ch2 / End
unlinking_from_ch2 / End		end / End

B.10 $\mathcal{L}(\texttt{FTF}_{\textsf{CalleePort}}) \subseteq \mathcal{L}(\texttt{CalleePort}_{\textsf{FREE}})$ (6)-(10)

$$\mathcal{R} = \left\{ \begin{array}{ll} \text{(idle,Idle),} & \text{(wait\_upack,SetupPhase),} \\ \text{(wait\_vail,SetupPhase),} & \text{(connected,CommPhase),} \\ \text{(unlinking\_from\_ch1,End),} & \text{(wait\_teardown,Unlinking),} \\ \text{(unlinking\_from\_ch2,TeardownPhase),} & \text{(end,End),} \end{array} \right\}$$

Source State $\texttt{FTF}_{\textsf{CalleePort}}$ / $\texttt{CalleePort}_{\textsf{FREE}}$	**Trigger**	**Destination State** $\texttt{FTF}_{\textsf{CalleePort}}$ / $\texttt{CalleePort}_{\textsf{FREE}}$
idle / Idle	box_in ? setup; ch1 ! upack	wait_upack / SetupPhase
wait_upack / SetupPhase		wait_vail / SetupPhase
wait_vail / SetupPhase	ch1 ! avail	connected / CommPhase
wait_vail / SetupPhase	ch1 ! unavail	wait_teardown / Unlinking
wait_teardown / Unlinking	ch1 ! td	unlinking_from_ch2 / TeardownPhase
connected / CommPhase	ch1 ? td; ch1 ! dnack	unlinking_from_ch1 / End
connected / CommPhase	ch1 ! td	unlinking_from_ch2 / TeardownPhase
unlinking_from_ch1 / End		unlinking_from_ch1 / End
unlinking_from_ch1 / End		end / End
unlinking_from_ch2 / TeardownPhase	ch1 ? td; ch1 ! dnack	unlinking_from_ch2 / TeardownPhase
unlinking_from_ch2 / TeardownPhase	ch1 ? dnack	end / End

B.11 $\mathcal{L}(\texttt{Caller}_{\textsf{CallerPort}}) \subseteq \mathcal{L}(\texttt{CallerPort}_{\textsf{BOUND}})$ (3)-(7)

$$\mathcal{R} = \left\{ \begin{array}{ll} \text{(end\_idle,EndIdle),} & \text{(wait\_upack,WaitUpack),} \\ \text{(wait\_vail,SetupPhase),} & \text{(end\_linked,CommPhase),} \\ \text{(unlinking,TeardownPhase),} & \text{(wait\_teardown,Unlinking),} \end{array} \right\}$$

Source State $\texttt{Caller}_{\textsf{CallerPort}}$ / $\texttt{CallerPort}_{\textsf{BOUND}}$	**Trigger**	**Destination State** $\texttt{Caller}_{\textsf{CallerPort}}$ / $\texttt{CallerPort}_{\textsf{BOUND}}$
end_idle / EndIdle	box_out ! setup	wait_upack / WaitUpack
wait_upack / WaitUpack	ch ? upack	wait_vail / SetupPhase
wait_vail / SetupPhase	ch ? avail	end_linked / CommPhase
wait_vail / SetupPhase	ch ? unavail	wait_teardown / Unlinking
wait_teardown / Unlinking	ch ? td; ch ! dnack	end_idle / EndIdle
end_linked / CommPhase	ch ? td; ch ! dnack	end_idle / EndIdle
end_linked / CommPhase		end_linked / CommPhase
end_linked / CommPhase	ch ! td	unlinking / TeardownPhase
unlinking / TeardownPhase	ch ? td; ch ! dnack	unlinking / TeardownPhase
unlinking / TeardownPhase	ch ? dnack	end_idle / EndIdle

B.12 $\mathcal{L}(\texttt{Callee}_{\textsf{CalleePort}}) \subseteq \mathcal{L}(\texttt{CalleePort}_{\textsf{BOUND}})$ (4)-(8)

$$\mathcal{R} = \left\{ \begin{array}{ll} \text{(end\_idle,EndIdle),} & \text{(send\_vail,SetupPhase),} \\ \text{(unlink\_send\_vail,Unlink\_SetupPhase),} & \text{(end\_linked,CommPhase),} \\ \text{(unlink\_end\_linked,Unlink\_CommPhase),} & \text{(send\_td,Unlinking),} \\ \text{(unlink\_send\_td,Unlink\_Unlinking),} & \text{(unlinking,TeardownPhase),} \\ \text{(unlink\_unlinking,Unlink\_TeardownPhase)} & \end{array} \right\}$$

Source State $\texttt{Callee}_{\textsf{CalleePort}}$ / $\texttt{CalleePort}_{\textsf{BOUND}}$	**Trigger**	**Destination State** $\texttt{Callee}_{\textsf{CalleePort}}$ / $\texttt{CalleePort}_{\textsf{BOUND}}$
end_idle / EndIdle	box_in ? setup; ch ! upack	send_vail / SetupPhase
send_vail / SetupPhase	ch ! avail	end_linked / CommPhase
send_vail / SetupPhase	ch ! unavail	send_td / Unlinking
send_vail / SetupPhase	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	unlink_send_vail / Unlink_SetupPhase
unlink_send_vail / Unlink_SetupPhase	ext ? dnack	send_vail / SetupPhase
send_td / Unlinking	ch ! td	unlinking / TeardownPhase
send_td / Unlinking	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	unlink_send_td / Unlink_Unlinking
unlink_send_td / Unlink_Unlinking	ext ? dnack	send_td / Unlinking

