Using LOTOS for Modelling and Analysing an Online Feature Manager: A Study

Stephan Reiff-Marganiec
sreiff@dcs.gla.ac.uk,
University of Glasgow, Department of Computer Science,
Glasgow, G12 8QQ, Scotland

June 11, 2001

1 Introduction

We consider the suitability of LOTOS for modelling and analysing an online feature manager. The work forms part of a larger project on hybrid methods for feature interaction detection and resolution.

In modern telecommunications networks features such as "Call Waiting" (a subscriber is informed when a call attempt is made to an busy line) or "Group Ringing" (where a number of telephones reacts to an incoming call attempt and the call can be answered at any phone) are common. The number of features is steadily increasing. Further, a general move towards an open market leads to more suppliers of features.

Features are added functionality, they are developed independent of each other and are tested. However, making features available together introduces the potential for interaction between them: two features might react to the same trigger (for example a call attempt to a busy line) and offer different ways of handling this. This problem is known as the *feature interaction problem.*

We advocate a solution to the problem in form of an online feature manager that detects and resolves interactions. The feature manager applies rules for resolution to resolve interactions at runtime. The rules are found during a formal analysis phase, the interactions are detected at runtime.

The formal analysis phase requires a sound understanding of the system which is best provided by a model of the system. As the resolution mechanism must be added to the system, we aim to develop a model that is operationally close to a real implementation of the system, thus facilitating an easy transfer of the results.

In this report we analyse the suitability of the LOTOS description technique and the available tool support w.r.t. the given problem.

1.1 Overview

We introduce the idea of an online feature manager and describe the basic functionality of the feature manager. An analysis of properties required in a modelling notation to describe the feature manager and allow to work on extended capabilities justifies why LOTOS seems a good choice. We then develop the model for interaction detection and show some experimental results. We present initial ideas for automatic resolution of detected interactions – the modelling of which leads to a reconsideration of the chosen modelling notation. The appendices contain source code of the LOTOS prototype model, the Perl script for customisation of the prototype model and instruction show to build an executable prototype.

2 An Online Feature Manager

The aim of this section is to provide an overview of the online feature manager. The basic idea and the detection method is due to Marples and Magill ([MM98, Mar00] for full details).

The role of the on-line feature manager is to mediate between new features, the legacy system, and the other components of the system (e.g. terminal devices). Specifically, the feature manager passes incoming messages from the switching hardware to the legacy software and all additional features; for simplicity we assume that this happens in parallel. The legacy software and the features are embedded within a cocoon that offers a consistent interface. For the remainder of this section we use the term *feature* to denote both the new features as well as the basic call software. (We expect the latter to be included in the legacy software.)

When a message is processed by a feature, two possible behaviours can occur: the message triggers a response (one or more messages) or it does not. All triggered responses are sent back to the feature manager, concluding with a *transaction finished* message. The latter is also sent by features not responding with a proper message[1].

The responses are collected by the feature manager and then evaluated. There are three possible outcomes:

1. no proper messages were received,
2. exactly one proper message were received,
3. multiple proper messages were received.

In the first two cases, the feature manager's role is straightforward: the basic call progresses (the former case) or the feature takes control of the call (the latter case). It is the third case which is interesting: the possibility that more than one feature might reply and the consequent potential for an interaction is indeed the motivation for the feature manager. Rather than resolve an interaction at this point, we explore the *space* of possible resolutions in the following way.

The feature manager stores the current state and initiates copies of the feature processes. The current state includes the list of events and other local information; it describes a *snapshot*

[1] Any message apart from the *transaction finished* message.

of the system from the viewpoint of the feature manager. Assuming that at least one message has been received, the first one is fed to the copies of the features, the responses are gathered and processed. Part of this processing may involve sending and receiving further messages to and from the (copies of the) features.

This process terminates when there are no further responses; at this point, we have a sequence of messages and responses, which we consider as a branch in a *behaviour tree.* To construct the rest of the tree, we restore the initial state, farm out the next event, generate a new branch, and so on. Once all events are processed we have defined a full *behaviour tree.* It remains to select the most promising path in that tree. *Which path is to be chosen?* is the key question our research project will answer, by providing a theory of features and events that ultimately guides the choice. The theory will also provide mechanisms to detect unsuitable paths early in their construction process. Due to the close connection between the underlying theory and the transactional online feature manager we refer to the proposed approach as *hybrid*, for further details we refer to [CR00, Rei00].

When the path is selected, it is committed by sending out the appropriate messages to the switching hardware. In detail, this means that all those messages produced by the features in response to the trigger event that are part of the selected branch need to be replayed and this time send out to the user or trunk. This playback is no longer piped through the feature manager's detection and resolution mechanism.

The approach depends on certain assumptions about the system, namely that we can create copies of all the associated processes. In a legacy system, one might emulate process copies if the system provides the functionality to reset a process to a certain state. In this case, to emulate multiple copies we maintain stacks of all messages, reset the control process and replay messages from the stack to obtain a process in the desired state. This is very much like failure recovery in database systems employing a rollback-commit strategy. For further details of the construction of the behaviour trees, see [CMM99].

The approach originally described by Marples and Magill only considers the reaction of features as described by the initial event and direct responses (i.e. feature behaviour that can be described as trigger-response(s) pair), whereas we will also consider hypothetical events to obtain a more complete picture of the solution space. This extension allows us to deal with features that require further input (e.g. from a user) and improves the quality of the solutions found, by considering the influence of a decision over a longer timescale. To motivate our concern with the longer term effects of a decision consider the following scenario: a user subscribes to three way calling and two features (A and B), both requiring a PIN. During a call both A and B request a PIN, triggered by the same event; the requests do not clash, hence an "ask for PIN" is committed to the user. The user enters a PIN, which is correct for one feature only (say A), so we have to resolve an interaction between "continue call" and "end call", as required by the features. We decide to continue the call, hence B is reset to the last stable state (awaiting a PIN) and the call continues. The user now flashes to initiate a three way call, and on the request to dial he dials a wrong number which coincidently is the correct PIN for feature B. The feature B now triggers its actions, whereas the user would have expected a message informing him/her that the telephone number was not recognised.

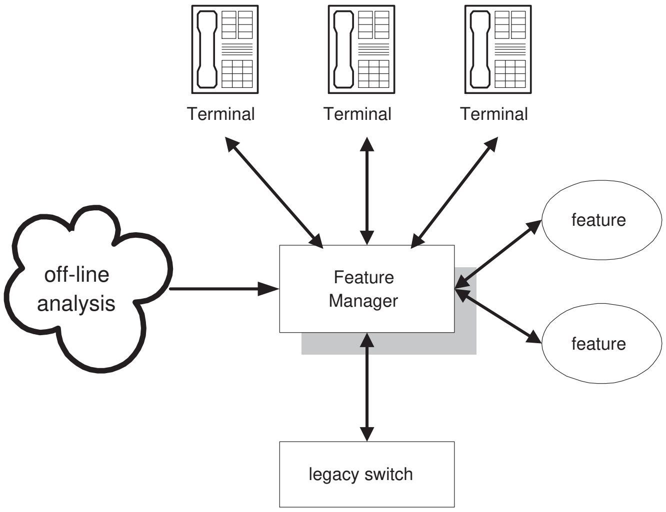

Figure 1: An Online Feature Manager – Systems Architecture

3 Notation

The feature manager developed by Marples and Magill provides a very good runtime detection method, however the resolution mechanism is very crude. In order to enhance the resolution a sound understanding of the approach is required and it was felt helpful to have a model of the system for experimental purposes.

A number of key concepts required in the modelling notation can be derived from an analysis of the detailed specification [Rei00]. We have presented an overview in section 2 and will not go into any specification details here, but rather summarise the elemental details. Features are black boxes communicating via message exchange with their environment. Some features might have internal loops, e.g. *three way calling* allows the user to toggle potentially infinitely often between the two calls. The feature manager gatheres the responses and generates the behaviour tree as described in the previous section. As the behaviour tree contains all possible solutions (that is all potential interleavings of executions of features), we have to commit to one of the given behaviours, we refer to the behaviour tree as *solution space*. Thus, any notation suitable for modelling should provide:

- a natural way of expressing events (or messages) as a basic concept
- a way of expressing looping behaviour of features
- tree data types (to handle the solution space)

In addition it would be preferable to have some of the following:

- a recognised notation (maybe even standardised)
- tool support for the notation (for simulation and verification)

After investigating various notations, process algebras seem a natural choice. The basic terms in process algebras are actions which are combined by sequencing and choice operators. Sending or receiving a message can be treated as events. Parallelism operators seem useful to describe interleavings. The requirement for complex data types suggests LOTOS [ISO89]. LOTOS combines process algebra with an algebraic data type definition language (Act One). Furthermore, LOTOS is standardised by ISO and well known in the telecommunications area. Extensive tool-sets for verification and simulation are available in the form of CAESAR/ALDEBARAN [CAD] and TOPO/LOLA [LOL, ?].

4 Constructing the Solution Space

Having decided on LOTOS as notation we can attempt to implement a model representing the specification of the system. A further requirement for this model is that it should follow the operational description provided by Marples, in particular the internal behaviour of features should be hidden from the feature manager.

The LOTOS model makes use of processes and abstract data types, where those processes representing parts of the system (like the feature manager) communicate via messages. In the model we define a data type for messages, which contains the IO aspect, event and argument. The IO aspect is either *rcv* or *snd* denoting the direction of a message from the viewpoint of a feature. The data type also includes tests for equality.

4.1 Overview of the Model

We model a telecommunications switch with feature manager, hence our main components are the feature manager and the features. We add a component, trigger generator, to close the system. Figure 2 shows the overall structure of the model. The trigger generator produces a trigger event (which in an operational system would be received from a terminal device or remote switch) and consumes the responses (which would be consumed by a remote switch or terminal device).

The feature process is discussed in detail in section 4.3; the feature manager process in section 4.4. Here we give an overview of the whole system.

A generated message is received by the feature manager which passes it on to the features in parallel. Each feature responds and the responses are collected in independent queues. A feedback process is used to reflect the responses back to all features and thus explore whether further responses are triggered; the process terminates once all queues are empty and no further responses are received from any feature. This process allows to explore all possible responses to a trigger message. The result is the solution space, stored in a tree data structure. This construction process is controlled by the feature manager.

In addition the feature manager controls the creation of copies of the features so that a rollback strategy can be applied. The idea is that a copy is created, then all further messages are sent to the copy until a rollback is required. Upon rollback, the copy is killed and the original feature, whose state remained unchanged provides the previous state of the system.

The solution space is passed to a pruning and extraction process which produce a single solution which then is committed to the environment.

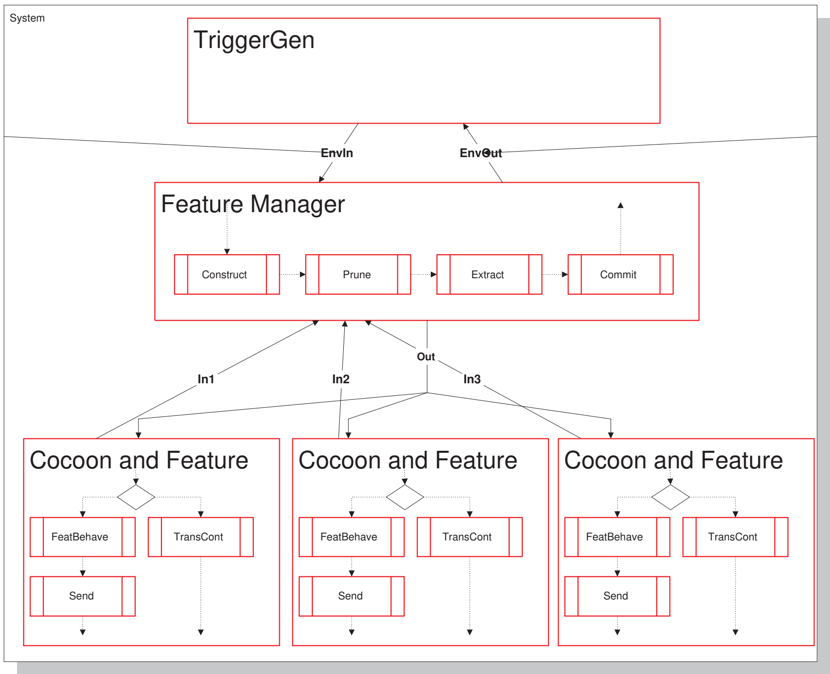

Figure 2: Overview of the Model

A special aspect of the model is that we can deploy as many features as we require. It is assumed that we have at least 2 features as otherwise no interaction can occur; such a model is of no interest to us. However, this requires data types and process parameters which depend on the number of actually deployed features. For example, a model with n features only requires n response queues in the feature manager and n channels for communication between the features and the feature manager. These numbers obviously change if we deploy fewer or more features.

In LOTOS the structure of processes is fixed, i.e. the number and structure of channels is clearly defined (note that E-LOTOS extends on this by allowing channel splitting, i.e. the structure of a channel can be adapted). When a process is declared it is given an interface of the form `procname [list of channels]`. The list of channels is of finite length and channel is named. Furthermore, synchronisation of processes is achieved by binding channels to each other: for example `proc1 [in] |[in]| proc2[in]` means that the two processes synchronise on channel `in`.

