Fully Automated Translation of BoxTalk to Promela

by

Tejas Kajarekar

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, 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Telecommunication systems are structured to enable incremental growth, so that new telecommunication features can be added to the set of existing features. With the addition of more features, certain existing features may exhibit unpredictable behaviour. This is known as the *feature interaction problem*, and it is very old problem in telecommunication systems. Jackson and Zave have proposed a technology, Distributed Feature Composition (DFC) to manage the feature interaction problem. DFC is a pipe-and-filter-like architecture where features are "filters" and communication channels connecting features are "pipes".

DFC does not prescribe how features are specified or programmed. Instead, Zave and Jackson have developed BoxTalk, a call-abstraction, domain-specific, high-level programming language for programming features. BoxTalk is based on the DFC protocol and it uses macros to combine common sequences of read and write actions, thus simplifying the details of the DFC protocol in feature models. BoxTalk features must adhere to the DFC protocol in order to be plugged into a DFC architecture (i.e., features must be "DFC compliant"). We want to use model checking to check whether a feature is DFC compliant. We express DFC compliance using a set of properties expressed as linear temporal logic formulas.

To use the model checker SPIN, BoxTalk features must be translated into Promela. Our automatic verification process comprises three steps:

- Explicate BoxTalk features by expanding macros and introducing implicit details.
- Mechanically translate explicated BoxTalk features into Promela models.
- Verify the Promela models of features using the SPIN model checker.

We present a case study of BoxTalk features, describing the original features and how they are explicated and translated into Promela by our software, and how they are proven to be DFC compliant.

Acknowledgements

I would like to thank my supervisor, Professor Joanne M. Atlee for her guidance and unquantifiable help, while working on this problem, and also while writing this thesis.

Many thanks to my committee members, Professor Richard Trefler, and Professor Nancy A. Day, for taking the time to read my thesis and provide valuable comments that helped me improve my thesis.

Dedication

Dedicated to my parents Shree and Jayu, for their support during last couple of years, and to my niece Shravani, and nephew Neil.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Telecommunication systems are structured to enable incremental growth so that new telecommunication features can be added to the set of existing features. With the addition of more features, certain existing features may exhibit unpredictable behaviour, such that the actual behaviours of features become inconsistent with their specified behaviours. This is known as the *feature interaction problem*, and it is very old problem in telecommunication systems.

Let us look at an example of the feature interaction problem where a customer has subscribed to Call Waiting (CW) and Call Forward on No Answer (CFNA) features. CW alerts its subscribers with a special tone when they are called while they are already on the phone. CFNA redirects an incoming call to another phone number if the subscriber does not answer the incoming call within a set number of rings. When the customer is involved in one phone call and receives another call, should the customer hear a special CW notification or should the call be forwarded to another phone number? To complicate the example further, let us assume that another feature, Answer Call, is also present. Answer Call is similar to CFNA, except that the Answer Call feature allows the calling party to leave a message in an answering device when the subscriber of Answer Call does not answer the phone after a set number of rings. Now when an incoming call is not answered, should the call be connected to the answering machine, or should the call be forwarded to another phone number; if the subscriber is already involved in another phone call, should the subscriber hear a special CW tone indicating the presence of the new incoming call?

The feature interaction problem is an old problem in telecommunication systems and it becomes very complex as more and more features, mostly call-processing features, are added to the system. It is very difficult to figure out whether the addition of a new feature will affect the existing ones. To redesign the existing features every time a new feature is introduced is not a viable option.

Jackson and Zave have proposed a technology called *Distributed Feature Composition* (DFC) to manage the feature interaction problem [9]. DFC has a pipe-and-filter-like architecture [13]. In DFC, features act as filters and internal calls act as pipes that connect the features. An **internal call** is a point-to-point, featureless connection obeying a fixed protocol. Internal calls allow the transmission of signals and media in both directions between the feature's endpoints. Features can place, receive, or tear down internal calls to other features. Each feature is independent of other features and does not share state. This independence makes the addition, deletion, or modification of features simple. These characteristics add to the power of the DFC architecture to manage the feature interaction problem.

DFC does not prescribe how features are modeled or programmed. Zave and Jackson have developed *BoxTalk*, a call-abstraction, domain-specific, high-level programming language which is used to program DFC features [15]. BoxTalk is based on the DFC protocol; however, it abstracts away common behaviour that is present in all DFC features. Abstracted behaviour is represented as BoxTalk macros and other implicit behaviour. That some behaviour is implicit adds mild complexity to the understanding of BoxTalk models; however, BoxTalk programmers do not have to program the redundant behaviour for themselves. A more thorough discussion of DFC and BoxTalk will follow in the next chapter.

1.2 Related Work

In this section, we briefly discuss related works of modelling and verifying DFC and BoxTalk features.

Gregory W. Bond et al. [2] developed ECLIPSE, a virtual telecommunications network based on IP, at AT&T Labs. ECLIPSE Statechart, a customized version of Unified Modelling Language (UML) Statechart behaviour description language, was developed to define behaviour of individual feature boxes. A feature communicates with its environment via ports and the feature's Statechart defines how the feature reacts to the messages it receives on its ports. The researchers at AT&T used model checking to check that an ECLIPSE feature obeys the communication protocols, e.g., acknowledges all requests to establish or tear down communication channels. They used the model checker Mocha [1] and developed a translator for translating ECLIPSE features automatically into the input language of Mocha.

Zave used Promela and Z to provide a full formal description of the service layer of a telecommunication system organized according to the DFC virtual architecture [14]. The DFC protocol was described using Promela [7], and the routing algorithm and routing data were described using Z. The two descriptions were coordinated together to describe a telecommunication system. The model checker SPIN was used to check that the protocols of the virtual network never deadlock.

Alma L. Juarez Dominguez described a compositional reasoning method consisting of model checking, language containment, and theorem proving to verify DFC compliance properties over chains of an unknown number of connected DFC features [4, 5]. DFC compliance was defined with respect to the call protocol using a set of LTL properties and, similar to our work, the values of signals sent or received were defined using global Boolean variables. Using the model checker SPIN, she checked that DFC signals received by a feature are propagated to the next feature in the call. She also checked that a feature receives only those signals from its environment which it expects and that it sends only those signals expected by the environment. Instead of verifying that every feature works within the environment of every other feature, she developed an abstract port model (which served as an environment) that captured the most general port behaviour and proved that every feature's ports obey the abstract port model. Abstract and concrete port models are described in terms of transitions consisting of a source state, a trigger (which receives or sends a signal), and a destination state. The abstract model consists of a *caller* port (one that places a call), a *callee* port (one that receives a call), a *combo* port (i.e., a port that can switch between caller and callee), and their *free* and *bound*[1] instances are arranged in a partial order based on language containment. Also using language containment, the behaviour of each port in a feature was proved to be within the behaviour of one of the abstract ports. The properties proved were proved for individual features. As a final step of compositional reasoning, the theorem prover HOL was used to connect the individual proofs by induction to prove that the DFC call protocol properties hold over segments of (unknown number of) connected DFC features.

Zarrin Langari and Richard Trefler proposed a visual semantic modelling approach using Graph Transformation Systems (GTS) to describe the dynamic behaviour of distributed communication protocols such as DFC [10]. They modelled each state of the system as a

[1]A new instance of a free feature is instantiated every time the feature is included in a call. There is only one instance of a bound feature per subscriber which is included in every call including the subscriber.

graph, and used GTS to show the evolving nature of the system. They used a three-level hierarchical model to describe the behaviour of DFC. The first level shows the functionality of each telephony feature as a Finite State Machine (FSM) graph. The second level shows the composition of telephony features and interactions among telephony features through communication channels. The third level shows the dynamic evolution of the topology of the telephony system. The system's state may change when features are added or deleted (thereby changing the topology), or when the state of an existing features changes. The third level graphs allow to analyze partial connections (telephone calls in DFC), without focusing on other distributed processes that are not directly involved in the call. This is advantageous in dealing with rather large communication protocols. In [11], they apply GTS modelling to verify invariant system properties of connection-oriented services such as DFC. They showed that the invariant system properties can be verified by analyzing finite set of transformation rules describing the GTS system model. If the property is satisfied by the initial state of the GTS model and all transformation rules are property preserving, then the property is satisfied by the GTS system. The transformation rule is said to be property preserving if it does not transform the system graph in a way that violates the property.

Naghmeh Ghafari and Richard Trefler presented an automated method for analyzing properties of piecewise (first-in-first-out) FIFO systems that communicate via unbounded channels [6]. Such systems can be used for modelling distributed protocols such as IP-telecommunication protocols and interacting web services. They present a procedure for building an abridged model of the FIFO system, which is an abstraction of reachable channel contents. BoxOS (ECLIPSE) is a virtual telecommunication network based on IP, developed at AT&T. They apply their procedure to BoxOS to check safety properties and end-to-end (path) properties, eg. a message sent from one end will eventually reach to the other end.

We are interested in automatic verification of BoxTalk features, which are more abstract than ECLIPSE features. As a first step, we concretize all of the abstractions in a BoxTalk feature model. That is, we expand all of the macros used by a feature and introduce other details that are implicit in the BoxTalk models. This concretization which we call *explication*, is necessary because the properties to be verified often refer to details that are abstracted away in BoxTalk features. We have developed a program to explicate BoxTalk features automatically, using the explication rules developed by Yuan Peng [12]. We use the model checker **SPIN** [7] to verify our explicated BoxTalk features. The input language of SPIN is **Promela**, and hence we translate our explicated BoxTalk features into Promela models. Since the explication process concretizes BoxTalk macros that are based on the DFC protocol, we test our program (explicator + translator) by checking output Promela models against DFC-compliance properties. These properties are not very interesting towards feature verification as BoxTalk macros handle DFC compliance. However, proving these DFC-compliance properties helps to demonstrate that our translator correctly explicates the BoxTalk features and that the resulting Promela models are ready to be model checked. The ultimate goal of this research is to prove arbitrary properties of one or more (combinations) of features (e.g., feature interaction), but this is beyond scope of this thesis.

Yuan Peng, in her Master's thesis [12] performed manual explication of BoxTalk features and hand translated those explicated features into Promela models. The resulting models were verified with the SPIN model checker against DFC-compliance properties. We automated the process of generating Promela models of BoxTalk features. Similar to the structure of Yuan Peng's models, our Promela models are expressed in terms of constructs that are common to multiple modelling languages (i.e., states, transitions, event queues, variables, and changes to all). The resulting Promela models will help understand how to structure a Promela model in terms of these constructs and will aid in the translations of languages or of general templates into Promela in future.

1.3 Contributions of our Work

Our translation process can be summarized in three steps as follows:

1. We parse a BoxTalk feature using GNU Flex and Bison. The parsed feature is stored in a suitable data structure for further processing.

2. We explicate the stored feature by expanding all the BoxTalk macros and introducing an explicit representation of implicit behaviour. The resulting explicated feature is stored internally for further translation.

3. As a last step, our program translates the explicated feature into a Promela model.

Figure 1.1 depicts a graphical representation of our method. The BoxTalk specification and Promela model are the input and output of our program, respectively. The rectangles represent the phases of our program (which starts with a parser and finishes with a translator). The dotted part represents the verification step using the SPIN model checker.

The goal of our work was to fully automate the translation of BoxTalk specifications to Promela models. We used the macro expansion rules from [12] to explicate BoxTalk features before translating them into Promela models. Some of the generated Promela models are

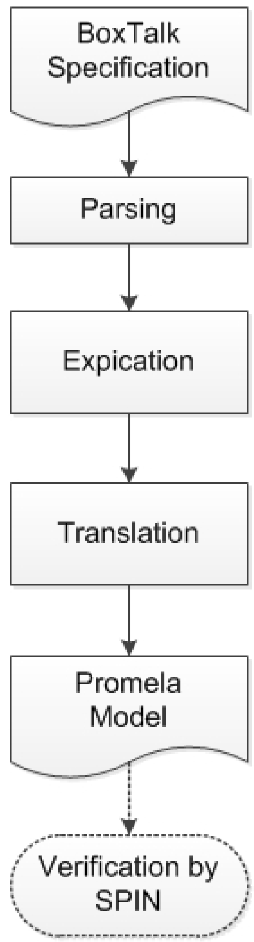


Figure 1.1: Method

upwards of `1K` lines of Promela code; developing such models by hand would be tedious and error-prone. With automated translation, we are able to translate BoxTalk features into Promela models with speed. We evaluate our work with a case study of BoxTalk features which we explicate, translate, and model check. Appendices contain the Promela models of some of the features from the case study.

1.4 Organization of this Document

The rest of this dissertation is organized as follows. Chapter 2 contains the background needed to understand the thesis: DFC, BoxTalk, the target model checker SPIN. Chapter 3 explains our explication strategy with two examples. Chapter 4 explains our translation of detailed BoxTalk models into Promela models. Chapter 5 describes our evaluation of the translator, using a case study of BoxTalk features. Chapter 6 concludes this dissertation.

Chapter 2

Background

This chapter provides the background required to understand this thesis. We start with a brief introduction to Distributed Feature Composition and then describe the Boxtalk modelling language. We also introduce the target model checker SPIN and its input language Promela.

2.1 DFC

Distributed Feature Composition (DFC) was designed to address the feature interaction problem. DFC is a component-based software architecture, where a complex system model is simplified by representing feature components as separate modules that are plugged into the architecture. DFC has a pipe-and-filter-like architecture [13]. Pipes are unidirectional streams of data and filters (i.e. features) are concurrent processes connected by pipes.

2.1.1 Usage

A traditional customer call is referred to as a **usage**. A usage is composed of several features which are linked together by internal calls. The phone or feature that places a call is termed the **caller** and the phone or feature that accepts the call is termed the **callee**. Figure 2.1 displays a simple usage in the DFC architecture. Boxes represent features and lines with arrow heads represent internal calls from the caller to the callee. An internal call is a bi-directional, point-to-point, feature-less connection that allows transmission of signals and media. At each end of an internal call is a **feature box**, which can place, receive, or tear down internal calls to other feature boxes. Feature boxes also act as interfaces to devices, trunks, and other resources.

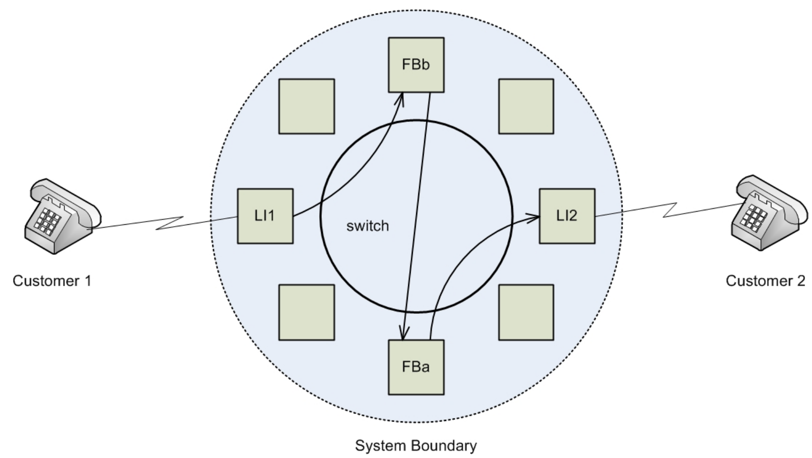


Figure 2.1: Usage - adopted from [9]

The usage in Figure 2.1 comprises a sequence of internal calls from line-interface box LI1 to feature box FBa to feature box FBb to line-interface box LI2.

The usage can be decomposed into **source** and **target** zones. Features in the source zone are features subscribed to by the caller; they are applied to all calls made by the caller. Features in the target zone are features subscribed to by the callee; they are applied to all calls directed to the target callee. Any feature box that is closer to the caller is **upstream** to other feature boxes that are further away from the caller. Feature boxes that are closer to the callee are **downstream** to those features that are closer to the caller. All of the caller's features are upstream to all of the callee's features.

2.1.2 DFC Protocol

Features use the DFC protocol to set up and tear down internal calls. The setup of an internal call from one feature to another feature is carried out by the router embedded in the DFC switch. *Setup*, *upack*, *teardown*, and *downack* are the primary DFC signals. The *setup* signal is used for setting up calls and every *setup* request must be acknowledged with an *upack* acknowledgement. In contrast, a *teardown* signal is used for tearing down calls, and every *teardown* signal must be acknowledged with a *downack* acknowledgement.

In DFC, call setup is piecewise; that is, every internal call is completed and acknowledged before setting up the next internal call. Figure 2.2 shows the piecewise setup. Piecewise setup ensures that the features do not have to wait idly for the receipt of an *upack* signal from the end of the usage. Instead, features can send and respond to signals immediately after they receive the *upack* signal.

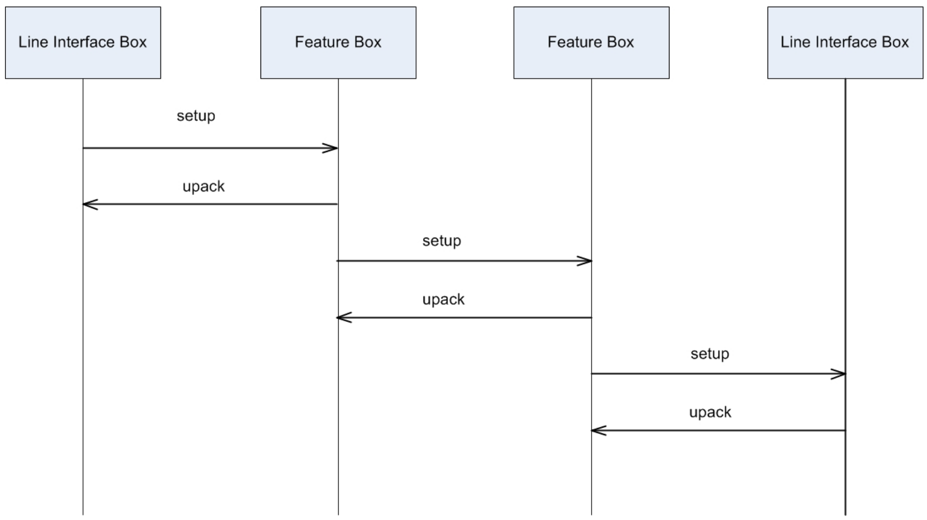


Figure 2.2: Piecewise Setup Process

In particular, features can respond immediately to the caller hanging up, rather than waiting until the usage is completely set up. In this manner, piecewise setup allows features to execute with more autonomy.

Internal calls are torn down in a similar fashion: a feature sends a *teardown* signal to its neighbouring features in the usage, each of which in turn sends an acknowledgement, *downack*, back. They also propagate the *teardown* signal to their neighbouring features, if any.

Apart from these four signals, there are four status signals used to convey the outcome of the original call request: *none*, *unknown*, *unavail*, and *avail*. Whereas an *upack* acknowledges the successful establishment of the internal call, the status signals are used to communicate whether the usage is successfully established. If the call setup is successful, signal *avail* is sent upstream. If the target address is invalid (i.e., does not exist), then signal *unknown* is sent upstream to the caller. If the callee is busy, signal *unavail* is sent upstream. Signal *none* cancels the effect of any of the three previous signals on an interface box. If the usage setup is not successful (for example, status signal *unknown* or *unavail*), the status signal is normally followed by a request to tear down the partial usage.

2.1.3 Calls, Call Variables, Port IDs

Internal calls between features are called **calls**. Whenever a new call is established, the feature allocates a port for that call and stores the port identifier in a call variable. **Call variables** refer to ports that represent the endpoints of active calls. All signals sent to or received from that call variable are actually sent to or received from the port associated with the variable. If a call is torn down, its port can be allocated to another call. Hence, port names do not uniquely identify calls. For simplicity, we often talk about calls being assigned to call variables.

Every feature box has one special port called *boxport* used for receiving *setup* signals. Figure 2.3 shows two feature boxes each with a *boxport* and two call variables, `in` and `out`.

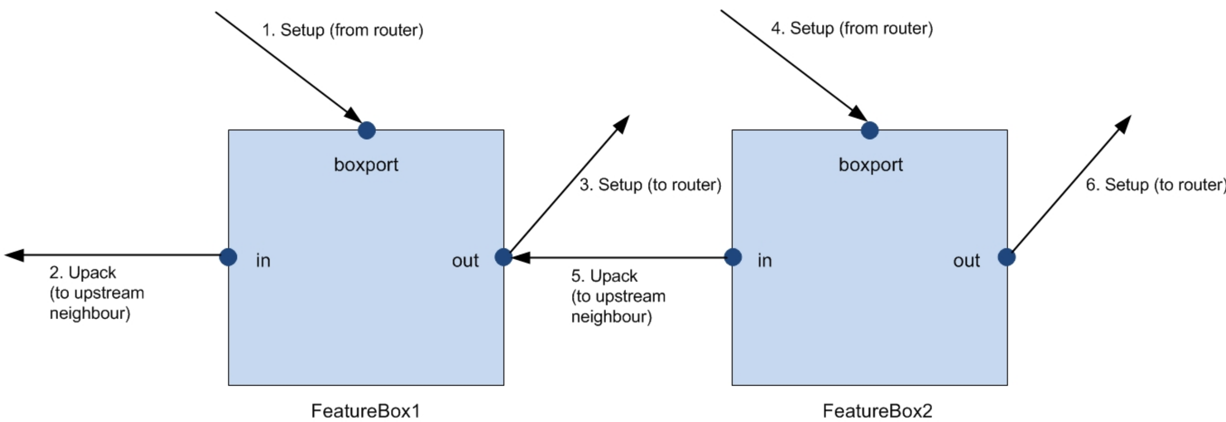


Figure 2.3: Ports

When *FeatureBox1* receives the *setup* signal on its *boxport*, it assigns the call to variable `in` and then sends acknowledgement *upack* to its upstream neighbour. The feature then continues the usage by forwarding the *setup* signal to the router via call variable *out*. The router determines the next box, *FeatureBox2*, and sends a *setup* signal to the *boxport* of that feature box. The *setup* signal contains *FeatureBox1's* address, to which *FeatureBox2* sends an *upack* signal. This establishes an internal call between *FeatureBox1* and *Feature-Box2*. *FeatureBox2* continues the usage by sending a *setup* signal to the router to set up the next internal call.

2.1.4 Free and Bound Boxes

Features are classified in two categories: **free features** and **bound features**. With free features, a new instance of the feature is instantiated every time the feature is included in a usage eg. Free Transparent Box (FTB). In contrast, there is only one instance of each bound feature per subscriber, and that one feature must be included in any usage involving the subscriber eg. Bound Transparent Box (BTB).

With respect to setting up and tearing down calls, free and bound features behave differently. A free feature receives only one *setup* signal in its lifetime, which causes the feature to be instantiated. In contrast, a bound feature can receive and react to multiple *setup* signals: one for every usage the subscriber is involved in. Moreover, a bound feature could receive a *setup* signal when the feature is already in the middle of a call.

A free feature ceases to exist once its calls are torn down. In contrast, a bound feature is normally ready to be added to a new usage as soon as it issues or receives a *teardown* signal along its current usage.

2.2 BoxTalk

BoxTalk is a high-level, domain-specific, call-abstraction language that facilitates easy and correct programming of DFC features [15]. In BoxTalk, DFC features are depicted as finite-state machines. A feature has ports for sending and receiving signals. Depending on the signals received, a feature performs different actions. BoxTalk uses an abstraction, a **call variable**, to refer to ports currently in use. Values of call variables can change over the course of a usage. We will see an example of this later in the chapter, when we discuss the *Call Waiting* feature box.

Four BoxTalk statements can alter the values of call variables:

- **rcv(c)** is an input event that reflects the receipt of a *setup* signal for setting up a new call assigned to call variable `c`.
- **ctu(i,c)** is the action that a feature performs to continue the setup of a usage, associating the new outgoing call to call variable `c`.
- Similar to **ctu()** is the macro **new(c)**, which initiates a new call and assigns the call to call variable `c`.

- An **assignment statement** is used to change the value of a call variable. For example, in call assignment *c1 , c2 = c2 , -*, call variable `c1` gets the value of call variable `c2`, and call variable `c2` gets the value *noCall*. Call assignments cause call variables to represent different calls at different points of a feature's execution.

2.2.1 States

BoxTalk supports four different types of control states:

- An **initial state** is depicted by a small black circle. Each feature has exactly one initial state. Initial states have no entering transitions.
- A **stable state** is shown as a rectangle. A feature box can have any number of stable states. Each stable state has at least one entering transition. Each exiting transition is triggered by a signal from the environment.
- A **transient state** is represented by a large, clear circle. Transient states are used to decompose a complex transition into a sequence of simple transitions. Transient states are non-responsive states, meaning that the feature does not read any new input in a transient state. Based on the evaluation of local variables, the outgoing transitions may take different actions and may lead to different states. At least one exiting transition out of a transient state should be enabled to ensure that execution is never blocked in a transient state.
- A **termination state** is represented by a heavy bar. A feature can have any number of termination states. Each termination state has at least one entering transition and no exiting transitions. A feature transitions to a termination state with the receipt of a *teardown* signal. Once the feature is in a termination state, it may react to other *teardown* signals with a *downack* signal, but ignores all other signals.

Apart from these explicit states, a feature also has an implicit **final state**, which exists only semantically. A final state has no graphical representation. A feature reaches its final state after all of its calls are completely torn down (i.e., all *teardown* signals have been acknowledged) and the feature is freed from the usage.

As we will see in Chapter 3 and Chapter 5, in original BoxTalk models, all states are depicted as mentioned above. In the explicated models, the *initial* state is represented by a small black circle, the *final* state is represented by two concentric circles (with a solid inner circle), and all of the remaining states are represented by rounded rectangles.

Active calls are calls which are fully established (i.e., received an acknowledgement *upack*). In a stable state, two *active* calls are **signal-linked** if their call variables are paired inside a parenthesis (for example, calls `a` and `s` are signal-linked in the *transparent* state in Call Waiting Feature, Figure 2.4). If two active calls are signal-linked, then any signal that arrives on either call is forwarded to the other call. This default behaviour of a signal-linked state is over-ridden by explicit transitions (we will see an example in Section 2.2.4).

2.2.2 Transitions

Transitions reflect state changes. They are shown as arrows going from a source state to a destination state. A transition's source and destination states may be the same state.

A transition exiting an initial state or any stable state is labelled with "trigger / action(s)", where:

- A **trigger** could be a simple input event, such as receiving a signal on a particular call or it can be a macro that combines an input event with actions, such as *rcv(callVariable)*.
- An **action** could be a simple action, such as sending a signal on a particular call or it can be a macro that combines multiple actions.

A transition is enabled if its trigger event is occurring. Actions are optional.

A transition exiting a transient state is labelled as "[guard] / action(s)", where the **guard** is a predicate on the state of the feature. The transition is enabled if the guard evaluates to true

2.2.3 Feature Behaviour

Features demonstrate the following types of behaviours:

1. **Reactive**: The feature reacts to an input from its environment by performing actions and perhaps changing state.
2. **Transparent**: If two active calls are signal-linked in a stable state, the default behaviour is to forward every signal received by one call to the other call. We say that a feature behaves "transparently" in a signal-linked state because the effect is

as if the feature does not exist and the two internal calls are directly connected. This default behaviour is over-ridden by explicit transitions triggered by specific input signals that cause the feature to exit the signal-linked state.

3. **Discarding events**: If a feature is in a non-signal-linked state and receives a signal that does not trigger any of the state's exiting transitions, the signal is discarded – meaning that no other feature in the usage will see the signal.

2.2.4 Call Waiting Feature Box

Let us look at an example BoxTalk feature for the feature Call Waiting (CW). CW is a bound feature that notifies its subscriber of an incoming call when the subscriber is already on the phone; it allows the subscriber to answer the new call without terminating the current call. Figure 2.4 shows a BoxTalk model of the CW feature.

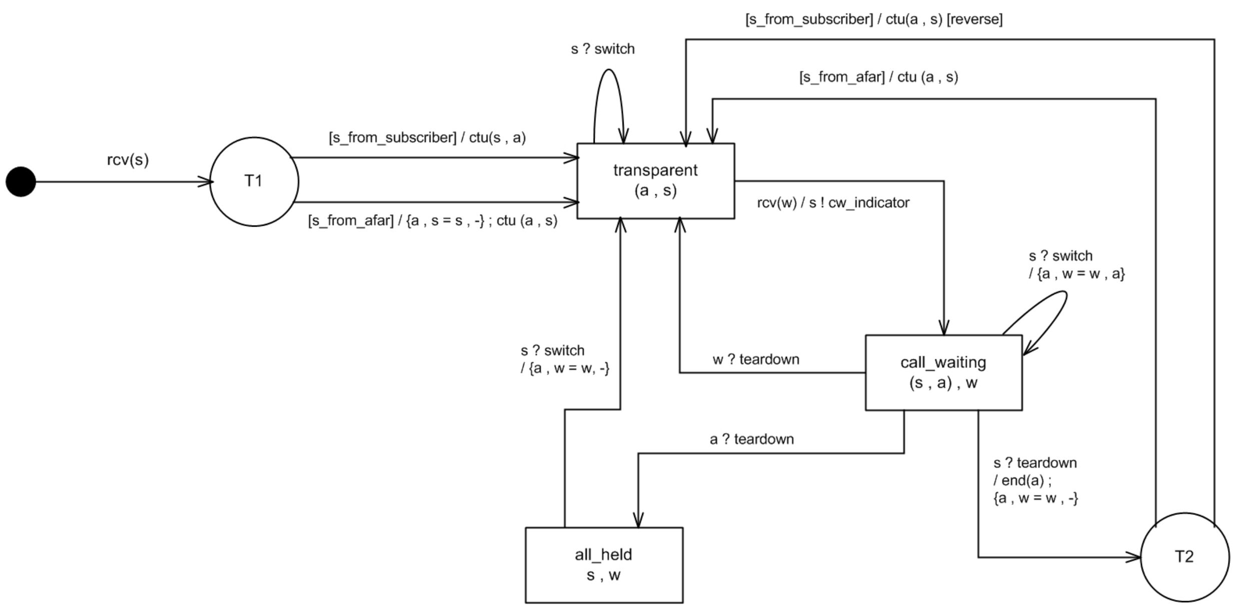


Figure 2.4: Call Waiting Feature Box - adapted from [15]

CW has three stable states (*transparent*, *call_waiting*, and *all_held*) and three call variables. Call `s` refers to the call that connects the feature to its subscriber; calls `a` and `w` refer to the *active* and *waiting* calls, respectively. *Active* call is a call that is voice-connected and *waiting* call is a call that is on hold; only one of these calls is connected with the subscriber at a time.

The CW feature is invoked when a new *setup* signal is received. In the *orienting* state, the feature orients itself with respect to its subscriber: if the call originates from the subscriber, then it remains associated with call variable `s`, where `s` denotes the subscriber; and the macro *ctu(s , a)* continues the usage by setting up the next call which is assigned to call variable `a`. If the initial call does not originate from the subscriber, then the subscriber is the intended callee; so the call variable values are switched and call variable `a` is associated with the initial call and call variable `s` is associated with *noCall*. The macro *ctu(a , s)* then continues the usage (towards the subscriber) via call variable `s`.

In the *transparent* state, the CW feature is dormant and the subscriber participates in the usage in a normal way. The calls associated with call variables `a` and `s` are signal-linked (i.e., all signals received by either call are forwarded to the other call). However, the feature never forwards a *switch*[1] signal from the subscriber as the *switch* signal is only meaningful as a subscriber command to the feature.

In the presence of a new call request, initially assigned to call variable `w`, the feature sends the subscriber a special tone and the feature transitions to the *call_waiting* state. The subscriber can send a *switch* signal to indicate that he or she wants to establish a voice connection with the *waiting* call. In this transition, the values of call variables `a` and `w` are swapped (with an assignment $a \, , \, w = w \, , \, a$), so the subscriber is now signal-linked with the other call. The subscriber can toggle back and forth between the two calls by repeatedly issuing the *switch* signal.

Any of the three parties can hang up at any time. If the user that is *waiting* hangs up, it is not noticed by the other two users; the feature simply transitions to the *transparent* state where the CW feature again lies dormant. If the *active* call hangs up, the feature transitions to the *all_held* state and waits for the subscriber to switch to the *waiting* call. If instead, the subscriber hangs up, call `a` is torn down. However, call `w` is still present. Rather than tearing down call `w`, the feature switches the values of call variables `a` and `w` to make the *waiting* call the *active* call, and then calls the subscriber back to re-establish the connection to the call that was on hold when the subscriber hung up.

[1] *Switch* is a CW feature specific signal

2.3 Model Checker SPIN

We translate explicated BoxTalk features into Promela (*Pro*cess *Me*ta *La*nguage) models because we want to use SPIN (*S*imple *P*romela *In*terpreter) [7] to verify explicated BoxTalk features. This section introduces the model checker SPIN and Section 2.4 talks about Promela. Readers may defer reading Section 2.3 and Section 2.4 until Chapter 5, which describes the translation from BoxTalk to Promela.

SPIN takes as input a behavioural model of the system-to-be-verified, expressed in Promela, and a set of properties of the system. SPIN exhaustively explores the execution paths of the model and checks whether a property holds on all paths. If the property does not hold on some path, then SPIN generates a counterexample: a trace of the execution path that violates the property. The most common types of errors caught by the SPIN model checker are deadlocks, violation of assertions, reachable bad states, and unreachable good states.

2.4 Promela

A typical Promela model is constructed from three basic objects:

- Process(es)
- Data objects
- Message channels

The program below shows a very simple Promela model.

```
active proctype main() {
int n = 5;
int sq;
sq = n * n;
printf("The square of %d is %d.", n , sq)
}
```

In this simple program, `active` and `proctype` are keywords used in Promela. The rest of the program has a simple C-like structure. The simulated execution of this process produces the following output:

```
$ The square of 5 is 25.
```

There is no semi-colon at the end of the last (printf) statement as semi-colons act as statement separators in Promela and not as statement terminators such as in C.

We discuss in detail all of the Promela constructs used by our models. For a thorough treatment on SPIN and Promela, please refer to [7].

2.4.1 Processes

A Promela model is composed of a set of processes that, together, describe the behaviour of a system. Each process is an instantiation of `proctype` and there must be at least one `proctype` declaration in the model. There are several ways to instantiate a process in Promela. We use the following approach:

```
active proctype process1() {
  printf("Process 1!")
}
```

Processes declared with the keyword `active` are instantiated automatically and are running when the simulation begins. Processes are always declared globally.

2.4.2 Data Objects

Data objects can be declared either globally or locally within a process. Table 2.1 lists the basic Promela data types along with their value ranges.

The data type `chan` is used to declare message-passing channels. For example, the following is a declaration of a channel `lc` of messages of type `bool` with the capacity of two messages:

```
chan lc = [2] of { bool };
```

The data type `mtype` is used to give mnemonic names to values. `mtype` declarations are usually placed at the start of a program. Separate `mtype` declarations in the same program are treated as one big `mtype` declaration. For example, the following two declarations

```
mtype = { m1 , m2 , m3 };
mtype = { m4 , m5 };
```

are treated internally by the program as a single declaration:

```
mtype = { m1 , m2 , m3 , m4, m5 };
```

Table 2.1: Basic Data Types

Type	Range
bit	0,1
bool	*false,true*
byte	0..255
chan	1..255
mtype	1..255
pid	0..255
short	$-2^{15}..2^{15}-1$
int	$-2^{31}..2^{31}-1$
unsigned	$0..2^{n}-1$

However, separate declarations offer better readability, and hence we use separate `mtype` declarations in our generated Promela models.

Multiple elements of the same type can be grouped together in an `array`. Arrays in Promela start with the index `zero` and different elements in the same array can be accessed by their index numbers. The following declares an array `c` of six message-passing channels:

```
chan c[6];
```

User-defined data structures can be defined in Promela using `typedef`:

```
typedef pstruct {
  mtype m1;
  chan c[3];
  bool pred = true
};
```

We use `typedef` in our models to define our constructs, which we will discuss in detail in Chapter 4.

2.4.3 Message Channels

Processes communicate with each other through message channels. The following declares a channel named `in` which is capable of storing up to five messages of type `mtype`

```
chan in = [5] of { mtype };
```

The following declaration is an array of six such message-passing channels

```
chan in[6] = [5] of { mtype };
```

The statement `in ! m1` sends a message `m1` (of type `mtype`) via channel `in`, and the statement `in ? m1` denotes the receipt of a message (assigned to variable `m1`) via channel `in`.

Rendezvous ports are used to synchronize the communication between two processes. Rendezvous communication occurs via channels of zero capacity. Such zero-capacity channels can pass messages, but cannot store messages. Message interactions via such rendezvous ports are by definition synchronous: communication proceeds only when both the sender and the receiver processes are ready for the rendezvous "handshake".

2.4.4 Executability

Statements in Promela model are either executable, meaning that they are able to run, or are blocked. Executable statements in Promela include the following:

- All `printf` statements
- Any statement guarded by an expression that evaluates to `true`
- Any `send` statement for which the associated channel has capacity for a new message
- Any `receive` statement for which the associated channel contains a message to be read
- Any `rendezvous communication` where both the sender and the receiver are ready for the handshake

Blocked statements include the following:

- Any statement guarded by an expression that evaluates to `false`
- Any `send` statement for which the associated channel is full
- Any `receive` statement for which the associated channel is empty
- Any `rendezvous communication` where either the sender or the receiver is not ready for the handshake

Promela has interleaving semantics of execution. Specifically, only one statement from one process can execute at a time. The scheduling algorithm nondeterministically chooses a process to execute from the set of executable processes.

2.4.5 Compound Statements

Promela supports five types of compound statements:

- Atomic sequences
- Deterministic steps[2]
- Selections
- Repetitions
- Escape sequences

An `atomic sequence` is used to group together two or more statements of one process, so that these statements execute as one statement without interleaving with other statements from other processes. Consider the following code:

```
active proctype process1() {
  statement1;
  atomic {
     statement2;
     statement3;
     statement4;
  }
}
```

In this simple process, `statement2`, `statement3`, and `statement4` execute in one step without any interruption from other processes. The only exception to this behaviour is when any of the statements are blocked. In such a case, control leaves the atomic block, executes one or more statements in some other process, and returns back to the blocked statement in the atomic block when it becomes executable. For example, consider the following code fragment:

```
chan c1 = [0] of { byte };
active proctype process1() { atomic { statement1; c1 ! 1 ; statement2 }
   }
active proctype process2() { atomic { c1 ? 1 ; statement3 } }
```

The execution will start with `process1()` as channel `c1` is empty initially. After executing statements `statement1` and `c1 ! 1` in the atomic sequence in `process1`, the `c1`

[2]We do not use deterministic steps, and hence we do not discuss them.

! 1 statement is blocked because of the incomplete rendezvous handshake. The atomic sequence of `process2()` is executed to completion, including the rendezvous handshake. Finally `process1()` resumes and executes `statement2`.
As will be seen, we use `atomic` statements to model state transitions, to reflect that state-transition actions take place in a single step.

A `selection` statement is used to nondeterministically select one option from a collection of conditional statements. Each conditional statement is composed of a `guard` and an `action`. A particular conditional statement is selected only if its `guard` evaluates to `true`, in which case, the respective `action` is executed. If more than one `guard` evaluates to `true`, one of the possible conditional statements' `actions` are nondeterministically selected for execution. The guards need not be mutually exclusive:

```
if
:: (a <= b) -> action1;
:: (a >= b) -> action2;
fi
```

In the example above, if `a` is less than `b`, then `action1` will be executed; if `a` is greater then `b`, then `action2` will be executed. However, if `a` is equal to `b`, then either `action1`, or `action2` will be nondeterministically selected for execution.

A `repetition` structure is a cyclic execution of a collection of conditional statements. It behaves the same way as a selection statement except that the statements execute repeatedly until a `break` statement is encountered, at which point the control passes to the statement immediately following the repetition structure. For example the following loop executes until `a == b`:

```
do
:: (a < b) -> b = b - a;
:: (a > b) -> a = a - b;
:: (a == b) -> break
od
```

The repetition structures are used to model the environment processes in our models. Section 4.1 describes the architecture of our Promela models.

An `escape` sequence is used to prioritize the execution of different statements in the same process. Consider the following, where `E` and `P` are arbitrary code fragments:

```
{ P } unless { E }
```

`P` (the main sequence) executes only if `E` (the escape sequence) is blocked. In other words, `E` has a priority over `P`.

2.4.6 Inline Functions

Inline functions in Promela are very similar to `C`-style macro definitions, but do not introduce any overhead during verification. A textual substitution of the inline function's body is made by the SPIN parser at every point of invocation. If the function holds parameters, the parser textually substitutes the formal parameters with the actual values.
An inline function has the following structure:

```
inline function_name( parameters_if_any ) {
  body
}
```

2.5 Property Language

In SPIN, correctness properties are formulated using the following constructs:

- Basic assertions
- End-state labels
- Progress-state labels
- Accept-state labels
- Never Claims
- Trace assertions
- Linear Temporal Logic formulas

In our work, we express properties in terms of Linear Temporal Logic (LTL) formulas and never claims.