Source State $\texttt{Callee}_{\mathsf{CalleePort}}$ / $\texttt{CalleePort}_{\mathsf{BOUND}}$	**Trigger**	**Destination State** $\texttt{Callee}_{\mathsf{CalleePort}}$ / $\texttt{CalleePort}_{\mathsf{BOUND}}$
end_linked / CommPhase	ch ? td; ch ! dnack	end_idle / EndIdle
end_linked / CommPhase		end_linked / CommPhase
end_linked / CommPhase	ch ! td	unlinking / TeardownPhase
end_linked / CommPhase	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	unlink_end_linked / Unlink_CommPhase
unlink_end_linked / Unlink_CommPhase	ext ? dnack	end_linked / CommPhase
unlinking / TeardownPhase	ch ? td; ch ! dnack	unlinking / TeardownPhase
unlinking / TeardownPhase	ch ? dnack	end_idle / EndIdle
unlinking / TeardownPhase	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	unlink_unlinking / Unlink_TeardownPhase
unlink_unlinking / Unlink_TeardownPhase	ext ? dnack	unlinking / TeardownPhase

B.13 $\mathcal{L}(\texttt{CW(subsc)}_{\textsf{\textbf{ComboPort}}}) \subseteq \mathcal{L}(\texttt{ComboPort}_{\textsf{\textbf{BOUND}}})$ (1)-(11)

$$\mathcal{R} = \left\{ \begin{array}{ll}
\text{(idle,Idle),} & \text{(wait_upack,SetupPhase),} \\
\text{(end_Setup,EndIdle),} & \text{(wait_upack_src,SetupPhaseE),} \\
\text{(wait_vail_src,SetupPhaseE),} & \text{(wait_dn_src,UnlinkingE),} \\
\text{(con1src_first1,CommPhase),} & \text{(con1src_second1,CommPhase),} \\
\text{(con1src_first2,CommPhase),} & \text{(con1src_second2,CommPhase),} \\
\text{(con2src,CommPhase),} & \text{(dn_extra_setup_src,U_CommPhase),} \\
\text{(unlink0src,EndIdle),} & \text{(unlink1src,TeardownPhase),} \\
\text{(unlink2src,EndIdle),} & \text{(unlink3src,EndIdle),} \\
\text{(unlink4src,EndIdle),} & \text{(setup_first_src,EndIdle),} \\
\text{(wait_vail_first1_src,EndIdle),} & \text{(wait_response_first_src,SetupPhaseR),} \\
\text{(wait_td_first_src,UnlinkingR),} & \text{(ulnk_old_sub_fst_src,SetupPhaseE),} \\
\text{(dn_ulnk_old_sub_fst_src,U_SetupPhaseE),} & \text{(wait_upack_first2_src,SetupPhaseE),} \\
\text{(dn_wait_upack_first2_src,U_SetupPhaseE),} & \text{(wait_vail_first2_src,SetupPhaseE),} \\
\text{(dn_wait_vail_first2_src,U_SetupPhaseE),} & \text{(wait_td_first1_src,CommPhase),} \\
\text{(setup_sec_src,EndIdle),} & \text{(wait_vail_sec1_src,EndIdle),} \\
\text{(wait_response_sec_src,SetupPhaseR),} & \text{(wait_td_sec_src,UnlinkingR),} \\
\text{(ulnk_old_sub_sec,SetupPhaseE),} & \text{(dn_ulnk_old_sub_sec,U_SetupPhaseE),} \\
\text{(wait_upack_sec2,SetupPhaseE),} & \text{(dn_wait_upack_sec2,U_SetupPhaseE),} \\
\text{(wait_vail_sec2,SetupPhaseE),} & \text{(dn_wait_vail_sec2,U_SetupPhaseE),} \\
\text{(wait_td_sec2,CommPhase),} & \text{(link_nonsubsc_first,EndIdle),}
\end{array} \right\}$$

$$\mathcal{R} = \left\{ \begin{array}{ll}
\text{(dn_link_nonsubsc_first,EndIdle),} & \text{(wait_response_first2,SetupPhaseR),} \\
\text{(dn_wait_response_first2,SetupPhaseR),} & \text{(wait_td_first1_trg,UnlinkingR),} \\
\text{(link_nonsubsc_sec,EndIdle),} & \text{(dn_link_nonsubsc_sec,EndIdle),} \\
\text{(wait_response_sec2,SetupPhaseR),} & \text{(dn_wait_response_sec2,SetupPhaseR),} \\
\text{(wait_td_sec1_trg,UnlinkingR),} & \text{(wait_upack_trg,EndIdle),} \\
\text{(wait_vail_trg,SetupPhaseR),} & \text{(wait_dn_trg,UnlinkingR),} \\
\text{(con1trg_first1,CommPhase),} & \text{(con1trg_second1,CommPhase),} \\
\text{(con1trg_first2,CommPhase),} & \text{(con1trg_second2,CommPhase),} \\
\text{(con2trg,CommPhase),} & \text{(dn_extra_setup_trg,U_CommPhase),} \\
\text{(unlink0trg,EndIdle),} & \text{(unlink1trg,TeardownPhase),} \\
\text{(unlink2trg,EndIdle),} & \text{(unlink3trg,EndIdle),} \\
\text{(unlink4trg,EndIdle),} & \text{(setup_first_trg,EndIdle),} \\
\text{(wait_vail_first1_trg,EndIdle),} & \text{(wait_response_first_trg,SetupPhaseR),} \\
\text{(wait_td_first_trg,UnlinkingR),} & \text{(ulnk_old_sub_fst_trg,SetupPhaseE),} \\
\text{(dn_ulnk_old_sub_fst_trg,U_SetupPhaseE),} & \text{(wait_upack_first2_trg,SetupPhaseE),} \\
\text{(dn_wait_upack_first2_trg,U_SetupPhaseE),} & \text{(wait_vail_first2_trg,SetupPhaseE),} \\
\text{(dn_wait_vail_first2_trg,U_SetupPhaseE),} & \text{(wait_td_first2_trg,CommPhase),} \\
\text{(setup_sec_trg,EndIdle),} & \text{(wait_vail_sec_trg,EndIdle),} \\
\text{(wait_response_sec_trg,SetupPhaseR),} & \text{(wait_td_sec_trg,UnlinkingR)}
\end{array} \right\}$$