In order to create models with certain numbers of features a Perl script has been developed to overcome the above restrictions. The script asks for the number of features and the actual features that shall be provided in the model, then reads a prototype model with place holders for the differing parts and constructs the correct model. The place holders are substituted with the correct customised LOTOS code for the requested model.

Example 1
This extract from the prototype model

```
hide Out, (*# GATES *), EnvIn, EnvOut in
  TriggerGen [EnvIn, EnvOut]
```

```
    |[EnvIn, EnvOut]|
    FM [EnvOut, EnvIn, Out, (*# GATES *)]
    |[Out, (*# GATES *)]|
    (
(*# BEHAVECOCOON *)
    )
```

is converted into the following LOTOS, *when the script is run to generate a model for 3 features (here 1 to 3):*

```
  hide Out, In1, In2, In3, EnvIn, EnvOut in
    TriggerGen [EnvIn, EnvOut]
    |[EnvIn, EnvOut]|
    FM [EnvOut, EnvIn, Out, In1, In2, In3]
    |[Out, In1, In2, In3]|
    (
      Cocoon [In1, Out] (0, emptystack, 1, 1)
      |[Out]|
      Cocoon [In2, Out] (0, emptystack, 2, 2)
      |[Out]|
      Cocoon [In3, Out] (0, emptystack, 3, 3)
    )
```

4.2 Processes and Data Types

LOTOS is very powerful by combining a process algebra with abstract data types. It furthermore allows the use of processes in a more procedural way by providing a mechanism to initialise and return values on process creation and termination, thus not strictly requiring channels for the interprocess communication. We make use of all these possibilities, and the question why one way has been favoured over another arises. We explain some of our motivation below.

The chosen balance between processes and data types influences the final model significantly. Gotzheim [Got87] analyses different implementations of a stack, ranging from only ADTs to only processes.

From Gotzheim's implementation it can be seen that an implementation using exclusively processes requires separate channels for each operation of the datatype. This is tractable if the datatype only provides a small number of operations, but increases the complexity of synchronisation in the specification significantly when a large number of operations is required. On the other hand, choosing processes allows for a reduction of value errors by disabling operations currently unavailable (.e.g. popping from an empty stack).

An implementation using only ADTs requires to provide ways of handling value errors – i.e. the ADT must provide test functions (e.g. *is_empty_stackS*) and whenever critical operations are used they must be guarded by respective tests. However, this comes at the advantage of reducing synchronisation and channels in the overall model.

4.2.1 Design Decisions

In the model we have decided that the fundamental parts of the system, feature manager, features and the trigger generator are non-terminating processes that communicate via message passing.

Each of these processes is split into several sub-processes which terminate and thus return control to their respective initialising process. Depending on the functionality of the sub-processes, a mixture of message passing via channels (if they are required to request further information from another system component) and value passing (e.g. the solution space is passed around as value) has been chosen.

Considering the issue as to when we have chosen a data type and when a process, we will consider two examples here: the feature behaviour has been implemented as a process, the solution space as a datatype.

The feature behaviour implements a feature (i.e. a state machine). It is essentially a choice between a number of state/message pairs. The process is instantiated by the cocoon and terminates returning a (possibly empty) sequence of messages (the feature response) and a new state. This process does not use any inter-process communication via channels.

The same could have been achieved using a lookup-table data type, but it was decided that this introduces too much additional machinery and therefor hides the resemblance with the state machine description of the feature.

On the other hand, the solution space has been implemented as data type. The major reason for this decision is the envisioned large number of operations required: we need to insert messages, traverse the solution space and furthermore need to remove bad solutions (i.e. those leading to unacceptable behaviour of the system) and finally extract one resolution.

There are several processes involved in the construction of the solution space, all of which communicate via channels with other system components. Adding the extra channels required for the process implementation of the solution space would lead to a less legible model.

Furthermore, the current implementation allows for the solution space to be regarded as first class object which can be passed through the stages of the detection and resolution process.

4.3 Cocoon and Feature Behaviour

A feature is hidden inside a transactional cocoon, in line with Marples approach. In our model there are two processes: the cocoon and the feature behaviour. We will discuss both here.

The cocoon is the process communicating with the feature manager. It provides transactional capabilities and hides the actual feature. All cocoons receive messages that are farmed out by the feature manager in parallel. Depending on the message received, several decisions have to be made. First, is the message destined for the particular cocoon, if so we process it, otherwise we ignore it. Transaction control messages can be aimed at particular cocoons (identified by their "connection number"; not their behaviour). After a commit only particular features shall be put in a new state hence the events are played back to those only. Second, once the relevance of the message is clear, it must be decided whether it is a transaction control message or a message aimed at the feature (i.e. normal call control).

In order to allow for transactional capabilities, Marples uses process forking as provided by the UNIX operating system. Unfortunately, we have to model this. Forking allows Marples to return to a previous state of a feature. To be faithful, we include this in our model by having an explicit stack of previous feature states and a current feature state as local values in the cocoon. Note that this implies that the cocoon is aware of the internal state of the feature, but this does not concern us as it is hidden from the feature manager. Transaction control changes these values depending on the control message:

start transaction The current feature state remains unchanged but is saved onto the stack. In other words we have created a "copy" of the feature. "Killing" the copy of the feature is achieved by popping the saved state.

abort transaction We wish to return to the feature as it was before receiving the first start transaction command, this can be achieved by setting the feature state to the bottom value from the stack and clearing the stack.

commit transaction Committing is the same as aborting, but we must now play back all messages to the chosen features. The feature manager places the cocoon in a special mode were it does not relay responses back to the feature manager and hence triggers no new/further solution space construction.

rollback transaction Rollback allows to return to the previous feature state by popping the topmost value of the stack and using it as new current state of the feature.

Call control messages (non-transaction control messages) lead to the creation of a feature behaviour process. This process is initialised with the current state of the feature (as known by the cocoon) and the trigger event as received from the feature manager. The feature behaviour process terminates returning a possibly empty sequence of responses of the feature.

Once the cocoon has regained control, the sequence of responses is split into individual messages which are sent to the feature manager along a dedicated channel for further processing. The cocoon is now ready to receive a new trigger event.

4.4 Feature Manager

The feature manager receives a message from the environment, then initiates a sequence of other processes, and upon their termination re-instantiates itself.

The received message is passed to a process which constructs the solution space. The resulting tree is passed to the pruning process. The pruning process is conceptually simple: it applies a pruning function which is provided by the tree datatype and returns a message sequence to be committed. Note that though here the difficulty of pruning is hidden, as the data type operations are non-trivial. We come back to this later. Commit finally uses the provided message sequence and farms out the responses to the environment. It also re-instantiates the features to the required states by playing back the messages to them.

Construct sends a start transaction message to the features and then farms out the trigger event. The initial responses are collected and placed in queues. Now all possible combinations of message orders have to be explored, a task carried out by the feedback process. Feedback is the process for the construction of the solution space, we consider it in detail next.

4.4.1 Feedback

The idea behind feedback is to see whether a feature in the system wishes to respond to a message sent out by another feature.

Feedback is required to detect sequential action interactions, which are interactions where the response of a feature triggers another – obviously this can happen many times. Furthermore any combination of shared trigger and sequential action interactions can occur, assuming a number of features react to a fed back message.

For example, a *call forwarding* feature might have responded with *alert Z*. This message might trigger a *call screening* feature, with the purpose of disallowing *alert* messages being send to *Z*.

Recall that the complete solution space has subtrees containing the responses if only one feature is activated, those where 2 features are activated and so on. In order to model this, a control process calls feedback as often as required with the respective number of features enabled. This also ensures that all possible combinations of features are indeed explored.

Example 2 *A sample control process for the feedback with three features involved:*

```
process FeedBackControl[Out, In1, In2, In3](q1, q2, q3: t_msgseq, origt: t_tree): exit (t_tree) :=
(
  FeedBack[Out, In1, In2, In3](q1, q2, q3, emptyststack, origt, 0, true, true, true)
  >> accept t: t_tree in
  FeedBack[Out, In1, In2, In3](q1, q2, emptyseq, emptyststack, t, 0, true, true, false)
  >> accept t: t_tree in
  FeedBack[Out, In1, In2, In3](q1, emptyseq, q3, emptyststack, t, 0, true, false, true)
  >> accept t: t_tree in
  FeedBack[Out, In1, In2, In3](q1, emptyseq, emptyseq, emptyststack, t, 0, true, false, false)
  >> accept t: t_tree in
  FeedBack[Out, In1, In2, In3](emptyseq, q2, q3, emptyststack, t, 0, false, true, true)
  >> accept t: t_tree in
  FeedBack[Out, In1, In2, In3](emptyseq, q2, emptyseq, emptyststack, t, 0, false, true, false)
  >> accept t: t_tree in
  FeedBack[Out, In1, In2, In3](emptyseq, emptyseq, q3, emptyststack, t, 0, false, false, true)
  >> accept t: t_tree in
  exit(t)
)
endproc (* FeedBackControl *)
```

The control process triggers individual feedback processes, which are more interesting. Such a feedback process is initialised with:

- the current response queues, containing the responses to the initial trigger event. Note that those of features that we are not interested in are empty,
- an empty rollback stack,
- the solution space constructed so far,
- the number of already attempted message orderings – here 0 and
- a set of flags indicating which features are currently activated.

There are 2 main cases: all the response queues are empty, i.e. their accumulated length is 0, or at least one is not empty (the accumulated length > 0). If all queues passed to feedback are empty, we did not have any responses to the last stimulus and all previously received responses have been fed back. The consequence is that we have reached the end of a branch in the tree – a possible solution is complete. There are 2 possible continuations, depending on the rollback stack: the stack is empty, meaning that we have finished as all possibilities have been explored. Or, the stack is not empty, indicating that there might be more choices earlier in the tree – which we explore by rolling back one level and recursing on the feedback process. The parameters are now:

- the response queues, as they were saved last on the stack,
- the rollback stack with the topmost element popped,
- the solution space constructed so far with the insertion point moved (we need to start a new branch),
- the number of previously tried message orderings (popped from stack) – the parameter $chosen$,
- the flags as before.

In the case that at least one queue is not empty, it has to be determined whether all possible orders have been attempted – a comparison of the number of non-empty queues and the parameter $chosen$. The two cases are: 1) $chosen$ is equal to the number of non-empty queues or 2) chosen is less than the number of non-empty queues.

In the first case we have tried all choices at this level and can simply rollback one level, as done previously for empty queues and a non-empty stack. The second case is more interesting as here messages are actually fed back to the features and further responses are collected.

The cocoons must be informed that the local states need to be backed up, i.e. a further exploration shall take place but we require the option to revert back to the current state. This is achieved by farming out the start transaction message. More than one queue might contain a message and hence we have to select one to be farmed out. There are two complications: which queue do we chose the message from, as finally all possible orders must be explored and what happens when 2 queues have the same message at their front. The parameter $chosen$ can be used to identify the queue we select the message from systematically. Note that $chosen$ ranges from 0 to $no\ of\ non-empty\ queues-1$, hence the $chosen+1$ non-empty queue is the candidate. Once a message queue is identified, the first message is removed and stored. The other queues are analysed to see whether they contain the same message at their front – if so the message is also stripped from them. This is justified by the fact that we do not want the user to receive a message twice, as was explained when the overlap was introduced in the initial model.

Finally, the selected message is farmed out to the features and the responses are collected. It is now necessary to empty all queues of features that we are currently not interested in (e.g. we might be constructing a branch with just 2 out of 5 features active). We now recurse on feedback with:

- the gathered responses appended to the possibly changed previous response queues,

- the initial state (queues and *chosen*) saved onto the rollback stack,
- the selected message inserted into the solution space,
- as we start a new level, *chosen* is reinitialised to 0 – we have not tried any possibilities yet, and
- the flags as before.

In this way, once all feedback processes have terminated, a tree containing the complete solution space has been generated. Note that this tree might contain several duplicate branches: assume 3 features (F_1, F_2, F_3) 2 of which (F_1, F_2) respond to a trigger. It is to be expected that a branch with all three features active is identical to one with feature F_3 disabled.

4.4.2 Storing the Solution Space: A Tree

The solution space has been modelled as a tree datatype. The nodes contain a message and a source of the message. The source is required for commit and some pruning techniques. A message with source 0 has been received from the environment, one with a value greater than 0 from a feature. The strictly positive value identifies the feature, where for example 3 means the feature responding on channel $In3$ and whose messages have been stored in queue 3 by the feature manager. The root node contains the initial trigger event, which always has source 0. Further each node is associated a list of children, the number of which we do not know in advance as it depends on the number of responding features.

mktree is the constructor for trees, it takes a node and a list of trees and returns a tree. Usually we wish to hide this constructor behind a simple insert function, just requiring a tree and the data to be inserted. In case of a binary search tree this usually searches the correct insertion point based on some ordering criteria (e.g. alphabetical). We want a similar solution, but we cannot provide such an order criteria.

