Mapping Boxtalk to Promela Model

by

Yuan Peng

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, 2007

©Yuan Peng, 2007
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

A telecommunication feature is an optional or incremental unit of functionality, such as call display (CD) and call forwarding (CF). A feature interaction occurs when, in the presence of other features, the actual behavior of a feature becomes inconsistent with its specified behavior. This feature interaction problem is a long-existing problem in telephony, and it becomes an increasingly pressing problem as more and more sophisticated features are developed and put into use. It takes a lot of effort to test that the addition of a new feature to a system doesn’t affect any existing features in an undesired way.

Distributed Feature Composition (DFC) proposed by Michael Jackson and Pamela Zave, is an architectural approach to the feature interaction problem. Telecommunication features are modeled as independent components, which we call boxes. Boxes are composed in a pipe-and-filter-like sequence to form an end-to-end call. Our work studies the behaviour of single feature boxes. We translate BoxTalk specifications into another format, that is more conducive to automated reasoning. We build formal models on the translated format, then the formal models are checked by a model checker, SPIN, against DFC compliance properties written in Linear Temporal Logic (LTL). From BoxTalk specifications to Promela models, the translation takes steps: 1) Explicating BoxTalk, which expands BoxTalk macros and presents its implicit behaviours as explicit transitions. 2) Define BoxTalk semantics in terms of Template Semantics. 3) Construct Promela model from Template Semantics HTS. Our case studies exercised this translation process, and the resulting models are proven to hold desired properties.
Acknowledgements

I would like to express my deep gratitude to my supervisor, Professor Joanne M. Atlee, for her guidance, knowledge, patience, time and energy, which have been invaluable to my research and the writing of my thesis. It is rewarding to have her. Without her, I could not imagine that I could get this far and finally graduate.

Special thanks to my committee members, Professor Daniel Berry and Professor Richard Trefler, for taking their precious time to review this thesis and provide valuable comments.

I appreciate Professor Nancy A. Day, for her valuable inputs to my research work. I appreciate Dr. Pamela Zave, for her clear explanations and quick responses to my questions, and detailed reviews on the explication and Template Semantics. Thanks Jianwei Niu, for her help on Template Semantics and encouragement.

Thanks to my friends, Yun Lu, Yun Xia, Yongsheng Yang and Heather Zhao, for listening and for their understanding. Thanks to Sandy Cheng and her husband Henry Du, for providing me a quiet place to write my thesis. Thanks to my manager Greg Balsdon and team lead Thomas Chang, for their support in terms of timing and schedule.

Most of all, I thank my father and mother for their love, trust and taking care of my young kids. My mom is always there for me whenever I need her.

None of this would have been possible without my husband, Ken. During the past Christmas season and on countless weekends, he showed great patience in playing with and taking care of the kids, so that I was able to work on the thesis.

I dedicate my thesis to my dear daughters, Gloria and Alyssa, who bringing joy, strength, peace and a true meaning to my life.
# Table of Contents

Chapter 1  Introduction .......................................................................................................................... 1

1.1  Motivations ...................................................................................................................................... 1

1.1.1  Feature Interactions .................................................................................................................. 1

1.2  Related Work .................................................................................................................................. 1

1.2.1  Contributions of This Work ...................................................................................................... 7

1.3  Organization of This Document ..................................................................................................... 7

Chapter 2  Background .......................................................................................................................... 8

2.1  DFC .................................................................................................................................................. 8

2.1.1  Calls and Usages ....................................................................................................................... 8

2.1.2  DFC Protocol ............................................................................................................................ 9

2.1.3  Box Classes: Free or Bound ..................................................................................................... 10

2.1.4  Calls, Call Variables, Port IDs ................................................................................................ 10

2.1.5  Free Box Vs. Bound Box .......................................................................................................... 11

2.2  BoxTalk ......................................................................................................................................... 12

2.2.1  States .......................................................................................................................................... 13

2.2.2  Transitions .................................................................................................................................. 13

2.2.3  Example: Call Waiting ............................................................................................................. 14

2.2.4  Sending and Receiving Signals ................................................................................................ 15

2.2.5  Conditions .................................................................................................................................. 15

2.2.6  Assignments .............................................................................................................................. 15

Chapter 3  Explicating BoxTalk ......................................................................................................... 17

3.1  Explicate BoxTalk – free box ......................................................................................................... 17

3.1.1  Expand Macros .......................................................................................................................... 18

3.1.1.1  Macros explicitly used in FTB ............................................................................................... 19

3.1.1.2  Macros implicitly used in FTB .............................................................................................. 19

3.1.1.3  Macros not used in FTB ........................................................................................................ 20

3.1.2  Hold Queue .................................................................................................................................. 21

3.1.3  Signal Linked Calls .................................................................................................................... 21

3.1.4  Feature Termination .................................................................................................................. 21
3.1.5  error State ..................................................................................................................... 22
3.1.6  abandonConnection state ............................................................................................. 22
3.2  Explicate BoxTalk – bound box .................................................................................. 22

Chapter 4  Template Semantics .......................................................................................... 28
4.1  Template Semantics ............................................................................................................. 28
4.1.1  Syntax of HTS ............................................................................................................... 28
4.1.2  Semantics of HTS ......................................................................................................... 29
4.2  BoxTalk Definition ................................................................................................................. 31
4.2.1  Syntactic Mapping ........................................................................................................ 31
4.2.2  Semantic Mapping ........................................................................................................ 32

Chapter 5  Mapping BoxTalk to Promela Model .................................................................... 35
5.1  Model Checker ..................................................................................................................... 35
5.1.1  SPIN ................................................................................................................................................. 35
5.1.2  Promela ........................................................................................................................................ 35
5.1.1.1  Processes ..................................................................................................................... 36
5.1.1.2  Data Objects ................................................................................................................ 36
5.1.1.3  Message Channels ....................................................................................................... 37
5.1.1.4  Rules of Execution ........................................................................................................ 38
5.1.1.5  Compound Statements ................................................................................................. 39
5.1.1.6  inline Functions ............................................................................................................. 42
5.2  The Promela Model .............................................................................................................. 42
5.2.1  Free Error Interface ...................................................................................................... 43
5.2.2  Type Definitions and Global Variable Declarations .................................................. 44
5.2.3  inline Functions ................................................................................................................. 46
5.2.4  Processes ..................................................................................................................... 48
5.2.4.1  Feature Box Process .................................................................................................... 49
5.2.4.2  Environment Process ................................................................................................... 50
5.2.5 .............................................................................................................................................. 51
Bound Transparent Box ................................................................................................................ 51

Chapter 6  Case Studies ........................................................................................................... 55
6.1  Receive Voice Mail .............................................................................................................. 55
6.2  Answer Confirm ..................................................................................................................... 59
List of Figures

Figure 1: Usage ............................................................................................................................... 8
Figure 2: Call setup ........................................................................................................................... 9
Figure 3: Piecewise setup ............................................................................................................... 9
Figure 4: Ports ............................................................................................................................... 11
Figure 5: Call teardown ................................................................................................................. 12
Figure 6: Call Waiting ..................................................................................................................... 14
Figure 7: Sets -- portAllocated, known, active .............................................................................. 16
Figure 8: FTB (Original specification) ............................................................................................ 17
Figure 9: FTB (Explicated model) ................................................................................................. 18
Figure 10: FTB -- abandonConnection state ............................................................................... 22
Figure 11: BTB (Original specification) .......................................................................................... 23
Figure 12: BTB (Explicated model) : main .................................................................................... 25
Figure 13: BTB (Explicated model) : post-process ....................................................................... 26
Figure 14: HTS .............................................................................................................................. 28
Figure 15: Step semantics ............................................................................................................. 30
Figure 16: Process Interleaving ..................................................................................................... 39
Figure 17: Atomic Step .................................................................................................................. 40
Figure 18: Architecture ................................................................................................................ 42
Figure 19: Architecture (bound feature box) ............................................................................... 43
Figure 20: EI (Original specification) ............................................................................................ 43
Figure 21: EI (Explicated model) ................................................................................................. 44
Figure 22: RVM (Original Specification) ....................................................................................... 55
Figure 23: RVM (Explicated Model) ............................................................................................. 57
Figure 24: Answer Confirm (Original Specification) ....................................................................... 59
Figure 25: AC (Explicated Model) ............................................................................................... 61
Figure 26: BPI (Original Specification) ........................................................................................ 63
Figure 27: BPI (Explicated Model) ............................................................................................... 65
Figure 28: BPI (Post-Processing) ................................................................................................. 65
List of Tables

Table 1: Basic Feature Context (BFC) ............................................................................................2
Table 2: Example Feature Models .................................................................................................3
Table 3: Example Feature Composition .........................................................................................4
Table 4: Chisel -- Sequence of events for receiving a call ..............................................................5
Table 5: Triggering Information Extraction ....................................................................................6
Table 6: Macro expansion .............................................................................................................20
Table 7: Template semantics ........................................................................................................30
Table 8: Map BoxTalk to HTS -- States ......................................................................................31
Table 9: Map BoxTalk to HTS -- Event .......................................................................................31
Table 10: Map BoxTalk to HTS -- Variables ..............................................................................32
Table 11: Map BoxTalk to HTS -- Explicit Transition .................................................................32
Table 12: Template Parameters for BoxTalk ..............................................................................33
Table 13: Promela Data Types .....................................................................................................37
Table 14: LTL ...............................................................................................................................67
Chapter 1 Introduction

1.1 Motivations

1.1.1 Feature Interactions

A telecommunication feature is an optional or incremental unit of functionality, such as Call Display (CD) and Call Forwarding (CF). A feature interaction occurs when, in the presence of other features, the actual behavior of a feature becomes inconsistent with its specified behavior. A simple example of a feature interaction is the combination of Call Waiting and Answer Call features. Call Waiting alerts subscriber A with a special tone when A is called while he is already on the phone. Answer Call directs the calling party to an answering service when subscriber A does not answer the phone after a designated number of rings, or when he is already on the phone. When A is already connected to a call and a second call comes in, should A be alerted about the incoming call or should the second call be forwarded to an answering service?

This feature interaction problem is a long-existing problem in telephony, and it becomes an increasingly pressing problem as more and more sophisticated features are developed and put into use. It takes a lot of effort to test that the addition of a new feature to a system doesn’t affect any existing features in an undesired way.

1.2 Related Work

A number of attempts have been made to resolve the feature interaction problem. They mainly fall into three categories:

- Formal modeling of features, and analysis of feature compositions
- Architectural approaches that avoid feature interactions by restricting how features execute or communicate
- Detection and resolution of feature interactions at run-time

F. Joe LIN and Yow-Jian LIN present a building block approach; features are modeled by Basic Call Models (BCMs) and Basic Feature Contexts (BFCs) [1].

BCMs represent the protocols at the user-network interface of telephone switching systems for establishing basic telephone service. There are two BCMs: one for call origination, called Originating BCM (OBCM), and the other for call termination, called Terminating BCM (TBCM). The states in BCMs indicate the various stages that a call progresses through until its completion.

BFCs represent compositions of environment (e.g., ORIG, TERM), and system modules (e.g., OBCM, TBCM, SYS). As shown in Table 1, there are three BFCs: 1) Originating BFC models the feature...
context that involves only the user in the originating side. The feature context is decomposed into two blocks: Block ORIG models the originating user’s behavior, and block SYS represents an abstraction of the system environment. 2) Terminating BFC models the feature context that involves only the user on the terminating side. Likewise, the feature context is decomposed into two blocks: Block TERM models the terminating user’s behavior, and block SYS models how the environment my interact with the TBCM. 3) Two-party BFC models the feature context that involves both the originating side and the terminating sides users. One can see that a two-party BFC is effectively the composition of an originating BFC and a terminating BFC. In BFC models, the arrows between the blocks indicate the communication channels.

### Table 1: Basic Feature Context (BFC)

<table>
<thead>
<tr>
<th>FEATURE CONTEXT</th>
<th>MODEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Originating BFC</td>
<td><img src="image" alt="Originating BFC Diagram" /></td>
</tr>
<tr>
<td>Terminating BFC</td>
<td><img src="image" alt="Terminating BFC Diagram" /></td>
</tr>
<tr>
<td>Two-Party BFC</td>
<td><img src="image" alt="Two-Party BFC Diagram" /></td>
</tr>
</tbody>
</table>

A feature can be modeled by appending a block representing the feature logic to the appropriate BFC. Table 2 shows feature models of *Originating Call Screening* (OCS), *Denied Termination* (DT) and *Call Waiting* (CW). As the figure indicates, OCS applies only to an originating BFC, and DT applies only to a terminating BFC. With the CW feature, if the first call is an outgoing call, then it applies to both an originating BFC and a terminating BFC, which is the case shown in the figure. The dashed lines connect a feature’s logic with its interacting BCMs.
<table>
<thead>
<tr>
<th>FEATURE</th>
<th>DESCRIPTION</th>
<th>MODEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Originating Call Screen</td>
<td>The user defines a screening list of telephone numbers. All outgoing calls to those numbers will be blocked.</td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>Denied Termination</td>
<td>The user denies the termination of all incoming calls.</td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>Call Waiting</td>
<td>The user is notified of an incoming call when he is already on another call. Then, he can switch between the two calls by flashing the hook.</td>
<td><img src="#" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**Table 2: Example Feature Models**

To detect interactions among features, the features under study are composed into a single context. Table 3 shows a composition of the feature models in Table 2. Then the composite model is analyzed by verification tools.
D. Amyot et al. use two complementary methodologies -- Use Case Maps (UCM) and LOTOS [2]-- to design features and detect feature interactions. UCM is a notation similar to Message Sequence Chart (MSC), without explicitly defining message exchanges between components. Instead, the causal relationships between functions from different components are captured. LOTOS is an executable notation, that has some similarities to SDL. In this method, feature designs are first captured in UCM, and are then hand-translated to LOTOS. This translation requires some formalization of the model, and experience shows that several potential design errors can be caught during this exercise. Then, the formal LOTOS model is executed to see its response to possible actions at each state, to see whether the specification accepts or rejects certain scenarios. The model can also be exhaustively analyzed by means of reachability analysis and model-checking tools.

Glenn Bruns et al. present a specification approach to the feature interaction problem based on the idea of features as service transformers [10]. Services and features are the two main elements in their models. A service describes what is provided by a server. It is defined by concrete events, local variables, abstract events, and reaction statements. Concrete events are the input and output events of a service. When an input event is received by a service, the service interprets it as an abstract event. For example, the concrete event *offhook* may be interpreted as an abstract *newcall* or

<table>
<thead>
<tr>
<th>FEATURE COMPOSITION (SINGLE USER)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Originating Call Screen + Denied Termination + Call Waiting</td>
</tr>
<tr>
<td>ORIG</td>
</tr>
<tr>
<td>CW</td>
</tr>
<tr>
<td>TERM</td>
</tr>
</tbody>
</table>

**Table 3: Example Feature Composition**
answer call event. With a given abstract event, reaction statements define variable updates and concrete output events.

A feature is defined as a service transformer. It transforms the service by defining new input events and new response events. The transformed service can then serve as the base of a subsequent service transformer.

With a model described as above, two kinds of feature interactions can be detected: 1) order-interaction, meaning that the order in which features are applied to a service affects the behaviour of the resulting system; and 2) output-interaction, meaning that a service can reach a state in which it outputs two events that conflict with each other. Checking for order-interaction is very expensive, as it involves checking that two systems in which features are applied in different orders are equivalent. Unless we know that order-interactions do not exist, checking for output-interactions is also very expensive.

The Chisel specification language proposed by A. Aho et al. [11] defines a feature as a sequence of events that can occur when a feature is active. For example, the function of receiving a call can be captured by the sequences shown in Table 4:

<table>
<thead>
<tr>
<th>SEQUENCE OF EVENTS</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartRinging A B, Off-hook A, StopRinging A B, On-hook A, Disconnect A B</td>
<td>The telephone rings, a user answers, and later hangs up to terminate the call</td>
</tr>
<tr>
<td>StartRinging A B, Off-hook A, StopRinging A B, Disconnect A B, On-hook A</td>
<td>As above, except that the calling party disconnects before the called party goes on-hook.</td>
</tr>
<tr>
<td>StartRinging A B, Disconnect A B, StopRinging A B</td>
<td>The telephone rings and the calling party disconnects before it is answered.</td>
</tr>
</tbody>
</table>

Table 4: Chisel -- Sequence of events for receiving a call

Each of the events in above table consists of an event type and one or more parameters. Most of the events are self-explanatory, like StartRinging A B, which says that subscriber A’s telephone rings for an incoming call from subscriber B. Disconnect A B says that the connection between A and B has been broken.

Every feature in Chisel executes on a specified platform, which defines the set of event sequences that are allowed. Two operations, Projection and Union, can be applied to feature specifications. The union operation is used to combine sets of event sequences, and the projection operation restricts a set of event sequences to the event types that a feature recognizes. A feature interaction is detected if, after projecting the sequences in the union of two features’ event sequences onto the event types recognized by one of the features, the resulting set of sequences is different from the original set of sequences for the feature. For example, Features Three-Way Calling and Call Waiting both use the same event Flash. The following sequence illustrates a Call Waiting sequence whose projection is not a Three-Way Calling sequence. The events are recognized by both features and the events are recognized only by Call Waiting.
StartRinging m n, Off-hook m, StopRinging A B, CallWaitingTone m q, Flash m, Disconnect m n, On-hook m

The above sequence represents a scenario in which subscriber m’s telephone rings, m answers, its telephone stops ringing, a call waiting tone alerts m that a second call is coming from q, m flashes the hook to switch to answer q, the first calling party n disconnects, and m hangs up eventually.

When we project the above sequence onto Three-Way Calling’s events, the projection eliminates event CallWaitingTone m q, which is not recognized by the Three-Way Calling feature. However, the resulting sequence is not a valid event sequence for Three-Way Calling because Three-Way Calling expects a Dialtone event after a Flash event.

Chisel is sufficiently precise to support automated translation to more formal languages, such as Message Sequence Chart, by ways of the translation tool SCF3/Sculptor. The converted formal language can be verified using formal methods.

H. Jouve et al. presents a static analysis method for detecting interactions between two features [12]. A service specification comprises a diagram expressing exchanges between the network (the abstraction of the complete physical network and its components) and the connected phones. A state in the diagram represents the telephone status. For example, idle(A) means that telephone A is not in use. Every transition between two states is labelled by a phone message and a network answer. A phone message such as A.call(B) means that telephone A calls telephone B. A network message like start(disctone).A means that the network issues a disconnect tone to telephone A.

The detection method combines two major steps: 1) extract the triggering information of features, and 2) analyze if two features share the same triggering messages (direct interaction) or one feature’s intentional message (an actual action) coincides with the second feature’s triggering message. Consider two features Call Forward on Busy (CFB) and Terminating Call Screening (TCS). CFB forwards incoming calls to another phone when its subscriber’s phone is busy. TCS prevents incoming calls from phones chosen by the subscriber. Table 5 illustrates the interactions:

<table>
<thead>
<tr>
<th>CALL CONFIGURATION</th>
<th>TRIGGERING MESSAGE</th>
<th>TRIGGERING CONDITION</th>
<th>INTENTIONAL MESSAGE</th>
<th>STATE CONDITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>B: CFB(C)</td>
<td>A. call(B)</td>
<td>~ idle(B) dialing(A)</td>
<td>A. call(C)</td>
<td>Idle(C) dialing(A)</td>
</tr>
<tr>
<td>B: TCS(A)</td>
<td>A. call(B)</td>
<td>dialing(A)</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>C: TCS(A)</td>
<td>A. call(C)</td>
<td>dialing(A)</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 5: Triggering Information Extraction

The call configurations B: CFB(C) and B: TCS(A) share the same triggering message, which reveals a direct interaction scenario: Telephone B subscribes to both CFB and TCS features. CFB forwards its incoming calls to phone C when B is in use, while TCS plays a refusal message to incoming calls from phone A. Then, how should telephone B respond to calls from A when telephone B is busy?

The intentional message of call configuration B: CFB(C) coincides with the triggering message of call configuration C: TCS(A), which reveals an indirect interaction scenario: B’s CFB feature forwards
incoming calls for telephone B to telephone C when B is in use, and C’s TCS feature plays a refusal message to incoming calls from phone A. Then, how should phone C responds to a call forwarded from B, and originally from A?

Distributed Feature Composition (DFC) proposed by Michael Jackson and Pamela Zave[3], is an architectural approach. Telecommunication features are modeled as independent components, which we call boxes. Boxes are composed in a pipe-and-filter-like sequence to form an end-to-end call. More detailed information about DFC will be given in the next chapter.

1.2.1 Contributions of This Work

Our work studies the behaviour of single feature boxes. We translate BoxTalk specifications [4] into another format, that is more conducive to automated reasoning. We build formal models of the translated format, which are then checked by a model checker, SPIN, against DFC compliance properties written in Linear Temporal Logic (LTL). From BoxTalk specifications to Promela models, the translation takes steps: 1) Explicating the BoxTalk model, which expands BoxTalk macros and presents its implicit behaviours as explicit transitions. 2) Define BoxTalk semantics in terms of Template Semantics [5]. 3) Construct a Promela model from Template Semantics. Our case studies exercise this translation process, and the resulting models are proven to hold desired properties. By guaranteeing good behavior of components, our work allows the feature box developer to focus more on coordinating components.

1.3 Organization of This Document

The rest of the document is organized as follows. Chapter 2 describes the background knowledge of DFC and BoxTalk. Chapter 3 presents the process of explicating BoxTalk. Chapter 4 explains Template Semantics and presents a Template Semantics definition for BoxTalk. Chapter 5 introduces background on Promela and SPIN and presents Promela models of feature boxes. Chapter 6 concludes with case studies. Chapter 7 summarizes our work.
Chapter 2 Background

In this chapter, we provide the background needed to understand this thesis. In particular, we briefly introduce some relevant concepts of Distributed Feature Composition and BoxTalk. The model checker SPIN and its input language Promela are introduced in later chapters.

2.1 DFC

The Distributed Feature Composition (DFC) architecture decomposes complex behavior into multiple simpler features, that are coordinated by the DFC protocol. DFC is designed for feature modularity, structured feature composition and analysis of feature interactions. It has a pipe-and-filter architectural style, features are developed independently and behave individually, and the system is a composition of the features.

2.1.1 Calls and Usages

A traditional customer call is referred to as a usage. Telecommunication features are viewed as boxes in DFC. A usage is built up by chaining feature boxes together with internal calls -- a featureless, point-to-point connection with a two-way signaling channel and any number of media channels. An internal call is like a plain old-fashioned telephone call. Besides representing features, boxes also represent interfaces to devices (e.g., telephones), trunks and other resources. An internal call is shown as an arrow from the box that placed the call to the box that received the call. So, a usage in DFC will look like Figure 1:

![Figure 1: Usage](image)

This usage involves two features and three internal calls. Line interfaces a and b provide interfaces to the caller’s and callee’s telephone devices, respectively. We refer to the caller and callee interface box as the source interface box and target interface box, respectively. Upstream and downstream represent relative positions between boxes along the flow of the pipeline. From the reference of any box F in usage, an upstream box is closer to the source interface box than box F is. For example, feature1 is upstream of feature2. Hereafter, we use the term call to refer the internal call placed between consecutive boxes.
2.1.2 DFC Protocol

A usage, as shown in Figure 1, is dynamically assembled. Each box sets up and tears down its internal calls by using the DFC protocol. The router embedded in the DFC architecture is responsible for setting up usages.

The primary DFC signals are setup, upack, teardown and downack. To initiate a call, a box sends a setup signal to the router. The router determines the next box in the usage and then forwards the setup signal to that box. Callee box accepts this call by sending an upack signal back and then propagates the setup signal to the router. Again, the router determines the next box in the chain. At this point, a call is set up between two boxes and a bi-directional signal channel between these two boxes has been established. Boxes can pass any signals along calls in both directions. A call setup is illustrated by Figure 2:

![Figure 2: Call setup](image)

One might notice that call setup in DFC is piecewise. That is, each internal call is completed before the next call in the pipeline is started.

![Figure 3: Piecewise setup](image)
Behaving this way allows more autonomy to each component (box). Imagine that each feature box doesn’t acknowledge a setup message immediately, but rather waits to receive an outcome from the rest of the call setup process. Then all feature boxes would be frozen until the usage reaches its endpoint. None of the boxes would receive or send message until an outcome message is sent back from the downstream boxes. Features would not be able to respond to the caller hanging up until after the end-to-end call is established or is determined to have failed.

Similarly, to actively terminate a call, box sends a teardown signal to its neighbor(s) in the usage. The receiving box sends back a downack signal and then propagates the teardown signal further down the chain. Upon receiving a downack signal, a box terminates and is freed from the usage.

The teardown signals may crossover. That is, box may receive and react to an incoming teardown signal while waiting for an acknowledgement of a previously sent teardown. We will see more discussion of this in later chapters.

Besides the basic signals for setting up and tearing down calls, there are four status signals: unknown, avail, unavail, and none. Different from signals for setting up and tearing down calls, status signals are generated by the target interface box and are linearly send back upstream along the established usage. Together, they cover different outcomes of a call setup: unknown indicates an invalid target address (i.e., the dialed number does not match a valid address), unavail represents a callee who is already connected in another usage, avail indicates that the call has successfully reached the callee’s interface box, and none cancels the effect of any of the three previous signals on a user interface.

2.1.3 Box Classes: Free or Bound

Boxes are categorized into two classes: Free and Bound. Most features are implemented as free feature boxes. This means that a new run-time instance of this feature is spawned each time the feature is included in a usage. In contrast, there is only one instance of each bound feature, and it is permanently associated with the user. That one instance of the feature is included in every usage involving its subscriber.

Bound feature boxes are needed to implement features that coordinate multiple usages, like Call Waiting (CW). This feature works by coordinating all of the signals that flow along all of the usages in which its subscriber is involved.

2.1.4 Calls, Call Variables, Port IDs

Calls are internal calls between boxes. The ports of a box are referred by Call variables. Port IDs are the identifiers of allocated communication channels. Abstractly, we think of a call variable as being assigned the value of a call, but technically, it is assigned the value of a port ID.

Internal calls are channels that connect named ports of consecutive boxes. In addition, every box has a special boxport for receiving setup requests from the router. Figure 4 shows two feature boxes, each with three ports: boxport, callee port in and caller port out. When feature box 1 receives a setup
message from the router, it allocates port in to participate in the requested internal call by sending back acknowledgement upack to its upstream neighbour. Then, port out sends out a new setup signal to the router to continue the usage. The router determines the next downstream neighbor, feature box 2, and forwards to it the setup signal, along with box 1’s address. Feature box 2 sends an upack directly to feature box 1 via this address. At this point, a connection is established between port out of box 1 and port in of box 2. Behaving the same way as box 1, box 2 continues to extend the usage.

![Figure 4: Ports](image)

Bound boxes have more ports. For example, a CW feature box has four ports that enable it to be engaged in three calls; A conference feature box has n ports which enable it to handle n-1 conference calls at one time. Like free boxes, a bound box has a reserved boxport for receiving setup signal from the DFC router.

### 2.1.5 Free Box Vs. Bound Box

With respect to setup and teardown processes, free and bound box behave differently. A free box accepts one setup signal during its lifetime. Actually, it is spawned in response to a setup signal, and processing the setup signal is the first thing it does. Subsequent setup signals are rejected by sending signal sequence: upack, unavail, teardown. Unlike free boxes, a bound box may receive and accept multiple setup signals. When a new setup signal comes from the subscriber, all old calls are torn down and the new setup is propagated. When a new setup signal comes from the far party while this box is involved in a usage, then depending on the feature, the box may reject the new setup by sending upack, unavail, teardown, or it may accept the new setup and alert the subscriber, as in Call Waiting feature.
The call-teardown process starts when a *teardown* signal is received from a neighbouring feature (as shown in Figure 5) or when the feature initiates a *teardown* signal. The process ends when *teardown* signal has propagated to all of the feature box’s calls, and all of the signals have been acknowledged. In Figure 5a, the feature box is involved in a usage with calls *a* and *b* connecting it to its neighbours. In Figure 5b, the box responds to a *teardown* signal from call *a* with a *downack* signal on call *a*. At this point, call *a* is considered torn down, and the box is disconnected from its one side neighbour. Then, the box propagates the *teardown* signal to its other neighbouring feature. If there is any *teardown* signal cross-over, the box responds with a *downack* signal. In Figure 5c, the box receives a *downack* signal. At this point, call *b* is considered torn down as well, and the box is disconnected from both neighbours.