2.5.1 Linear-time Temporal Logic

LTL models time sequentially and infinitely into the future [8]. LTL formulas are built using atomic propositions denoted by small letters, logical connectives such as $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$, and temporal operators such as X, $\Box$, $\Diamond$, U, W, R[3]. Logical connectives are defined as follows:

$\neg\phi$ is the negation of ϕ
$\phi \wedge \psi$ is the conjunction of ϕ and ψ
$\phi \vee \psi$ is the disjunction of ϕ and ψ
$\phi \rightarrow \psi$ means ϕ implies ψ (i.e., if ϕ then ψ)

The following description of LTL is from [3]:
State formulas are formulas that are `true` in a specific state, and *path formulas* are formulas that are `true` along a specific path. Temporal operators [3] (that we use) are defined as follows:

- The $\Diamond$ ("eventually" or "in the future") operator is used to assert that a property will hold at some state on the path
- The operator $\Box$ ("always" or "globally") specifies that a property holds at every state on the path
- The U ("until") operator specifies that there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.

LTL consists of formulas that have the form A[4]f where f is a path formula in which the only state subformulas permitted are atomic propositions [3]. An LTL path formula is either:

- If $p \in AP$, then p is a path formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\Box f$, $\Diamond f$, and fUg are path formulas.

Table 2.2 lists all of the logical connectives and temporal operators, along with their representation in SPIN.

[3] We do not use the W (weak until), R (release) and X (next) operators, and the $\leftrightarrow$ connective.
[4] A stands for all computation paths.

Table 2.2: Logical and Temporal Operators in LTL

	Operator	Logic	SPIN
Logical Connectives	not	$\neg$	!
	and	$\wedge$	&&
	or	$\vee$	||
	implies	$\rightarrow$	->
Temporal Operators	eventually	$\Diamond$	<>
	always	$\Box$	[]
	until	U	U

2.5.2 Never Claim

A `Never Claim` as the name suggests, specifies finite or infinite system behaviour that should never occur. A never claim has the following syntax:

`never { sequence }`

Never claims can be written manually or can be generated mechanically from LTL formulas.

In addition, we use basic assertions (`assert{false}`) to prove that our models are not vacuously true. Please refer to Chapter 5 for details.

Chapter 3

Explicating BoxTalk

BoxTalk is a call-abstraction language in which commonalities that occur in all features are abstracted away. This not only provides correct and efficient programming, but also emphasizes each feature's unique behaviour. However, for feature analysis, we cannot work with these abstracted features. Abstractions in BoxTalk are as follows:

- **Macros:** A macro combines a sequence of read, write, or assignment actions. BoxTalk macros include *rcv()*, *new()*, *ctu()*, *gone()*, and *end()*. Section 3.1 explains how these macros are expanded.
- **Hold queue:** Call setup is a two phase process; (1) sending a *setup* signal and (2) waiting for an acknowledgement *upack*. Whenever a feature sends a *setup* signal through any port, a hold queue is constructed for that port. Until the call is fully established (i.e., an *upack* signal is received on that port) all signals to be sent via that port are stored in the hold queue. When an acknowledgement *upack* is received on that port, the contents of the hold queue, if any, are forwarded to the newly established call.
- **Signal linkage:** Signal linkage was discussed in Chapter 2. In a stable state, two *active* calls are signal-linked if their call variables are paired inside a parenthesis. If two active calls are signal-linked in any state, then the default behaviour of the feature is to forward any status signal that arrives on either call to the other call.
- **Feature termination:** When all active calls of a free feature end, the feature transitions to a final state.

For feature verification, we need to concretize all of these abstractions. This process is called **detailing** or **explication**.

To explicate BoxTalk features, we must first parse the BoxTalk features. The BoxTalk grammar that was available to us was ambiguous and we had to resolve all ambiguities in order to parse the features. Appendix A1 lists the original grammar, and Appendix A2 lists our modified grammar. We developed a scanner using GNU Flex and developed a parser using GNU Bison. The explicated features are represented in our program as an annotated graph data structure in which the graph *nodes* represent BoxTalk *states* and the graph *edges* represent BoxTalk *transitions*. In the remainder of this chapter, we walk through the process of detailing the BoxTalk features with two running examples.

3.1 Macro-Expansion Rules

We worked with the macro-expansion rules from [12]. Table 3.1 displays the rules. The dotted part of the second rule represents our modifications to the existing rules. In this section, we first explain each original macro-expansion rule, and then explain our modifications (if any):

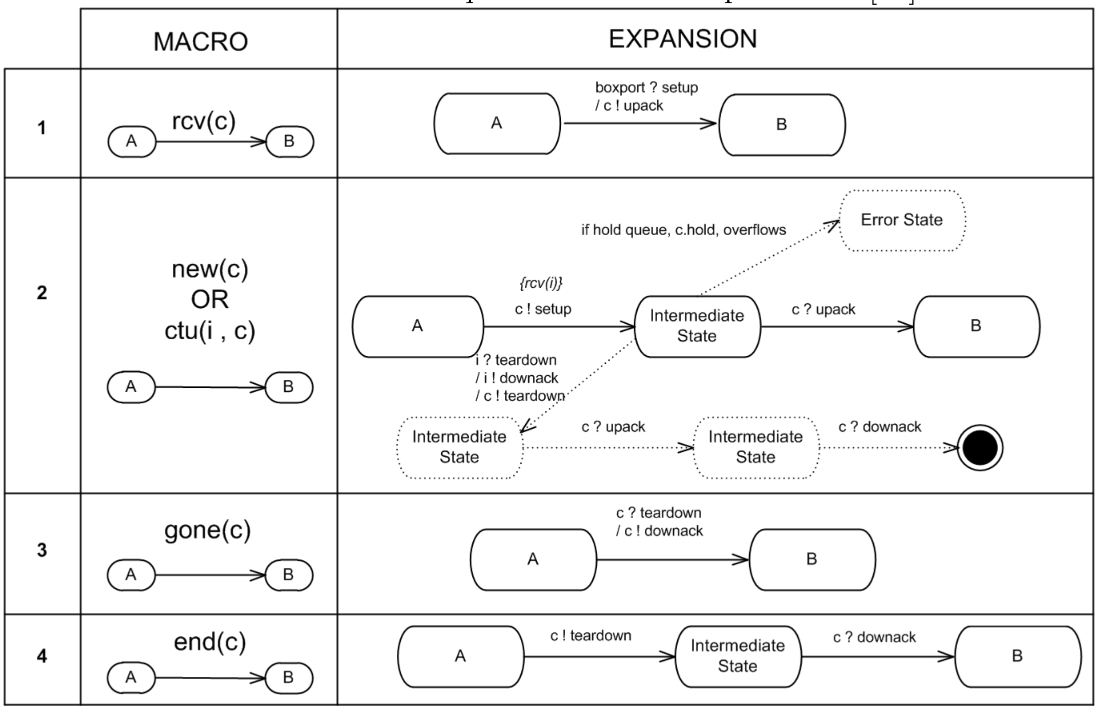
- *rcv(c)*: The feature receives a *setup* signal and sets up a new call assigned to call variable `c`. The macro *rcv(c)* is fully expanded as:

  boxport ? setup / c ! upack

- *new(c) / ctu(i,c)*: The expansion rules for macros *new(c)* and *ctu(i,c)* are very similar, hence we discuss their expansions together. The macro *new(c)* places a new call and assigns the call to call variable `c`. The macro *ctu(i,c)* continues the existing usage (in `i`) by setting up the next call in the usage and assigning the new call to call variable `c`. Each of these macros expand into a sequence of two transitions, with a new intermediate state being generated. In the first transition, a *setup* signal is sent, and in the intermediate state, the feature waits for an acknowledgement *upack* for call `c`. In the second transition, the acknowledgement is received for call `c`. The destination state of the second transition is the destination state of the original transition. Generally, the name of the intermediate state is *connecting_c*; however, the name may be different if other macros/actions are present in the original transition.

  We now discuss our modifications to the existing rules of [12] for expanding macros *new()* and *ctu()*. First, in the intermediate state, call `c` is not fully set up until the

Table 3.1: Macro expansion rules - adapted from [12]



receipt of an acknowledgement signal. However, signals to be sent along this half-complete call should not be lost. A **hold queue**, *c.hold*, is constructed to hold all of the signals to be sent via this call. Once call `c` is fully setup, with the receipt of an *upack* signal, the contents of the hold queue, if any, are forwarded to the newly established call.

Second, in BoxTalk, hold queues are infinitely long and never overflow. However, for finite analysis, we need to put a bound on the sizes of hold queues in our models. We introduce an *error* state that represents an overflow of a hold queue: in a half-complete call, when a hold queue reaches its capacity and that call receives another signal, the feature transitions to the *error* state. *Error* states are *final* states.

Third, the caller may hang up at any time (even as a new call is being set up). To handle this special case, extra states and transitions are required. The dotted part in bottom half of graphical rule expansion (Table 3.1) represents the sequence of transitions that model this behaviour. If the caller hangs up (represented by *i ? teardown / i ! downack*) in the intermediate state before call `c` is fully set up, the feature must terminate call `c` in a particular fashion. First, a *teardown* signal is sent via call c and the feature transitions to a second new intermediate state (usually named *abandonConnection_c*). Recall that call c has not yet received an *upack* signal. Thus, the other end of call c must acknowledge the call setup before acknowledging the call teardown. When call c receives an *upack* signal, the feature transitions to a third new intermediate state (usually named *terminating_c*), where call c waits for a *downack* signal. When call c receives a *downack* signal, the feature transitions to the *final* state.

- *gone(c)*: This macro models the case in which the remote end of call c initiates a teardown of call c. It is expanded as:

  c ? teardown / c ! downack

  If call c is signal-linked with another call, say call o, the macro expansion also includes the action *end(o)*.

- *end(c)*: This macro models the case in which the feature initiates the teardown of call c. The macro is expanded into a sequence of two transitions with a new intermediate state (usually named *terminating_c*). In the first transition, a *teardown* signal is sent on call c, and the feature transitions to the new intermediate state. In this state, the feature waits for a *downack* signal – the receipt of which transitions the feature to the *final* state.

3.2 Explication Algorithm

In this section, we present our explication algorithm, which includes expanding macros (based on explication rules discussed in Section 3.1) and other abstractions discussed at the start of this chapter. In the next section, we explain every step of our algorithm in detail with the example of Free Transparent Box. First we present the pseudo code of our algorithm and then we explain it.

The explication algorithm constructs a new model. The original BoxTalk model is defined in terms of $\langle S^s, S^t, T\rangle$ where S^s is the set of *stable* states, S^t is the set of *transient* states, and T is the set of transitions. The new explicated BoxTalk model is defined in terms of $\langle ES, ET\rangle$ where ES is the set of states, and ET is the set of transitions. We further classify ES as follows:

- ES^a – Set of states in which the caller has status *active*[1] but is not *signal-linked*
- ES^{sl} – Set of *signal-linked* states
- ES^c – Set of *connecting* states
- ES^t – Set of *terminating* states

In all of the procedures, parts of the code contained in braces '{' and '}' are comments. We use a plus sign (+) to refer to combining action labels in one transition. If actions are combined in a transition, then all actions have to occur in that transition.

Procedures 3.2, 3.3, 3.4, and 3.5 are part of the same algorithm; we split these for ease of reading. Algorithm 3.1 calls these procedures to explicate the BoxTalk model with set of transitions T.

The new model is constructed in an incremental fashion (i.e., we build one transition in each step) instead of expanding an abstraction in one step.

As a first step of the explication algorithm (Procedure 3.2), for each transition in the original BoxTalk specification, we expand all of the macros in that transition. If more than one macro is present, the first transition of each expanded macro, as described in Table 3.1, is combined into a single joint transition in the new model (Procedure 3.2, lines 4 - 28). The name of the intermediate state generated depends on the macro combinations being explicated. The macro combinations in our algorithm are not exhaustive. However, the macro combinations suffice for all of the BoxTalk models available to us. As it can be seen from Table 3.1, expansion of macros *new()*, *ctu()*, and *end()* require acknowledgements. Recursive function $complete()$ handles receipt of pending acknowledgements. Call variable sets (Section 3.3.3) help us keep track of all the calls and their pending acknowledgements.

The recursive function $complete(tp_i, es_j)$ (Procedure 3.7) is called from Procedure 3.2 (Procedure 3.2, line 52). It completes the expansion of macros that require acknowledgements. For each pending acknowledgement, the function creates an outgoing transition from the *source* state to model the fact that acknowledgements can be received in any order (Procedure 3.7, lines 2 and 13). If the destination state(s) of these transitions also have pending acknowledgement(s), the function is called recursively (Procedure 3.7, line 26). Eventually, the destination state of the original BoxTalk specification is reached.

The function $search(es_n, ES, tp_i)$ (Procedure 3.6) avoids the creation of duplicate states in the new model. The function is called every time a new state (es_n) is encountered. This

[1] All fully established calls have status *active*.

function checks if the to-be-created state already exists in the set of states ES and, if it does, the function returns the existing state (Procedure 3.6, line 4).

The function $callsets(es_s, es_t, \{Set\ of\ Macros\})$ (Procedure 3.9) displays how the call variable sets are updated when specific macros are present in the original transition. For different macro combinations, "**if**" statements on lines 3, 6, 9, 12, and 15 are combined accordingly.

Procedure 3.3 handles the possibility of the caller hanging up from states in which the caller's status is *active*. Recursive function $terminate(es_j)$ (Procedure 3.8) handles receipt of pending acknowledgements. This function, which is called from Procedure 3.3 (Procedure 3.3, line 14), is similar to function *complete*(), which is called from Procedure 3.2. However, there are subtle differences between the two recursive function. In function *complete*(), the *eventual* destination state after receipt of all pending acknowledgements is the destination state of the original BoxTalk specification, whereas, in function $terminate()$, the *eventual* destination state is state *final*. In function *complete*(), all half-complete calls require only one acknowledgement, *upack*, for completion of their setup. In function $terminate()$, all half-complete calls[2] first require an acknowledgement *upack*, and then a *downack*, for completion of their termination.

Procedure 3.4 handles feature termination from signal-linked states (Section 3.3.2). Procedure 3.5 augments *signal-linked*, *connecting*, and *terminating* states of the feature with self transitions (Section 3.3.4).

In the remainder of this section, we present our algorithm composed of several procedures.

Algorithm 3.1 Explication Algorithm

$macro\_expansion(T)$
$caller\_hang\_up(ES^a)$
$termination\_sl(ES^{sl})$
$self\_transitions(ES^{sl}, ES^c, ES^t)$

[2]These calls are torn down when the caller hangs up.

Procedure 3.2 Macro Expansion: Function:- $macro\_expansion(T)$ – (Section 3.3.1)

T = Transitions in the original BoxTalk model
$\forall\ tp_i \in T$
    Create a new transition et_j that combines the non-macro labels of tp_i with the labels of the first transitions of the expanded macros of tp_i
    $\mathbf{et_j}$.**source.name** = $tp_i.source.name$
    if $et_j.source$ is a *transient* state **then** {Macros do not codify guards}
        $\mathbf{et_j}$.**guard** = $tp_i.guard$
    end if
    {Destination state ($et_j.dest$), trigger ($et_j.trigger$) and actions ($et_j.actions$) depend on macro combinations}
    switch
        case rcv(i) + ctu(i,c):
            $\mathbf{et_j}$.**dest.name** = $connecting\_c$; $\mathbf{et_j}$.**actions** = $i\ !\ upack + c\ !\ setup + tp_i.actions$; $\mathbf{et_j}$.**trigger** = $boxport\ ?\ setup$
            $callsets(et_j.source, et_j.dest, \{rcv(i), ctu(i, c)\})$
            break
        case rcv(i):
            $\mathbf{et_j}$.**dest.name** = $tp_i.dest.name$; $\mathbf{et_j}$.**actions** = $i\ !\ upack + tp_i.actions$; $\mathbf{et_j}$.**trigger** = $boxport\ ?\ setup$
            $callsets(et_j.source, et_j.dest, \{rcv(i)\})$
            break
        case gone(i) + end(c):
            $\mathbf{et_j}$.**dest.name** = $terminating\_c$; $\mathbf{et_j}$.**actions** = $i\ !\ downack + c\ !\ teardown + tp_i.actions$; $\mathbf{et_j}$.**trigger** = $i\ ?\ teardown$
            $callsets(et_j.source, et_j.dest, \{gone(i), end(c)\})$
            break
        case gone(c):
            $\mathbf{et_j}$.**dest.name** = $tp_i.dest.name$; $\mathbf{et_j}$.**actions** = $c\ !\ downack + tp_i.actions$; $\mathbf{et_j}$.**trigger** = $c\ ?\ teardown$
            $callsets(et_j.source, et_j.dest, \{gone(c)\})$
            break
        case ctu(i,c1) + ctu(i,c2):
            $\mathbf{et_j}$.**dest.name** = $trying\_c1\_c2$; $\mathbf{et_j}$.**actions** = $c1\ !\ setup + c2\ !\ setup + tp_i.actions$; $\mathbf{et_j}$.**trigger** = $tp_i.trigger$
            $callsets(et_j.source, et_j.dest, \{ctu(i, c1), ctu(i, c2)\})$
            break
        case end(c1) + end(c2):
            $\mathbf{et_j}$.**dest.name** = $ending\_c1\_c2$; $\mathbf{et_j}$.**actions** = $c1\ !\ teardown + c2\ !\ teardown + tp_i.actions$; $\mathbf{et_j}$.**trigger** = $tp_i.trigger$
            $callsets(et_j.source, et_j.dest, \{end(c1), end(c2)\})$
            break
        case end(c) + new(r):
            $\mathbf{et_j}$.**dest.name** = $switching$; $\mathbf{et_j}$.**actions** = $c\ !\ teardown + r\ !\ setup + tp_i.actions$; $\mathbf{et_j}$.**trigger** = $tp_i.trigger$
            $callsets(et_j.source, et_j.dest, \{end(c), new(r)\})$
            break
        case new(c) or ctu(i,c):
            $\mathbf{et_j}$.**dest.name** = $connecting\_c$; $\mathbf{et_j}$.**actions** = $c\ !\ setup + tp_i.actions$; $\mathbf{et_j}$.**trigger** = $tp_i.trigger$
            $callsets(et_j.source, et_j.dest, \{new(c)\})$ or $callsets(et_j.source, et_j.dest, \{ctu(i, c)\})$
            break
        case end(c):
            $\mathbf{et_j}$.**dest.name** = $terminating\_c$; $\mathbf{et_j}$.**actions** = $c\ !\ teardown + tp_i.actions$; $\mathbf{et_j}$.**trigger** = $tp_i.trigger$
            $callsets(et_j.source, et_j.dest, \{end(c5)\})$
            break
        case `default`: {If no macros are present}
            $\mathbf{et_j}$.**dest.name** = $tp_i.dest.name$; $\mathbf{et_j}$.**actions** = $tp_i.actions$; $\mathbf{et_j}$.**trigger** = $tp_i.trigger$
    if $et_j.dest == tp_i.dest$ **then**
        $et_j.dest = search(et_j.dest, ES, tp_i)$
    else
        $et_j.dest = search(et_j.dest, ES, NULL)$
    end if
    $complete(tp_i, et_j.dest)$

Procedure 3.3 Caller hanging up: Function:- *caller_hang_up*(ES^a) – (Section 3.3.1)

1: $\forall\ es_i^a \in ES^a$

2: $\forall\ Ca_k \in es_i^a.Active$

3: Add a new outgoing transition (et_j) {reflecting Ca_k hanging up}

$\mathbf{et_j.trigger} = Ca_k\ ?\ teardown$;

$\mathbf{et_j.actions} = Ca_k\ !\ downack$

`copy` all call variable sets of es_i^a to $et_j.dest$

`remove` Ca_k from $et_j.dest.Active$

4: **if** $et_j.dest.Requested \neq \varnothing$ **then**

5: $\forall\ Cr_l \in et_j.dest.Requested$

6: $\mathbf{et_j.actions} = et_j.actions + Cr_l\ !\ teardown$

`remove` Cr_l from $et_j.dest.Requested$

`add` Cr_l into $et_j.dest.Abandoned$

7: **end if**

8: **if** $et_j.dest.Active \neq \varnothing$ **then**

9: $\forall\ Ca_h \in et_j.dest.Active$

10: $\mathbf{et_j.actions} = et_j.actions + Ca_h\ !\ teardown$

`remove` Ca_h from $et_j.dest.Active$

`add` Ca_h into $et_j.dest.Terminating$

11: **end if**

12: $\mathbf{et_j.source.name} = es_i^a.name$;

$\mathbf{et_j.dest.name} = abandonConnection\_calls$ {*calls* are all the call variables belonging to $es_i^a.Requested$}

13: $\mathbf{et_j.dest} = search(et_j.dest, ES, NULL)$

14: $terminate(et_j.dest)$

Procedure 3.4 Termination from signal-linked states: Function:- $termination\_sl(ES^{sl})$ – (Section 3.3.2)

$\forall\ es_i^{sl} \in ES^{sl}$
    $Ca1$ = *Signal-linked* Call #1
    $Ca2$ = *Signal-linked* Call #2
    {Consider the case where $Ca1$ hangs up}
    Add a new outgoing transition (et_{j1}) {reflecting $Ca1$ hanging up}
    $\mathbf{et_{j1}.trigger} = Ca1\ ?\ teardown$;
    $\mathbf{et_{j1}.actions} = Ca1\ !\ downack + Ca2\ !\ teardown$;
    $\mathbf{et_{j1}.source.name} = es_i^{sl}.name$; $\mathbf{et_{j1}.dest.name} = terminating\_Ca2$
    `copy` all call variable sets of es_i^{sl} to $et_{j1}.dest$
    `remove` $Ca1$ from $et_{j1}.dest.Active$
    `remove` $Ca2$ from $et_{j1}.dest.Active$
    `add` $Ca2$ into $et_{j1}.dest.Terminating$
    $et_{j1}.dest = search(et_{j1}.dest, ES, NULL)$
    Add a new outgoing transition (et_{j2}) from *terminating_*$Ca2$ triggered by $Ca2$ receiving *downack*
    $\mathbf{et_{j2}.trigger} = Ca2\ ?\ downack$; $\mathbf{et_{j2}.actions} = \varnothing$;
    $\mathbf{et_{j2}.source.name} = terminating\_Ca2$; $\mathbf{et_{j2}.dest.name} = final$ {In state $final$, all call variable sets are empty}
    $et_{j2}.dest = search(et_{j2}.dest, ES, NULL)$
    {Consider the case where $Ca2$ hangs up}
    Add a new outgoing transition (et_{j3}) {reflecting $Ca2$ hanging up}
    $\mathbf{et_{j3}.trigger} = Ca2\ ?\ teardown$;
    $\mathbf{et_{j3}.actions} = Ca2\ !\ downack + Ca1\ !\ teardown$;
    $\mathbf{et_{j3}.source.name} = es_i^{sl}.name$; $\mathbf{et_{j3}.dest.name} = terminating\_Ca1$
    `copy` all call variable sets of es_i^{sl} to $et_{j3}.dest$
    `remove` $Ca2$ from $et_{j3}.dest.Active$
    `remove` $Ca1$ from $et_{j3}.dest.Active$
    `add` $Ca1$ into $et_{j3}.dest.Terminating$
    $et_{j3}.dest = search(et_{j3}.dest, ES, NULL)$
    Add a new outgoing transition (et_{j4}) from *terminating_*$Ca1$ triggered by $Ca1$ receiving *downack*
    $\mathbf{et_{j4}.trigger} = Ca_{sl1}\ ?\ downack$; $\mathbf{et_{j4}.actions} = \varnothing$;
    $\mathbf{et_{j4}.source.name} = terminating\_Ca1$; $\mathbf{et_{j4}.dest.name} = final$
    $et_{j4}.dest = search(et_{j4}.dest, ES, NULL)$

Procedure 3.5 Self Transitions: Function:- $self\_transitions(ES^{sl}, ES^{c}, ES^{t})$ – (Section 3.3.4)

$\forall\ es_i^{sl} \in ES^{sl}$
    $Ca1$ = *Signal-linked* Call #1
    $Ca2$ = *Signal-linked* Call #2
    Add two outgoing transitions (et_{j1} and et_{j2}) from state es_i^{sl} with same destination state (es_i^{sl})
    $\mathbf{et_{j1}.trigger} = Ca1\ ?\ sig$; $\mathbf{et_{j1}.actions} = Ca2\ !\ sig$;
    $\mathbf{et_{j1}.source} = \mathbf{et_{j1}.dest} = es_i^{sl}$
    $\mathbf{et_{j1}.guard}$ =[$sig \neq SIGNAL_{t1}$] {$SIGNAL_{t1}$ is any signal for $Ca1$ that explicitly triggers a transition that exits state es_i^{sl}}
    $\mathbf{et_{j2}.trigger} = Ca2\ ?\ sig$; $\mathbf{et_{j2}.actions} = Ca1\ !\ sig$;
    $\mathbf{et_{j2}.source} = \mathbf{et_{j2}.dest} = es_i^{sl}$
    $\mathbf{et_{j2}.guard}$ =[$sig \neq SIGNAL_{t2}$] {$SIGNAL_{t2}$ is any signal for $Ca2$ that explicitly triggers a transition that exits state es_i^{sl}}
$\forall\ es_i^{c} \in ES^{c}$
    $Ca \in es_i^{c}.Active$
    $Cr \in es_i^{c}.Requested$
    Add an outgoing transition (et_j) from state es_i^c with same destination state (es_i^c)
    $\mathbf{et_{j}.trigger} = Ca\ ?\ sig$; $\mathbf{et_{j}.actions} = Cr\_hold\ !\ sig$;
    $\mathbf{et_{j}.source} = \mathbf{et_{j}.dest} = es_i^{c}$
    $\mathbf{et_{j}.guard}$ =[$sig \neq SIGNAL_t$ && $Cr\_hold \neq Full$] {$SIGNAL_t$ is any signal for Ca that explicitly triggers a transition that exits state es_i^c, and Cr_hold is the hold queue}
$\forall\ es_i^{t} \in ES^{t}$
    $Ct \in es_i^{c}.Terminating$
    Add an outgoing transition (et_j) from state es_i^t with same destination state (es_i^t)
    $\mathbf{et_{j}.trigger} = Ct\ ?\ teardown$; $\mathbf{et_{j}.actions} = Ct\ !\ downack$;
    $\mathbf{et_{j}.source} = \mathbf{et_{j}.dest} = es_i^{t}$

Procedure 3.6 Function:- $ES\ search(es_n, ES, tp_k)$ – (Section 3.3.3)

```
∀ es_i ∈ ES {ES is the set of existing states}
    if es_i.name == es_n.name then
        if es_i.Active == es_n.Active &&
          es_i.Requested == es_n.Requested &&
          es_i.Abandoned == es_n.Abandoned &&
          es_i.Terminating == es_n.Terminating then
            return es_i
        end if
    end if
    if es_n.Active ≠ ∅ then
        if tp_k ≠ NULL && tp_k.dest ∈ signal-linked then
            push es_n into set of signal-linked states ES^sl
        else
            push es_n into set of active states ES^a
        end if
    end if
    if es_n.Requested ≠ ∅ then
        push es_n into set of connecting states ES^c
    end if
    if es_n.Terminating ≠ ∅ then
        push es_n into set of terminating states ES^t
    end if
    return es_n
```

Procedure 3.7 Function:- $complete(tp_i, es_j)$: Receipt of pending acknowledgements

1: $\forall\ Cr_i \in es_j.Requested$
 {If macro *new()* or *ctu()* is explicated in Procedure 3.2, $es_j.Requested$ will not be empty}
2: Create an outgoing transition (et_{k1}) from state es_j triggered by a corresponding *upack*
 $\mathbf{et_{k1}.trigger} = Cr_i$? *upack*; $\mathbf{et_{k1}.actions} = \varnothing$; $\mathbf{et_{k1}.source} = es_j$
 `copy` all call variable sets of es_j to $et_{k1}.dest$
 `remove` Cr_i from $et_{k1}.dest.Requested$
 `add` Cr_i into $et_{k1}.dest.Active$ {Receipt of an *upack*}
3: **if** $et_{k1}.dest.Terminating \neq \varnothing$ **then** {If there are calls with pending *downacks*}
4: $\mathbf{et_{k1}.dest.name} = waiting\_call\_down$ {*call* is the contents of $et_{k1}.dest.Terminating$}
5: **else if** $et_{k1}.dest.Requested \neq \varnothing$ {If some calls are still pending *upack*}
6: $\mathbf{et_{k1}.dest.name} = connecting\_call$
7: **else** {All acknowledgements received}
8: $\mathbf{et_{k1}.dest.name} = tp_i.dest.name$
9: **end if**
10: **if** $et_{k1}.dest == tp_i.dest$ **then** $\mathbf{et_{k1}.dest} = search(et_{k1}.dest, ES, tp_i)$
11: **else** $\mathbf{et_{k1}.dest} = search(et_{k1}.dest, ES, NULL)$ **end if**
12: $\forall\ Ct_i \in es_j.Terminating$
 {If macro *end()* is explicated in Procedure 3.2, $es_j.Terminating$ will not be empty}
13: Create an outgoing transition (et_{k2}) from state es_j triggered by a corresponding *downack*
 $\mathbf{et_{k2}.trigger} = Ct_i$? *downack*; $\mathbf{et_{k2}.actions} = \varnothing$; $\mathbf{et_{k2}.source} = es_j$
 `copy` all call variable sets of es_j to $et_{k2}.dest$
 `remove` Ct_i from $et_{k2}.dest.Terminating$ {Receipt of a *downack*}
14: **if** $et_{k2}.dest.Requested \neq \varnothing$ **then** {If there are calls with pending *upacks*}
15: $\mathbf{et_{k2}.dest.name} = connecting\_call$
16: **else if** $et_{k2}.dest.Terminating \neq \varnothing$ {If some calls are still pending *downack*}
17: $\mathbf{et_{k2}.dest.name} = waiting\_call\_down$
18: **else** {All acknowledgements received}
19: $\mathbf{et_{k2}.dest.name} = tp_i.dest.name$
20: **end if**
21: **if** $et_{k2}.dest == tp_i.dest$ **then** $\mathbf{et_{k2}.dest} = search(et_{k2}.dest, ES, tp_i)$
22: **else** $\mathbf{et_{k2}.dest} = search(et_{k2}.dest, ES, NULL)$ **end if**
23: $\forall\ t_i \in es_j.OUTTRAN$ {$es_j.OUTTRAN$ is the set of transitions that exit state es_j created in lines 2 and/or 13}
24: $es_{curr} = t_i.dest$
25: **if** $es_{curr}.Requested \neq \varnothing \vee es_{curr}.Terminating \neq \varnothing$ **then** {If acks. pending}
26: $complete(tp_i, es_{curr})$
27: **end if**

Procedure 3.8 Function:- *terminate*(es_j): Receipt of pending acknowledgements when caller hangs up

1: $\forall\ Ch_i \in es_j.Abandoned$
2: Create an outgoing transition (et_{k1}) from state es_j triggered by a corresponding *upack*
$\mathbf{et_{k1}.trigger} = Ch_i\ ?\ upack$; $\mathbf{et_{k1}.actions} = \varnothing$; $\mathbf{et_{k1}.source} = es_j$
`copy` all call variable sets of es_j to $et_{k1}.dest$
`remove` Ch_i from $et_{k1}.dest.Abandoneded$
`add` Ch_i into $et_{k1}.dest.Terminating$ {Receipt of an *upack*}
3: **if** $et_{k1}.dest.Terminating \neq \varnothing$ && $et_{k1}.dest.Abandoned \neq \varnothing$ **then**
4: $\mathbf{et_{k1}.dest.name} = abandoning\_call$
5: **else if** $et_{k1}.dest.Terminating \neq \varnothing$ **then** {If there are calls with pending *downacks*}
6: $\mathbf{et_{k1}.dest.name} = terminating\_call$
7: **else if** $et_{k1}.dest.Abandoned \neq \varnothing$ {If some half-complete, torn down calls are still pending *upack*}
8: $\mathbf{et_{k1}.dest.name} = waiting\_call\_up$
9: **end if**
10: $\mathbf{et_{k1}.dest} = search(et_{k1}.dest, ES, NULL)$
11: $\forall\ Ct_i \in es_j.Terminating$
12: Create an outgoing transition (et_{k2}) from state es_j triggered by a corresponding *downack*
$\mathbf{et_{k2}.trigger} = Ct_i\ ?\ downack$; $\mathbf{et_{k2}.actions} = \varnothing$; $\mathbf{et_{k2}.source} = es_j$
`copy` all call variable sets of es_j to $et_{k2}.dest$
`remove` Ct_i from $et_{k2}.dest.Terminating$ {Receipt of a *downack*}
13: **if** $et_{k2}.dest.Terminating \neq \varnothing$ && $et_{k2}.dest.Abandoned \neq \varnothing$ **then**
14: $\mathbf{et_{k2}.dest.name} = abandoning\_call$
15: **else if** $et_{k2}.dest.Abandoned \neq \varnothing$ **then** {If there are half-complete, torn-down calls with pending *upacks*}
16: $\mathbf{et_{k2}.dest.name} = waiting\_call\_up$
17: **else if** $et_{k2}.dest.Terminating \neq \varnothing$ {If some calls are still pending *downack*}
18: $\mathbf{et_{k2}.dest.name} = terminating\_call$
19: **else** {All acknowledgements received}
20: $\mathbf{et_{k2}.dest.name} = final$
21: **end if**
22: $\mathbf{et_{k2}.dest} = search(et_{k2}.dest, ES, NULL)$
23: $\forall\ t_i \in es_j.OUTTRAN$ {$es_j.OUTTRAN$ is the set of transitions that exit state es_j created in lines 2 and/or 12}
24: $es_{curr} = t_i.dest$
25: **if** $es_{curr}.Abandoned \neq \varnothing \vee es_{curr}.Terminating \neq \varnothing$ **then** {If acks. pending}
26: $terminate(es_{curr})$
27: **end if**

Procedure 3.9 Function:- $callsets(es_s, es_t, \{Set\ of\ Macros\})$

```
copy all call variable sets of es_s to es_t {es_s is the source state and es_t is the destination state}
∀ c
    if macro rcv(c) ∈ Set of Macros then
        add c into es_t.Active
    end if
    if macro gone(c) ∈ Set of Macros then
        remove c from es_t.Active
    end if
    if macro new(c) ∈ Set of Macros then
        add c into es_t.Requested
    end if
    if macro ctu(i, c) ∈ Set of Macros then
        add c into es_t.Requested
    end if
    if macro end(c) ∈ Set of Macros then
        remove c from es_t.Active
        add c into es_t.Terminating
    end if
```

3.3 Explicating BoxTalk - Free Features

In this section, we present in detail the explication of free BoxTalk features, using the explication of Free Transparent Box (FTB) as an example.

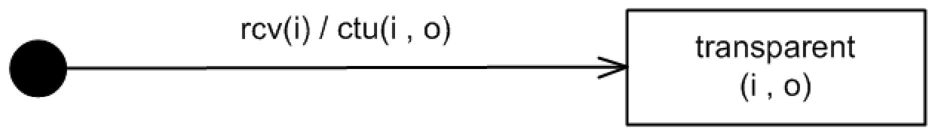


Figure 3.1: FTB - Original Specification

FTB is a simple feature that behaves only transparently. It is used for demonstration purposes only (though its behaviour is included in other more complex features that have signal-linked calls). Figure 3.1 displays the original specification of FTB. It has only two states, one *initial* state and one stable state named *transparent*, and one transition from the *initial* state to the *transparent* state. The feature is added to the usage with the received call request, which is assigned to call variable `i`. The feature continues the usage by setting up the next call `o`.

The *transparent* state represents the feature after call `o` receives an *upack* signal. At this point, the two calls (`i` and `o`) are *signal-linked*.

Our explication of a free feature is a three step process:

1. The first step expands all of the macros present in the transitions of the original specification. Intermediate states and new transitions may be created in this step and explicated in future steps. A final state may be created (if not already present); it is the destination state of some of the new transitions in the explicated model.

2. The second step handles the termination of signal-linked calls. Extra states and transitions may be created in this step as well.

3. The third step augments the feature with self-transitions. A self-transition is a transition whose source and destination are the same state.

We explain all of these steps in detail as we walk through the explication of the feature FTB.

3.3.1 Step 1 - Expanding Macros

The explication of macros is based on Table 3.1. Figure 3.2 shows how the FTB model appears after macros *rcv(i)* and *ctu(i,o)* have been expanded.

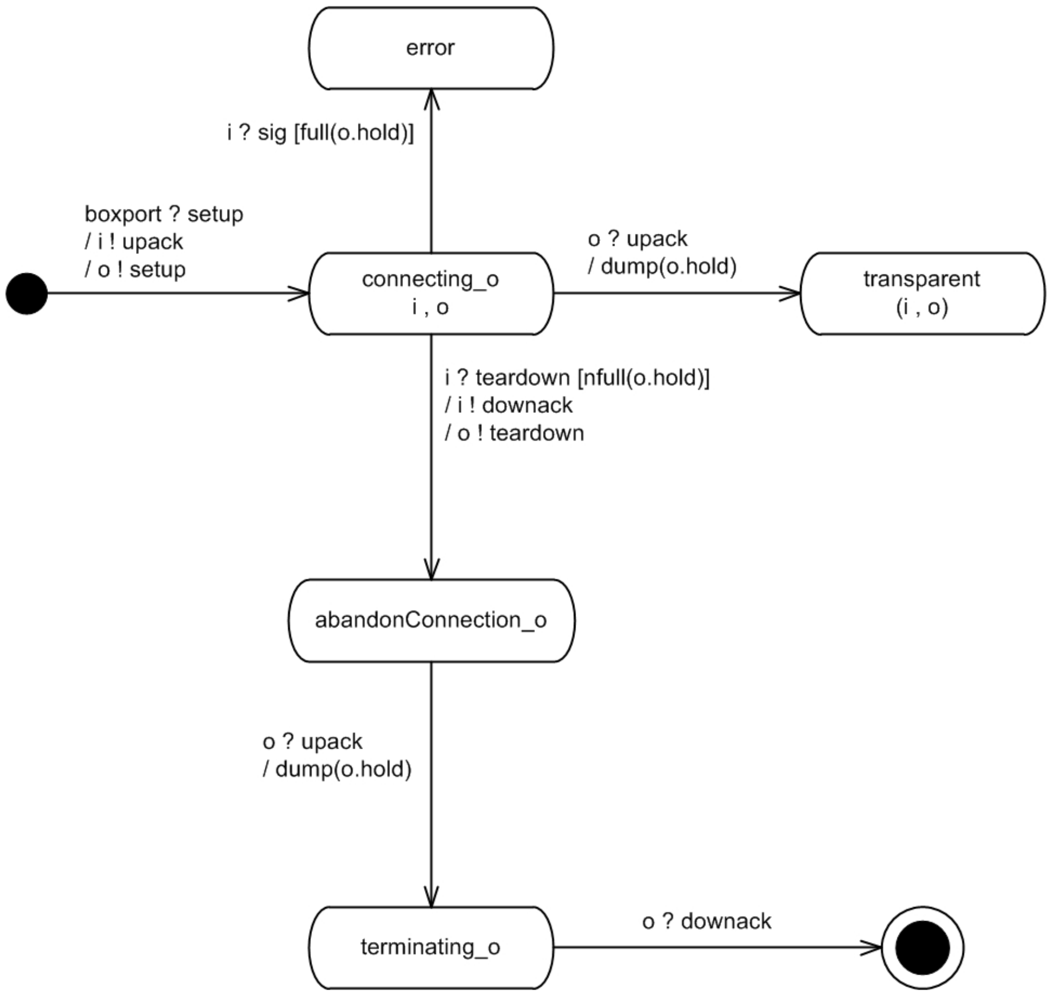


Figure 3.2: FTB - Explicated Specification (Step 1)

3.3.2 Step 2 - Call Termination

Implicit in every signal-linked state is the possibility that one of the signal-linked calls will end because the remote party of that call hangs up. Therefore, in every signal-linked state, our explication program adds an outgoing transition for each active call `c` triggered by event *gone(c)* to reflect the case that a *teardown* signal is received on that call. The action on each of these new transitions is to terminate the other signal-linked call. Thus from state *transparent* of FTB (with active calls `i` and `o`), there are two exiting transitions, *gone(i) / end(o)* and *gone(o) / end(i)*, each representing the possibility of the receipt of a *teardown* signal by one active call and the termination of the other active call.

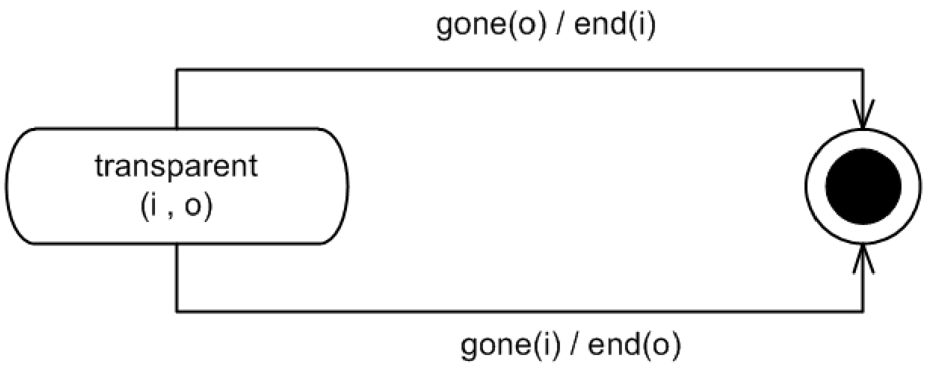


Figure 3.3: FTB - Explicated Specification (Step 2)

Figure 3.3 shows the two outgoing transitions from state *Transparent*. With respect to FTB, calls `i` and `o` are the only two calls in the signal-linked state *transparent*. Hence the destination of these two transitions is state *final*. However, if other calls are present, then the destination state will be different and will depend on the state of the other calls. For example, states *trying*, *ending_r*, and *confirming* of Answer Confirm's explicated specification, Figure 5.9, each have a pair of signal-linked calls and a third call. The destination states are different depending on the third call.