When inserting a new message we usually wish to insert it after the last inserted message, or in case that we have performed a roll back we are required to insert the message as new branch of the parent of the last inserted message. Have we rolled back several levels, we have to insert a new branch at a respective ancestor node.

In an imperative solution this would be relatively easy to handle: we keep a pointer pointing to the root of the tree and have a further insertion pointer. Upon a new insertion this is moved to become a child of the inserted node, upon roll back it is moved up the tree. However, in Act One we do not have pointers and thus must find an alternative solution. There are three candidates: keeping track of the path to the next insertion point and storing this path along with the tree, identifying an enumeration of nodes that clearly identifies a location, and using a marker for the insertion point.

Keeping track of the path to an insertion point allows for quick retrieval of the point, but requires a separate structure to be kept alongside the tree and thus is not very space efficient. Enumeration of locations is better as it will only require a single number to be kept in addition to the tree, however no simple enumeration system could be identified. Our chosen solution is the last one and a dedicated node called “marker” was introduced. This node always occurs as leaf and furthermore will always be the rightmost leaf.

Let us consider this approach in more detail: suppose a new node is inserted into a tree then it does not have any children, as we have not yet explored the solution further. The next message in the same solution will be a child of the current node, here the only one and thus the left and rightmost one. Have we reached the end of a solution we rollback one level and want to insert a new subtree with the same parent as the last inserted node. This leads to the addition of a new tree in the treelist of the parent, and the implementation of the treelist always adds new trees to the right, hence the marker should now form this rightmost tree in the list. If we move further up the tree we can ever only add branches to the right.

Following this, insert has to search for the marker and insert the new node instead of the marker. The new insertion point is a child of the just inserted node. There are three cases:

- an empty tree: we just insert the node as root node, the marker is a child of this node,
- a non-empty tree with the rightmost subtree of the root node being the marker: we insert the node replacing the marker, the new marker is a child of this node,
- a non-empty tree with the rightmost subtree of the root node not being the marker: we recursively insert the node into the rightmost subtree.

When we apply rollback in our algorithm, no new node is inserted into the tree, but it is required to shift the insertion point. An operation *movemarker* is provided by the datatype. The current position of the marker is identified and it is moved to become the rightmost child of its current grandparent. We distinguish the following cases:

- an empty tree: there is no marker to move,
- a tree containing just one node (marker not counted): there is no grandparent, so we cannot move the marker,
- a tree containing two nodes (marker not counted): the marker is moved to become a new subtree of the root node,
- a tree containing more than two nodes: move the marker recursively within the rightmost subtree.

The two operations and the constructor is all that is required to construct the solution space (the datatype makes use of some query functions to determine whether a tree is empty or a node is the marker internally). We will add further operations to the tree later, when we consider pruning of the solution space.

5 Experimental Results

In the previous sections we have explained the construction of the solution space. A number of experiments has been conducted and we show the resulting solution spaces for a number of examples in Fig. 3. The features for the examples are implementations of those features presented for the 2nd Feature Interaction detection Contest [KMMR00].

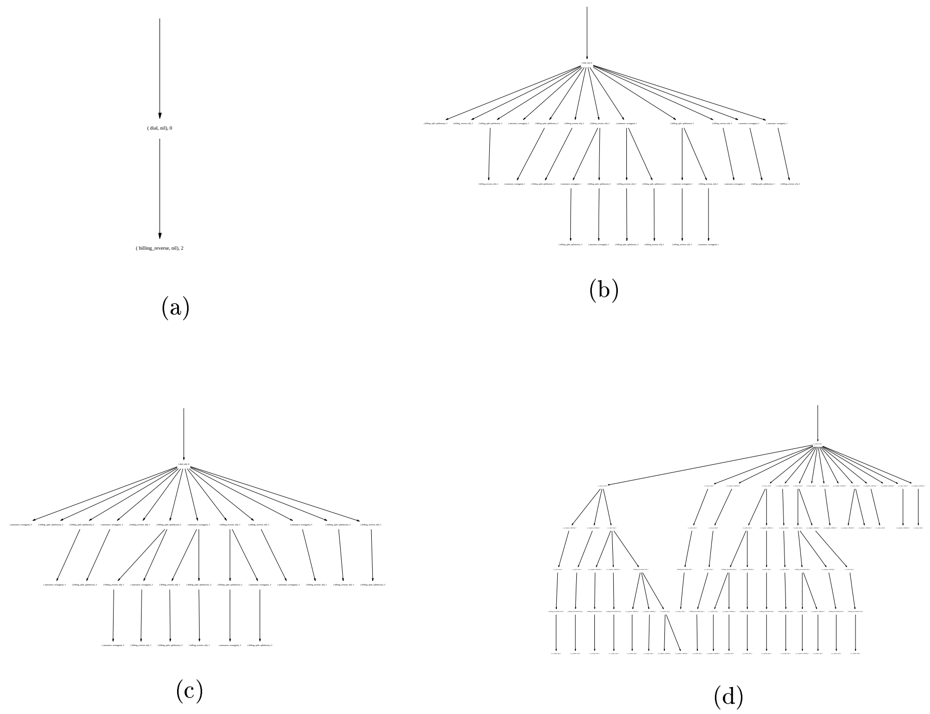

Figure 3: Solution Space and Resolutions for Enhanced Features

Example 3 (2 Features – No Interaction)
The Call Forwarding on Busy and Reverse Charging features are presented with a dial trigger event. Both features are in their initial state. Only Reverse Charging Reacts to this event. Upon feedback no further reaction is triggered, hence the two feature do not interact.

Example 4 (3 Features – Interaction)
The Teenline, Reverse Charging and Split Billing features are presented with a dial trigger event. All three features react to the trigger (Teenline was in state 1, the other two were in their initial states), thus an interaction is detected and a number of possible interleavings are explored.

Example 5 (4 Features – Interaction)
The Teenline, Reverse Charging and Split Billing features as before are combined with Call Forwarding on Busy. The system is presented with a dial trigger event. Three features react (again Teenline was in state 1, the other features were in their initial states), thus an interaction is detected and a number of possible interleavings are explored. The solution space offers the same solutions as in Example 4, however the slightly different appearance is caused by a different order of the features.

Example 6 (8 Features – Interaction)
8 Features – Teenline, Call Forwarding on Busy, Terminating Call Screening, Reverse Charging, Split Billing, Basic Call, Calling Number Deliver and Call Transfer are present in the system. The trigger event is i_alert. All features apart from Teenline are in their initial states, Teenline is in state 1. The resulting solution space indicates a number of shared trigger and sequential action interactions.

The solution spaces have been presented graphically, the output from Lotos is less immediate. For example 1 the output is as follows:

```
TGen ! message(rcv, dial, 0, 0, nil)
full tree: ! mktree(mknode(message(rcv, dial, 0, 0, nil), 0),
            insl(insl(emptylist, emptytree),
            mktree(mknode(message(snd, billing_reverse, 0, 0, nil),
            succ(succ(0))), emptylist)))
```

Comparing the runtime [2] behaviour and the size of the model for the above examples, we can see that both increase with the an increasing number of features. The following table shows the figures for the above examples:

	Source Code in Lines	Runtime
Example 3	2577	< 1 sec
Example 4	2614	7 secs
Example 5	2667	9 secs
Example 6	3712	43 min, 14 secs

The experiments show, that it is not practical to run a large number of experiments. It was expected that the construction of an individual solution space is time consuming. The figures obtained here are very unlikely to support that this approach can indeed be used at runtime [3]. However, results are the expected ones, thus providing some evidence that the implementation is correct.

6 Thoughts on Resolution

The previous sections provide a background on which a theory of resolutions can be developed. Obviously the elimination of bad solutions and choice of the best resolution is required.

Two approaches can be envisioned to achieve this. The first is straight forward: the whole solution space is constructed, then the rules are applied and the solution space is reduced until only one resolution is left. This approach is referred to as "Construct-Prune" and was described in sections 5.7 and 5.8. This is highly complex in both space and time requirements, as it requires the construction of a large number of branches that will be thrown away. Due to the fact that the approach is runtime, complexity is an important issue.

The second approach overcomes this problem by using an "on-the-fly" idea, were the solution space is pruned while it is being constructed.

Basically, on-the-fly resolution will work by trying to identify bad solutions while constructing a solution. As soon as a solution can be seen to be bad (independent of potential further behaviour), the construction will be aborted, the already constructed part removed and

[2] Measured on SPARCstation 20 running SunOS 5.5.1

[3] Runtime results obtained using a Haskell model and also using the Desk [MTMS95] testbed are much better

construction of another resolution will be attempted. This will largely reduce the complexity if many features are involved and bad resolutions can be identified early in the construction.

For both approaches a method of describing bad solutions is required. We have decided to describe the behaviour using rules rules, where a rule describes a sequence of messages that should never occur. Thus the problem of resolving detected interactions is tranformed to the problem of identifying suitable rules – the application of which is fairly trivial. Rules can take on different forms and here only an overview is given. Three classes of rules can be distinguished: message dependent rules, message independent rules and examples. Examples describe concrete good or bad behaviour and are considered as a last resort.

Message independent rules can include concepts such as priorities of features and the satisfaction of the maximal number of features. A feature is considered to be satisfied when its intended behaviour can be exhibited, and clearly it is best to chose a resolution that satisfies the largest number of features. These rules will be useful for extracting one resolution.

Message dependent rules can be seen to be the most powerful, but also those requiring the most information. In order for message dependent rules to be useful, a semantics of the messages is required. This allows to develop relations on messages, such as a class of treatments, or a class of billing messages. This understanding provides us with the required knowledge to build grammars describing good or bad behaviour, for example "a treatment following an onhook message is not useful unless there was an offhook in between".

Three message independent rules have been implemented, they are all functions on the tree data type of the form $f : t\_tree \rightarrow t\_tree$. The functions are *extract_remduplicates*, *rem_shortest* and *rem_least*. *extract_remduplicates* removes duplicate branches (the reason for their occurrence had been discussed earlier at the end of section 4.6.1). *rem_shortest* returns only the longest branch in the tree, in case there is a choice of a number of equally long branches the left most one is returned. *rem_least* returns the left most branch satisfying the largest number of features.

The implementation of these functions is non-trivial due to the complexity of the tree data type, so we will not go into detail here.

Considering the implementation of message dependent rules leads to a rather different picture. The rules describing unacceptable behaviour are of the form "onhook followed by onhook with no offhook occurring in between is bad". We can express all those rules using regular expression over an alphabet of messages. However, a complete mechanism for describing regular expressions and implementation of respective matching algorithms (usually involving either nondeterministic or deterministic finite state automata) is required. The algorithms for matching regular expressions should be called from a pruning function on the tree data type, thus they should be implemented in ActOne – a major effort that introduces doubts as to whether LOTOS is the right formalism.

7 LOTOS – The Right Choice?

Due to the complex nature of several of the data types and the extensive use of data types we begin to question whether LOTOS is the best choice. We will analyse the pro's and con's in more detail.

Describing the process structure in LOTOS is very natural. Implementing data types does not cause major problems. However, more complex data types with relatively complex operations are very tedious to build. The solution space clearly is such a datatype. Further, enumeration data types (i.e. those that cannot be implemented recursively, like events) are not easily available. Nevertheless, as it is possible to implement the complex data types at a trade off of simplicity of the specification we need to consider what we want to achieve with the model.

Lotos has strong tool support, so it could be assumed that the extra effort in developing the model can be justified by having these tools available. The tools cover different areas, here simulation and verification are of most interest.

Indeed, very good experience was gained using LOLA for simulation of the model. Many bugs could be detected and resolved. A very nice feature is the rewrite mechanism for operations on abstract data types, allowing to debug and gain a better intuition. However, the complexity of the underlying data types and algorithms make step wise simulation impractical when considering more realistic examples.

TOPO allows one to generate prototypes that can then simply be compiled using a C compiler and executed. Special statements allow for data to be printed. This facility has been used to gather the experimental data. The trigger generator prints the sent message, the feature manager outputs the respective generated solution space. The output of the solution space is simply the tree constructors applied to the provided data, e.g. "mktree(n, emptytreelist)" would be valid output.

Testing and simulation form only a part of the possibilities a formal model provides. An attempt was made to use CADP to check for deadlock. However, the process of generating the state space was extremely slow and finally failed on the size of the model. At this point it was concluded that we need to simplify the model. A simplification of the model is not acceptable, as we would lose too many interesting properties in the abstraction. So we can conclude that CADP is not suitable for the current task.

LOLA allows for tester processes to be synchronised with the model and for weak bisimilarity to be observed. A tester process could be a model generating the solution space not using feedback, but rather hard coded interleavings. However, the advantage of this approach over simulation is not clear in our case, as the solution space depends on the currently activated features and the trigger message.

What do we want to prove? became a very important question. It turns out that we wish to reason about the solution space and the resolutions, i.e. *data*. Properties include that the solution space should contain all possible traces that can be obtained from the features being executed concurrently. Any trace in the solution space should be able to be simulated by movements of the feature automata. As for resolutions, a resolution should be a solution that does not violate any resolution rules.

The LOTOS tools, being process algebra tools, mainly allow reasoning about processes. Data is often secondary in this context. In our approach however, data is essential and we require reasoning over data.