A free box dies when it is freed from a usage. As a bound box, as soon as a *teardown* signal is issued on each of its calls, it is ready to be included in a new usage. The rest of the teardown process (waiting for *downack*) is processed in the background.

In summary, the DFC architecture offers great independency to its feature boxes, so that each feature can be developed, enhanced, and verified separately. It also provides a clean and structured means to study the feature-interaction problem.

In the next section, we discuss the contents of a box.

### 2.2 BoxTalk

BoxTalk is a high-level, domain-specific programming language for programming DFC feature boxes. BoxTalk defines features in terms of extended finite state machines, with typed variables, but without concurrency or state hierarchy. A box receives signals through named ports, performs local actions (eg. changing state, setting local variables), and outputs signals to named ports.

Although a box can have an unlimited number of ports, it may be awkward to refer to a call by means of specific ports, as different situations may call for different ports. BoxTalk introduced a higher-level concept, *call variable*, that refers to ports. A box program declares some number of *call variables*. A call variable either has a distinguished initial value *noCall* or a unique ID, which refers to a port. Call variables and ports are at different levels and serve for different purposes. Ports are more concrete, they are physically located on boxes. Like all other variables, call variables are more abstract and flexible, they may refer to different calls at different times. For example, Call Waiting has call variable *w* that refer to the call that is currently on hold. We will see the convenience of call variables in later examples.
2.2.1 States

BoxTalk has five types of states: initial, stable, transient, termination and final states. The first four types of states have graphical representations, while the last type, final state, exists only semantically.

An initial state is represented as a small black circle. There is exactly one initial state for each feature box instance. A feature box in an initial state is ready to receive new calls.

Stable states are represented as rectangles. A feature box rests in a stable state until a new signal is received from the environment.

Transient states are represented as large clear circles. Transient states are used to decompose a complex transition into a sequence of transition segments that execute at the same time. As such, transient states are intermediate states rather than real execution states. A box does not read new input when in a transient state. However, transitions out of a transient state may lead to different next states based on the evaluation of the box’s local variables. At least one transition out of a transient state should always be possible - execution should never be blocked in a transient state.

Termination states are represented as heavy bars. A box transitions to a termination state on receipt of a teardown signal if the default action is not overridden by any explicit transition. When in a termination state, a box may react to teardown signals from other named ports by responding with a downack signal but throws away all other signals except downack.

Final states have no graphical representation. There is exactly one final state for each free box. It is reached from a termination state upon receipt or sending of a downack signal. In a final state, all active calls have been ended and the box is freed from any usage.

2.2.2 Transitions

Transitions reflect state changes. They are depicted as arrows from the current state to a destination state. Transitions out of an initial or stable state have labels whose format is “trigger / actions”. Transitions leaving transient states have labels whose format is “guard / actions”. The guard and actions are optional.

A trigger of a transition could be a simple receive event: one input signal read on one input queue, which is depicted as callVariable ? signal; or a macro that combines the receipt of a signal and a sequence of actions, and has the form macroName(callVariable). For example, c? teardown means that a teardown signal is read by the port associated with call variable c; rcv(c) means that a setup signal arrives and the new call is allocated a new port ID, which is assigned to call variable c.

Similarly, actions could be simple send actions, which are depicted as callVariable ! signal; or assignments that change the values of call variables; or macros that combine signal sending and other actions; or any combination of the above. The execution of an action shouldn’t be blocked if its precondition is satisfied.

A guard is a predicate on the state of the box. Which transition to take from a transient state is determined by the evaluation of the transitions’ guards. Guards on branches are not necessarily
exclusive. However, the disjunction of the transitions’ guards must evaluate to true to ensure executability.

Thus, a complete transition is triggered by an input signal, may enabled by guard predicates, and may be followed by one or more transition segments out of transient states, and ending in a stable or termination state.

### 2.2.3 Example: Call Waiting

Let us look at the **Call Waiting** (CW) feature box as an example. CW is a feature that allows a user to be notified of another incoming call while a call is already in progress, and gives the user the ability to answer the second call while the first call remains on hold.

![Call Waiting Diagram](image)

**Figure 6: Call Waiting**

The observable behaviours of the CW box are presented in three stable states: **transparent**, **call_waiting** and **all_held**. The box executes via transitions. Transitions may cause the box to change to a new state, or may return execution to the same state.

In state **transparent**, the CW feature is inactive. The subscriber participates in a call as usual, until the receipt of another call request. Then the subscriber hears a special tone that indicates the presence of a new call. In the meantime, the box transitions into the **call_waiting** state. State **call_waiting** represents an active CW feature. The called subscriber may answer the new call by flashing the hook to put on hold the original conversation. By flashing hook repeatedly, the subscriber is able to switch back and forth between the two calls. Any party may hang-up in the **call_waiting** state. If the call that is on hold hangs up, it is not noticeable by the two parties who are speaking and the box transitions to the **transparent** state. If the connected call hangs up, the box transitions to the **all_held** state. The remaining two calls cannot talk to each other unless the subscriber flashes the hook. This action leads the box back to the **transparent** state.
2.2.4 Sending and Receiving Signals

We have touched the sending and receiving of signals in earlier sections. Simply, there is a call variable associated with each port. The call variable stores the port identifier assigned to the port at run-time. All signals sent to or received from a call variable are actually sent to or received from the variable's associated port. However, a call variable is able to efficiently represent the idea of role change during a usage. For example, in the call_waiting state, the event of flashing hook represented as s? switch causes the feature to swap port identifiers stored in the connected and on-hold call variables a (active) and w (waiting), respectively. Physically, the ports connected to each call remain unchanged.

We have discussed previously the signals for setting up and tearing down calls and for communicating status. In addition, some signals are feature specific, like the switch signal in CW. The switch signal is generated by flashing the hook, and is meaningful to the CW feature box only.

Signals pass between boxes along internal calls. How do signals pass through boxes? If two ports are signal-linked, represented as a tuple of two call variables inside a parentheses in a stable state, then signals can pass between the signal-linked ports, from the input port to the output port. This represents the default behavior of feature boxes. This default behavior can be overridden by transitions triggered by specific input signals under specific conditions (expressed as guards). For example, in the call_waiting state, a switch signal will cause a role change of call variables a and w. While in the transparent state, a switch signal doesn’t cause any change in observable behaviour of the box; switch signal will be thrown away after a self-transition.

2.2.5 Conditions

Conditions are predicates over local variables or over data parameters in received signals. For example, on receiving a setup signal, the CW box needs to determine from which direction the setup signal is travelling. A True evaluation of predicate s_from_subscriber represents a call originated by the subscriber. In contrast, a true evaluation of s_from_afar represents a call originated by a far party. Based on the value of these predicates, the feature either propagates the setup signal towards the far party (if the setup signal was received from the subscriber), or propagates it to the subscriber (if the setup signal was received from the far party). When the subscriber terminates a conversation while a third call is on hold, box will call the subscriber back. In this case, the predicate a_from_subscriber and a_from_afar are used to configure the new setup fields.

2.2.6 Assignments

Call variables hold port identifiers. In the CW example, call variable a stands for active: it stores the port ID associated with the far party that is actively connected to the subscriber; call variable w stands for waiting: it holds the port ID of the far party that is on hold. If the subscriber switches between the two remote parties, that switch can be realized by swapping the values of call variables a and w, syntactically represented as

---

1 S_from_subscriber – The "s" in the condition refers to the source field of the received input message.
\[ a, w = w, a; \]

so that variable \( a \) refers to the newly active call and variable \( w \) refers to the call now on hold.

Some other statements also change the value of call variables: \( \text{rcv}(c) \) receives a new call, and the port identifier of the new call is assigned to call variable \( c \); \( \text{new}(c) \) and \( \text{ctu}(c0,c) \) place new calls, and the port identifier of the new call is assigned to call variable \( c \).

Call variables can be classified in sets \( \text{portAllocated}, \text{known}, \) and \( \text{active} \). The set \( \text{portAllocated} \) contains the call variables associated with all calls currently allocated to ports. The set \( \text{known} \) is the union of calls that are accessible. The set \( \text{active} \) contains the identifiers of all active calls. They have subtle differences. A call becomes \( \text{active} \) immediately upon the receipt of or the start of a new \( \text{setup} \) signal. If the call that a call variable refers to has been torn down and this call variable hasn’t yet been assigned a new value, then this call variable is in \( \text{known} \), but not in \( \text{portAllocated} \).

The set variables \( \text{portAllocated}, \text{known} \) and \( \text{active} \) are not used in our explicated models.

Figure 7 further illustrates the differences between sets:

---

**Figure 7: Sets -- portAllocated, known, active**
Chapter 3 Explicating BoxTalk

There is a lot of implicit, unexpressed behaviour in a BoxTalk model. Behaviour that is common to all BoxTalk models is abstracted away to accentuate each model's unique behaviour. Examples of such abstractions include:

- **Macros**, such as $rcv(i)$ that implement a sequence of read, write, and assignment actions; $ctu(i, o)$ that represent one or more intermediate states and a sequence of transitions. They are implicit by default. Their behavior can be overridden by explicit transitions.

- **Hold queue**. When a feature box sends out a new or propagated setup signal, a hold queue is constructed for the caller port. All subsequent signals sent to that port are queued in the hold queue, until the port receives an acknowledgement that the call is successfully setup. For example, in $ctu(i, o)$, a setup signal is sent through named port $o$ to the router, to continue the usage. A hold queue $o.hold$ is constructed to hold the subsequent signals sent out on port $o$ until an upack signal is received on port $o$.

- **Signal linkage**. In some stable states, the feature box is connected to two neighbouring features, each via an active call. Signals from either neighbouring feature will be passed on to the other neighbour if not reacted to by transitions. This behaviour is the default reaction to receiving an input signal, but it can be overridden by explicit transitions.

- **Feature termination**. A free feature box transitions from a termination state to a final state on certain conditions.

Such implicit behaviour must be explicitly represented in a feature box model, in order to model check the model. This process is called **explication**. We will explain this process by way of examples.

3.1 Explicate BoxTalk – free box

Consider the feature box *Free Transparent Box* (FTB). Figure 8 shows its original BoxTalk specification.

![Figure 8: FTB (Original specification)](image)

FTB has two states (initial state and transparent state) and one transition from initial state to transparent state. Upon receiving a new call $i$, the feature box places new call $o$ and transitions from the initial state (represented as a small black circle) to transparent state. Once call $o$ is established, FTB acts as a signal-linkage between the two calls. That is, whenever the feature receives a signal

---

2 An intermediate state may be either stable or transient.
from either call, the feature forwards the signal to the other call -- as if the feature were transparent and were simply a pipe connecting calls \( i \) and \( o \).

Figure 9: FTB (Explicated model)

Figure 9 shows the result of explicating the specification from Figure 8. In explicated models, we use a rounded rectangle to represent a states without distinguishing graphically between stable or transient states. A double black circle represents the final state. We use a black circle to represent the initial state.

The explicated FTB specification contains eight states and almost twenty transitions. We will discuss the explication process in detail, starting with the transition from the initial state.

3.1.1 Expand Macros

BoxTalk use macros, such as \texttt{new()} and \texttt{rcv()} to combine a sequence of read and write actions, which builds moderate complexity into BoxTalk semantics. Handling complexity this way makes the box programmer’s life easier: he doesn’t have to program the full detailed behaviour himself. In addition, it enables a more reliable and consistent implementation, as the semantics of BoxTalk need only be verified and implemented once.

We discuss macros in the following order:

$\subseteq$ macros explicitly used in FTB – \texttt{rcv()} and \texttt{ctu()}

$\subseteq$ macros implicitly used in FTB – \texttt{gone()} and \texttt{end()}
§ macros not used in FTB – new()

3.1.1.1 Macros explicitly used in FTB

In original specification of FTB, macros \texttt{rcv()} and \texttt{ctu()} are explicitly used.

Macro \texttt{rcv}(i) combines a guard and action. It sets up a new call on receiving a \texttt{setup} signal. The new call is identified as \texttt{i}. The macro maps to \texttt{boxport ? setup / i ! upack}.

The \texttt{boxport} is a reserved port that is designated for \texttt{setup} signals from the DFC router. When the router has a \texttt{setup} to process, it determines the next box in the usage. In case of free boxes, the router spawns a new instance of the next box and then forwards the \texttt{setup} signal to the box’s \texttt{boxport}. In case of bound boxes, the router simply forwards the \texttt{setup} signal to box’s \texttt{boxport}. Upon a \texttt{setup} signal arriving on \texttt{boxport}, a box allocates a port for this new call, known by the box as call \texttt{i}. The acknowledgement \texttt{upack} is sent on the newly established call \texttt{i}. The \texttt{setup} signal contains the name of the sending port, the box knows to whom to send the \texttt{upack} signal. At this point, a call between the box and its upstream caller box has been established.

Macro \texttt{ctu()} is an action that propagates an input request to set up a usage, by sending a \texttt{setup} signal to the router to create the next call segment of the usage. We expand the macro into two transitions, one that sends the \texttt{setup} signal and waits in an intermediate state, and a second transition receives the corresponding \texttt{upack} signal.

Referring to the FTB example, \texttt{ctu(i, o)} maps to the state \texttt{connecting}, with call variable \texttt{i} and \texttt{o}, transition \texttt{o!setup} to \texttt{connecting} state, and transition \texttt{o?upack} from \texttt{connecting} state.

The box doesn’t know the next box in the usage, but it knows that port \texttt{o} will be the port to the resulting call, so the \texttt{setup} signal is sent out via port \texttt{o} to the router. The router uses information about the subscriber’s feature subscriptions to identify the next box in the usage, and to forward the \texttt{setup} signal to that target box. Box transitions from the intermediate state to \texttt{transparent} state once \texttt{upack} is received at port \texttt{o}.

3.1.1.2 Macros implicitly used in FTB

Besides the macros discussed above, macros \texttt{gone()} and \texttt{end()} are implicitly used in FTB.

In original specification of FTB, there is no explicit out-transitions of \texttt{transparent} state. Instead, the receipt of a \texttt{teardown} signal from an active call at any stable state terminates the entire box program, which includes ending all other active calls. Syntactically, the process we just described is expressed by BoxTalk macros \texttt{gone()} and \texttt{end()}.

Macro \texttt{gone(c)} implements behaviours in reaction to the box receiving its first \texttt{teardown} signal, on call \texttt{c}, assuming that the box has not sent out any \texttt{teardown} signals. The macro causes the box to transition to a termination state. The macro is fully expanded as \texttt{c?teardown / c!downack}; if there is any other active call, say \texttt{c0}, that is signal-linked with call \texttt{c}, then \texttt{gone(c)} will trigger macro \texttt{end(c0)}.

Macro \texttt{end(c0)} initiates the \texttt{teardown} of call \texttt{c0}. It begins with sending a \texttt{teardown} signal; however, the teardown phase is not completed until the other end of the call acknowledges the \texttt{teardown} via a
downack signal. Similar to ctu(), end(c0) maps to an intermediate state, with transitions entering and leaving the state, labeled with c!teardown and c?downack, respectively.

In transparent state of the original FTB specification, both gone(i) and end(o) are implicit. Macro gone(i) represents the receipt of a teardown signal on call i, the sending of a downack signal, and, the propagation of the teardown signal to end call o.

State terminatingO is the intermediate state introduced by end(o). The box sends a teardown signal on call o, and then waits in the intermediate state until acknowledgement downack arrives on call o.

Symmetrically, in transparent state of the original FTB specification, an implicit gone(o)/end(i) maps to state terminatingl with incoming transition labeled with o?teardown/o/downack/i/teardown, and outgoing transition to final state labeled i?downack.

3.1.1.3 Macros not used in FTB

Macro new(c) is used to initiate a new usage, via a setup signal to set up the first internal call of that usage. This new call is assigned to port c.

The semantics of new() is much like that of ctu(). The difference is that new() indicates initiative, whereas ctu() emphasizes propagating. To implement new(c), the feature box sends a setup signal to the DFC router and allocates port c to wait for the upack signal.

Like ctu(), the macro new(c) expands to two transitions and an intermediate state. c!setup transitions the box from initial state to the intermediate state, and transition c?upack leads the box from the intermediate state to transparent state.

Table 4 summarizes macro expansion we described above.

<table>
<thead>
<tr>
<th>MACRO</th>
<th>EXPANSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcv(i)</td>
<td><img src="rcv_diagram" alt="Diagram" /></td>
</tr>
<tr>
<td>ctu(i, o)</td>
<td><img src="ctu_diagram" alt="Diagram" /></td>
</tr>
<tr>
<td>gone(o)</td>
<td><img src="gone_diagram" alt="Diagram" /></td>
</tr>
<tr>
<td>end(i)</td>
<td><img src="end_diagram" alt="Diagram" /></td>
</tr>
<tr>
<td>new(c)</td>
<td><img src="new_diagram" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Table 6: Macro expansion
3.1.2 Hold Queue

Setting up a call consists of two phases: 1) sending a setup signal; and 2) receiving an acknowledgement signal upack. The setup signal is sent to the router, which forwards the signal to next box. Until the acknowledgement signal is received, the call is not yet established, and there is no “call” to send signals to. Instead, any signals destined for an unestablished call are held locally, in a **hold queue** we constructed that preserves the order in which the signals to the call were issued. Once the call is established, as evidenced by the upack signal, the held signals are forwarded to the call.

In the explicated FTB model, at the connecting state, call i is active while call o is not fully setup. Instead, a hold queue o.hold holds the signals sent to call o. This hold queue becomes active when the box sends the setup signal out on call o. Once call o is fully established, evidenced by the arrival of an upack signal on call o, the contents in the hold queue are output to call o, preserving the order in which signals were to sent to call o. The hold queue may overflow, in which case the box transitions to the error state, which we discuss in a later subsection.

3.1.3 Signal Linked Calls

A two-way signal linkage exists between pairs of calls at some states of the box. At such states, when a signal arrives from either call, the default behavior is that the signal does not trigger any action; instead, the box simply passes the signal to the other call. A state at which signal linkage is the only function is usually named “transparent state”, and we refer its behaviours as “transparent behaviours”. In other words, the presence of the feature box that is behaving transparently acts as if it were not in the usage, and instead, the two neighbouring features were directly connected to each other.

In the explicated FTB model in Figure 9, signal_linkage is represented as a pair of parenthesized call variables, (i,o), in a stable state. Transparent behaviour is realized by two transitions i?sig / o!sig and o?sig / i!sig, in which, sig represents any signal other than teardown. The teardown signal will trigger the feature box to transition to a termination state.

3.1.4 Feature Termination

In the termination states of feature boxes, all calls become inactive, but calls that are not fully torn down still have ports allocated to them. That means that the box still receives signals from those ports and discards them, except for signal teardown and downack. The box responds with a downack signal when it receives a teardown signal. The receipt of a downack signal indicates the completion of the teardown process: the ports allocated are disassociated, free box transitions to its final state, while bound box transitions to its initial state.

The explicated FTB specification has two termination states, terminatingO and terminatingI, and one final state. State terminatingO is reached by gone(i) from state transparent. In this state, the FTB box has no active calls. However, a port is still allocated for call o while the box waits for a downack acknowledgement on call o. The box reacts to a teardown signal on call o with a downack signal, and it ignores other signals. Symmetrically, state terminatingI is reached by gone(o) from state
transparent. The box waits for a downack acknowledgement on call i and reacts to a teardown signal with a downack signal. The final state is reached only when all ports are freed on receiving a teardown signal or a downack signal for each teardown signal sent. This feature box is terminated and withdrawn from the usage when it transitions to the final state.

3.1.5 error State

Error states are not part of the syntax and semantics of BoxTalk. We introduce error states to keep the size of the model finite, and to enable finite state analysis (like model checking). For example, we put a limit on the capacity of hold queues. When a hold queue is over-full, the box transitions to an error state. Error states are final states.

3.1.6 abandonConnection state

As a side effect of introducing a connecting intermediate state, more intermediate states might be introduced. State abandonConnection is such a state. It is reached when a teardown signal is received from call i in state connecting. To understand this scenario, we consider call i that is connected to the caller end, and the call o to the callee end being established. Before call o is established, the caller changes his mind and hangs up. Thus, the box receives setup and teardown signals in sequence from call i. The box propagates teardown to call o (i.e., saves the signal in hold queue o.hold). When call o is established on receiving the upack signal, the contents of the hold queue including the teardown signal are propagated to call o; call o responds with a downack signal. This sequence is depicted in Figure 10.

Figure 10: FTB -- abandonConnection state

3.2 Explicate BoxTalk – bound box

In this section, we present the explication of bound boxes. A bound box has some unique explications, which we will explain in detail when walking through the Bound Transparent Box (BTB) example. Figure 11 shows its original specification.
BTB has four states: initial state, orienting state, transparent state, and receiving state. The transparent state is a stable state, while orienting and receiving are transient states. The transient states are decision-making states: Depending on the evaluation of a local variable, the box takes different actions. Let us look at BTB starting from the initial state.

In the initial state of BTB, upon receiving a new setup signal, the box allocates a port for the call, stores the port ID in call variable t, and then transitions to the orienting state.

State orienting is a transient state. When sitting in this state, the box won’t take new signals from the environment. Instead, it tests predicate t_from_sub, which is true if the call originates from the subscriber of the feature box. The symbol ! stands for negation of the disjunction of other predicates on outgoing transitions (similar to an “else” clause). Thus, the box transitions from orienting to transparent state, and the box is added to a usage regardless of whether the subscriber is the original caller. Different actions are performed in the two cases. If the predicate evaluates to TRUE, the box associates the call with variable s, then continues the usage to far end. Otherwise, call variable f is associated with the port, and the usage extends to subscriber end.

Statements like s, t = t, - change the values of the call variables. In this assignment, call variable s gets the value of call variable t, while call variable t gets the value nocal (which is the default value of calls).

In the transparent state, a signal linkage is established between calls associated with variables s and f. A teardown signal from either end will lead the box to exit from the transparent state. Interestingly, the box goes to the initial state instead of to a termination state on teardown signals.

Unlike free boxes, bound boxes may receive and react to a setup signal at any stable state. If a new call request arrives when the box is sitting in the transparent state, then, as before, the box will allocate a port, and store the port ID in call variable t, and then test the predicate t_from_sub at transient state receiving. If the new call is issued by the subscriber, the box tears down old calls s and f and establishes the new call: the box transitions to transparent state with the new established call. Otherwise, the box announces its unavailability: it sends out the status signal unavail to indicate the box is busy. When the box is unavailable, signals upack, unavail, and teardown are sent in sequence: signal upack is in response to the setup signal, and signals unavail and teardown are issued to terminate the connection.
Because the box needs to test whether the incoming call is from the subscriber end and must act accordingly, call \( t \), a temporary call variable, is used to process new setup signals immediately. Right after sending back an acknowledgement, call variable \( t \) is evaluated with predicate \( t_{\text{from_sub}} \) and the box reacts accordingly. In summary, there are four situations:

- When receiving a setup signal in the initial state, and \( t_{\text{from_sub}} \) evaluates to true, call \( s \) (associated with the call from the subscriber) will take the value of call \( t \).
- When receiving a setup signal in the initial state, and \( t_{\text{from_sub}} \) evaluates to false, call \( f \) (associated with the call from the far party) will take the value of call \( t \).
- When receiving a new setup signal in any stable state other than initial, and \( t_{\text{from_sub}} \) evaluates to true, the box will tear down all current active calls and start over to establish new calls.
- When receiving a new setup signal in any stable state other than initial, and \( t_{\text{from_sub}} \) evaluates to false, the box will ignore the new setup signal by tearing down call \( t \), and keeping other active calls.

BTB reveals that the initial state of a bound box is the final state as well. As soon as the old call is terminated, the box is ready to accept new calls.
Figure 12: BTB (Explicated model) : main
Figure 13: BTB (Explicated model): post-process

Figure 12 together with Figure 13 show the explicated specification of BTB.

BTB is explicated as two concurrent finite state machines: one main machine and one post-processing machine. Whenever a teardown is sent out on a call variable in the main machine, this call variable is ready to take a new value. Thus, the post-processing machine is responsible for following through or tearing down the call associated with the variable’s port ID. In this way, BTB can deal with two usages asynchronously: the main machine can establish a new usage while the post-processing machine is tearing down the old one.

The main machine of BTB in the explicated model has nine states, in which, state initial, orienting, transparent and receiving correspond to the states in original specification. States connecting_f, connecting_s, deciding_1 and deciding_2 are intermediate state. State error is added to ensure finite analysis.

Just like state connecting in FTB, connecting_s and connecting_f are states in which the box waits for acknowledgement signals of a new call. However, the two states are stable states, in which a new setup signal could instead be received. States deciding_1 and deciding_2 are further introduced as decision-making states for reacting to new setup signals. They function as transient states, like state orienting in the original specification.

All decision-making states work in this manner: If the new call request is issued by the subscriber, the box tears down all existing calls and establishes a new one; if the new call request is issued by a far party, the box announces its unavailability and frees the port allocated to the new call.
We construct a hold queue for call variables \( f \) and \( s \) at state `connecting_f` and `connecting_s`, respectively. The `error` state is reached when either hold queue overflows.

The calls \( s \) and \( f \) are signal linked in the `transparent` state. Transparent behavior is realized in the explicated model by the pair of transitions `s?sig / f!sig` and `f?sig / s!sig`. When a `teardown` signal arrives from either call, the box transitions to the `initial` state, instead of to a termination state as in a free box. At the same time, a trigger to enable the `post-processing` machine is issued.

The `post-processing` machine cleans up terminating calls, by waiting for `unpack` and `downack` signals. The `post-processing` machine deals with one call at a time. For terminating call \( t \), it transitions to `t_work` state and waits for a `downack` signal on call \( t \). In the cases of call \( s \) and \( f \), if the call is terminated before the `unpack` signal is received from the callee, the `post-processing` machine will need to collect two acknowledgements: `unpack` then `downack`. Local variable `communicating` is used to determine whether there is an outstanding `unpack` signal to receive.

A bound box is not involved in any usage if both machine sit in their respective initial states and the set `portAllocated` is empty.
Chapter 4 Template Semantics

This work is part of a larger effort to map domain-specific notations to analysis tools, by way of template semantics. This chapter reviews template semantics, and expresses BoxTalk semantics in terms of template semantics models and semantic parameter values.

4.1 Template Semantics

Template semantics is a template-based approach to capture the semantics that are common among several model-based specification and design notations. By parameterizing notations’ common execution semantics, each notation can be described in terms of its parameters to the template semantics. Template semantics descriptions can also be the basis for tools that are configured using the semantic parameter values. In particular, we are interested in writing a translator from template-semantics models to SPIN, where the translator is configured using semantic parameter values. We can configure such a translator with semantic parameters that reflect BoxTalk semantics.

4.1.1 Syntax of HTS

The basic computation model is a nonconcurrent, hierarchical transition system (HTS). An HTS is an extended finite state machine, adapted from basic transition systems [7] and statecharts [8].

As shown in Figure 14, A hierarchical transition system (HTS) is an 8-tuple, < S, S^I, S^F, S^H, E, V, V^T, T>, where

S S is a finite set of states.
S^I S^I and S^F are predicates describing the sets of initial states and final states, respectively.

---

Figure 14: HTS
$S^H$ defines the state hierarchy as a partial ordering on states.
$E$ is a finite set of events.
$V$ is a finite set of typed variables.
$V_I$ is a predicate that defines the possible initial values of variables.
$T$ is a finite set of transitions of the form $<src, trig, cond, act, dest, prty>$.

where $src$ and $dest$ are the transition’s source and destination states, $trig$ represents triggering events, $cond$ is the transition’s guard condition (a predicate over $V$), $act$ is a sequence of actions that execute when the transition executes, and $prty$ is the transition’s optional explicitly-defined priority.

### 4.1.2 Semantics of HTS

The semantics of an HTS is presented in terms of snapshot relations.

A **snapshot** is an observable point in the execution of an HTS. Snapshots capture execution states that represent what control states the system is sitting in, what the current values of variables are, what internal events have been generated, and which transitions are enabled.

A snapshot is an 8-tuple $<CS, IE, AV, O, CS_a, AV_a, IE_a, I_a>$, where $CS$ represents current states, $IE$ represents current internal events, $AV$ represents current variable values, $O$ represents current outputs, and the rest four are auxiliary elements that accumulate data about states, variables values, and internal and external events, respectively.