Figure 3.4 shows the explicated FTB model after executing Step 2 and expanding the introduced macros. The states and transitions introduced in Step 1 are shown in gray, and the new states and transitions introduced by Step 2 are shown in black. (The self-transitions, shown as dashed lines, will be inserted in Step 3.)

3.3.3 Identifying Common States and State Names

Both steps 1 and 2 introduce new states, and sometimes "new" states are equivalent to existing states in the model. For example, state *terminating_o* was introduced by a macro expansion in Step 1 of the explication process and was introduced again in Step 2 as a new

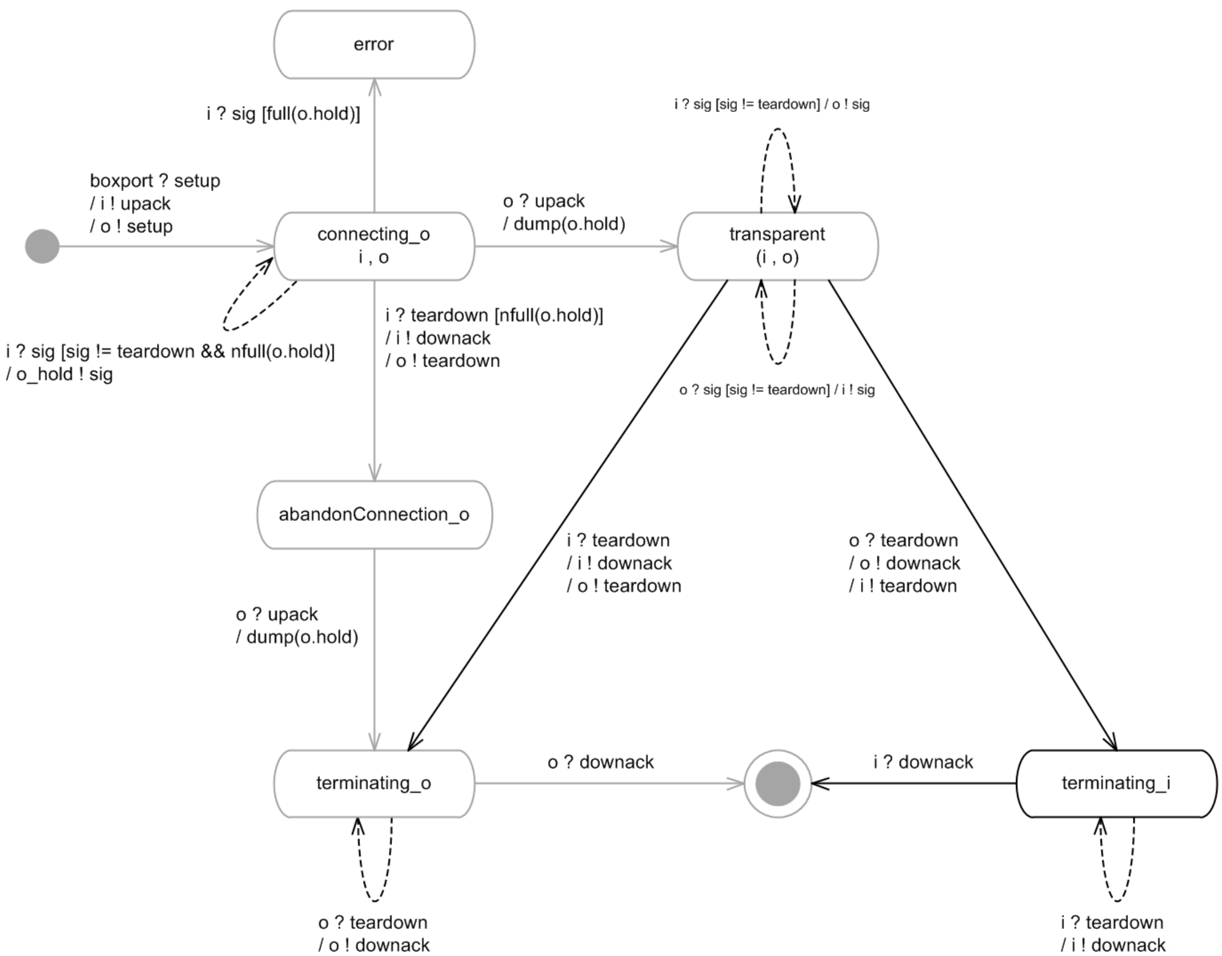


Figure 3.4: FTB - Explicated Specification

state that models the termination of a call. To detect when new states are equivalent to existing states, we annotate each state with four sets of call variables - `active`, `requested`, `abandoned`, and `terminating`. We use these four sets as follows:

- Set `active` stores the calls that are active (i.e., whose setup is complete) in the corresponding state.
- Set `requested` stores calls whose setup is in progress (i.e., the calls for which a *setup* signal has been issued, but for which the *upack* signal has not yet been received).
- Set `abandoned` stores those calls that were aborted in the process of being requested

(i.e., the calls where the caller hangs up before the *upack* signal was received).

- Set `terminating` stores the calls that are in the process of being dismantled (i.e., the calls for which a *teardown* signal has been issued, but for which the *downack* signal has not yet been received).

Our program uses these sets to assign names to states generated in the explication process. For example, if a single call is waiting for a *downack* acknowledgement (i.e., only one call `c` in set `terminating`), such a states is named *terminating_c*. If a half-established call (call `c`) is torn down (because the caller hung up), the resulting intermediate state is named *abandonConnection_c*. Whenever a request to terminate one call and set up another call is part of the same transition (i.e., set `requested` and set `terminating` are updated in the same transition), the resulting (intermediate) state is named *switching*. The source and destination state names of the transitions from the original BoxTalk specification remain unchanged.

We also use these call sets (in conjunction with the default new-state names) to identify common states created as a result of explicating different macros. This assists us in avoiding duplicate states. For example, consider state *terminating_o*. The first instance of state *terminating_o* is created while expanding macro *ctu()*. From state *connecting_o* to state *terminating_o*, call variable `o` is moved from set `requested` to set `abandoned` to set `terminating`. The second instance of state *terminating_o* is reached directly from state *transparent* when expanding macro *end()*. Call variable `o` is moved from set `active` to set `terminating`. Based on the generated name of the state (i.e., *terminating_o*) as well as the set contents (i.e., call variable `o` in set `terminating`), our program identifies the two *terminating_o* states to be the same.

3.3.4 Step 3 - Self Transitions

This step introduces the self transitions in *signal-linked* states, *connecting* states, and *terminating* states.

The default behaviour of features in a *signal-linked* state is to forward every signal, except the *teardown* signal or any signal that explicitly triggers an exiting transition, that it receives on one call to the signal-linked call. To support analysis, we explicate this behaviour as actions on self-transitions. The state *transparent* of the FTB has two such new transitions.

Connecting states display similar behaviour, except that the signals received on the established internal call are forwarded to the *hold* queue to be stored until the to-be signal-linked call is established. This self-transition has an additional guard that checks whether the *hold* queue has overflowed. The *connecting* state of FTB feature has one such new transition.

In the *terminating* states, it is possible for both ends of a call to initiate the call's teardown at (nearly) the same time. Thus, it is possible in a *terminating* state that a *teardown* signal is received on a call that is already half torn down. As per DFC protocol, the feature responds with a *downack* signal. States *terminating_i* and *terminating_o* in the explicated FTB specification both have such self transitions.

This concludes our discussion of explication of free BoxTalk features with the example of FTB.

3.4 Explicating BoxTalk - Bound Features

In this section, we explain our explication of bound features using the example of the Bound Transparent Box (BTB). Figure 3.5 shows the original BTB specification. BTB is a simple bound feature. It is used to model signal linkage between two calls.

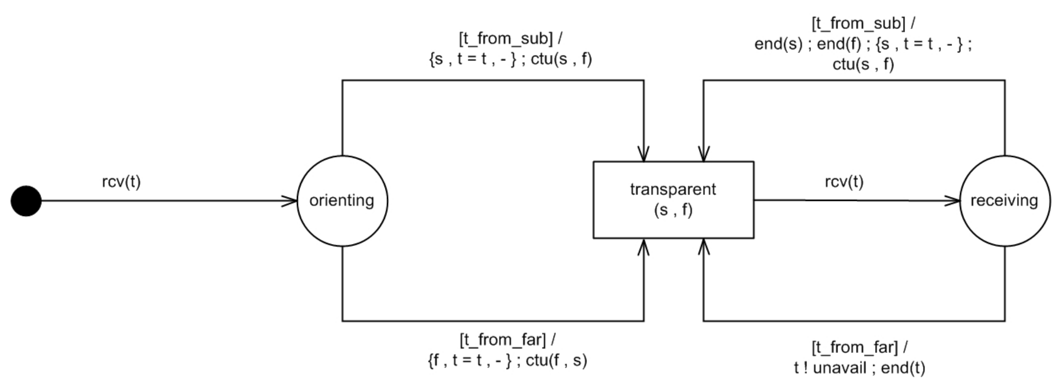


Figure 3.5: BTB - Original Specification

BTB is invoked when a new *setup* signal is received for a call, initially assigned to call variable `t`. In state *orienting*, the source of the call is tested to determine if the call is from the subscriber. If it is, then call variable `s` (associated with the subscriber) is assigned to the call, and the call that continues the usage is assigned call variable `f` (associated with the far party). The call assignments are reverse if call `t` is not from the subscriber. In state *transparent*, if BTB receives a new *setup* signal, the behaviour depends on whether the setup request is from the subscriber. If so, then the bound feature accepts the new call and tears down all old calls (see the top transition from the *receiving* state to the *transparent* state); otherwise, the new request is rejected (see the bottom transition from the *receiving* state to the *transparent* state). In the later case, the signal sequence *upack*, *unavail*, and *teardown* is sent.

Our explication process of a bound feature is a four-step process:

1. The first step expands all macros explicitly present in the original specification. This step is the same as the first step for free features. If a call terminate in this step, a post-processing machine is constructed to complete the termination of this call (i.e., to wait for the receipt of an appropriate *downack* signal).

2. The second step handles feature *"termination"*. A bound feature never terminates. Instead, when its calls terminate, the feature transitions to its *"initial"* state where it waits to be connected into the next usage. In fact, the feature transitions to the *initial* state once it is known that all of its current calls are terminating - but before the termination of its calls is complete.

3. A bound feature receives all *setup* signals destined for its subscriber. As such, it is possible for a bound feature to receive a *setup* signal for a new call while in the middle of another call. This step augments the feature model to include the receipt of and reaction to *setup* signals received while the feature is in a stable state.

4. The final step handles self transitions and is the same as that for free features.

Since some of these steps are the same as steps in the process to explicate free features, we explain in detail only steps two and three, which are unique to bound features.

3.4.1 Step 1 -Macro Expansion

This step is the same as that for the free features, as macros are expanded in a similar fashion. The only difference is the explication of the macro *end()*. Figure 3.6 shows the BTB model after executing Step 1.

The tear down of calls in bound features should be instantaneous, i.e., as soon as the *teardown* signal is issued, the terminating calls should be immediately ready to be included in the next usage involving the subscriber. However, the tear down of any calls involves a sequence of signals that are not instantaneous. For example, in BTB, when a *teardown* signal is sent on call variable `t`, this call variable is expected to be immediately available to

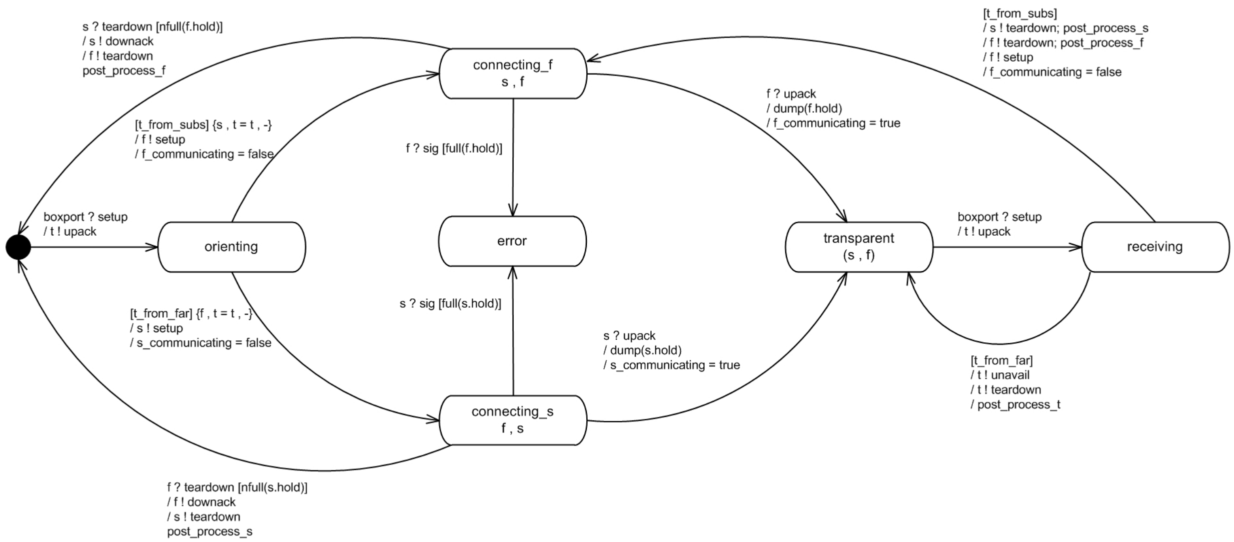


Figure 3.6: BTB - Explicated Specification (Step 1)

represent a new call. The task of completing the teardown of the old call associated with the variable `t` is delegated to a "post-processing machine". The post-processing machine executes in parallel with the feature's main machine, and its sole purpose is to complete the teardown process of terminating calls. That way, the main machine can set up a new call or a usage, while the post-processing machine tears down the old ones asynchronously. The number of post-processing machines is equal to the number of call variables in the feature.

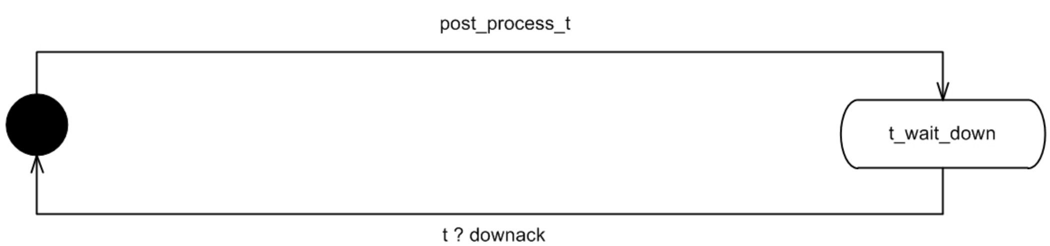


Figure 3.7: BTB - Post Processing Machine (Type 1)

Figure 3.7 and Figure 3.8 show two forms of post-processing machines for BTB. The post-processing machine of Figure 3.7 covers the case where the call to be terminated (was fully set up and) is waiting for acknowledgement *downack*. As the terminating call only needs a

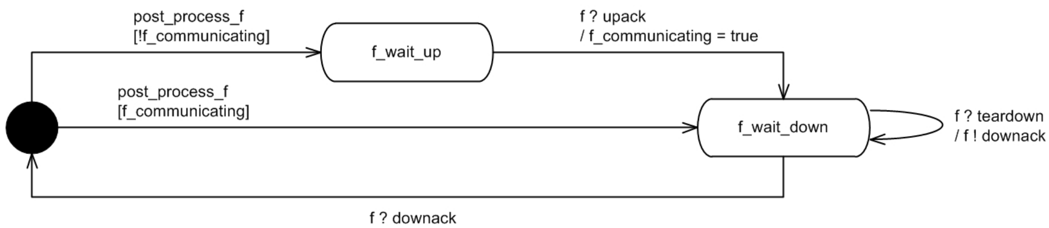


Figure 3.8: BTB - Post Processing Machine (Type 2)

downack acknowledgement, the intermediate state generated is named *c_wait_down*, where `c` is the terminating call. The post-processing machine of Figure 3.8 is used for those calls that might not be fully set up when they are terminated. In this case, the machine first transitions to the associated *c_wait_up* state and with the receipt of an acknowledgement *upack*, it transitions to the associated *c_wait_down* state. The post-processing machine for call `s` in BTB is similar to that for call `f`, shown in Figure 3.8. Boolean variable *c_communicating* is introduced in the feature model to keep track of whether a call `c` would require an *upack* signal if it were suddenly terminated. In BTB, the feature machine tracks the status of call variables `s` and `f` using their communicating variables; and the respective post-processing machines use the values of those variables to determine whether they wait for an *upack* acknowledgement.

As with FTB, in the *connecting_c* state, the half-established call may terminate if the party attached to the other end of the call hangs up. However, with BTB (and other bound features), the feature transitions to the *initial* state instead of to the *terminating* state as in FTB. This is because there is only one instance of each bound feature per subscriber and that one feature instance must be involved in any usage involving the subscriber. That is, the same bound feature instance is reused in each usage, rather than the feature instance terminating at the end of one usage and a new one instantiating with the next usage; hence, a terminating bound feature transitions to the *initial* state. In fact, this transition should happen as soon as all calls end with *gone()* or *end()* macros from any state (leading to feature termination), so that the feature instance is immediately ready to participate in another usage.

3.4.2 Step 2 - Call Termination

In state *transparent* of BTB, calls `s` and `f` are signal-linked. As in FTB, the receipt of a *teardown* signal on either call initiates the termination of the other signal-linked call. As discussed in the previous section, the feature activates post-processing machines to complete the termination of the calls. Figure 3.9 shows the partially explicated BTB model after Step 2 is executed. New transitions introduced in Step 2 are shown in black, old states and transitions of Step 1 are shown in gray.

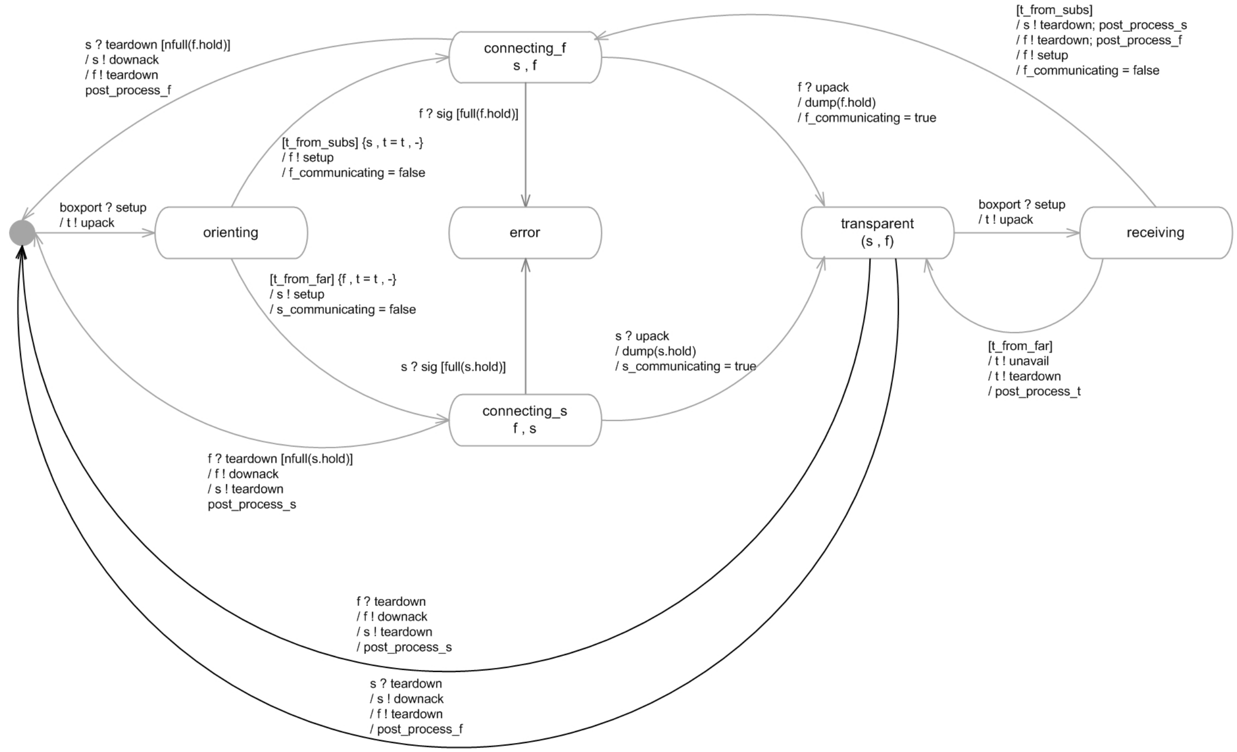


Figure 3.9: BTB - Explicated Specification (Step 2)

3.4.3 Step 3 - Setup Signals

A BoxTalk specification says how the feature should behave if a *setup* request is received in any state in the original specification. But what if a *setup* request is received in one of the states that is introduced as part of the explication process? States *connecting_s* and *connecting_f* are two such states in which a new *setup* request can be received. This step introduces two new states, *deciding_1* and *deciding_2*, which mimic the behaviour of the state *receiving* in BTB: if the new call request is issued by the subscriber, the box tears down all old calls and accepts the new call; otherwise the box rejects the new call by sending the signal *unavail*. Figure 3.10 shows the addition of these two states (shown in black) to the existing model (shown in gray). The self-transitions that will be inserted in Step 4 are shown as dashed lines. New transitions (black colour) and self-transitions are also labelled with a slightly larger font, for easier reading.

3.4.4 Step 4 - Self Transitions

This step is similar to that for free features. The exception is that bound features do not have terminating states as the feature transitions to the *initial* state whenever all of its calls are terminated, allowing the feature to participate in a new usage immediately. Therefore, our program introduces self-transitions only to the *connecting_c* states and all the signal-linked states.

This concludes our discussion of explication of bound BoxTalk features with the example of BTB.

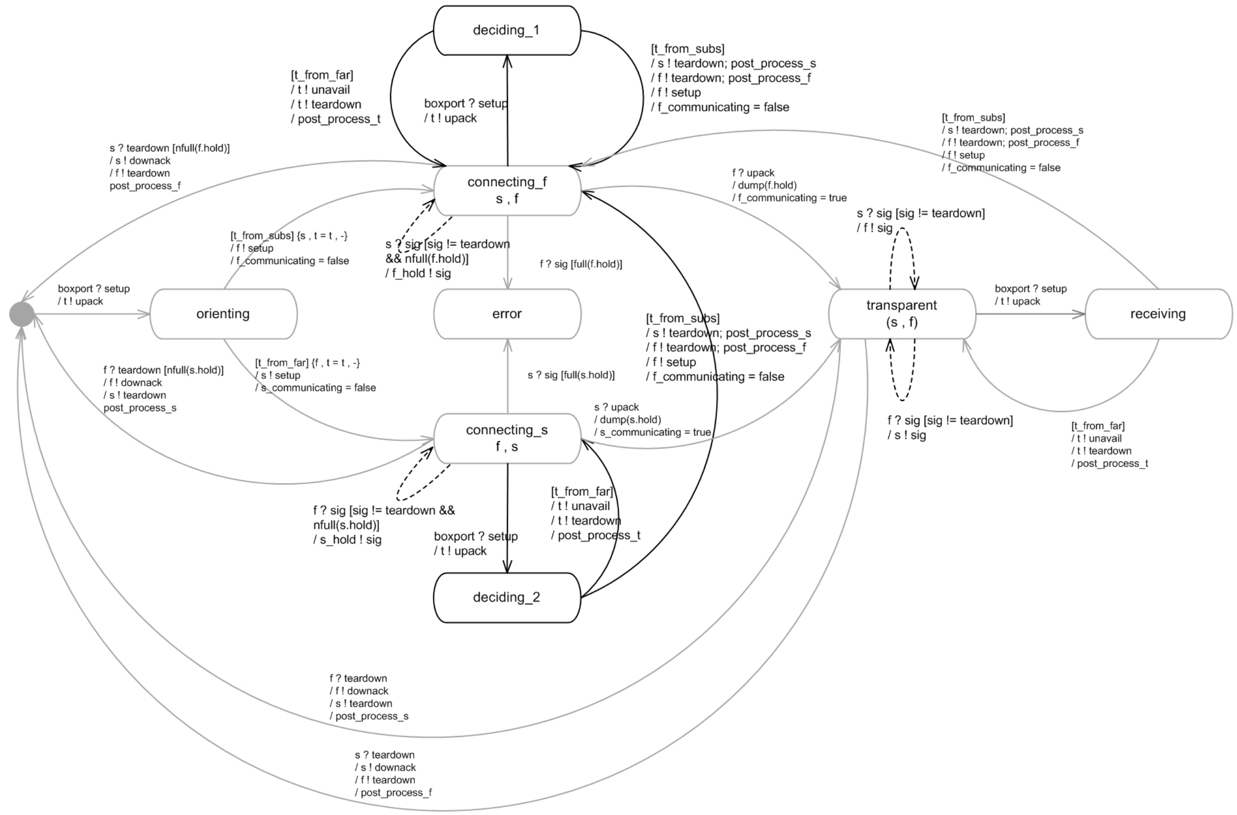


Figure 3.10: BTB - Explicated Specification

Chapter 4

Mapping Explicated BoxTalk to Promela

As part of our thesis work, we have automated the process of generating executable Promela models from explicated BoxTalk features. Chapter 2 introduced the target model checker SPIN and its input language Promela. In this chapter, we present the structure of our generated models along with our translation process by using the examples of free and bound features.

4.1 Promela Models of Features

The generated Promela model analyzes the behaviour of a single BoxTalk feature in isolation, running in the DFC environment (i.e., receiving and sending DFC signals on ports). Our generated Promela models have one `active` (main) process, which represents the feature of interest. Another process models the environment as an `active` process that communicates with the main process via rendezvous communication channels. For each port in a BoxTalk feature specification, there are two unidirectional channels, a `port_in` message channel that passes signals from the environment process to the feature process, and a `port_out` rendezvous channel that passes signals from the feature process to the environment process. There is also a channel `box_in`, which is used to send *setup* signals to the feature process.

Free and bound feature models have different architectures. Figure 4.1 displays the architecture of our free feature models and Figure 4.2 displays the architecture of bound feature models.

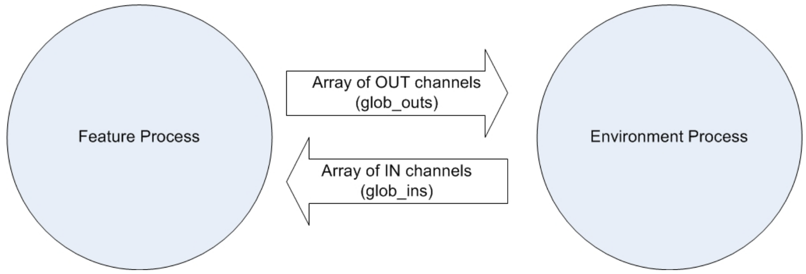


Figure 4.1: Promela Architecture - Free Boxes - adapted from [12]

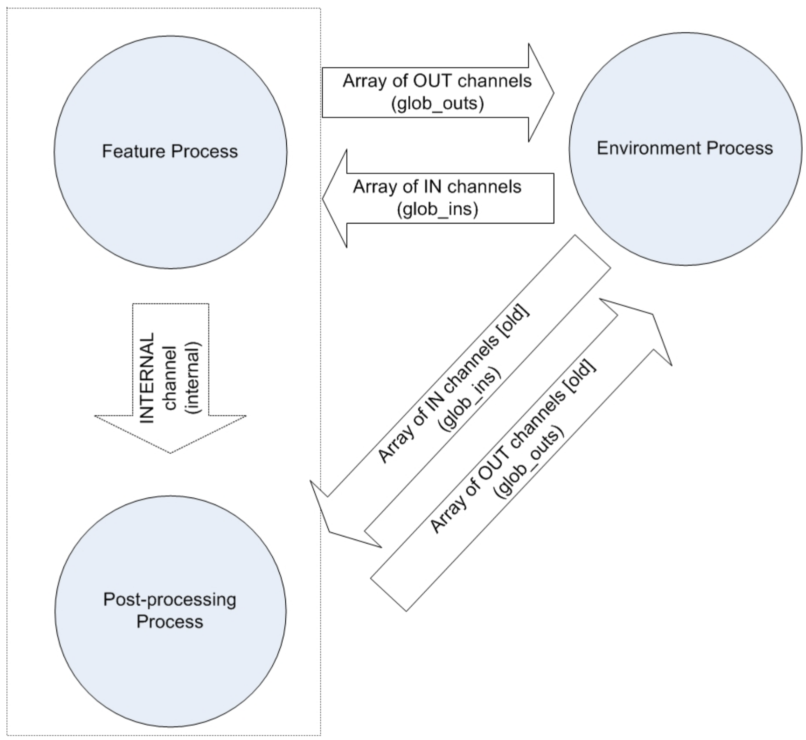


Figure 4.2: Promela Architecture - Bound boxes - adapted from [12]

A free feature model has only one main Promela process and one environment process, with arrays of zero-capacity `output` and `input` channels that pass messages to and from the environment process, respectively. A bound feature model has additional `active` processes that model the post-processing machines. The number of post-processing machine processes in a Promela model corresponds to the number of call variables in the bound feature. There are unidirectional channels which send signals from the main feature process to the post-processing-machine processes. The job of the post-processing process is to complete the teardown of calls that are terminating. The environment process also sends acknowledgements to post-processing machine process(es) for terminating calls and vice versa.

There is a limitation to what our Promela models can handle. Timer variables discussed in Chapter 5 is one such limitation. Certain features use timer variables to terminate the feature using timeouts. Our translator ignores conditions and actions on timer variables in feature transitions.

4.1.1 Generating a Promela Model from a Free BoxTalk Feature

Our generated Promela models are composed of three main parts:

- Type definitions and global variable declarations
- Inline functions
- Process definitions

We explain each part for a free feature using Free Transparent Box (FTB), which was introduced in Section 3.2, as an example.

To reduce the number of passes through the input (explicated BoxTalk model) while building the corresponding Promela model, we store intermediate results in five separate files:

- Type definitions, global variables, and inline functions `dump(c1 , c2)` and `reset()`
- Inline functions `en_events(n)` and `en_cond(n)`
- Inline function `next_trans(n)`
- Feature process
- Environment process

At the end of the translation process, all of these files are concatenated together to form one single Promela model.

4.1.2 Type Definitions and Global Variable Declarations

The arrays of `input` and `output` channels and the shared variables are declared globally. Every model starts with a definition of signals, states, and user-defined types. There is an `mtype` declaration for the set of signals sent to and from the feature and another one for the states belonging to the feature. The declarations for the FTB model are as follows[1]:

```
7   mtype = { teardown , downack , other , setup , upack };

8   mtype = { initial , connecting_o , transparent , abandonConnectiono ,
    terminating_o ,
9                final , terminating_i , error };
```

All input channels, one for each feature port, are declared together in a single array. Since FTB has two feature ports plus *boxport*, the declaration of input channels is as follows:

```
26   chan glob_ins[3] = [0] of {mtype};
```

There is an analogous declaration for an array of output channels:

```
34   chan glob_outs[3] = [0] of {mtype};
```

There is a type definition for `Transition` that is the same for all features. It is as follows:

```
11   typedef Transition {
12       mtype dest;
13       chan in_chan;
14       bool en_flag = false;
15   };
```

Each transition has exactly one destination state of type `mtype` and receives an input signal on a specific input channel. The Boolean variable `en_flag` is an indication of whether the transition is enabled to be executed. Its value is set in the inline function `en_trans`.

A set of *global-monitor variables* are declared, which are used to verify certain properties. For example,

```
51   bool rcv_setup = false;
52   bool send_upack = false;
53   bool o_send_setup = false;
54   bool o_rcv_upack = false;
55   bool i_rcv_teardown = false;
56   bool i_send_downack = false;
57   bool o_send_teardown = false;...
```

[1]The numbers on the left indicate the line numbers of the model in Appendix A3

Such Boolean variables are updated (set to `true` or `false`) when the associated signal is sent to or received from the environment process (via rendezvous channels). We cannot express properties about signals sent on rendezvous channels (because we cannot query their contents as the channels have zero-capacity). Moreover, as we will see later in this chapter, we use Promela program labels to model feature states, and we cannot formulate properties over labels. Therefore, we declare monitor variables that record the occurrence of signal event and current states of processes, and we formulate properties over these Boolean variables. For the complete list of the global-monitor variables used in FTB, please refer to Appendix A3.

For FTB feature, the type definition for the set of input queues, `in_q`, is modelled as follows:

```
typedef in_q {
    byte box_in = 0;
    byte i_in = 1;
    byte o_in = 2;
    bool box_in_ready = true;
    bool i_in_ready = false;
    bool o_in_ready = false;
    byte selected
};
```

For each *input* channel X in `glob_ins[]`:

- there is a byte variable "X_in" that holds the index of that channel in `glob_ins[]`[2]
- there is a Boolean variable "X_in_ready" that indicates whether the feature is in a state that is ready to receive a signal on channel `X`

For free features, only the ready variable `box_in_ready` is `true` in the initial state; other ready variables become `true` only after the calls, `i` and `o`, are initiated. When more than one *input* channel is active (i.e., has incoming signals), the byte variable `selected` has the value of a randomly-selected input channel (set in function `reset()`) from among the channels that have incoming signals.

The type definition for the set of output queues, `out_q`, is analogous to the type definition of the set of input queues `in_q`.

For each *output* channel X in `glob_outs`:

[2]The advantage of using byte variables (`box_in`, `i`, `o`) instead of index numbers directly is described at the end of this section, after all type definitions have been introduced.

- there is a byte variable "*X*_out" that holds the index of that channel in `glob_outs[]`
- there is a hold queue for each channel that the feature initiates, which is used to store signals to be sent via that channel (until that call is fully established)

The `out_q` type definition for FTB model is as follows:

```
typedef out_q {
    byte box_out = 0; /* Never used, only declared for symmetry. */
    byte i_out = 1;
    byte o_out = 2;
    chan o_hold = [5] of {mtype};
};
```

A *snapshot* is an observable point in the execution state. The type definition for a snapshot includes the current state `cs`, the input queue `in_q`, and the output queue `out_q`:

```
typedef SnapShot {
  mtype cs;
  in_q inq;
  out_q out
};
```

Given these definitions, and given a *Snapshot* variable `ss`, we can write Promela expressions that reference communication channels in terms of BoxTalk names rather than explicit index numbers. For example, `glob_ins[ss.inq.i_in]` refers to the communication channel corresponding to call variable `i`, and is equivalent to `glob_ins[1]`.

4.1.3 Inline Functions

Promela inline functions are similar to C-style macros but do not introduce any overhead during verification. We use inline functions `dump()`, `reset()`, `en_events()`, `en_cond`, `en_trans`, and `next_trans` in our models. We only show parts of the code; for the entire feature model, please refer to Appendix A3.

The inline function `dump(c1 , c2)` is used to empty the contents of the *hold_queue* `c1` to channel `c2`.

```
inline dump(c1 , c2) {
    byte aSig;
    do
    ::c1 ? aSig -> c2 ! aSig;
    ::empty(c1) -> break;
```

```
        od
    };
```

In every *non-transient* state, the inline function `reset()` selects a random *input* channel from among the channels receiving a signal, and sets the byte variable `selected` of `in_q` to the selected channel. This function also resets all of the global-monitor variables to `false`. The definition of `reset()` for FTB is as follows:

```
inline reset() {
    rcv_setup = false;
    send_upack = false;
    o_send_setup = false;
    o_rcv_upack = false;
    i_rcv_teardown = false;
  ...

    if
    ::glob_ins[ss.inq.box_in] ? sig -> ss.inq.selected = ss.inq.box_in;
    ::glob_ins[ss.inq.i_in] ? sig -> ss.inq.selected = ss.inq.i_in;
    ::glob_ins[ss.inq.o_in] ? sig -> ss.inq.selected = ss.inq.o_in;
    fi
};
```

The inline function `en_events` checks if the selected *input* channel matches the event channel of the n^{th} transition `t[n]`.

Transitions exiting from *transient* states do not have in-channels, as transient states are non-responsive states. In these transitions, `en_events` is `true` by default. The `en_events(n)` function is as follows:

```
inline en_events(n) {
   glob_ins[ss.inq.selected] == t[n].in_chan;
};
```

The inline function `en_cond(n)` checks whether the guard condition of the n^{th} transition `t[n]` is `true`. `en_cond()` is `true` if the input signal matches the transition's triggering event. As part of this check, the function also checks whether the *hold queue* has reached its capacity when signals are written to the *hold queue* (lines 107 and 108).

In case of transitions exiting transient states, which do not read input signals, the guard predicate is evaluated. The environment process nondeterministically sets one of the guard predicates to `true`. Following is the code snippet of inline function `en_cond(n)` for FTB:

```
101 inline en_cond(n) {
102     if
103     ::(n == 0) && (sig == setup );
```

```
   ::(n == 1) && (sig == upack );
   ::(n == 2) && (sig == teardown );
   ::(n == 3) && ( sig != teardown && nfull(ss.out.o_hold) );
   ::(n == 4) && ( sig != teardown && full(ss.out.o_hold) );
  ...
  fi;
};
```

shows that transition 0 is triggered by the *setup* signal, etc.

A transition `t[n]` is enabled only when (1) its event queue is selected for reading, (2) the signal read matches the triggering event, and (3) the transition's other guard conditions hold. In case of transitions exiting *transient* states, the transition whose guard condition holds is enabled.

The inline function `en_trans()` uses the results from `en_events()` and `en_cond()` to determine whether a transition is enabled:

```
inline en_trans(n) {
   if
   ::en_events(n) ->
      if
      ::en_cond(n) -> t[n].en_flag = true;
      ::else -> t[n].en_flag = false;
      fi;
   ::else -> t[n].en_flag = false;
   fi;
};
```

The inline function `next_trans(n)` represents the execution of the enabled transition: the current state changes to the transition's destination state, output signals are sent on the *output* channels *glob_outs*, and variables (including the global monitor variables) are updated:

```
inline next_trans(n) {
  if
   ::(n == 0) ->    rcv_setup = true;
                      ss.inq.i_in_ready = true;
                      glob_outs[ss.out.i_out] ! upack;
                      send_upack = true;
                       glob_outs[ss.out.o_out] ! setup;
                      ss.inq.o_in_ready = true;
```

```
                        o_send_setup = true;
                        ss.cs = t[0].dest;

    ::(n == 1) ->
                        o_rcv_upack = true;
                        dump(ss.out.o_hold , glob_outs[ss.out.o_out]);
                        ss.cs = t[1].dest;
  ...
  fi;
};
```

4.1.4 Processes

Our generated Promela model includes two processes: a `feature process` and an `environment process`. Both of the processes are `active` and running at the start of a simulation of the model.

The feature process uses inline functions `reset()`, `en_trans()`, and `next_trans()` to model transitions. A typical state and its set of exiting transitions appear as follows:

```
connecting_o_state:
  atomic {
      reset();
      en_trans(1);
      en_trans(2);
      en_trans(3);
      en_trans(4);

      if
      ::t[1].en_flag -> next_trans(1); goto transparent_state;
      ::t[2].en_flag -> next_trans(2); goto abandonConnectiono_state;
      ::t[3].en_flag -> next_trans(3); goto connecting_o_state;
      ::t[4].en_flag -> next_trans(4); goto error_state;
      ::else -> goto connecting_o_state;

      fi;
}
```

where `connecting_o_state` is a Promela label for the *connecting_o* state. Labels in Promela models serve as targets of `goto` statements. Any statement or any control-flow construct can be preceded by a label. Label names must be unique in a model and cannot be the same as `mtype` names. Hence, state labels in our Promela models are the state names from the original BoxTalk specification appended with "`_state`".

The state transitions in BoxTalk are atomic and take place in one single step. Therefore, the state label is followed by an atomic block that reflects the set of possible exiting transitions as follows:

1. The execution step starts by reseting all of the global variables and randomly selecting an input queue to read from.

2. Next, inline function `en_trans()` determines which among the state's exiting transitions are enabled and sets their `en_flag` values to `true`.

3. Finally, the `if` selection construct nondeterministically selects and executes (via `next_trans()`) one of the enabled transitions, followed by a `goto` statement that transfers control to the transition's destination state.

The environment process models the environment of the feature: it produces all input signals that the feature can receive and consumes all signals that the feature can send.