We conclude that LOTOS seemed a natural choice for modelling our processes. LOTOS proved useful for gaining a better understanding of the system during simulation. However, the data intensity of our model strongly suggests to abandon LOTOS and find a more suitable formalism to deal with data.

7.1 LoLA and Topo under Linux

Note that LoLA and Topo are not available for Linux, however the source code requires only a few changes and then a version suitable for Linux can be compiled. Interestingly, the generated prototypes can be compiled on Linux, but cannot be executed (they simply crash). However they can be compiled and run successfully on Solaris. The cause is not obvious and has not been investigated further.

8 Summary and Further Work

The ideas underpinning the online feature manager have been presented and a Lotos model of the feature manager was developed. A number of shortcomings with respect to Lotos and our approach have been identified, notably that Lotos, despite providing a abstract data type definition language, is not very suitable for complex data types. However, the simulation tools available helped to gain a better understanding of the model.

Further work includes attempting to use a different formalism for the model, in particular a functional programming language. The language Haskell was chosen and the Lotos model has been converted and extended with a pruning mechanism. The data type definitions are far simpler than their Lotos equivalent, however it should be noted that the understanding gained from developing the Lotos model was essential in developing the functional language equivalent of the process part.

It still remains to be shown that the construction is indeed correct. Furthermore a more complete set of pruning rules has to be identified from case studies.

Acknowledgments

I would like to thank Muffy Calder for guidance while using Lotos and also for advising on style, contents and language during the writing of this report.

References

[CAD] CADP. (caesar/aledebaran development package).

[CM00] M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI.* IOS Press (Amsterdam), May 2000.

[CMM99] M. Calder, E. Magill, and D. Marples. A hybrid approach to software interworking problems: Managing interactions between legacy and evolving telecommunications software. *IEE Proceedings - Software*, 146(3):167–180, June 1999.

[CO95] K. E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications Systems III.* IOS Press (Amsterdam), October 1995.

[CR00] M. Calder and S. Reiff. Modelling legacy telecommunications switching systems for interaction analysis. In Peter Henderson, editor, *Systems Engineering Business Process Change*, pages 182–195. Springer Verlag, London, May 2000.

[Got87] R. Gotzheim. Specifying abstract data types with lotos. *Protocol Specification, Testing and Verification*, VI:15–26, 1987.

[ISO89] ISO. *ISO standard 8807:* LOTOS – *A formal description technique based on the temporal ordering of observational behaviour.* ISO/IEC, 1989.

[KB98] K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V.* IOS Press (Amsterdam), September 1998.

[KMMR00] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Second feature interaction contest. In *[CM00]*, pages 293–310, May 2000.

[LOL] LOLA. (LOtos LAboratory), `http://yeti.dit.upm.es/ lotos/tools/lola.html`.

[Mar00] D. Marples. *Detection and Resolution in of Feature Interactions in Telecommunications Systems at Runtime.* PhD Thesis, Communications Division, Department of Electrical and Electronic Engineering, University of Strathclyde, 2000.

[MM98] D. Marples and E. H. Magill. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In *[KB98]*, pages 115–134, September 1998.

[MTMS95] D. Marples, S. Tsang, E. H. Magill, and D. G. Smith. A platform for modelling feature interaction detection and resolution techniques. In *[CO95]*, pages 185–199, October 1995.

[Rei00] S. Reiff. Identifying resolution choices for an online feature manager. In *[CM00]*, pages 113–128, May 2000.