Template semantics classifies events into two classes: Internal events $IE$ and external events $I_a$. In representing BoxTalk semantics, $I_a$ represents an HTS’s set of input queues carrying signals from the environment (i.e., neighbouring features), and, $IE$ represents the next signal to be processed – the signal at the head of a randomly chosen queue. In other words, only signals stored in IE directly impacts system’s behaviour.

An execution **step** in an HTS represents the execution of zero or more transitions. A transition executes because it is triggered by an event or is enabled by a guard condition over system variables. A **micro-step** results from executing exactly one transition. A **macro-step** is a sequence of zero or more micro-steps that is initiated by new inputs from the environment. That is, only at the beginning of macro-step, is an HTS sensing inputs from its environment. In Figure 15, each rounded rectangle box represents a system state (i.e, snapshot). At snapshot $SS_0$, an external input triggers a change to snapshot $SS_1$. However, this state is not stable; it further transitions to $SS_2$, and so on, until the execution reaches a stable snapshot, meaning that no more transitions are enabled unless new external events are received. The relation between micro-steps and macro-steps is shown in Figure 15.
The execution semantics are expressed in terms of a set of parameterized definitions that define allowable changes to snapshot values.

In Table 5, *italics* is used to represent template semantic parameterized definitions, and *bold* font is used to represent template parameters.

<table>
<thead>
<tr>
<th>Template Definitions</th>
<th>Template Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>reset_CS, reset_IE, reset_AV, reset_O etc.</td>
</tr>
<tr>
<td>enabled_trans</td>
<td>en_states, en_events, en_cond</td>
</tr>
<tr>
<td>apply</td>
<td>next_CS, next_IE, next_AV, next_O etc.</td>
</tr>
</tbody>
</table>

Table 7: Template semantics

reset(ss, I): ss’ – resets the current snapshot ss with new input I at the beginning of a macro-step, returning a new snapshot ss’. *reset()* is defined in terms of eight *reset_*X parameters, each of which specifies how a snapshot element X is reset.

enabled_trans(ss, T): T’ – computes the set of transitions enabled in the current snapshot ss by evaluating the source state(s), triggering event(s), and enabling condition(s) for each transition in the set T. The returned set T’ is a subset of T.

apply(ss, τ, ss’) – applies the executing transitions τ’s actions (i.e., τ’s generated events and variable assignments) to the current snapshot ss, to derive the next snapshot ss’. *apply()* is defined in terms of eight *next_*X parameters, each of which specifies how a snapshot element X is updated.

Template semantics users are expected to configure the semantics of their notation by providing values of template parameters.

- Up to eight *reset_*X parameters are used in the template definition of *reset()*: each parameter is a function that resets one snapshot element, X, removing old data and incorporating new system inputs I.

- Three *en_states, en_events, and en_cond* parameters are used in the template definition *enabled_trans()* These predicates specify when a transition is enabled with respect to its source state(s), its triggering event(s), and its enabling condition(s), respectively.
§ Up to eight next\_X parameters are used in the template definition apply(): each parameter is a predicate that constrains how one snapshot element X is updated with respect to a transition's actions.

§ Parameter pri defines a priority scheme among enabled transitions.

§ Parameter macro-semantics determines what macro-step semantics is used. Possible parameter values are simple or stable. In simple semantics, every macro-step is either a micro-step or an idle step, and new environmental inputs are sensed in every step. In stable semantics, a macro-step is a maximal sequence of micro-steps, starting with a reset() snapshot and ending with a stable snapshot, in which no transition is enabled.

4.2 BoxTalk Definition

BoxTalk can be defined in terms of Template Semantics. We first show how a BoxTalk syntactically maps onto a HTS, and then we present the semantic mapping. Providing a template semantics definition for BoxTalk integrates this work into a larger project, which maps domain-specific notations to analysis tools, including SMV [6] and SPIN.

4.2.1 Syntactic Mapping

As shown in Table 6, every BoxTalk state maps to an HTS state. Moreover, BoxTalk initial states map to HTS initial states, and BoxTalk final states map to HTS final states. There is no state hierarchy in BoxTalk.

<table>
<thead>
<tr>
<th>BOXTALK</th>
<th>HTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state</td>
<td>State</td>
</tr>
<tr>
<td>Transient state</td>
<td></td>
</tr>
<tr>
<td>Stable state</td>
<td></td>
</tr>
<tr>
<td>Termination state</td>
<td></td>
</tr>
<tr>
<td>Final state (from semantics)</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Map BoxTalk to HTS -- States

BoxTalk signals map to HTS events. Signals from a BoxTalk feature’s environment map to HTS external events. Generated signals, such as those that trigger post-processing machines, map to HTS internal events.

<table>
<thead>
<tr>
<th>BOXTALK</th>
<th>HTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal</td>
<td>Event</td>
</tr>
</tbody>
</table>

Table 9: Map BoxTalk to HTS -- Event
BoxTalk variables map to HTS variables. BoxTalk call variables hold port IDs and are initialized to a distinguished value \textit{noCall}. BoxTalk signal variables store the most recently received message from each input queue, and are initialized to a distinguished value \textit{noSig}.

Each call variable is associated, at most, with three signal queues: \textit{in}, \textit{out}, and \textit{hold}. Every signal queue is unidirectional: Feature boxes receive signals from queue \textit{in}, send out signals through queue \textit{out}, and store outgoing signals in queue \textit{hold}. In later this chapter, we will see how these queues are represented as snapshot elements, and manipulated by template parameters.

For calls between feature boxes, variable \textit{communicating} is associated with the caller port, and is used during the call set-up phase to indicate whether or not the call has been completely established. A call is not fully set-up until the signal \textit{upack} is received.

<table>
<thead>
<tr>
<th><strong>BOXTALK</strong></th>
<th><strong>INITIAL VALUE</strong></th>
<th><strong>HTS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Call variable: port ID</td>
<td>noCall</td>
<td></td>
</tr>
<tr>
<td>Signal variable: signal</td>
<td>noSig</td>
<td></td>
</tr>
<tr>
<td>\textit{communicating}: boolean</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>Feature-specific variable</td>
<td>&lt;feature-specific&gt;</td>
<td>Variable</td>
</tr>
</tbody>
</table>

Table 10: Map BoxTalk to HTS -- Variables

BoxTalk transitions map to HTS transitions, in HTS format <$src$, trig, cond, act, dest, prty>. All BoxTalk transitions have the same priority: \textit{H}, which denotes High priority. BoxTalk transitions that originate from transient states don’t have triggering events, so their triggering events are empty.

<table>
<thead>
<tr>
<th><strong>BOXTALK</strong></th>
<th><strong>HTS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$src$, trig [cond] / act, dest $src \in$ responsive states</td>
<td>&lt;$src$, trig, cond, act, dest, H&gt;</td>
</tr>
<tr>
<td>$src$, [cond] / act, dest $src \in$ transient state</td>
<td>&lt;$src$, $\emptyset$, cond, act, dest, H&gt;</td>
</tr>
</tbody>
</table>

Table 11: Map BoxTalk to HTS -- Explicit Transition

4.2.2 Semantic Mapping

We have discussed all syntactic mappings from BoxTalk models to HTS models. In this subsection, we talk about how to express BoxTalk execution semantics in terms of template semantics parameter values.

BoxTalk has stable macro-step semantics. That is, an execution step in BoxTalk starts by reading input from some input port, and executes all enabled transitions until the execution reaches a snapshot in which there are no more enabled transitions. Then a new macro-step begins.

A transition is enabled only when its source state belongs to the set of current states \textit{CS}, its triggering event is at the head of the chosen input queue, and its guard condition is satisfied by current variable.
Table 10 captures the definition of BoxTalk semantics in terms of template-semantics parameters. We explain the contents of Table 7, a row at a time.

<table>
<thead>
<tr>
<th>SNAPSHOT ELEMENT</th>
<th>START OF MACRO-STEP</th>
<th>EXECUTING TRANSITION $\tau$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CS'$</td>
<td>$CS$</td>
<td>$dest(\tau)$</td>
</tr>
<tr>
<td>$IE_a'$</td>
<td>choose $c \in (\text{portAllocated} \cup {\text{boxport}})$</td>
<td>$IE_a$</td>
</tr>
<tr>
<td>$IE'$</td>
<td>head($IE_a'.in$)</td>
<td>internal(gen($\tau$))</td>
</tr>
<tr>
<td>$I_a'$</td>
<td>${\text{enQ}(c.in, I) \mid c \in (\text{portAllocated} \cup {\text{boxport}} \setminus IE_a}) \cup {\text{deQ}(\text{enQ}(IE_a'.in, I))}$</td>
<td>$I_a$</td>
</tr>
<tr>
<td>$O'$</td>
<td>${c.hold \mid AV \models c.\text{communicating} = \text{false}} \cup {\text{emptyQ}(c.out) \mid AV \models c.\text{communicating} = \text{true}} \cup {\text{emptyQ}(c.hold) \mid AV \models c.\text{communicating} = \text{true}}$</td>
<td>${\text{enQ}(c.hold, gen(\tau)) \mid AV \models c.\text{communicating} = \text{false}} \cup {\text{enQ}(c.out, gen(\tau)) \mid AV \models c.\text{communicating} = \text{true}} \cup {\text{enQ}(c.out, \text{concatenate}(c.hold, gen(\tau)))} \cup (AV \models c.\text{communicating} = \text{false}) \land (AV' \models c.\text{communicating} = \text{true})}</td>
</tr>
<tr>
<td>$AV'$</td>
<td>$AV$</td>
<td>assign($AV$, eval($AV$, asn($\tau$)))</td>
</tr>
</tbody>
</table>

$en\_states(ss, \tau) \equiv src(\tau) \in CS$

$en\_events(ss, \tau) \equiv \text{trig}(\tau) \in IE$

$en\_cond(ss, \tau) \equiv AV \models cond(\tau)$

$macro\_semantics \equiv \text{stable}$

Table 12: Template Parameters for BoxTalk

$CS$ is the current state of each HTS. As there is no state hierarchy in BoxTalk, $CS$ is always a single state. $CS$ is updated only on execution of a transition $\tau$, when $CS'$ becomes the destination state of transition $\tau$.

Input signals can come potentially from multiple input queues. A macro-step starts by selecting one port and reading from its associated input queue. We use snapshot element $IE_a$ to record the nondeterministically selected port from which the next input is read. This selection starts the macro-step.

IE stores the current event. At the start of the macro-step, it holds the front element of the selected input queue ($IE_a$). At the end of each micro-step, IE holds the set of internal events generated during the macro step.

$I_a$ represents the HTS’s set of input queues. At the start of a macro-step, input events $I$ received from the environment (i.e., on calls to the HTS’s neighbouring HTS’s), are appended to the input queue.
associated with the signal’s associated call. For the input queue selected to be read from, the first element from its queue is removed. $I_a$ remains unchanged by the execution of transitions.

$O$ represents the set of output queues and hold queues. A macro-step starts with an empty set of outputs, and with a hold queue that may have contents. When transition $\tau$ executes, any events it generates are appended to the appropriate hold queue if the target call is not active, and are appended to the appropriate output queue otherwise. As soon as a call becomes established, the contents of its hold queue, together with any generated events, are copied to the call’s output queue, and the corresponding hold queue is emptied at the start of the next macro-step. These cases can also be distinguished by variable $communicating$ on each call: before and after transition executes, its value remains FALSE, TURE, or turns from FALSE to TURE, respectively.

$AV$ stores the current variable values. Variable values are updated at the end of each micro-step.

This concludes our discussion on mapping BoxTalk to HTS. Hereafter, mapping BoxTalk to a Promela model is discussed in terms of mapping a HTS model.
Chapter 5 Mapping BoxTalk to Promela Model

Starting from an explicated BoxTalk specification, a hand translation to an executable Promela model is implemented for both free and bound feature boxes. In this chapter, we introduce background on SPIN and Promela first, then talk about the structure of a generated Promela model in detail by walking through one free box and one bound box example.

5.1 Model Checker SPIN

In this subsection, we introduce our target model checker – SPIN. We will use SPIN to verify properties about BoxTalk specifications. After detailed discussions on its input language and property language, we explain that SPIN matches our needs.

Like other model checkers, SPIN requires

§ A formal model that describes the system specification or design. Promela (Process Meta-Language) is the input language for SPIN. It is an intuitive, program-like notation. Promela models are relatively abstract and focus on the design's process interactions, rather than on implementation and computation details.

§ The set of properties to be verified. SPIN accepts properties expressed as assertions, labels, never claims, and Linear Temporal Logic (LTL) formula.

Given an input model and property, SPIN exhaustive searches of all possible execution paths in the model, checking that the property holds in every execution state. If the property is violated in some execution state, an error trace leading to that state is displayed by the simulator.

SPIN is efficient for verifying models of distributed software systems. It has been used to detect design errors in various applications. Typically, the errors include deadlock, violated assertions, reachable bad states, and unreachable good states.

5.1.1 Promela

A Promela model is constructed from three basic building blocks:

§ Processes (asynchronous) -- used to model an object which has independent pre-defined behaviour, including reactions to messages

§ Data objects (structured data) – used to model C-like variables that define the data associated within processes and message channels.

§ Message channels (buffered and unbuffered) -- via which processes talk to each other

Code excerpts below show a basic Promela model that has no message channels, where one process named main is defined. active and proctype are keywords in Promela. Within process main, a C-like statement printf is defined. When executed, this process will print “hello world” on the screen.

active proctype main ( )
{  
  printf ("hello world\n")  
}

Promela offers a great number of constructs to build models. We discuss only those we adopted in our models. The interest reader can refer to [9] for a more thorough description of SPIN and Promela.

5.1.1.1 Processes

Processes are used to define behaviour. A process begins with keyword proctype. There must be at least one process in a model. There are two ways to instantiate processes in Promela:

§ Prefix the process declaration with keyword active. This causes the process to be instantiated in the initial system state.

§ Call the process from within the initial process init or any running process, using a predefined operator run.

The two approaches are shown below. The two code fragments will have the same output, but the rightmost code fragment creates an extra process.

-- Approach 1 --    -- Approach 2 --
active proctype main()  proctype main()  
{     {       printf("hello world\n")         printf("hello world\n")    }     }

init
{        run main()    }

We use the first approach in our models.

5.1.1.2 Data Objects

There are two levels of scope in Promela models: global and local to a process. Process types are always declared globally. Data objects and message channels can be declared either globally or locally.

Basic data types along with their value ranges are summarized in Table 11.

<table>
<thead>
<tr>
<th>Type</th>
<th>Typical Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>0, 1</td>
</tr>
<tr>
<td>bool</td>
<td>false, true</td>
</tr>
<tr>
<td>byte</td>
<td>0..255</td>
</tr>
<tr>
<td>chan</td>
<td>1..255</td>
</tr>
<tr>
<td>mtype</td>
<td>1..255</td>
</tr>
</tbody>
</table>
Table 13: Promela Data Types

Most data types have counterparts in the C programming language with two exceptions: chan and mtype. chan is used for declaring message-passing channels. Variables of type mtype are used for declaring user-defined symbolic values. mtype declarations are usually made at the start of the specification.

\[
\text{mtype} = \{\text{sig1, sig2, sig3, sig4}\} \\
\text{mtype} = \{\text{state1, state2, state3, state4}\}
\]

Multiple declarations of mtype are indistinguishable from a single mtype declaration. The above declarations, taken together, are equivalent to the following single declaration:

\[
\text{mtype} = \{\text{state1, state2, state3, state4, sig1, sig2, sig3, sig4}\}
\]

We use multiple declarations in our models for a clearer presentation.

array is a recognized data structure in Promela. It is used to organize multiple elements of the same type. It can be used with any data types listed in Table 13: Promela Data Types.

The array elements are distinguished from one another by their array index. The first element in an array has index zero. The number of elements is specified in the array declaration. The declaration

\[
\text{chan in} = [5] \text{of} \{\text{mtype}\}
\]
declares an array whose name is in, and whose capacity is five message channels.

Like the C programming language, Promela has a simple mechanism, typedef, for defining new structured types. We use typedef to define structured data types that correspond to constructs in our BoxTalk models:

\[
\text{typedef Transition} \{
\text{mtype dest;}
\text{chan in_chan;}
\text{chan out_chan;}
\text{bool en_flag = false;}
\}
\]

A structured type Transition is composed of an mtype, two chan, and one bool element. It groups information such as the transition’s destination state dest; the trigger signal stored at the head of message channel in_chan; one or more output channels out_chan for output signals, if any; and the evaluation of the transition’s guard predicate en_flag.

5.1.1.3 Message Channels

Processes exchange data through message channels. In the declaration

\[
\text{chan in} = [5] \text{of} \{\text{mtype}\}
\]
a channel named \textit{in} is declared, and it is capable of storing up to 5 messages of type \textit{mtype}. Note that this declaration is distinct from an array declaration. The declaration

\begin{verbatim}
chan in [5] = [5] of {mtype}
\end{verbatim}

declares an array of five channels, each with capacity of five messages. Operations \texttt{send} and \texttt{receive} on channels are expressed as

\begin{verbatim}
in ! sig1
in ? sig2
\end{verbatim}

respectively.

SPIN provides other more complex \texttt{send} and \texttt{receive} operations, like \texttt{sorted send} and \texttt{random receive}, which are not used in this work and thus are beyond the scope of our discussion.

Rendezvous communication is of special interest. It is realized by communication over a channel declared to have zero capacity; such a channel can pass but cannot store messages. We use rendezvous communication to enforce synchronous communication between processes.

Rendezvous communication is binary: only two processes, a sender and a receiver, can meet in a rendezvous handshake. Adopting rendezvous communication is a simplification and abstraction that allows us to better understand how processes execute and work together.

5.1.1.4 Rules of Execution

Semantics of execution define how processes execute, and define what constitutes an execution step. Any statement in Promela model is either \texttt{executable} or \texttt{blocked}. Executable means passable, runnable. Blocked means unexecutable. Executable statements include:

\begin{itemize}
  \item All print statements and assignments
  \item Any expression that evaluates to true
  \item Any send statement for which there is space in the message channel to write
  \item Any receive statement for which there are messages in the message channel to read
  \item A rendezvous communication, where both the sender and receiver process are ready to handshake
\end{itemize}

Blocked statements include

\begin{itemize}
  \item Any expression that evaluates to false
  \item Any send statement that writes to a full message channel
  \item Any receive statement that reads from an empty message channel
  \item A rendezvous communication, where either the sender or receiver is not ready to handshake
\end{itemize}

Promela has an interleaving semantics. At any point of execution, there is only one process executing, and the scheduling algorithm, which determines which process will execute and for how long, is nondeterministic. For example, the code fragment

\begin{verbatim}
active proctype P1( ){t1a; t1b}
\end{verbatim}
active proctype P2( ) {t2a; t2b}
defines two processes: \(P_1\) and \(P_2\), each of which consists of two statements. Each individual statement, like \(t1a\), executes atomically; such execution is called a **execution step**. The two statements induce three execution states in each individual process:

- State 0, before any statement executes
- State 1, after the first statement executes but before the second executes
- State 2, after the second statement executes

The execution state space of \(P_1\) and \(P_2\) together is the Cartesian product of the processes’ individual state space, from state \((0,0)\) to state \((2,2)\) as shown in Figure 16.

![Figure 16: Process Interleaving](image)

Suppose that the Promela model’s execution state is \((0,0)\). If both \(t1a\) and \(t2a\) are executable, then the system randomly picks one to run. If \(t1a\) blocks, process \(P_2\) takes control to execute \(t2a\). When the execution of \(t2a\) is complete and \(t1a\) becomes executable again, control may pass back to process \(P_1\), or process \(P_2\) may continue executing and run \(t2b\). Thus, there is a non-deterministic choice at each execution point.

### 5.1.1.5 Compound Statements

Promela has five types of compound statements, four of which we use in our models:

- Atomic sequence
- Selections
§ Repetitions
§ Escape sequence

An atomic sequence is used to group a sequence of statements so that all statements are executed indivisibly, like a single statement, without being interleaved with statements from other processes. The code fragment

```
active proctype P1( ){atomic(t1a; t1b)}
active proctype P2( ){t2a; t2b}
```

defines two processes: P1 and P2, where each process consists of two statements. Different from the previous example, an atomic wraps around statements t1a and t1b. Thus, process P1 semantically has only one transition. As shown in Figure 17, statements t1a and t1b execute in an un-interrupted manner, unless t1b blocks inside of the atomic step. In that case, the atomicity is lost, and the intermediate states, such as (1,0), might be visited. We use dotted arrows to denote the execution paths that are taken only when the atomicity is lost.

![Figure 17: Atomic Step](image)

For example, the code fragment

```
chan q = [0] of {bit}
active proctype X( ) {atomic { A; q!0; B }}
active proctype Y( ) {atomic { q?0 -> C }}
```

defines a rendezvous channel q and two process X and Y, which communicate through that channel. Inside each process, an atomic sequence is defined. As there is nothing in channel q for process Y to fetch in the system initial state, execution starts in process X with statement A. When the rendezvous handshake is executed, the atomic sequence in X is broken, and control passes to process Y, which
now starts the execution of statement C. When process Y terminates, control returns and execution resumes on B.

We use atomic sequence to wrap state transitions in our models, to reflect the idea that transitions between states are atomic. All statements that make up a state transition execute in sequence within one Promela step. As we just discussed, if blocked statements are encountered, exceptional behavior will take place.

The selection statement is used for selecting non-deterministically one option from a collection of conditional statements. Each conditional statement consists of a guard and an action. A conditional statement is enabled if its guard evaluates to true at the start of the selection statement. When the selection statement is reached, all of the guards on all of its branches are evaluated. If more than one evaluates to true, then some enabled conditional statement is non-deterministically selected and its actions are executed. If no guard evaluates to true, then the whole selection statement is blocked until one guard becomes true. Its syntax has the form:

```
if
  :: a option1;
  :: !a option2;
fi
```

The guards need not be mutually exclusive. We use the selection structure to implement non-determinism in our model.

A repetition structure (do statement) is used to repeatedly select from a collection of conditional statements. With respect to the choices, a do statement behaves in the same way as an if statement, except that the branches are repeatedly evaluated and one is chosen for execution, until a break statement is encountered. In that case, control is transferred to the end of the loop.

The following two code fragments have the same effect:

```
label1:
  if
    :: a --> option1;
    :: !a --> option2;
  fi
  goto label1;
```

We model environmental behaviour as one repetition structure that repeatedly selects for execution one of the environment’s possible actions.

The fourth compound statement, an escape sequence, is used to distinguish between high and low priority transitions within a single process. It has the syntax:

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

where P and E represent arbitrary Promela fragments. Before each execution of P, the executability of E is checked. P executes only if E is blocked. We use escape sequences to prioritize the behaviours of the environment process.
5.1.1.6 *inline* Functions

Promela supports *inline* functions, as a means of providing some of the structuring mechanisms of a traditional procedure call without introducing any overhead.

*inline* has a syntax of

```c
inline name( parameter){
  body
}
```

The Promela parser textually replaces each invocation of an *inline* function with the function’s body. Parameters are optional. If used, the parameters’ actual values textually replace the formal parameter’s placeholders in the function’s body.

5.2 The Promela Model

A feature box is represented as one or more SPIN processes. The environment is also modeled as a process that has rendezvous communication with the box process(es). Every port p on the box is associated with one *p_in* channel and one *p_out* zero-capacity channel. Both channels are unidirectional. *p_in* passes signals from the environment to the box. *p_out* passes signals from the box to the environment. As the box has multiple ports, array of *in* and *out* channels exist between box and environment.

![Figure 18: Architecture](image)

Free and bound feature boxes have different Promela models. A free feature box is represented as one process. A bound feature box is represented as one main process and one post-processing process, with a unidirectional internal channel that sends signals from the main process to the post-processing process. Signals passed through channel *internal* invoke the post-processing process which does clean-up work.
In accordance with this design, an array of in-channels and out-channels are declared globally. In addition, all variables that are accessible by multiple processes are declared globally. The *snapshot* variable is global in nature by its definition.

A typical Promela model contains three parts:

- type definitions and global variable declarations
- *inline* functions
- processes

Before we look into each part, we introduce a free feature box *Error Interface* (EI). Then, we will use EI’s Promela model to explain our work.

### 5.2.1 Free Error Interface

*Error Interface* (EI) box handles address errors that arise during routing. The router routes to EI if the target address is not a valid telephone number.

The EI feature simply accepts a call, sends signal *unknown* upstream, and tears down the call. Figure 20 shows its original specification:

![Figure 20: EI (Original specification)](image)

The associated explicated model is shown below:
We fully expand the macro $rcv(c)$ in the original specification into $boxport ? setup / c ! upack$. According to the DFC protocol, feature boxes send a $teardown$ signal immediately, following the $status$ signals to reject a call: the signal sequence $upack$, $unknown$ and $teardown$ transitions EI to the $terminating$ state. In the $terminating$ state, the box waits for an acknowledgement signal. In the mean time, it is still responsive to a crossover $teardown$ signal. The box transitions to $final$ state when its $teardown$ signal is acknowledged with a $downack$ signal.

The corresponding Promela model is attached as appendix A. We will walk through the code in the following discussions.

5.2.2 Type Definitions and Global Variable Declarations

At the top of a Promela model, signals, states and user-defined types that represent template semantics are defined.

**Signals**: $mtype$ of relevant signals. We also include a fake signal $other$, which represents signals that the feature box under study doesn’t respond to. The signals for EI are declared as

```plaintext
4 mtype = { teardown, downack, other, setup, upack, unknown };
```

**States**: $mtype$, whose values are the names of all the feature’s states. The states for EI are declared as

```plaintext
5 mtype = { initial, terminating, final };
```

The snapshot element $la$ from template semantics contains sets of input queues. In our Promela model, instead of declaring channels directly inside $la$, we organize the in-channels to the box process as an array $glob_ins$ and statically assign call variables an array index statically inside $la_type$:

```plaintext
36 chan glob_ins[2] = [0] of {mtype};
```

```plaintext
14 typedef Ia_type {
15 byte box_in = 0;
16 bool box_in_ready = true;
17 byte c_in = 1;
18 bool c_in_ready = false;
19 byte selected
20 };
```

In the above code, the channel array $glob_ins$ contains two zero-capacity channels which pass $mtype$ messages. The in-channel associated with $boxport$ and call variable $c$ are given index 0 and 1, respectively.
The boolean variables callVariable_ready is introduced to optimize the verifications, which indicate if the feature is in a state that it reads from a specified channel. For free feature boxes, the channel box_in is ready only in the box’s initial state. Thus, we deactivate this channel once the first setup signal has been received on this channel.

\[
\begin{align*}
184 & : ss.Ia.box_in_ready -> \\
185 & \quad ss.Ia.box_in_ready = false; \\
186 & \quad glob_ins[ss.Ia.box_in] ! setup;
\end{align*}
\]

The channel c_in is different. It is inactive in the box’s initial state, and becomes active once the call request is acknowledged, and the environment process can write to it.

\[
\begin{align*}
94 & : (n==0) -> \\
95 & \quad ss.Ia.c_in_ready = true; \\
96 & \quad t[0].out_chan!upack;
\end{align*}
\]

At times when more than one in-channel is active and has signals arriving, the indicator selected records the randomly selected channel to be read. This selection is done in the inline function reset(), which will be discussed in the next subsection.

Similarly, we organize the out-channels from the box process as an array glob_outs and statically assign call variables an array index inside O_type:

\[
\begin{align*}
37 & \quad chan glob_outs[2] = [0] of \{mtype\}; \\
22 & \quad typedef O_type \{ \\
23 & \quad \quad byte box_out = 0; \\
24 & \quad \quad byte c_out = 1; \\
25 & \quad \};
\end{align*}
\]

In the above code, the channel array glob_outs contains two zero capacity channels which pass mtype messages. The out-channel associated with boxport and call variable c are given index 0 and 1, respectively.