The following code displays the environment process of FTB:

```
active proctype env() {
 mtype i_sigt ,o_sigt  , o_sigu ;



end:  do

      :: ss.inq.box_in_ready ->
          ss.inq.box_in_ready = false;
          glob_ins[ss.inq.box_in] ! setup;

      ::ss.inq.i_in_ready ->
        if
        :: glob_ins[ss.inq.i_in] ! teardown;
        :: glob_ins[ss.inq.i_in] ! other;
        fi unless {
           (i_sigt == teardown) ->
             glob_ins[ss.inq.i_in] ! downack;
             i_sigt = 0;
        }
      ::ss.inq.o_in_ready ->
        if
        :: glob_ins[ss.inq.o_in] ! teardown;
        :: glob_ins[ss.inq.o_in] ! other;
        fi unless {
```

```
387             if
388             ::(o_sigu == upack) ->
389              glob_ins[ss.inq.o_in] ! upack;
390              o_sigu = 0;
391             ::(o_sigt == teardown && o_sigu == 0) ->
392               glob_ins[ss.inq.o_in] ! downack;
393              o_sigt = 0;
394             fi;
395             }
396     od
397     unless {
398         if
399         ::atomic { glob_outs[ss.out.o_out] ? setup ->
400          o_sigu = upack;
401          }
402         ::glob_outs[ss.out.i_out] ? upack;
403         ::glob_outs[ss.out.i_out] ? downack;
404         ::atomic { glob_outs[ss.out.i_out] ? teardown ->
405          i_sigt = teardown;
406          }
407         ::glob_outs[ss.out.i_out] ? other;
408         ::atomic { glob_outs[ss.out.o_out] ? teardown ->
409          o_sigt = teardown;
410          }
411         ::glob_outs[ss.out.o_out] ? downack;
412         ::glob_outs[ss.out.o_out] ? other;
413         fi;
414     }
415     goto end;
416 }
```

1. The `do` construct models the sending of input signals. The "ready" clauses identify which ports of the feature are expecting input from the environment process. One ready port is nondeterministically chosen and an appropriate signal is sent on the chosen port. For example, in FTB, if `ss.in.i_in_ready` is `true` and a *teardown* signal is received from the feature, then a *downack* signal has been sent on the input channel (lines 373, 378 - 380).

2. The `if` construct (on line 398 following the `unless` keyword) models all aspects of the environment process receiving feature output. If there are multiple output signals, one signal is chosen nondeterministically.

3. The `unless` construct (on line 397) is used to prioritize the receiving of signals from the feature over the sending of new input signals to the feature.

4. The environment process should never end; we use an end state label “`end`” to mark it as a valid end state. End-state labels are any labels that start with *end*.

This concludes our discussion of generating a Promela model from a free BoxTalk feature.

4.1.5 Generating a Promela Model from a Bound Feature

The translation of a bound BoxTalk feature into Promela is similar to the translation of a free BoxTalk feature into Promela. Thus, we explain in this subsection only those aspects of the translation that are unique to bound features. We use Bound Transparent Box (BTB), which was introduced in Section 3.3, as a running example. The Promela model for BTB is presented in Appendix A4.

As explained in Section 3.3, the explicated BoxTalk model of a bound feature has post-processing machines that model the termination of calls. The post-processing machines run in parallel with the feature machine and allows the feature machine to handle new calls while old calls are being torn down. A bound feature is modeled in Promela as two active processes: a main feature process and an environment process; and a number of post-processing processes (also active processes), one per call in a feature. The feature process communicates with the post-processing processes via `internal` channels.

Separate call variables are used by the main feature process and the post-processing processes. Channels to the main feature process,“*X*_in”, represent BoxTalk call variables for connecting and active calls *X*. Channels to the post-processing processes,“old_*X*_in”, represent BoxTalk call variables for calls that are being terminated. The type definition for the set of input queues, `in_q`, for BTB is as follows:

```
typedef in_q {
    byte box_in = 0;
    byte old_t_in = 1;
    byte old_s_in = 2;
    byte old_f_in = 3;
    byte t_in = 4;
    byte s_in = 5;
    byte f_in = 6;
    bool box_in_ready = true;
    bool old_t_in_ready = false;
    bool old_s_in_ready = false;
    bool old_f_in_ready = false;
    bool t_in_ready = false;
    bool s_in_ready = false;
```

```
    bool f_in_ready = false;
    byte selected
};
```

We declare separate call variables (indexes into communication channels) for connecting and active calls (used by the main feature process) and for terminating calls (used by the post-processing processes) in the type definition of set of output queues, `out_q` as well. The type definition of `out_q` is as follows:

```
typedef out_q {
    byte box_out = 0; /* Never used, only declared. */
    byte old_t_out = 1;
    byte old_s_out = 2;
    byte old_f_out = 3;
    byte t_out = 4;
    byte s_out = 5;
    byte f_out = 6;
    chan f_hold = [1] of {mtype};
    chan s_hold = [1] of {mtype};
};
```

Bound features have an additional set of channels, `inter_q`, to represent the internal channels between the feature's main process and its post-processing processes. We use a rendezvous channel(s) for this purpose to make sure that the main process does not terminate another call before the post-processing process has finished terminating past calls. We modified our Promela model of bound features to have multiple post-processing machine processes, one per each call of a feature. This ensures that requests to terminate multiple calls in a single transition are not dropped. For example, in BTB feature, from state *requesting* to state *connecting_f* where calls `s` and `f` are terminated. The Promela model of BTB feature has three post-processing machine processes. For BTB feature, the type definition for `inter_q` is defined as follows:

```
typedef internal {
  chan internal_t = [0] of {mtype};
  chan internal_s = [0] of {mtype};
  chan internal_f = [0] of {mtype};
};
```

BTB has three additional `reset_pp_`*X*`()` inline function that are used in the post-processing processes (for call *X*) in place of the `reset()` function:

```
inline reset_pp_t() {
  glob_ins[ss.inq.old_t_in] ? sig -> ss.inq.selected = ss.inq.old_t_in
};
```

```

inline reset_pp_s() {
    glob_ins[ss.inq.old_s_in] ? sig -> ss.inq.selected = ss.inq.old_s_in
};

inline reset_pp_f() {
    glob_ins[ss.inq.old_f_in] ? sig -> ss.inq.selected = ss.inq.old_f_in
};
```

Bound features also include a variable *X_communicating* to determine the acknowledgements required by a terminating call. If the call to be terminated was fully set up in the main machine, it only requires a *downack* acknowledgement. If, however, a call is to be terminated before it is fully set up, then it requires two acknowledgements: first an *upack* acknowledgement and then a *downack*. Boolean variable *X_communicating* is used to keep track of whether call `X` requires an *upack* acknowledgement in the post-processing machine. Whenever a *setup* is issued for any call `X`, it is initialized to *false*, and reset to *true* with the receipt of an acknowledgement *upack* in the main machine.

This concludes our discussion of generating Promela models from bound BoxTalk features.

4.2 Promela Model Comparisons

In this section, we present a brief comparison of our mechanically generated Promela models to the hand-crafted Promela models in Yuan Peng's [12] and Alma L. Juarez Dominguez' [4] theses.

The goal of Yuan Peng's [12] work was to devise a mapping from BoxTalk specifications to Promela models. The goal of our work was to fully automate the translation of BoxTalk specifications to Promela models. We used Promela models from Yuan Peng's thesis as reference models for our translation and hence there is a high correlation between our Promela models and her hand-translated Promela models.

Similar to hand-translated Promela models of Yuan Peng, our mechanically generated models are composed of type definitions and global variables, two `active` processes – one feature process and one environment process, and inline functions. The bound features also include post-processing machines to handle completion of terminating calls.

The feature process and the environment process communicate with each other (i.e., send signals) over rendezvous channels. Since rendezvous channels cannot store messages, we cannot formulate properties of signals being sent over such channels. Therefore, similar to

Yuan Peng's models, our models incorporate global monitor variables to record signaling events, and we use these variables directly in our properties.

The environment process models the environment of the feature and the feature process models the transitions of the explicated BoxTalk models. We use control-flow labels to model states of the explicated BoxTalk models. These labels also serve as targets to `goto` statements to reflect state transitions. In the feature process, followed by each control-flow label[3], there is an atomic block that models state transitions. In this atomic block, the inline function *reset()*[4] resets all global monitor variables and selects a random channel from among channels receiving input. Then inline function *en_trans()* checks whether the selected channel matches the transitions event channel, and if a match is found, it checks whether the input signal matches the transitions triggering event. If this condition also matches, the inline function *next_trans()* executes the transition by sending output signals on output channels and updating the destination of the transition. The `goto` statement transfers the program control to the label corresponding to the destination state of the transition.

Despite these similarities, there are subtle differences between our Promela models and Yuan Peng's Promela models. The type definition of Transition in her Promela models includes variables *out_chan* of type `chan`. In the inline function *next_trans*, the output signals are sent on channels *out_chan*, and these channels are matched with the global output channels (*glob_outs*) in the feature process. In our Promela models, we send output signals directly on the global output channels and our type definition of Transition does not include variables *out_chan*.

She used one single post-processing machine process in bound features to handle completion of terminating calls. Bound feature's main machine communicates with its post-processing machine via rendezvous channels. With a single post-processing machine approach, whenever the feature's main machine will end multiple calls at the same time, the request to terminate the first call will be received by the post-processing machine, and all other subsequent requests will be discarded. To overcome this problem, the number of post-processing machine processes in our Promela models is equal to the number of calls in the bound features.

Now we compare our approach with Alma L. Juarez Dominguez' thesis [4]. Alma L. Juarez Dominguez presented a compositional reasoning method consisting of model checking, language containment, and theorem proving to verify DFC compliance properties over chains on unknown number of connected DFC features. She used the model checker SPIN to verify

[3] *Final* state and *error* state labels are exceptions as they are final states of (series of) transitions.
[4] Atomic blocks following transient states do not have *reset()* function calls.

expected input/output properties and call protocol properties. The expected input/output properties are specified as LTL invariants and express that the feature interacting with an environment of neighbouring features receives only the signals it expects from the environment, and sends only the signals expected by the environment. The call protocol properties are also expressed in LTL and state that signals sent from one end of a call segment eventually reach the other end (end-to-end path properties).

Her Promela models consist of the entire DFC architecture for constructing usages. Specifically, her models include interface box processes (i.e., Caller and Callee processes), all of the feature processes[5], and router processes, one per each feature, plus a generic router process. Initially, one instance of the Caller process is created which runs the generic user router process and forwards the *setup* signal to the router. Based on the user's subscriptions and feature precedence, the router process initializes the *next* feature process in the usage and forwards the *setup* signal further. The feature precedence is hard-coded into the router processes and the user's subscriptions are modelled using SPIN's nondeterminism. The feature process sends acknowledgement *upack* directly to the Caller and also runs an instance of its feature-specific router process, which initializes the feature process corresponding to the next feature in the usage. In this way, the usage is dynamically assembled from Caller to Callee via the features the user subscribes to. Hence, she uses processes which are not declared `active`, and uses `init` for process initializations to dynamically create usages. In contrast, we analyze the behaviours of individual BoxTalk features running in DFC environment, and we use `active` feature and environment processes. There are certain similarities between our approaches as well. Similar to our approach, she also uses control-flow labels to model states, and models state transitions with atomic blocks following these labels. She also uses Boolean variables to record signalling events.

Instead of verifying every feature in the environment of every other feature, she developed an abstract port model that captures the most general port behaviour that serves as an abstract environment. She verified each individual feature in the abstract environment and proved that every feature's port obeys the abstract port model. The abstract and concrete port models are described in terms of state transitions and consist of a source state, a destination state, and the triggering event.

She verified call protocol properties on fixed DFC segments and used theorem prover HOL to connect the individual proofs by induction to prove that DFC call protocol properties hold over segments of unknown number of connected DFC features.

[5]She verified Free Transparent Feature (FTF), Call Forwarding (CF), Originating Call Screening (OCS), and Call Waiting (CW) features.

Chapter 5

Case Studies

In this chapter, we evaluate our translator by applying it to a set of BoxTalk features. The translated Promela models are verified against a set of properties that we discuss at the end of this chapter. The case study consists of the following BoxTalk features:

- Error Interface (EI): Used by the router to handle routing errors.
- Receive Voice Mail (RVM): Allows the caller to record a voice message when the callee does not answer the call.
- Black Phone Interface (BPI): Acts as an interface between the DFC protocol and a telephone.
- Answer Confirm (AC): Ensures that the a successfully established usage has reached a human callee.
- Quiet Time (QT): Subscribed to by people who do not wish to be disturbed (i.e., called), QT offers the callers options to choose from.
- Parallel Find Me (PFM): Tries to direct a phone call to its subscriber's current location by trying multiple locations in parallel.
- Sequential Find Me (SFM): Similar to PFM, but SFM tries multiple locations sequentially.

For space reasons, we include Promela models of only EI, RVM, and BPI features in Appendix A5, A6, and A7, respectively.

5.1 Error Interface

The Error Interface (EI) feature is a free feature that is used by the router to handle call requests to invalid addresses. If a call setup fails because the target address does not exist, then the router routes the usage to this feature. Figure 5.1 shows the original specification of the EI feature.

Figure 5.1: EI - Original BoxTalk Specification

Specifically, the EI feature accepts the call, sends a signal *unknown* upstream, and then immediately tears down the call. Figure 5.2 shows the explicated EI specification:

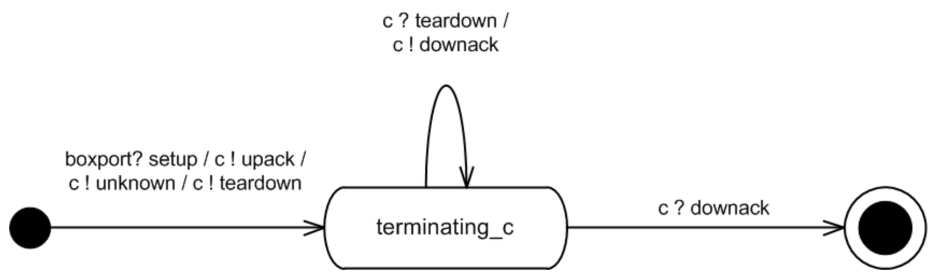


Figure 5.2: EI - Explicated Specification

The *rcv(c)* macro in the original specification is expanded in the explicated specification to *boxport ? setup / c ! upack*. The *end(c)* macro in the original specification is expanded to *c ! teardown* and a new destination state, *terminating_c*, in which the feature waits for a *downack* signal. There is a possibility in the *terminating_c* state that a *teardown* signal is received when call `c` is already half torn-down in which case the feature responds with a *downack* signal. With the receipt of a *downack* signal in the *terminating_c* state, the feature transitions to state *final*.

Appendix A5 contains the Promela model of the explicated EI specification.

5.2 Receive Voice Mail

Receive Voice Mail (RVM) is a target-zone feature that allows a caller to record a voice message when the subscriber (i.e., the callee) refuses or is unable to accept a call. Figure 5.3 shows the original BoxTalk specification of the RVM feature.

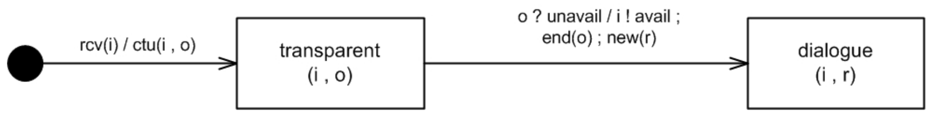


Figure 5.3: RVM - Original BoxTalk Specification

The transition to the *transparent* state is the same as that in FTB: the feature is added to the usage and assigned to call variable `i` and the feature continues the usage via call `o`. If call `o` receives an *unavail* signal from downstream, it indicates that the callee is not available. The feature absorbs this signal (i.e., the signal *unavail* is not propagated upstream), and sends an *avail* signal upstream instead. (Sending signal *avail* upstream encourages the caller to remain in the usage.) The feature then tears down call `o` and initiates a call to the Voice Message Service, which is assigned to call variable `r`. On completion of this call, the feature then transitions to the state *dialogue*, in which the caller and the Voice Message Service are signal-linked. When the caller finishes sending a message, the caller may hang up, which causes the feature to transition to the *final* state.

Figure 5.4 shows the explicated specification of the RVM feature. The left-hand side of the model (up to state *transparent*) captures the behaviour of the feature when the called party is available. This behaviour is equivalent to the functionality of the FTB feature. The right-hand side of the model expresses the behaviour of the feature when the called party is not available.

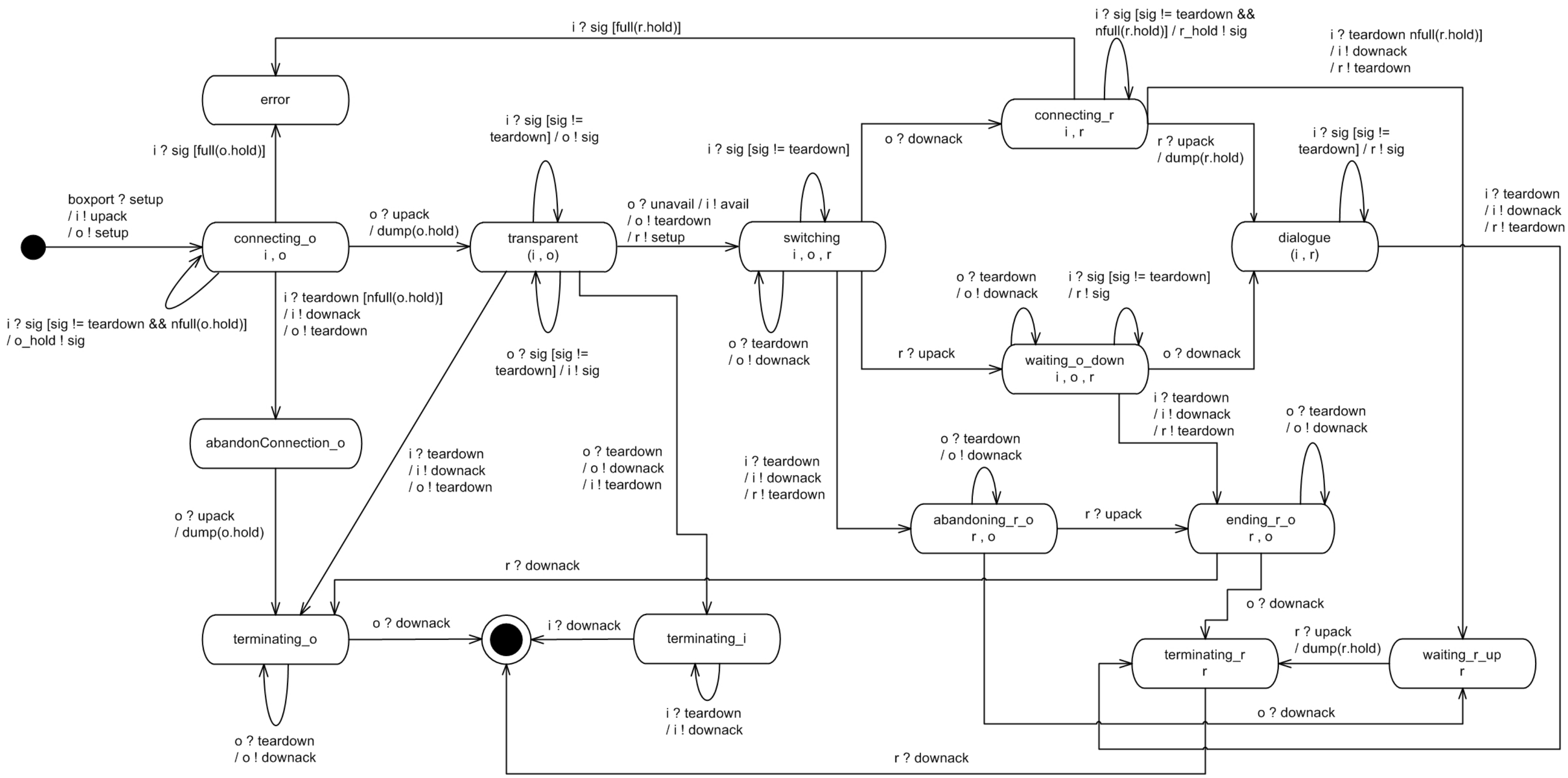


Figure 5.4: RVM - Explicated Specification

State *switching* is an intermediate state that represents the situation in which the called party is not available and the feature has terminated call `o` and has initiated call `r` to the Voice Message Service. The name *switching* is assigned to this state by our program's state naming scheme which was introduced in Chapter 3, Section 3.2.3. In state *switching*, call `o` is waiting for a *downack* signal to complete its termination, and call `r` is waiting for an *upack* signal to complete its connection. These acknowledgements can be received in any order, and therefore there are two transitions exiting this state, each modelling the receipt of acknowledgements, but in different order. A third exiting transition models the case in which caller `i` hangs up. If the caller hangs up before call `r` receives an acknowledgement *upack*, the DFC protocol requires that acknowledgements *upack* and *downack* be received on call `r` for the call to be terminated.

State *dialogue* is a signal-linked state in which calls `i` and `r` are signal-linked and signals received from either call are forwarded to the other call. In state *dialogue*, the feature terminates when the caller hangs up. Modelled by an implicit *gone(i) / end(r)* in the original specification, these macros are expanded in the explicated model as explained in Chapter 3. The feature transitions to an intermediate state, state *terminating_r*, in which call `r` waits for an acknowledgement *downack* and transitions to the *final* state. Appendix A6 shows the Promela model for RVM.

5.3 Black Phone Interface

Black Phone Interface (BPI) is a bound BoxTalk feature. BPI acts as an interface between the DFC protocol and the telephone device (and its user). That is, it translates user inputs into DFC signals, and translates received DFC signals into tones that the user hears. For example, user action *onhook* is similar to call tear down, user action *dialed* is similar to setting up a new call. We modify the original BoxTalk specification (that is, the input of our program) to introduce call variable `a` to model a "channel" to the user for ease of reading. Actions *offhook*, *onhook*, and *dialed* are user inputs that are received on call `a`.

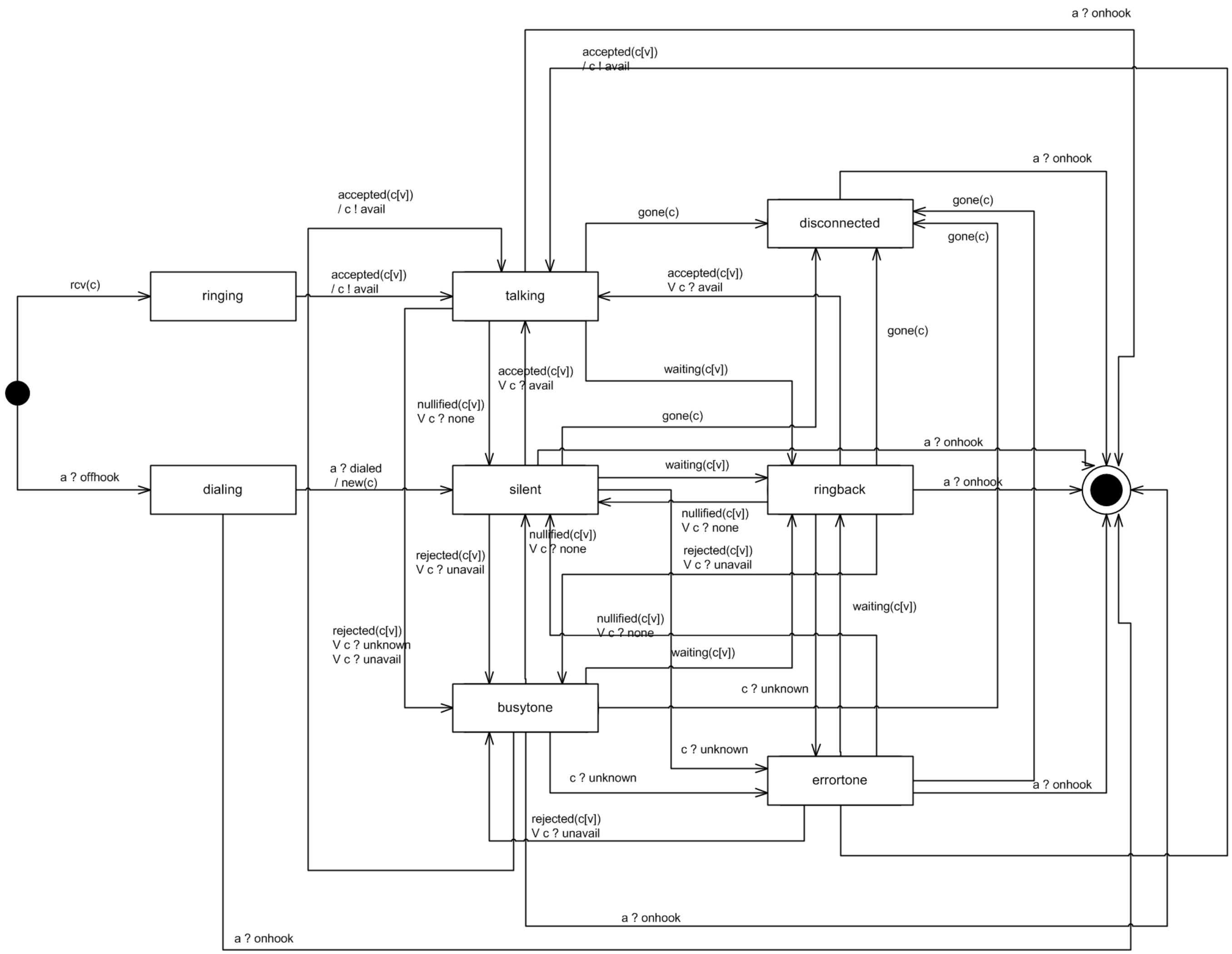


Figure 5.5: BPI - Original BoxTalk Specification

Figure 5.5 displays the original BoxTalk specification of the BPI feature. BPI has only one call, call `c`, as BPI is an endpoint of a usage. BPI reacts to DFC call signals, media channels, and user actions. There is a great deal of signaling redundancy in the BPI feature. For example, signals *accepted, rejected, nullified* on a media channel have exactly the same effect on the phone as DFC call signals *avail, unavail, none*, respectively. The media signal *waiting* has no DFC counterpart. `c[v]` represents the voice channel `v` on call `c`. *accepted(c[v])* means that signal *accepted* is received on the voice channel `v` of call `c`. States *dialing*, *ringback*, *busytone* and *errortone* are tone-generating states and their names indicate the tones the user should be hearing. Received calls are modelled via the path that passes through the state *ringing*. Outgoing calls are modelled via the path that passes through the state *dialing*.

Whenever the remote party hangs up (*gone(c)*), the feature transitions to the *disconnected* state. The feature cannot do anything in this state, and simply waits for the user to hang up (user action *onhook*), at which point the feature transitions to the *final* state.

Figure 5.6 displays the explicated BPI feature and Figure 5.7 displays the post-processing machine of the BPI feature. The macros in the original specification are expanded in the explicated specification as explained in Chapter 3. With user action *onhook*, the feature eventually transitions to the *final* state in the original specification; however, in the explicated model of BPI, the feature transitions to the *initial* state so that the feature can be invoked again with the next usage involving the subscriber. A post-processing machine is then called to complete the call termination (i.e., receipt of a *downack* signal). The Promela model for BPI feature is given in Appendix A7.

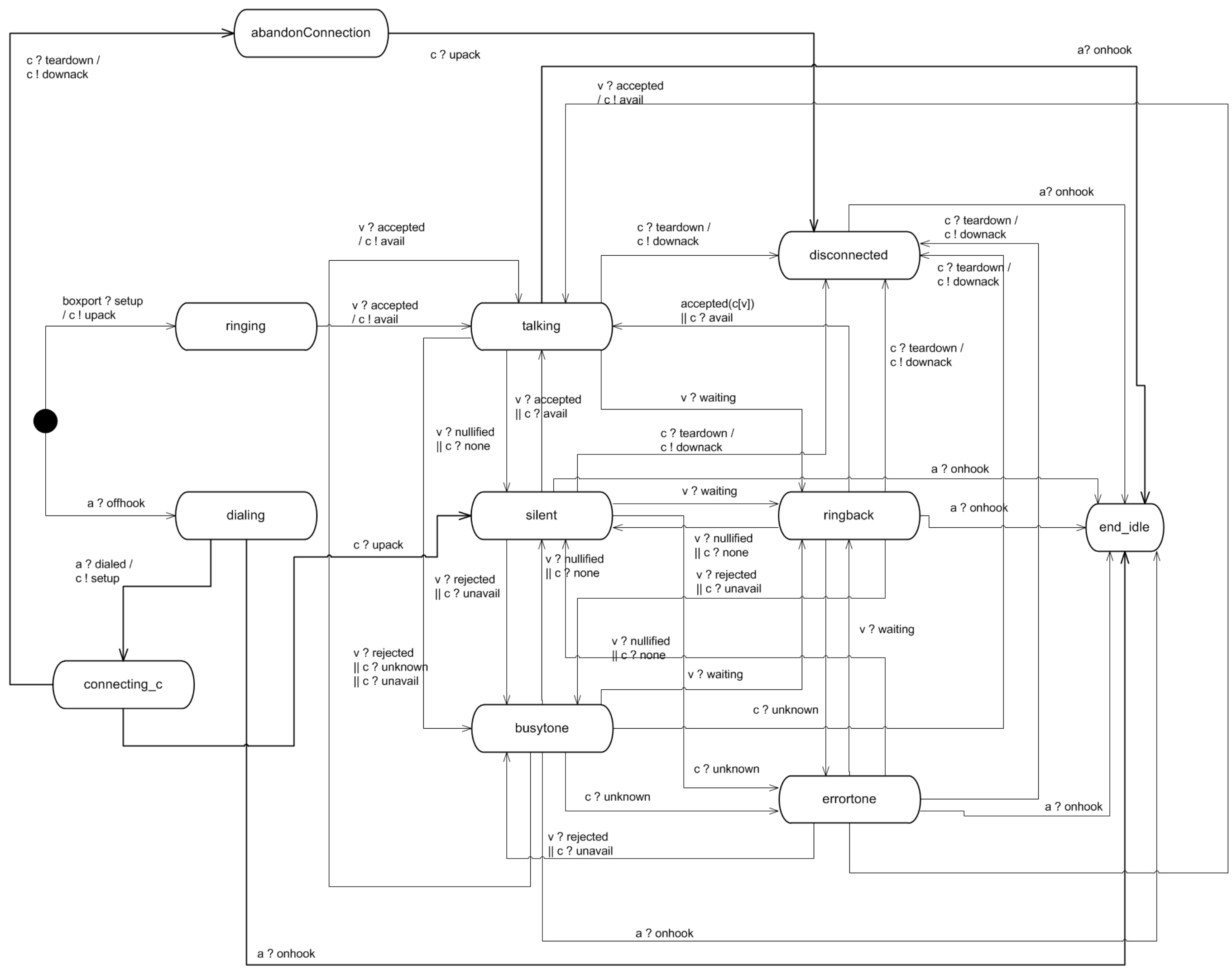


Figure 5.6: BPI - Explicated Specification

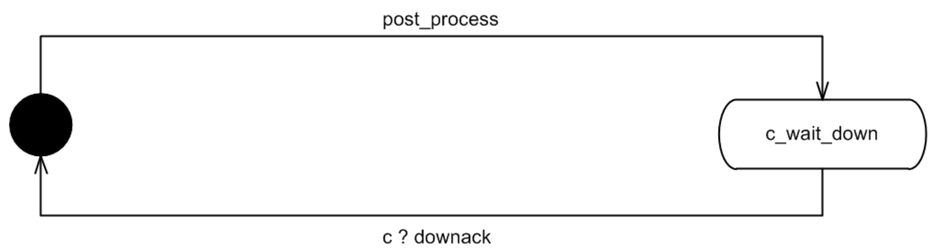


Figure 5.7: BPI Post Processing Machine

As explication of the remaining features from the case study is same as that explained in Chapter 3, we explain only feature-specific peculiarities for the rest of the features.

5.4 Answer Confirm

The Answer Confirm (AC) feature is designed to ascertain that a successfully established usage has reached a human callee by demanding that the callee press a touch-tone button on his or her phone. In the event of the button not being pressed, the feature suppresses the success outcome. It is a free feature.

Figure 5.8 displays the original BoxTalk specification of the AC feature. There are two different transitions from state *trying* to state *final*. For ease of reading, we show such multiple transitions with a single transition and enumerate it with transitions labels.

In state *trying*, the feature waits for the outcome signals from downstream. If an outcome signal *avail* is received on call `o`, the AC feature calls the Voice Message Service (call `r`) which will confirm that the callee has answered the phone, and transitions to state *confirming*. The feature remains in state *confirming* until a special confirmation is received from downstream. When the feature receives the feature-specific signal *"confirmed"*, the feature transitions to state *transparent*, sends signal *avail* upstream, and terminates the call to the Voice Message Service.

We modified the original model by introducing signal *"nonconfirmed"* (i.e., lack of confirmation) as the counter-part of *"confirmed"*. When the feature receives the signal *"nonconfirmed"* on call `r` in state *trying*, all active calls are terminated and the feature transitions to the *final* state.

Figure 5.9 shows the explicated model of the AC feature.

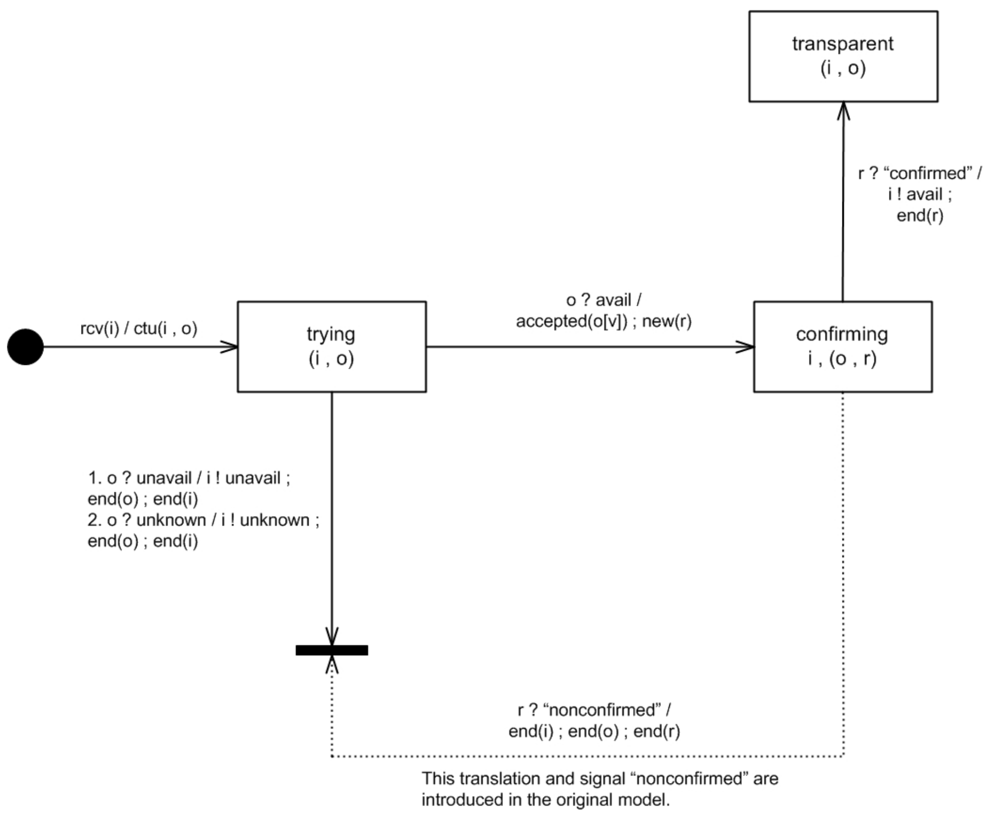


Figure 5.8: AC - Original BoxTalk Specification

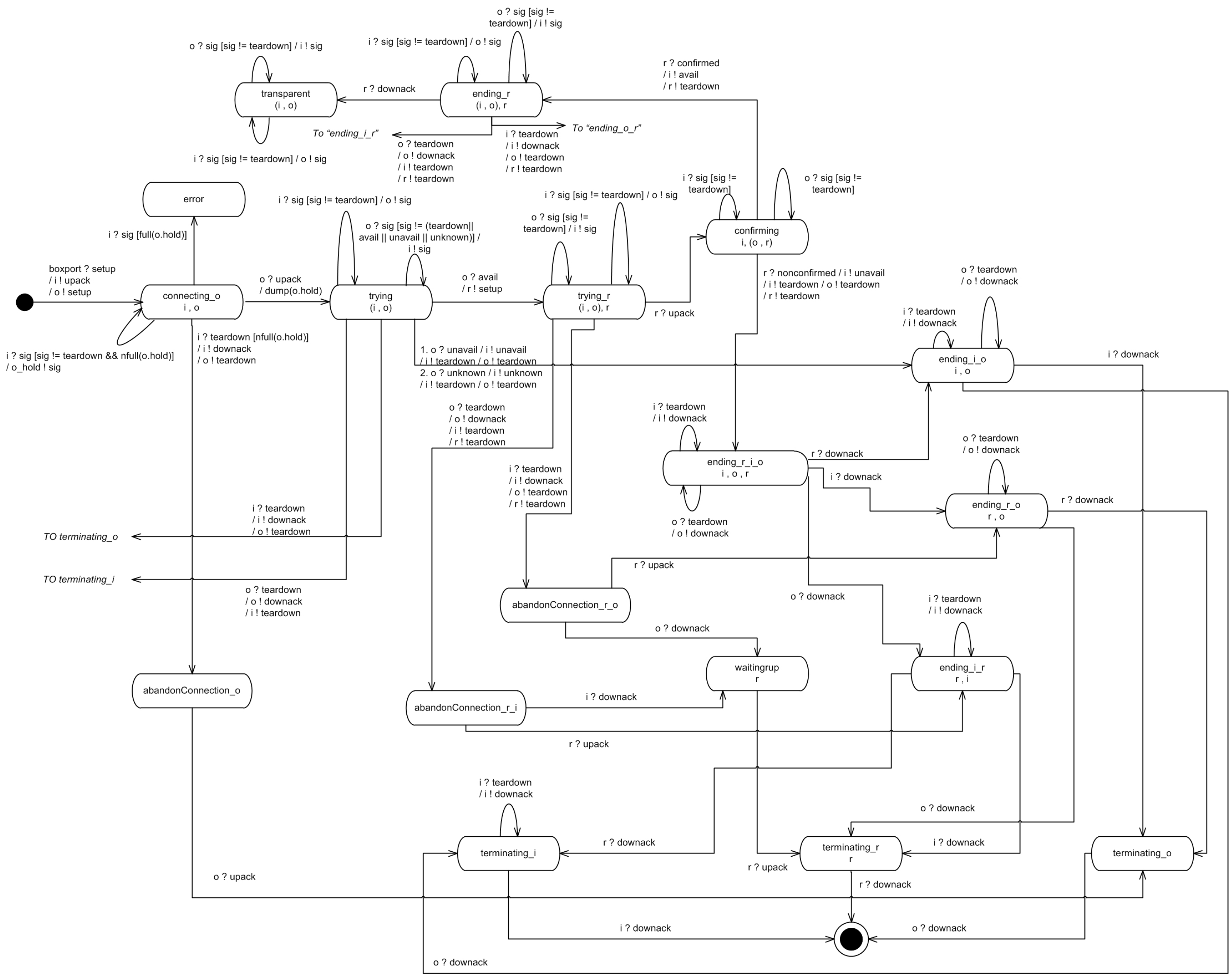


Figure 5.9: AC - Explicated Specification

5.5 Quiet Time

Quiet Time (QT) is a free, target-zone feature that is used by its subscribers if they do not want to be disturbed by a phone call. Figure 5.10 shows the original BoxTalk specification of the QT feature. After receiving the *setup* signal in its *initial* state, the QT feature transitions to the transient state *enabling*.

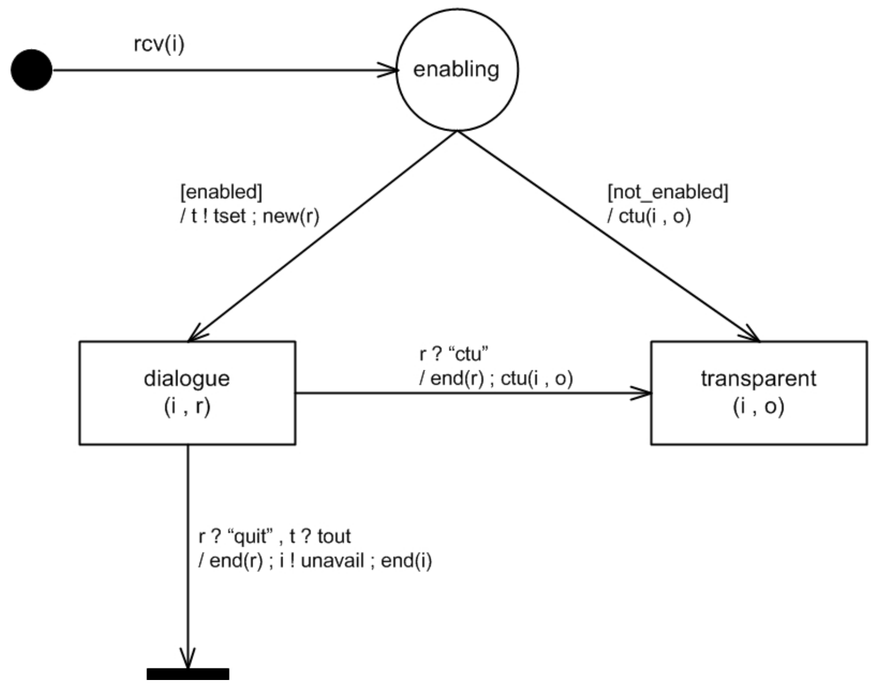


Figure 5.10: QT - Original BoxTalk Specification

From state *enabling*, different actions are taken depending on whether the feature is enabled. If not enabled, the feature transitions to stable state *transparent* and continues the usage via call `o`. In state *transparent*, the two calls `i` and `o` are signal-linked.