A Lotos Prototype

```
*(*
#TEMPLATE
#TEMPLATE Version 1.3 of the telephone switch with feature manager
#TEMPLATE Stephan Reiff
#TEMPLATE Started: 17-07-2000
#TEMPLATE Last Change: 06-10-2000
#TEMPLATE
#TEMPLATE This is a template.
#TEMPLATE To obtain Lotos Code, run fillgaps.pl
#TEMPLATE
#TEMPLATE Changes over previous version:
#TEMPLATE * Feedback is completed to work on n features (new recursive definition)
#TEMPLATE * Tree is implemented
#TEMPLATE
#TEMPLATE currently dest field of messages is abused for control: 0: broadcasted msg, n > 0: feat id
#TEMPLATE       src field is unused
#TEMPLATE       so we really use the event, ioasp and arguments of the messages
#TEMPLATE Change 06-10-2000: BCS added as Feature 8
#TEMPLATE                  GetChosenMsg: Call to RemFromAllQueues
#TEMPLATE Change 25-01-2001: all features adapted to new state machines in thesis
#TEMPLATE                  complete set of features implemented
#TEMPLATE
*)

specification System [EnvOut]: noexit

(*
 boolean and natural Number types are pasted from bool_nat.lot library
 Constants 1 - 10 in nat have been added 21-08-2000
 subtraction operator for nat       added 06-09-2000
 division operator for nat          added 16-10-2000
*)
  type boolean is
    sorts
      bool               x iff y = x implies y and (y implies x);
        x eq y = x iff y;     x ne y = x xor y;
        true and x = x;       false and x = false;
        true or x = true;     false or x = x;
    opns
      true       (*! constructor *) : -> bool
      false      (*! constructor *) : -> bool
      not                           : bool -> bool
      _and_, _or_, _xor_,
      _implies_, _iff_, _eq_, _ne_ : bool, bool -> bool
    eqns
      forall x, y: bool
      ofsort bool
        not(true) = false;    not(false) = true;        x implies y = y or not(x);
        x and true = x;       x and false = false;      x iff y = x implies y and (y implies x);
        x or true = true;     x or false = x;           x eq y = x iff y;      x ne y = x xor y;
        x xor y = x and not(y) or (y and not(x));       true and x = x;        false and x = false;
  endtype

  type basicnaturalnumber is
    sorts
      nat
    opns
      0    (*! constructor *) : -> nat
      succ (*! constructor *) : nat -> nat
      _+_, _*_, _**_          : nat, nat -> nat
    eqns
      forall m, n: nat
      ofsort nat
        m + 0 = m;            m + succ(n) = succ(m) + n;
        m * 0 = 0;            m * succ(n) = m + (m * n);
        m ** 0 = succ(0);     m ** succ(n) = m * (m ** n);
  endtype

  type naturalnumber is
    basicnaturalnumber,boolean
    opns
      _eq_, _ne_, _lt_, _le_, _ge_, _gt_         : nat, nat -> bool
      _-_, _/_                                   : nat, nat -> nat
       1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
      11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
      21, 22, 23, 24, 25, 26, 27, 28, 29, 30    : -> nat
    eqns
      forall m, n: nat
      ofsort bool
        0 eq 0 = true;                0 eq succ(m) = false;
        succ(m) eq 0 = false;         succ(m) eq succ(n) = m eq n;
        m ne n = not(m eq n);
        0 lt 0 = false;               0 lt succ(n) = true;
        succ(n) lt 0 = false;         succ(m) lt succ(n) = m lt n;
        m le n = m lt n or (m eq n);
        m ge n = not(m lt n);         m gt n = not(m le n);
      ofsort nat
        1 = succ(0);        11 = succ(10);        21 = succ(20);
        2 = succ(1);        12 = succ(11);        22 = succ(21);
        3 = succ(2);        13 = succ(12);        23 = succ(22);
        4 = succ(3);        14 = succ(13);        24 = succ(23);
```

```
      5 = succ(4);        15 = succ(14);        25 = succ(24);
      6 = succ(5);        16 = succ(15);        26 = succ(25);
      7 = succ(6);        17 = succ(16);        27 = succ(26);
      8 = succ(7);        18 = succ(17);        28 = succ(27);
      9 = succ(8);        19 = succ(18);        29 = succ(28);
      10 = succ(9);       20 = succ(19);        30 = succ(29);
      0 - m = 0;     (* dodgy but avoids undefined *)          m - 0 = m;
      succ(m) - succ(n) = m - n;
      m / 0 = 0;     (* dodgy but avoids undefined *)          0 / m = 0;
      m gt n => m / n = 1 + ((m - n) / n);                     m eq n => m / n = 1;
      m lt n => m / n = 0;
  endtype

(*
 t_event defines: - constants for all events in the system
                  - eq and neq to compare events
 there is always a finite number of events (application domain restrcition)
*)
  type t_event is
    boolean
    sorts
      t_event
    opns
      (* call control *)
      onhook              (*! constructor *) : -> t_event    dial        (*! constructor *) : -> t_event
      offhook             (*! constructor *) : -> t_event    flash       (*! constructor *) : -> t_event

      announce            (*! constructor *) : -> t_event    connect     (*! constructor *) : -> t_event
      display             (*! constructor *) : -> t_event    stopalert   (*! constructor *) : -> t_event
      ringtone            (*! constructor *) : -> t_event    alert       (*! constructor *) : -> t_event
      busytone            (*! constructor *) : -> t_event    cwtone      (*! constructor *) : -> t_event
      dialtone            (*! constructor *) : -> t_event    storemsg    (*! constructor *) : -> t_event
      timeouttone         (*! constructor *) : -> t_event    storeread   (*! constructor *) : -> t_event
      disconnecttone      (*! constructor *) : -> t_event    storeclear  (*! constructor *) : -> t_event

      o_notify            (*! constructor *) : -> t_event    o_connect     (*! constructor *) : -> t_event
      i_notify            (*! constructor *) : -> t_event    i_connect     (*! constructor *) : -> t_event
      o_inform            (*! constructor *) : -> t_event    o_disconnect  (*! constructor *) : -> t_event
      i_inform            (*! constructor *) : -> t_event    i_disconnect  (*! constructor *) : -> t_event
      o_alert             (*! constructor *) : -> t_event    o_timeout     (*! constructor *) : -> t_event
      i_alert             (*! constructor *) : -> t_event    i_timeout     (*! constructor *) : -> t_event
      o_free              (*! constructor *) : -> t_event    o_msg         (*! constructor *) : -> t_event
      i_free              (*! constructor *) : -> t_event    i_msg         (*! constructor *) : -> t_event
      o_busy              (*! constructor *) : -> t_event    o_request     (*! constructor *) : -> t_event
      i_busy              (*! constructor *) : -> t_event    i_request     (*! constructor *) : -> t_event
      o_stopalert         (*! constructor *) : -> t_event
      i_stopalert         (*! constructor *) : -> t_event

      (* billing control *)
      billing_offhook     (*! constructor *) : -> t_event      billing_split  (*! constructor *) : -> t_event
      billing_onhook      (*! constructor *) : -> t_event      billing_start  (*! constructor *) : -> t_event
      billing_forwarded   (*! constructor *) : -> t_event      billing_stop   (*! constructor *) : -> t_event
      billing_reverse     (*! constructor *) : -> t_event

      (* transaction control *)
      rollback_transaction (*! constructor *) : -> t_event
      commit_transaction   (*! constructor *) : -> t_event
      abort_transaction    (*! constructor *) : -> t_event
      start_transaction    (*! constructor *) : -> t_event
      respcomplete         (*! constructor *) : -> t_event
      (* equality tests for events *)
      _eq_ , _neq_                             : t_event , t_event -> bool
    eqns
      forall a, b: t_event
      ofsort bool
        onhook eq onhook = true;                      onhook eq a = false;
        offhook eq offhook = true;                    offhook eq a = false;
        dial eq dial = true;                          dial eq a = false;
        flash eq flash = true;                        flash eq a = false;

        dialtone eq dialtone = true;                  dialtone eq a = false;
        ringtone eq ringtone = true;                  ringtone eq a = false;
        busytone eq busytone = true;                  busytone eq a = false;
        timeouttone eq timeouttone = true             timeouttone eq a = false;
        disconnecttone eq disconnecttone = true;      disconnecttone eq a = false;
        connect eq connect = true;                    connect eq a = false;
        stopalert eq stopalert = true;                stopalert eq a = false;
        alert eq alert = true;                        alert eq a = false;
        announce eq announce = true;                  announce eq a = false;
        display eq display = true;                    display eq a = false;
        cwtone eq cwtone = true;                      cwtone eq a = false;
        storemsg eq storemsg = true;                  storemsg eq a = false;
        storeread eq storeread = true;                storeread eq a = false;
        storeclear eq storeclear = true;              storeclear eq a = false;

        o_notify eq o_notify = true;                  o_notify eq a = false;
        i_notify eq i_notify = true;                  i_notify eq a = false;
        o_inform eq o_inform = true;                  o_inform eq a = false;
        i_inform eq i_inform = true;                  i_inform eq a = false;
        o_alert eq o_alert = true;                    o_alert eq a = false;
        i_alert eq i_alert = true;                    i_alert eq a = false;
        o_stopalert eq o_stopalert = true;            o_stopalert eq a = false;
        i_stopalert eq i_stopalert = true;            i_stopalert eq a = false;
        o_free eq o_free = true;                      o_free eq a = false;
```

```
        i_free  eq  i_free  = true;                   i_free  eq a = false;
        o_busy eq o_busy = true;                      o_busy eq a = false;
        i_busy eq i_busy = true;                      i_busy eq a = false;
        o_connect eq o_connect = true;                o_connect eq a = false;
        i_connect eq i_connect = true;                i_connect eq a = false;
        o_disconnect eq o_disconnect = true;          o_disconnect eq a = false;
        i_disconnect eq i_disconnect = true;          i_disconnect eq a = false;
        o_timeout eq o_timeout = true;                o_timeout eq a = false;
        i_timeout eq i_timeout = true;                i_timeout eq a = false;
        o_msg eq o_msg = true;                        o_msg eq a = false;
        i_msg eq i_msg = true;                        i_msg eq a = false;
        o_request eq o_request = true;                o_request eq a = false;
        i_request eq i_request = true;                i_request eq a = false;

        billing_offhook  eq  billing_offhook  = true;       billing_offhook  eq a = false;
        billing_onhook eq billing_onhook = true;            billing_onhook eq  a = false;
        billing_forwarded  eq billing_forwarded = true;     billing_forwarded  eq a = false;
        billing_reverse  eq  billing_reverse  = true;       billing_reverse  eq  a = false;
        billing_split  eq  billing_split  = true;           billing_split  eq  a = false;
        billing_start  eq  billing_start  = true;           billing_start  eq a = false;
        billing_stop  eq  billing_stop  = true;             billing_stop  eq  a = false;

        rollback_transaction  eq rollback_transaction  = true;  rollback_transaction  eq a = false;
        commit_transaction eq commit_transaction = true;        commit_transaction eq a = false;
        abort_transaction eq abort_transaction = true;          abort_transaction eq a = false;
        start_transaction  eq start_transaction  = true;        start_transaction  eq a = false;

        respcomplete eq respcomplete = true;          respcomplete eq a = false;

        a neq b = not(a eq b)
  endtype

(*
 t_arg  defines  arguments for messages as constants
*)
  type t_arg is
    boolean
    sorts
      t_arg
    opns
      nil          (*! constructor *) : -> t_arg      nomsg        (*! constructor *) : -> t_arg
      askpin       (*! constructor *) : -> t_arg      msgtext      (*! constructor *) : -> t_arg
      wrongpin     (*! constructor *) : -> t_arg      callminder   (*! constructor *) : -> t_arg
      time         (*! constructor *) : -> t_arg      id           (*! constructor *) : -> t_arg
      screened     (*! constructor *) : -> t_arg      ringback     (*! constructor *) : -> t_arg
      splitfactor  (*! constructor *) : -> t_arg      fastalert    (*! constructor *) : -> t_arg
      callerid     (*! constructor *) : -> t_arg      cwhold       (*! constructor *) : -> t_arg
      barred       (*! constructor *) : -> t_arg      originator   (*! constructor *) : -> t_arg
      _eq_                            : t_arg , t_arg -> bool
    eqns
      forall  a: t_arg
      ofsort bool
        nil eq nil = true;                    nil eq a = false;
        askpin eq askpin = true;              askpin eq a = false;
        wrongpin eq wrongpin = true;          wrongpin eq a = false;
        time eq time = true;                  time eq a = false;
        screened eq screened = true;          screened eq a = false;
        splitfactor  eq  splitfactor  = true;     splitfactor  eq a = false;
        callerid  eq  callerid  = true;       callerid  eq a = false;
        barred eq barred = true;              barred eq a = false;
        nomsg eq nomsg = true;                nomsg eq a = false;
        msgtext eq msgtext = true;            msgtext eq a = false;
        callminder eq callminder = true;      callminder eq a = false;
        id eq id = true;                      id  eq a = false;
        callerid  eq  callerid  = true;       callerid  eq a = false;
        ringback eq ringback = true;          ringback eq a = false;
        fastalert  eq  fastalert  = true;     fastalert  eq a = false;
        cwhold eq cwhold = true;              cwhold eq a = false;
        originator  eq  originator  = true;   originator  eq a = false;
  endtype

(*
 t_io  defines  the  possible  values  for  ioaspects  and  comparision functions
   - input:  rcv
   - output: snd
   - eq, neq
*)
  type t_io is
    boolean
    sorts
      t_io
    opns
      rcv    (*! constructor *) : -> t_io
      snd    (*! constructor *) : -> t_io
      _eq_ , _neq_              : t_io , t_io -> bool
    eqns
      forall  a, b: t_io
      ofsort bool
        rcv eq rcv = true;        rcv eq snd = false;
        snd eq rcv = false;       snd eq snd = true;
        a neq b = not(a eq b)
  endtype

(*
```

```
 t_msg is the data type for messages: messages are basically Quintuples.
 the data type provides