A template semantics Snapshot represents an observable execution state. We capture current state cs, in-channels la, and out-channels O with a user-type Snapshot in Promela:

\[
\begin{align*}
27 & \quad typedef SnapShot \{ \\
28 & \quad mtype cs; \\
29 & \quad Ia_type Ia; \\
30 & \quad O_type O \\
31 & \};
\end{align*}
\]

Thus, with declaration

\[
\begin{align*}
39 & \quad SnapShot ss;
\end{align*}
\]

channels within glob_ins and glob_outs can be accessed by variable names, rather than numbers. For example, glob_ins[ss.Ia.box_in] is equivalent to glob_ins[0]. Thus, call variables are actually indices into the channel array.

\[\text{3The out-channel of boxport is not actually used. It is declared so that call variable assignments are symmetric with in-channel assignments.}\]
Transition: user-type definition. A state transition in BoxTalk is triggered by the arrival of a signal on a port, and may cause signal generations on ports. The definition Transition groups the destination state, in-channel, out-channels, and a flag. Transition in EI is defined as

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

The mtype dest represents the transition’s destination state. The uni-directional message channels in_chan and out_chan passes signals into and out of the box, respectively. The boolean variable en_flag indicates whether this transition is selected to execute. All transitions of the feature box are organized in an array. They are declared and instantiated inside of the EI process:

```c
Transition t[3];
... 
//statically declare transitions
Transition t[0].dest = terminating;
Transition t[0].in_chan = glob_ins[ss.Ia.box_in];
Transition t[0].out_chan = glob_outs[ss.O.c_out];
Transition t[1].dest = final;
Transition t[1].in_chan = glob_ins[ss.Ia.c_in];
Transition t[2].dest = terminating;
Transition t[2].in_chan = glob_ins[ss.Ia.c_in];
Transition t[2].out_chan = glob_outs[ss.O.c_out];
```

Each transition has one deterministic destination state, one in_chan, and zero or more out_chans associated with call variables. In the case where there is no signal generated by a transition, the out_chan member is undefined. In the case where generated signals are sent on different channels, more than one out_chan is defined.

The value of en_flag is set in the inline function en_trans(), which is discussed in next subsection.

This concludes our discussion of type declarations, signals, states, type Transition, Ia_type, O_type and Snapshot. In summary, the static information of a BoxTalk feature is expressed by the declaration of signals and states, while dynamic information about a BoxTalk feature’s execution is expressed in Transition and snapshot elements: cs captures current states, la captures signals waiting to be read, O capture signals being output, and Snapshot combines cs, la and O.

5.2.3 inline Functions

Promela inline functions provide some of the structuring mechanism of a traditional procedure call, without introducing any overhead during the verification process. Compared to C-style macros, inline functions are preferable: when an error is reported in an inline function, the reported line number refers to the actual location of the false statement; with a macro, the reported line number refers to the point of invocation of the macro.

46
We use *inline* functions to simplify our process models, and to separate out code that implements the semantics of individual template-semantics parameters. Common *inline* functions used in our Promela models are *reset*, *en_events*, *en_cond*, *en_trans*, and *next_trans*.

**reset()** : In each *non-transient* state, the box randomly selects an input queue to be read from, and flags the specific channel been chosen. This behaviour is realized in the Promela model using the *if* construct. In addition, the boolean variables *rcvd_\_x* and *sent_\_x*, used for reasoning about certain properties, are reset to *false*. We will revisit them when talking about properties.

```promela
57 inline reset() {
58  rcvd_setup = false;
59  sent_upack = false;
60  rcvd_teardown = false;
61  sent_downack = false;
62  sent_teardown = false;
63  rcvd_downack = false;
64  sent_unknown = false;
65  if :: glob_ins[ss.Ia.box_in]\_sig -> ss.Ia.selected = ss.Ia.box_in;
66  :: glob_ins[ss.Ia.c_in]\_sig -> ss.Ia.selected = ss.Ia.c_in;
67  fi;
68 }
```

**en_events()** : The Box evaluates the triggers of each transition to determine if it is enabled. *en_events()* is true when the channel being selected matches the in-channel of the transition being evaluated. In the case where no input is expected (e.g., in a *transient* state), *en_events()* defaults to value *true*.

```promela
78 inline en_events(n){
79  glob_ins[ss.Ia.selected] == t[n].in_chan;
80 }
```

**en_cond()** : The box evaluates the guard condition of each transition to determine if it is enabled. *en_cond()* is true when the signal type and data parameters of the input signal match the transition’s trigger event signature. In the case where no input is expected (e.g., in a *transient* state), a guard condition is evaluated.

```promela
83 inline en_cond(n){
84  if 
85  ::(n==0) && sig == setup;
86  ::(n==1) && sig == downack;
87  ::(n==2) && sig == teardown;
88  fi;
89 }
```

The *inline en\_cond()* for EI says that transition 0 expects signal *setup*, transition 1 expects signal *downack*, and transition 2 expects signal *teardown*.
en_trans() : A transition executes only when the expected signal arrives on the specified input channel, as represented by the transition’s trigger event. A true evaluation of en_events() and en_cond() will set the flag of en_trans().

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

Unfortunately, an inline function cannot be used as the operand of an expression (as in the above code). Instead, we use two nested if constructs to achieve the same effect:

```c
112 inline en_trans(n){
113   if
114     :: en_events(n) ->
115     if
116       :: en_cond(n) -> t[n].en_flag = true;
117       :: else -> t[n].en_flag = false;
118     fi;
119     :: else -> t[n].en_flag = false;
120   fi;
121 }
```

next_trans() : realizes the execution of an enabled transition. This inline function transitions the execution to the destination state, updates variable values and writes signals on out-channels.

```c
92  inline next_trans(n){
93    if
94    ::(n==0) -> rcvd_setup = true;
95    ss.Ia.c_in_ready = true;
96    t[0].out_chan!upack;
97    sent_upack = true;
98    t[0].out_chan!unknown;
99    sent_unknown = true;
100   t[0].out_chan!teardown;
101   sent_teardown = true;
102   ss.cs = t[0].dest;
103    ::(n==1) -> rcvd_downack = true;
104    ss.cs = t[1].dest;
105    ::(n==2) -> rcvd_teardown = true;
106    t[2].out_chan!downack;
107    sent_downack = true;
108   fi;
109 }
```

5.2.4 Processes

The Promela model for the EI feature has two processes: the box process and environment process. Both processes are declared as active. That is, they are required to be running in the initial system state.
5.2.4.1 Feature Box Process

The feature box process reflects the structure of the corresponding BoxTalk HTS model. The process is decomposed to states. Each state is identified by a state-name-label. A state label leads to an *atomic* structure that is executed as an indivisible unit. Each *atomic* structure is organized into 4 phases: 1) randomly select a nonempty in-channel, and then read from this, 2) evaluate every possible transition out of this state, to set/unset flags, 3) Nondeterministically execute one enabled transition, and 4) transfer control to its destination state.

Let us take state *terminating* as an example:

```plaintext
157 terminating_state:
158 atomic{
159  reset();
160  en_trans(1);
161  en_trans(2);
162  if :: t[1].en_flag -> next_trans(1); goto final_state;
163  :: t[2].en_flag -> next_trans(2); goto terminating_state;
164  :: else -> goto terminating_state;
165 } fi;
```

*terminating_state* is a label to identify a unique control state within a process. Any statement or control-flow construct can be preceded by a label. A label name doesn’t need to be declared, but it has to be unique within the surrounding process. Moreover, no label name should be the same as any declared *mtype* variable. In EI, we have declared *terminating* as a *mtype* variable, so we use *terminating_state* as the label name to avoid confusion.

Because transitions are atomic, we use *atomic* to surround all of the statements that follow a state label. However, because there are rendezvous *send* statements in *next_trans()*, atomicity is lost and control passes from sender to receiver. Control can return back later and allow atomic execution of the rest of the sequence. Atomicity is lost also when a *goto* statement passes control out of the atomic sequence; but in the above case, the goto statement is the last to execute anyways.

Phrase 1 is implemented by *inline* function *reset()*, and phrase 2 is implemented by *inline* function *en_trans().* The *if* selection construct on line 165 makes a non-deterministic choice between enabled transitions. That is, if both *t[1].en_flag* and *t[2].en_flag* are true, either transition 1 (*next_trans(1); goto final_state*) or transition 2 (*next_trans(2), goto terminating_state*) executes. In the case where neither of them is enabled, an *else* statement is used to prevent potential deadlocks. It returns control back to the state label, in effect causing no transition to execute. The *else* statement executes only if no other statement within the *if* construct is executable.

We use *goto* statements to change the flow of control between state labels. For the purpose of changing states, a *goto* statement is always executable and has no other effect, and thus fits in our model perfectly.
A box process consists of the code fragments associated with each state in the HTS. We insert a dummy \textit{skip} action at the final state(s), so that at least one statement follows the label.

\begin{verbatim}
172 final_state:
173  skip;
\end{verbatim}

\subsection*{5.2.4.2 Environment Process}

The environment process models the expected context in which a feature box executes. The expected context is the router and its neighboring feature boxes. The router sends the box \textit{setup} signals and receives the box’s \textit{continue} signals. The neighboring feature boxes pass along the \textit{unknown} signal EI box generates, absorb \textit{upack} and \textit{downack} signals, respond to \textit{teardown} signals with \textit{downack} signals immediately, and passes signals to the EI box. The environment process mimics the environment of a feature and exercises all possible input that the feature might receive.

The environment process of EI is provided below:

\begin{verbatim}
181 active proctype env() {
182    end: do
183        :: ss.Ia.box_in_ready ->
184          ss.Ia.box_in_ready = false;
185        glob_ins[ss.Ia.box_in] ! setup;
186        :: ss.Ia.c_in_ready ->
187          if
188            :: glob_ins[ss.Ia.c_in] ! teardown;
189            :: glob_ins[ss.Ia.c_in] ! other;
190          fi;
191        od
192    unless {
193      if
194        :: glob_outs[ss.O.c_out] ? upack;
195        :: glob_outs[ss.O.c_out] ? unknown;
196        :: atomic { glob_outs[ss.O.c_out] ? teardown ->
197          glob_ins[ss.Ia.c_in] ! downack;
198        }
199      }
200        :: glob_outs[ss.O.c_out] ? downack;
201      fi;
202    }
203    goto end;
204 }
\end{verbatim}

A Promela \textit{do repetition} construct (opening at line 183 and closing at line 192) is used to allow the environment process to keep running until the box process stops at a valid end state. Inside the \textit{do} construct, we have two choices: put signals on the in-channel of \textit{boxport} or on the in-channel of port \textit{c} when their flags are set. The only allowable signal for \textit{boxport} is \textit{setup}, and the \textit{setup} signal is put on the \textit{boxport} channel only once. The environment might put signal \textit{teardown} or any other signal on the in-channel of \textit{c}.

The normal way to terminate the \textit{repetition} construct is with a \textit{break} statement. In our model, we marked the \textit{do repetition} construct with an end-state label \textit{end}, indicating that it is an acceptable termination point.
The Promela `unless` construct is used to prioritize statements. Its syntax has the form that two arbitrary Promela fragments are separated by `unless`. The first fragment is called the main sequence and the second fragment is called the escape sequence. The semantics of the statement are that the main sequence is executable only if the escape sequence is blocked.

Construct `unless` (at line 193) reflects other environment actions: receive outputs from the feature box, send signals in response to a previously received signal. In the EI example, the environment absorbs signals `upack`, `unknown` and `downack`, and responds to a received `teardown` signal with a `downack` signal.

Why do we prioritize receiving operations over sending operations? As we stated earlier in this chapter, an array of zero-capacity (rendezvous) channels are defined between the environment process and the box process. That is, the channel can pass, but cannot store messages. Message interactions via such rendezvous ports are by definition synchronous, which means that deadlock is possible.

In the case that both the box process and the environment process are trying to send a signal (that is, the box process is executing a `next_trans()` while the environment has determined that an in-channel is active and is trying to put a signal on it), both processes wait for the remote party to be ready to receive the signal; as a consequence, both processes are blocked.

By assigning higher priority to receiving operations, the environment is always ready to receive the box’s output signals. The environment process sends out signals only if there is no message to receive.

The `goto` statement at the bottom of the environment process ensures that the process executes repeatedly.

This concludes our discussions of translating a free feature box into a Promela model. Mapping a bound feature box is similar. We focus only on the differences in the next subsection.

### 5.2.5 Bound Transparent Box

A bound box is uniquely associated with a subscribing address. The **Bound Transparent Box (BTB)** is a simple feature that demonstrates the properties of bound boxes. It accepts a `setup` signal in every responsive state. Instead of transitioning to a `final` state and dying after a usage ends, a bound box returns to its initial state, ready to participate in the next call.

A Bound feature box may receive multiple `setup` signals in its lifetime. There are two cases to consider: (1) the feature needs to deal with `setup` signals that the feature cannot accept, and (2) the feature needs to finish dealing with a `teardown` signal, while accepting a new `setup` signal. In the first case, the box rejects the new call request by responding with signal sequence `upack`, `unavail`, and `teardown`, and continuing with its previous usage. In this case, who waits for the acknowledgement of `teardown`? In the second case, the box tears down all the old calls it was involved in and, accepts the new `setup` request, and continues with the new usage. In this case, who ensures that the old calls are completely torn down?
To solve these problems, we construct a post-processing machine, that runs in parallel with the feature box, to deal with the clean-up work. Thus, BTB maps to two processes; one main process and one post-processing process. The post-processing process fulfills the feature box’s obligation to adhere to the DFC protocol, while the main process implements the feature’s essential behaviour.

The post-processing process has the same structure as the main process: static information of transitions is initialized, followed by state-name-labels. The behaviours at a state are wrapped in an atomic sequence, appended to each state-name-label. In the idle state, the post-processing process reads from channel internal instead of from the environment. In all other states, it only reads from a specific set of in-channels to avoid consuming signals destined for the main process.

Like free boxes, signals and states are mapped to mtype. States and signals for the main process and the post-process are declared separately, for clearer representation:

```c
4   mtype = { teardown, downack, other, setup, upack, unavail };  
5   mtype = { post_process_t, post_process_f, post_process_s };  
6   mtype = { initial, orienting, connecting_f, connecting_s,  
7       transparent,  
8       deciding_1, deciding_2, error, receiving };  
9   mtype = { idle, t_work, s_wait_up, s_work, f_wait_up, f_work };  
```

In BTB, call variables take different values from time to time. Three variables are used in BTB: t, s and f. Call variable s refers to a call connecting the box to its subscriber. In contrast, call variable f refers to a call connecting the box to a far party. Call variable t is merely a place holder; s or f take its value later on.

As in free boxes, call variables are indices into arrays of channels. There are times when call variables take on new values, while the post-processing process finishes the termination of old calls. We implement separate in-channels for the main process and the post-processing process. We use old_callVariable to hold indices to the old channels for post-processing.

```c
20  typedef Ia_type {  
21     byte box_in = 0;  
22     byte old_t_in = 1;  
23     byte old_s_in = 2;  
24     byte old_f_in = 3;  
25     byte t_in = 4; //never used  
26     byte s_in = 5;  
27     byte f_in = 6;  
28     bool box_in_ready = true;  
29     bool old_t_in_ready = false;  
30     bool old_s_in_ready = false;  
31     bool old_f_in_ready = false;  
32     bool s_in_ready = false;  
33     bool f_in_ready = false;  
34     byte selected;  
35  };
```

In addition, we define t_pp() for bound boxes:

```c
124  inline reset_pp() {  
125     if
```
Symmetrically, we also separate the out-channels of the main process and the post-processing process. The hold queues are temporary signal holders for their out-channels. The capacity of the hold queues is our choice.

A bound box has a unique user-type IE_type, which is an internal channel to pass signals from the main process to the post-processing process. Channel internal has zero capacity. This approach ensures that the main process is blocked until the post-processing has completed processing one task, and returned back to its idle state.

BTB has a feature-specific predicate, t_from_sub, which is declared as a bool variable in our model. If t_from_sub is true, then call t is actually from the subscriber, and model assigns the value of call variable t to s, and uses call variable f to propagate this call to the downstream neighbour. The model does the opposite if the value of t_from_sub is false: it assigns the value of t to call variable f, and use call variable s to propagate this call to the subscriber. As call variable t stores different calls at different times, we use current_t_from_sub to record the new value and variable t_from_sub to record the previous value.

BTB uses boolean variables callVar_communicating to mark status of a call. Such a variable is true if an acknowledgement upack has been received. When a call is passed to the post-processing process, the value of callVar_communicating is copied to old_callVar_communicating, which will help the post-processing process to correctly terminate calls: either wait for a downack signal only, or wait for both upack and downack signals.

A few new inline functions have been abstracted in order to keep code simple and neat:

- setup_initial (b), where b is the predicate t_from_sub. This inline is called whenever a new setup is accepted and b is evaluated. It sets the flag on both subscriber and far-party sides, according
to value of $b$, and sets values of the corresponding $callVar_{communicating}$ variables appropriately.

```c
inline setup_initial(b){
    ss.Ia.s_in_ready = true;
    ss.Ia.f_in_ready = true;
    if (b) ->
        f_communicating = false;
        s_communicating = true;
    (b) ->
        s_communicating = false;
        f_communicating = true;
    fi
};
```

§ $teardown_{cleanup} (c)$, where $c$ is an integer variable whose value indicates a call. Once the box issues a $teardown$ signal, control transfers to the $post-processing$ process. $teardown_{cleanup} (c)$ activates the appropriate $in$ channel, to receive the corresponding $downack$ signal.

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

§ $dump (c1, c2)$, where $c1$ and $c2$ are two different channels. This $inline$ function dumps the contents of $c1$ into $c2$. Before a call is established, signals from upstream will be stored in a $hold$ queue. The contents of the $hold$ queue will be dumped to the downstream $out$ channel once an $upack$ signal is received.

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

Unlike the $inline$ functions we introduced previously, the above three don’t match any $Template Semantics$ definition. Their sole purpose is to avoid code duplication in our models.

The Promela model of BTB is presented as Appendix B.
Chapter 6 Case Studies

In this chapter, we go through the Promela models for free feature boxes *Receive Voice Mail* (RVM) and *Answer Confirm* (AC), and bound feature box *Black Phone Interface* (BPI). The feature boxes that make up the case studies were all of features provided by AT&T researcher Dr. Pamela Zave, as example features on which to evaluate our translation.

A usage generated by the DFC routing algorithm can be divided into source regions and target regions. A feature box is incorporated into a usage in a source region if the source address has subscribed to this feature. Similarly, a feature box is incorporated into a usage in a target region if the target address has subscribed to this feature. If a feature box is found only in source regions, like Speed Dialing, it is a source feature. If a feature box is found only in target regions, like Call Forwarding, it is a target feature.

### 6.1 Receive Voice Mail

*Receive Voice Mail* (RVM) is a target feature that records for its subscriber voice mail from callers.

![RVM Diagram](image)

As shown in Figure 22, the feature's essential functionality is triggered by the receipt of an *unavail* signal from downstream on call o. A voice server, which logs voice messages for its subscribers, is accessible through new call r.

RVM has four states: *initial*, *transparent*, *dialogue* and *termination* states. When RVM is involved in a usage but is not yet activated, it stays in the *transparent* state. When signal *unavail* arrives on call o, which indicates the unavailability of the callee, the box absorbs the *unavail* signal (i.e., it does not propagate the signal to the rest of the features in the usage), and issues the signal *avail* upstream instead (otherwise, if signal *unavail* reaches the caller, the caller will hang-up). Then, the box tears down the call to the callee and issues a new call r to the voice server. In the *dialogue* state, the caller and the voice server are signal linked, and a voice channel is opened between them. The box transitions to *termination* state when the caller finishes leaving a message and hangs up.

For simplicity, we assume ideal server behaviour. That is,
A connection to the voice server can always be established. The server will accept every setup request issued by the box.

The server always responds to setup and teardown signals with upack and downack signals, respectively.

When the server determines that the message recording is completed, it tears down call r.

The server does not issue any unexpected signals.

RVM embeds all of the behaviours that a Free Transparent Box has. The explicated model of RVM is shown in Figure 23.
Figure 23: RVM (Explicated Model)
First, the following expansions are similar to those for a Free Transparent Box (FTB):

- In between the initial state and the transparent state, an intermediate state connecting is introduced, in which the box waits for an upack signal on call o. Call variable i and o are not signal linked in the connecting state.
- State error is reached from the connecting state when the hold queue overflows. This allows finite analysis.
- State terminatingI and terminatingO are intermediate states within the two-phase teardown process, in which the box waits for a downack signal from the later terminated call.
- State final is reached from the terminating states when all allocated port have been freed (i.e., teardown or downack signals have been received on all allocated ports). Reaching this state indicates that a usage is over.
- State abandonConnection is introduced to model the situation in which the caller hangs up before the call is fully setup. In this state, the box has sent out signal setup and teardown in sequence, and expects to receive upack and downack signals in sequence in a DFC compliant environment.

In addition to the above, a number of feature-specific states are introduced in the explicated model:

State switching is an intermediate state, in which the box waits for the call to the callee to be fully torn down, and the call to the voice server to be fully set up. Upon reaching this state, the caller has been notified of the availability of the callee, but no call is connected to the caller. The box responds to a teardown signal from the caller by sending a downack signal and transitioning to the abandoning state, and throwing away any other signals from the caller. The box transitions to state connectingR on receiving a downack signal from the callee, and ignores all other signals except teardown signals. On receiving an upack signal from the voice server, the box transitions to state waitingOdown.

As the signals 1) downack from call o and 2) upack from call r may be received in any order, the two intermediate states connectingR and waitingOdown are used to record each possible ordering.

In the connectingR state, the box waits for an upack signal from call r, which indicates that the connection to the voice server has been established; then, the box transitions to state dialogue. In the meantime, the caller may hangup. The box responds to a teardown signal by sending a downack signal, propagating the teardown signal to call r, and transitioning to waitingRup state.

In waitingOdown state, the caller is connected to the voice server. However, the box won’t pass along any unexpected signal to server, because signal linkage is not established until state dialogue is reached. If signal teardown is received from the caller, the box transitions to state endOnR. As call o is not completely torn down, the box may receive signals, besides downack, from it. The box responds to a possible teardown signal and ignores all signals other than downack. Signal downack on call o causes the box to transition to state dialogue. We assume that the server connection is stable; that is, call r does not issue a teardown signal in this state.

State dialogue represents a connected usage between the caller and the voice mail service. The caller may leave a voice message on the server through a voice channel. The caller may hangup at any time in this state. Or, if the voice server determines that the message recording is completed, the
server may issue a **teardown** signal along call $r$. In response to the above actions, the box transitions to state **terminatingR** or **terminatingl**, respectively.

The remaining states **abandoning**, **endingOnR**, **waitingRup** and **terminatingR** are not observable by the user. In those states, the caller has hung up already, but the remain calls that need to be torn down.

In the state **abandoning**, the box waits for two events to happen: 1) a **downack** signal from call $o$, and 2) an **upack** signal from call $r$. These two signals may be received in any order. On receiving signal sequence **upack**, **downack**, the box transitions through state **endingOnR** to state **terminatingR**. On receiving signal sequence **downack**, **upack**, the box transitions through state **waitingRup** to state **terminatingR**. In the state **abandoning**, the box responds only to **teardown** and **downack** signals from call $o$, and ignores all other signals. The box will not receive any unexpected signals from the call connected to the voice server.

In the state **endingOnR**, the box waits for a **downack** signal from both calls $o$ and $r$. Again, if the **downack** signal from call $o$ is received first, then the box transitions to state **terminatingR**; if the **downack** signal from call $r$ is received first, then the box transitions to state **terminatingl**. In the state **endingOnR**, the box responds to a possible **teardown** signal from call $o$ and ignores all other signals.

In the state **waitingRup**, the box waits for an **upack** signal from call $r$ only. On receiving this signal, the box transitions to state **terminatingR**.

The box behaviour in state **terminatingR** is similar to that in states **terminatingl** or **terminatingO**. On receiving a **downack** signal from call $r$, the box transitions to the **final** state.

The Promela model of RVM is presented in Appendix C for reference.

### 6.2 Answer Confirm

The feature **Answer Confirm** (AC) ensures the success of a call by demanding that the callee press a button to confirm receiving the call. If the button is not pressed, AC will suppress the success outcome. The feature excludes the activation of voice mail as a successful call.

![Figure 24: Answer Confirm (Original Specification)](image)
Like all other feature boxes, AC enters in a usage when it receives a setup signal on its boxport. It continues the usage by propagating the setup signal and stays in state trying, waiting for outcome signals from downstream. If a signal indicating an unsuccessful outcome (unavail or unknown) is received on call o, then AC passes the outcome signal upstream and terminates. AC’s essential function activates when a status signal avail is received from downstream. To further confirm that the success outcome is the result of reaching the callee, and not his voice mail, the AC feature connects to a server via call r and transitions to state confirming. AC holds the previously success outcome signal received until a special confirmation is received by the server. Then, the avail status is propagated upstream, the connection to the AC server is terminated, and the AC box transitions to state transparent; its presence in the usage is not observable from then on. The signal confirm is a feature-specific pseudo signal. We introduce signal nonconfirm as the opposite (i.e., lack of confirmation) in the explicated model.
Figure 25: AC (Explicated Model)
The explicated model for feature box AC is shown in Figure 26. In between the initial and trying states, state connectingO is the intermediate state brought by the new call on o (expressed as ctu(i, o)). In this state, the new call o is not fully setup, and signals passed from upstream are stored in a hold queue until an upack signal is received. The feature box may go to a non-recovery error state if the hold queue overflows. If the caller hangs up in this state, the box transitions to state abandonConnection and waits for acknowledgement signals.

The state connectingR is the intermediate state introduced by the new call on r (expressed as new(r)). In state connectingR, call i and o are signal linked, so signals that do not trigger any action in the AC box are copied from one call to the other. We assume ideal server behaviour, which means that there is no hold queue constructed for call r, and the AC server is assumed to respond in a timely manner. That is, if the server is reached by call r and does not receive confirmation from the user within a reasonable time period, the server will give up and tear down call r.

In state confirming, call o and r are signal linked. That is, the user is able to respond to the server’s request (to press a certain button); signals passed from the caller side are ignored.

The state confirmed is the intermediate state that models BoxTalk macro end(r), in which the box waits for a downack signal on call r. The caller and the callee are signal linked again. When call r is fully torn down, the box transitions to the transparent state, in which the feature box is invisible.

The feature box handles three calls i, o and r at most. Termination of a feature box has three phases: 1) All calls need to be torn down, as represented by state endingAll, in which the signal teardown has been sent out on each call, and any received downack signal transitions the box to the next phase. 2) Two calls still need to be torn down, as represented by states endingInO, endingOnR, and endingInR, in which the box responds to teardown signals and ignores other signals. Any received downack signal causes the box to transition to the next phase. 3) One call still needs to be torn down, as represented by states terminatingI, terminatingO, and terminatingR, in which the box responds to crossover teardown signals and ignore others. The final downack signal causes the box to transition to its final state.

The Promela model for the explicated AC model is presented in Appendix D for reference.

In the environment process, we used a nested unless structure to enforce priorities:

- High – the environment receives a signal from the box. The signal is recorded if it is setup or teardown
- Medium – the environment sends signals to respond to a previously received setup or teardown signal
- Low – the environment sends other signals

### 6.3 Black Phone Interface

**Black Phone Interface** (BPI) is a bound box. It generates the different tones that the user of the phone hears. As an interface box, BPI translates between the DFC protocol and the telephone device.
Figure 26: BPI (Original Specification)
BPI involves only one call $c$. However, there is a great deal of signaling redundancy in the BPI BoxTalk specification. The box reacts not only to call level signals, but also to media channels and user actions. $c[v]$ represents a voice channel on call $c$. User actions include offhook, dialed and onhook. The names of the states indicate the tones that the user should be hearing. The path that passes through state ringing models the tones on an incoming call. The path that passes through state dialing models the tones on an outgoing call. Among states talking, silent, ringback, busytone and errortone, the box transitions from one state to another on specified signals:

- On accepted($c[v]$) or signal avail, the box transitions to the talking state.
- On nullified($c[v]$) or signal none, the box transitions to the silent state. That is, all previous tones generated are cancelled.
- On waiting($c[v]$), the box transitions to the ringback state.
- On rejected($c[v]$) or signal unavail, the box transitions to the busytone state.
- On signal unknown, the box transitions to the error state.