Source State `CW(subsc)`$_{\text{ComboPort}}$ / `ComboPort`$_{\text{BOUND}}$	**Trigger**	**Destination State** `CW(subsc)`$_{\text{ComboPort}}$ / `ComboPort`$_{\text{BOUND}}$
end_Setup / EndIdle	box_in ? setup; sub ! upack	wait_upack_src / SetupPhaseE
wait_upack_src / SetupPhaseE		wait_vail_src / SetupPhaseE
wait_vail_src / SetupPhaseE	sub ! avail	con1src_first1 / CommPhase
wait_vail_src / SetupPhaseE	sub ! unavail	wait_dn_src / UnlinkingE
wait_dn_src / UnlinkingE	sub ! td	unlink1src / TeardownPhase
con1src_first1 / CommPhase	sub ! td	unlink1src / TeardownPhase
con1src_first1 / CommPhase	sub ? td; sub ! dnack	unlink0src / EndIdle
con1src_first1 / CommPhase		con1src_first2 / CommPhase
con1src_first2 / CommPhase		con2src / CommPhase
unlink0src / EndIdle		unlink0src / EndIdle
unlink0src / EndIdle		end_Setup / EndIdle
unlink1src / TeardownPhase	sub ? td; sub ! dnack	unlink1src / TeardownPhase
unlink1src / TeardownPhase	sub ? dnack	end_Setup / EndIdle
con1src_second1 / CommPhase	sub ! td	unlink1src / TeardownPhase
con1src_second1 / CommPhase	sub ? td; sub ! dnack	unlink2src / EndIdle
con1src_second1 / CommPhase		con1src_second2 / CommPhase
con1src_second2 / CommPhase		con2src / CommPhase
unlink2src / EndIdle		unlink2src / EndIdle

Source State CW(subsc)$_{\mathsf{ComboPort}}$ / ComboPort$_{\mathsf{BOUND}}$	**Trigger**	**Destination State** CW(subsc)$_{\mathsf{ComboPort}}$ / ComboPort$_{\mathsf{BOUND}}$
unlink2src / EndIdle		end_Setup / EndIdle
con2src / CommPhase		con1src_first1 / CommPhase
con2src / CommPhase		con1src_second1 / CommPhase
con2src / CommPhase	sub ? td; sub ! dnack	unlink3src / EndIdle
con2src / CommPhase	sub ? td; sub ! dnack	unlink4src / EndIdle
con2src / CommPhase	sub ? td; sub ! dnack	unlink4trg / EndIdle
con2src / CommPhase	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	dn_extra_setup_src / U_CommPhase
dn_extra_setup_src / U_CommPhase	ext ? dnack	con2src / CommPhase
unlink3src / EndIdle		unlink3src / EndIdle
unlink3src / EndIdle		setup_sec_src / EndIdle
setup_sec_src / EndIdle	box_out ! setup; sub ? upack	wait_response_sec_src / SetupPhaseR
setup_sec_src / EndIdle	box_in ? setup; sub ! upack	ulnk_old_sub_sec / SetupPhaseE
setup_sec_src / EndIdle		wait_vail_sec1_src / EndIdle
wait_vail_sec1_src / EndIdle		link_nonsubsc_sec / EndIdle
wait_response_sec_src / SetupPhaseR	sub ? avail	con1trg_second1 / CommPhase
wait_response_sec_src / SetupPhaseR	sub ? unavail	wait_td_sec_src / UnlinkingR
wait_td_sec_src / UnlinkingR	sub ? td; sub ! dnack	unlink2src / EndIdle

Source State `CW(subsc)`$_{\mathsf{ComboPort}}$ / `ComboPort`$_{\mathsf{BOUND}}$	**Trigger**	**Destination State** `CW(subsc)`$_{\mathsf{ComboPort}}$ / `ComboPort`$_{\mathsf{BOUND}}$
ulnk_old_sub_sec / SetupPhaseE		wait_upack_sec2 / SetupPhaseE
ulnk_old_sub_sec / SetupPhaseE	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	dn_ulnk_old_sub_sec / U_SetupPhaseE
dn_ulnk_old_sub_sec / U_SetupPhaseE	ext ? dnack	ulnk_old_sub_sec / SetupPhaseE
wait_upack_sec2 / SetupPhaseE		wait_vail_sec2 / SetupPhaseE
wait_upack_sec2 / SetupPhaseE	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	dn_wait_upack_sec2 / U_SetupPhaseE
dn_wait_upack_sec2 / U_SetupPhaseE	ext ? dnack	wait_upack_sec2 / SetupPhaseE
wait_vail_sec2 / SetupPhaseE	sub ! avail	con2src / CommPhase
wait_vail_sec2 / SetupPhaseE	sub ! avail	wait_td_sec2 / CommPhase
wait_vail_sec2 / SetupPhaseE	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	dn_wait_vail_sec2 / U_SetupPhaseE
dn_wait_vail_sec2 / U_SetupPhaseE	ext ? dnack	wait_vail_sec2 / SetupPhaseE
wait_td_sec2 / CommPhase		con1src_second1 / CommPhase
unlink4src / EndIdle		unlink4src / EndIdle
unlink4src / EndIdle		setup_first_src / EndIdle
setup_first_src / EndIdle	box_out ! setup; sub ? upack	wait_response_first_src / SetupPhaseR