        - a constructor function (message)
           - a constant respcomplete
           - projection functions for extracting the parts of the quintuple
           - partitioning functions (Control messages and Feature messages)
           - equalities: equal, event equal and inverse
*)
  type t_msg is
    boolean,t_event,naturalnumber,t_arg,t_io
    sorts
      t_msg
    opns
      message        (*! constructor *) :  t_io , t_event , nat, nat, t_arg -> t_msg
      p_ioasp                           : t_msg -> t_io
      p_event                           : t_msg -> t_event
      p_source, p_dest                  : t_msg -> nat
      p_argument                        : t_msg -> t_arg
      is_fmsg , is_tcmsg , is_rcomp     : t_msg -> bool
      are_eventequal, are_equal,
       are_inverse                      : t_msg, t_msg -> bool
    eqns
      forall  e: t_event ,
              s , d: nat,
              io : t_io ,
              a: t_arg ,
              ms, ms1: t_msg
      ofsort t_event
        p_event(message(io, e, s, d, a)) = e;
      ofsort nat
        p_source(message(io, e, s, d, a)) = s;
        p_dest(message(io, e, s, d, a)) = d;
      ofsort t_io
        p_ioasp  (message(io, e, s, d, a)) = io;
      ofsort t_arg
        p_argument(message(io, e, s, d, a)) = a;
      ofsort bool
        is_fmsg(ms) = not(is_tcmsg(ms));
        is_tcmsg(ms) = ((p_event(ms) eq rollback_transaction) or
                        (p_event(ms) eq start_transaction ) or
                        (p_event(ms) eq commit_transaction) or
                        (p_event(ms) eq abort_transaction));
        are_eventequal(ms1, ms) = (p_event(ms) eq p_event(ms1));
        are_equal(ms1, ms) = ((p_event(ms) eq p_event(ms1)) and (p_ioasp(ms) eq p_ioasp(ms1)));
         are_inverse (ms1, ms) = ((p_event(ms) eq p_event(ms1)) and (not(p_ioasp(ms) eq p_ioasp(ms1))));
        (p_event(ms) eq respcomplete) => is_rcomp(ms) = true;
        not(p_event(ms) eq respcomplete) => is_rcomp(ms) = false;
  endtype

(*
 t_msgseq defines a sequence of messages
*)
  type t_msgseq is
    boolean,t_msg, naturalnumber
    sorts
      t_msgseq
    opns
      emptyseq (*! constructor *) : -> t_msgseq
      is_empty                    : t_msgseq -> bool
       first                      : t_msgseq -> t_msg
      remove                      : t_msgseq -> t_msgseq
      add      (*! constructor *) : t_msgseq, t_msg -> t_msgseq
      length                      : t_msgseq -> nat
      concat                      : t_msgseq, t_msgseq -> t_msgseq
       possiblechoices            : t_msgseq, nat -> nat
    eqns
      forall  m: t_msg,
              ms, ms1: t_msgseq,
              n: nat
      ofsort bool
        is_empty(emptyseq) = true;
        is_empty(add(ms, m)) = false;
      ofsort t_msg
         first (add(emptyseq, m)) = m;
        not(is_empty(ms)) => first (add(ms, m)) = first(ms);
      ofsort t_msgseq
        remove(add(emptyseq, m)) = emptyseq;
        not(is_empty(ms)) => remove(add(ms, m))= add(remove(ms), m);
      ofsort nat
        length(emptyseq) = 0;
        length(add(ms, m)) = succ(length(ms));
         possiblechoices (emptyseq, n) = n;
        not(is_empty(ms1)) => possiblechoices(ms1, n) = succ(n);
      ofsort t_msgseq
        concat(ms, emptyseq) = ms;
        not(is_empty(ms1)) => concat(ms, ms1) = concat(add(ms, first(ms1)), remove(ms1));
  endtype

  type t_qlist  is
    boolean,t_msgseq
    sorts
       t_qlist
    opns
      emptyqlist  (*! constructor *) : ->  t_qlist
```

```
      is_empty                        :  t_qlist  -> bool
       getfirst                       :  t_qlist  -> t_msgseq
      remfirst                        :  t_qlist  -> t_qlist
      add          (*! constructor *) :  t_qlist , t_msgseq -> t_qlist
    eqns
      forall ms, ms1: t_msgseq,
             ql :  t_qlist
      ofsort bool
        is_empty(emptyqlist) = true;
        is_empty(add(ql, ms)) = false;
      ofsort t_msgseq
         getfirst (add(emptyqlist, ms)) = ms;
        not(is_empty(ql)) => getfirst  (add(ql, ms)) = getfirst (ql );
      ofsort  t_qlist
        remfirst (add(emptyqlist, ms)) = emptyqlist;
        not(is_empty(ql)) => remfirst(add(ql, ms))= add(remfirst(ql), ms);
  endtype


  type t_natset is
    naturalnumber, boolean
    sorts
      t_natset
    opns
      emptyset     (*! constructor *) : -> t_natset
       is_inset                        : t_natset , nat -> bool
      is_emptyset                      : t_natset -> bool
      insert       (*! constructor *) : t_natset , nat -> t_natset
      insertelem                       : t_natset , nat -> t_natset
      union                            : t_natset , t_natset -> t_natset
      length                           : t_natset -> nat
    eqns
      forall n, m: nat,
             ns, ns1: t_natset
      ofsort bool
        is_emptyset(emptyset) = true;
        is_emptyset( insert (emptyset, n)) = false ;
         is_inset (emptyset, n) = false ;
         is_inset ( insert (emptyset, n), n) = true;
        (n eq m) => is_inset(insert(ns , n), m) = true;
        (n ne m) => is_inset(insert(ns , n), m) = is_inset(ns , m);
      ofsort t_natset
        insertelem(emptyset, n) = insert(emptyset, n);
        not( is_inset (ns , n)) =>
          insertelem(ns , n) = insert(ns , n);
         is_inset (ns , n) =>
          insertelem(ns , n) = ns;
        union(ns, emptyset) = ns;
        not( is_inset (ns , n)) =>
          union(ns,  insert (ns1 , n)) = insert (union(ns, ns1), n);
         is_inset (ns , n) =>
          union(ns,  insert (ns1 , n)) = union(ns, ns1);
      ofsort nat
        length(emptyset) = 0;
        length( insert (ns , n)) = length(ns) + 1;
  endtype

  type t_msgseqfid is
    t_msgseq, t_natset , boolean
    sorts
      t_msgseqfid
    opns
      emptyseqfid                      : -> t_msgseqfid
      seqfid       (*! constructor *) : t_msgseq, t_natset -> t_msgseqfid
      p_msgseq                         : t_msgseqfid -> t_msgseq
      p_fset                           : t_msgseqfid -> t_natset
    eqns
      forall ns: t_natset ,
             ms: t_msgseq
      ofsort t_msgseq
        p_msgseq(seqfid(ms, ns)) = ms;
      ofsort t_natset
         p_fset ( seqfid (ms, ns)) = ns;
      ofsort t_msgseqfid
        emptyseqfid = seqfid(emptyseq, emptyset);
  endtype

(*
 t_tree is the resolution tree
   insertion takes place at the marker (if the tree is not empty)
   movemarker allows to shift the marker one level up and to the right
   if the marker is not moved, it is a child of the last inserted node
*)
  type t_tree is
    boolean, naturalnumber, t_msg, t_msgseqfid, t_natset , t_msgseq
    sorts
      t_tree ,  t_list , t_node
    opns
      emptytree (*! constructor *) : -> t_tree
      marker    (*! constructor *) : ->t_tree
      mktree    (*! constructor *) : t_node,  t_list  -> t_tree
      is_empty                     : t_tree -> bool
      is_marker                    : t_tree -> bool
      insert                       : t_tree , t_node -> t_tree
```

```
    preorder                    : t_tree -> t_msgseqfid
    tree_prune                  : t_tree -> t_tree
    movemarker                  : t_tree -> t_tree
    eqtree                      : t_tree , t_tree -> bool
    extract_remduplicates       : t_tree -> t_tree
    rem_shortest                : t_tree -> t_tree
    maxdepth                    : t_tree -> nat
    rem_least                   : t_tree -> t_tree
    maxfeat                     : t_tree -> t_natset

    emptylist (*! constructor *) : -> t_list
    is_emptyl                   : t_list -> bool
    insl      (*! constructor *) : t_list , t_tree -> t_list
    preorderlist                : t_list -> t_msgseqfid
    eqlist                      : t_list , t_list -> bool
    length                      : t_list -> nat
    getnth                      : t_list , nat -> t_tree
    remlast                     : t_list -> t_list
    getlast                     : t_list -> t_tree
    firsthalf                   : t_list -> t_list
    secondhalf                  : t_list -> t_list
    firsthalf2                  : t_list , nat, t_list -> t_list
    secondhalf2                 : t_list , nat, t_list -> t_list
    listremduplicates           : t_list -> t_list
    noduplmerge                 : t_list , t_list -> t_list
    insnodupl                   : t_list , t_tree -> t_list
    rem_shortfromlist           : t_list -> t_list
    maxdepthl                   : t_list -> nat
    rem_leastfromlist           : t_list -> t_list
    maxfeatl                    : t_list -> t_natset

    mknode (*! constructor *) : t_msg, nat -> t_node
    p_nmsg                    : t_node -> t_msg
    p_nfid                    : t_node -> nat
    eqnode                    : t_node, t_node -> bool
  eqns
    forall t1, t2, t3: t_tree,
           n, n1, n2 : t_node,
           s1, s2 : t_msgseqfid,
           l1, l2: t_list,
           f: nat,
           m: t_msg
    ofsort bool
      is_empty(emptytree) = true;          is_empty(mktree(n,l1)) = false;
      is_marker(marker) = true;            is_marker(t1) = false;
      is_emptyl(emptylist) = true;         is_emptyl( insl (l1, t1)) = false;
      (* 2 nodes are equal if both their fid and msg are equal *)
      eqnode(n1, n2) = are_equal(p_nmsg(n1), p_nmsg(n2)) and (p_nfid(n1) eq p_nfid(n2));
      (* 2 trees are equal if: - they are both empty *)
      (*                        - their roots are eual and their treelist are equal *)
      eqtree(emptytree, emptytree) = true;
      not(is_empty(t1)) =>
        eqtree(t1, emptytree) = false;
      not(is_empty(t1)) =>
        eqtree(emptytree, t1) = false;
      not(is_marker(t1)) =>
        eqtree(t1, marker) = false;
      not(is_marker(t1)) =>
        eqtree(marker, t1) = false;
      eqtree(mktree(n1, l1), mktree(n2, l2)) = eqnode(n1, n2) and eqlist(l1, l2);
      (* 2 tree lists are eual if: - they are both empty *)
      (*                           - all elements at the same position are equal *)
      eqlist (emptylist, emptylist) = true;
      not(is_emptyl(l1)) =>
        eqlist (emptylist, l1) = false;
      not(is_emptyl(l1)) =>
        eqlist (l1, emptylist) = false;
      eqlist ( insl (l1, t1), insl (l2, t2)) = ( eqlist (l1, l2)) and (eqtree(t1, t2));
    ofsort t_tree
      insert (emptytree, n) = mktree(n, insl(emptylist, marker));
      insert (mktree(n1, insl(l1, marker)), n)
                      = mktree(n1, insl(l1, mktree(n, insl(emptylist, marker))));
      not(is_marker(t1)) =>
        insert (mktree(n1, insl(l1, t1)), n) = mktree(n1, insl(l1, insert (t1, n)));
      movemarker(emptytree) = emptytree;
      movemarker(mktree(n1, insl(l1, marker))) = mktree(n1, insl(l1, marker));
      movemarker(mktree(n1, insl(l1, mktree(n2, insl(l2, marker)))))
                      = mktree(n1, insl( insl (l1, mktree(n2, l2)), marker));
      not(is_marker(t1)) and not(is_empty(t1)) =>
        movemarker(mktree(n1, insl(l1, mktree(n2, insl(l2, t1)))))
                      = mktree(n1, (insl(l1, movemarker(mktree(n2, insl(l2, t1))))));
      getnth(emptylist, f) = emptytree;
      (f gt length(l1)) =>
        getnth(l1, f) = emptytree;
      (f eq length(l1)) =>
        getnth(l1, f) = getlast (l1);
      (f lt length(l1)) =>
        getnth(l1, f) = getnth(remlast(l1), f);
      getlast (emptylist) = emptytree;
      getlast ( insl (l1, t1)) = t1;
      extract_remduplicates(emptytree) = emptytree;
      extract_remduplicates(marker) = emptytree;
      extract_remduplicates(mktree(n1, l1)) = mktree(n1, listremduplicates(l1));
      (* keep longest branch *)
```

```
        rem_shortest(marker) = emptytree;
        rem_shortest(emptytree) = emptytree;
        rem_shortest(mktree(n1, emptylist)) = mktree(n1, emptylist);
        not(is_emptyl(l1)) =>
          rem_shortest(mktree(n1, l1)) = mktree(n1, rem_shortfromlist(l1));
        (* keep most features branch *)
        rem_least(marker) = emptytree;
        rem_least(emptytree) = emptytree;
        rem_least(mktree(n1, emptylist)) = mktree(n1, emptylist);
        not(is_emptyl(l1)) =>
          rem_least(mktree(n1, l1)) = mktree(n1, rem_leastfromlist(l1));
        (* Remove duplicates, then rem_least feature branches *)
        tree_prune(t1) = rem_least(extract_remduplicates(t1));
      ofsort t_msgseqfid
        preorder(emptytree) = emptyseqfid;
        preorder(marker) = emptyseqfid;
        preorder(mktree(n, l1)) = seqfid(concat(add(emptyseq, p_nmsg(n)), p_msgseq(preorderlist(l1))),
                                         union(insertelem(emptyset, p_nfid(n)), p_fset(preorderlist(l1))));

        preorderlist(emptylist) = emptyseqfid;
        preorderlist(insl(l1, t1)) = seqfid(concat(p_msgseq(preorderlist(l1)), p_msgseq(preorder(t1))),
                                            union(p_fset(preorderlist(l1)), p_fset(preorder(t1))));
      ofsort t_msg
        p_nmsg(mknode(m, f)) = m;
      ofsort nat
        p_nfid(mknode(m, f)) = f;
        length(emptylist) = 0;
        length(insl(l1, t1)) = length(l1) + 1;
        maxdepth(emptytree) = 0;
        maxdepth(marker) = 0;
        maxdepth(mktree(n, emptylist)) = 1;
        not(is_emptyl(l1)) =>
          maxdepth(mktree(n, l1)) = maxdepthl(l1) + 1;
        maxdepthl(emptylist) = 0;
        maxdepthl(insl(emptylist, t1)) = maxdepth(t1);
        maxdepth(t1) gt maxdepthl(l1) =>
          maxdepthl(insl(l1, t1)) = maxdepth(t1);
        maxdepth(t1) le maxdepthl(l1) =>
          maxdepthl(insl(l1, t1)) = maxdepthl(l1);
      ofsort t_natset
        maxfeat(emptytree) = emptyset;
        maxfeat(marker) = emptyset;
        maxfeat(mktree(n, emptylist)) = insertelem(emptyset, p_nfid(n));
        not(is_emptyl(l1)) =>
          maxfeat(mktree(n, l1)) = insertelem(maxfeatl(l1), p_nfid(n));

        maxfeatl(emptylist) = emptyset;
        maxfeatl(insl(emptylist, t1)) = maxfeat(t1);
        length(maxfeat(t1)) gt length(maxfeatl(l1)) =>
          maxfeatl(insl(l1, t1)) = maxfeat(t1);
        length(maxfeat(t1)) le length(maxfeatl(l1)) =>
          maxfeatl(insl(l1, t1)) = maxfeatl(l1);
      ofsort t_list
        remlast(emptylist) = emptylist;
        remlast(insl(l1, t1)) = l1;
        length(l1) eq 0 => firsthalf(l1) = emptylist;
        length(l1) eq 0 => secondhalf(l1) = emptylist;
        length(l1) eq 1 => firsthalf(l1) = l1;
        length(l1) eq 1 => secondhalf(l1) = emptylist;
        length(l1) eq 2 => firsthalf(l1) = insl(emptylist, getnth(l1, 1));
        length(l1) eq 2 => secondhalf(l1) = insl(emptylist, getnth(l1, 2));
        length(l1) gt 2 => firsthalf(l1) = firsthalf2(l1, length(l1) / 2, emptylist);
        length(l1) gt 2 => secondhalf(l1) = secondhalf2(l1, length(l1) / 2, emptylist);