The box transitions to the disconnected state on receiving the gone($c$) signal. However, the box cannot go back to a tone-generation state from there; it transitions to the final state on receiving an onhook signal. The final state can be reached only on the user-action onhook.
Prior to the BPI feature, all of our feature models had only signal channels. The BPI feature involves a voice channel by which voice data may be transmitted in an internal call. BPI uses call v to model the voice channel and introduces pseudo signals accepted, waiting, rejected and nullified. Among these signals, accepted and avail, rejected and unavail, nullified and none have the same effect in most cases, but they are not equivalent. For example, on the transition from state ringing to talking, the signal accepted is the trigger while avail is the outcome. The waiting pseudo signal doesn’t have a counterpart in the DFC protocol.
Call a senses user input to the telephone device. The pseudo user-action signals are **offhook**, **dialed**, **onhook** and **other**.

Different from the BTB feature box, the original specification of BPI has a **final** state. That is, the user action **onhook** leads to the termination of the box program. In our explicated model, we let the box return to the **initial** state on receiving a **teardown** signal and we initiate the **post-processing** machine to do cleanup.

We introduce an intermediate state **connecting**, which is not a tone-generating state. It simply divides call setup into two phases, and the box waits for an **upack** signal in this state.

The Promela model for the explicated BPI feature is presented in Appendix E for reference.

### 6.4 Properties

#### 6.4.1 Properties to Prove

Our purpose for translating from BoxTalk to SPIN is to check that a feature specification conforms to the DFC protocol. Generally, we expect the following properties to hold in our Promela models:

1. A received **teardown** signal is eventually acknowledged by a **downack** signal sent in response (For calls connected to upstream, downstream, and/or server)

2. A received **setup** signal is eventually acknowledged by an **upack** signal sent in response (For call connected to upstream)

3. No port sends a status signal before sending an **upack** signal when the port is allocated

4. No port sends a status signal after sending a **teardown** signal, terminating the call

A **bound** box participates in multiple calls during its lifetime. For example, if the BTB feature progresses through states **initial**, **orienting**, **connecting_f**, and **deciding_1**, then the box will have received two **setup** signals on its **boxport**, one when in the **initial** state and the other when in the **connecting_f** state. We expect property 5 to hold in a BTB model:

5. Each received **setup** signal is eventually acknowledged by a corresponding **upack** signal sent in response

We sometimes need to reason about a property at a specific state of a specific process. For example, we expect the following property to hold in the feature box BTB:

6. If the **main** process is in **orienting_state** and the **post-processing** process is in a non-**idle_state**, then the **post-processing** process is tearing down the previous call. That means that the **main** process has received a new **setup** signal, and has advanced to **orienting** state.

Similarly, we expect the following property to hold in **bound** feature box BPI:

7. If the **main** process stays in its **initial** state and the **post-processing** process is in a non-**idle** state, then the **post-processing** process is terminating the current call.
6.4.2 Property Language

Properties written in English are prone to misunderstanding. In order to express properties precisely, we use the formal language Linear Temporal Logic (LTL) and never claims.

6.4.2.1 LTL

A formal language is a language that is defined by precise mathematical or machine processable formulas. LTL is built up from a set of proposition variables, logic connectives and temporal operators. LTL formulae are used to specify liveness properties. A liveness property is a property stating that something good will eventually happen. For example, statement If a user requests access to the critical section, he will eventually be granted access is a liveness property.

Given expressions $a$ and $b$, the following are LTL expressions:

\[
\begin{align*}
!a & \equiv \text{not } a \\
\land a \land b & \equiv a \text{ and } b \\
\lor a \lor b & \equiv a \text{ or } b \\
\rightarrow a \rightarrow b & \equiv a \text{ implies } b
\end{align*}
\]

A LTL formula is evaluated in states with respect to a sequence of future states. Given expressions $\Phi$ and $\Psi$, and an execution sequence $s_0, s_1, \ldots s_n$, the following are LTL expressions:

<table>
<thead>
<tr>
<th>LTL FORMULA</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X\Phi$</td>
<td>$X\Phi$ is true in $s_i$ if $\Phi$ holds in state $s_{i+1}$</td>
</tr>
<tr>
<td>$\Box \Phi$</td>
<td>$\Box \Phi$ is true in $s_i$ if $\Phi$ holds on the entire subsequent path $s_n, s_{i+1}, \ldots s_n$</td>
</tr>
<tr>
<td>$\Diamond \Phi$</td>
<td>$\Diamond \Phi$ is true in $s_i$ if $\Phi$ holds in some state $s_j (j \geq i)$</td>
</tr>
<tr>
<td>$\Psi \mathbf{U} \Phi$</td>
<td>$\Psi \mathbf{U} \Phi$ is true in $s_i$ if $\Phi$ holds at $s_j (j \geq i)$, and $\Psi$ holds at states $s_n, s_{i+1}, \ldots s_{j-1}$</td>
</tr>
</tbody>
</table>

Table 14: LTL

6.4.2.2 Never Claim

Among correctness claims in SPIN, a never claim is very expressive. It is commonly used to express finite or infinite system behaviour that should never happen. Most conveniently, never claims can be generated mechanically from LTL formula.

Syntactically, a never claim is presented as

\[
\text{never \{} \text{ statements } \}\]

We will discuss more in the following subsections.
6.4.2.3 List of Properties

The properties written in English in section 6.4.1 are formalized as:

1. □ (rcvd_teardown → ◊ sent_downack)
2. □ (rcvd_setup → ◊ sent_upack)
3. a. □ (rcvd_setup → (!sent_avail U sent_upack))
   b. □ (rcvd_setup → (!sent_unavail U sent_upack))
   c. □ (rcvd_setup → (!sent_unknown U sent_upack))
4. a. □ ((sent_teardown ∨ rcvd_teardown) → □ (!sent_avail))
   b. □ ((sent_teardown ∨ rcvd_teardown) → □ (!sent_unavail))
   c. □ ((sent_teardown ∨ rcvd_teardown) → □ (!sent_unknown))
5. □ ((rcvd_teardown ∧ (current_call = = num1)) → ◊ (sent_downack ∧ (current_call = = num1)))
6. never { (BTB@ orienting_state && ! (pp@ end_idle_state)) && ! (pp_call = = last_call) }
7. never { (BPI@ end_initial_state && ! (pp@ end_idle_state)) && ! (pp_call = = current_call) }

The translation from English to formulae for properties 1, 2, 3 and 4 is straightforward. In next subsection, we explain how the translations are made for properties 5, 6 and 7.

6.4.2.3 Translation from English to Formula

To check whether each teardown signal is followed by a corresponding downack signal, as we stated in property 5, we associate each usage with a unique numeric label. Properties are augmented with additional clauses that check these label values. Note that current_call is an integer variable, and num1 can be of any positive integer value.

In properties 6 and 7, we introduce symbol @ to relate processes to their current state, with syntax

processName @ stateLabel

Then, BTB@ orienting_state represents that the main process is at orienting_state. Variable pp_call holds the most recent call that the post-processing process is tearing down, variable current_call holds the most recent call that the main process is processing, and variable last_call holds the call that the main process has handed over to the post-processing process. Then, pp_call = = last_call captures the property the post-processing is tearing down the previous call.

To better understand how property 6 is transformed to a never claim, let us do some simplification first. Let

p represent BTB@ orienting_state
q represent pp@ end_idle_state
The English description is expressed directly as $(p \land ! q) \rightarrow r$. Thus,

\[(p \land ! q) \rightarrow r \]
\[
\iff ! (p \land ! q) \lor r
\iff ! ((p \land ! q) \land ! r)
\]

Taken from here, with $!$ represented by never and $p$, $q$, $r$ replaced, the property statement and never claim are equivalent. This also explains the transformation of property 7.

### 6.4.3 Embed correctness variables

As a last step before using SPIN to verify the properties, we need to embed correctness requirements into Promela model.

Correctness properties can refer only to elements (like variables, labels) in the Promela model, but the correctness properties that we want to prove refer to the sending or receipt of specific signals. To solve this dilemma, we define global boolean variables for any event that is referred to in a correctness property. The variables are initialized to false. They are set in a $next\_trans( )$ clause if the specified signal is received or sent within the transition, and then are unset in the $reset( )$ function invoked at the destination state.

For example, to verify property $\square (rcvd\_teardown \rightarrow \Diamond sent\_downack)$ in RVM, we introduce

```promela
bool rcvd_setup, sent_upack = false;
```

which are defined and initialized at line 56. They are set to true in $next\_trans(0)$ at line 168 and 171 respectively, and then set to false in $reset( )$ at line 79 and 80.

### 6.4.4 Results of Verification

Our Promela models are well structured to modularize the Promela representations of each template-semantics parameter (to make it easier to translate models written in other notations into Promela). As such, our models are not optimized for BoxTalk models or SPIN verification, and each run takes several minutes.

Among free boxes, the EI box generates the status signal unknown, we have proved that properties 1, 2, 3c and 4c hold; The RVM box generates status signal avail, we have proved that properties 1, 2, 3a and 4a hold; With the AC feature, box generates all three status signal avail, unavail, and unknown on different conditions, we have proved that properties 1, 2, 3a, 3b, 3c, 4a, 4b and 4c hold.

With bound boxes, we also proved that properties 5 and 6 hold for the BTB box, and property 7 holds for the BPI box.
Chapter 7 Conclusions

We summarize what work we did for model checking BoxTalk specifications in this chapter. To build a Promela model, which is model checked by the SPIN model checker, we performed the following steps on a BoxTalk specification:

§ Explicate the BoxTalk model
§ Define explicated BoxTalk models in terms of Template Semantics
§ Map BoxTalk-relevant Template Semantics to Promela model

7.1 Explicate BoxTalk Specification

There is a lot of implicit, unexpressed behaviour in a BoxTalk model. In order to model check the model, the implicit behaviour must be explicitly represented in a feature box model. Thus, explicating the BoxTalk specification is our first step. We start with expanding macros in BoxTalk, such as \texttt{new()} and \texttt{rcv()}, which are used to combine a sequence of read and write actions. Then, as setting up a call consists of two phases (1) sending a setup signal and (2) receiving an acknowledgement signal \texttt{upack}), we introduce hold queues and intermediate states to model this process. Before a call is fully established, the feature box waits in an intermediate state, any signals destined for an unestablished call are held locally in a \textit{hold queue}. Once the call is established, as evidenced by the \texttt{upack} signal, the held signals are forwarded to the call. Regarding feature termination, we distinguish between termination states and final states. In the termination states of feature boxes, all calls become inactive, but calls are not considered fully torn down until they have no ports allocated. That means that the box still receives signals from those ports and discards them, except for signals \texttt{teardown} and \texttt{downack}. The box responds with a \texttt{downack} signal when it receives a \texttt{teardown} signal. The receipt of a \texttt{downack} signal indicates the completion of the teardown process: the ports allocated are disassociated, a free box transitions to its final state, while a bound box transitions to its initial state. In addition, we introduce error states to keep the size of the model finite, which enables finite state analysis.

A bound feature box is explicated as two concurrent finite state machines: one \textit{main} machine and one \textit{post-processing} machine. The \textit{main} machine is ready to participate in a new usage as soon as the previous usage is terminating, and the \textit{post-processing} machine cleans up the terminating calls in the background by waiting for \texttt{upack} and \texttt{downack} signals.

7.2 Template Semantics Definition

\textit{Template Semantics} captures the semantics that are common among several model-based specification and design notations. By parameterizing notations' common execution semantics, each notation can be described in terms of parameters to the template semantics. Template semantics descriptions can also be the basis for tools that are configured using the semantic parameter values. In particular, we are interested in writing a translator from template-semantics models to SPIN, where
the translator is configured using semantic parameter values. We can configure such a translator with semantic parameters that reflect BoxTalk semantics. Thus, defining a BoxTalk specification in terms of Template Semantics integrates our work into a larger project.

This step includes two aspects: syntactic mapping and semantic mapping. We first showed how a BoxTalk model syntactically maps onto an HTS (the basic computation model of Template Semantics), and then presented the semantic mapping.

7.3 Promela Model

Starting from an explicated BoxTalk specification, a hand translation to an executable Promela model is implemented for both free and bound feature boxes.

A feature box is represented as one or more SPIN processes. A free feature box is represented as one process. A bound feature box is represented as one main process and one post-processing process, with a unidirectional internal channel that sends signals from the main process to the post-processing process. The environment is also modeled as a process that has rendezvous communication with the feature-box process(es).

In detail, the feature-box process reflects the structure of the corresponding BoxTalk HTS model. The process is decomposed to states. Each state is identified by a state-name-label. A state label leads to an atomic structure that is executed as an indivisible unit. The environment process mimics the environment of a feature and exercises all possible input that the feature might receive.

Inside processes, we used inline functions to simplify our process models, and to separate out code that implements the semantics of individual template-semantics parameters.

7.4 Case Studies and Results

We built Promela models for free feature boxes Error Interface (EI), Receive Voice Mail (RVM) and Answer Confirm (AC), bound feature boxes Bound Transparent Box (BTB) and Black Phone Interface (BPI) .

We used the SPIN model checker to verify that each of the features in the case studies adhered to the DFC protocol. In particular, we proved that the following properties hold:

1. A received teardown signal is eventually acknowledged by a downack signal sent in response (For calls connected to upstream, downstream, and/or server)
2. A received setup signal is eventually acknowledged by an upack signal sent in response (For call connected to upstream)
3. No port sends a status signal before sending an upack signal when the port is allocated
4. No port sends a status signal after sending a teardown signal, terminating the call
5. In bound boxes, each received setup signal is eventually acknowledged by a corresponding upack signal sent in response
6. In *bound* feature box BTB, if the *main* process is in *orienting state* and the *post-processing* process is in a non-*idle state*, then the *post-processing* process is tearing down the previous call. That means that the *main* process has received a new *setup* signal, and has advanced to *orienting* state.

7. In *bound* feature box BPI, if the *main* process stays is in its *initial* state and the *post-processing* process is in a non-*idle* state, then the *post-processing* process is terminating the current call.

Our Promela models are well structured to modularize the Promela representations of each template-semantics parameter (to make it easier to write Promela models for other notations in the future). As such, our models are not optimized for BoxTalk models or SPIN verification, and each run takes several minutes.
Appendix A

Promela model – Free Error Interface
/*==============================================================================*/
/* type definitions */

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

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

typedef Ia_type {
byte box_in = 0;
bool box_in_ready = true;
byte c_in = 1;
bool c_in_ready = false;
byte selected
};

typedef O_type {
byte box_out = 0;
byte c_out = 1;
};

typedef SnapShot {
mtype cs;
Ia_type Ia;
O_type O
};

/*==============================================================================*/
/* global variable declarations */

chan glob_ins[2] = [0] of {mtype};
chan glob_outs[2] = [0] of {mtype};
SnapShot ss;

/* global monitor variables */

bool rcvd_setup = false;
bool sent_upack = false;
bool rcvd_teardown = false;
bool sent_downack = false;
bool sent_teardown = false;
bool rcvd_downack = false;
bool sent_unknown = false;
/* ==============================================================*/
/* inline functions */
inline reset() {
    rcvd_setup = false;
    sent_upack = false;
    rcvd_teardown = false;
    sent_downack = false;
    sent_teardown = false;
    rcvd_downack = false;
    sent_unknown = false;
    if
        :: glob_ins[ss.la.box_in]?sig -> ss.la.selected = ss.la.box_in;
        :: glob_ins[ss.la.c_in]?sig -> ss.la.selected = ss.la.c_in;
    fi;
}

inline en_events(n){
    glob_ins[ss.la.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) ->
            rcvd_setup = true;
            ss.la.c_in_ready = true;
            t[0].outChan whispers upack;
            sent_upack = true;
            t[0].outChan whispers unknown;
            sent_unknown = true;
            t[0].outChan whispers teardown;
            ss.cs = t[0].dest;
        ::(n==1) ->
            rcvd_downack = true;
            ss.cs = t[1].dest;
        ::(n==2) ->
            rcvd_teardown = true;
            t[2].outChan whispers downack;
}

sent_downack = true;
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;
}

/*===============================================================*/
/* free error interface box process */
active proctype EI() {
  mtype sig;
  Transition t[3];
  ss.cs = initial;
  //statically declare transitions
t[0].dest = terminating;
t[0].in_chan = glob_ins[ss.Ia.box_in];
t[0].out_chan = glob_outs[ss.O.c_out];
t[1].dest = final;
t[1].in_chan = glob_ins[ss.Ia.c_in];
t[2].dest = terminating;
t[2].in_chan = glob_ins[ss.Ia.c_in];
t[2].out_chan = glob_outs[ss.O.c_out];
initial_state:
  atomic{
    reset();
en_trans(0);
  if :: t[0].en_flag -> next_trans(0); goto terminating_state;
:: else -> goto initial_state;
fi;
  }
terminating_state:
  atomic{
    reset();
en_trans(1);

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

final_state:
skip;
}

/*=================================================================* /
/* environment process */
/*=================================================================* /
active proctype env() {
end: do
:: ss.Ia.box_in_ready ->
  ss.Ia.box_in_ready = false;
  glob_ins[ss.Ia.box_in] ! setup;
:: ss.Ia.c_in_ready ->
  if
    :: glob_ins[ss.Ia.c_in] ! teardown;
    :: glob_ins[ss.Ia.c_in] ! other;
    fi;
  od
unless {
  if
    :: glob_outs[ss.O.c_out] ? upack;
    :: glob_outs[ss.O.c_out] ? unknown;
    :: atomic {
      :: glob_outs[ss.O.c_out] ? teardown ->
        glob_ins[ss.Ia.c_in] ! downack;
    }
    :: glob_outs[ss.O.c_out] ? downack;
    fi;
  }
  goto end;
}
Appendix B
Promela model – Bound Transparent Box
/** type definitions */

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

mtype = { post_process_t, post_process_f, post_process_s };

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

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

typedef Transition {
  mtype dest;
  chan in_chan;
  chan out_chan;
  chan out_chan2;
  chan out_chan3;
  bool en_flag = false;
};

typedef Ia_type {
  byte box_in = 0;
  byte old_t_in = 1;
  byte old_s_in = 2;
  byte old_f_in = 3;
  byte t_in = 4;//never used
  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 s_in_ready = false;
  bool f_in_ready = false;
  byte selected;
};

typedef O_type {
  byte box_out = 0;
  byte old_t_out = 1;//never used
  byte old_s_out = 2;
  byte old_f_out = 3;
  byte t_out = 4;
  byte s_out = 5;
  byte f_out = 6;
  chan s_hold = [1] of {mtype};
  chan f_hold = [1] of {mtype};
};

typedef IE_type {
  chan internal = [0] of {mtype};
};

typedef SnapShot {

mtype cs;
mtype cs_post_process;
la_type la;
O_type O;
IE_type IE;
);

//==============================================*/
/* global variable declarations */
/* chan glob_ins[7] = [0] of {mtype};
chan glob_outs[7] = [0] of {mtype};
SnapShot ss;
mtype sig;
mtype inter_sig;
bool s_communicating = true;
bool f_communicating = true;
bool old_s_communicating = true;
bool old_f_communicating = true;
bool t_from_subs = true;
bool current_t_from_subs = true;
Transition t[36];
/* global monitor variables */
bool rcvd_setup = false;
bool sent_upack = false;
bool sent_unavail = false;
bool sent_teardown = false;
bool s_rcvd_teardown = false;
bool s_sent_downack = false;
bool f_rcvd_teardown = false;
bool f_sent_downack = false;
byte last_call = 0;
byte current_call = 0;
byte pp_call = 0;
byte counter = 0;
/*==============================================*/
/* inline functions */
inline setup_initial(b){
ss.Ia.s_in_ready = true;
ss.Ia.f_in_ready = true;
if :: (b) -> f_communicating = false;
:: (!b) -> s_communicating = false;
f_communicating = true;
fi
};
inline teardown_cleanup(c) {
    if
        :: (c==0) -> ss.Ia.old_t_in_ready = true;
        :: (c==1) -> ss.Ia.s_in_ready = false;
        ss.Ia.old_s_in_ready = true;
        :: (c==2) -> ss.Ia.f_in_ready = false;
        ss.Ia.old_f_in_ready = true;
        old_s_communicating = s_communicating;
        old_f_communicating = f_communicating;
    fi
};

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

inline reset() {
    rcvd_setup = false;
    sent_upack = false;
    sent_unavail = false;
    sent_teardown = false;
    s_rcvd_teardown = false;
    s_sent_downack = false;
    f_rcvd_teardown = false;
    f_sent_downack = false;
    if
        :: glob_ins[ss.Ia.box_in]?sig -> ss.Ia.selected = ss.Ia.box_in;
        :: glob_ins[ss.Ia.s_in]?sig -> ss.Ia.selected = ss.Ia.s_in;
        :: glob_ins[ss.Ia.f_in]?sig -> ss.Ia.selected = ss.Ia.f_in;
    fi;
};

inline reset_pp() {
    if
        :: glob_ins[ss.Ia.old_s_in]?sig -> ss.Ia.selected = ss.Ia.old_s_in;
        :: glob_ins[ss.Ia.old_f_in]?sig -> ss.Ia.selected = ss.Ia.old_f_in;
        :: glob_ins[ss.Ia.old_t_in]?sig -> ss.Ia.selected = ss.Ia.old_t_in;
    fi;
};

inline en_events(n) {
    if
        ::(n==0) && ss.Ia.selected == ss.Ia.box_in;

class en
{
  ::(n==0) && sig == setup;
  ::(n==1) && current_t_from_subs;
  ::(n==2) && !current_t_from_subs;
  ::(n==3) && sig == upack;
  ::(n==4) && sig == upack;
  ::(n==5) && (sig == teardown) && !ss.la.old_f_in_ready;
  ::(n==6) && (sig == teardown);
  ::(n==7) && sig == setup;
  ::(n==8) && current_t_from_subs;
  ::(n==9) && !current_t_from_subs;
  ::(n==10) && (sig != teardown) && nfull(t[10].out_chan);
  ::(n==11) && sig == setup;
};

inline en_cond(n){
  if
  ::(n==0) && sig == setup;
  ::(n==1) && sig == setup;
  ::(n==2) && sig == setup;
  ::(n==3) && sig == setup;
  ::(n==4) && sig == setup;
  ::(n==5) && sig == setup;
  ::(n==6) && sig == setup;
  ::(n==7) && sig == setup;
  ::(n==8) && sig == setup;
  ::(n==9) && sig == setup;
  ::(n==10) && sig == setup;
  ::(n==11) && sig == setup;
}
::(n==12) && current_t_from_subs;
::(n==13) && !current_t_from_subs;
::(n==14) && (sig != teardown) && full(t[14].out_chan);
::(n==15) && sig != teardown && (ss.la.old_f_in_ready);
::(n==16) && sig == setup;
::(n==17) && current_t_from_subs;
::(n==18) && !current_t_from_subs;
::(n==19) && sig != teardown && full(t[19].out_chan);
::(n==20) && (sig == teardown) && ss.la.old_f_in_ready;
::(n==21) && sig == teardown;
::(n==22) && (inter_sig == post_process_s) && !old_s_communicating;
::(n==23) && sig == upack;
::(n==24) && sig == downack;
::(n==25) && (inter_sig == post_process_s) && old_s_communicating;
::(n==26) && sig == teardown;
::(n==27) && (inter_sig == post_process_f) && !old_f_communicating;
::(n==28) && sig == upack;
::(n==29) && sig == downack;
::(n==30) && (inter_sig == post_process_f) && old_f_communicating;
::(n==31) && sig == teardown;
::(n==32) && inter_sig == post_process_t;
::(n==33) && sig == downack;
fi;

inline next_trans(n){
  if
    ::(n==0) -> rcvd_setup = true;
    current_t_from_subs = t_from_subs;
    last_call = current_call;
    current_call = counter;
    t[0].out_chan!upack;
    sent_upack = true;
    ss.cs = t[0].dest;

    ::(n==1) -> setup_initial(current_t_from_subs);
    t[1].out_chan!setup;
    ss.cs = t[1].dest;

    ::(n==2) -> setup_initial(current_t_from_subs);
    t[2].out_chan!setup;
    ss.cs = t[2].dest;

    ::(n==3) -> dump(ss.O.f_hold, glob_outs[ss.O.f_out]);
    f_communicating = true;
    ss.cs = t[3].dest;

    ::(n==4) -> dump(ss.O.s_hold, glob_outs[ss.O.s_out]);
    s_communicating = true;
ss.cs = t[4].dest;

::(n==5) -> s_rcvd_teardown = true;
   t[5].out_chan!downack;
   s_sent_downack = true;
   pp_call = current_call;
   t[5].out_chan!teardown;
   ss.IE.internal!post_process_f;
   ss.cs = t[5].dest;

::(n==6) -> f_rcvd_teardown = true;
   t[6].out_chan!downack;
   f_sent_downack = true;
   pp_call = current_call;
   t[6].out_chan!teardown;
   ss.IE.internal!post_process_s;
   ss.cs = t[6].dest;

::(n==7) -> rcvd_setup = true;
   current_t_from_subs = t_from_subs;
   last_call = current_call;
   current_call = counter;
   t[7].out_chan!upack;
   sent_upack = true;
   ss.cs = t[7].dest;

::(n==8) -> pp_call = last_call;
   t[8].out_chan!teardown;
   ss.IE.internal!post_process_s;
   t[8].out_chan!teardown;
   ss.IE.internal!post_process_f;
   setup_initial(current_t_from_subs);
   t[8].out_chan!setup;
   ss.cs = t[8].dest;

::(n==9) -> t[9].out_chan!unavail;
   sent_unavail = true;
   t[9].out_chan!teardown;
   sent_teardown = true;
   ss.IE.internal!post_process_t;
   ss.cs = t[9].dest;

::(n==10) -> t[10].out_chan!sig;
   ss.cs = t[10].dest;

::(n==11) -> rcvd_setup = true;
   current_t_from_subs = t_from_subs;
   last_call = current_call;
   current_call = counter;
   t[11].out_chan!upack;
   sent_upack = true;
   ss.cs = t[11].dest;
::(n==12) -> pp.call = last_call;
        t[12].out_chan!teardown;
        ss.IE.internal!post_process_s;
        t[12].out_chan2!teardown;
        ss.IE.internal!post_process_f;
        setup_initial(current_t_from_subs);
        t[12].out_chan3!setup;
        ss.cs = t[12].dest;

::(n==13) -> t[13].out_chan!unavail;
        sent_unavail = true;
        t[13].out_chan!teardown;
        sent_teardown = true;
        ss.IE.internal!post_process_t;
        ss.cs = t[13].dest;

::(n==14) -> ss.la.s_in_ready = false;
        ss.la.f_in_ready = false;
        ss.cs = t[14].dest;

::(n==15) -> t[15].out_chan!sig;
        ss.cs = t[15].dest;

::(n==16) -> rcvd_setup = true;
        current_t_from_subs = t_from_subs;
        last_call = current_call;
        current_call = counter;
        t[16].out_chan!upack;
        sent_upack = true;
        ss.cs = t[16].dest;

::(n==17) -> pp_call = last_call;
        t[17].out_chan!teardown;
        ss.IE.internal!post_process_s;
        t[17].out_chan2!teardown;
        ss.IE.internal!post_process_f;
        setup_initial(current_t_from_subs);
        t[17].out_chan3!setup;
        ss.cs = t[17].dest;

::(n==18) -> t[18].out_chan!unavail;
        sent_unavail = true;
        t[18].out_chan!teardown;
        sent_teardown = true;
        ss.IE.internal!post_process_t;
        ss.cs = t[18].dest;