Source State `CW(subsc)`$_{\mathsf{ComboPort}}$ / `ComboPort`$_{\mathsf{BOUND}}$	**Trigger**	**Destination State** `CW(subsc)`$_{\mathsf{ComboPort}}$ / `ComboPort`$_{\mathsf{BOUND}}$
setup_first_src / EndIdle	box_in ? setup; sub ! upack	ulnk_old_sub_fst_src / SetupPhaseE
setup_first_src / EndIdle		wait_vail_first1_src / EndIdle
wait_vail_first1_src / EndIdle		link_nonsubsc_first / EndIdle
wait_response_first_src / SetupPhaseR	sub ? avail	con1trg_first1 / CommPhase
wait_response_first_src / SetupPhaseR	sub ? unavail	wait_td_first_src / UnlinkingR
wait_td_first_src / UnlinkingR	sub ? td; sub ! dnack	unlink0src / EndIdle
ulnk_old_sub_fst_src / SetupPhaseE		wait_upack_first2_src / SetupPhaseE
ulnk_old_sub_fst_src / SetupPhaseE	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	dn_ulnk_old_sub_fst_src / U_SetupPhaseE
dn_ulnk_old_sub_fst_src / U_SetupPhaseE	ext ? dnack	ulnk_old_sub_fst_src / SetupPhaseE
wait_upack_first2_src / SetupPhaseE		wait_vail_first2_src / SetupPhaseE
wait_upack_first2_src / SetupPhaseE	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	dn_wait_upack_first2_src / U_SetupPhaseE
dn_wait_upack_first2_src / U_SetupPhaseE	ext ? dnack	wait_upack_first2_src / SetupPhaseE
wait_vail_first2_src / SetupPhaseE	sub ! avail	con2src / CommPhase
wait_vail_first2_src / SetupPhaseE	sub ! avail	wait_td_first1_src / CommPhase
wait_vail_first2_src / SetupPhaseE	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	dn_wait_vail_first2_src / U_SetupPhaseE

Source State `CW(subsc)`$_{\text{ComboPort}}$ / `ComboPort`$_{\text{BOUND}}$	**Trigger**	**Destination State** `CW(subsc)`$_{\text{ComboPort}}$ / `ComboPort`$_{\text{BOUND}}$
dn_wait_vail_first2_src / U_SetupPhaseE	ext ? dnack	wait_vail_first2_src / SetupPhaseE
wait_td_first1_src / CommPhase		con1src_first1 / CommPhase
link_nonsubsc_sec / EndIdle	box_out ! setup; sub ? upack	wait_response_sec2 / SetupPhaseR
link_nonsubsc_sec / EndIdle		dn_link_nonsubsc_sec / EndIdle
dn_link_nonsubsc_sec / EndIdle		link_nonsubsc_sec / EndIdle
wait_response_sec2 / SetupPhaseR	sub ? avail	con2trg / CommPhase
wait_response_sec2 / SetupPhaseR	sub ? unavail	wait_td_sec1_trg / UnlinkingR
wait_response_sec2 / SetupPhaseR		dn_wait_response_sec2 / SetupPhaseR
dn_wait_response_sec2 / SetupPhaseR		wait_response_sec2 / SetupPhaseR
wait_td_sec1_trg / UnlinkingR	sub ? td; sub ! dnack	unlink3src / EndIdle
link_nonsubsc_first / EndIdle	box_out ! setup; sub ? upack	wait_response_first2 / SetupPhaseR
link_nonsubsc_first / EndIdle		dn_link_nonsubsc_first / EndIdle
dn_link_nonsubsc_first / EndIdle		link_nonsubsc_first / EndIdle
wait_response_first2 / SetupPhaseR	sub ? avail	con2trg / CommPhase
wait_response_first2 / SetupPhaseR	sub ? unavail	wait_td_first1_trg / UnlinkingR
wait_response_first2 / SetupPhaseR		dn_wait_response_first2 / SetupPhaseR
dn_wait_response_first2 / SetupPhaseR		wait_response_first2 / SetupPhaseR