If instead the feature is enabled, the Voice Message Service is called (call `r`) and the feature transitions to stable state *dialogue*. In state dialogue, the two calls `i` and `r` are signal-linked and the caller engages in an Interactive Voice Response (IVR)[1] dialogue with the Voice Message Service. The IVR dialogue announces that the callee does not wish to be disturbed and offers the caller a number of different options to choose from. If the caller still wishes to talk to the callee despite the warning message, the caller can select option "continue" and

[1]http://en.wikipedia.org/wiki/Interactive_voice_response#Voice-Activated_Dialling

the Voice Message Service sends signal "ctu" to the QT feature. In this case, the feature tears down call `r` and continues the usage by setting up call `o`. Alternatively, the caller can abandon the call and leave a message by selecting option *"quit"* in state *dialogue*; the feature will eventually transition to state *final*.

The distinctive behaviour of QT feature (and PFM and SFM) is the use of timeouts. The timer variable `t` is set to a fixed time period (*t ! tset*) on entry to state *dialogue*. If the caller does not select any option in state *dialogue*, the feature will timeout (*t ? tout*) and will transition to state *final*. Due to the limitation of what we can check, our Promela model translator ignores conditions and actions involving timer variables when it encounters them. Figure 5.11 displays the explicated specification of the QT feature[2].

[2]We do not show self-transitions for the remaining features.

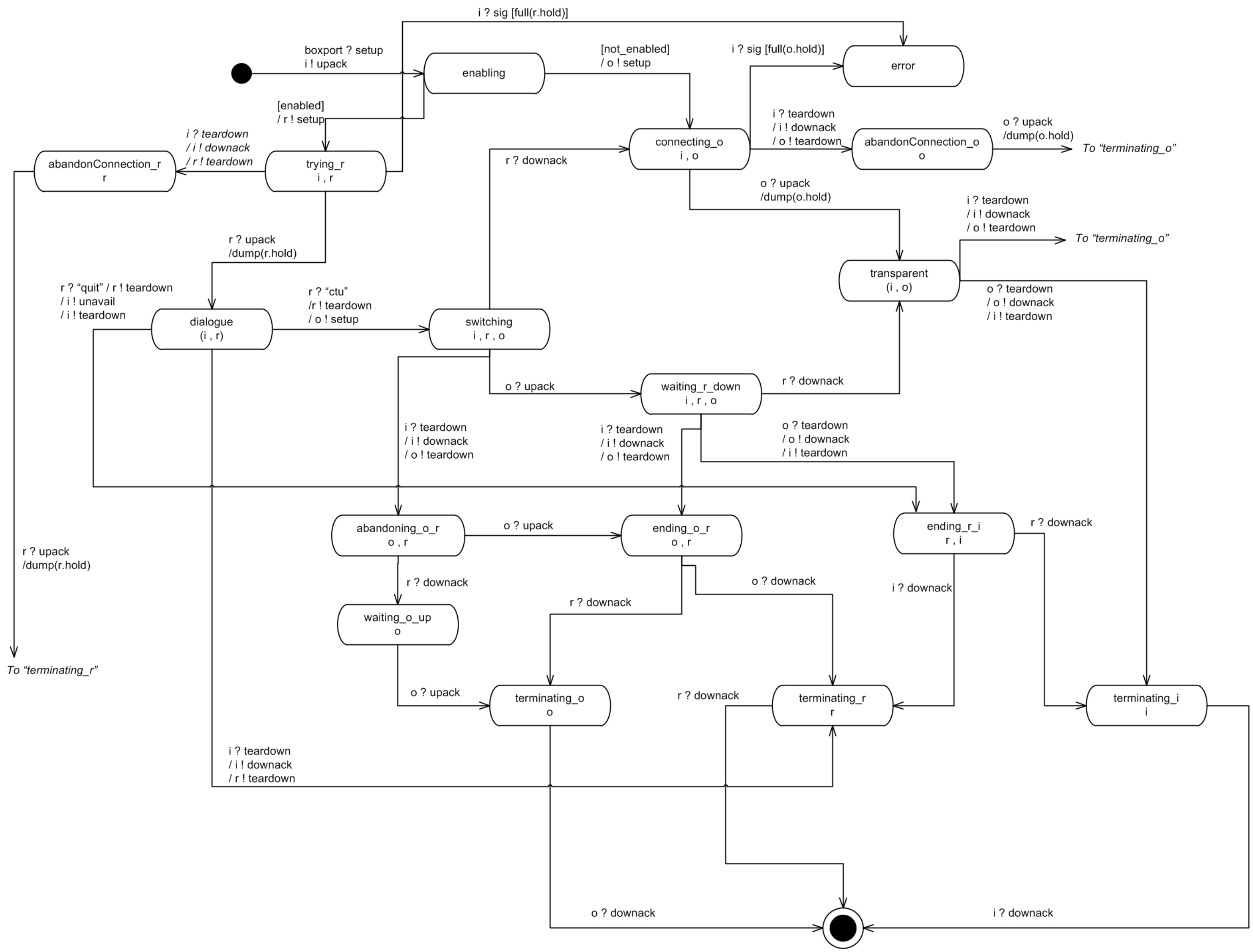


Figure 5.11: QT - Explicated Specification

5.6 Parallel Find Me

Parallel Find Me (PFM) is a free, target-zone feature that tries to direct a phone call to its subscriber's current location by translating the target of the usage to multiple addresses. Figure 5.12 displays the original BoxTalk specification of PFM. There are six different transitions from state *tworings* to state *onering* and two different transitions from state *onering* to state *final*.

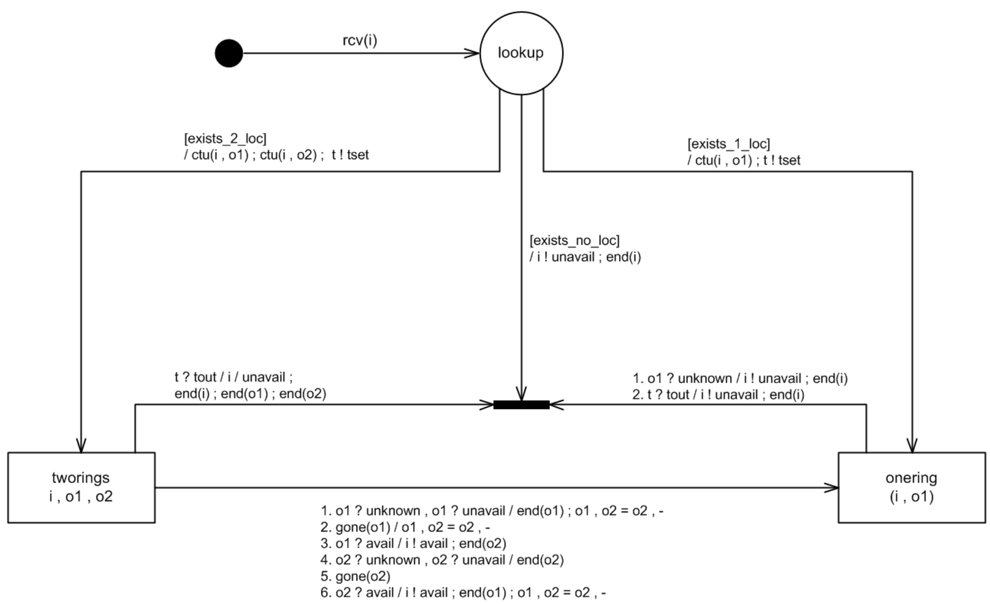


Figure 5.12: PFM - Original BoxTalk Specification

After receiving the *setup* signal in the *initial* state, the feature transitions to transient state *lookup*. The transient state *lookup* is used by the PFM feature to determine whether zero or more locations can be tried. In the original BoxTalk specification, in state *lookup* a *database query* initializes the locations that can be dialed, and also the data needed to evaluate the predicates *no_loc_exists*, *exists_1_loc*, and *exists_2_loc* (i.e. whether zero, one or two locations can be dialed)[3].

We abstracted the provided model to omit the *database query* and elide the specific locations. We know that exactly one of the transitions exiting the transient state has a guard

[3]The PFM model that was provided to us handles a maximum of two locations. In an actual feature, more locations could be tried in parallel.

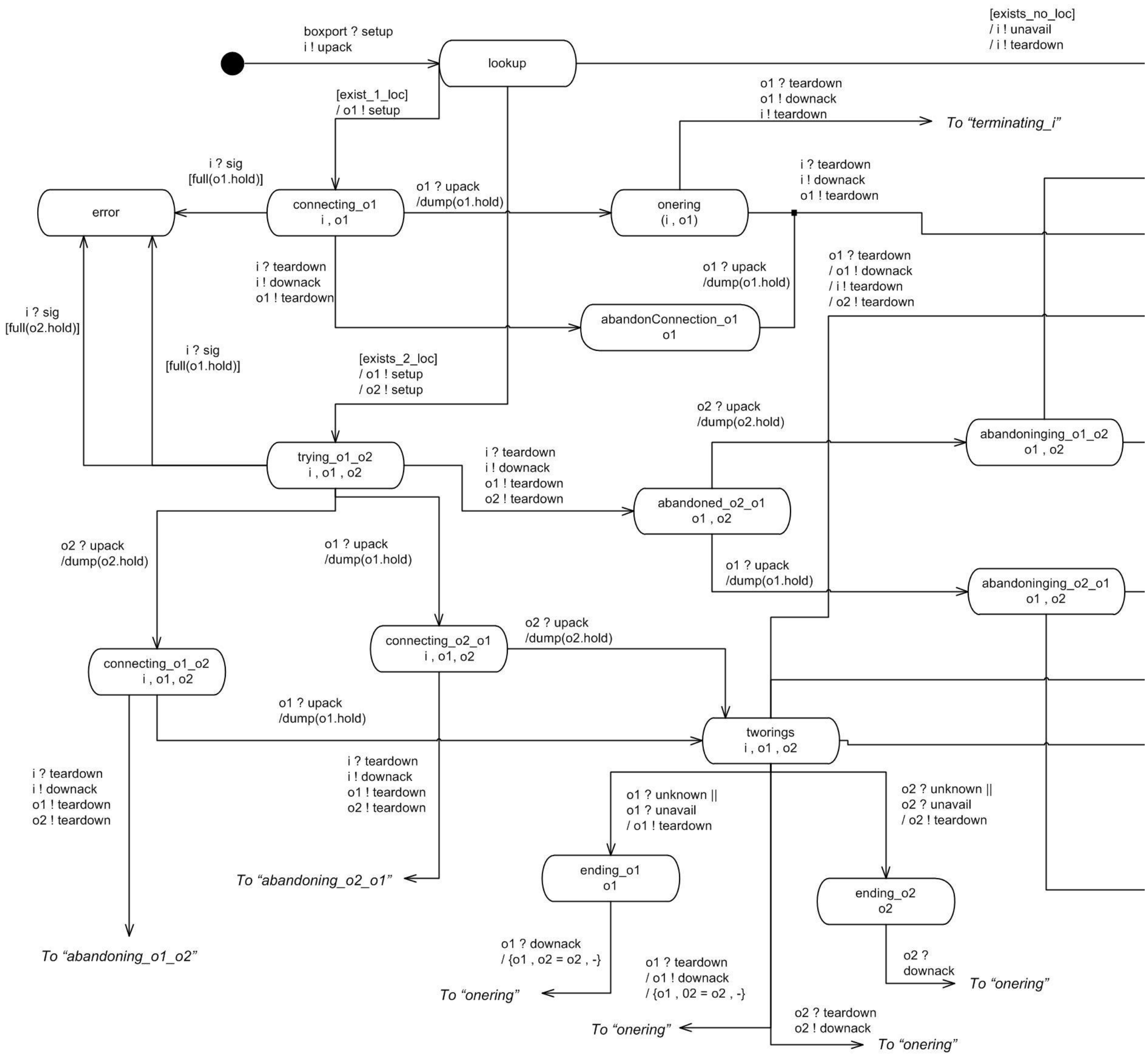
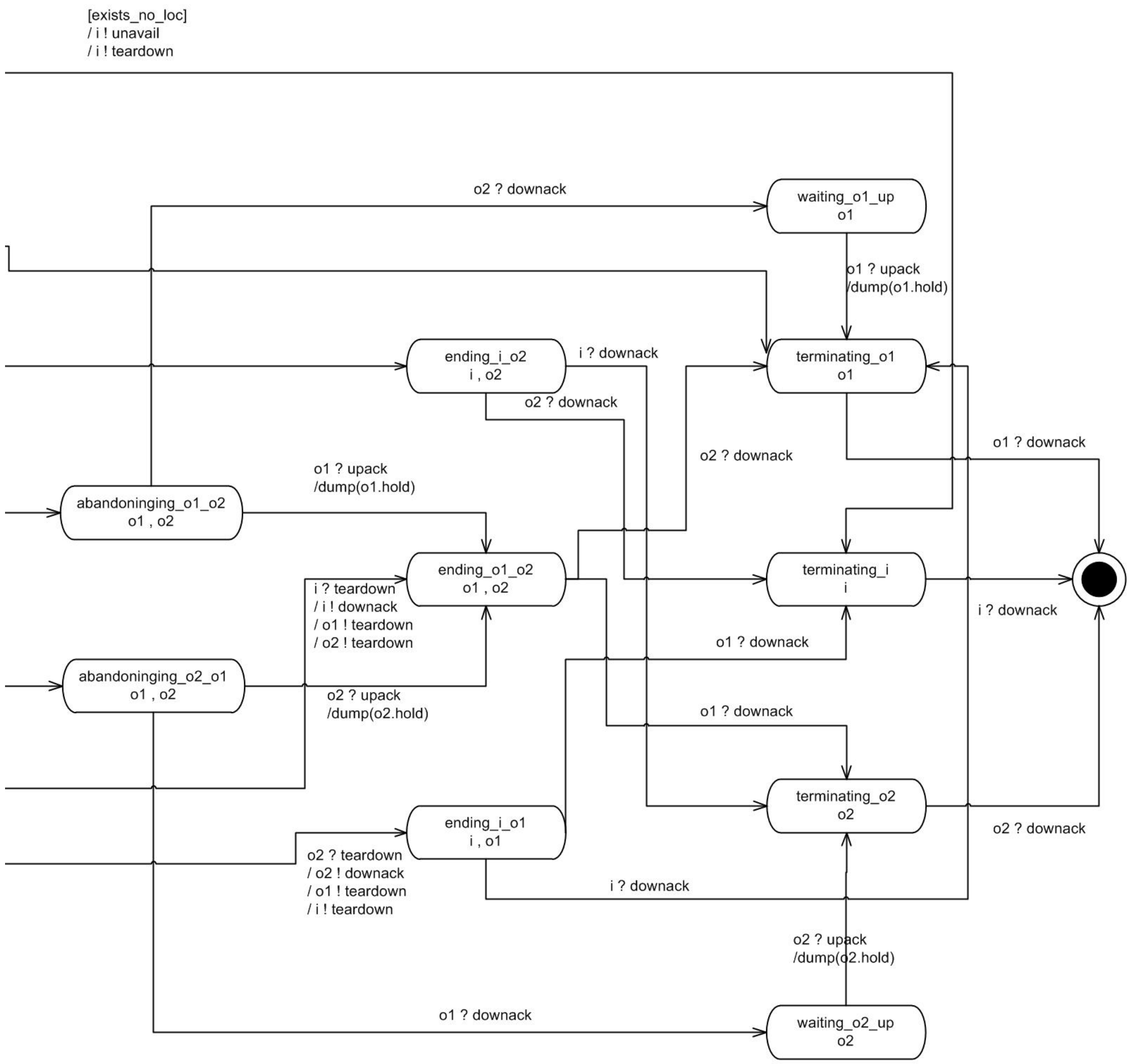


Figure 5.13: PFM - Explicated Specification

that evaluates to `true`. We make use of SPIN's nondeterminism to model that the number of addresses to try could be zero, one, or two.

If there are no locations to try (i.e., *no_loc_exists* evaluates to `true`), signal *unavail* is sent on call `i` and the feature transitions to the *final* state.



If there is only one location to try (i.e., *exists_1_loc* evaluates to `true`), the feature continues the usage via call `o1` and transitions to state *onering*. In this state, if call `o1` receives an *unknown* or an *unavail* signal, then the feature fails to find the subscriber: signal *unavail* is sent on call `i` and the feature transitions to the *final* state. State *onering* acts as a *transparent* state where calls `i` and `o1` are signal-linked, and indicates a successful Find Me.

If there are two locations to be tried (i.e., *exists_2_loc* evaluates to `true`), the feature continues the usage to calls `o1` and `o2` in parallel and transitions to state *tworings*. While these two locations are being called in parallel, a special signal "wait" is sent on the voice channel of call `i` (`i[v]`) so that the caller hears a ring-back and will not hang up. In state *tworings*, if call `o1` receives signal *unknown* or *unavail*, the feature tears down that call, assigns the value of call variable `o2` to call variable `o1`, and transitions to state *onering*. Similarly, if call `o1` hangs up in state *tworings*, the feature transitions to state *onering* and makes the same call-variable assignments. From state *tworings*, the feature will also transition to state *onering* if call `o2` receives an *unknown* or an *unavail* signal, or hangs up.

The feature uses timeouts. The timer variable `t` is set to a fixed period (*t ! tset*). If variable `t` times out (*t ? tout*), signal *unavail* is sent on call `i` and the feature transitions to the *final* state. Figure 5.13 displays the explicated model of PFM feature box. From state *connecting_o1_o2*, fully established call `o2` may hang-up causing calls `i` and `o1` to tear down. The feature will transition to state *abandoning_o1_i*. From state *abandoning_o1_i*, if call `i` receives acknowledgement *downack*, the feature will transition to state *abandonConnection_o1*. If call *o1* receives an *upack*, the feature will transition to state *ending_i_o1* where both the calls wait for *downack* acknowledgements. Similar transitions follow from state *connecting_o2_o1* when call `o1` hangs up. For space reasons, we decided not to include these transitions in the figure.

5.7 Sequential Find Me

Sequential Find Me (SFM) is also a free, target-zone feature that serves the same purpose as the PFM feature, except that SFM tries different locations sequentially. Figure 5.14 depicts the original BoxTalk specification of the SFM feature.

After receiving the *setup* signal in its *initial* state, the SFM feature transitions to the transient state *lookup*. Transient state *lookup* is used by SFM to determine whether zero or more locations can be tried. In the original BoxTalk specification, in state *lookup* the *database query* is used to initialize the locations that can be dialed, and also the data needed to evaluate the predicates *loc_list_empty* and *loc_list_n_empty* (i.e., whether any locations can be dialed or not). Similar to PFM, the Promela model of SFM is abstracted and uses SPIN's nondeterminism in place of actual location list.

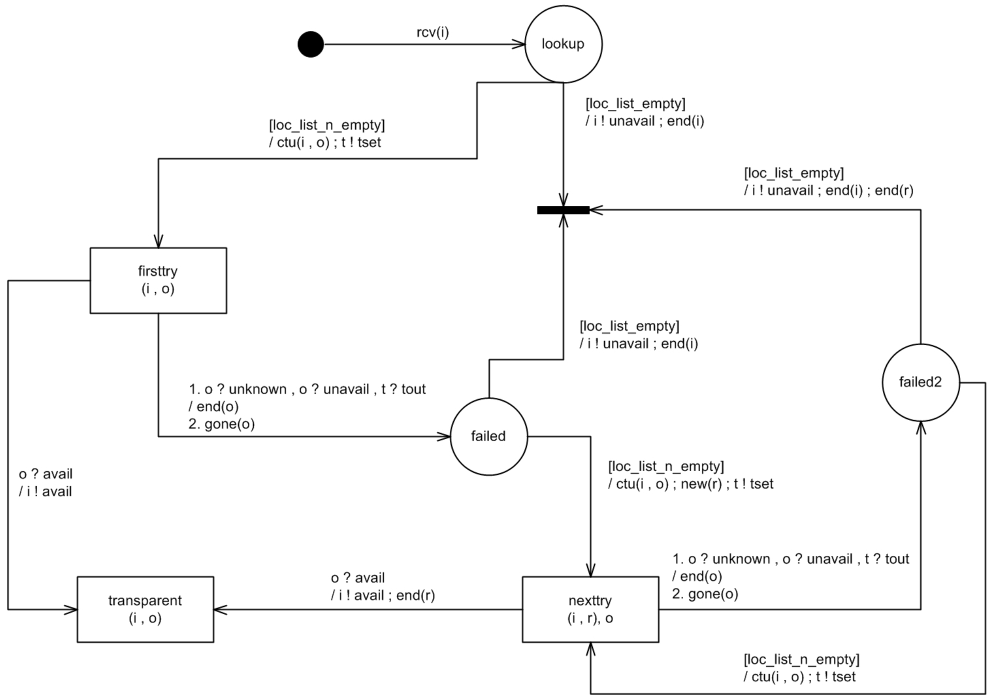


Figure 5.14: SFM - Original BoxTalk Specification

If there are no locations to try (i.e., *loc_list_empty* evaluates to `true`), signal *unavail* is sent on call `i` and the feature transitions to the *final* state.

If there are location(s) to try (i.e., *loc_list_n_empty* evaluates to `true`), the feature continues the usage via call `o` and transitions to state *firsttry*. If the feature receives signal *unknown* or *unavail*, or call `o` hangs up in state *firsttry*, the feature transitions to transient state *failed* and tears down call `o`.

In transient state *failed*, the feature evaluates the two predicates, and if the location list is empty, the feature transitions to the *final* state by sending the signal *unavail* on call `i`. If indeed the location list is not empty, the feature calls the Voice Message Service (call `r`) to play an announcement for the caller so that the caller will not hang up before all other locations in the location list have been tried, one after the other. The feature transitions to the state *nexttry*.

From state *nexttry*, with every failed attempt to locate the subscriber (*o ? unknown* or *o ? unavail*) or if call `o` hangs up, the feature transitions to state *failed2*. If there is still a location to try, a call to the new location is set up and the feature transitions again to state *nexttry*. Eventually, either the usage succeeds and transitions to state *transparent*, or the location list is exhausted (all locations are tried and all attempts of locating the subscriber fail) and the feature transitions to the *final* state. Similar to the PFM feature, the SFM feature employs timer variable `t` to set deadlines for each attempt to find the subscriber at a particular location. Figure 5.15 displays the explicated model of the SFM feature. The dotted transition from state *trying_o_r* to state *nexttry* is similar to the transitions from state *trying_o1_o2* to state *tworings* in explicated model of PFM, Figure 5.13.

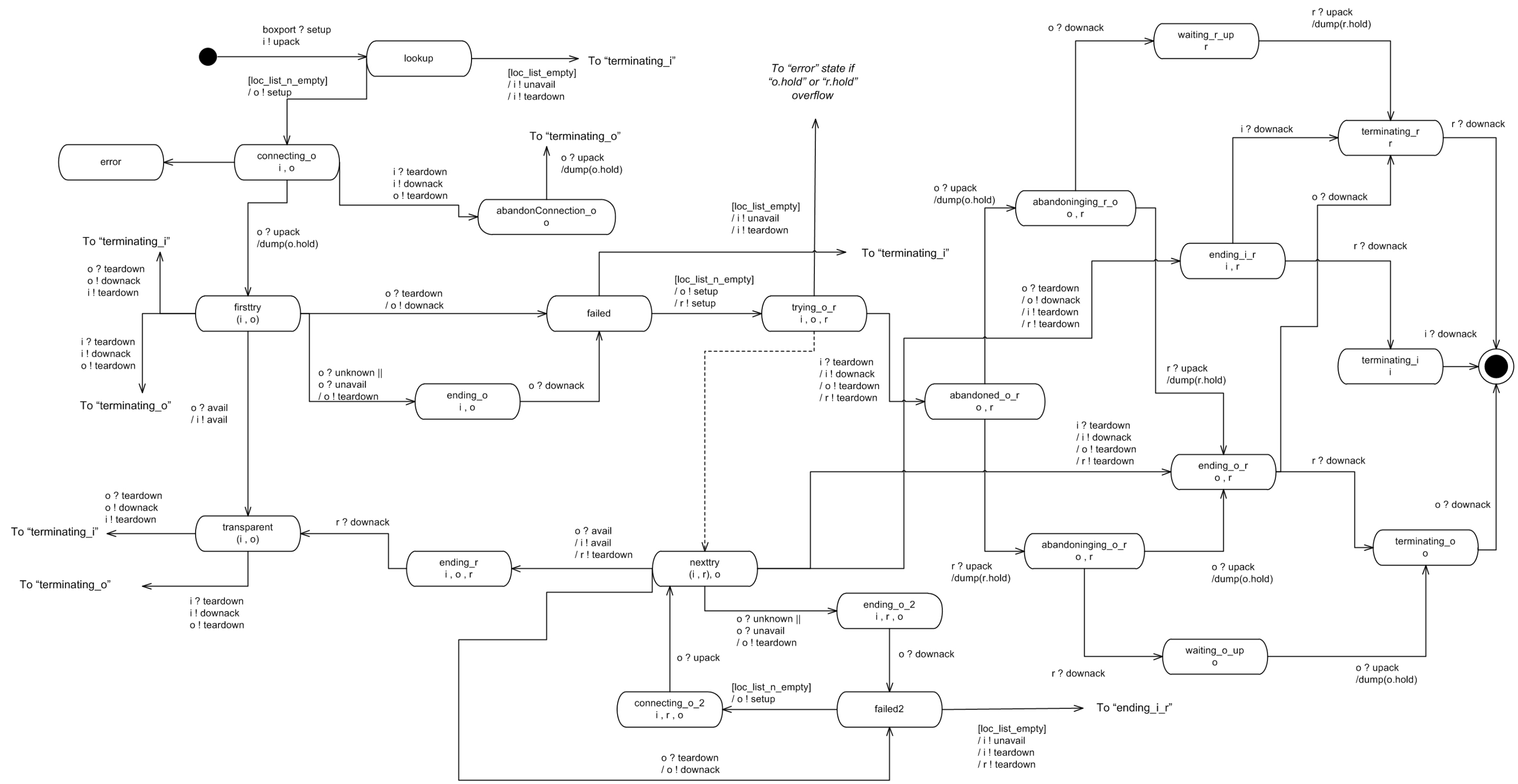


Figure 5.15: SFM - Explicated Specification

5.8 Properties

In this section, we explain the properties that we prove, and talk about the property specification language. To verify any system using SPIN, we need to have Promela model(s) of that system. Promela is the input language of the SPIN model checker. Therefore, we translate our BoxTalk features into Promela models. By automating the translation process, we ensure fast and efficient translation.

5.8.1 Properties of Interest

We prove that the same set of properties used in [12] hold of our Promela models. These properties encompass the basic behaviour of the DFC protocol. Since our translation includes explication of BoxTalk, whose macros encode DFC protocol compliance, checking *these* properties effectively checks that our explication process correctly expands the BoxTalk macros and other implicit behaviours.

1. A *setup* signal is eventually acknowledged with an *upack* signal.
2. A *teardown* signal is eventually acknowledged with a *downack* signal.
3. A feature cannot send any status signals (on output channels) before sending an *upack* signal.
4. A feature cannot send any status signals (on output channels) after sending a *teardown* signal.
5. In bound features, every received *setup* signal is acknowledged with an *upack* signal[4]
6. This is a BTB specific property. When BTB receives a new *setup* signal and advances to state *orienting*, if during this, the post-processing process is not in state *end_idle*, it implies that it is tearing down the **previous** call.
7. This property is BPI specific. If the main process stays in state *initial* and the post-processing process is not in state *end_idle*, the post-processing process is tearing down the **current** call.

[4]Bound features can receive and react to multiple *setup* signals in their lifetime.

To make sure that we are not falsely proving these properties, we use an antecedent that the feature process in not in the *error* state in conjunction with properties 1 to 5. We use the remote reference operator of SPIN [7]:

```
name@label
```

which states that the `proctype` *name* is in the local control state marked by control-flow label, *label*.
We use the negation of the remote reference operator in conjunction with properties 1 to 5, to state that the *feature* process is not in the *error* state when we prove the properties. The formulated properties in Section 5.8.3 show the use of this antecedent.

5.8.2 Global Monitor Variables as Embedded Correctness Variables

As explained in Chapter 4, Section 4.1.2, we cannot formulate properties that refer to signals received or sent because we use rendezvous channels for communication, and we model states with state labels. Rendezvous channels have zero capacity and we cannot query their contents. Hence, we use global monitor variables such as `rcv_setup`, `send_downack`, etc. that record signal events. These variables are reset to `false` in the inline function `reset()` and updated (i.e., set to `true`) in the inline function `next_trans()` when the associated signals are sent or received.

The correctness properties that we prove refer to the receipt of acknowledgements or to signals sent. Their expression in SPIN refers to the corresponding global monitor variable instance.

5.8.3 Formulated Properties

The properties listed in English in Section 5.8.1 are expressed as LTL formulas and never claims as follows:

1. `( !FeatureProcess`[5]`@error_state ) && ( [] ( rcv_setup -> <> send_upack ) )`

2. `( !FeatureProcess@error_state ) && ( [] ( rcv_teardown -> <> send_downack ) )`

[5] Name of the feature process being checked.

3. `( !FeatureProcess@error_state ) && ( [] ( rcv_setup -> ( ( !send_avail ∧ !send_unavail ∧ !send_unknown ) U send_upack ) ) )`

4. `( !FeatureProcess@error_state ) && ( [] ( ( send_teardown || rcv_teardown ) -> [] ( !send_avail ∧ !send_unavail ∧ !send_unknown ) ) )`

5. `( !BTB@error_state ) && ( [] ( ( rcv_setup && ( current_call == num1 ) ) -> <>( send_upack && ( current_call == num1 ) ) ) ) )`

6. `never { ( BTB@orienting_state && !( pp_s@end_idle_state ) ) && !( pp_call == last_call ) }`

7. `never { ( BPI@initial_state && !( pp@end_idle_state ) ) && !( pp_call == current_call ) }`

5.8.4 Explanation in English

In the above list, properties `1`, `2`, `3`, and `4` are very straight forward and easy to understand.

Property `5` is similar to property `1` with an additional clause which states that the call sending a *setup* signal is the same one which receives an *upack* response. Since bound boxes can have more than one *setup* signal, but each one represents a different call attempt, call attempts are uniquely numbered to match *setup* signals with their corresponding *upack* signals.

Variable `current_call` has the value of the most recently set up call, `last_call` has the value of the call that was set up just before `current_call` (the second-most-recently set up call), and `pp_call` has the value of the most recent call that the post-processing machine is tearing down. If (`pp_call` == `last_call`), then the post-processing machine is tearing down the previous call.

For properties `6` and `7`, we use `@` to indicate that a process is in a particular state. For example, `BTB@ orienting_state` states that the main process of BTB is in `orienting_state`. Properties `6` and `7` are unintuitive. To better visualize the structure of the properties, let us first abbreviate the clauses in the formula. Let us denote

`BTB@ orienting_state` in property `6` by `p`
`pp_s@ end_idle_state` by `q`
`pp_call == last_call` by `r`.
The English description of the original property can be expressed as $((p \wedge \neg q) \rightarrow r)$. It can be expressed only in terms of $\wedge$ and $\neg$ logical connectives as follows:

$$
\begin{aligned}
& (p \wedge \neg q) \rightarrow r \\
\Leftrightarrow\ & \neg(p \wedge \neg q) \vee r && \text{as } a \rightarrow b \equiv \neg a \vee b \\
\Leftrightarrow\ & \neg((p \wedge \neg q) \wedge \neg r) && \text{as } a \vee b \equiv \neg(\neg a \wedge \neg b)
\end{aligned}
$$

The property in the list above has `p`, `q`, and `r` replaced by their original forms and $\neg$ is represented by `never`. The translation of property `7` is similar.

5.9 Model Checking and Results of Verification

In this section, we present our model checking attempts and the results of verification.

For model checking, we used Spin version 6.0.1 running on Linux platform (Ubuntu 10.10) on an Acer Aspire Laptop with Intel® Pentium® dual core processor T2300 (2.00 GHz) with 3.00 GHz RAM.

Initial attempts to verify properties described in Section 5.8 revealed errors in the translation program. After fixing all the typographical errors in our generated models (and our translator program), we started model checking. In our generated models, we used state labels in the environment process. For example, the following is a part of Free Transparent Feature's Promela model:

```
if
::ss.cs == connecting_o ->
    glob_ins[ss.in.o_in] ! upack;
fi;
```

However, with this approach, we discovered that the model always transitions to state *transparent* from state *connecting_o* with the receipt of acknowledgement *upack*. State *abandonConnection_o* was never reached in the verification run.

We encountered another error while we were checking Receive Voice Mail model. SPIN reported the following error:

```
$ pan:1: invalid end state (at depth 224)
$ pan: wrote freeboxrvm.pml.trail
```

When we examined the generated counterexample, it was found out that the environment process was issuing the wrong acknowledgement. The counterexample generated followed the path from *initial* state to *connecting_o* state to *transparent* state to *switching* state to *abandoning_r_o* state. Please refer to Figure 5.4. From state *switching*, if caller `i` hangs up, (*i ? teardown / i ! downack*), a *teardown* signal is issued on the half complete call `r`. The environment process was sending acknowledgement *downack* on call `r`, however, the half established call `r` was first expecting an acknowledgement *upack*, which it never received.

After fixing these errors, we were able to prove all of the properties stated in Section 5.8. In Appendices A3, A4, A5, A6, and A7, we present Promela models of explicated FTB, BTB, EI, RVM, and BPI features respectively with the properties proved for each model. For space reasons, we do not include Promela models of other features.

We also proved that the properties being checked are not vacuously true in our models by inserting `assert(false)` after each state label. By checking these assertions, SPIN model checker determines that every state is reachable. For example, after inserting `assert(false)` following label `abandonConnection_o_state` in RVM feature, SPIN produced the following error:

```
$ pan:1: assertion violated 0 (at depth 161)
$ pan: wrote freeboxrvm.pml.trail
```

The time to verify a model increases rapidly as the model size increase. For instance, SPIN model checking of the EI feature with three states and three transitions runs in under a minute, whereas SPIN model checking of AC feature with 20 states and 67 transitions requires 18 minutes to generate the result.

Chapter 6

Conclusion

We developed a fully automated translator from BoxTalk features to Promela models. We verified the translation by checking the resulting Promela models against DFC-compliance properties using the SPIN model checker.

6.1 Explicating BoxTalk features

BoxTalk is a domain-specific, call-abstraction, high-level programming language used to program DFC features. BoxTalk abstracts the common behaviour that is present in all DFC features into BoxTalk macros and other implicit behaviour. However, to analyze BoxTalk features, the feature models need to explicitly represent the implicit behaviour. A large part of our work involved explicating BoxTalk features to explicitly represent features' implicit behaviour.

Explication of free and bound BoxTalk features follow slightly different steps. The explication process of free BoxTalk features takes place as follows:

1. Expand all macros present in the transitions of the original BoxTalk specification. This step may generate new states and transitions that may be explicated in other steps.

2. In every signal-linked state, some active call may end because the remote party of that call hangs up, which causes the other signal-linked call to terminate. This step explicates the receipt of events initiating the teardown and the completion of the

tearing down of both the signal-linked calls. Extra states and transitions may be generated in this step as well.

3. In certain states, the feature reacts to signals without changing state. For example, in signal-linked states, the default behaviour of the feature is to forward any signal from one signal-linked call to the other. Such behaviour is explicated as self transitions that have the same source and destination state. This step augments the feature with these self transitions.

The explication process of bound BoxTalk features is similar to that of free BoxTalk features, with some unique explications:

1. This step expands all macros present in the transitions of the original BoxTalk specification. The only difference is the explication of macro *end()*. Since there is only one instance of a bound feature per subscriber, the teardown of calls in bound features must be instantaneous. However, the teardown of a call comprises steps which are not instantaneous (e.g., waiting for a *downack* acknowledgement). A post-processing machine, one per each call variable in a feature, is created, in each bound feature, to complete the teardown process of terminating calls. This way, the feature machine can set up a new call, or participate in a new usage, and the post-processing machine can tear down the old call in parallel. Since this one instance of a bound feature must be included in every usage involving the subscriber, whenever all calls terminate, the feature transitions to the *initial* state so that it can immediately participate in the next usage.

2. This step handles call termination from signal-linked states.

3. Bound features can receive and react to *setup* signals in any stable state. Different actions are taken depending on whether the *setup* request is from the subscriber or from the far party. This step handles the receipt of such *setup* signals.

4. This step, which is similar to Step 3 for free features, handles self-transitions.

6.2 Translation to Promela

Our program translates explicated BoxTalk features into Promela models.

For each free BoxTalk feature, there is one active Promela process for the feature machine and another active Promela process representing the environment process. The environment process models the environment of the feature, generating signals that the feature can receive from its environment and receiving the feature's output. For bound BoxTalk features, there are additional active processes, one per call in the usage, that model the post-processing machine.

6.3 Modifications to Yuan Peng's Thesis

In her Master's thesis, Yuan Peng [12] manually explicated a set of BoxTalk features and hand translated the explicated models into Promela models. We have fully automated the process of explicating and translating BoxTalk specifications into Promela models. We use a multi-step explication process to explicate free and bound features (Section 6.1). We also modified explication rules of macros *new()* and *ctu()* from the existing rules developed by Yuan Peng. Our modified rules include the special case of explication when the caller hangs up in an intermediate state where the "new" call is half established, (i.e., waiting for an acknowledgement *upack*). The feature terminates in a special way where the half established call waits for two acknowledgements, first an *upack* and then a *downack*. In the intermediate state where new call is half established, a hold queue is constructed to hold all of the signals to be sent via this call. If the hold queue overflows, the feature transitions to the *error* state. Our modified rules of macros *new()* and *ctu()* also handle this case.

Our bound features differ from [12], in that they have multiple post-processing machines, one per each call variable of the bound feature. Since a bound feature's main machine communicates with its post-processing machine(s) via rendezvous channel(s), then if there is only a single post-processing machine to process the termination of multiple calls, then, whenever the feature's main machine tries to end multiple calls, the post-processing machine would receive the first request to end a call and all subsequent requests would be discarded.

6.4 Case Study

We used a case study to evaluate our translator by translating a set of available BoxTalk specifications into Promela models and proving that they are DFC compliant. Since the translation of BoxTalk specifications to Promela models includes expanding of BoxTalk macros that encode DFC compliance, we check our Promela models against properties that encompass DFC protocol. Checking that the Promela models satisfy these properties also proves that the models have been correctly explicated.

All of the properties that we proved were DFC-protocol properties. The generated models can also be used to prove other correctness properties of the features (e.g., that execution should never halt in a *transient* state or that the path between two responsive states must be cycle-free). However, we did not attempt to identify correctness criteria to be proved of the case-study features as this was outside the scope of our thesis. We were interested in transforming original specifications into Promela models so that such verifications are possible.

We verified individual BoxTalk features and their interactions with the environment. However, real systems involve combinations of features. For example, any feature should include an *Error Interface Box* in case a caller dials an invalid number. With bound features, the subscriber may subscribe to more than one bound feature. As a future work, our program can be extended to facilitate the model checking of combinations of features and to examine how one feature interacts with another feature (and not just the environment).