        f eq 0 => firsthalf2(l1, f, l2) = l2;
        f gt 0 => firsthalf2(l1, f, l2) = firsthalf2(l1, f - 1, insl(l2, getnth(l1, f)));
        f eq length(l1) => secondhalf2(l1, f, l2) = l2;
        f lt length(l1) => secondhalf2(l1, f, l2) = secondhalf2(l1, f + 1, insl(l2, getnth(l1, f + 1)));
        length(l1) eq 0 => listremduplicates(l1) = emptylist;
        length(l1) eq 1 =>
          listremduplicates(l1) = insl(emptylist, extract_remduplicates(getlast(l1)));
        (length(l1) eq 2) and not (eqtree(getnth(l1, 1), getnth(l1, 2))) =>
          listremduplicates(l1) = insl(insl(emptylist, extract_remduplicates(getnth(l1, 1))),
                                       extract_remduplicates(getnth(l1, 2)));
        (length(l1) eq 2) and eqtree(getnth(l1, 1), getnth(l1, 2)) =>
          listremduplicates(l1) = insl(emptylist, extract_remduplicates(getnth(l1, 1)));
        (length(l1) gt 2) =>
          listremduplicates(l1) = noduplmerge(listremduplicates(firsthalf(l1)),
                                              listremduplicates(secondhalf(l1)));
        noduplmerge(l1, emptylist) = l1;
        noduplmerge(l1, l2) = noduplmerge(insnodupl(l1, getlast(l2)), remlast(l2));
        insnodupl(emptylist, t1) = insl(emptylist, t1);
        eqtree(t1, t2) => insnodupl(insl(l1, t1), t2) = insl(l1, t1);
        not(eqtree(t1, t2)) => insnodupl(insl(l1, t1), t2) = insl(insnodupl(l1, t2), t1);
        (* keeps longest branch // leftmost if in doubt *)
        rem_shortfromlist(emptylist) = emptylist;
        rem_shortfromlist(insl(emptylist, t1)) = insl(emptylist, rem_shortest(t1));
        not(is_emptyl(l1)) and (maxdepth(t1) gt maxdepthl(l1)) =>
          rem_shortfromlist(insl(l1, t1)) = insl(emptylist, rem_shortest(t1));
        not(is_emptyl(l1)) and (maxdepth(t1) le maxdepthl(l1)) =>
          rem_shortfromlist(insl(l1, t1)) = rem_shortfromlist(l1);
        (* keeps branch with max features // leftmost if in doubt *)
        rem_leastfromlist(emptylist) = emptylist;
        rem_leastfromlist(insl(emptylist, t1)) = insl(emptylist, rem_least(t1));
        not(is_emptyl(l1)) and (length(maxfeat(t1)) gt length(maxfeatl(l1))) =>
```

```
        rem_leastfromlist ( insl (l1 , t1)) = insl (emptylist , rem_least(t1 ));
      not(is_emptyl(l1 )) and (length(maxfeat(t1)) le length(maxfeatl(l1))) =>
        rem_leastfromlist ( insl (l1 , t1)) = rem_leastfromlist (l1 );
  endtype

(*
 t_natstack is a data type for a stack of natural numbers
*)
  type t_natstack is
    boolean,naturalnumber
    sorts
      t_natstack
    opns
      emptystack (*! constructor *) : -> t_natstack
      is_empty                      : t_natstack -> bool
      top                           : t_natstack -> nat
      pop                           : t_natstack -> t_natstack
      push       (*! constructor *) : t_natstack , nat -> t_natstack
      bottom                        : t_natstack -> nat
    eqns
      forall n: nat,
             ns: t_natstack
      ofsort bool
        is_empty(emptystack) = true;       is_empty(push(ns, n)) = false;
      ofsort nat
        top (push(emptystack, n)) = n;
        not(is_empty(ns)) => top(push(ns, n)) = n;
        bottom(emptystack) = 0; (* to avoid undefined case *)
        bottom(push(emptystack, n)) = n;
        not(is_empty(ns)) => bottom(push(ns, n)) = bottom(ns);
      ofsort t_natstack
        pop(push(emptystack, n)) = emptystack;
        not(is_empty(ns)) => pop(push(ns, n)) = ns;
  endtype

(*
 t_state is the datatype for states of the FM, basically a tuple
*)
  type t_state is
    t_msgseq
    sorts
      t_state

    opns
      state      (*! constructor *) : (*# TMSGSEQ *), nat -> t_state
                 (* take n queues and the number of choices left and make a state *)
(*# TSTATEOPNS *)
      p_chosen                      : t_state -> nat
    eqns
      forall (*# QUEUES *): t_msgseq,
             c: nat
      ofsort t_msgseq
(*# TSTATEEQNS *)
      ofsort nat
        p_chosen(state((*# QUEUES *), c)) = c;
  endtype

(*
 t_statestack is a data type for a stack of states
*)
  type t_statestack is
    boolean,naturalnumber,t_state
    sorts
      t_statestack
    opns
      emptyststack (*! constructor *) : -> t_statestack
      is_empty     (*!                : t_statestack -> bool
      top          (*!                : t_statestack -> t_state
      pop          (*!                : t_statestack -> t_statestack
      push         (*! constructor *) : t_statestack , t_state -> t_statestack
    eqns
      forall st: t_state ,
           sts: t_statestack
      ofsort bool
        is_empty(emptyststack) = true;
        is_empty(push(sts, st)) = false ;
      ofsort t_state
        top(push(emptyststack, st)) = st;
        not(is_empty(sts)) => top(push(sts, st)) = st;
      ofsort t_statestack
        pop(push(emptyststack, st)) = emptyststack;
        not(is_empty(sts)) => pop (push (sts, st)) = sts;
  endtype

(* ============================================================================ *)
(* ============================================================================ *)
(* ============================================================================ *)

(*
 The system behaviour: three features in paralell with the FM and the triggergenerator
 The cocoons and the FM synchronise on Out, the channel used for broadcasting trigger
 events and feedback
*)
behaviour
```

```

(*
  Process Gate Parameters are in the order Out, In (the FM has EnvOut, EnvIn, out, in's)
  The gates are always named from a viewpoint of the process
  In the spec it is neccessary to connect them corrctly!
  For clarity consult the Process Diagram in mr0800
*)
  hide Out, (*# GATES *), EnvIn, EnvOut in
    TriggerGen [EnvIn, EnvOut]
    |[EnvIn, EnvOut]|
    FM [EnvOut, EnvIn, Out, (*# GATES *)]
    |[Out, (*# GATES *)]|
    (
(*# BEHAVECOCOON *)
    )

where

(*
 FM: The feature manager Construct-Prune implementation
 The feature manager - reads a msg from EnvIn,
                     - passes the msg on to Construct
                          (which deals with the tree construction
                           and feedback mechanism and then
                           returns a solution tree)
                     - the solution tree is passed to Prune
                          (which prunes the tree and returns a sequence)
                     - the sequence is then passed to Commit
                          (which deals with setting the features in the
                           appropriate states and sending the commited
                           behaviour to the environment)
                     - and finally the control is returned to the FM process.
*)
  process FM[EnvOut, EnvIn, Out, (*# GATES *)]: noexit :=
  (
    EnvIn ? msg: t_msg;
    Construct[Out, (*# GATES *)](msg, emptytree)
    >>accept t1: t_tree in
    (
      Prune[EnvOut](t1)
      >>accept ms: t_msgseqfid in
      (
        Commit[EnvOut, Out](ms)
        >>
        FM[EnvOut, EnvIn, Out, (*# GATES *)]
      )
    )
  )
  endproc (* FM *)

(*
 Construct
  this process deals with constructing the tree. In order to do so,
  it farms out msg, awaits responses and deals with them.
  upon termination it returns the solution tree.
*)
  process Construct[Out, (*# GATES *)](msg: t_msg, t: t_tree): exit (t_tree) :=
  (
    (* send START to all features and broadcast message *)
    Out ! message(rcv, start_transaction , 0, 0,  nil );
    Out ! msg;
    (* Collect Responses *)
    CollectResp[(*# GATES *)]((*# EMPTYSEQ *), (*# FALSE *))
    >>accept (*# QUEUES *): t_msgseq in
    (
      FeedBackControl[Out, (*# GATES *)]((*# QUEUES *), insert(t, mknode(msg, 0)))
      (* tree already contains orig msg *)
      >>accept t: t_tree in
        exit(t)
    )
  )

  where

    (* Gather collected responses from all features *)
    process CollectResp [(*# GATES *)]((*# QUEUES *): t_msgseq, (*# COLLECT *): bool): exit
                                                  ((*# TMSGSEQ *)) :=
    (
(*# COLLECTBLOCK *)
      []
      [(*# CONDITION *)] -> (* all finished, exit with queues *)
        exit((*# QUEUES *))
    )
    endproc (* CollectResp *)

    process FeedBackControl[Out, (*# GATES *)]((*# QUEUES *): t_msgseq, origt: t_tree): exit (t_tree) :=
    (
(*# FEEDBACKCONTROL *)
      exit(t)
    )
    endproc (* FeedBackControl *)

    process FeedBack[Out, (*# GATES *)]((*# QUEUES *): t_msgseq, stk: t_statestack,
                                        t: t_tree , chosen: nat, (*# FEATURES *): bool): exit (t_tree) :=
```

```
    (
      [(*# LENGTH *) eq 0] ->
                (* there were no responses, all queues length 0 *)
      (
        [is_empty(stk)] ->
        (
          exit(t)  (* finished, all possibilities have been explored *)
        )
        []
        [not(is_empty(stk))] -> (* maybe there are more choices earlier in the tree? *)
        (
          Out ! message(rcv, rollback_transaction , 0, 0, nil);
          FeedBack[Out, (*# GATES *)]((*# TOP *),
                                      pop(stk), movemarker(t), p_chosen(top(stk)), (*# FEATURES *))
                (* new branch is to be started, move insert marker of tree to right place *)
        )
      )
      []
      [(*# LENGTH *) gt 0] ->
            (* there was at least 1 response, feed responses back *)
      (
        [chosen lt (*# POSSCHOICES *)] ->
        (
          Out ! message(rcv, start_transaction , 0, 0, nil);
          GetMsgFromQ((*# QUEUES *), chosen)
          >> accept msg: t_msg, qno: nat, (*# NQUEUES *): t_msgseq, newchosen: nat in
          (
            Out ! msg;
            CollectResp [(*# GATES *)]((*# EMPTYSEQ *), (*# FALSE *))
            >>accept (*# TMPQUEUES *): t_msgseq in
            (
              IgnoreRespFromFeat((*# TMPQUEUES *), (*# FEATURES *))
              >>accept (*# TEMPQUEUES *): t_msgseq in
                FeedBack[Out, (*# GATES *)]((*# CONCAT *),
                                      push(stk, state((*# QUEUES *), newchosen)),
                                      insert(t, mknode(msg, qno)), 0, (*# FEATURES *))
                          (* 0 choices taken as new level has been started, but backup current value *)
            )
          )
        )
        []
        [chosen eq (*# POSSCHOICES *)] ->
          FeedBack[Out, (*# GATES *)]((*# EMPTYSEQ *), stk, t, 0, (*# FEATURES *))
            (* we have tried all choices on this level, just rollback *)
      )
    )

    where

      (* if we deal with less than n features we want to ignore responses *)
      process IgnoreRespFromFeat((*# QUEUES *): t_msgseq, (*# FEATURES *): bool): exit
                                              ((*# TMSGSEQ *)) :=
      (
(*# IGNORERESP *)
      )
      endproc (* IgnoreRespFromFeat *)

      (* get a message from one queue, ensuring they all will betaken eventually *)
      process GetMsgFromQ((*# QUEUES *): t_msgseq, chosen: nat): exit
                           (t_msg, nat, (*# TMSGSEQ *), nat) :=
      (
        GetChosenMsg((*# MAKEQLIST *), (*# QUEUES *), chosen, 0)
        >> accept m: t_msg, qno: nat, (*# NQUEUES *): t_msgseq in
        exit(m, qno, (*# NQUEUES *), chosen + 1)
      )

      where

        process GetChosenMsg(qlist: t_qlist, (*# QUEUES *): t_msgseq, chosen, tested: nat) : exit
                              (t_msg, nat, (*# TMSGSEQ *)) :=
        (
          [(chosen + 1) eq 1] ->
          (
            [not(is_empty( getfirst ( qlist )))] ->
            (
              RemFromAllQueues((*# MAKEQLIST *), emptyqlist, first(getfirst(qlist)))
              >> accept (*# NQUEUES *): t_msgseq in
              exit( first ( getfirst ( qlist )), tested + 1, (*# NQUEUES *))
            )
            []
            [is_empty( getfirst ( qlist ))] ->
              GetChosenMsg(remfirst(qlist), (*# QUEUES *), chosen, tested + 1)
          )
          []
          [(chosen + 1) gt 1] ->
          (
            [not(is_empty( getfirst ( qlist )))] ->
              GetChosenMsg(remfirst(qlist), (*# QUEUES *), chosen - 1, tested + 1)
            []
            [is_empty( getfirst ( qlist ))] ->
              GetChosenMsg(remfirst(qlist), (*# QUEUES *), chosen, tested + 1)
          )
        )
        endproc (* GetChosenMsg *)
```

```
          process RemFromAllQueues(ql1, ql2: t_qlist, m: t_msg) : exit
                                    ((*# TMSGSEQ *)) :=
          (
            [is_empty(ql1)] -> (* exit *)
              exit((*# DEQUEUELIST *))
            []
            [not(is_empty(ql1))] ->
            (
              [are_equal( first ( getfirst (ql1 )), m)] ->
                RemFromAllQueues(remfirst(ql1), add(ql2, remove(getfirst(ql1))), m)
              []
              [not(are_equal( first ( getfirst (ql1 )), m))] ->
                RemFromAllQueues(remfirst(ql1), add(ql2, getfirst(ql1 )), m)
            )
          )
          endproc (* RemFromAllQueues *)

        endproc (* GetMsgFromQ *)

      endproc (* FeedBack *)

    endproc (* Construct *)

  (*
   Prune: the prune process is straigth forward,
          the pruning is implemented in the tree data type
          on termination a message sequence is returned -- the behaviour to be committed
  *)
    process Prune[EnvOut](t: t_tree): exit (t_msgseqfid) :=
    (
      EnvOut!extract_remduplicates(t)(*| C printf (" full tree: ! %s \n", kd_draw ($!1)); |*);
      PruneSub(t)
      >> accept nt: t_tree in
      (
        exit(preorder(nt))
      )
    )

    where

      process PruneSub(t: t_tree): exit (t_tree) :=
      (
        exit(tree_prune(t))
      )
      endproc (* PruneSub *)

    endproc (* Prune *)

  (*
   Commit process
  *)
    process Commit[EnvOut, Out](msf: t_msgseqfid): exit :=
    (
      [not(is_empty(p_msgseq(msf)))] ->
      (
        SetFeatures[Out](p_msgseq(msf), p_fset(msf), 0, (*# NOOFFEAT *)) (* set all features to the right states *)
                                    (* been at 0 feat , have to do 3 *)
        >>
        CommitUser[EnvOut](p_msgseq(msf)) (* send responses to user *)
      )
      []
      [is_empty(p_msgseq(msf))] ->
        exit
    )

    where

      (* commit responses to user *)
      process CommitUser[EnvOut](ms: t_msgseq): exit :=
      (
        [not(is_empty(ms))] ->
        (
          [p_ioasp( first (ms)) eq snd] -> (* this was a response *)
          (
            EnvOut ! first(ms); (* send it to the user *)
            CommitUser[EnvOut](remove(ms))
          )
          []
          [p_ioasp( first (ms)) eq rcv] -> (* this was an input *)
          (
            CommitUser[EnvOut](remove(ms))
          )
        )
        []
        [(is_empty(ms))] ->
          exit
      )
      endproc (* CommitUser *)

      (* establish correct state of features *)