Source State CW(subsc)$_{\text{ComboPort}}$ / ComboPort$_{\text{BOUND}}$	**Trigger**	**Destination State** CW(subsc)$_{\text{ComboPort}}$ / ComboPort$_{\text{BOUND}}$
wait_td_first1_trg / UnlinkingR	sub ? td; sub ! dnack	unlink4src / EndIdle
end_Setup / EndIdle		wait_upack_trg / EndIdle
wait_upack_trg / EndIdle	box_out ! setup; sub ? upack	wait_vail_trg / SetupPhaseR
wait_vail_trg / SetupPhaseR	sub ? avail	con1trg_first1 / CommPhase
wait_vail_trg / SetupPhaseR	sub ? unavail	wait_dn_trg / UnlinkingR
wait_dn_trg / UnlinkingR	sub ? td; sub ! dnack	unlink0trg / EndIdle
con1trg_first1 / CommPhase	sub ! td	unlink1trg / TeardownPhase
con1trg_first1 / CommPhase	sub ? td; sub ! dnack	unlink0trg / EndIdle
con1trg_first1 / CommPhase		con1trg_first2 / CommPhase
con1trg_first2 / CommPhase		con2trg / CommPhase
unlink0trg / EndIdle		unlink0trg / EndIdle
unlink0trg / EndIdle		end_Setup / EndIdle
unlink1trg / TeardownPhase	sub ? td; sub ! dnack	unlink1trg / TeardownPhase
unlink1trg / TeardownPhase	sub ? dnack	end_Setup / EndIdle
con1trg_second1 / CommPhase	sub ! td	unlink1trg / TeardownPhase
con1trg_second1 / CommPhase	sub ? td; sub ! dnack	unlink2trg / EndIdle
con1trg_second1 / CommPhase		con1trg_second2 / CommPhase
con1trg_second2 / CommPhase		con2trg / CommPhase

Source State `CW(subsc)`$_{\mathsf{ComboPort}}$ / `ComboPort`$_{\mathsf{BOUND}}$	**Trigger**	**Destination State** `CW(subsc)`$_{\mathsf{ComboPort}}$ / `ComboPort`$_{\mathsf{BOUND}}$
unlink2trg / EndIdle		unlink2trg / EndIdle
unlink2trg / EndIdle		end_Setup / EndIdle
con2trg / CommPhase		con1trg_first1 / CommPhase
con2trg / CommPhase		con1trg_second1 / CommPhase
con2trg / CommPhase	sub ? td; sub ! dnack	unlink3trg / EndIdle
con2trg / CommPhase	sub ? td; sub ! dnack	unlink4trg / EndIdle
con2trg / CommPhase	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	dn_extra_setup_trg / U_CommPhase
dn_extra_setup_trg / U_CommPhase	ext ? dnack	con2trg / CommPhase
unlink3trg / EndIdle		unlink3trg / EndIdle
unlink3trg / EndIdle		setup_sec_trg / EndIdle
setup_sec_trg / EndIdle	box_out ! setup; sub ? upack	wait_response_sec_trg / SetupPhaseR
setup_sec_trg / EndIdle	box_in ? setup; sub ! upack	ulnk_old_sub_sec / SetupPhaseE
setup_sec_trg / EndIdle		wait_vail_sec_trg / EndIdle
wait_vail_sec_trg / EndIdle		link_nonsubsc_sec / EndIdle
wait_response_sec_trg / SetupPhaseR	sub ? avail	con1trg_second1 / CommPhase
wait_response_sec_trg / SetupPhaseR	sub ? unavail	wait_td_sec_trg / UnlinkingR
wait_td_sec_trg / UnlinkingR	sub ? td; sub ! dnack	unlink2trg / EndIdle

Source State `CW(subsc)`$_{\text{ComboPort}}$ / `ComboPort`$_{\text{BOUND}}$	**Trigger**	**Destination State** `CW(subsc)`$_{\text{ComboPort}}$ / `ComboPort`$_{\text{BOUND}}$
unlink4trg / EndIdle		unlink4trg / EndIdle
unlink4trg / EndIdle		setup_first_trg / EndIdle
setup_first_trg / EndIdle	box_out ! setup; sub ? upack	wait_response_first_trg / SetupPhaseR
setup_first_trg / EndIdle	box_in ? setup; sub ! upack	ulnk_old_sub_fst_trg / SetupPhaseE
setup_first_trg / EndIdle		wait_vail_first1_trg / EndIdle
wait_vail_first1_trg / EndIdle		link_nonsubsc_first / EndIdle
wait_response_first_trg / SetupPhaseR	sub ? avail	con1trg_first1 / CommPhase
wait_response_first_trg / SetupPhaseR	sub ? unavail	wait_td_first_trg / UnlinkingR
wait_td_first_trg / UnlinkingR	sub ? td; sub ! dnack	unlink0trg / EndIdle
ulnk_old_sub_fst_trg / SetupPhaseE		wait_upack_first2_trg / SetupPhaseE
ulnk_old_sub_fst_trg / SetupPhaseE	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	dn_ulnk_old_sub_fst_trg / U_SetupPhaseE
dn_ulnk_old_sub_fst_trg / U_SetupPhaseE	ext ? dnack	ulnk_old_sub_fst_trg / SetupPhaseE
wait_upack_first2_trg / SetupPhaseE		wait_vail_first2_trg / SetupPhaseE
wait_upack_first2_trg / SetupPhaseE	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	dn_wait_upack_first2_trg / U_SetupPhaseE
dn_wait_upack_first2_trg / U_SetupPhaseE	ext ? dnack	wait_upack_first2_trg / SetupPhaseE
wait_vail_first2_trg / SetupPhaseE	sub ! avail	con2src / CommPhase

Source State `CW(subsc)`$_{\mathsf{ComboPort}}$ / `ComboPort`$_{\mathsf{BOUND}}$	**Trigger**	**Destination State** `CW(subsc)`$_{\mathsf{ComboPort}}$ / `ComboPort`$_{\mathsf{BOUND}}$
wait_vail_first2_trg / SetupPhaseE	sub ! avail	wait_td_first2_trg / CommPhase
wait_vail_first2_trg / SetupPhaseE	box_in ? setup; ext ! upack; ext ! unavail; ext ! td	dn_wait_vail_first2_trg / U_SetupPhaseE
dn_wait_vail_first2_trg / U_SetupPhaseE	ext ? dnack	wait_vail_first2_trg / SetupPhaseE
wait_td_first2_trg / CommPhase		con1trg_first1 / CommPhase