APPENDICES

Appendix A

Original Grammar

```
PARSER_BEGIN(Boxtalk)

public class Boxtalk {
   public static void main(String args[]) throws ParseException {
      Boxtalk parser = new Boxtalk(System.in);
      parser.Input();
   }
}

PARSER_END(Boxtalk)

// LEXICAL PART

SKIP : {
   " "  |
   "\t" |
   "\n" |
   "\r"
}

MORE : {
   "//" : IN_SINGLE_LINE_COMMENT |
   "/*" : IN_MULTI_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN : {
   <SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT
```

```
}

<IN_MULTI_LINE_COMMENT>
SPECIAL_TOKEN : {
    <MULTI_LINE_COMMENT: "*/" > : DEFAULT
}

<IN_SINGLE_LINE_COMMENT,IN_MULTI_LINE_COMMENT>
MORE : {
    < ~[] >
}

TOKEN : {
    < ARC: "Arc" > |
    < AVAIL: "avail" > |
    < BOUND: "Bound" > |
    < CALL: "Call" > |
    < CLASS: "Class" > |
    < CLS: "cls" > |
    < CTU: "ctu" > |
    < DLD: "dld" > |
    < END: "end" > |
    < FREE: "Free" > |
    < GONE: "gone" > |
    < GRAPH: "Graph" > |
    < INIT: "Init" > |
    < NEW: "new" > |
    < NONE: "none" > |
    < NOSIG: "nosig" > |
    < NOTES: "Notes" > |
    < OUT: "out" > |
    < RCV: "rcv" > |
    < REV: "rev" > |
    < SET: "Set" > |
    < SETUP: "setup" > |
    < SIGNAL: "Signal" > |
    < SLOT: "Slot" > |
    < SRC: "src" > |
    < STABLE: "Stable" > |
    < STAT: "stat" > |
    < STRING: "String" > |
    < SUBS: "subs" > |
    < TERM: "Term" > |
    < TEXT: "text" > |
    < TIMER: "Timer" > |
```

```
    < TOUT: "tout" > |
    < TRANSIENT: "Transient" > |
    < TRG: "trg" > |
    < TSET: "tset" > |
    < TYPE: "type" > |
    < UNAVAIL: "unavail" > |
    < UNKNOWN: "unknown" > |
    < VIDEO: "video" > |
    < VOICE: "voice" > |
    < ZONE: "zone" >
}

TOKEN : {
    < STRINGLIT: "\"" ( ~["\""] )* "\"" >
}

TOKEN : {
    < JAVALIT: "\$" ( ~["\$"] )* "\$" >
}

TOKEN : {
    < ID: ["a"-"z","A"-"Z","_"] ( ["a"-"z","A"-"Z","_","0"-"9"] )* >
}

TOKEN : {
    < DIGITS: ["0"-"9"] ( ["0"-"9"] )* >
}

// THE GRAPH
// Currently Missing:  call arrays
// Additional Syntactic Constraints:
// 1) There is exactly one initial state, which has no in-transitions.
// 2) Each transient state has at least one in-transition and at least one
//    out-transition.
// 3) Each stable state has at least one in-transition.
// 4) Each termination state has at least one in-transition and no
//    out-transitions.
// 5) If a CallVarName appears in a StableStateItem, it must either be
//    declared as a call variable in the Notes, or it must be the name of a
//    pseudocall. If it is the name of a pseudocall, it can only appear in
  a
//    Linkage in which all the other LinkageObj have the form "c[m]".
// 6) With the one exception in (5), a Linkage must link either all
//    CallVarNames or all "c[m]" expressions.
// 7) An expression "c[m]" in a LinkageObj must match the declarations
```

```
//    according to the semantics of media processing in Boxtalk.
// 8) If a ProgName appears in a TransientState, it must be defined in the
//    Notes as a void program, in either Java or Boxtalk.
// 9) Each path between responsive states must be cycle-free.

void Input() : {} {
   <GRAPH> "{" GraphItemSet() "}" <NOTES> "{" BoxtalkNotes() "}"
}

void GraphItemSet() : {} {
   ( ResponsiveState() | TransientState() | Arc() )*
}

void ResponsiveState() : {} {
   <INIT> StateName() "{" "}" |
   <TERM> StateName() "{" "}" |
   <STABLE> StateName()
            "{" ( StableStateItem() ( "," StableStateItem() )* )? "}"
}

void StableStateItem() : {} {
   CallVarName() |
   "(" Linkage() ")"
}

void Linkage() : {} {
   LinkageObj() ( ( "<" | ">" ) LinkageObj()                |
                  "," LinkageObj() ( "," LinkageObj() )*
                )
}

void LinkageObj() : {} {
   CallVarName() ( "[" SlotName() "]" )?
}

void TransientState() : {} {
   <TRANSIENT> StateName() "{" ( ProgName() )? "}"
}

void Arc() : {} {
    <ARC> "{" StateName() "->" StateName() "{" ArcBody() "}" "}"
}

// TRANSITION (ARC, FOR SHORT) BODIES
// Additional Syntactic Constraints:
```

```
// 1) If an arc originates at a responsive state, the conditions in its
   CondList
//    must begin with "rcv", "gone", or "callvarname?".
// 2) If an arc originates at a transient state, the conditions in its
   CondList
//    must be BoolExps or ProgNames or "!".
// 3) "!" is the only Cond in its CondList.
// 4) Each transient state has at most one out-transition with Cond "!".
// 5) All CallVarNames parsed in Conds must be declared as call variables in
//    the Notes, with the exception of one parsed preceding "tout", which
//    must be declared as a timer variable in the Notes.
// 6) If a ProgName is used as a Cond, it must be defined in the Notes as a
//    BoolExp or as a Java program that returns a Boolean value.
// 7) All CallVarNames parsed in Unconds must be declared as call variables
   in
//    the Notes, with the exception of one parsed preceding "tset", which
//    must be declared as a timer variable in the Notes.
// 8) If a ProgName is used as an Uncond, it must be defined in the Notes as
    an
//    DataAssign or as a void Java Program.
// 9) If a ProgName is used as an argument of a cls(), it must be defined in
//    the Notes as a CallClass.

void ArcBody() : {} {
   CondList() ( "/" UncondSeq() )?
}

void CondList() : {} {
   Cond() ( "," Cond() )*
}

void Cond() : {} {
   <RCV> "(" CallVarName() ")" ( "{" FieldInfo() "}" )? |
   <GONE> "(" CallVarName() ")" |
   LOOKAHEAD(3)
   CallVarName() "?" <TOUT> |
   LOOKAHEAD(6)
   CallVarName() ( "[" SlotName() "]" )? "?" SignalExp() |
   LOOKAHEAD(2)
   BoolExp() |
   ProgName() |
   "!"
}

void UncondSeq() : {} {
```

```
    Uncond() ( ";" Uncond() )*
}

void Uncond() : {} {
    <NEW> "(" CallVarName() ")" "{" FieldInfo() "}" |
    <CTU> "(" CallVarName() "," CallVarName() ")"
            ( "{" FieldInfo() "}" )?                    |
    <REV> "(" CallVarName() "," CallVarName() ")"
            ( "{" FieldInfo() "}" )?                    |
    <END> "(" CallVarName() ")"                         |
    <CLS> "(" CallVarName() "," ProgName() ")"          |
    LOOKAHEAD(3)
    CallVarName() "!" <TSET> "{" FieldInfo() "}"        |
    LOOKAHEAD(2)
    CallVarName() ( "[" SlotName() "]" )? "!" SignalExp() |
    LOOKAHEAD(3)
    CallAssign()     |
    LOOKAHEAD(3)
    SignalAssign()   |
    LOOKAHEAD(2)
    StringAssign()   |
    ProgName()
}

// STATEMENTS
// Additional Syntactic Constraints:
// 1) In a CallAssign, the numbers of terms on both sides of the "=" must
//    be the same.
// 2) A CallAssign must preserve the property that no two call variables
//    have the same value, unless the common value is NoCall ("-").
// 3) A SignalName on the left side of a SignalAssign must be declared as a
//    signal variable name in the Notes.

void CallAssign() : {} {
    CallVarName() ( "," CallVarName() )* "=" CallList()
}

void CallList() : {} {
    CallExp() ( "," CallExp() )*
}

void SignalAssign() : {} {
    SignalName() "=" SignalExp()
}
```

```
void StringAssign() : {} {
   StringVarName() "=" StringExp()
}

// EXPRESSIONS
// Additional Syntactic Constraints:
// 1) If a ProgName is used as FieldInfo, it must be defined in the Notes as
//    FieldInfo.
// 2) A SignalName might be a signal type such as "avail" or it might be the
//    name of a signal variable. I am putting them in the same name space
//    because it would be very confusing to allow a programmer to use "avail
   "
//    as the name of a signal variable. It also makes programming easier.
// 3) Usually we can tell from how it is used in a signal expression whether
//    a SignalName is a signal type or a variable:
//       s{src=me}   s is a signal type, this is a signal literal
//       c:s         s is a signal type, this expression is a signal
   variable
//       s+{src=me}  s is a signal variable, this expression uses an
   override
// 4) However, if a SignalName s is used by itself as a signal expression,
   we
//    can't tell from context whether s is a signal name and the expression
//    is a literal, or whether s is a signal variable. In this case we must
//    look to see whether s is declared as a signal variable name.

void BoolExp() : {} {
   ConjunctExp() ( "||" ConjunctExp() )*
}

void ConjunctExp() : {} {
   EqualityExp() ( "&&" EqualityExp() )*
}

void EqualityExp() : {} {
   ( LOOKAHEAD(2) StringExp() | CallExp() )
      ( "==" | "!=" ) ( LOOKAHEAD(2) StringExp() | CallExp() ) |
   ( "!" )? "(" BoolExp() ")"
}

void CallExp() : {} {
   CallVarName() | "-"
}

void SignalExp() : {} {
```

```
   LOOKAHEAD(2)
   SignalName() "{" FieldInfo() "}" |
   LOOKAHEAD(2)
   CallVarName() ( "[" SlotName() "]" )? ":" SignalName()
                 ( "+" "{" FieldInfo() "}" )?
                 ( "-" "{" FieldName() ( "," FieldName() )* "}" )? |
   SignalName() ( "+" "{" FieldInfo() "}" )?
                ( "-" "{" FieldName() ( "," FieldName() )* "}" )?
}

void FieldInfo() : {} {
   LOOKAHEAD(2)
   FieldPair() ( "," FieldPair() )* |
   ProgName()
}

void FieldPair() : {} {
   FieldName() "=" ( LOOKAHEAD(2) StringExp() | CallExp() )
}

void StringExp() : {} {
   LOOKAHEAD(2)
   SignalExp() "." FieldName() | <SUBS> | <STRINGLIT> | StringVarName()
}

// THE NOTES
// Additional Syntactic Constraints:
// 1) FieldInfo in a Program must be a real list of fields and values, not
   just
//    a ProgName.
// 2) A ProgName used as a call class in a declaration must be defined in
   the
//    ProgramPart as a CallClass.
// 3) A ProgName defined in the ProgramPart as "{}" is defining a CallClass
   with
//    no slots.

void BoxtalkNotes() : {} {
   ( <BOUND> | <FREE> ) BoxName() "{" DeclPart() ( ProgramPart() )? "}"
}

void DeclPart() : {} {
   Decl() ( LOOKAHEAD(2) ";" Decl() )*
}
```

```
void ProgramPart() : {} {
  ";" ( Program() )+
}

void Decl() : {} {
  ( <CALL> CallVarName() ( "," CallVarName() )* ( <CLASS> ProgName() )? ) |
  ( <TIMER> CallVarName() ( "," CallVarName() )* ) |
  ( <SIGNAL> SignalName() ( "," SignalName() )* ) |
  ( <STRING> StringVarName() ( "," StringVarName() )* )
}

void Program() : {} {
  <CLASS> ProgName() "{" CallClass() "}" |
  <SET> SignalName() "{" SignalName() ( "," SignalName() )+ "}" |
  ProgName() "{" (
     LOOKAHEAD(2) BoolExp() |
     LOOKAHEAD(3) DataAssign() |
     FieldInfo() |
     <JAVALIT>
  )? "}"
}

void DataAssign() : {} {
  ( LOOKAHEAD(3) SignalAssign() | StringAssign() )
     ( ";" ( LOOKAHEAD(3) SignalAssign() | StringAssign() ) )*
}

void CallClass() : {} {
  <SLOT> SlotName() "=" MediaName() "[" ( <DIGITS> | "*" ) "]"
  ( ";" <SLOT> SlotName() "=" MediaName() "[" ( <DIGITS> | "*" ) "]" )*
}

// NAME SPACES
void BoxName() : {} { <ID> }
void StateName() : {} { <ID> }
void CallVarName() : {} { <ID> }
void SignalName() : {} {
  <SETUP> | <AVAIL> | <UNAVAIL> | <UNKNOWN> | <NONE> | <STAT> | <OUT> |
  <NOSIG> | <ID> }
void FieldName() : {} {
  <SRC> | <TRG> | <DLD> | <ZONE> | <TYPE> | <ID> }
void StringVarName() : {} { <ID> }
void ProgName() : {} { <ID> }
void SlotName() : {} { <ID> }
void MediaName() : {} { <VOICE> | <TEXT> | <VIDEO> }
```

Appendix B

Modified Grammar

The bold and italicized font mark our changes to the original grammar

%glr-parser	
union {	char *string;
	char symbol;
	int number; }
{	/* Declarations (data structures, functions)*/ }
token <string>	ID STRINGLIT SETUP AVAIL UNAVAIL UNKNOWN
	NONE OUT STAT NOSIG SRC TRG DLD ZONE TYPE SUBS
	BOOL VOICE VIDEO TEXT
token <symbol>	‘{’ ‘}’ ‘(’ ‘)’ ‘[’ ‘]’ ‘<’ ‘>’ ‘,’ ‘;’ ‘/’ ‘?’ ‘!’ ‘=’ ‘-’ ‘+’ ‘:’ ‘.’ ‘*’
token <number>	NUMBER
token	GRAPH INIT STABLE TRANSIENT CTU NEW RCV GONE END
	ARC TRANS NOTES TERM REV SIGNAL TOUT TSET
token	SET TIMER BOUND CALL CLASS CLS STRING FREE AND
	NOTEQUAL OR EQUAL SLOT
type	<string> statename callvarname linkageobjs linkageobj
	slotname tvarname stablestateitem linkage
type	<string> fieldname progname signalname callvarnamess
	stringvarnamess transientstate
type	<string> fieldnames fieldpair fieldinfo stringexp
	stringvarname signalexp tvarnames signalnamess
	callexp calllist callvarnames boolvarname boolvarnames
type <string>	boxname signaln medianame

input: ***NOTES ‘{’ boxtalknotes ‘}’ GRAPH ‘{’ graphitemsets ‘}’***
| ***FREE boxname NOTES*** ‘{’ boxtalknotes ‘}’ GRAPH ‘{’ graphitemsets ‘}’
| ***BOUND boxname NOTES*** ‘{’ boxtalknotes ‘}’ GRAPH ‘{’ graphitemsets ‘}’

graphitemsets: graphitemset
| graphitemsets graphitemset

graphitemset: responsivestate
| transientstate
| arc

responsivestate: INIT statename ‘{’ ‘}’
| TERM statename ‘{’ ‘}’
| STABLE statename ‘{’ ‘}’
| STABLE statename ‘{’ stablestateitems ‘}’

stablestateitems: stablestateitem
| stablestateitems ‘,’ stablestateitem

stablestateitem: callvarname
| ‘(’ linkage ‘)’

linkage: linkageobj ‘<’ linkageobj
| linkageobj ‘>’ linkageobj
| linkageobjs

linkageobjs: linkageobj
| linkageobjs ‘,’ linkageobj

linkageobj: callvarname
| callvarname ‘[’ slotname ‘]’

transientstate: TRANSIENT statename ‘{’ ‘}’
| TRANSIENT statename ‘{’ progname ‘}’

arc: ARC ‘{’ statename TRANS statename ‘{’ arcbody ‘}’ ‘}’
| ***ARC ‘{’ statename ‘{’ arcbody ‘}’ statename ‘}’***

arcbody: condlist
| condlist '/' uncondseq

condlist: cond
| condlist ',' cond

cond: RCV '(' callvarname ')'
| RCV '(' callvarname ')' '{' fieldinfo '}'
| GONE '(' callvarname ')'
| callvarname '?' TOUT
| callvarname '?' signalexp
| callvarname '[' slotname ']' '?' signalexp
| boolexp
| progname
| ***signalname '(' callvarname '[' slotname ']' ')'***
| ***signalname '(' callvarname ')'***
| '!'

uncondseq: uncond
| uncondseq ';' uncond

uncond: NEW '(' callvarname ')' '{' fieldinfo '}'
| CTU '(' callvarname ',' callvarname ')'
| CTU '(' callvarname ',' callvarname ')' '{' fieldinfo '}'
| REV '(' callvarname ',' callvarname ')'
| REV '(' callvarname ',' callvarname ')' '{' fieldinfo '}'
| END '(' callvarname ')'
| CLS '(' callvarname ',' progname ')'
| callvarname '!' TSET '{' fieldinfo '}'
| callvarname '!' signalexp
| callvarname '[' slotname ']' '!' signalexp
| progname
| ***fieldpairs***

```
boolexp:       conjunctexp
               | boolexp OR conjunctexp

conjunctexp:   equalityexp
               | conjunctexp AND equalityexp

equalityexp:   stringexp EQUAL stringexp
               | stringexp NOTEQUAL stringexp
               | stringexp EQUAL ‘-’
               | stringexp NOTEQUAL ‘-’
               | ‘-’ EQUAL stringexp
               | ‘-’ NOTEQUAL stringexp
               | ‘(’ boolexp ‘)’
               | ‘!’ ‘(’ boolexp ‘)’
```

signalexp: signalname '{' fieldinfo '}'
| callvarname ':' signalname
| callvarname '[' slotname ']' ':' signalname
| callvarname ':' signalname '+' '{' fieldinfo '}'
| callvarname '[' slotname ']' ':' signalname '+' '{' fieldinfo '}'
| callvarname ':' signalname '-' '' fieldnames ''
| callvarname '[' slotname ']' ':' signalname '-' '{' fieldnames '}'
| signalname
| signalname '+' '{' fieldinfo '}'
| signalname '' '{' fieldnames '}'

fieldnames: fieldname
| fieldnames ',' fieldname

fieldinfo: fieldpair
| fieldinfo ',' fieldpair

fieldpairs: fieldpair
| fieldpairs ',' fieldpair

fieldpair: fieldname '=' stringexp
| ***fieldname '=' '-'***
| ***signaln '=' signalexp***

stringexp: signalexp '.' fieldname
| SUBS
| STRINGLIT
| stringvarname

boxtalknotes:	FREE boxname ‘{’ declpart ‘}’
	| BOUND boxname ‘{’ declpart ‘}’
	| FREE boxname ‘{’ declpart programpart ‘}’
	| BOUND boxname ‘{’ declpart programpart ‘}’
	| declpart programpart
	| declpart
declpart:	decl
	| declpart ‘;’ decl
programpart:	‘;’ programpt
programpt:	program
	| programpt program
decl:	CALL callvarnamess
	| CALL callvarnamess CLASS progname
	| CLASS progname CALL callvarnamess
	| TIMER tvarnames
	| SIGNAL signalnamess
	| STRING stringvarnamess
callvarnamess:	callvarname
	| callvarnamess ‘,’ callvarname
program:	CLASS progname ‘{’ ‘}’
	| CLASS progname ‘{’ callclass ‘}’
	| SET signalname ‘{’ signalnames ‘}’
	| progname ‘{’ boolexp ‘}’
	| progname ‘{’ fieldpairz ‘}’
	| progname ‘{’ STRINGLIT ‘}’
	| progname ‘{’ ‘}’

fieldpairz: fieldpair
| fieldpairz ‘;’ fieldpair

signalnamess: signalname
| signalnamess ‘,’ signalname

stringvarnamess: stringvarname
| stringvarnamess ‘,’ stringvarname

tvarnames: tvarname
| tvarnames ‘,’ tvarname

signalnames: signalname
| signalnames ‘,’ signalname

callcls: SLOT slotname ‘=’ medianame ‘[’ NUMBER ‘]’
| SLOT slotname ‘=’ medianame ‘[’ ‘*’ ‘]’

callclass: callcls
| callclass ‘;’ callcls
statename: ID

callvarname: ID

stringvarname: ID

progname: ID

slotname: ID

tvarname: ID

medianame:	VOICE
	| TEXT
	| VIDEO
signalname:	SETUP
	| AVAIL
	| UNAVAIL
	| UNKNOWN
	| NONE
	| STAT
	| OUT
	| NOSIG
	| ID
signaln:	SETUP
	| AVAIL
	| UNAVAIL
	| UNKNOWN
	| NONE
	| STAT
	| OUT
	| NOSIG
fieldname:	SRC
	| TRG
	| DLD
	| ZONE
	| TYPE
	| ID
boxname:	ID

Appendix C

Promela model - Free Transparent Box

```
/*-----------------------------------------------------------*/
/*   FreeTransparentBox   */
/*-----------------------------------------------------------*/

/*   type definitions   */

mtype = { teardown , downack , other , setup , upack };
mtype = { initial , connecting_o , transparent , abandonConnectiono , terminating_o ,
    final , terminating_i , error };

 typedef Transition {
      mtype dest;
      chan in_chan;
      bool en_flag = false;
  };

 typedef in_q {
    byte box_in = 0;
    byte i_in = 1;
    byte o_in = 2;
    bool box_in_ready = true;
    bool i_in_ready = false;
    bool o_in_ready = false;
    byte selected
 };
 chan glob_ins[3] = [0] of {mtype};
```

```
typedef out_q {
  byte box_out = 0;
  byte i_out = 1;
  byte o_out = 2;
  chan o_hold = [5] of {mtype};
};
chan glob_outs[3] = [0] of {mtype};

typedef SnapShot {
  mtype cs;
  in_q inq;
  out_q out
 };

/*-----------------------------------------------------*/
/*  global variable declarations  */
/*-----------------------------------------------------*/

mtype sig;
SnapShot ss;
  Transition t[14];

/*  Global Monitor Variables  */
bool rcv_setup = false;
bool send_upack = false;
bool o_send_setup = false;
bool o_rcv_upack = false;
bool i_rcv_teardown = false;
bool i_send_downack = false;
bool o_send_teardown = false;
bool o_rcv_teardown = false;
bool o_send_downack = false;
bool i_send_teardown = false;
bool o_rcv_downack = false;
bool i_rcv_downack = false;

/*------------------------------------------------------*/
/*  Inline Functions  */

inline dump(c1 , c2) {
    byte aSig;
    do
    ::c1 ? aSig -> c2 ! aSig;
    ::empty(c1) -> break;
```

```
      od
  };

inline reset() {
  rcv_setup = false;
  send_upack = false;
  o_send_setup = false;
  o_rcv_upack = false;
  i_rcv_teardown = false;
  i_send_downack = false;
  o_send_teardown = false;
  o_rcv_teardown = false;
  o_send_downack = false;
  i_send_teardown = false;
  o_rcv_downack = false;
  i_rcv_downack = false;

  if
  :: glob_ins[ss.inq.box_in] ? sig -> ss.inq.selected = ss.inq.box_in;
  :: glob_ins[ss.inq.i_in] ? sig -> ss.inq.selected = ss.inq.i_in;
  :: glob_ins[ss.inq.o_in] ? sig -> ss.inq.selected = ss.inq.o_in;
  fi
};

inline en_events(n) {
    glob_ins[ss.inq.selected] == t[n].in_chan;
};


inline en_cond(n) {
if
    ::(n == 0) && (sig == setup );

  ::(n == 1) && (sig == upack );
  ::(n == 2) && (sig == teardown );
  ::(n == 3) && ( sig != teardown && nfull(ss.out.o_hold) );
  ::(n == 4) && ( sig != teardown && full(ss.out.o_hold) );

  ::(n == 5) && (sig == teardown );
  ::(n == 6) && (sig == teardown );
  ::(n == 7) && ( sig != teardown  );
  ::(n == 8) && ( sig != teardown  );

  ::(n == 9) && (sig == upack );

```

```
  ::(n == 10) && (sig == downack );
  ::(n == 11) && (sig == teardown );


  ::(n == 12) && (sig == downack );
  ::(n == 13) && (sig == teardown );
fi;
};

inline next_trans(n) {
if


  ::(n == 0) ->        rcv_setup = true;
          ss.inq.i_in_ready = true;
         glob_outs[ss.out.i_out] ! upack;
          send_upack = true;
         glob_outs[ss.out.o_out] ! setup;
          ss.inq.o_in_ready = true;
          o_send_setup = true;
          ss.cs = t[0].dest;

  ::(n == 1) ->
          o_rcv_upack = true;
          dump(ss.out.o_hold , glob_outs[ss.out.o_out]);
          ss.cs = t[1].dest;

  ::(n == 2) ->
          i_rcv_teardown = true;
          i_send_downack = true;
         glob_outs[ss.out.i_out] ! downack;
          ss.inq.i_in_ready = false;
          o_send_teardown = true;
         glob_outs[ss.out.o_out] ! teardown;
          ss.cs = t[2].dest;

  ::(n == 3) ->
         ss.out.o_hold ! sig;
          ss.cs = t[3].dest;

  ::(n == 4) ->
          ss.inq.i_in_ready = false;
          ss.inq.o_in_ready = false;
          ss.cs = t[4].dest;

```

```
::(n == 5) ->
        i_rcv_teardown = true;
        i_send_downack = true;
       glob_outs[ss.out.i_out] ! downack;
        ss.inq.i_in_ready = false;
        o_send_teardown = true;
       glob_outs[ss.out.o_out] ! teardown;
        ss.cs = t[5].dest;

::(n == 6) ->
        o_rcv_teardown = true;
        o_send_downack = true;
       glob_outs[ss.out.o_out] ! downack;
        ss.inq.o_in_ready = false;
        i_send_teardown = true;
       glob_outs[ss.out.i_out] ! teardown;
        ss.cs = t[6].dest;

::(n == 7) ->
       glob_outs[ss.out.o_out] ! sig;
        ss.cs = t[7].dest;

::(n == 8) ->
       glob_outs[ss.out.i_out] ! sig;
        ss.cs = t[8].dest;

::(n == 9) ->
        o_rcv_upack = true;
        dump(ss.out.o_hold , glob_outs[ss.out.o_out]);
        ss.cs = t[9].dest;

::(n == 10) ->
        o_rcv_downack = true;
        ss.inq.o_in_ready = false;
        ss.cs = t[10].dest;

::(n == 11) ->
        o_rcv_teardown = true;
        o_send_downack = true;
       glob_outs[ss.out.o_out] ! downack;
        ss.cs = t[11].dest;

::(n == 12) ->
        i_rcv_downack = true;
        ss.inq.i_in_ready = false;
```

```
            ss.cs = t[12].dest;

  ::(n == 13) ->
            i_rcv_teardown = true;
            i_send_downack = true;
           glob_outs[ss.out.i_out] ! downack;
            ss.cs = t[13].dest;

fi;
};

 inline en_trans(n) {
   if
   ::en_events(n) ->
     if
     ::en_cond(n) -> t[n].en_flag = true;
     ::else -> t[n].en_flag = false;
     fi;
   ::else -> t[n].en_flag = false;
   fi;
 };

active proctype FreeTransparentBox() {

  ss.cs = initial;
   t[0].dest = connecting_o;
   t[0].in_chan = glob_ins[ss.inq.box_in];
   t[1].dest = transparent;
   t[1].in_chan = glob_ins[ss.inq.o_in];
   t[2].dest = abandonConnectiono;
   t[2].in_chan = glob_ins[ss.inq.i_in];
   t[3].dest = connecting_o;
   t[3].in_chan = glob_ins[ss.inq.i_in];
   t[4].dest = error;
   t[4].in_chan = glob_ins[ss.inq.i_in];
   t[5].dest = terminating_o;
   t[5].in_chan = glob_ins[ss.inq.i_in];
   t[6].dest = terminating_i;
   t[6].in_chan = glob_ins[ss.inq.o_in];
   t[7].dest = transparent;
   t[7].in_chan = glob_ins[ss.inq.i_in];
   t[8].dest = transparent;
   t[8].in_chan = glob_ins[ss.inq.o_in];
   t[9].dest = terminating_o;
   t[9].in_chan = glob_ins[ss.inq.o_in];
```

```
    t[10].dest = final;
    t[10].in_chan = glob_ins[ss.inq.o_in];
    t[11].dest = terminating_o;
    t[11].in_chan = glob_ins[ss.inq.o_in];
    t[12].dest = final;
    t[12].in_chan = glob_ins[ss.inq.i_in];
    t[13].dest = terminating_i;
    t[13].in_chan = glob_ins[ss.inq.i_in];


initial_state:
 atomic {
      reset();
      en_trans(0);

      if
      ::t[0].en_flag -> next_trans(0); goto connecting_o_state;
      ::else -> goto initial_state;

      fi;
}


connecting_o_state:
 atomic {
      reset();
      en_trans(1);
      en_trans(2);
      en_trans(3);
      en_trans(4);

      if
      ::t[1].en_flag -> next_trans(1); goto transparent_state;
      ::t[2].en_flag -> next_trans(2); goto abandonConnectiono_state;
      ::t[3].en_flag -> next_trans(3); goto connecting_o_state;
      ::t[4].en_flag -> next_trans(4); goto error_state;
      ::else -> goto connecting_o_state;

      fi;
}


transparent_state:
 atomic {
      reset();
```

```
        en_trans(5);
        en_trans(6);
        en_trans(7);
        en_trans(8);

        if
        ::t[5].en_flag -> next_trans(5); goto terminating_o_state;
        ::t[6].en_flag -> next_trans(6); goto terminating_i_state;
        ::t[7].en_flag -> next_trans(7); goto transparent_state;
        ::t[8].en_flag -> next_trans(8); goto transparent_state;
        ::else -> goto transparent_state;

        fi;
}


abandonConnectiono_state:
 atomic {
        reset();
        en_trans(9);

        if
        ::t[9].en_flag -> next_trans(9); goto terminating_o_state;
        ::else -> goto abandonConnectiono_state;

        fi;
}


terminating_o_state:
 atomic {
        reset();
        en_trans(10);
        en_trans(11);

        if
        ::t[10].en_flag -> next_trans(10); goto final_state;
        ::t[11].en_flag -> next_trans(11); goto terminating_o_state;
        ::else -> goto terminating_o_state;

        fi;
}


terminating_i_state:
```

```
 atomic {
      reset();
      en_trans(12);
      en_trans(13);

      if
      ::t[12].en_flag -> next_trans(12); goto final_state;
      ::t[13].en_flag -> next_trans(13); goto terminating_i_state;
      ::else -> goto terminating_i_state;

      fi;
}

      error_state:
   final_state:
      progress:

      skip;
};

active proctype env() {
 mtype i_sigt  ,o_sigt   , o_sigu  ;



end:  do

      ::  ss.inq.box_in_ready ->
          ss.inq.box_in_ready = false;
          glob_ins[ss.inq.box_in] ! setup;

      ::ss.inq.i_in_ready ->
        if
        ::  glob_ins[ss.inq.i_in] ! teardown;
        ::  glob_ins[ss.inq.i_in] ! other;
        fi unless {
           (i_sigt == teardown) ->
             glob_ins[ss.inq.i_in] ! downack;
             i_sigt = 0;
        }
      ::ss.inq.o_in_ready ->
        if
        ::  glob_ins[ss.inq.o_in] ! teardown;
        ::  glob_ins[ss.inq.o_in] ! other;
        fi unless {
```

```
            if
            ::(o_sigu == upack) ->
             glob_ins[ss.inq.o_in] ! upack;
             o_sigu = 0;
            ::(o_sigt == teardown && o_sigu == 0) ->
              glob_ins[ss.inq.o_in] ! downack;
             o_sigt = 0;
            fi;
            }
    od
    unless {
        if
        ::atomic { glob_outs[ss.out.o_out] ? setup ->
         o_sigu = upack;
         }
        ::glob_outs[ss.out.i_out] ? upack;
        ::glob_outs[ss.out.i_out] ? downack;
        ::atomic { glob_outs[ss.out.i_out] ? teardown ->
         i_sigt = teardown;
         }
        ::glob_outs[ss.out.i_out] ? other;
        ::atomic { glob_outs[ss.out.o_out] ? teardown ->
         o_sigt = teardown;
         }
        ::glob_outs[ss.out.o_out] ? downack;
        ::glob_outs[ss.out.o_out] ? other;
        fi;
    }
    goto end;
  }
ltl p0 {(!FreeTransparentBox@error_state) && [](rcv_setup -> <> send_upack)}
ltl p1 {(!FreeTransparentBox@error_state) && [](i_rcv_teardown -> <> i_send_downack)}
ltl p2 {(!FreeTransparentBox@error_state) && [](i_send_teardown -> <> i_rcv_downack)}
ltl p3 {(!FreeTransparentBox@error_state) && [](o_send_setup -> <> o_rcv_upack)}
ltl p4 {(!FreeTransparentBox@error_state) && [](o_rcv_teardown -> <> o_send_downack)}
ltl p5 {(!FreeTransparentBox@error_state) && [](o_send_teardown -> <> o_rcv_downack)}
```

Appendix D

Promela model - Bound Transparent Box

```
/*--------------------------------------------------------------*/
/*  BoundTransparentBox */
/*--------------------------------------------------------------*/

/*  type definitions  */

mtype = { teardown , downack , other , setup , upack  , unavail };

 mtype = { post_process_t , post_process_s , post_process_f };
   mtype = { initial , orienting , connecting_f , deciding_1 , transparent ,
       connecting_s , deciding_2 , receiving , error };

   mtype = { idle , t_work , s_work , f_wait_up , f_work ,
        s_wait_up };

  typedef Transition {
      mtype dest;
      chan in_chan;
      bool en_flag = false;
   };

  typedef in_q {
    byte box_in = 0;
    byte old_t_in = 1;
    byte old_s_in = 2;
    byte old_f_in = 3;
```

```
  byte t_in = 4;
  byte s_in = 5;
  byte f_in = 6;
  bool box_in_ready = true;
  bool old_t_in_ready = false;
  bool old_s_in_ready = false;
  bool old_f_in_ready = false;
  bool t_in_ready = false;
  bool s_in_ready = false;
  bool f_in_ready = false;
  byte selected
};
chan glob_ins[7] = [0] of {mtype};

typedef out_q {
   byte box_out = 0;
  byte old_t_out = 1;
  byte old_s_out = 2;
  byte old_f_out = 3;
  byte t_out = 4;
  byte s_out = 5;
  byte f_out = 6;
  chan f_hold = [5] of {mtype};
  chan s_hold = [5] of {mtype};
};
chan glob_outs[7] = [0] of {mtype};

typedef internal {
  chan internal_t = [0] of {mtype};
  chan internal_s = [0] of {mtype};
  chan internal_f = [0] of {mtype};
  };

typedef SnapShot {
  mtype cs;
  mtype cs_post_process;
  in_q inq;
  out_q out;
  internal intq;};

/*------------------------------------------------------------*/

/*  Global Variable Declarations  */

SnapShot ss;
```

```
mtype sig;
mtype inter_sig;
bool t_from_subs = true;
bool current_t_from_subs = true;
bool s_communicating = true;
bool old_s_communicating = true;
bool f_communicating = true;
bool old_f_communicating = true;
Transition t[36];

/*  Global Monitor Variables  */

bool rcv_setup = false;
bool send_upack = false;
bool f_send_setup = false;
bool s_send_setup = false;
bool f_rcv_upack = false;
bool s_rcv_teardown = false;
bool s_send_downack = false;
bool f_send_teardown = false;
bool t_send_unavail = false;
bool t_send_teardown = false;
bool s_send_teardown = false;
bool f_rcv_teardown = false;
bool f_send_downack = false;
bool s_rcv_upack = false;


 byte counter = 0;
 byte last_call = 0;
 byte pp_call = 0;
 byte current_call = 0;

inline dump(c1 , c2) {
     byte aSig;
     do
      ::c1 ? aSig -> c2 ! aSig;
      ::empty(c1) -> break;
     od
 };

inline setup_initial(b) {
   ss.inq.s_in_ready = true;
   ss.inq.f_in_ready = true;
    if
```

```
    ::(b) ->
       s_communicating = true;
       f_communicating = false;
    ::(!b) ->
       s_communicating = false;
       f_communicating = true;
    fi;
};

inline teardown_cleanup(c) {
    if
    ::(c == 0) ->
      ss.inq.old_t_in_ready = true;
    ::(c == 1) ->
      ss.inq.s_in_ready = false;
      ss.inq.old_s_in_ready = true;
      old_s_communicating = s_communicating;
    ::(c == 2) ->
      ss.inq.f_in_ready = false;
      ss.inq.old_f_in_ready = true;
      old_f_communicating = f_communicating;
    fi;
};

inline reset() {
rcv_setup = false;
send_upack = false;
f_send_setup = false;
s_send_setup = false;
f_rcv_upack = false;
s_rcv_teardown = false;
s_send_downack = false;
f_send_teardown = false;
t_send_unavail = false;
t_send_teardown = false;
s_send_teardown = false;
f_rcv_teardown = false;
f_send_downack = false;
s_rcv_upack = false;

  if
  :: glob_ins[ss.inq.box_in] ? sig -> ss.inq.selected = ss.inq.box_in;
  :: glob_ins[ss.inq.s_in] ? sig -> ss.inq.selected = ss.inq.s_in;
  :: glob_ins[ss.inq.f_in] ? sig -> ss.inq.selected = ss.inq.f_in;
  fi
```

```
};

inline en_events(n) {
 if
    ::(n == 0) && ss.inq.selected == ss.inq.box_in;
    ::(n == 1) && true;
    ::(n == 2) && true;
    ::(n == 3) && ss.inq.selected == ss.inq.box_in;
    ::(n == 4) && ss.inq.selected == ss.inq.f_in;
    ::(n == 5) && ss.inq.selected == ss.inq.s_in;
    ::(n == 6) && ss.inq.selected == ss.inq.s_in;
    ::(n == 7) && ss.inq.selected == ss.inq.s_in;
    ::(n == 8) && true;
    ::(n == 9) && true;
    ::(n == 10) && ss.inq.selected == ss.inq.box_in;
    ::(n == 11) && ss.inq.selected == ss.inq.f_in;
    ::(n == 12) && ss.inq.selected == ss.inq.s_in;
    ::(n == 13) && ss.inq.selected == ss.inq.f_in;
    ::(n == 14) && ss.inq.selected == ss.inq.s_in;
    ::(n == 15) && ss.inq.selected == ss.inq.box_in;
    ::(n == 16) && ss.inq.selected == ss.inq.s_in;
    ::(n == 17) && ss.inq.selected == ss.inq.f_in;
    ::(n == 18) && ss.inq.selected == ss.inq.f_in;
    ::(n == 19) && ss.inq.selected == ss.inq.f_in;
    ::(n == 20) && true;
    ::(n == 21) && true;
    ::(n == 22) && true;
    ::(n == 23) && true;
    ::(n == 24) && true;
    ::(n == 25) && true;
    ::(n == 26) && true;
    ::(n == 27) && true;
    ::(n == 28) && true;
    ::(n == 29) && ss.inq.selected == ss.inq.old_t_in;
    ::(n == 30) && ss.inq.selected == ss.inq.old_s_in;
    ::(n == 31) && ss.inq.selected == ss.inq.old_s_in;
    ::(n == 32) && ss.inq.selected == ss.inq.old_f_in;
    ::(n == 33) && ss.inq.selected == ss.inq.old_f_in;
    ::(n == 34) && ss.inq.selected == ss.inq.old_f_in;
    ::(n == 35) && ss.inq.selected == ss.inq.old_s_in;
 fi;
};

inline reset_pp_t() {
  glob_ins[ss.inq.old_t_in] ? sig -> ss.inq.selected = ss.inq.old_t_in
```

```
};

inline reset_pp_s() {
  glob_ins[ss.inq.old_s_in] ? sig -> ss.inq.selected = ss.inq.old_s_in
};

inline reset_pp_f() {
  glob_ins[ss.inq.old_f_in] ? sig -> ss.inq.selected = ss.inq.old_f_in
};


inline en_cond(n) {
if
  ::(n == 0) && ( sig == setup );

  ::(n == 1) && current_t_from_subs;
  ::(n == 2) && !current_t_from_subs;

  ::(n == 3) && ( sig == setup );
  ::(n == 4) && ( sig == upack );
  ::(n == 5) && ( sig == teardown ) && !ss.inq.old_f_in_ready;
  ::(n == 6) && ( sig != teardown && nfull(ss.out.f_hold) );
  ::(n == 7) && ( sig != teardown && full(ss.out.f_hold) );

  ::(n == 8) && !current_t_from_subs;
  ::(n == 9) && current_t_from_subs;

  ::(n == 10) && ( sig == setup );
  ::(n == 11) && ( sig == teardown );
  ::(n == 12) && ( sig == teardown ) && !ss.inq.old_f_in_ready;
  ::(n == 13) && ( sig != teardown  );
  ::(n == 14) && ( sig != teardown  );

  ::(n == 15) && ( sig == setup );
  ::(n == 16) && ( sig == upack );
  ::(n == 17) && ( sig == teardown );
  ::(n == 18) && ( sig != teardown && nfull(ss.out.s_hold) );
  ::(n == 19) && ( sig != teardown && full(ss.out.s_hold) );

  ::(n == 20) && !current_t_from_subs;
  ::(n == 21) && current_t_from_subs;

  ::(n == 22) && current_t_from_subs;
  ::(n == 23) && !current_t_from_subs;