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

::(n==20) -> t[20].out_chan!downack;
        pp_call = current_call;
::(n==21) -> t[21].out_chan!downack;
pp_call = current_call;
t[21].out_chan2!teardown;
ss.IE.internal!post_process_f;
ss.cs = t[20].dest;
::(n==22) -> ss.cs_post_process = t[22].dest;
::(n==23) -> ss.cs_post_process = t[23].dest;
::(n==24) -> ss.Ia.old_s_in_ready = false;
ss.cs_post_process = t[24].dest;
::(n==25) -> ss.cs_post_process = t[25].dest;
::(n==26) -> t[26].out_chan!downack;
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.Ia.old_f_in_ready = false;
ss.cs_post_process = t[29].dest;
::(n==30) -> ss.cs_post_process = t[30].dest;
::(n==31) -> t[31].out_chan!downack;
ss.cs_post_process = t[31].dest;
::(n==32) -> ss.cs_post_process = t[32].dest;
::(n==33) -> ss.Ia.old_t_in_ready = false;
ss.cs_post_process = t[33].dest;
::(n==34) -> t[34].out_chan!sig;
ss.cs = t[34].dest;
::(n==35) -> t[35].out_chan!sig;
ss.cs = 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;
);
/*===============================================================*/
/* bound transparent box process */
active proctype BTB() {
    ss.cs = initial;
    //statically declare transitions
t[0].dest = orienting;
t[0].in_chan = glob_ins[ss.Ia.box_in];
t[0].out_chan = glob_outs[ss.O.t_out];

t[1].dest = connecting_f;
t[1].out_chan = glob_outs[ss.O.box_out];

t[2].dest = connecting_s;
t[2].out_chan = glob_outs[ss.O.box_out];

t[3].dest = transparent;
t[3].in_chan = glob_ins[ss.Ia.f_in];

t[4].dest = transparent;
t[4].in_chan = glob_ins[ss.Ia.s_in];

t[5].dest = initial;
t[5].in_chan = glob_ins[ss.Ia.s_in];
t[5].out_chan = glob_outs[ss.O.s_out];
t[5].out_chan2 = glob_outs[ss.O.f_out];

t[6].dest = initial;
t[6].in_chan = glob_ins[ss.Ia.f_in];
t[6].out_chan = glob_outs[ss.O.f_out];
t[6].out_chan2 = glob_outs[ss.O.s_out];

t[7].dest = receiving;
t[7].in_chan = glob_ins[ss.Ia.box_in];
t[7].out_chan = glob_outs[ss.O.t_out];

t[8].dest = connecting_f;
t[8].out_chan = glob_outs[ss.O.s_out];
t[8].outChan2 = glob_outs[ss.O.f_out];
t[8].outChan3 = glob_outs[ss.O.box_out];

t[9].dest = transparent;
t[9].out_chan = glob_outs[ss.O.t_out];

t[10].dest = connecting_f;
t[35].dest = transparent;
t[35].in_chan = glob_ins[ss.la.f_in];
t[35].out_chan = glob_outs[ss.O.s_out];

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;
    fi;
  }

connecting_f_state:
  atomic{
    reset();
    en_trans(3);
    en_trans(10);
    en_trans(11);
    en_trans(14);
    en_trans(20);
    
    if :: t[3].en_flag -> next_trans(3); goto transparent_state;
    :: t[10].en_flag -> next_trans(10); goto connecting_f_state;
    :: t[11].en_flag -> next_trans(11); goto deciding_1_state;
    :: t[14].en_flag -> next_trans(14); goto error_state;
    :: t[20].en_flag -> next_trans(20); goto end_initial_state;
    :: else -> goto connecting_f_state;
    fi;
  }

connecting_s_state:
  atomic{
    reset();
    en_trans(4);
    en_trans(15);
en_trans(16);
en_trans(19);
en_trans(21);

if :: t[4].en_flag -> next_trans(4); goto transparent_state;
:: t[15].en_flag -> next_trans(15); goto connecting_s_state;
:: t[16].en_flag -> next_trans(16); goto deciding_2_state;
:: t[19].en_flag -> next_trans(19); goto error_state;
:: t[21].en_flag -> next_trans(21); goto end_initial_state;
:: else -> goto connecting_s_state;
fi;

}

deciding_1_state:
atomic{
en_trans(12);
en_trans(13);

if :: t[12].en_flag -> next_trans(12); goto connecting_f_state;
:: t[13].en_flag -> next_trans(13); goto connecting_f_state;
fi;
}

deciding_2_state:
atomic{
en_trans(17);
en_trans(18);

if :: t[17].en_flag -> next_trans(17); goto connecting_f_state;
:: t[18].en_flag -> next_trans(18); goto connecting_s_state;
fi;
}

transparent_state:
atomic{
reset();
en_trans(5);
en_trans(6);
en_trans(7);
en_trans(34);
en_trans(35);

if :: t[5].en_flag -> next_trans(5); goto end_initial_state;
:: t[6].en_flag -> next_trans(6); goto end_initial_state;
:: t[7].en_flag -> next_trans(7); goto receiving_state;
:: t[34].en_flag -> next_trans(34); goto transparent_state;
:: t[35].en_flag -> next_trans(35); goto transparent_state;
:: else -> goto transparent_state;
receiving_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 transparent_state;
fi;
}
error_state:
skip;
}
/*=================================================================* /
/* post-processing process */
active proctype pp() {
byte inter_sig;
ss.cs_post_process = idle;
t[22].dest = s_wait_up;
t[23].dest = s_work;
t[23].in_chan = glob_ins[ss.Ia.old_s_in];
t[24].dest = idle;
t[24].in_chan = glob_ins[ss.Ia.old_s_in];
t[25].dest = s_work;
t[26].dest = s_work;
t[26].in_chan = glob_ins[ss.Ia.old_s_in];
t[26].out_chan = glob_outs[ss.O.old_s_out];
t[27].dest = f_wait_up;
t[28].dest = f_work;
t[28].in_chan = glob_ins[ss.Ia.old_f_in];
t[29].dest = idle;
t[29].in_chan = glob_ins[ss.Ia.old_f_in];
t[30].dest = f_work;
t[31].dest = f_work;
t[31].in_chan = glob_ins[ss.Ia.old_f_in];
t[31].out_chan = glob_outs[ss.O.old_f_out];
t[32].dest = t_work;
t[33].dest = idle;
t[33].in_chan = glob_ins[ss.Ia.old_t_in];

end_idle_state:
atomic{
    ss.IE.internal?inter_sig;
    en_trans(22);
    en_trans(25);
    en_trans(27);
    en_trans(30);
    en_trans(32);

    if :: t[22].en_flag -> next_trans(22); goto s_wait_up_state; :: t[25].en_flag -> next_trans(25); goto s_work_state; :: t[27].en_flag -> next_trans(27); goto f_wait_up_state; :: t[30].en_flag -> next_trans(30); goto f_work_state; :: t[32].en_flag -> next_trans(32); goto t_work_state; :: else -> goto end_idle_state; fi;
}

s_wait_up_state:
atomic{
    reset_pp();
    en_trans(23);

    if :: t[23].en_flag -> next_trans(23); goto s_work_state; :: else -> goto s_wait_up_state; fi;
}

s_work_state:
atomic{
    reset_pp();
    en_trans(24);
    en_trans(26);

    if :: t[24].en_flag -> next_trans(24); goto end_idle_state; :: t[26].en_flag -> next_trans(26); goto s_work_state; :: else -> goto s_work_state; fi;
}
f_wait_up_state:
atomic{
    reset_pp();
en_trans(27);
    if :: t[27].en_flag -> next_trans(27); goto f_work_state;
    :: else -> goto f_wait_up_state;
fi;
}
f_work_state:
atomic{
    reset_pp();
en_trans(29);
en_trans(31);
    if :: t[29].en_flag -> next_trans(29); goto end_idle_state;
    :: t[31].en_flag -> next_trans(31); goto f_work_state;
    :: else -> goto f_work_state;
fi;
}
t_work_state:
atomic{
    reset_pp();
en_trans(33);
    if :: t[33].en_flag -> next_trans(33); goto end_idle_state;
    :: else -> goto t_work_state;
fi;
}

/*=================================================================*/
/* environment process */
active proctype env() {
end:
do :: ss.la.box_in_ready && !ss.la.old_s_in_ready &&
    !ss.la.old_f_in_ready &&&ss.la.old_t_in_ready ->
    if :: atomic{ t_from_subs = true; counter = counter+1; glob_ins[ss.la.box_in]!setup; }
    :: atomic{ t_from_subs = false; counter = counter+1; glob_ins[ss.la.box_in]!setup; }
}
:: ss.la.s_in_ready && !ss.la.old_t_in_ready ->
if
:: glob_ins[ss.la.s_in] ! other;
:: glob_ins[ss.la.s_in] ! teardown;
fi;
:: ss.la.old_s_in_ready && !ss.la.old_t_in_ready ->
if
:: glob_ins[ss.la.old_s_in] ! downack;
:: glob_ins[ss.la.old_s_in] ! upack;
fi;
:: ss.la.f_in_ready && !ss.la.old_t_in_ready ->
if
:: glob_ins[ss.la.f_in] ! other;
:: glob_ins[ss.la.f_in] ! teardown;
fi;
:: ss.la.old_f_in_ready && !ss.la.old_t_in_ready ->
if
:: glob_ins[ss.la.old_f_in] ! downack;
:: glob_ins[ss.la.old_f_in] ! upack;
fi;
:: ss.la.old_t_in_ready -> glob_ins[ss.la.old_t_in] ! downack;
od
unless{
:: glob_outs[ss.O.box_out] ? setup ->
if
:: (current_t_from_subs) -> glob_ins[ss.la.f_in]!upack;
:: else -> glob_ins[ss.la.s_in]!upack;
fi
:: glob_outs[ss.O.t_out] ? upack;
:: glob_outs[ss.O.t_out] ? unavail;
:: atomic{ glob_outs[ss.O.t_out] ? teardown -> teardown_cleanup(0);} 
:: glob_outs[ss.O.s_out] ? other;
:: atomic{ glob_outs[ss.O.s_out] ? teardown -> teardown_cleanup(1);} 
:: glob_outs[ss.O.s_out] ? downack -> ss.la.s_in_ready = false;
:: glob_outs[ss.O.f_out] ? downack -> ss.la.f_in_ready = false;
:: glob_outs[ss.O.f_out] ? other;
:: atomic{ glob_outs[ss.O.f_out] ? teardown -> teardown_cleanup(2);} 
:: glob_outs[ss.O.old_s_out] ? downack;
:: glob_outs[ss.O.old_f_out] ? downack;
fi;
goto end;
};
Appendix C
Receive Voice Mail
/**=============================================================*/
/* type definitions */
/**
mtype = { teardown, downack, other, setup, upack, avail, unavail, unknown };  
mtype = { initial, transparent, connecting, abandonConnection,  
    terminatingO, terminatingI, final, error, switching,  
    waitingOdow, connectingR, abandoning, dialogue, endingOnR,  
    waitingRup, terminatingR };  
*/

typedef Transition {
    mtype dest;  
    chan in_chan;  
    chan out_chan;  
    chan out_chan2;  
    chan out_chan3;  
    bool en_flag = false;  
};  

typedef Ia_type {
    byte box_in = 0;  
    byte i_in = 1;  
    byte o_in = 2;  
    byte r_in = 3;  
    bool box_in_ready = true;  
    bool i_in_ready = false;  
    bool o_in_ready = false;  
    bool r_in_ready = false;  
    byte selected  
};  

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

typedef SnapShot {
    mtype cs;  
    Ia_type Ia;  
    O_type O  
};  

/*==============================================================*/
/* global variable declarations */
/**
 chan glob_ins[4] = [0] of {mtype};  
 chan glob_outs[4] = [0] of {mtype};  
 SnapShot ss;  
*/
/* global monitor variables */
bool rcvd_setup, sent_upack = false;
bool o_sent_setup, o_rcvd_upack = false;
bool r_sent_setup, r_rcvd_upack = false;
bool i_sent_teardown, i_rcvd_downack = false;
bool o_sent_teardown, o_rcvd_downack = false;
bool r_sent_teardown, r_rcvd_downack = false;
bool o_rcvd_teardown = false;
bool o_rcvd_status = false;
/* inline functions */
inline dump(c1, c2){
  byte aSig;
  do
    :: c1?aSig -> c2!aSig;
    :: empty(c1) -> break;
  od;
}
inline reset() {
  rcvd_setup = false;
sent_upack = false;
o_sent_setup = false;
o_rcvd_upack = false;
r_sent_setup = false;
r_rcvd_upack = false;
i_sent_teardown = false;
i_rcvd_downack = false;
o_sent_teardown = false;
o_rcvd_downack = false;
r_sent_teardown = false;
r_rcvd_downack = false;
o_rcvd_teardown = false;
o_rcvd_status = false;
if  
  :: glob_ins[ss.la.box_in]?sig -> ss.la.selected = ss.la.box_in;
  :: glob_ins[ss.la.o_in]?sig -> ss.la.selected = ss.la.o_in;
  :: glob_ins[ss.la.i_in]?sig -> ss.la.selected = ss.la.i_in;
  :: glob_ins[ss.la.r_in]?sig -> ss.la.selected = ss.la.r_in;
fi
}

inline en_events(n) {
    glob_ins[ss.Ia.selected] == t[n].in_chan;
};
inline en_cond(n) {
    if 
    ::(n==0) && (sig==setup);
    ::(n==1) && (sig==teardown) && nfull(t[1].out_chan);
    ::(n==2) && (sig==upack);
    ::(n==3) && (sig==teardown);
    ::(n==4) && (sig==teardown);
    ::(n==5) && (sig==teardown);
    ::(n==6) && (sig==teardown);
    ::(n==7) && (sig==teardown);
    ::(n==8) && (sig==teardown);
    ::(n==9) && (sig==teardown);
    ::(n==10) && (sig==teardown) && (sig!=downack);
    ::(n==11) && (sig==downack);
    ::(n==12) && (sig==teardown);
    ::(n==13) && (sig==teardown) && (sig==downack);
    ::(n==14) && (sig==teardown);
    ::(n==15) && (sig==teardown) && full(t[15].out_chan);
    ::(n==16) && (sig==unavail);
    ::(n==17) && (sig==avail);
    ::(n==18) && (sig==unknown);
    ::(n==19) && (sig==upack);
    ::(n==20) && (sig==downack);
    ::(n==21) && (sig==teardown);
    ::(n==22) && (sig==teardown) && (sig==downack);
    ::(n==23) && (sig==teardown);
    ::(n==24) && (sig==teardown);
    ::(n==25) && (sig==teardown);
    ::(n==26) && (sig==teardown);
    ::(n==27) && (sig==teardown) && (sig==downack);
    ::(n==28) && (sig==teardown);
    ::(n==29) && (sig==teardown);
    ::(n==30) && (sig==teardown);
    ::(n==31) && (sig==teardown);
    ::(n==32) && (sig==teardown);
    ::(n==33) && (sig==teardown);
    ::(n==34) && (sig==upack);
    ::(n==35) && (sig==teardown);
    ::(n==36) && (sig==teardown) && (sig==downack);
    ::(n==37) && (sig==teardown);
    ::(n==38) && (sig==teardown);
    ::(n==39) && (sig==teardown);
    ::(n==40) && (sig==downack);
    ::(n==41) && (sig==downack);
    ::(n==42) && (sig==teardown);
    ::(n==43) && (sig==teardown) && (sig==downack);
inline next_trans(n) {
    if
        ::(n==0) -> rcvd_setup = true;
        ss.la.i_in_ready = true;
        t[0].out_chan!upack;
        sent_upack = true;
        t[0].out_chan2!setup;
        o_sent_setup = true;
        ss.la.o_in_ready = true;
        ss.cs = t[0].dest;
        ::(n==1) -> t[1].out_chan!sig;
        ss.cs = t[1].dest;
        ::(n==2) -> o_rcvd_upack = true;
        dump(ss.O.o_hold, glob_outs[ss.O.o_out]);
        ss.cs = t[2].dest;
        ::(n==3) -> t[3].out_chan!sig;
        ss.cs = t[3].dest;
        ::(n==4) -> t[4].out_chan!sig;
        ss.cs = t[4].dest;
        ::(n==5) -> t[5].out_chan!downack;
        ss.la.i_in_ready = false;
        t[5].out_chan2!teardown;
        o_sent_teardown = true;
        ss.cs = t[5].dest;
        ::(n==6) -> o_rcvd_teardown = true;
        t[6].out_chan!downack;
        ss.la.o_in_ready = false;
        t[6].out_chan2!teardown;
        i_sent_teardown = true;
        ss.cs = t[6].dest;
        ::(n==7) -> t[7].out_chan!downack;
        ss.la.i_in_ready = false;
        t[7].out_chan2!teardown;
        o_sent_teardown = true;
        ss.cs = t[7].dest;
        ::(n==8) -> o_rcvd_upack = true;
        dump(ss.O.o_hold, glob_outs[ss.O.o_out]);
        ss.cs = t[8].dest;
        ::(n==9) -> o_rcvd_teardown = true;
        t[9].out_chan!downack;
        ss.cs = t[9].dest;
        ::(n==10) -> ss.cs = t[10].dest;
        ::(n==11) -> o_rcvd_downack = true;
        ss.la.o_in_ready = false;
        ss.cs = t[11].dest;
        ::(n==12) -> t[12].out_chan!downack;
        ss.cs = t[12].dest;
}
::(n==13) -> ss.cs = t[13].dest;
::(n==14) -> i_rcvd_downack = true;
ss.Ia.i_in_ready = false;
ss.cs = t[14].dest;
::(n==15) -> ss.cs = t[15].dest;
::(n==16) -> o_rcvd_status = true;
t[16].out_chan!avail;
t[16].out_chan2!teardown;
o_sent_teardown = true;
ss.cs = t[16].dest;
::(n==17) -> o_rcvd_status = true;
t[17].out_chan!avail;
ss.cs = t[17].dest;
::(n==18) -> o_rcvd_status = true;
t[18].out_chan!unknown;
ss.cs = t[18].dest;
::(n==19) -> r_rcvd_upack = true;
ss.cs = t[19].dest;
::(n==20) -> o_rcvd_downack = true;
ss.Ia.o_in_ready = false;
ss.cs = t[20].dest;
::(n==21) -> o_rcvd teardown = true;
t[21].out_chan!downack;
ss.cs = t[21].dest;
::(n==22) -> ss.cs = t[22].dest;
::(n==23) -> t[23].out_chan!downack;
ss.Ia.i_in_ready = false;
t[23].out_chan2!teardown;
r_sent_teardown = true;
ss.cs = t[23].dest;
::(n==24) -> ss.cs = t[24].dest;
::(n==25) -> o_rcvd_downack = true;
ss.Ia.o_in_ready = false;
ss.cs = t[25].dest;
::(n==26) -> o_rcvd teardown = true;
t[26].out_chan!downack;
ss.cs = t[26].dest;
::(n==27) -> ss.cs = t[27].dest;
::(n==28) -> t[28].out_chan!downack;
ss.Ia.i_in_ready = false;
t[28].out_chan2!teardown;
r_sent_teardown = true;
ss.cs = t[28].dest;
::(n==29) -> ss.cs = t[29].dest;
::(n==30) -> r_rcvd_upack = true;
ss.cs = t[30].dest;
::(n==31) -> t[31].out_chan!downack;
ss.Ia.i_in_ready = false;
t[31].out_chan2!teardown;
r_sent_teardown = true;
ss.cs = t[31].dest;
::(n==32) -> ss.cs = t[32].dest;
::(n==33) -> r_rcvd_upack = true;
ss.cs = t[33].dest;
::(n==34) -> o_rcvd_downack = true;
ss.cs = t[34].dest;
::(n==35) -> ss.cs = t[35].dest;
::(n==36) -> ss.cs = t[36].dest;
::(n==37) -> t[37].out_chan!downack;
ss.cs = t[37].dest;
::(n==38) -> t[38].out_chan!downack;
ss.cs = t[38].dest;
::(n==39) -> t[39].out_chan!sig;
ss.cs = t[39].dest;
::(n==40) -> r_rcvd_downack = true;
ss.cs = t[40].dest;
::(n==41) -> o_rcvd_downack = true;
ss.cs = t[41].dest;
::(n==42) -> o_rcvd_teardown = true;
::(n==43) -> ss.cs = t[43].dest;
::(n==44) -> r_rcvd_upack = true;
::(n==45) -> r_rcvd_downack = true;
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;
}
/*===============================================================*/
/* free receive voice mail box process */
active proctype RVM() {
  mtype sig;
  Transition t[46];
  ss.cs = initial;
  //statically declare transitions
  t[0].dest = connecting;
  t[0].in_chan = glob_ins[ss.Ia.box_in];
  t[0].out_chan = glob_outs[ss.O.i_out];
  t[0].out_chan2 = glob_outs[ss.O.box_out];
  t[1].dest = connecting;
  t[1].in_chan = glob_ins[ss.Ia.i_in];
  t[1].out_chan = ss.O.o_hold;
  t[2].dest = transparent;
  t[2].in_chan = glob_ins[ss.Ia.o_in];
  t[3].dest = transparent;
  t[3].in_chan = glob_ins[ss.Ia.i_in];
  t[3].out_chan = glob_outs[ss.O.o_out];
  t[4].dest = transparent;
  t[4].in_chan = glob_ins[ss.Ia.o_in];
  t[4].out_chan = glob_outs[ss.O.i_out];
  t[5].dest = terminatingO;
  t[5].in_chan = glob_ins[ss.Ia.i_in];
  t[5].out_chan = glob_outs[ss.O.i_out];
  t[5].out_chan2 = glob_outs[ss.O.o_out];
  t[6].dest = terminatingI;
  t[6].in_chan = glob_ins[ss.Ia.o_in];
  t[6].out_chan = glob_outs[ss.O.o_out];
  t[6].out_chan2 = glob_outs[ss.O.i_out];
  t[7].dest = abandonConnection;
  t[7].in_chan = glob_ins[ss.Ia.i_in];
  t[7].out_chan = glob_outs[ss.O.i_out];
  t[7].out_chan2 = glob_outs[ss.O.o_out];
  t[8].dest = terminatingO;
  t[8].in_chan = glob_ins[ss.Ia.o_in];
  t[9].dest = terminatingO;
  t[9].in_chan = glob_ins[ss.Ia.o_in];
  t[9].out_chan = glob_outs[ss.O.o_out]
}
t[10].dest = terminatingO;
  t[10].in_chan = glob_ins[ss.la.o_in];

  t[11].dest = final;
  t[11].in_chan = glob_ins[ss.la.o_in];

  t[12].dest = terminatingI;
  t[12].in_chan = glob_ins[ss.la.i_in];
  t[12].out_chan = glob_outs[ss.O.i_out];

  t[13].dest = terminatingI;
  t[13].in_chan = glob_ins[ss.la.i_in];

  t[14].dest = final;
  t[14].in_chan = glob_ins[ss.la.i_in];

  t[15].dest = error;
  t[15].in_chan = glob_ins[ss.la.i_in];
  t[15].out_chan = ss.O.o_hold;  

  t[16].dest = switching;
  t[16].in_chan = glob_ins[ss.la.o_in];
  t[16].out_chan = glob_outs[ss.O.i_out];
  t[16].out_chan2 = glob_outs[ss.O.o_out];
  t[16].out_chan3 = glob_outs[ss.O.box_out];

  t[17].dest = transparent;
  t[17].in_chan = glob_ins[ss.la.o_in];
  t[17].out_chan = glob_outs[ss.O.i_out];

  t[18].dest = transparent;
  t[18].in_chan = glob_ins[ss.la.o_in];
  t[18].out_chan = glob_outs[ss.O.i_out];

  t[19].dest = waitingOdown;
  t[19].in_chan = glob_ins[ss.la.r_in];

  t[20].dest = connectingR;
  t[20].in_chan = glob_ins[ss.la.o_in];

  t[21].dest = switching;
  t[21].in_chan = glob_ins[ss.la.o_in];
  t[21].out_chan = glob_outs[ss.O.o_out];

  t[22].dest = switching;
  t[22].in_chan = glob_ins[ss.la.o_in];

  t[23].dest = abandoning;
  t[23].in_chan = glob_ins[ss.la.i_in];
  t[23].out_chan = glob_outs[ss.O.i_out];
  t[23].out_chan2 = glob_outs[ss.O.r_out];

  t[24].dest = switching;
t[24].in_chan = glob_ins[ss.la.i_in];

427     t[25].dest = dialogue;
428     t[25].in_chan = glob_ins[ss.la.o_in];

430     t[26].dest = waitingOdown;
431     t[26].in_chan = glob_ins[ss.la.o_in];
432     t[26].out_chan = glob_outs[ss.O.o_out];

434     t[27].dest = waitingOdown;
435     t[27].in_chan = glob_ins[ss.la.o_in];

437     t[28].dest = endingOnR;
438     t[28].in_chan = glob_ins[ss.la.i_in];
439     t[28].out_chan = glob_outs[ss.O.i_out];
440     t[28].out_chan2 = glob_outs[ss.O.r_out];

442     t[29].dest = waitingOdown;
443     t[29].in_chan = glob_ins[ss.la.i_in];

445     t[30].dest = dialogue;
446     t[30].in_chan = glob_ins[ss.la.r_in];

448     t[31].dest = waitingRup;
449     t[31].in_chan = glob_ins[ss.la.i_in];
450     t[31].out_chan = glob_outs[ss.O.i_out];
451     t[31].out_chan2 = glob_outs[ss.O.r_out];

453     t[32].dest = connectingR;
454     t[32].in_chan = glob_ins[ss.la.i_in];

456     t[33].dest = endingOnR;
457     t[33].in_chan = glob_ins[ss.la.r_in];

459     t[34].dest = waitingRup;
460     t[34].in_chan = glob_ins[ss.la.o_in];

462     t[35].dest = abandoning;
463     t[35].in_chan = glob_ins[ss.la.o_in];
464     t[35].out_chan = glob_outs[ss.O.o_out];

466     t[36].dest = abandoning;
467     t[36].in_chan = glob_ins[ss.la.o_in];

469     t[37].dest = terminatingI;
470     t[37].in_chan = glob_ins[ss.la.r_in];
471     t[37].out_chan = glob_outs[ss.O.r_out];
472     t[37].out_chan2 = glob_outs[ss.O.i_out];

474     t[38].dest = terminatingR;
475     t[38].in_chan = glob_ins[ss.la.i_in];
476     t[38].out_chan = glob_outs[ss.O.i_out];
477     t[38].out_chan2 = glob_outs[ss.O.r_out];
initial_state:
    atomic{
        reset();

        en_trans(0);

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

connecting_state:
    atomic{
        reset();

        en_trans(1);
        en_trans(2);
        en_trans(7);
        en_trans(15);

        if
          :: t[1].en_flag -> next_trans(1); goto connecting_state;
          :: t[2].en_flag -> next_trans(2); goto transparent_state;
          :: t[7].en_flag -> next_trans(7); goto abandonConnection_state;
          :: t[15].en_flag -> next_trans(15); goto error_state;
          :: else -> goto connecting_state;
        fi;
transparent_state:
    atomic{
        reset();
        en_trans(3);
        en_trans(4);
        en_trans(5);
        en_trans(6);
        if :: t[3].en_flag -> next_trans(3); goto transparent_state;
        :: t[4].en_flag -> next_trans(4); goto transparent_state;
        :: t[5].en_flag -> next_trans(5); goto terminatingO_state;
        :: t[6].en_flag -> next_trans(6); goto terminatingI_state;
        :: t[16].en_flag -> next_trans(16); goto switching_state;
        :: t[17].en_flag -> next_trans(17); goto transparent_state;
        :: t[18].en_flag -> next_trans(18); goto transparent_state;
        :: else -> goto transparent_state;
        fi;
    }

abandonConnection_state:
    atomic{
        reset();
        en_trans(8);
        if :: t[8].en_flag -> next_trans(8); goto terminatingO_state;
        :: else -> goto abandonConnection_state;
        fi;
    }

terminatingO_state:
    atomic{
        reset();
        en_trans(9);
        en_trans(10);
        en_trans(11);
        if :: t[9].en_flag -> next_trans(9); goto terminatingO_state;
        :: t[10].en_flag -> next_trans(10); goto terminatingO_state;
        :: t[11].en_flag -> next_trans(11); goto final_state;
        :: else -> goto terminatingO_state;
        fi;
    }

terminatingI_state:
    atomic{
reset();

en_trans(12);
en_trans(13);
en_trans(14);

if :: t[12].en_flag -> next_trans(12); goto terminatingI_state;
en_trans(13); goto terminatingI_state;
en_trans(14); goto final_state;
:: else -> goto terminatingI_state;
fi;
}

switching_state:

atomic{
reset();
en_trans(19);
en_trans(20);
en_trans(21);
en_trans(22);
en_trans(23);
en_trans(24);

if :: t[19].en_flag -> next_trans(19); goto waitingOdown_state;
en_trans(20); goto connectingR_state;
en_trans(21); goto switching_state;
en_trans(22); goto switching_state;
en_trans(23); goto abandoning_state;
en_trans(24); goto switching_state;
:: else -> goto switching_state;
fi;
}

waitingOdown_state:

atomic{
reset();
en_trans(25);
en_trans(26);
en_trans(27);
en_trans(28);
en_trans(29);

if :: t[25].en_flag -> next_trans(25); goto dialogue_state;
en_trans(26); goto waitingOdown_state;
en_trans(27); goto waitingOdown_state;
en_trans(28); goto endingOnR_state;
en_trans(29); goto waitingOdown_state;
:: else -> goto waitingOdown_state;
fi;
}

connectingR_state:
atomic{
    reset();
    en_trans(30);
    en_trans(31);
    en_trans(32);
    if
        :: t[30].en_flag -> next_trans(30); goto dialogue_state;
        :: t[31].en_flag -> next_trans(31); goto waitingRup_state;
        :: t[32].en_flag -> next_trans(32); goto connectingR_state;
        :: else -> goto connectingR_state;
    fi;
}

abandoning_state:
atomic{
    reset();
    en_trans(33);
    en_trans(34);
    en_trans(35);
    en_trans(36);
    if
        :: t[33].en_flag -> next_trans(33); goto endingOnR_state;
        :: t[34].en_flag -> next_trans(34); goto waitingRup_state;
        :: t[35].en_flag -> next_trans(35); goto abandoning_state;
        :: t[36].en_flag -> next_trans(36); goto abandoning_state;
        :: else -> goto abandoning_state;
    fi;
}

dialogue_state:
atomic{
    reset();
    en_trans(37);
    en_trans(38);
    en_trans(39);
    if
        :: t[37].en_flag -> next_trans(37); goto terminatingl_state;
        :: t[38].en_flag -> next_trans(38); goto terminatingR_state;
        :: t[39].en_flag -> next_trans(39); goto dialogue_state;
        :: else -> goto dialogue_state;
    fi;
}


endingOnR_state:
 atomic{
    reset();
    en_trans(40);
    en_trans(41);
    en_trans(42);
    en_trans(43);
    if :: t[40].en_flag -> next_trans(40); goto terminatingO_state;
    :: t[41].en_flag -> next_trans(41); goto terminatingR_state;
    :: t[42].en_flag -> next_trans(42); goto endingOnR_state;
    :: t[43].en_flag -> next_trans(43); goto endingOnR_state;
    :: else -> goto endingOnR_state;
    fi;
}

waitingRup_state:
 atomic{
    reset();
    en_trans(44);
    if :: t[44].en_flag -> next_trans(44); goto terminatingR_state;
    :: else -> goto waitingRup_state;
    fi;
}

terminatingR_state:
 atomic{
    reset();
    en_trans(45);
    if :: t[45].en_flag -> next_trans(45); goto final_state;
    :: else -> goto terminatingR_state;
    fi;
}

error_state:
 final_state:
 progress:
    skip;
};