B.14 $\mathcal{L}(\texttt{CW(first)}_{\textsf{ComboPort}}) \subseteq \mathcal{L}(\texttt{ComboPort}_{\textsf{BOUND}})$ (1)-(11)

$$\mathcal{R} = \left\{ \begin{array}{ll}
\text{(end_Setup,EndIdle),} & \text{(wait_upack_src,EndIdle),} \\
\text{(wait_vail_src,SetupPhaseR),} & \text{(wait_dn_src,UnlinkingR),} \\
\text{(con1src_first1,CommPhase),} & \text{(con1src_second1,EndIdle),} \\
\text{(con1src_first2,CommPhase),} & \text{(con1src_second2,SetupPhaseE),} \\
\text{(con2src,CommPhase),} & \text{(dn_extra_setup_src,U_CommPhase),} \\
\text{(unlink0src,TeardownPhase),} & \text{(unlink1src,EndIdle),} \\
\text{(unlink2src,EndIdle),} & \text{(unlink3src,TeardownPhase),} \\
\text{(unlink4src,CommPhase),} & \text{(setup_first_src,CommPhase),} \\
\text{(wait_vail_first1_src,CommPhase),} & \text{(wait_response_first_src,CommPhase),} \\
\text{(wait_td_first_src,CommPhase),} & \text{(ulnk_old_sub_fst_src,CommPhase),} \\
\text{(dn_ulnk_old_sub_fst_src,U_CommPhase),} & \text{(wait_upack_first2_src,CommPhase),} \\
\text{(dn_wait_upack_first2_src,U_CommPhase),} & \text{(wait_vail_first2_src,CommPhase),} \\
\text{(dn_wait_vail_first2_src,U_CommPhase),} & \text{(wait_td_first1_src,CommPhase),} \\
\text{(setup_sec_src,EndIdle),} & \text{(wait_vail_sec1_src,SetupPhaseE),} \\
\text{(wait_response_sec_src,EndIdle),} & \text{(wait_td_sec_src,EndIdle),} \\
\text{(ulnk_old_sub_sec,EndIdle),} & \text{(dn_ulnk_old_sub_sec,EndIdle),} \\
\text{(wait_upack_sec2,EndIdle),} & \text{(dn_wait_upack_sec2,EndIdle),} \\
\text{(wait_vail_sec2,SetupPhaseR),} & \text{(dn_wait_vail_sec2,SetupPhaseR),} \\
\text{(wait_td_sec2,UnlinkingR),} & \text{(link_nonsubsc_first,CommPhase),}
\end{array} \right\}$$

$$\mathcal{R} = \left\{ \begin{array}{ll}
\text{(dn_link_nonsubsc_first,U_CommPhase),} & \text{(wait_response_first2,CommPhase),} \\
\text{(dn_wait_response_first2,U_CommPhase),} & \text{(wait_td_first1_trg,CommPhase),} \\
\text{(link_nonsubsc_sec,CommPhase),} & \text{(dn_link_nonsubsc_sec,U_CommPhase),} \\
\text{(wait_response_sec2,CommPhase),} & \text{(dn_wait_response_sec2,U_CommPhase),} \\
\text{(wait_td_sec1_trg,CommPhase),} & \text{(wait_upack_trg,SetupPhaseE),} \\
\text{(wait_vail_trg,SetupPhaseE),} & \text{(wait_dn_trg,UnlinkingE),} \\
\text{(con1trg_first1,CommPhase),} & \text{(con1trg_second1,EndIdle),} \\
\text{(con1trg_first2,CommPhase),} & \text{(con1trg_second2,SetupPhaseE),} \\
\text{(con2trg,CommPhase),} & \text{(dn_extra_setup_trg,U_CommPhase),} \\
\text{(unlink0trg,TeardownPhase),} & \text{(unlink1trg,EndIdle),} \\
\text{(unlink2trg,EndIdle),} & \text{(unlink3trg,TeardownPhase),} \\
\text{(unlink4trg,CommPhase),} & \text{(setup_first_trg,CommPhase),} \\
\text{(wait_vail_first1_trg,CommPhase),} & \text{(wait_response_first_trg,CommPhase),} \\
\text{(wait_td_first_trg,CommPhase),} & \text{(ulnk_old_sub_fst_trg,CommPhase),} \\
\text{(dn_ulnk_old_sub_fst_trg,U_CommPhase),} & \text{(wait_upack_first2_trg,CommPhase),} \\
\text{(dn_wait_upack_first2_trg,U_CommPhase),} & \text{(wait_vail_first2_trg,CommPhase),} \\
\text{(dn_wait_vail_first2_trg,U_CommPhase),} & \text{(wait_td_first2_trg,CommPhase),} \\
\text{(setup_sec_trg,EndIdle),} & \text{(wait_vail_sec_trg,SetupPhaseE),} \\
\text{(wait_response_sec_trg,EndIdle),} & \text{(wait_td_sec_trg,EndIdle)}
\end{array} \right\}$$

Similar to Section B.13.