```

```
   ::(n == 24) && (inter_sig == post_process_t);
   ::(n == 25) && (inter_sig == post_process_s) && old_s_communicating;
   ::(n == 26) && (inter_sig == post_process_f) && !old_f_communicating;
   ::(n == 27) && (inter_sig == post_process_f) && old_f_communicating;
   ::(n == 28) && (inter_sig == post_process_s) && !old_s_communicating;
   ::(n == 29) && sig == downack;
   ::(n == 30) && sig == downack;
   ::(n == 31) && sig == teardown;
   ::(n == 32) && sig == upack;
   ::(n == 33) && sig == downack;
   ::(n == 34) && sig == teardown;
   ::(n == 35) && sig == upack;
 fi;
};

/*---------------------------------------------------------*/

/*  Inline Functions  */

inline next_trans(n) {
if


  ::(n == 0) ->      rcv_setup = true;
      glob_outs[ss.out.t_out] ! upack;
      send_upack = true;
      current_t_from_subs = t_from_subs;
      last_call = current_call;
      current_call = counter;
      ss.cs = t[0].dest;

  ::(n == 1) ->      f_send_setup = true;
      glob_outs[ss.out.f_out] ! setup;
      setup_initial(current_t_from_subs)
      ss.cs = t[1].dest;

  ::(n == 2) ->      s_send_setup = true;
      glob_outs[ss.out.f_out] ! setup;
      setup_initial(current_t_from_subs)
      ss.cs = t[2].dest;

  ::(n == 3) ->      rcv_setup = true;
      current_t_from_subs = t_from_subs;
      last_call = current_call;
      current_call = counter;
```

```
        send_upack = true;
        glob_outs[ss.out.t_out] ! upack;
        ss.cs = t[3].dest;

    ::(n == 4) ->      f_rcv_upack = true;
        dump(ss.out.f_hold , glob_outs[ss.out.f_out]);
        f_communicating = true;
        ss.cs = t[4].dest;

    ::(n == 5) ->      s_rcv_teardown = true;
        s_send_downack = true;
        glob_outs[ss.out.s_out] ! downack;
        f_send_teardown = true;
        glob_outs[ss.out.f_out] ! teardown;
        pp_call = current_call;
        ss.intq.internal_f ! post_process_f;
        ss.cs = t[5].dest;

    ::(n == 6) ->      ss.out.f_hold ! sig;
        ss.cs = t[6].dest;

    ::(n == 7) ->      ss.inq.s_in_ready = false;
        ss.inq.f_in_ready = false;
        ss.cs = t[7].dest;

    ::(n == 8) ->      t_send_unavail = true;
        glob_outs[ss.out.t_out] ! unavail;
        t_send_teardown = true;
        glob_outs[ss.out.t_out] ! teardown;
        ss.intq.internal_t ! post_process_t;
        ss.cs = t[8].dest;

    ::(n == 9) ->      s_send_teardown = true;
        glob_outs[ss.out.s_out] ! teardown;
        pp_call = last_call;
        ss.intq.internal_s ! post_process_s;
        f_send_teardown = true;
        glob_outs[ss.out.f_out] ! teardown;
        ss.intq.internal_f ! post_process_f;
        f_send_setup = true;
        glob_outs[ss.out.f_out] ! setup;
        setup_initial(current_t_from_subs)
        ss.cs = t[9].dest;

    ::(n == 10) ->     rcv_setup = true;
```

```
        current_t_from_subs = t_from_subs;
        last_call = current_call;
        current_call = counter;
        send_upack = true;
        glob_outs[ss.out.t_out] ! upack;
        ss.cs = t[10].dest;

    ::(n == 11) ->     f_rcv_teardown = true;
        f_send_downack = true;
        glob_outs[ss.out.f_out] ! downack;
        s_send_teardown = true;
        glob_outs[ss.out.s_out] ! teardown;
        pp_call = current_call;
        ss.intq.internal_s ! post_process_s;
        ss.cs = t[11].dest;

    ::(n == 12) ->     s_rcv_teardown = true;
        s_send_downack = true;
        glob_outs[ss.out.s_out] ! downack;
        f_send_teardown = true;
        glob_outs[ss.out.f_out] ! teardown;
        pp_call = current_call;
        ss.intq.internal_f ! post_process_f;
        ss.cs = t[12].dest;

    ::(n == 13) ->     glob_outs[ss.out.s_out] ! sig;
        ss.cs = t[13].dest;

    ::(n == 14) ->     glob_outs[ss.out.f_out] ! sig;
        ss.cs = t[14].dest;

    ::(n == 15) ->     rcv_setup = true;
        current_t_from_subs = t_from_subs;
        last_call = current_call;
        current_call = counter;
        send_upack = true;
        glob_outs[ss.out.t_out] ! upack;
        ss.cs = t[15].dest;

    ::(n == 16) ->     s_rcv_upack = true;
        dump(ss.out.s_hold , glob_outs[ss.out.s_out]);
        s_communicating = true;
        ss.cs = t[16].dest;

    ::(n == 17) ->     f_rcv_teardown = true;
```

```
      f_send_downack = true;
      glob_outs[ss.out.f_out] ! downack;
      s_send_teardown = true;
      glob_outs[ss.out.s_out] ! teardown;
      pp_call = current_call;
      ss.intq.internal_s ! post_process_s;
      ss.cs = t[17].dest;

  ::(n == 18) ->     ss.out.s_hold ! sig;
      ss.cs = t[18].dest;

  ::(n == 19) ->     ss.inq.s_in_ready = false;
      ss.inq.f_in_ready = false;
      ss.cs = t[19].dest;

  ::(n == 20) ->     t_send_unavail = true;
      glob_outs[ss.out.t_out] ! unavail;
      t_send_teardown = true;
      glob_outs[ss.out.t_out] ! teardown;
      ss.intq.internal_t ! post_process_t;
      ss.cs = t[20].dest;

  ::(n == 21) ->     s_send_teardown = true;
      glob_outs[ss.out.s_out] ! teardown;
      pp_call = last_call;
      ss.intq.internal_s ! post_process_s;
      f_send_teardown = true;
      glob_outs[ss.out.f_out] ! teardown;
      ss.intq.internal_f ! post_process_f;
      f_send_setup = true;
      glob_outs[ss.out.f_out] ! setup;
      setup_initial(current_t_from_subs)
      ss.cs = t[21].dest;

  ::(n == 22) ->     s_send_teardown = true;
      glob_outs[ss.out.s_out] ! teardown;
      pp_call = last_call;
      ss.intq.internal_s ! post_process_s;
      f_send_teardown = true;
      glob_outs[ss.out.f_out] ! teardown;
      ss.intq.internal_f ! post_process_f;
      f_send_setup = true;
      glob_outs[ss.out.f_out] ! setup;
      setup_initial(current_t_from_subs)
      ss.cs = t[22].dest;
```

```
  ::(n == 23) ->      t_send_unavail = true;
      glob_outs[ss.out.t_out] ! unavail;
      t_send_teardown = true;
      glob_outs[ss.out.t_out] ! teardown;
      ss.intq.internal_t ! post_process_t;
      ss.cs = t[23].dest;

   ::(n == 24) ->     ss.cs_post_process = t[24].dest;

   ::(n == 25) ->     ss.cs_post_process = t[25].dest;

   ::(n == 26) ->     ss.cs_post_process = t[26].dest;

   ::(n == 27) ->     ss.cs_post_process = t[27].dest;

   ::(n == 28) ->     ss.cs_post_process = t[28].dest;

   ::(n == 29) ->     ss.inq.old_t_in_ready = false;
      ss.cs_post_process = t[29].dest;

   ::(n == 30) ->     ss.inq.old_s_in_ready = false;
      ss.cs_post_process = t[30].dest;

   ::(n == 31) ->     glob_outs[ss.out.old_s_out] ! downack;
       ss.cs_post_process = t[31].dest;

   ::(n == 32) ->     ss.cs_post_process = t[32].dest;

   ::(n == 33) ->     ss.inq.old_f_in_ready = false;
       ss.cs_post_process = t[33].dest;

   ::(n == 34) ->     glob_outs[ss.out.old_f_out] ! downack;
       ss.cs_post_process = t[34].dest;

   ::(n == 35) ->     ss.cs_post_process = t[35].dest;
  fi;
};

 inline en_trans(n) {
   if
   :: en_events(n) ->
     if
     :: en_cond(n) -> t[n].en_flag = true;
     :: else -> t[n].en_flag = false;
```

```
      fi;
    :: else -> t[n].en_flag = false;
    fi;
  };

/*------------------------------------------------------------------*/

active proctype BTB() {

  ss.cs = initial;

    t[0].dest = orienting;
    t[0].in_chan = glob_ins[ss.inq.box_in];

    t[1].dest = connecting_f;

    t[2].dest = connecting_s;

    t[3].dest = deciding_1;
    t[3].in_chan = glob_ins[ss.inq.box_in];

    t[4].dest = transparent;
    t[4].in_chan = glob_ins[ss.inq.f_in];

    t[5].dest = initial;
    t[5].in_chan = glob_ins[ss.inq.s_in];

    t[6].dest = connecting_f;
    t[6].in_chan = glob_ins[ss.inq.s_in];

    t[7].dest = error;
    t[7].in_chan = glob_ins[ss.inq.s_in];

    t[8].dest = connecting_f;

    t[9].dest = connecting_f;

    t[10].dest = receiving;
    t[10].in_chan = glob_ins[ss.inq.box_in];

    t[11].dest = initial;
    t[11].in_chan = glob_ins[ss.inq.f_in];

    t[12].dest = initial;
    t[12].in_chan = glob_ins[ss.inq.s_in];
```

```

    t[13].dest = transparent;
    t[13].in_chan = glob_ins[ss.inq.f_in];

    t[14].dest = transparent;
    t[14].in_chan = glob_ins[ss.inq.s_in];

    t[15].dest = deciding_2;
    t[15].in_chan = glob_ins[ss.inq.box_in];

    t[16].dest = transparent;
    t[16].in_chan = glob_ins[ss.inq.s_in];

    t[17].dest = initial;
    t[17].in_chan = glob_ins[ss.inq.f_in];

    t[18].dest = connecting_s;
    t[18].in_chan = glob_ins[ss.inq.f_in];

    t[19].dest = error;
    t[19].in_chan = glob_ins[ss.inq.f_in];

    t[20].dest = connecting_s;

    t[21].dest = connecting_f;

    t[22].dest = connecting_f;

    t[23].dest = transparent;

end_initial_state:
 atomic {
      reset();
      en_trans(0);

      if
      ::t[0].en_flag -> next_trans(0); goto orienting_state;
      ::else -> goto end_initial_state;
      fi;
}

orienting_state:
 atomic {
      en_trans(1);
      en_trans(2);
```

```

      if
      ::t[1].en_flag -> next_trans(1); goto connecting_f_state;
      ::t[2].en_flag -> next_trans(2); goto connecting_s_state;
      ::else -> goto orienting_state;
     fi;
}

connecting_f_state:
 atomic {
      reset();
      en_trans(3);
      en_trans(4);
      en_trans(5);
      en_trans(6);
      en_trans(7);

      if
      ::t[3].en_flag -> next_trans(3); goto deciding_1_state;
      ::t[4].en_flag -> next_trans(4); goto transparent_state;
      ::t[5].en_flag -> next_trans(5); goto end_initial_state;
      ::t[6].en_flag -> next_trans(6); goto connecting_f_state;
      ::t[7].en_flag -> next_trans(7); goto error_state;
      ::else -> goto connecting_f_state;
      fi;
}

deciding_1_state:
 atomic {
      en_trans(8);
      en_trans(9);

      if
      ::t[8].en_flag -> next_trans(8); goto connecting_f_state;
      ::t[9].en_flag -> next_trans(9); goto connecting_f_state;
      ::else -> goto deciding_1_state;
      fi;
}

transparent_state:
 atomic {
      reset();
      en_trans(10);
      en_trans(11);
      en_trans(12);
```

```
        en_trans(13);
        en_trans(14);

        if
        ::t[10].en_flag -> next_trans(10); goto receiving_state;
        ::t[11].en_flag -> next_trans(11); goto end_initial_state;
        ::t[12].en_flag -> next_trans(12); goto end_initial_state;
        ::t[13].en_flag -> next_trans(13); goto transparent_state;
        ::t[14].en_flag -> next_trans(14); goto transparent_state;
        ::else -> goto transparent_state;
        fi;
}

connecting_s_state:
 atomic {
        reset();
        en_trans(15);
        en_trans(16);
        en_trans(17);
        en_trans(18);
        en_trans(19);

        if
        ::t[15].en_flag -> next_trans(15); goto deciding_2_state;
        ::t[16].en_flag -> next_trans(16); goto transparent_state;
        ::t[17].en_flag -> next_trans(17); goto end_initial_state;
        ::t[18].en_flag -> next_trans(18); goto connecting_s_state;
        ::t[19].en_flag -> next_trans(19); goto error_state;
        ::else -> goto connecting_s_state;
        fi;
}

deciding_2_state:
 atomic {       //assert(false);
        en_trans(20);
        en_trans(21);

        if
        ::t[20].en_flag -> next_trans(20); goto connecting_s_state;
        ::t[21].en_flag -> next_trans(21); goto connecting_f_state;
        ::else -> goto deciding_2_state;
        fi;
}

receiving_state:
```

```
 atomic {
     en_trans(22);
     en_trans(23);

     if
     ::t[22].en_flag -> next_trans(22); goto connecting_f_state;
     ::t[23].en_flag -> next_trans(23); goto transparent_state;
     ::else -> goto receiving_state;
     fi;
}

  error_state:
  skip;
};

active proctype pp_t() {

  ss.cs_post_process = idle;

   t[24].dest = t_work;

   t[29].dest = idle;
   t[29].in_chan = glob_ins[ss.inq.old_t_in];

end_idle_state:
 atomic {
   ss.intq.internal_t ? inter_sig;
     en_trans(24);

     if
     ::t[24].en_flag -> next_trans(24); goto t_work_state;
     ::else -> goto end_idle_state;
     fi;
}

t_work_state:
 atomic {
   reset_pp_t();
     en_trans(29);

     if
     ::t[29].en_flag -> next_trans(29); goto end_idle_state;
     ::else -> goto t_work_state;
     fi;
}
```

```

};

active proctype pp_s() {

    ss.cs_post_process = idle;

   t[25].dest = s_work;

    t[28].dest = s_wait_up;

    t[30].dest = idle;
    t[30].in_chan = glob_ins[ss.inq.old_s_in];

    t[31].dest = s_work;
    t[31].in_chan = glob_ins[ss.inq.old_s_in];

    t[35].dest = s_work;
    t[35].in_chan = glob_ins[ss.inq.old_s_in];

end_idle_state:
 atomic {
    ss.intq.internal_s ? inter_sig;
      en_trans(25);
      en_trans(28);

      if
      ::t[25].en_flag -> next_trans(25); goto s_work_state;
      ::t[28].en_flag -> next_trans(28); goto s_wait_up_state;
      ::else -> goto end_idle_state;
      fi;
}

s_wait_up_state:
 atomic {
    reset_pp_s();
      en_trans(35);

      if
      ::t[35].en_flag -> next_trans(35); goto s_work_state;
      ::else -> goto s_wait_up_state;
      fi;
}

s_work_state:
```

```
 atomic {
    reset_pp_s();
      en_trans(30);
      en_trans(31);

      if
      ::t[30].en_flag -> next_trans(30); goto end_idle_state;
      ::t[31].en_flag -> next_trans(31); goto s_work_state;
      ::else -> goto s_work_state;
      fi;
}

};


active proctype pp_f() {

    ss.cs_post_process = idle;

  t[26].dest = f_wait_up;

    t[27].dest = f_work;

    t[32].dest = f_work;

    t[32].in_chan = glob_ins[ss.inq.old_f_in];

    t[33].dest = idle;
    t[33].in_chan = glob_ins[ss.inq.old_f_in];

    t[34].dest = f_work;
    t[34].in_chan = glob_ins[ss.inq.old_f_in];

end_idle_state:
 atomic {
    ss.intq.internal_f ? inter_sig;
      en_trans(26);
      en_trans(27);

      if
      ::t[26].en_flag -> next_trans(26); goto f_wait_up_state;
      ::t[27].en_flag -> next_trans(27); goto f_work_state;
      ::else -> goto end_idle_state;
      fi;
}
```

```
f_wait_up_state:
 atomic {
   reset_pp_f();
     en_trans(32);

     if
     ::t[32].en_flag -> next_trans(32); goto f_work_state;
     ::else -> goto f_wait_up_state;
     fi;
}

f_work_state:
 atomic {
   reset_pp_f();
     en_trans(33);
     en_trans(34);

     if
     ::t[33].en_flag -> next_trans(33); goto end_idle_state;
     ::t[34].en_flag -> next_trans(34); goto f_work_state;
     ::else -> goto f_work_state;
     fi;
}

};

active proctype env() {
mtype f_sigu , s_sigu;
 end:
    do

   :: ss.inq.box_in_ready  && !ss.inq.old_t_in_ready
    && !ss.inq.old_s_in_ready && !ss.inq.old_f_in_ready ->
     if
     ::atomic{
      t_from_subs = true;
       counter = counter + 1;
       glob_ins[ss.inq.box_in] ! setup;
     }
     ::atomic{
       t_from_subs = false;
       counter = counter + 1;
       glob_ins[ss.inq.box_in] ! setup;
     }
```

```
      fi;
      :: ss.inq.s_in_ready && !ss.inq.old_t_in_ready ->
        if
        :: glob_ins[ss.inq.s_in] ! other;
        :: glob_ins[ss.inq.s_in] ! teardown;
        fi unless {
          if
          :: (s_sigu == upack) ->
           if
           :: (current_t_from_subs) -> glob_ins[ss.inq.f_in] ! upack;
             s_sigu = 0;
           :: else -> glob_ins[ss.inq.s_in] ! upack;
             s_sigu = 0;
           fi;
          fi;
        }
      :: ss.inq.old_s_in_ready && !ss.inq.old_t_in_ready ->
        if
        :: glob_ins[ss.inq.old_s_in] ! downack;
        :: glob_ins[ss.inq.old_s_in] ! upack;
        fi;
      :: ss.inq.f_in_ready && !ss.inq.old_t_in_ready ->
       if
        :: glob_ins[ss.inq.f_in] ! other;
        :: glob_ins[ss.inq.f_in] ! teardown;
        fi unless {
         if
         :: (f_sigu == upack) ->
           if
           :: (current_t_from_subs) -> glob_ins[ss.inq.f_in] ! upack;
             f_sigu = 0;
           :: else -> glob_ins[ss.inq.s_in] ! upack;
             f_sigu = 0;
           fi;
          fi;
        }
      :: ss.inq.old_f_in_ready && !ss.inq.old_t_in_ready ->
        if
        :: glob_ins[ss.inq.old_f_in] ! downack;
        :: glob_ins[ss.inq.old_f_in] ! upack;
        fi;
      :: ss.inq.old_t_in_ready -> glob_ins[ss.inq.old_t_in] ! downack;
od
 unless {
      if
```

```
        ::atomic{
          glob_outs[ss.out.f_out] ? setup -> f_sigu = upack;}
        ::atomic{
          glob_outs[ss.out.s_out] ? setup -> s_sigu = upack;}
        ::glob_outs[ss.out.t_out] ? upack;
        ::glob_outs[ss.out.t_out] ? unavail;
        ::atomic {
           glob_outs[ss.out.t_out] ? teardown -> teardown_cleanup(0);}
        ::atomic{glob_outs[ss.out.s_out] ? downack -> ss.inq.s_in_ready = false;}
        ::atomic {
           glob_outs[ss.out.s_out] ? teardown -> teardown_cleanup(1);}
        ::glob_outs[ss.out.s_out] ? other;
        ::atomic {
         glob_outs[ss.out.f_out] ? teardown -> teardown_cleanup(2);}
        ::atomic{ glob_outs[ss.out.f_out] ? downack -> ss.inq.f_in_ready = false;}
        ::glob_outs[ss.out.f_out] ? other;
        ::glob_outs[ss.out.old_s_out] ? downack;
        ::glob_outs[ss.out.old_f_out] ? downack;
        fi;
        }
        goto end;
 }
 ltl p0 {!(BTB@error_state) && ([]((rcv_setup && (current_call == 5))-> <> (send_upack &&(current_call == 5)))))}
 never{(BTB@orienting_state && (!(pp_f@end_idle_state) && !(pp_s@end_idle_state) && !(pp_t@end_idle_state))) && !(pp_call == last_call)}
```

Appendix E

Promela model - Error Interface

```
/*----------------------------------------------------------*/
/*   ErrorInterface                              */
/*----------------------------------------------------------*/

/*  type definitions  */

mtype = { teardown , downack , other , setup , upack  , unknown };
mtype = { initial , terminating_c , final };

 typedef Transition {
     mtype dest;
     chan in_chan;
     bool en_flag = false;
  };

 typedef in_q {
   byte box_in = 0;
   byte c_in = 1;
   bool box_in_ready = true;
   bool c_in_ready = false;
   byte selected
 };
 chan glob_ins[2] = [0] of {mtype};

 typedef out_q {
   byte box_out = 0;
   byte c_out = 1;
 };
 chan glob_outs[2] = [0] of {mtype};
```

```
typedef SnapShot {
  mtype cs;
  in_q inq;
  out_q out
 };

/*-----------------------------------------------------*/
/*  global variable declarations  */
/*-----------------------------------------------------*/

SnapShot ss;
Transition t[3];
mtype sig;

/*  Global Monitor Variables  */
bool rcv_setup = false;
bool send_upack = false;
bool c_send_unknown = false;
bool c_send_teardown = false;
bool c_rcv_downack = false;
bool c_rcv_teardown = false;
bool c_send_downack = false;

/*------------------------------------------------------*/
/*  Inline Functions  */

inline dump(c1 , c2) {
    byte aSig;
    do
    ::c1 ? aSig -> c2 ! aSig;
    ::empty(c1) -> break;
    od
 };

inline reset() {
  rcv_setup = false;
  send_upack = false;
  c_send_unknown = false;
  c_send_teardown = false;
  c_rcv_downack = false;
  c_rcv_teardown = false;
  c_send_downack = false;

  if
```

```
  ::glob_ins[ss.inq.box_in] ? sig -> ss.inq.selected = ss.inq.box_in;
  ::glob_ins[ss.inq.c_in] ? sig -> ss.inq.selected = ss.inq.c_in;
  fi
};

inline en_events(n) {
    glob_ins[ss.inq.selected] == t[n].in_chan;
};


inline en_cond(n) {
  if
  ::(n == 0) && (sig == setup );

  ::(n == 1) && (sig == downack );
  ::(n == 2) && (sig == teardown );
  fi;
};

inline next_trans(n) {
if
  ::(n == 0) ->        rcv_setup = true;
          ss.inq.c_in_ready = true;
         glob_outs[ss.out.c_out] ! upack;
          send_upack = true;
         glob_outs[ss.out.c_out] ! unknown;
          c_send_unknown = true;
         glob_outs[ss.out.c_out] ! teardown;
          c_send_teardown = true;
          ss.cs = t[0].dest;

  ::(n == 1) ->
          c_rcv_downack = true;
          ss.inq.c_in_ready = false;
          ss.cs = t[1].dest;

  ::(n == 2) ->
          c_rcv_teardown = true;
          c_send_downack = true;
         glob_outs[ss.out.c_out] ! downack;
          ss.cs = t[2].dest;

fi;
};

```

```
 inline en_trans(n) {
   if
   ::en_events(n) ->
     if
     ::en_cond(n) -> t[n].en_flag = true;
     ::else -> t[n].en_flag = false;
     fi;
   ::else -> t[n].en_flag = false;
   fi;
 };

active proctype ErrorInterface() {

  ss.cs = initial;
   t[0].dest = terminating_c;
   t[0].in_chan = glob_ins[ss.inq.box_in];
   t[1].dest = final;
   t[1].in_chan = glob_ins[ss.inq.c_in];
   t[2].dest = terminating_c;
   t[2].in_chan = glob_ins[ss.inq.c_in];


initial_state:
 atomic {
     reset();
     en_trans(0);

     if
     ::t[0].en_flag -> next_trans(0); goto terminating_c_state;
     ::else -> goto initial_state;

     fi;
}


terminating_c_state:
 atomic {
     reset();
     en_trans(1);
     en_trans(2);

     if
     ::t[1].en_flag -> next_trans(1); goto final_state;
     ::t[2].en_flag -> next_trans(2); goto terminating_c_state;
     ::else -> goto terminating_c_state;
```

```

      fi;
}

      error_state:
final_state:
      progress:

      skip;
};

active proctype env() {
 mtype c_sigt ;



end:   do

      :: ss.inq.box_in_ready ->
           ss.inq.box_in_ready = false;
           glob_ins[ss.inq.box_in] ! setup;

      :: ss.inq.c_in_ready ->
         if
         :: glob_ins[ss.inq.c_in] ! teardown;
         :: glob_ins[ss.inq.c_in] ! other;
         fi
   od
   unless {
         if
         :: glob_outs[ss.out.c_out] ? upack;
         :: glob_outs[ss.out.c_out] ? unknown;
         :: atomic { glob_outs[ss.out.c_out] ? teardown ->
         glob_ins[ss.inq.c_in] ! downack;
         }
         :: glob_outs[ss.out.c_out] ? downack;
         :: glob_outs[ss.out.c_out] ? other;
         fi;
   }
   goto end;
 }
ltl p0 {[](rcv_setup -> <>send_upack)}
ltl p1 {[](c_rcv_teardown -> <>c_send_downack)}
ltl p2 {[](rcv_setup -> ((!c_send_unknown) U send_upack))}
ltl p3 {[]((c_rcv_teardown || c_send_teardown) -> [](!c_send_unknown))}
```

Appendix F

Promela Model - Receive Voice Mail

```
/*---------------------------------------------------*/
/*   ReceiveVoiceMail                         */
/*---------------------------------------------------*/

/*  type definitions  */

mtype = { teardown , downack , other , setup , upack  , unavail , avail , dummy };
mtype = { initial , connecting_o , transparent , switching , waitingodown ,
    connecting_r , dialogue , abandonConnectiono , terminating_o , final ,
      terminating_i , abandoning_r_o , ending_o_r , waitingrup , terminating_r ,
      error };

 typedef Transition {
     mtype dest;
     chan in_chan;
     bool en_flag = false;
  };

 typedef in_q {
    byte box_in = 0;
    byte i_in = 1;
    byte r_in = 2;
    byte o_in = 3;
    bool box_in_ready = true;
    bool i_in_ready = false;
    bool r_in_ready = false;
```

```
  bool o_in_ready = false;
  byte selected
};
chan glob_ins[4] = [0] of {mtype};

typedef out_q {
  byte box_out = 0;
  byte i_out = 1;
  byte r_out = 2;
  byte o_out = 3;
  chan o_hold = [5] of {mtype};
  chan r_hold = [5] of {mtype};
};
chan glob_outs[4] = [0] of {mtype};

typedef SnapShot {
  mtype cs;
  in_q inq;
  out_q out
 };

/*------------------------------------------------------------*/
/*   global variable declarations   */
/*------------------------------------------------------------*/

mtype sig;
SnapShot ss;
Transition t[43];

/*   Global Monitor Variables   */
bool rcv_setup = false;
bool send_upack = false;
bool o_send_setup = false;
bool o_rcv_upack = false;
bool i_rcv_teardown = false;
bool i_send_downack = false;
bool o_send_teardown = false;
bool o_rcv_unavail = false;
bool i_send_avail = false;
bool r_send_setup = false;
bool o_rcv_teardown = false;
bool o_send_downack = false;
bool i_send_teardown = false;
bool r_rcv_upack = false;
bool o_rcv_downack = false;
```

```
bool r_send_teardown = false;
bool r_rcv_dummy = false;
bool i_rcv_downack = false;
bool r_rcv_downack = false;

/*----------------------------------------------------------------*/
/*  Inline Functions  */

inline dump(c1 , c2) {
    byte aSig;
    do
    ::c1 ? aSig -> c2 ! aSig;
    ::empty(c1) -> break;
    od
 };

inline reset() {
  rcv_setup = false;
  send_upack = false;
  o_send_setup = false;
  o_rcv_upack = false;
  i_rcv_teardown = false;
  i_send_downack = false;
  o_send_teardown = false;
  o_rcv_unavail = false;
  i_send_avail = false;
  r_send_setup = false;
  o_rcv_teardown = false;
  o_send_downack = false;
  i_send_teardown = false;
  r_rcv_upack = false;
  o_rcv_downack = false;
  r_send_teardown = false;
  r_rcv_dummy = false;
  i_rcv_downack = false;
  r_rcv_downack = false;

  if
  ::glob_ins[ss.inq.box_in] ? sig -> ss.inq.selected = ss.inq.box_in;
  ::glob_ins[ss.inq.i_in] ? sig -> ss.inq.selected = ss.inq.i_in;
  ::glob_ins[ss.inq.r_in] ? sig -> ss.inq.selected = ss.inq.r_in;
  ::glob_ins[ss.inq.o_in] ? sig -> ss.inq.selected = ss.inq.o_in;
  fi
};
```

```
inline en_events(n) {
    glob_ins[ss.inq.selected] == t[n].in_chan;
};

inline en_cond(n) {
if
  ::(n == 0) && (sig == setup );

  ::(n == 1) && (sig == upack );
  ::(n == 2) && (sig == teardown );
  ::(n == 3) && ( sig != teardown && nfull(ss.out.o_hold) );
  ::(n == 4) && ( sig != teardown && full(ss.out.o_hold) );

  ::(n == 5) && (sig == unavail );
  ::(n == 6) && (sig == teardown );
  ::(n == 7) && (sig == teardown );
  ::(n == 8) && ( sig != teardown  );
  ::(n == 9) && ( sig != teardown  );

  ::(n == 10) && (sig == upack );
  ::(n == 11) && (sig == downack );
  ::(n == 12) && (sig == teardown );
  ::(n == 13) && ( sig != teardown && nfull(ss.out.r_hold) );
  ::(n == 14) && (sig == teardown );
  ::(n == 15) && ( sig != teardown && full(ss.out.r_hold) );

  ::(n == 16) && (sig == downack );
  ::(n == 17) && (sig == teardown );
  ::(n == 18) && ( sig != teardown  );
  ::(n == 19) && (sig == teardown );
  ::(n == 20) && (sig == dummy );

  ::(n == 21) && (sig == upack );
  ::(n == 22) && (sig == teardown );
  ::(n == 23) && ( sig != teardown  );
  ::(n == 24) && ( sig != teardown && full(ss.out.r_hold) );

  ::(n == 25) && (sig == teardown );
  ::(n == 26) && ( sig != teardown  );
  ::(n == 27) && (sig == dummy );

  ::(n == 28) && (sig == upack );

  ::(n == 29) && (sig == downack );
  ::(n == 30) && (sig == teardown );
```

```
  ::(n == 31) && (sig == downack );
  ::(n == 32) && (sig == teardown );

  ::(n == 33) && (sig == upack );
  ::(n == 34) && (sig == downack );
  ::(n == 35) && (sig == teardown );

  ::(n == 36) && (sig == downack );
  ::(n == 37) && (sig == downack );
  ::(n == 38) && (sig == teardown );
  ::(n == 39) && (sig == dummy );

  ::(n == 40) && (sig == upack );

  ::(n == 41) && (sig == downack );
  ::(n == 42) && (sig == dummy );
fi;
};

inline next_trans(n) {
if
  ::(n == 0) ->        rcv_setup = true;
          ss.inq.i_in_ready = true;
         glob_outs[ss.out.i_out] ! upack;
          send_upack = true;
         glob_outs[ss.out.o_out] ! setup;
          ss.inq.o_in_ready = true;
          o_send_setup = true;
          ss.cs = t[0].dest;

  ::(n == 1) ->
          o_rcv_upack = true;
          dump(ss.out.o_hold , glob_outs[ss.out.o_out]);
          ss.cs = t[1].dest;

  ::(n == 2) ->
          i_rcv_teardown = true;
          i_send_downack = true;
         glob_outs[ss.out.i_out] ! downack;
          ss.inq.i_in_ready = false;
          o_send_teardown = true;
         glob_outs[ss.out.o_out] ! teardown;
          ss.cs = t[2].dest;