/*=================================================================*/
/* environment process */
active proctype env()

  mtype i_sig, o_sig, r_sig;

end: do

  :: ss.la.box_in_ready ->
    ss.la.box_in_ready = false;
    glob_ins[ss.la.box_in]!setup;
  :: ss.la.i_in_ready ->
    if
      :: glob_ins[ss.la.i_in]!teardown;
      :: glob_ins[ss.la.i_in]!other;
    fi unless{
      (i_sig == teardown) ->
      glob_ins[ss.la.i_in]!downack;
      i_sig = 0
    }

  :: ss.la.o_in_ready ->
    if
      :: glob_ins[ss.la.o_in]!teardown;
      :: glob_ins[ss.la.o_in]!other;
    fi unless{
      (o_sig == teardown) ->
      glob_ins[ss.la.o_in]!downack;
      o_sig = 0
    }

  :: ss.la.r_in_ready ->
    if
      :: ss.cs == dialogue -> glob_ins[ss.la.r_in]!teardown;
    fi unless{
      (r_sig == teardown) ->
      glob_ins[ss.la.r_in]!downack;
      r_sig = 0
    }

od

unless{
  if
    :: atomic{ glob_outs[ss.O.box_out]!setup ->
    if
      :: ss.cs == connecting ->
    if
      :: glob_ins[ss.la.o_in]!upack;
      glob_ins[ss.la.o_in]!avail;
    if
      :: glob_ins[ss.la.o_in]!upack;
      glob_ins[ss.la.o_in]!unavail;
    if
      :: glob_ins[ss.la.o_in]!upack;
      glob_ins[ss.la.o_in]!unknown;
    if
      :: ss.cs == connectingR ->
    fi;
    glob_ins[ss.la.r_in]!upack;
  fi;

}
:: glob_outs[ss.O.i_out]?upack;
:: glob_outs[ss.O.i_out]?downack;
:: glob_outs[ss.O.i_out]?teardown -> i_sig = teardown;
:: glob_outs[ss.O.i_out]?avail;
:: glob_outs[ss.O.i_out]?unavail;
:: glob_outs[ss.O.i_out]?unknown;
:: glob_outs[ss.O.i_out]?other;
:: glob_outs[ss.O.o_out]?downack;
:: glob_outs[ss.O.o_out]?teardown -> o_sig = teardown;
:: glob_outs[ss.O.o_out]?other;
:: glob_outs[ss.O.r_out]?downack;
:: glob_outs[ss.O.r_out]?teardown -> r_sig = teardown;
:: glob_outs[ss.O.r_out]?other;
:: glob_outs[ss.O.r_out]?other;
fi;

go to end;
Appendix D
Answer Confirm
/*=============================================================*/
/* type definitions */
/*=============================================================*/
mtype = { other, teardown, downack, setup, upack, avail, unavail,
unknown, confirm, nonconfirm }; 
mtype = { initial, connectingO, abandonConnection, trying,
connectingR, confirming, confirmed, transparent,
terminatingO, terminatingI, endingAll, endingOnR,
endingInR, endingInO, terminatingR, final, error } ;

typedef Transition { 
  mtype dest;
  chan in_chan;
  chan out_chan;
  chan out_chan2;
  chan out_chan3;
  bool en_flag = false;
};

typedef Ia_type { 
  byte box_in = 0;
  byte i_in = 1;
  byte o_in = 2;
  byte r_in = 3;
  bool box_in_ready = true;
  bool i_in_ready = false;
  bool o_in_ready = false;
  bool r_in_ready = false;
  byte selected
};

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

typedef SnapShot { 
  mtype cs;
  Ia_type Ia;
  O_type O
};

/*==============================================================*/
/* global variable declarations */

chan glob_ins[4] = [0] of {mtype};
chan glob_outs[4] = [0] of {mtype};
SnapShot ss;

117
/* inline functions */

```c
inline emptyChannel(c){
    byte aSig;
    do
        :: c?aSig;
        :: empty(c) -> break;
    od;
};

inline reset() {
    if :: glob_ins[ss.Ia.box_in]?sig -> ss.Ia.selected = ss.Ia.box_in;
    :: glob_ins[ss.Ia.o_in]?sig -> ss.Ia.selected = ss.Ia.o_in;
    :: glob_ins[ss.Ia.i_in]?sig -> ss.Ia.selected = ss.Ia.i_in;
    :: glob_ins[ss.Ia.r_in]?sig -> ss.Ia.selected = ss.Ia.r_in;
    fi;
};

inline en_events(n){
    glob_ins[ss.Ia.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(t[3].out_chan);
    ::(n==4) && (sig != teardown) && full(t[4].out_chan);
    ::(n==5) && sig == upack;
    ::(n==6) && sig == avail;
    ::(n==7) && sig == teardown;
    ::(n==8) && sig != teardown;
    ::(n==57) && sig == teardown;
    ::(n==58) && (sig!=teardown)&& (sig!=avail)&& (sig!=unavail)&& (sig!=unknown);
    ::(n==59) && sig == unavail;
    ::(n==60) && sig == unknown;
    ::(n==9) && sig == upack;
    ::(n==10) && sig == teardown;
    ::(n==11) && sig == teardown;
    ::(n==12) && sig != teardown;
    ::(n==13) && sig != teardown;
    ::(n==14) && sig == confirm;
    ::(n==15) && sig == nonconfirm;
    ::(n==16) && sig == teardown;
    ::(n==17) && sig == teardown;
    ::(n==18) && sig != teardown;
```

/* ==============================================================*/
::(n==19) && sig != teardown;
::(n==20) && sig == downack;
::(n==53) && sig == teardown;
::(n==54) && sig == teardown;
::(n==55) && sig != teardown;
::(n==56) && sig != teardown;
::(n==21) && sig == teardown;
::(n==22) && sig == teardown;
::(n==23) && sig != teardown;
::(n==24) && sig != teardown;
::(n==25) && sig == downack;
::(n==26) && sig == teardown;
::(n==27) && (sig != teardown) && (sig != downack);
::(n==28) && sig == downack;
::(n==29) && sig == teardown;
::(n==30) && (sig != teardown) && (sig != downack);
::(n==31) && sig == downack;
::(n==32) && sig == teardown;
::(n==33) && sig == downack;
::(n==34) && sig == teardown;
::(n==35) && (sig != teardown) && (sig == downack);
::(n==36) && sig == teardown;
::(n==37) && (sig != teardown) && (sig != downack);
::(n==38) && sig == downack;
::(n==39) && sig == downack;
::(n==40) && sig == teardown;
::(n==41) && (sig != teardown) && (sig != downack);
::(n==42) && sig == downack;
::(n==43) && sig == downack;
::(n==44) && sig == teardown;
::(n==45) && (sig != teardown) && (sig != downack);
::(n==46) && sig == downack;
::(n==47) && sig == downack;
::(n==48) && sig == teardown;
::(n==49) && (sig != teardown) && (sig != downack);
::(n==50) && sig == teardown;
::(n==51) && (sig != teardown) && (sig != downack);
::(n==52) && sig == downack;
fi;
}

inline next_trans(n){
if 
::(n==0) ->  ss.la.i_in_ready = true;
t[0].out_chan!upack;
t[0].out_chan2!setup;
ss.cs = t[0].dest;
::(n==1) ->  emptyChannel(ss.O.o_hold);
ss.cs = t[1].dest;
::(n==2) ->  t[2].out_chan!downack;
ss.la.i_in_ready = false;
}
t[2].out_chan2!teardown;
ss.cs = t[2].dest;
::(n==3) ->  t[3].out_chan!sig;
ss.cs = t[3].dest;
::(n==4) ->  ss.la.i_in_ready = false;
ss.la.o_in_ready = false;
ss.cs = t[4].dest;
::(n==5) ->  ss.cs = t[5].dest;
::(n==6) ->  t[6].out_chan!setup;
ss.la.r_in_ready = true;
ss.cs = t[6].dest;
::(n==7) ->  t[7].out_chan!downack;
ss.la.i_in_ready = false;
t[7].out_chan2!teardown;
::(n==8) ->  t[8].out_chan!sig;
ss.cs = t[8].dest;
::(n==57) ->  t[57].out_chan!downack;
ss.la.o_in_ready = false;
t[57].out_chan2!teardown;
ss.cs = t[57].dest;
::(n==58) ->  t[58].out_chan!sig;
ss.cs = t[58].dest;
::(n==59) ->  t[59].out_chan!unavail;
t[59].out_chan2!teardown;
ss.cs = t[59].dest;
::(n==60) ->  t[60].out_chan!unknown;
t[60].out_chan2!teardown;
ss.cs = t[60].dest;
::(n==9) ->  ss.cs = t[9].dest;
::(n==10) ->  t[10].out_chan!downack;
ss.la.i_in_ready = false;
t[10].out_chan2!teardown;
ss.cs = t[10].dest;
::(n==11) ->  t[11].out_chan!downack;
ss.la.o_in_ready = false;
t[11].out_chan2!teardown;
ss.cs = t[11].dest;
::(n==12) ->  t[12].out_chan!sig;
ss.cs = t[12].dest;
::(n==13) ->  t[13].out_chan!sig;
ss.cs = t[13].dest;
::(n==14) ->  t[14].out_chan!avail;
t[14].out_chan2!teardown;
ss.cs = t[14].dest;
::(n==15) ->  t[15].out_chan!unavail;
t[15].out_chan2!teardown;
ss.cs = t[15].dest;
::(n==16) ->  t[16].out_chan!downack;
ss.la.i_in_ready = false;
t[16].out_chan2!teardown;
t[16].out_chan3!teardown;
ss.cs = t[16].dest;
::(n==17) -> t[17].out_chan!downack;
ss.la.o_in_ready = false;
t[17].out_chan2!teardown;
t[17].out_chan3!teardown;
ss.cs = t[17].dest;
::(n==18) -> ss.cs = t[18].dest;
::(n==19) -> t[19].out_chan!sig;
ss.cs = t[19].dest;
::(n==20) -> ss.la.r_in_ready = false;
ss.cs = t[20].dest;
::(n==53) -> t[53].out_chan!downack;
ss.la.i_in_ready = false;
t[53].out_chan2!teardown;
ss.cs = t[53].dest;
::(n==54) -> t[54].out_chan!downack;
ss.la.o_in_ready = false;
t[54].out_chan2!teardown;
ss.cs = t[54].dest;
::(n==55) -> t[55].out_chan!sig;
ss.cs = t[55].dest;
::(n==56) -> t[56].out_chan!sig;
ss.cs = t[56].dest;
::(n==21) -> t[21].out_chan!downack;
t[21].out_chan2!teardown;
ss.la.i_in_ready = false;
::(n==22) -> t[22].out_chan!downack;
ss.la.o_in_ready = false;
t[22].out_chan2!teardown;
ss.cs = t[22].dest;
::(n==23) -> t[23].out_chan!sig;
ss.cs = t[23].dest;
::(n==24) -> t[24].out_chan!sig;
ss.cs = t[24].dest;
::(n==25) -> ss.la.o_in_ready = false;
ss.cs = t[25].dest;
::(n==26) -> t[26].out_chan!downack;
ss.cs = t[26].dest;
::(n==27) -> ss.cs = t[27].dest;
::(n==28) -> ss.la.i_in_ready = false;
ss.cs = t[28].dest;
::(n==29) -> t[29].out_chan!downack;
ss.cs = t[29].dest;
::(n==30) -> ss.cs = t[30].dest;
::(n==31) -> ss.la.i_in_ready = false;
ss.cs = t[31].dest;
::(n==32) -> ss.la.o_in_ready = false;
ss.cs = t[32].dest;
::(n==33) -> ss.la.r_in_ready = false;
ss.cs = t[33].dest;
::(n==34) -> t[34].out_chan!downack;
ss.cs = t[34].dest;
::(n==35) -> ss.cs = t[35].dest;
::(n==36) -> t[36].out_chan!downack;
ss.cs = t[36].dest;
::(n==37) -> ss.cs = t[37].dest;
::(n==38) -> ss.la.o_in_ready = false;
ss.cs = t[38].dest;
::(n==39) -> ss.la.r_in_ready = false;
ss.cs = t[39].dest;
::(n==40) -> t[40].out_chan!downack;
ss.cs = t[40].dest;
::(n==41) -> ss.cs = t[41].dest;
::(n==42) -> ss.la.i_in_ready = false;
ss.cs = t[42].dest;
::(n==43) -> ss.la.r_in_ready = false;
ss.cs = t[43].dest;
::(n==44) -> t[44].out_chan!downack;
ss.cs = t[44].dest;
::(n==45) -> ss.cs = t[45].dest;
::(n==46) -> ss.la.i_in_ready = false;
ss.cs = t[46].dest;
::(n==47) -> ss.la.o_in_ready = false;
ss.cs = t[47].dest;
::(n==48) -> t[48].out_chan!downack;
ss.cs = t[48].dest;
::(n==49) -> ss.cs = t[49].dest;
::(n==50) -> t[50].out_chan!downack;
ss.cs = t[50].dest;
::(n==51) -> ss.cs = t[51].dest;
::(n==52) -> ss.la.r_in_ready = false;
ss.cs = t[52].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;
}

/*===============================================================*/
/* free answer confirm box process */
active proctype AC() {

mtype sig;
Transition t[61];
ss.cs = initial;

//statically declare transitions
t[0].dest = connectingO;
t[0].in_chan = glob_ins[ss.Ia.box_in];
t[0].out_chan = glob_outs[ss.O.i_out];
t[0].out_chan2 = glob_outs[ss.O.box_out];

t[1].dest = trying;
t[1].in_chan = glob_ins[ss.Ia.o_in];
t[1].out_chan = ss.O.o_hold;

t[2].dest = abandonConnection;
t[2].in_chan = glob_ins[ss.Ia.i_in];
t[2].out_chan = glob_outs[ss.O.i_out];
t[2].out_chan2 = glob_outs[ss.O.o_out];

t[3].dest = connectingO;
t[3].in_chan = glob_ins[ss.Ia.i_in];
t[3].out_chan = ss.O.o_hold;

t[4].dest = error;
t[4].in_chan = glob_ins[ss.Ia.i_in];
t[4].out_chan = ss.O.o_hold;

t[5].dest = terminatingO;
t[5].in_chan = glob_ins[ss.Ia.o_in];

t[6].dest = connectingR;
t[6].in_chan = glob_ins[ss.Ia.o_in];
t[6].out_chan = glob_outs[ss.O.box_out];

t[7].dest = terminatingO;
t[7].in_chan = glob_ins[ss.Ia.i_in];
t[7].out_chan = glob_outs[ss.O.i_out];
t[7].out_chan2 = glob_outs[ss.O.o_out];

t[8].dest = trying;
t[8].in_chan = glob_ins[ss.Ia.i_in];
t[8].out_chan = glob_outs[ss.O.o_out];

t[57].dest = terminatingI;
t[57].in_chan = glob_ins[ss.Ia.o_in];
t[57].out_chan = glob_outs[ss.O.o_out];
t[57].out_chan2 = glob_outs[ss.O.i_out];

t[58].dest = trying;
t[58].in_chan = glob_ins[ss.Ia.o_in];
t[58].out_chan = glob_outs[ss.O.i_out];
t[59].dest = endingInO;
  t[59].in_chan = glob_ins[ss.Ia.o_in];
  t[59].out_chan = glob_outs[ss.O.i_out];
  t[59].out_chan2 = glob_outs[ss.O.o_out];
  t[60].dest = endingInO;
  t[60].in_chan = glob_ins[ss.Ia.o_in];
  t[60].out_chan = glob_outs[ss.O.i_out];
  t[60].out_chan2 = glob_outs[ss.O.o_out];

  t[9].dest = confirming;
  t[9].in_chan = glob_ins[ss.Ia.r_in];

  t[10].dest = terminatingO;
  t[10].in_chan = glob_ins[ss.Ia.i_in];
  t[10].out_chan = glob_outs[ss.O.i_out];
  t[10].out_chan2 = glob_outs[ss.O.o_out];

  t[11].dest = terminatingI;
  t[11].in_chan = glob_ins[ss.Ia.o_in];
  t[11].out_chan = glob_outs[ss.O.o_out];
  t[11].out_chan2 = glob_outs[ss.O.i_out];

  t[12].dest = connectingR;
  t[12].in_chan = glob_ins[ss.Ia.i_in];
  t[12].out_chan = glob_outs[ss.O.o_out];

  t[13].dest = connectingR;
  t[13].in_chan = glob_ins[ss.Ia.o_in];
  t[13].out_chan = glob_outs[ss.O.i_out];

  t[14].dest = confirmed;
  t[14].in_chan = glob_ins[ss.Ia.r_in];
  t[14].out_chan = glob_outs[ss.O.i_out];
  t[14].out_chan2 = glob_outs[ss.O.o_out];
  t[15].dest = endingAll;
  t[15].in_chan = glob_ins[ss.Ia.r_in];
  t[15].out_chan = glob_outs[ss.O.i_out];
  t[15].out_chan2 = glob_outs[ss.O.o_out];
  t[15].out_chan3 = glob_outs[ss.O.r_out];

  t[16].dest = endingOnR;
  t[16].in_chan = glob_ins[ss.Ia.i_in];
  t[16].out_chan = glob_outs[ss.O.i_out];
  t[16].out_chan2 = glob_outs[ss.O.o_out];
  t[16].out_chan3 = glob_outs[ss.O.r_out];

  t[17].dest = endingInR;
  t[17].in_chan = glob_ins[ss.Ia.o_in];
  t[17].out_chan = glob_outs[ss.O.o_out];
  t[17].out_chan2 = glob_outs[ss.O.i_out];
  t[17].out_chan3 = glob_outs[ss.O.r_out];
t[18].dest = confirming;
t[18].in_chan = glob_ins[ss.Ia.i_in];

t[19].dest = confirming;
t[19].in_chan = glob_ins[ss.Ia.o_in];
t[19].out_chan = glob_outs[ss.O.r_out];

t[20].dest = transparent;
t[20].in_chan = glob_ins[ss.Ia.r_in];

t[53].dest = endingOnR;
t[53].in_chan = glob_ins[ss.Ia.i_in];
t[53].out_chan = glob_outs[ss.O.i_out];
t[53].out_chan2 = glob_outs[ss.O.o_out];

t[54].dest = endingInR;
t[54].in_chan = glob_ins[ss.Ia.o_in];
t[54].out_chan = glob_outs[ss.O.o_out];
t[54].out_chan2 = glob_outs[ss.O.i_out];

t[55].dest = confirmed;
t[55].in_chan = glob_ins[ss.Ia.i_in];
t[55].out_chan = glob_outs[ss.O.o_out];

t[56].dest = confirmed;
t[56].in_chan = glob_ins[ss.Ia.o_in];
t[56].out_chan = glob_outs[ss.O.i_out];

t[21].dest = terminatingO;
t[21].in_chan = glob_ins[ss.Ia.i_in];
t[21].out_chan = glob_outs[ss.O.i_out];
t[21].out_chan2 = glob_outs[ss.O.o_out];

t[22].dest = terminatingI;
t[22].in_chan = glob_ins[ss.Ia.o_in];
t[22].out_chan = glob_outs[ss.O.o_out];
t[22].out_chan2 = glob_outs[ss.O.i_out];

t[23].dest = transparent;
t[23].in_chan = glob_ins[ss.Ia.i_in];
t[23].out_chan = glob_outs[ss.O.o_out];

t[24].dest = transparent;
t[24].in_chan = glob_ins[ss.Ia.o_in];
t[24].out_chan = glob_outs[ss.O.i_out];

t[25].dest = final;
t[25].in_chan = glob_ins[ss.Ia.i_in];

t[26].dest = terminatingO;
t[26].in_chan = glob_ins[ss.Ia.o_in];
t[26].out_chan = glob_outs[ss.O.o_out];
t[27].dest = terminatingO;
t[27].in_chan = glob_ins[ss.la.o_in];

t[28].dest = final;
t[28].in_chan = glob_ins[ss.la.i_in];

t[29].dest = terminatingI;
t[29].in_chan = glob_ins[ss.la.i_in];
t[29].out_chan = glob_outs[ss.O.i_out];

t[30].dest = terminatingI;
t[30].in_chan = glob_ins[ss.la.i_in];

t[31].dest = endingOnR;
t[31].in_chan = glob_ins[ss.la.i_in];

t[32].dest = endingInR;
t[32].in_chan = glob_ins[ss.la.o_in];

t[33].dest = endingInO;
t[33].in_chan = glob_ins[ss.la.r_in];

t[34].dest = endingAll;
t[34].in_chan = glob_ins[ss.la.i_in];
t[34].out_chan = glob_outs[ss.O.i_out];

t[35].dest = endingAll;
t[35].in_chan = glob_ins[ss.la.i_in];

t[36].dest = endingAll;
t[36].in_chan = glob_ins[ss.la.o_in];
t[36].out_chan = glob_outs[ss.O.o_out];

t[37].dest = endingAll;
t[37].inChan = glob_ins[ss.la.o_in];

t[38].dest = terminatingR;
t[38].inChan = glob_ins[ss.la.o_in];

t[39].dest = terminatingO;
t[39].inChan = glob_ins[ss.la.r_in];

t[40].dest = endingOnR;
t[40].inChan = glob_ins[ss.la.o_in];
t[40].outChan = glob_outs[ss.O.o_out];

t[41].dest = endingOnR;
t[41].inChan = glob_ins[ss.la.o_in];

t[42].dest = terminatingR;
t[42].inChan = glob_ins[ss.la.i_in];
t[43].dest = terminatingI;
t[43].in_chan = glob_ins[ss.la.r_in];
t[44].dest = endingInR;
t[44].in_chan = glob_ins[ss.la.i_in];
t[44].out_chan = glob_outs[ss.O.i_out];
t[45].dest = endingInR;
t[45].in_chan = glob_ins[ss.la.i_in];
t[46].dest = terminatingO;
t[46].in_chan = glob_ins[ss.la.i_in];
t[47].dest = terminatingI;
t[47].in_chan = glob_ins[ss.la.o_in];
t[48].dest = endingInO;
t[48].in_chan = glob_ins[ss.la.i_in];
t[48].out_chan = glob_ins[ss.O.i_out];
t[49].dest = endingInO;
t[49].in_chan = glob_ins[ss.la.i_in];
t[50].dest = endingInO;
t[50].in_chan = glob_ins[ss.la.o_in];
t[50].out_chan = glob_ins[ss.O.o_out];
t[51].dest = endingInO;
t[51].in_chan = glob_ins[ss.la.o_in];
t[52].dest = final;
t[52].in_chan = glob_ins[ss.la.r_in];

initial_state:
  atomic{
    reset();
    en_trans(0);
    if:: t[0].en_flag -> next_trans(0); goto connectingO_state;
    :: else -> goto initial_state;
    fi;
  }