B.15 $\mathcal{L}(\texttt{CW(second)}_{\textsf{ComboPort}}) \subseteq \mathcal{L}(\texttt{ComboPort}_{\textsf{BOUND}})$ (1)-(11)

$$
\mathcal{R} = \left\{ \begin{array}{ll}
\text{(end\_Setup,EndIdle),} & \text{(wait\_upack\_src,EndIdle),} \\
\text{(wait\_vail\_src,EndIdle),} & \text{(wait\_dn\_src,EndIdle),} \\
\text{(con1src\_first1,EndIdle),} & \text{(con1src\_second1,CommPhase),} \\
\text{(con1src\_first2,SetupPhaseE),} & \text{(con1src\_second2,CommPhase),} \\
\text{(con2src,CommPhase),} & \text{(dn\_extra\_setup\_src,U\_CommPhase),} \\
\text{(unlink0src,EndIdle),} & \text{(unlink1src,EndIdle),} \\
\text{(unlink2src,TeardownPhase),} & \text{(unlink3src,CommPhase),} \\
\text{(unlink4src,TeardownPhase),} & \text{(setup\_first\_src,EndIdle),} \\
\text{(wait\_vail\_first1\_src,SetupPhaseE),} & \text{(wait\_response\_first\_src,EndIdle),} \\
\text{(wait\_td\_first\_src,EndIdle),} & \text{(ulnk\_old\_sub\_fst\_src,EndIdle),} \\
\text{(dn\_ulnk\_old\_sub\_fst\_src,EndIdle),} & \text{(wait\_upack\_first2\_src,EndIdle),} \\
\text{(dn\_wait\_upack\_first2\_src,EndIdle),} & \text{(wait\_vail\_first2\_src,SetupPhaseR),} \\
\text{(dn\_wait\_vail\_first2\_src,SetupPhaseR),} & \text{(wait\_td\_first1\_src,UnlinkingR),} \\
\text{(setup\_sec\_src,CommPhase),} & \text{(wait\_vail\_sec1\_src,CommPhase),} \\
\text{(wait\_response\_sec\_src,CommPhase),} & \text{(wait\_td\_sec\_src,CommPhase),} \\
\text{(ulnk\_old\_sub\_sec,CommPhase),} & \text{(dn\_ulnk\_old\_sub\_sec,U\_CommPhase),} \\
\text{(wait\_upack\_sec2,CommPhase),} & \text{(dn\_wait\_upack\_sec2,U\_CommPhase),} \\
\text{(wait\_vail\_sec2,CommPhase),} & \text{(dn\_wait\_vail\_sec2,U\_CommPhase),} \\
\text{(wait\_td\_sec2,CommPhase),} & \text{(link\_nonsubsc\_first,CommPhase),}
\end{array} \right\}
$$

$$\mathcal{R} = \left\{ \begin{array}{ll}
\text{(dn_link_nonsubsc_first,U_CommPhase),} & \text{(wait_response_first2,CommPhase),} \\
\text{(dn_wait_response_first2,U_CommPhase),} & \text{(wait_td_first1_trg,CommPhase),} \\
\text{(link_nonsubsc_sec,CommPhase),} & \text{(dn_link_nonsubsc_sec,U_CommPhase),} \\
\text{(wait_response_sec2,CommPhase),} & \text{(dn_wait_response_sec2,U_CommPhase),} \\
\text{(wait_td_sec1_trg,CommPhase),} & \text{(wait_upack_trg,EndIdle),} \\
\text{(wait_vail_trg,EndIdle),} & \text{(wait_dn_trg,EndIdle),} \\
\text{(con1trg_first1,EndIdle),} & \text{(con1trg_second1,CommPhase),} \\
\text{(con1trg_first2,SetupPhaseE),} & \text{(con1trg_second2,CommPhase),} \\
\text{(con2trg,CommPhase),} & \text{(dn_extra_setup_trg,U_CommPhase),} \\
\text{(unlink0trg,EndIdle),} & \text{(unlink1trg,EndIdle),} \\
\text{(unlink2trg,TeardownPhase),} & \text{(unlink3trg,CommPhase),} \\
\text{(unlink4trg,TeardownPhase),} & \text{(setup_first_trg,EndIdle),} \\
\text{(wait_vail_first1_trg,SetupPhaseE),} & \text{(wait_response_first_trg,EndIdle),} \\
\text{(wait_td_first_trg,EndIdle),} & \text{(ulnk_old_sub_fst_trg,EndIdle),} \\
\text{(dn_ulnk_old_sub_fst_trg,EndIdle),} & \text{(wait_upack_first2_trg,EndIdle),} \\
\text{(dn_wait_upack_first2_trg,EndIdle),} & \text{(wait_vail_first2_trg,SetupPhaseR),} \\
\text{(dn_wait_vail_first2_trg,SetupPhaseR),} & \text{(wait_td_first2_trg,UnlinkingR),} \\
\text{(setup_sec_trg,CommPhase),} & \text{(wait_vail_sec_trg,CommPhase),} \\
\text{(wait_response_sec_trg,CommPhase),} & \text{(wait_td_sec_trg,CommPhase)}
\end{array} \right\}$$

Similar to Section B.13.

Appendix C

Glossary of Terms

Address Defined by syntactic restrictions on a sequence of symbols from the addressing alphabet.

Bound feature box Process that is dedicated to a particular address, and even if it is already in use within an existing usage, the same feature box is made part of the new usage. An example of a bound feature box is call waiting, which is a persistent process, and can be part of the source or target region of a usage.

Box Process that performs either feature or interface functions.