```

```
:: (n == 3) ->
        ss.out.o_hold ! sig;
        ss.cs = t[3].dest;

:: (n == 4) ->
        ss.inq.i_in_ready = false;
        ss.inq.r_in_ready = false;
        ss.inq.o_in_ready = false;
        ss.cs = t[4].dest;

:: (n == 5) ->
        o_rcv_unavail = true;
        i_send_avail = true;
        glob_outs[ss.out.i_out] ! avail;
        o_send_teardown = true;
        glob_outs[ss.out.o_out] ! teardown;
        r_send_setup = true;
        glob_outs[ss.out.r_out] ! setup;
        ss.inq.r_in_ready = true;
        ss.cs = t[5].dest;

:: (n == 6) ->
        i_rcv_teardown = true;
        i_send_downack = true;
        glob_outs[ss.out.i_out] ! downack;
        ss.inq.i_in_ready = false;
        o_send_teardown = true;
        glob_outs[ss.out.o_out] ! teardown;
        ss.cs = t[6].dest;

:: (n == 7) ->
        o_rcv_teardown = true;
        o_send_downack = true;
        glob_outs[ss.out.o_out] ! downack;
        ss.inq.o_in_ready = false;
        i_send_teardown = true;
        glob_outs[ss.out.i_out] ! teardown;
        ss.cs = t[7].dest;

:: (n == 8) ->
        glob_outs[ss.out.o_out] ! sig;
        ss.cs = t[8].dest;

:: (n == 9) ->
        glob_outs[ss.out.i_out] ! sig;
```

```
        ss.cs = t[9].dest;

:: (n == 10) ->
        r_rcv_upack = true;
        dump(ss.out.r_hold , glob_outs[ss.out.r_out]);
        ss.cs = t[10].dest;

:: (n == 11) ->
        o_rcv_downack = true;
        ss.inq.o_in_ready = false;
        ss.cs = t[11].dest;

:: (n == 12) ->
        i_rcv_teardown = true;
        i_send_downack = true;
       glob_outs[ss.out.i_out] ! downack;
        ss.inq.i_in_ready = false;
        r_send_teardown = true;
       glob_outs[ss.out.r_out] ! teardown;
        ss.cs = t[12].dest;

:: (n == 13) ->
       ss.out.r_hold ! sig;
        ss.cs = t[13].dest;

:: (n == 14) ->
        o_rcv_teardown = true;
        o_send_downack = true;
       glob_outs[ss.out.o_out] ! downack;
        ss.cs = t[14].dest;

:: (n == 15) ->
        ss.inq.i_in_ready = false;
        ss.inq.r_in_ready = false;
        ss.inq.o_in_ready = false;
        ss.cs = t[15].dest;

:: (n == 16) ->
        o_rcv_downack = true;
        ss.inq.o_in_ready = false;
        ss.cs = t[16].dest;

:: (n == 17) ->
        i_rcv_teardown = true;
        i_send_downack = true;
```

```
        glob_outs[ss.out.i_out] ! downack;
         ss.inq.i_in_ready = false;
         r_send_teardown = true;
        glob_outs[ss.out.r_out] ! teardown;
         ss.cs = t[17].dest;

::(n == 18) ->
         ss.cs = t[18].dest;

::(n == 19) ->
         o_rcv_teardown = true;
         o_send_downack = true;
        glob_outs[ss.out.o_out] ! downack;
         ss.cs = t[19].dest;

::(n == 20) ->
         r_rcv_dummy = true;
         ss.cs = t[20].dest;

::(n == 21) ->
         r_rcv_upack = true;
         dump(ss.out.r_hold , glob_outs[ss.out.r_out]);
         ss.cs = t[21].dest;

::(n == 22) ->
         i_rcv_teardown = true;
         i_send_downack = true;
        glob_outs[ss.out.i_out] ! downack;
         ss.inq.i_in_ready = false;
         r_send_teardown = true;
        glob_outs[ss.out.r_out] ! teardown;
         ss.cs = t[22].dest;

::(n == 23) ->
         ss.cs = t[23].dest;

::(n == 24) ->
         ss.inq.i_in_ready = false;
         ss.inq.r_in_ready = false;
         ss.inq.o_in_ready = false;
         ss.cs = t[24].dest;

::(n == 25) ->
         i_rcv_teardown = true;
         i_send_downack = true;
```

```
            glob_outs[ss.out.i_out] ! downack;
            ss.inq.i_in_ready = false;
            r_send_teardown = true;
            glob_outs[ss.out.r_out] ! teardown;
            ss.cs = t[25].dest;

    ::(n == 26) ->
            glob_outs[ss.out.r_out] ! sig;
            ss.cs = t[26].dest;

    ::(n == 27) ->
            r_rcv_dummy = true;
            ss.cs = t[27].dest;

    ::(n == 28) ->
            o_rcv_upack = true;
            dump(ss.out.o_hold , glob_outs[ss.out.o_out]);
            ss.cs = t[28].dest;

    ::(n == 29) ->
            o_rcv_downack = true;
            ss.inq.o_in_ready = false;
            ss.cs = t[29].dest;

    ::(n == 30) ->
            o_rcv_teardown = true;
            o_send_downack = true;
            glob_outs[ss.out.o_out] ! downack;
            ss.cs = t[30].dest;

    ::(n == 31) ->
            i_rcv_downack = true;
            ss.inq.i_in_ready = false;
            ss.cs = t[31].dest;

    ::(n == 32) ->
            i_rcv_teardown = true;
            i_send_downack = true;
            glob_outs[ss.out.i_out] ! downack;
            ss.cs = t[32].dest;

    ::(n == 33) ->
            r_rcv_upack = true;
            dump(ss.out.r_hold , glob_outs[ss.out.r_out]);
            ss.cs = t[33].dest;
```

```
    ::(n == 34) ->
            o_rcv_downack = true;
            ss.inq.o_in_ready = false;
            ss.cs = t[34].dest;

    ::(n == 35) ->
            o_rcv_teardown = true;
            o_send_downack = true;
           glob_outs[ss.out.o_out] ! downack;
            ss.cs = t[35].dest;

    ::(n == 36) ->
            o_rcv_downack = true;
            ss.inq.o_in_ready = false;
            ss.cs = t[36].dest;

    ::(n == 37) ->
            r_rcv_downack = true;
            ss.inq.r_in_ready = false;
            ss.cs = t[37].dest;

    ::(n == 38) ->
            o_rcv_teardown = true;
            o_send_downack = true;
           glob_outs[ss.out.o_out] ! downack;
            ss.cs = t[38].dest;

    ::(n == 39) ->
            r_rcv_dummy = true;
            ss.cs = t[39].dest;

    ::(n == 40) ->
            r_rcv_upack = true;
            ss.cs = t[40].dest;

    ::(n == 41) ->
            r_rcv_downack = true;
            ss.inq.r_in_ready = false;
            ss.cs = t[41].dest;

    ::(n == 42) ->
            r_rcv_dummy = true;
            ss.cs = t[42].dest;
```

```
fi;
};

 inline en_trans(n) {
    if
    ::en_events(n) ->
      if
      ::en_cond(n) -> t[n].en_flag = true;
      ::else -> t[n].en_flag = false;
      fi;
    ::else -> t[n].en_flag = false;
    fi;
 };

active proctype ReceiveVoiceMail() {

   ss.cs = initial;
    t[0].dest = connecting_o;
    t[0].in_chan = glob_ins[ss.inq.box_in];
    t[1].dest = transparent;
    t[1].in_chan = glob_ins[ss.inq.o_in];
    t[2].dest = abandonConnectiono;
    t[2].in_chan = glob_ins[ss.inq.i_in];
    t[3].dest = connecting_o;
    t[3].in_chan = glob_ins[ss.inq.i_in];
    t[4].dest = error;
    t[4].in_chan = glob_ins[ss.inq.i_in];
    t[5].dest = switching;
    t[5].in_chan = glob_ins[ss.inq.o_in];
    t[6].dest = terminating_o;
    t[6].in_chan = glob_ins[ss.inq.i_in];
    t[7].dest = terminating_i;
    t[7].in_chan = glob_ins[ss.inq.o_in];
    t[8].dest = transparent;
    t[8].in_chan = glob_ins[ss.inq.i_in];
    t[9].dest = transparent;
    t[9].in_chan = glob_ins[ss.inq.o_in];
    t[10].dest = waitingodown;
    t[10].in_chan = glob_ins[ss.inq.r_in];
    t[11].dest = connecting_r;
    t[11].in_chan = glob_ins[ss.inq.o_in];
    t[12].dest = abandoning_r_o;
    t[12].in_chan = glob_ins[ss.inq.i_in];
    t[13].dest = switching;
    t[13].in_chan = glob_ins[ss.inq.i_in];
```

```
t[14].dest = switching;
t[14].in_chan = glob_ins[ss.inq.o_in];
t[15].dest = error;
t[15].in_chan = glob_ins[ss.inq.i_in];
t[16].dest = dialogue;
t[16].in_chan = glob_ins[ss.inq.o_in];
t[17].dest = ending_o_r;
t[17].in_chan = glob_ins[ss.inq.i_in];
t[18].dest = waitingodown;
t[18].in_chan = glob_ins[ss.inq.i_in];
t[19].dest = waitingodown;
t[19].in_chan = glob_ins[ss.inq.o_in];
t[20].dest = waitingodown;
t[20].in_chan = glob_ins[ss.inq.r_in];
t[21].dest = dialogue;
t[21].in_chan = glob_ins[ss.inq.r_in];
t[22].dest = waitingrup;
t[22].in_chan = glob_ins[ss.inq.i_in];
t[23].dest = connecting_r;
t[23].in_chan = glob_ins[ss.inq.i_in];
t[24].dest = error;
t[24].in_chan = glob_ins[ss.inq.i_in];
t[25].dest = terminating_r;
t[25].in_chan = glob_ins[ss.inq.i_in];
t[26].dest = dialogue;
t[26].in_chan = glob_ins[ss.inq.i_in];
t[27].dest = dialogue;
t[27].in_chan = glob_ins[ss.inq.r_in];
t[28].dest = terminating_o;
t[28].in_chan = glob_ins[ss.inq.o_in];
t[29].dest = final;
t[29].in_chan = glob_ins[ss.inq.o_in];
t[30].dest = terminating_o;
t[30].in_chan = glob_ins[ss.inq.o_in];
t[31].dest = final;
t[31].in_chan = glob_ins[ss.inq.i_in];
t[32].dest = terminating_i;
t[32].in_chan = glob_ins[ss.inq.i_in];
t[33].dest = ending_o_r;
t[33].in_chan = glob_ins[ss.inq.r_in];
t[34].dest = waitingrup;
t[34].in_chan = glob_ins[ss.inq.o_in];
t[35].dest = abandoning_r_o;
t[35].in_chan = glob_ins[ss.inq.o_in];
t[36].dest = terminating_r;
```

```
    t[36].in_chan = glob_ins[ss.inq.o_in];
    t[37].dest = terminating_o;
    t[37].in_chan = glob_ins[ss.inq.r_in];
    t[38].dest = ending_o_r;
    t[38].in_chan = glob_ins[ss.inq.o_in];
    t[39].dest = ending_o_r;
    t[39].in_chan = glob_ins[ss.inq.r_in];
    t[40].dest = terminating_r;
    t[40].in_chan = glob_ins[ss.inq.r_in];
    t[41].dest = final;
    t[41].in_chan = glob_ins[ss.inq.r_in];
    t[42].dest = terminating_r;
    t[42].in_chan = glob_ins[ss.inq.r_in];


initial_state:
 atomic {
      reset();
      en_trans(0);

      if
      ::t[0].en_flag -> next_trans(0); goto connecting_o_state;
      ::else -> goto initial_state;

      fi;
}


connecting_o_state:
 atomic {
      reset();
      en_trans(1);
      en_trans(2);
      en_trans(3);
      en_trans(4);

      if
      ::t[1].en_flag -> next_trans(1); goto transparent_state;
      ::t[2].en_flag -> next_trans(2); goto abandonConnectiono_state;
      ::t[3].en_flag -> next_trans(3); goto connecting_o_state;
      ::t[4].en_flag -> next_trans(4); goto error_state;
      ::else -> goto connecting_o_state;

      fi;
}
```

```
transparent_state:
 atomic {
      reset();
      en_trans(5);
      en_trans(6);
      en_trans(7);
      en_trans(8);
      en_trans(9);

      if
      ::t[5].en_flag -> next_trans(5); goto switching_state;
      ::t[6].en_flag -> next_trans(6); goto terminating_o_state;
      ::t[7].en_flag -> next_trans(7); goto terminating_i_state;
      ::t[8].en_flag -> next_trans(8); goto transparent_state;
      ::t[9].en_flag -> next_trans(9); goto transparent_state;
      ::else -> goto transparent_state;

      fi;
}


switching_state:
 atomic {
      reset();
      en_trans(10);
      en_trans(11);
      en_trans(12);
      en_trans(13);
      en_trans(14);
      en_trans(15);

      if
      ::t[10].en_flag -> next_trans(10); goto waitingodown_state;
      ::t[11].en_flag -> next_trans(11); goto connecting_r_state;
      ::t[12].en_flag -> next_trans(12); goto abandoning_r_o_state;
      ::t[13].en_flag -> next_trans(13); goto switching_state;
      ::t[14].en_flag -> next_trans(14); goto switching_state;
      ::t[15].en_flag -> next_trans(15); goto error_state;
      ::else -> goto switching_state;

      fi;
}
```

```
waitingodown_state:
 atomic {
      reset();
      en_trans(16);
      en_trans(17);
      en_trans(18);
      en_trans(19);
      en_trans(20);

      if
      ::t[16].en_flag -> next_trans(16); goto dialogue_state;
      ::t[17].en_flag -> next_trans(17); goto ending_o_r_state;
      ::t[18].en_flag -> next_trans(18); goto waitingodown_state;
      ::t[19].en_flag -> next_trans(19); goto waitingodown_state;
      ::t[20].en_flag -> next_trans(20); goto waitingodown_state;
      ::else -> goto waitingodown_state;

      fi;
}


connecting_r_state:
 atomic {
      reset();
      en_trans(21);
      en_trans(22);
      en_trans(23);
      en_trans(24);

      if
      ::t[21].en_flag -> next_trans(21); goto dialogue_state;
      ::t[22].en_flag -> next_trans(22); goto waitingrup_state;
      ::t[23].en_flag -> next_trans(23); goto connecting_r_state;
      ::t[24].en_flag -> next_trans(24); goto error_state;
      ::else -> goto connecting_r_state;

      fi;
}


dialogue_state:
 atomic {
      reset();
      en_trans(25);
```

```
    en_trans(26);
    en_trans(27);

    if
    ::t[25].en_flag -> next_trans(25); goto terminating_r_state;
    ::t[26].en_flag -> next_trans(26); goto dialogue_state;
    ::t[27].en_flag -> next_trans(27); goto dialogue_state;
    ::else -> goto dialogue_state;

    fi;
}


abandonConnectiono_state:
 atomic {
    reset();
    en_trans(28);

    if
    ::t[28].en_flag -> next_trans(28); goto terminating_o_state;
    ::else -> goto abandonConnectiono_state;

    fi;
}


terminating_o_state:
 atomic {
    reset();
    en_trans(29);
    en_trans(30);

    if
    ::t[29].en_flag -> next_trans(29); goto final_state;
    ::t[30].en_flag -> next_trans(30); goto terminating_o_state;
    ::else -> goto terminating_o_state;

    fi;
}


terminating_i_state:
 atomic {
    reset();
    en_trans(31);
```

```
      en_trans(32);

      if
      ::t[31].en_flag -> next_trans(31); goto final_state;
      ::t[32].en_flag -> next_trans(32); goto terminating_i_state;
      ::else -> goto terminating_i_state;

      fi;
}


abandoning_r_o_state:
 atomic {
      reset();
      en_trans(33);
      en_trans(34);
      en_trans(35);

      if
      ::t[33].en_flag -> next_trans(33); goto ending_o_r_state;
      ::t[34].en_flag -> next_trans(34); goto waitingrup_state;
      ::t[35].en_flag -> next_trans(35); goto abandoning_r_o_state;
      ::else -> goto abandoning_r_o_state;

      fi;
}


ending_o_r_state:
 atomic {
      reset();
      en_trans(36);
      en_trans(37);
      en_trans(38);
      en_trans(39);

      if
      ::t[36].en_flag -> next_trans(36); goto terminating_r_state;
      ::t[37].en_flag -> next_trans(37); goto terminating_o_state;
      ::t[38].en_flag -> next_trans(38); goto ending_o_r_state;
      ::t[39].en_flag -> next_trans(39); goto ending_o_r_state;
      ::else -> goto ending_o_r_state;

      fi;
}
```

```
waitingrup_state:
 atomic {
      reset();
      en_trans(40);

      if
      ::t[40].en_flag -> next_trans(40); goto terminating_r_state;
      ::else -> goto waitingrup_state;

      fi;
}


terminating_r_state:
 atomic {
      reset();
      en_trans(41);
      en_trans(42);

      if
      ::t[41].en_flag -> next_trans(41); goto final_state;
      ::t[42].en_flag -> next_trans(42); goto terminating_r_state;
      ::else -> goto terminating_r_state;

      fi;
}

      error_state:
final_state:
      progress:

      skip;
};

active proctype env() {
 mtype i_sigt ,r_sigt  , r_sigu ,o_sigt  , o_sigu ;



end:  do

      ::  ss.inq.box_in_ready ->
           ss.inq.box_in_ready = false;
```

```
        glob_ins[ss.inq.box_in] ! setup;

    ::ss.inq.i_in_ready ->
      if
      :: glob_ins[ss.inq.i_in] ! teardown;
      :: glob_ins[ss.inq.i_in] ! other;
      fi unless {
         (i_sigt == teardown) ->
             glob_ins[ss.inq.i_in] ! downack;
             i_sigt = 0;
      }
    ::ss.inq.r_in_ready ->
      if
      ::glob_ins[ss.inq.r_in] ! dummy;
      fi unless {
          if
          ::(r_sigu == upack) ->
            glob_ins[ss.inq.r_in] ! upack;
           r_sigu = 0;
          ::(r_sigt == teardown && r_sigu == 0) ->
            glob_ins[ss.inq.r_in] ! downack;
           r_sigt = 0;
          fi;
          }
    ::ss.inq.o_in_ready ->
      if
      :: glob_ins[ss.inq.o_in] ! teardown;
      :: glob_ins[ss.inq.o_in] ! other;
      fi unless {
          if
          ::(o_sigu == upack) ->
            glob_ins[ss.inq.o_in] ! upack;
           glob_ins[ss.inq.o_in] ! avail;
           o_sigu = 0;
          ::(o_sigt == teardown && o_sigu == 0) ->
            glob_ins[ss.inq.o_in] ! downack;
           o_sigt = 0;
          fi;
          }
  od
  unless {
      if
      ::atomic { glob_outs[ss.out.r_out] ? setup ->
       r_sigu = upack;
       }
```

```
          ::atomic { glob_outs[ss.out.o_out] ? setup ->
           o_sigu = upack;
           }
          ::glob_outs[ss.out.i_out] ? upack;
          ::glob_outs[ss.out.i_out] ? downack;
          ::glob_outs[ss.out.i_out] ? avail;
          ::atomic { glob_outs[ss.out.i_out] ? teardown ->
           i_sigt = teardown;
           }
          ::glob_outs[ss.out.i_out] ? other;
          ::atomic { glob_outs[ss.out.r_out] ? teardown ->
           r_sigt = teardown;
           }
          ::glob_outs[ss.out.r_out] ? other;
          ::atomic { glob_outs[ss.out.o_out] ? teardown ->
           o_sigt = teardown;
           }
          ::glob_outs[ss.out.o_out] ? downack;
          ::glob_outs[ss.out.o_out] ? other;
          fi;
     }
     goto end;
  }
 ltl p0 {(!ReceiveVoiceMail@error_state) && [](rcv_setup -> <> send_upack)}
 ltl p1 {(!ReceiveVoiceMail@error_state) && [](i_rcv_teardown -> <> i_send_downack)}
 ltl p2 {(!ReceiveVoiceMail@error_state) && [](i_send_teardown -> <> i_rcv_downack)}
 ltl p3 {(!ReceiveVoiceMail@error_state) && [](o_send_setup -> <> o_rcv_upack)}
 ltl p4 {(!ReceiveVoiceMail@error_state) && [](o_rcv_teardown -> <> o_send_downack)}
 ltl p5 {(!ReceiveVoiceMail@error_state) && [](o_send_teardown -> <> o_rcv_downack)}
 ltl p6 {(!ReceiveVoiceMail@error_state) && []((i_rcv_teardown || i_send_teardown) -> [](!i_send_avail))}
 ltl p7 {(!ReceiveVoiceMail@error_state) && []((rcv_setup) -> (!i_send_avail U send_upack))}
 ltl p8 {(!ReceiveVoiceMail@error_state) && [](r_send_setup -> <> r_rcv_upack)}
 ltl p9 {(!ReceiveVoiceMail@error_state) && [](r_send_teardown -> <> r_rcv_downack)}
```

Appendix G

Promela Model - Black Phone Interface

```
/*--------------------------------------------------------------*/
/*  BlackPhoneInterface */
/*--------------------------------------------------------------*/

/*  type definitions  */


mtype = { teardown , downack , other , setup , upack  , offhook ,
  accepted , avail , dialed , onhook , waiting , rejected ,
    unknown , unavail , nullified , none };

   mtype = { post_process };
   mtype = { initial , ringing , dialing , talking , connecting_c ,
       silent , final , ringback , busytone , errortone ,
       disconnected };

   mtype = { idle , c_work };

  typedef Transition {
       mtype dest;
       chan in_chan;
       chan out_chan[1];
  bool en_flag = false;
   };

  typedef in_q {
    byte box_in = 0;
```

```
  byte old_c_in = 1;
  byte c_in = 2;
  byte v_in = 3;
  byte a_in = 4;
  bool box_in_ready = true;
  bool old_c_in_ready = false;
  bool c_in_ready = false;
  bool a_in_ready = true;
  byte selected
};

chan glob_ins[5] = [0] of {mtype};

typedef out_q {
  byte box_out = 0;
  byte c_out = 1;
  chan c_hold = [3] of {mtype};
};

chan glob_outs[2] = [0] of {mtype};

typedef internal {
  chan internal_c = [0] of {mtype};
  };

typedef SnapShot {
  mtype cs;
  mtype cs_post_process;
  in_q inq;
  out_q out;
  internal intq;};

/*-------------------------------------------------*/
/*  global variable declarations  */
/*-------------------------------------------------*/
bool rcv_setup = false;
bool send_upack = false;
bool c_rcv_teardown = false;
bool c_send_downack = false;
bool c_send_teardown = false;
bool c_send_setup = false;
bool c_rcv_upack = false;

SnapShot ss;
  mtype sig;
```

```
    mtype inter_sig;

Transition t[51];

 byte counter = 0;
 byte last_call = 0;
 byte pp_call = 0;
 byte current_call = 0;

inline setup_initial() {
   ss.inq.c_in_ready = true;
  };

 inline teardown_cleanup() {
   ss.inq.c_in_ready = false;
   ss.inq.old_c_in_ready = true;
  };
inline reset() {
   rcv_setup = false;
   send_upack = false;
   c_rcv_teardown = false;
   c_send_downack = false;
   c_send_teardown = false;
   c_send_setup = false;
   c_rcv_upack = false;

   if
   ::glob_ins[ss.inq.box_in] ? sig -> ss.inq.selected = ss.inq.box_in;
   ::glob_ins[ss.inq.c_in] ? sig -> ss.inq.selected = ss.inq.c_in;
   ::glob_ins[ss.inq.v_in] ? sig -> ss.inq.selected = ss.inq.v_in;
   ::glob_ins[ss.inq.a_in] ? sig -> ss.inq.selected = ss.inq.a_in;
   fi
};

inline en_events(n) {
 if
    ::(n == 0) && ss.inq.selected == ss.inq.box_in;
    ::(n == 1) && ss.inq.selected == ss.inq.a_in;
    ::(n == 2) && ss.inq.selected == ss.inq.v_in;
    ::(n == 3) && ss.inq.selected == ss.inq.a_in;
    ::(n == 4) && ss.inq.selected == ss.inq.a_in;
    ::(n == 5) && ss.inq.selected == ss.inq.v_in;
    ::(n == 6) && ss.inq.selected == ss.inq.v_in;
    ::(n == 7) && ss.inq.selected == ss.inq.v_in;
    ::(n == 8) && ss.inq.selected == ss.inq.c_in;
```

```
    ::(n == 9) && ss.inq.selected == ss.inq.c_in;
    ::(n == 10) && ss.inq.selected == ss.inq.c_in;
    ::(n == 11) && ss.inq.selected == ss.inq.c_in;
    ::(n == 12) && ss.inq.selected == ss.inq.a_in;
    ::(n == 13) && ss.inq.selected == ss.inq.c_in;
    ::(n == 14) && ss.inq.selected == ss.inq.v_in;
    ::(n == 15) && ss.inq.selected == ss.inq.v_in;
    ::(n == 16) && ss.inq.selected == ss.inq.v_in;
    ::(n == 17) && ss.inq.selected == ss.inq.c_in;
    ::(n == 18) && ss.inq.selected == ss.inq.c_in;
    ::(n == 19) && ss.inq.selected == ss.inq.c_in;
    ::(n == 20) && ss.inq.selected == ss.inq.c_in;
    ::(n == 21) && ss.inq.selected == ss.inq.a_in;
    ::(n == 22) && ss.inq.selected == ss.inq.v_in;
    ::(n == 23) && ss.inq.selected == ss.inq.v_in;
    ::(n == 24) && ss.inq.selected == ss.inq.v_in;
    ::(n == 25) && ss.inq.selected == ss.inq.c_in;
    ::(n == 26) && ss.inq.selected == ss.inq.c_in;
    ::(n == 27) && ss.inq.selected == ss.inq.c_in;
    ::(n == 28) && ss.inq.selected == ss.inq.c_in;
    ::(n == 29) && ss.inq.selected == ss.inq.c_in;
    ::(n == 30) && ss.inq.selected == ss.inq.a_in;
    ::(n == 31) && ss.inq.selected == ss.inq.v_in;
    ::(n == 32) && ss.inq.selected == ss.inq.v_in;
    ::(n == 33) && ss.inq.selected == ss.inq.v_in;
    ::(n == 34) && ss.inq.selected == ss.inq.c_in;
    ::(n == 35) && ss.inq.selected == ss.inq.c_in;
    ::(n == 36) && ss.inq.selected == ss.inq.c_in;
    ::(n == 37) && ss.inq.selected == ss.inq.c_in;
    ::(n == 38) && ss.inq.selected == ss.inq.a_in;
    ::(n == 39) && ss.inq.selected == ss.inq.v_in;
    ::(n == 40) && ss.inq.selected == ss.inq.v_in;
    ::(n == 41) && ss.inq.selected == ss.inq.v_in;
    ::(n == 42) && ss.inq.selected == ss.inq.v_in;
    ::(n == 43) && ss.inq.selected == ss.inq.c_in;
    ::(n == 44) && ss.inq.selected == ss.inq.c_in;
    ::(n == 45) && ss.inq.selected == ss.inq.c_in;
    ::(n == 46) && ss.inq.selected == ss.inq.c_in;
    ::(n == 47) && ss.inq.selected == ss.inq.a_in;
    ::(n == 48) && ss.inq.selected == ss.inq.a_in;
    ::(n == 49) && true;
    ::(n == 50) && ss.inq.selected == ss.inq.old_c_in;
  fi;
};
```

```
inline reset_pp() {
  if
  :: glob_ins[ss.inq.old_c_in] ? sig -> ss.inq.selected = ss.inq.old_c_in
;
  fi;
};


inline en_cond(n) {
if

  ::(n == 0) && ( sig == setup );
  ::(n == 1) && ( sig == offhook );

  ::(n == 2) && ( sig == accepted );

  ::(n == 3) && ( sig == dialed );
  ::(n == 4) && ( sig == onhook );

  ::(n == 5) && ( sig == waiting );
  ::(n == 6) && ( sig ==  rejected );
  ::(n == 7) && ( sig == nullified );
  ::(n == 8) && ( sig == unknown );
  ::(n == 9) && ( sig == unavail );
  ::(n == 10) && ( sig == none );
  ::(n == 11) && ( sig == teardown );
  ::(n == 12) && ( sig == onhook );

  ::(n == 13) && ( sig == upack );

  ::(n == 14) && ( sig == waiting );
  ::(n == 15) && ( sig == accepted );
  ::(n == 16) && ( sig ==  rejected );
  ::(n == 17) && ( sig == unknown );
  ::(n == 18) && ( sig == unavail );
  ::(n == 19) && ( sig == avail );
  ::(n == 20) && ( sig == teardown );
  ::(n == 21) && ( sig == onhook );


  ::(n == 22) && ( sig == accepted );
  ::(n == 23) && ( sig ==  rejected );
  ::(n == 24) && ( sig == nullified );
  ::(n == 25) && ( sig == unknown );
  ::(n == 26) && ( sig == unavail );
```

```
  ::(n == 27) && ( sig == avail );
  ::(n == 28) && ( sig == none );
  ::(n == 29) && ( sig == teardown );
  ::(n == 30) && ( sig == onhook );

  ::(n == 31) && ( sig == waiting );
  ::(n == 32) && ( sig == accepted );
  ::(n == 33) && ( sig == nullified );
  ::(n == 34) && ( sig == unknown );
  ::(n == 35) && ( sig == avail );
  ::(n == 36) && ( sig == none );
  ::(n == 37) && ( sig == teardown );
  ::(n == 38) && ( sig == onhook );

  ::(n == 39) && ( sig == waiting );
  ::(n == 40) && ( sig == accepted );
  ::(n == 41) && ( sig ==  rejected );
  ::(n == 42) && ( sig == nullified );
  ::(n == 43) && ( sig == unavail );
  ::(n == 44) && ( sig == avail );
  ::(n == 45) && ( sig == none );
  ::(n == 46) && ( sig == teardown );
  ::(n == 47) && ( sig == onhook );

  ::(n == 48) && ( sig == onhook );

   ::(n == 49) && inter_sig == post_process;
   ::(n == 50) && sig == downack;
 fi;
};

/*-----------------------------------------------------------*/
/*   Inline Functions   */

inline next_trans(n) {
if


  ::(n == 0) ->    rcv_setup = true;
       setup_initial();
       glob_outs[ss.out.c_out] ! upack;
       send_upack = true;
       current_call = counter;
       ss.cs = t[0].dest;

```

```
::(n == 1) ->         ss.cs = t[1].dest;

::(n == 2) ->         glob_outs[ss.out.c_out] ! avail;
    ss.cs = t[2].dest;

::(n == 3) ->         c_send_setup = true;
    setup_initial();
    glob_outs[ss.out.c_out] ! setup;
    ss.cs = t[3].dest;

::(n == 4) ->         ss.cs = t[4].dest;
    pp_call = current_call;

::(n == 5) ->         ss.cs = t[5].dest;

::(n == 6) ->         ss.cs = t[6].dest;

::(n == 7) ->         ss.cs = t[7].dest;

::(n == 8) ->         ss.cs = t[8].dest;

::(n == 9) ->         ss.cs = t[9].dest;

::(n == 10) ->        ss.cs = t[10].dest;

::(n == 11) ->          c_rcv_teardown = true;
      c_send_downack = true;
      glob_outs[ss.out.c_out] ! downack;
      ss.inq.c_in_ready = false;
      ss.cs = t[11].dest;

::(n == 12) ->        c_send_teardown = true;
    glob_outs[ss.out.c_out] ! teardown;
    ss.intq.internal_c ! post_process;
    pp_call = current_call;
    ss.cs = t[12].dest;

::(n == 13) ->        c_rcv_upack = true;
      ss.cs = t[13].dest;

::(n == 14) ->        ss.cs = t[14].dest;

::(n == 15) ->        ss.cs = t[15].dest;

::(n == 16) ->        ss.cs = t[16].dest;
```

```
::(n == 17) ->        ss.cs = t[17].dest;

::(n == 18) ->        ss.cs = t[18].dest;

::(n == 19) ->        ss.cs = t[19].dest;

::(n == 20) ->          c_rcv_teardown = true;
      c_send_downack = true;
      glob_outs[ss.out.c_out] ! downack;
      ss.inq.c_in_ready = false;
      ss.cs = t[20].dest;

::(n == 21) ->        c_send_teardown = true;
    glob_outs[ss.out.c_out] ! teardown;
     ss.intq.internal_c ! post_process;
     pp_call = current_call;
    ss.cs = t[21].dest;

::(n == 22) ->        ss.cs = t[22].dest;

::(n == 23) ->        ss.cs = t[23].dest;

::(n == 24) ->        ss.cs = t[24].dest;

::(n == 25) ->        ss.cs = t[25].dest;

::(n == 26) ->        ss.cs = t[26].dest;

::(n == 27) ->        ss.cs = t[27].dest;

::(n == 28) ->        ss.cs = t[28].dest;

::(n == 29) ->          c_rcv_teardown = true;
      c_send_downack = true;
      glob_outs[ss.out.c_out] ! downack;
      ss.inq.c_in_ready = false;
      ss.cs = t[29].dest;

::(n == 30) ->        c_send_teardown = true;
    glob_outs[ss.out.c_out] ! teardown;
    ss.intq.internal_c ! post_process;
    pp_call = current_call;
    ss.cs = t[30].dest;
```

```
::(n == 31) ->        ss.cs = t[31].dest;

::(n == 32) ->        ss.cs = t[32].dest;

::(n == 33) ->        ss.cs = t[33].dest;

::(n == 34) ->        ss.cs = t[34].dest;

::(n == 35) ->        ss.cs = t[35].dest;

::(n == 36) ->        ss.cs = t[36].dest;

::(n == 37) ->          c_rcv_teardown = true;
      c_send_downack = true;
      glob_outs[ss.out.c_out] ! downack;
      ss.inq.c_in_ready = false;
      ss.cs = t[37].dest;

::(n == 38) ->        c_send_teardown = true;
    glob_outs[ss.out.c_out] ! teardown;
    ss.intq.internal_c ! post_process;
    pp_call = current_call;
    ss.cs = t[38].dest;

::(n == 39) ->        ss.cs = t[39].dest;

::(n == 40) ->        ss.cs = t[40].dest;

::(n == 41) ->        ss.cs = t[41].dest;

::(n == 42) ->        ss.cs = t[42].dest;

::(n == 43) ->        ss.cs = t[43].dest;

::(n == 44) ->        ss.cs = t[44].dest;

::(n == 45) ->        ss.cs = t[45].dest;

::(n == 46) ->          c_rcv_teardown = true;
      c_send_downack = true;
      glob_outs[ss.out.c_out] ! downack;
      ss.inq.c_in_ready = false;
      ss.cs = t[46].dest;

::(n == 47) ->        c_send_teardown = true;
```

```
      glob_outs[ss.out.c_out] ! teardown;
       ss.intq.internal_c ! post_process;
      pp_call = current_call;
      ss.cs = t[47].dest;

  ::(n == 48) ->       ss.cs = t[48].dest;

 ::(n == 49) ->     ss.cs_post_process = t[49].dest;

 ::(n == 50) ->     ss.inq.old_c_in_ready = false;
    ss.cs_post_process = t[50].dest;

 fi;
};

 inline en_trans(n) {
    if
    ::en_events(n) ->
      if
      ::en_cond(n) -> t[n].en_flag = true;
      ::else t[n].en_flag = false;
      fi;
    ::else t[n].en_flag = false;
    fi;
 };

active proctype BPI() {
    ss.cs = initial;

   t[0].dest = ringing;
    t[0].in_chan = glob_ins[ss.inq.box_in];

    t[1].dest = dialing;
    t[1].in_chan = glob_ins[ss.inq.a_in];

    t[2].dest = talking;
    t[2].in_chan = glob_ins[ss.inq.v_in];

    t[3].dest = connecting_c;
    t[3].in_chan = glob_ins[ss.inq.a_in];

    t[4].dest = initial;
    t[4].in_chan = glob_ins[ss.inq.a_in];

    t[5].dest = ringback;
```

```
t[5].in_chan = glob_ins[ss.inq.v_in];

t[6].dest = busytone;
t[6].in_chan = glob_ins[ss.inq.v_in];

t[7].dest = silent;
t[7].in_chan = glob_ins[ss.inq.v_in];

t[8].dest = busytone;
t[8].in_chan = glob_ins[ss.inq.c_in];

t[9].dest = busytone;
t[9].in_chan = glob_ins[ss.inq.c_in];

t[10].dest = silent;
t[10].in_chan = glob_ins[ss.inq.c_in];

t[11].dest = disconnected;
t[11].in_chan = glob_ins[ss.inq.c_in];

t[12].dest = initial;
t[12].in_chan = glob_ins[ss.inq.a_in];

t[13].dest = silent;
t[13].in_chan = glob_ins[ss.inq.c_in];

t[14].dest = ringback;
t[14].in_chan = glob_ins[ss.inq.v_in];

t[15].dest = talking;
t[15].in_chan = glob_ins[ss.inq.v_in];

t[16].dest = busytone;
t[16].in_chan = glob_ins[ss.inq.v_in];

t[17].dest = errortone;
t[17].in_chan = glob_ins[ss.inq.c_in];

t[18].dest = busytone;
t[18].in_chan = glob_ins[ss.inq.c_in];

t[19].dest = talking;
t[19].in_chan = glob_ins[ss.inq.c_in];

t[20].dest = disconnected;
```

```
t[20].in_chan = glob_ins[ss.inq.c_in];

t[21].dest = initial;
t[21].in_chan = glob_ins[ss.inq.a_in];

t[22].dest = talking;
t[22].in_chan = glob_ins[ss.inq.v_in];

t[23].dest = busytone;
t[23].in_chan = glob_ins[ss.inq.v_in];

t[24].dest = silent;
t[24].in_chan = glob_ins[ss.inq.v_in];

t[25].dest = errortone;
t[25].in_chan = glob_ins[ss.inq.c_in];

t[26].dest = busytone;
t[26].in_chan = glob_ins[ss.inq.c_in];

t[27].dest = talking;
t[27].in_chan = glob_ins[ss.inq.c_in];

t[28].dest = silent;
t[28].in_chan = glob_ins[ss.inq.c_in];

t[29].dest = disconnected;
t[29].in_chan = glob_ins[ss.inq.c_in];

t[30].dest = initial;
t[30].in_chan = glob_ins[ss.inq.a_in];

t[31].dest = ringback;
t[31].in_chan = glob_ins[ss.inq.v_in];

t[32].dest = talking;
t[32].in_chan = glob_ins[ss.inq.v_in];

t[33].dest = silent;
t[33].in_chan = glob_ins[ss.inq.v_in];

t[34].dest = errortone;
t[34].in_chan = glob_ins[ss.inq.c_in];

t[35].dest = talking;
```

```
    t[35].in_chan = glob_ins[ss.inq.c_in];

    t[36].dest = silent;
    t[36].in_chan = glob_ins[ss.inq.c_in];

    t[37].dest = disconnected;
    t[37].in_chan = glob_ins[ss.inq.c_in];

    t[38].dest = initial;
    t[38].in_chan = glob_ins[ss.inq.a_in];

    t[39].dest = ringback;
    t[39].in_chan = glob_ins[ss.inq.v_in];

    t[40].dest = talking;
    t[40].in_chan = glob_ins[ss.inq.v_in];

    t[41].dest = busytone;
    t[41].in_chan = glob_ins[ss.inq.v_in];

    t[42].dest = silent;
    t[42].in_chan = glob_ins[ss.inq.v_in];

    t[43].dest = busytone;
    t[43].in_chan = glob_ins[ss.inq.c_in];

    t[44].dest = talking;
    t[44].in_chan = glob_ins[ss.inq.c_in];

    t[45].dest = silent;
    t[45].in_chan = glob_ins[ss.inq.c_in];

    t[46].dest = disconnected;
    t[46].in_chan = glob_ins[ss.inq.c_in];

    t[47].dest = initial;
    t[47].in_chan = glob_ins[ss.inq.a_in];

    t[48].dest = initial;
    t[48].in_chan = glob_ins[ss.inq.a_in];

end_initial_state:
 atomic {
      reset();
      en_trans(0);
```

```
        en_trans(1);

        if
        ::t[0].en_flag -> next_trans(0); goto ringing_state;
        ::t[1].en_flag -> next_trans(1); goto dialing_state;
        ::else -> goto end_initial_state;

        fi;
}

ringing_state:
 atomic {
        reset();
        en_trans(2);

        if
        ::t[2].en_flag -> next_trans(2); goto talking_state;
        ::else -> goto ringing_state;

        fi;
}

dialing_state:
 atomic {
        reset();
        en_trans(3);
        en_trans(4);

        if
        ::t[3].en_flag -> next_trans(3); goto connecting_c_state;
        ::t[4].en_flag -> next_trans(4); goto end_initial_state;
        ::else -> goto dialing_state;

        fi;
}

talking_state:
 atomic {
        reset();
        en_trans(5);
        en_trans(6);
        en_trans(7);
        en_trans(8);
        en_trans(9);
        en_trans(10);
```

```
      en_trans(11);
      en_trans(12);

      if
      ::t[5].en_flag -> next_trans(5); goto ringback_state;
      ::t[6].en_flag -> next_trans(6); goto busytone_state;
      ::t[7].en_flag -> next_trans(7); goto silent_state;
      ::t[8].en_flag -> next_trans(8); goto busytone_state;
      ::t[9].en_flag -> next_trans(9); goto busytone_state;
      ::t[10].en_flag -> next_trans(10); goto silent_state;
      ::t[11].en_flag -> next_trans(11); goto disconnected_state;
      ::t[12].en_flag -> next_trans(12); goto end_initial_state;
      ::else -> goto talking_state;

      fi;
}

connecting_c_state:
 atomic {
      reset();
      en_trans(13);

      if
      ::t[13].en_flag -> next_trans(13); goto silent_state;
      ::else -> goto connecting_c_state;

      fi;
}

silent_state:
 atomic {
      reset();
      en_trans(14);
      en_trans(15);
      en_trans(16);
      en_trans(17);
      en_trans(18);
      en_trans(19);
      en_trans(20);
      en_trans(21);

      if
      ::t[14].en_flag -> next_trans(14); goto ringback_state;
      ::t[15].en_flag -> next_trans(15); goto talking_state;
      ::t[16].en_flag -> next_trans(16); goto busytone_state;
```

```
    ::t[17].en_flag -> next_trans(17); goto errortone_state;
    ::t[18].en_flag -> next_trans(18); goto busytone_state;
    ::t[19].en_flag -> next_trans(19); goto talking_state;
    ::t[20].en_flag -> next_trans(20); goto disconnected_state;
    ::t[21].en_flag -> next_trans(21); goto end_initial_state;
    ::else -> goto silent_state;

    fi;
}

ringback_state:
 atomic {
    reset();
    en_trans(22);
    en_trans(23);
    en_trans(24);
    en_trans(25);
    en_trans(26);
    en_trans(27);
    en_trans(28);
    en_trans(29);
    en_trans(30);

    if
    ::t[22].en_flag -> next_trans(22); goto talking_state;
    ::t[23].en_flag -> next_trans(23); goto busytone_state;
    ::t[24].en_flag -> next_trans(24); goto silent_state;
    ::t[25].en_flag -> next_trans(25); goto errortone_state;
    ::t[26].en_flag -> next_trans(26); goto busytone_state;
    ::t[27].en_flag -> next_trans(27); goto talking_state;
    ::t[28].en_flag -> next_trans(28); goto silent_state;
    ::t[29].en_flag -> next_trans(29); goto disconnected_state;
    ::t[30].en_flag -> next_trans(30); goto end_initial_state;
    ::else -> goto ringback_state;

    fi;
}

busytone_state:
 atomic {
    reset();
    en_trans(31);
    en_trans(32);
    en_trans(33);
    en_trans(34);
```

```
      en_trans(35);
      en_trans(36);
      en_trans(37);
      en_trans(38);

      if
      ::t[31].en_flag -> next_trans(31); goto ringback_state;
      ::t[32].en_flag -> next_trans(32); goto talking_state;
      ::t[33].en_flag -> next_trans(33); goto silent_state;
      ::t[34].en_flag -> next_trans(34); goto errortone_state;
      ::t[35].en_flag -> next_trans(35); goto talking_state;
      ::t[36].en_flag -> next_trans(36); goto silent_state;
      ::t[37].en_flag -> next_trans(37); goto disconnected_state;
      ::t[38].en_flag -> next_trans(38); goto end_initial_state;
      ::else -> goto busytone_state;

      fi;
}

errortone_state:
 atomic {
      reset();
      en_trans(39);
      en_trans(40);
      en_trans(41);
      en_trans(42);
      en_trans(43);
      en_trans(44);
      en_trans(45);
      en_trans(46);
      en_trans(47);

      if
      ::t[39].en_flag -> next_trans(39); goto ringback_state;
      ::t[40].en_flag -> next_trans(40); goto talking_state;
      ::t[41].en_flag -> next_trans(41); goto busytone_state;
      ::t[42].en_flag -> next_trans(42); goto silent_state;
      ::t[43].en_flag -> next_trans(43); goto busytone_state;
      ::t[44].en_flag -> next_trans(44); goto talking_state;
      ::t[45].en_flag -> next_trans(45); goto silent_state;
      ::t[46].en_flag -> next_trans(46); goto disconnected_state;
      ::t[47].en_flag -> next_trans(47); goto end_initial_state;
      ::else -> goto errortone_state;

      fi;
```

```
}

disconnected_state:
 atomic {
      reset();
      en_trans(48);

      if
      ::t[48].en_flag -> next_trans(48); goto end_initial_state;
      ::else -> goto disconnected_state;

      fi;
}

      error_state:

   skip;
};

active proctype pp() {
   byte inter_sig1;
   ss.cs_post_process = idle;


   t[49].dest = c_work;
   t[50].dest = idle;
   t[50].in_chan = glob_ins[ss.inq.old_c_in];



 end_idle_state:
 atomic {
   ss.intq.internal_c ? inter_sig1;
      en_trans(49);

      if
      ::t[49].en_flag -> next_trans(49); goto c_work_state;
      ::else -> goto end_idle_state;

      fi;
}


c_work_state:
 atomic {
```

```
    reset_pp();
      en_trans(50);

      if
      ::t[50].en_flag -> next_trans(50); goto end_idle_state;
      ::else -> goto c_work_state;

      fi;
}
}

active proctype env() {

 end:
 do
 ::ss.inq.box_in_ready && (ss.cs == initial) ->
    counter = counter + 1;
    glob_ins[ss.inq.box_in]!setup;
 ::ss.inq.a_in_ready ->
 if
 ::(ss.cs == initial) -> glob_ins[ss.inq.a_in] ! offhook;
    glob_ins[ss.inq.a_in] ! dialed;
 ::!(ss.cs == initial) && !(ss.cs == ringing) && !(ss.inq.old_c_in_ready
   ) ->
    glob_ins[ss.inq.a_in] ! onhook;
::(ss.cs == disconnected) -> glob_ins[ss.inq.a_in] ! onhook;
 ::else -> glob_ins[ss.inq.a_in] ! other;
 fi;
 ::ss.inq.c_in_ready && !(ss.cs == ringing) ->
 if
    ::glob_ins[ss.inq.c_in] ! teardown;
    ::!(ss.cs == errortone) -> glob_ins[ss.inq.c_in] ! unknown;
    ::!(ss.cs == busytone) -> glob_ins[ss.inq.c_in] ! unavail;
    ::!(ss.cs == talking) -> glob_ins[ss.inq.c_in] ! avail;
    ::!(ss.cs == silent) -> glob_ins[ss.inq.c_in] ! none;
    ::!(ss.cs == talking) -> glob_ins[ss.inq.v_in] ! accepted;
    ::!(ss.cs == ringback) -> glob_ins[ss.inq.v_in] ! waiting;
    ::!(ss.cs == busytone) -> glob_ins[ss.inq.v_in] ! rejected;
    ::!(ss.cs == nullified) -> glob_ins[ss.inq.v_in] ! nullified;
 fi;
 ::ss.inq.old_c_in_ready -> glob_ins[ss.inq.old_c_in] ! downack;
 od
 unless {
 if
 ::atomic{ glob_outs[ss.out.c_out] ? setup -> glob_ins[ss.inq.c_in] !
```

```
upack; }
 ::atomic{ glob_outs[ss.out.c_out] ? upack -> glob_ins[ss.inq.v_in] ! accepted; }
 ::glob_outs[ss.out.c_out] ? avail;
 ::glob_outs[ss.out.c_out] ? downack;
 ::atomic{ glob_outs[ss.out.c_out] ? teardown -> teardown_cleanup(); }
 fi;
 }
 goto end;
};
never{(BPI@end_initial_state && !(pp@end_idle_state)) && !(pp_call == current_call)}
```

References

[1] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. *MOCHA: Modularity in model checking*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer Berlin / Heidelberg, 1998. 3

[2] Gregory W. Bond, Franjo Ivancic, Nils Klarlund, and Richard Trefler. Eclipse feature logic analysis. In *2nd IP-Telephony Workshop*, pages 49–56, New York, April 2001. 2

[3] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT press, Cambridge, 1999. 25

[4] Alma L. Juarez Dominguez. Verification of dfc call protocol correctness criteria. Master's thesis, School of Computer Science, University of Waterloo, 2005. 3, 68, 69

[5] Alma L. Juarez Dominguez and Nancy A. Day. Compositional reasoning for port-based distributed systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 376–379. ACM, 2005. 3

[6] Naghmeh Ghafari and Richard Trefler. Piecewise fifo channels are analyzable. In *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag New York Inc, 2006. 4

[7] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, second edition, 2003. 3, 4, 18, 19, 93

[8] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, second edition, 2004. 24

[9] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, 1998. 2, 10

[10] Zarrin Langari and Richard Trefler. Formal modeling of communication protocols by graph transformation. In *FM 06*, volume 4085 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2006. 3

[11] Zarrin Langari and Richard Trefler. Application of graph transformation in verification of dynamic systems. In *Proceedings of the 7th International Conference on Integrated Formal Methods*, pages 261–276. Springer-Verlag, 2009. 4

[12] Yuan Peng. Mapping boxtalk to promela model. Master's thesis, School of Computer Science, University of Waterloo, 2007. x, xi, 4, 5, 28, 29, 56, 68, 92, 99

[13] Mary Shaw and David Garlan. *Software Architecture*. Prentice-Hall, Inc., 1996. 2, 9

[14] 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*, volume 173, pages 395–420, 1999. 3

[15] Pamela Zave and Michael Jackson. A call abstraction for component coordination. In *Proceedings of the ICALP2002 satellite workshop: Formal Methods and Component Interaction*, June 2002. xi, 2, 13, 16