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

if :: t[1].en_flag -> next_trans(1); goto trying_state;
:: t[2].en_flag -> next_trans(2); goto abandonConnection_state;
:: t[3].en_flag -> next_trans(3); goto connectingO_state;
:: t[4].en_flag -> next_trans(4); goto error_state;
:: else -> goto connectingO_state;
fi;
}

abandonConnection_state:
atomic{
reset();
en_trans(5);
if :: t[5].en_flag -> next_trans(5); goto terminatingO_state;
:: else -> goto abandonConnection_state;
fi;
}

trying_state:
progress:
atomic{
reset();
en_trans(6);
en_trans(7);
en_trans(8);
en_trans(57);
en_trans(58);
en_trans(59);
en_trans(60);
if :: t[6].en_flag -> next_trans(6); goto connectingR_state;
:: t[7].en_flag -> next_trans(7); goto terminatingO_state;
:: t[8].en_flag -> next_trans(8); goto trying_state;
:: t[57].en_flag -> next_trans(57); goto terminatingI_state;
:: t[58].en_flag -> next_trans(58); goto trying_state;
:: t[59].en_flag -> next_trans(59); goto endingInO_state;
:: t[60].en_flag -> next_trans(60); goto endingInO_state;
:: else -> goto trying_state;
fi;
}

connectingR_state:
atomic{
reset();
en_trans(9);
if

:: t[9].en_flag -> next_trans(9); goto confirming_state;
:: t[10].en_flag -> next_trans(10); goto terminatingO_state;
:: t[11].en_flag -> next_trans(11); goto terminatingI_state;
:: t[12].en_flag -> next_trans(12); goto connectingR_state;
:: t[13].en_flag -> next_trans(13); goto connectingR_state;
:: else -> goto connectingR_state;
fi;
}

confirming_state:
progress0:
atomic{
reset();

en_trans(14);
en_trans(15);
en_trans(16);
en_trans(17);
en_trans(18);
en_trans(19);

if
:: t[14].en_flag -> next_trans(14); goto confirmed_state;
:: t[15].en_flag -> next_trans(15); goto endingAll_state;
:: t[16].en_flag -> next_trans(16); goto endingOnR_state;
:: t[17].en_flag -> next_trans(17); goto endingInR_state;
:: t[18].en_flag -> next_trans(18); goto confirming_state;
:: t[19].en_flag -> next_trans(19); goto confirming_state;
:: else -> goto confirming_state;
fi;
}

confirmed_state:
atomic{
reset();

en_trans(20);
en_trans(53);
en_trans(54);
en_trans(55);
en_trans(56);

if
:: t[20].en_flag -> next_trans(20); goto transparent_state;
:: t[53].en_flag -> next_trans(53); goto endingOnR_state;
:: t[54].en_flag -> next_trans(54); goto endingInR_state;
:: t[55].en_flag -> next_trans(55); goto confirmed_state;
:: t[56].en_flag -> next_trans(56); goto confirmed_state;
:: else -> goto confirmed_state;
fi;
}

progress1:
atomic{
reset();
en_trans(21);
en_trans(22);
en_trans(23);
en_trans(24);
if
:: t[21].en_flag -> next_trans(21); goto terminatingO_state;
:: t[22].en_flag -> next_trans(22); goto terminatingI_state;
:: t[23].en_flag -> next_trans(23); goto transparent_state;
:: t[24].en_flag -> next_trans(24); goto transparent_state;
:: else -> goto transparent_state;
fi;
}

terminatingO_state:
atomic{
reset();
en_trans(25);
en_trans(26);
en_trans(27);
if
:: t[25].en_flag -> next_trans(25); goto final_state;
:: t[26].en_flag -> next_trans(26); goto terminatingO_state;
:: t[27].en_flag -> next_trans(27); goto terminatingO_state;
:: else -> goto terminatingO_state;
fi;
}

terminatingI_state:
atomic{
reset();
en_trans(28);
en_trans(29);
en_trans(30);
if
:: t[28].en_flag -> next_trans(28); goto final_state;
:: t[29].en_flag -> next_trans(29); goto terminatingI_state;
:: t[30].en_flag -> next_trans(30); goto terminatingI_state;
:: else -> goto terminatingI_state;
fi;
}

endingAll_state:
atomic{
    reset();
en_trans(31);
en_trans(32);
en_trans(33);
en_trans(34);
en_trans(35);
en_trans(36);
en_trans(37);

if :: t[31].en_flag -> next_trans(31); goto endingOnR_state;
:: t[32].en_flag -> next_trans(32); goto endingInR_state;
:: t[33].en_flag -> next_trans(33); goto endingInO_state;
:: t[34].en_flag -> next_trans(34); goto endingAll_state;
:: t[35].en_flag -> next_trans(35); goto endingAll_state;
:: t[36].en_flag -> next_trans(36); goto endingAll_state;
:: t[37].en_flag -> next_trans(37); goto endingAll_state;
:: else -> goto endingAll_state;
fi;
}

endingOnR_state:
atomic{
    reset();
en_trans(38);
en_trans(39);
en_trans(40);
en_trans(41);

if :: t[38].en_flag -> next_trans(38); goto terminatingR_state;
:: t[39].en_flag -> next_trans(39); goto terminatingO_state;
:: t[40].en_flag -> next_trans(40); goto endingOnR_state;
:: t[41].en_flag -> next_trans(41); goto endingOnR_state;
:: else -> goto endingOnR_state;
fi;
}

endingInR_state:
atomic{
    reset();
en_trans(42);
en_trans(43);
en_trans(44);
en_trans(45);
if :: t[42].en_flag -> next_trans(42); goto terminatingR_state;
:: t[43].en_flag -> next_trans(43); goto terminatingI_state;
:: t[44].en_flag -> next_trans(44); goto endingInR_state;
:: t[45].en_flag -> next_trans(45); goto endingInR_state;
:: else -> goto endingInR_state;
fi;
}

endingInO_state:
atomic{
  reset();
  en_trans(46);
  en_trans(47);
  en_trans(48);
  en_trans(49);
  en_trans(50);
  en_trans(51);
  if :: t[46].en_flag -> next_trans(46); goto terminatingO_state;
  :: t[47].en_flag -> next_trans(47); goto terminatingI_state;
  :: t[48].en_flag -> next_trans(48); goto endingInO_state;
  :: t[49].en_flag -> next_trans(49); goto endingInO_state;
  :: t[50].en_flag -> next_trans(50); goto endingInO_state;
  :: t[51].en_flag -> next_trans(51); goto endingInO_state;
  :: else -> goto endingInO_state;
  fi;
}

terminatingR_state:
atomic{
  reset();
  en_trans(52);
  if :: t[52].en_flag -> next_trans(52); goto final_state;
  :: else -> goto terminatingR_state;
  fi;
}

error_state:

final_state:
skip;

/*=================================================================*/
/* environment process */

/* ================================ */
/* environment process */
active proctype env() {
  mtype i_sig, o_sig, r_sig;

  { end:  do
    :: ss.Ia.box_in_ready ->
      ss.Ia.box_in_ready = false;
      glob_ins[ss.Ia.box_in]!setup;
    :: ss.Ia.i_in_ready ->
      if
        :: glob_ins[ss.Ia.i_in]!teardown;
        :: glob_ins[ss.Ia.i_in]!other;
        fi unless{
          (i_sig == teardown) ->
            glob_ins[ss.Ia.i_in]!downack;
            i_sig = 0;
        }
    end
    :: ss.Ia.o_in_ready ->
      if
        :: glob_ins[ss.Ia.o_in]!teardown;
        :: glob_ins[ss.Ia.o_in]!other;
        fi unless{
          (o_sig == teardown) ->
            glob_ins[ss.Ia.o_in]!downack;
            o_sig = 0;
          :: (o_sig == setup) ->
            o_sig = 0;
          fi;
        }
      :: glob_ins[ss.Ia.o_in]!upack;
      :: glob_ins[ss.Ia.o_in]!avail;
      :: glob_ins[ss.Ia.o_in]!lupack;
      :: glob_ins[ss.Ia.o_in]!lunavail;
      :: glob_ins[ss.Ia.o_in]!teardown;
      :: glob_ins[ss.Ia.o_in]!upack;
      :: glob_ins[ss.Ia.o_in]!unknown;
      :: glob_ins[ss.Ia.o_in]!teardown;
      fi;
    end
    :: ss.Ia.r_in_ready ->
      if
        :: (r_sig == teardown) ->
          glob_ins[ss.Ia.r_in]!downack;
          r_sig = 0;
        :: (ss.cs == confirming) ->
          if
            :: glob_ins[ss.Ia.r_in]!confirm;
            :: glob_ins[ss.Ia.r_in]!nonconfirm;
            fi;
          fi;
    end
  end
}
fi;
  od
}
  unless{
    if
      :: glob_outs[ss.O.box_out]?setup -> 
        if
          :: ss.cs == connectingO -> 
            o_sig = setup;
          :: ss.cs == connectingR -> 
            glob_ins[ss.Ia.r_in]?upack;
        fi;
      :: glob_outs[ss.O.i_out]?upack;
      :: glob_outs[ss.O.i_out]?downack;
      :: atomic{ glob_outs[ss.O.i_out]?teardown -> 
        i_sig = teardown;
      }
    :: glob_outs[ss.O.i_out]?avail;
    :: glob_outs[ss.O.i_out]?unavail;
    :: glob_outs[ss.O.i_out]?unknown;
    :: glob_outs[ss.O.i_out]?other;
    :: glob_outs[ss.O.o_out]?downack;
    :: atomic{ glob_outs[ss.O.o_out]?teardown -> 
        o_sig = teardown;
    }
    :: glob_outs[ss.O.o_out]?other;
  }
  :: glob_outs[ss.O.r_out]?downack;
  :: atomic{ glob_outs[ss.O.r_out]?teardown -> 
    r_sig = teardown;
  }
  :: glob_outs[ss.O.r_out]?other;
  fi;
}
  goto end;
Appendix E
Blace Phone Interface
/*=============================================================*/
/* type definitions */

mtype = { teardown, downack, setup, upack, avail, unavail, unknown, none};
mtype = { offhook, dialed, onhook, other};
mtype = { accepted, waiting, rejected, nullified};
mtype = { post_process};

mtype = { initial, ringing, dialing, connecting, silent, ringback,
          busytone, errortone, talking, disconnected};
mtype = { idle, work};

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

typedef Ia_type {
    byte box_in = 0;
    byte c_in = 1;
    byte v_in = 2;
    byte a_in = 3;
    byte old_c_in = 4;
    bool box_in_ready = true;
    bool c_in_ready = false;
    bool a_in_ready = true;
    bool old_c_in_ready = false;
    byte selected;
};

typedef O_type {
    byte box_out = 0;
    byte c_out = 1;
};

typedef IE_type {
    chan internal = [0] of {mtype};
};

typedef SnapShot {
    mtype cs;
    mtype cs_post_process;
    Ia_type Ia;
    O_type O;
    IE_type IE;
};

/*==============================================================*/
/* global variable declarations */

chan glob_ins[5] = [0] of {mtype};
chan glob_outs[2] = [0] of {mtype};
SnapShot ss;
type sig;
type inter_sig;
Transition t[53];
byte counter = 0;
/*==============================================================*/
/* inline functions */
inline setup_initial(){
    ss.Ia.c_in_ready = true;
};
inline teardown_cleanup(){
    ss.Ia.c_in_ready = false;
    ss.Ia.old_c_in_ready = true;
};
inline reset() {
    if :: glob_ins[ss.Ia.box_in]?sig -> ss.Ia.selected = ss.Ia.box_in;
    :: glob_ins[ss.Ia.c_in]?sig -> ss.Ia.selected = ss.Ia.c_in;
    :: glob_ins[ss.Ia.v_in]?sig -> ss.Ia.selected = ss.Ia.v_in;
    :: glob_ins[ss.Ia.a_in]?sig -> ss.Ia.selected = ss.Ia.a_in;
    fi;
};
inline reset_pp() {
    if :: glob_ins[ss.Ia.old_c_in]?sig -> ss.Ia.selected = ss.Ia.old_c_in;
    fi;
};
inline en_events(n){
    if ::(n==0) && ss.Ia.selected == ss.Ia.box_in;
    ::(n==1) && ss.Ia.selected == ss.Ia.a_in;
    ::(n==2) && ss.Ia.selected == ss.Ia.v_in;
    ::(n==3) && ss.Ia.selected == ss.Ia.a_in;
    ::(n==4) && ss.Ia.selected == ss.Ia.a_in;
    ::(n==5) && ss.Ia.selected == ss.Ia.c_in;
    ::(n==6) && ss.Ia.selected == ss.Ia.c_in;
    ::(n==7) && ss.Ia.selected == ss.Ia.a_in;
    ::(n==8) && ss.Ia.selected == ss.Ia.v_in;
    ::(n==9) && ss.Ia.selected == ss.Ia.v_in;
    ::(n==10) && ss.Ia.selected == ss.Ia.v_in;
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 == upack;
::(n==6) && sig == teardown;
::(n==7) && sig == onhook;
::(n==8) && sig == waiting;
::(n==9) && sig == accepted;
::(n==10) && sig == rejected;
::(n==11) && sig == unknown;
::(n==12) && sig == unavail;
::(n==13) && sig == avail;
::(n==14) && sig == teardown;
::(n==15) && sig == onhook;
::(n==16) && sig == accepted;
::(n==17) && sig == rejected;
::(n==18) && sig == nullified;
::(n==19) && sig == unknown;
::(n==20) && sig == unavail;
::(n==21) && sig == avail;
::(n==22) && sig == none;
::(n==23) && sig == teardown;
::(n==24) && sig == onhook;
::(n==25) && sig == waiting;
::(n==26) && sig == accepted;
::(n==27) && sig == nullified;
::(n==28) && sig == unknown;
::(n==29) && sig == avail;
::(n==30) && sig == none;
::(n==31) && sig == teardown;
::(n==32) && sig == onhook;
::(n==33) && sig == waiting;
::(n==34) && sig == accepted;
::(n==35) && sig == rejected;
::(n==36) && sig == nullified;
::(n==37) && sig == unavail;
::(n==38) && sig == avail;
::(n==39) && sig == none;
::(n==40) && sig == teardown;
::(n==41) && sig == onhook;
::(n==42) && sig == waiting;
::(n==43) && sig == rejected;
::(n==44) && sig == nullified;
::(n==45) && sig == unknown;
::(n==46) && sig == unavail;
::(n==47) && sig == none;
::(n==48) && sig == teardown;
::(n==49) && sig == onhook;
::(n==50) && sig == onhook;
::(n==51) && inter_sig == post_process;
::(n==52) && sig == downack;
fi;

inline next_trans(n)
if (n==0) -> setup_initial();
t[0].out_chan!upack;
ss.cs = t[0].dest;

::(n==1) -> ss.cs = t[1].dest;

::(n==2) -> t[2].out_chan!avail;
ss.cs = t[2].dest;

::(n==3) -> setup_initial();
t[3].out_chan!setup;
ss.cs = t[3].dest;

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

::(n==5) -> ss.cs = t[5].dest;

::(n==6) -> t[6].out_chan!downack;
ss.Ia.c_in_ready = false;
ss.cs = t[6].dest;

::(n==7) -> t[7].out_chan!teardown;
ss.IE.internal!post_process;
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) -> ss.cs = t[11].dest;

::(n==12) -> ss.cs = t[12].dest;

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

::(n==14) -> t[14].out_chan!downack;
ss.Ia.c_in_ready = false;
ss.cs = t[14].dest;

::(n==15) -> t[15].out_chan!teardown;
ss.IE.internal!post_process;
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) -> ss.cs = t[20].dest;
::(n==21) -> ss.cs = t[21].dest;
::(n==22) -> ss.cs = t[22].dest;
::(n==23) -> t[23].out_chan!downack;
ss.Ia.c_in_ready = false;
ss.cs = t[23].dest;
::(n==24) -> t[24].out_chan!teardown;
ss.IE.internal!post_process;
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) -> ss.cs = t[29].dest;
::(n==30) -> ss.cs = t[30].dest;
::(n==31) -> t[31].out_chan!downack;
ss.Ia.c_in_ready = false;
ss.cs = t[31].dest;
::(n==32) -> t[32].out_chan!teardown;
ss.IE.internal!post_process;
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[35].dest;
::(n==37) -> ss.cs = t[37].dest;
::(n==38) -> ss.cs = t[38].dest;
::(n==39) -> ss.cs = t[39].dest;
::(n==40) -> t[40].out_chan!downack;
ss.Ia.c_in_ready = false;
ss.cs = t[40].dest;
::(n==41) -> t[41].out_chan!teardown;
    ss.IE.internal!post_process;
    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) -> ss.cs = t[46].dest;
::(n==47) -> ss.cs = t[47].dest;
::(n==48) -> t[48].out_chan!downack;
    ss.Ia.c_in_ready = false;
    ss.cs = t[48].dest;
::(n==49) -> t[49].out_chan!teardown;
    ss.IE.internal!post_process;
    ss.cs = t[49].dest;
::(n==50) -> ss.cs = t[50].dest;
::(n==51) -> ss.cs_post_process = t[51].dest;
::(n==52) -> ss.Ia.old_c_in_ready = false;
    ss.cs_post_process = t[52].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;
};

/*==================================================================*/
/* bound black phone interface box process */
active proctype BPI() {
    ss.cs = initial;
    //statically declare transitions

t[0].dest = ringing;
t[0].in_chan = glob_ins[ss.la.box_in];
t[0].out_chan = glob_outs[ss.O.c_out];

376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424

t[1].dest = dialing;
t[1].in_chan = glob_outs[ss.la.a_in];
t[2].dest = talking;
t[2].in_chan = glob_ins[ss.la.v_in];
t[2].out_chan = glob_outs[ss.O.c_out];
t[3].dest = connecting;
t[3].in_chan = glob_ins[ss.la.a_in];
t[3].out_chan = glob_outs[ss.O.box_out];
t[4].dest = initial;
t[4].in_chan = glob_ins[ss.la.a_in];
t[5].dest = silent;
t[5].in_chan = glob_ins[ss.la.c_in];
t[6].dest = disconnected;
t[6].in_chan = glob_ins[ss.la.c_in];
t[6].out_chan = glob_outs[ss.O.c_out];
t[7].dest = initial;
t[7].in_chan = glob_ins[ss.la.a_in];
t[7].out_chan = glob_outs[ss.O.c_out];
t[8].dest = ringback;
t[8].in_chan = glob_ins[ss.la.v_in];
t[9].dest = talking;
t[9].in_chan = glob_ins[ss.la.v_in];
t[10].dest = busytone;
t[10].in_chan = glob_ins[ss.la.v_in];
t[11].dest = errortone;
t[11].in_chan = glob_ins[ss.la.c_in];
t[12].dest = busytone;
t[12].in_chan = glob_ins[ss.la.c_in];
t[13].dest = talking;
t[13].in_chan = glob_ins[ss.la.c_in];
t[14].dest = disconnected;
t[14].in_chan = glob_ins[ss.la.c_in];
t[14].out_chan = glob_outs[ss.O.c_out];
t[15].dest = initial;
t[15].in_chan = glob_ins[ss.la.a_in];
t[15].out_chan = glob_outs[ss.O.c_out];

426
t[16].dest = talking;
427t[16].in_chan = glob_ins[ss.la.v_in];
428
429t[17].dest = busytone;
430t[17].in_chan = glob_ins[ss.la.v_in];
431
432t[18].dest = silent;
433t[18].in_chan = glob_ins[ss.la.v_in];
434
435t[19].dest = errortone;
436t[19].in_chan = glob_ins[ss.la.c_in];
437
438t[20].dest = busytone;
439t[20].in_chan = glob_ins[ss.la.c_in];
440
441t[21].dest = talking;
442t[21].in_chan = glob_ins[ss.la.c_in];
443
444t[22].dest = silent;
445t[22].in_chan = glob_ins[ss.la.c_in];
446
447t[23].dest = disconnected;
448t[23].in_chan = glob_ins[ss.la.c_in];
449
t[23].out_chan = glob_outs[ss.O.c_out];
450
451t[24].dest = initial;
452t[24].in_chan = glob_ins[ss.la.a_in];
453
t[24].out_chan = glob_outs[ss.O.c_out];
454
455t[25].dest = ringback;
456t[25].in_chan = glob_ins[ss.la.v_in];
457
458t[26].dest = talking;
459t[26].in_chan = glob_ins[ss.la.v_in];
460
461t[27].dest = silent;
462t[27].in_chan = glob_ins[ss.la.v_in];
463
464t[28].dest = errortone;
465t[28].in_chan = glob_ins[ss.la.c_in];
466
467t[29].dest = talking;
468t[29].in_chan = glob_ins[ss.la.c_in];
469
470t[30].dest = silent;
471t[30].in_chan = glob_ins[ss.la.c_in];
472
473t[31].dest = disconnected;
474t[31].in_chan = glob_ins[ss.la.c_in];
475
t[31].out_chan = glob_outs[ss.O.c_out];
476
477
t[32].dest = initial;
t[32].in_chan = glob_ins[ss.la.a_in];
t[32].out_chan = glob_outs[ss.O.c_out];

t[33].dest = ringback;
t[33].in_chan = glob_ins[ss.la.v_in];

t[34].dest = talking;
t[34].in_chan = glob_ins[ss.la.v_in];

t[35].dest = busytone;
t[35].in_chan = glob_ins[ss.la.v_in];

t[36].dest = silent;
t[36].in_chan = glob_ins[ss.la.v_in];

t[37].dest = busytone;
t[37].in_chan = glob_ins[ss.la.c_in];

t[38].dest = talking;
t[38].in_chan = glob_ins[ss.la.c_in];

t[39].dest = silent;
t[39].in_chan = glob_ins[ss.la.c_in];

t[40].dest = disconnected;
t[40].in_chan = glob_ins[ss.la.c_in];
t[40].out_chan = glob_outs[ss.O.c_out];

t[41].dest = initial;
t[41].in_chan = glob_ins[ss.la.a_in];
t[41].out_chan = glob_outs[ss.O.c_out];

t[42].dest = ringback;
t[42].in_chan = glob_ins[ss.la.v_in];

t[43].dest = busytone;
t[43].in_chan = glob_ins[ss.la.v_in];

t[44].dest = silent;
t[44].in_chan = glob_ins[ss.la.v_in];

t[45].dest = busytone;
t[45].in_chan = glob_ins[ss.la.c_in];

t[46].dest = busytone;
t[46].in_chan = glob_ins[ss.la.c_in];

t[47].dest = silent;
t[47].in_chan = glob_ins[ss.la.c_in];

t[48].dest = disconnected;
t[48].in_chan = glob_ins[ss.la.c_in];
t[48].out_chan = glob_outs[ss.O.c_out];
533
t[49].dest = initial;
534
t[49].in_chan = glob_ins[ss.Ia.a_in];
535
t[49].out_chan = glob_outs[ss.O.c_out];
536
t[50].dest = initial;
537
t[50].in_chan = glob_ins[ss.Ia.a_in];
538
dEnd_initial_state:
539
atomic{
 reset();
544
 en_trans(0);
545
 en_trans(1);
546
 if 548
 :: t[0].en_flag -> next_trans(0); goto ringing_state;
549
 :: t[1].en_flag -> next_trans(1); goto dialing_state;
550
 :: else -> goto end_initial_state;
551
 fi;
552
}
553
ringing_state:
554 atomic{
 reset();
558
 en_trans(2);
559
 if 561
 :: t[2].en_flag -> next_trans(2); goto talking_state;
562
 :: else -> goto ringing_state;
563
 fi;
564
}
565
dialing_state:
566 atomic{
 reset();
570
 en_trans(3);
571
 en_trans(4);
572
 if 574
 :: t[3].en_flag -> next_trans(3); goto connecting_state;
575
 :: t[4].en_flag -> next_trans(4); goto end_initial_state;
576
 :: else -> goto dialing_state;
577
 fi;
578
}
579
connecting_state:
580 atomic{
 reset();
582


if
:: t[5].en_flag -> next_trans(5); goto silent_state;
:: t[6].en_flag -> next_trans(6); goto disconnected_state;
:: t[7].en_flag -> next_trans(7); goto end_initial_state;
:: else -> goto connecting_state;
fi;
}

silent_state:
atomic{
reset();
en_trans(8);
en_trans(9);
en_trans(10);
en_trans(11);
en_trans(12);
en_trans(13);
en_trans(14);
en_trans(15);
if
:: t[8].en_flag -> next_trans(8); goto ringback_state;
:: t[9].en_flag -> next_trans(9); goto talking_state;
:: t[10].en_flag -> next_trans(10); goto busytone_state;
:: t[11].en_flag -> next_trans(11); goto errortone_state;
:: t[12].en_flag -> next_trans(12); goto busytone_state;
:: t[13].en_flag -> next_trans(13); goto talking_state;
:: t[14].en_flag -> next_trans(14); goto disconnected_state;
:: t[15].en_flag -> next_trans(15); goto end_initial_state;
:: else -> goto silent_state;
fi;
}

ringback_state:
atomic{
reset();
en_trans(16);
en_trans(17);
en_trans(18);
en_trans(19);
en_trans(20);
en_trans(21);
en_trans(22);
en_trans(23);
en_trans(24);
if
:: t[16].en_flag -> next_trans(16); goto talking_state;
:: t[17].en_flag -> next_trans(17); goto busytone_state;
:: t[18].en_flag -> next_trans(18); goto silent_state;
:: t[19].en_flag -> next_trans(19); goto errortone_state;
:: t[20].en_flag -> next_trans(20); goto busytone_state;
:: t[21].en_flag -> next_trans(21); goto talking_state;
:: t[22].en_flag -> next_trans(22); goto silent_state;
:: t[23].en_flag -> next_trans(23); goto disconnected_state;
:: t[24].en_flag -> next_trans(24); goto end_initial_state;
:: else -> goto ringback_state;
fi;

} 648
649
650  busytone_state:
651  atomic{
652        reset();
653
654        en_trans(25);
655        en_trans(26);
656        en_trans(27);
657        en_trans(28);
658        en_trans(29);
659        en_trans(30);
660        en_trans(31);
661        en_trans(32);
662
663        if
664        :: t[25].en_flag -> next_trans(25); goto ringback_state;
665        :: t[26].en_flag -> next_trans(26); goto talking_state;
666        :: t[27].en_flag -> next_trans(27); goto silent_state;
667        :: t[28].en_flag -> next_trans(28); goto errortone_state;
668        :: t[29].en_flag -> next_trans(29); goto talking_state;
669        :: t[30].en_flag -> next_trans(30); goto silent_state;
670        :: t[31].en_flag -> next_trans(31); goto disconnected_state;
671        :: t[32].en_flag -> next_trans(32); goto end_initial_state;
672        :: else -> goto busytone_state;
673        fi;
674  }
675
676  errortone_state:
677  atomic{
678        reset();
679
680        en_trans(33);
681        en_trans(34);
682        en_trans(35);
683        en_trans(36);
684        en_trans(37);
685        en_trans(38);
686        en_trans(39);
687        en_trans(40);
688        en_trans(41);
689
if
:: t[33].en_flag -> next_trans(33); goto ringback_state;
:: t[34].en_flag -> next_trans(34); goto talking_state;
:: t[35].en_flag -> next_trans(35); goto busytone_state;
:: t[36].en_flag -> next_trans(36); goto silent_state;
:: t[37].en_flag -> next_trans(37); goto busytone_state;
:: t[38].en_flag -> next_trans(38); goto talking_state;
:: t[39].en_flag -> next_trans(39); goto silent_state;
:: t[40].en_flag -> next_trans(40); goto disconnected_state;
:: t[41].en_flag -> next_trans(41); goto end_initial_state;
:: else -> goto errortone_state;
fi;
}
talking_state:
atomic{
    reset();
en_trans(42);
en_trans(43);
en_trans(44);
en_trans(45);
en_trans(46);
en_trans(47);
en_trans(48);
en_trans(49);
    if
:: t[42].en_flag -> next_trans(42); goto ringback_state;
:: t[43].en_flag -> next_trans(43); goto busytone_state;
:: t[44].en_flag -> next_trans(44); goto silent_state;
:: t[45].en_flag -> next_trans(45); goto busytone_state;
:: t[46].en_flag -> next_trans(46); goto busytone_state;
:: t[47].en_flag -> next_trans(47); goto silent_state;
:: t[48].en_flag -> next_trans(48); goto disconnected_state;
:: t[49].en_flag -> next_trans(49); goto end_initial_state;
:: else -> goto talking_state;
fi;
}
disconnected_state:
atomic{
    reset();
en_trans(50);
    if
:: t[50].en_flag -> next_trans(50); goto end_initial_state;
:: else -> goto disconnected_state;
fi;
}
}
active proctype pp() {
    byte inter_sig;
    ss.cs_post_process = idle;
    t[51].dest = work;
    t[52].dest = idle;
    t[52].in_chan = glob_ins[ss.la.old_c_in];
    end_idle_state:
        atomic{
            ss.IE.internal?inter_sig;
            en_trans(51);
            if
                :: t[51].en_flag -> next_trans(51); goto work_state;
                :: else -> goto end_idle_state;
            fi;
        }
    work_state:
        atomic{
            reset_pp();
            en_trans(52);
            if
                :: t[52].en_flag -> next_trans(52); goto end_idle_state;
                :: else -> goto work_state;
            fi;
        }
}

active proctype env() {
    end:
        do
            :: ss.la.box_in_ready && (ss.cs == initial) ->
                counter = counter + 1;
                glob_ins[ss.la.box_in]!setup;
            :: ss.la.a_in_ready ->
                if
:: (ss.cs == initial) -> glob_ins[ss.la.in] ! offhook;

glob_ins[ss.la.in] ! dialed;

:: !(ss.cs == initial) && !(ss.cs == ringing) && !(ss.la.old_c_in_ready) ->
glob_ins[ss.la.in] ! onhook;

:: else -> glob_ins[ss.la.in] ! other;
fi;

:: ss.la.c_in_ready && !(ss.cs == ringing) ->

if

:: glob_ins[ss.la.c_in] ! teardown;
:: !(ss.cs == errortone) -> glob_ins[ss.la.c_in] ! unknown;
:: !(ss.cs == busytone) -> glob_ins[ss.la.c_in] ! unavail;
:: !(ss.cs == talking) -> glob_ins[ss.la.c_in] ! avail;
:: !(ss.cs == silent) -> glob_ins[ss.la.c_in] ! none;
:: !(ss.cs == talking) -> glob_ins[ss.la.v_in] ! accepted;
:: !(ss.cs == ringback) -> glob_ins[ss.la.v_in] ! waiting;
:: !(ss.cs == busytone) -> glob_ins[ss.la.v_in] ! rejected;
:: !(ss.cs == nullified) -> glob_ins[ss.la.v_in] ! nullified;
fi;

:: ss.la.old_c_in_ready -> glob_ins[ss.la.old_c_in] ! downack;

do

unless{

if

:: atomic{ glob_outs[ss.O.box_out] ? setup -> glob_ins[ss.la.c_in] ! upack;}
:: atomic{ glob_outs[ss.O.c_out] ? upack -> glob_ins[ss.la.v_in] ! accepted;}
:: glob_outs[ss.O.c_out] ? avail;
:: glob_outs[ss.O.c_out] ? downack;
:: atomic{ glob_outs[ss.O.c_out] ? teardown -> teardown_cleanup();}
fi;
}

goto end;

};

152
Bibliography