Call/Communication channel Channel between ports of two different boxes, transmitting signals between boxes in first in–first out (FIFO) order and following the DFC call protocol.

Feature Function for the users of a system, performed on top of basic services. An example of a feature is call waiting.

Free feature box Process for which a new instance is generated every time the feature is to be included in a usage. An example of a free feature box is call forwarding, which is not persistent and gets created upon request.

Interface box Process that provide an interface to physical devices to communicate to users or to other networks. An example of an interface box is a Caller or Callee.

Operational data Data relations that can be read and written by feature boxes.

Port Interaction that a box has with a communication channel connected to another box. The end of the channel connected to the box that initiates a call by sending a `setup` is called a *caller port*, and the port on the other end of the communication channel is called a *callee port*.

Precedences Relations that are partial orders constraining the order in which feature boxes are placed in the source and target region.

Router Process that helps in the generation of the usage, setting up the communication channels between boxes.

Signal Message of the DFC protocol. It has a signal type and some set of named, typed fields. Primary signals: `setup`, `upack`, `teardown`, `downack`. Status signals: `avail`, `unavail`, `unknown`, `none`.

Source region Feature boxes that the caller customer (at the source address) is subscribed to.

Subscriptions Mapping from the address in use to feature box types.

Target region Feature boxes that the called customer (at the target address) is subscribed to.

Usage Graph composed by boxes and calls describing the response of a request for a telecommunication service at certain time.

Bibliography

[1] Rajeev Alur, Thomas A. Henzinger, F.Y.C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. Mocha: Modularity in model checking. In *CAV*, volume 1427 of *LNCS*, pages 521–525. Springer, 1998.

[2] D. Amyot and L. Logrippo, editors. *International Workshop on Feature Interactions.* IOS Press, 2003.

[3] Greg Bond, Franjo Ivancić, Nils Klarlund, and Richard Trefler. ECLIPSE Feature Logic Analysis. *2nd International IP-Tel*, 2001.

[4] Muffy Calder and Alice Miller. Generalising Feature Interactions in Email. In Daniel Amyot and Luigi Logrippo, editors, *Feature Interactions in Telecommunications and Software Systems VII*, pages 187–204. IOS Press, 2003.

[5] E. W. Dijkstra. Notes on structured programming. *Structured Programming*, (70-WSK-03):1–84, April 1970.

[6] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[7] Robert J. Hall. Feature interactions in electronic mail. In Muffy H. Calder and Evan H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 67–82, Amsterdam, Netherlands, May 2000. IOS Press.

[8] Jonathan D. Hay and Joanne M. Atlee. Composing features and resolving interactions. In David S. Rosenblum, editor, *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-00)*, pages 110–119, NY, November 2000. ACM Software Engineering Notes, ACM Press.

[9] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, first edition, 2003.

[10] Gerard J. Holzmann. ON-THE-FLY, LTL MODEL CHECKING with SPIN, http://spinroot.com/spin/whatispin.html, October, 2004.

[11] Daniel Jackson. Abstract model checking of infinite specifications. In *Second International Symposium of Formal Methods Europe on Industrial Benefit of Formal Methods*, volume 873 of *LNCS*, pages 519 – 531. Springer-Verlag, 1994.

[12] Michael Jackson and Pamela Zave. Distributed Feature Composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering XXIV(10)*, pages 831–847, August 1998.

[13] Michael Jackson and Pamela Zave. *The DFC Manual.* AT&T Labs, November 2003.

[14] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, fourth edition, 1999.

[15] K.Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *Jour. of the ACM*, 49(4):538–576, July 2002.

[16] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[17] Kenneth L. McMillan. Verification of infinite state systems by compositional model checking. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, number 1703 in LNCS, pages 219 – 234, 1999.

[18] Wolfgang Thomas. *Automata on infinite objects.* MIT Press, 1991.

[19] Pierre Wolper and Moshe Y. Vardi. Reasoning about infinite computation paths. *24th Symposium on Foundations of Computer Science*, pages 185–194, 1983.

[20] Pamela Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, 26(8):20–29, August 1993.

[21] Pamela Zave. 'Calls considered harmful' and other observations: a tutorial on telephony. In *Services and Visualization: Towards User-Friendly Design: ACoS'98, VISUAL'98, AIN'97. Selected Papers*, volume 1385 of *LNCS*. Springer, 1998.

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

[23] Pamela Zave. Feature-oriented description, formal methods, and DFC. In Stephen Gilmore and Mark Ryan, editors, *Language Constructs for Describing Features*, pages 11–26. Springer-Verlag, 2000/2001.

[24] Pamela Zave. An experiment in Feature Engineering. In Annabelle McIver and Carroll Morgan, editors, *Monographs In Computer Science, Programming Methodology*, pages 353–377. Springer-Verlag, 2003.

[25] Pamela Zave. Ideal Connection Paths in DFC. Technical report, AT&T Laboratories–Research, November 2003.

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

[27] Pamela Zave. Personal communication, July, 2004.

[28] Pamela Zave and Michael Jackson. DFC modifications II: Protocol extensions. Technical report, AT&T Laboratories–Research, November 1999.

[29] Pamela Zave and Michael Jackson. DFC modifications I: Routing extensions. Technical report, AT&T Laboratories–Research, May 2000.

[30] Pamela Zave and Michael 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.

[31] P. Ann Zimmer and Joanne M. Atlee. Categorizing and prioritizing telephony features. In *Feature Interactions in Telecommunications and Software Systems*, 2005. To appear.