      process SetFeatures[Out](ms: t_msgseq, ns: t_natset, beenat, haveto: nat): exit :=
      (
        [beenat eq haveto] -> (* have we dealt with all features? *)
```

```
          exit
        []
        [beenat lt haveto] -> (* there are more features to be set *)
        (
          [ is_inset (ns, beenat + 1)] -> (* we want to keep this feature *)
          (
            Out ! message(snd, commit_transaction, 0, beenat + 1, nil ); (* go into playback phase *)
            CommitFeat[Out](ms, beenat + 1)                              (* then playback msgs *)
            >>
            SetFeatures[Out](ms, ns, beenat + 1, haveto)
                          (* continue with next feature *)
          )
          []
          [not( is_inset (ns, beenat + 1))] -> (* we do not want to keep this feature *)
          (
            Out ! message(snd, abort_transaction , 0, beenat + 1, nil );
                        (* tell this feature to abort *)
            SetFeatures[Out](ms, ns, beenat + 1, haveto)
                          (* continue with next feature *)
          )
        )
      )
      where

        (* playbeack messages to features *)
        process CommitFeat[Out](ms: t_msgseq, fid: nat): exit :=
        (
          [not(is_empty(ms))] ->
          (
            Out ! message(p_ioasp(first (ms)), p_event( first (ms)), p_source( first (ms)),
                               fid , p_argument(first(ms)));
                  (* make sure that dest is not 0 but a feature number *)
            CommitFeat[Out](remove(ms), fid)
          )
          []
          [(is_empty(ms))] ->
          (
            Out ! message(snd, commit_transaction, 0, fid , nil );
            (* end playback phase *)
            exit
          )
        )
        endproc (* CommitFeat *)

      endproc (* SetFeatures *)

    endproc (* Commit *)

  (*
   The Cocoon (or Feature Controller)
  *)
    process Cocoon [Out, In0] (cstate: nat,rbstk: t_natstack , fid , whichfeat: nat): noexit :=
    (
      In0 ? msg: t_msg;
      (
        (* broadcasted or dedicated control message *)
        [is_tcmsg(msg) and ((p_dest(msg) eq fid) or (p_dest(msg) eq 0))] ->
        (
          TransControl(cstate, rbstk , msg)
          >>accept newstack: t_natstack, newstate: nat, playback: bool in
          (
            [playback] -> (* we are comitting, so just set feature to the requested state *)
            (
              PlayBackCocoon[In0](newstate, newstack, fid, whichfeat)
              >> accept newstate2: nat in
              Cocoon[Out, In0](newstate2, emptystack, fid, whichfeat)
              (* after playback we do not need the stack anymore, we start fresh *)
            )
            []
            [not(playback)] -> (* we are not in a playback phase, so we still construct *)
              Cocoon[Out, In0](newstate, newstack, fid, whichfeat)
          )
        )
        []
        (* control message for another feature *)
        [is_tcmsg(msg) and (p_dest(msg) ne fid) and (p_dest(msg) ne 0)] ->
          Cocoon[Out, In0](cstate, rbstk , fid , whichfeat)
        []
        (* broadcasted or dedicated feature message *)
        [is_fmsg(msg) and ((p_dest(msg) eq fid) or ( p_dest(msg) eq 0))] ->
        (
          FeatBehave(cstate, msg, whichfeat)
          >>accept newstate: nat, response: t_msgseq in
          (
            Send [Out](response, newstate)
            >>accept newstate2: nat in
              Cocoon[Out, In0](newstate2, rbstk, fid , whichfeat)
          )
        )
        []
        (* feature message for another feature *)
        [is_fmsg(msg) and ((p_dest(msg) ne fid) and (p_dest(msg) ne 0))] ->
          Cocoon[Out, In0](cstate, rbstk , fid , whichfeat)
      )
```

```
  )

  where

    (* unfold message sequence into individual messages *)
    process Send[Out](resp: t_msgseq, newstate: nat): exit (nat) :=
    (
      [is_empty(resp)] ->
      (
        Out ! message(snd, respcomplete, 0, 0, nil);
        exit(newstate)
      )
      []
      [not(is_empty (resp))] ->
      (
        Out ! first (resp);
        Send[Out](remove (resp), newstate)
      )
    )
    endproc (* Send *)

    (* this is a playback of messages, just send to feature and accept responses *)
    process PlayBackCocoon[In0](cstate: nat, rbstk: t_natstack, fid , whichfeat: nat ) : exit (nat) :=
    (
      In0 ? msg: t_msg;
      (
        [is_fmsg(msg) and ((p_dest(msg) eq fid) or (p_dest(msg) eq 0))] ->
           (* there is a message for the feature ... *)
           (* just send the message out, discard the responses but change the state! *)
        (
          FeatBehave(cstate, msg, whichfeat)
          >>accept newstate: nat, response: t_msgseq in
            PlayBackCocoon[In0](newstate, rbstk, fid, whichfeat)
        )
        []
        [(p_event(msg) eq commit_transaction) and ((p_dest(msg) eq fid) or (p_dest(msg) eq 0))] ->
           (* playback has finished , just be a normal cocoon in the new state *)
          exit(cstate)
      )
    )
    endproc (* PlayBackCocoon *)

  endproc (* Cocoon *)

(*
 Feature behaviour (or in DM terms feature proper).
 The Fidentifier parameter selects which feature is used
 1: Teenline
 2: CFB
 3: CFB_Everyone
 4: TCS
 5: TCS_Everyone1
 6: TCS_Everyone2
 7: Reverse Charging
 8: Split Billing
 9: Basic Call
10: CND
11: CND_Everyone
12: CNDB
13: Call Transfer
14: Call Transfer Everyone
15: Call Waiting
16: Call Waiting Everyone
17: Group Ringing
18: Group Ringing Everyone
19: Ringback
20: Ringback Everyone1
21: Ringback Everyone2
22: Three Way Calling
23: Voice Mail - record
24: Voice Mail Everyone
25: Voice Mail - playback
*)
  process FeatBehave (s: nat, msg: t_msg, fid: nat): exit (nat, t_msgseq) :=
  (
    [fid eq 1] -> (* the teenline feature *)
    (
      [(s eq 0) and (p_event(msg) eq offhook)] -> (* teentime *)
        exit (1, add(emptyseq, message(snd, announce, 0, 0, askpin)))
(*        []*)
(*        [(s eq 0) and (p_event(msg) eq offhook)] ->*) (* not teentime, the tau *)
(*          exit (1, add(emptyseq, message(snd, billing_offhook , 0, 0, time)))*)
      []
      [(s eq 0) and (p_event(msg) neq offhook)] ->
        exit (0, emptyseq)
      []
      [(s eq 1) and (p_event(msg) eq onhook)] ->
        exit (0, emptyseq)
      []
      [(s eq 1) and (p_event(msg) eq dial)] -> (* wrong pin *)
        exit (2, add(emptyseq, message(snd, announce, 0, 0, wrongpin)))
(*        []*)
(*        [(s eq 1) and (p_event(msg) eq dial)] ->*) (* right pin , the tau *)
(*          exit (0, emptyseq)*)
```

```
      []
      [(s eq 1) and (p_event(msg) eq i_alert)] ->
         exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
      []
      [(s eq 1) and ((p_event(msg) neq onhook) or (p_event(msg) neq dial) or
                     (p_event(msg) neq i_alert))] ->
        exit (1, emptyseq)
      []
      [(s eq 2) and (p_event(msg) eq onhook)] ->
        exit (0, emptyseq)
      []
      [(s eq 2) and (p_event(msg) eq i_alert)] ->
         exit (2, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
      []
      [(s eq 2) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_alert))] ->
        exit (2, emptyseq)
    )
    []
    [fid eq 2] -> (* call forwarding busy *)
                 (* Change wrt Contest: trigger is not incoming call but
                    feedback of busy announcement ... we do not have state info *)
                 (* Alterantive would be non determinism, but that's not neccessary *)
    (
      [(s eq 0) and (p_event(msg) eq o_busy)] ->
        exit (0, add(add(add(emptyseq, message(snd, o_alert, 0, 0, nil)),
                                         message(snd, billing_forwarded, 0, 0, nil)),
                                         message(snd, o_notify, 0, 0, nil)))
      []
      [(s eq 0) and (p_event(msg) neq o_busy)] ->
        exit (0, emptyseq)
    )
    []
    [fid eq 3] -> (* call forwarding busy everyone *)
    (
      [(s eq 0) and (p_event(msg) eq i_notify)] ->
         exit (1, emptyseq)
      []
      [(s eq 0) and (p_event(msg) neq i_notify)] ->
         exit (0, emptyseq)
      []
      [(s eq 1) and (p_event(msg) eq i_alert)] ->
         exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
      []
      [(s eq 1) and (p_event(msg) eq i_free)] ->
         exit (0, emptyseq)
      []
      [(s eq 1) and (p_event(msg) eq i_busy)] ->
         exit (0, emptyseq)
      []
      [(s eq 1) and (p_event(msg) eq onhook)] ->
         exit (0, add(emptyseq, message(snd, o_stopalert, 0, 0, nil)))
      []
      [(s eq 1) and ((p_event(msg) neq i_free) or (p_event(msg) neq i_alert) or
                     (p_event(msg) neq i_busy) or (p_event(msg) neq onhook)) ] ->
         exit (1, emptyseq)
    )
    []
    [fid eq 4] -> (* terminating call screening *)
    (
      [(s eq 0) and (p_event(msg) eq i_alert)] -> (* the screened transition *)
         exit (1, add(emptyseq, message(snd, o_request, 0, 0, callerid)))
      []
      [(s eq 0) and (p_event(msg) neq i_alert)] ->
         exit (0, emptyseq)
      []
(*      [(s eq 1) and (p_event(msg) eq i_inform)] -> *) (* not barred, the tau *)
(*         exit (0, emptyseq) *)
      [(s eq 1) and (p_event(msg) eq i_inform)] ->
         exit (0, add(emptyseq, message(snd, announce, 0, 0, screened)))
      []
      [(s eq 1) and (p_event(msg) eq i_alert)] ->
         exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
      []
      [(s eq 1) and ((p_event(msg) neq i_inform) or (p_event(msg) neq i_alert))] ->
         exit (0, emptyseq)
    )
    []
    [fid eq 5] -> (* terminating call screening everyone 1 *)
    (
      [(s eq 0) and (p_event(msg) eq i_inform)] ->
         exit (0, add(emptyseq, message(snd, announce, 0, 0, screened)))
      []
      [(s eq 0) and (p_event(msg) neq i_inform)] ->
         exit (0, emptyseq)
    )
    []
    [fid eq 6] -> (* terminating call screening everyone 2 *)
    (
      [(s eq 0) and (p_event(msg) eq i_request)] ->
         exit (0, add(emptyseq, message(snd, o_inform, 0, 0, barred)))
      []
      [(s eq 0) and (p_event(msg) neq i_request)] ->
         exit (0, emptyseq)
    )
```

```
      []
      [fid eq 7] -> (* reverse charging *)
      (
(*       [(s eq 0) and (p_event(msg) eq dial)] -> *) (* the tau transition *)
(*          exit (0, emptyseq)*)
(*       []*)
        [(s eq 0) and (p_event(msg) eq dial)] -> (* the reverse transition *)
           exit (0, add(emptyseq, message(snd, billing_reverse , 0, 0, nil )))
        []
        [(s eq 0) and (p_event(msg) neq dial)] ->
           exit (0, emptyseq)
      )
      []
      [fid eq 8] -> (* split billing *)
      (
(*       [(s eq 0) and (p_event(msg) eq dial)] -> *) (* the tau transition *)
(*          exit (0, emptyseq)*)
(*       []*)
        [(s eq 0) and (p_event(msg) eq dial)] -> (* the split transition *)
           exit (0, add(emptyseq, message(snd, billing_split , 0, 0, splitfactor )))
        []
        [(s eq 0) and (p_event(msg) neq dial)] ->
           exit (0, emptyseq)
      )
      []
      [fid eq 9] -> (* basic call *)
      (
        [(s eq 0) and (p_event(msg) eq offhook)] ->
           exit (1, add(add(emptyseq, message(snd, billing_offhook , 0, 0, time)),
                                   message(snd, dialtone , 0, 0, nil )))
        []
        [(s eq 0) and (p_event(msg) eq i_alert)] ->
           exit (6, add(add(emptyseq, message(snd, o_free, 0, 0, nil )),
                                   message(snd, alert , 0, 0, nil )))
(*       []*)
(*       [(s eq 0) and (p_event(msg) eq i_alert)] ->*)
(*          exit (0, add(add(add(add(emptyseq, message(snd, o_free, 0, 0, nil)),*)
(*                                         message(snd, alert , 0, 0, nil )),*)
(*                                         message(snd, o_timeout , 0, 0, nil )),*)
(*                                         message(snd, stopalert , 0, 0, nil )))*)
        []
        [(s eq 0) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq offhook))] ->
           exit (0, emptyseq)
        []
        [(s eq 1) and (p_event(msg) eq dial)] ->
           exit (2, add(emptyseq, message(snd, o_alert , 0, 0, nil )))
        []
        [(s eq 1) and (p_event(msg) eq onhook)] ->
           exit (0, add(emptyseq, message(snd, billing_onhook, 0, 0, time)))
        []
        [(s eq 1) and (p_event(msg) eq i_alert )] ->
           exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
        []
        [(s eq 1) and ((p_event(msg) neq dial) or (p_event(msg) neq onhook) or
                   (p_event(msg) neq i_alert ))] ->
           exit (1, emptyseq)
        []
        [(s eq 2) and (p_event(msg) eq onhook)] ->
           exit (0, add(add(emptyseq, message(snd, billing_onhook, 0, 0, time)),
                                   message(snd, o_stopalert , 0, 0, nil )))
        []
        [(s eq 2) and (p_event(msg) eq i_busy)] ->
           exit (5, add(emptyseq, message(snd, busytone, 0, 0, nil)))
        []
        [(s eq 2) and (p_event(msg) eq i_free )] ->
           exit (3, add(emptyseq, message(snd, ringtone, 0, 0, nil )))
        []
        [(s eq 2) and (p_event(msg) eq i_alert )] ->
           exit (2, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
        []
        [(s eq 2) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_free) or
                   (p_event(msg) neq i_busy) or (p_event(msg) neq i_alert ))] ->
           exit (2, emptyseq)
        []
        [(s eq 3) and (p_event(msg) eq onhook)] ->
           exit (0, add(add(emptyseq, message(snd, billing_onhook, 0, 0, time)),
                                   message(snd, o_stopalert , 0, 0, nil )))
        []
        [(s eq 3) and (p_event(msg) eq i_timeout)] ->
           exit (5, add(emptyseq, message(snd, timeouttone, 0, 0, nil)))
        []
        [(s eq 3) and (p_event(msg) eq i_connect)] ->
           exit (4, add(add(emptyseq, message(snd, billing_start , 0, 0, time)),
                                   message(snd, connect , 0, 0, nil )))
        []
        [(s eq 3) and (p_event(msg) eq i_alert )] ->
           exit (3, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
        []
        [(s eq 3) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_timeout) or
                   (p_event(msg) neq i_connect) or (p_event(msg) neq i_alert ))] ->
           exit (3, emptyseq)
        []
        [(s eq 4) and (p_event(msg) eq onhook)] ->
           exit (0, add(add(add(emptyseq, message(snd, billing_stop, 0, 0, time)),
```

```
                                         message(snd, billing_onhook , 0, 0,  time)),
                                         message(snd, o_disconnect , 0, 0,  nil )))
   []
   [(s eq 4) and (p_event(msg) eq i_disconnect)] ->
      exit (5, add(emptyseq, message(snd, disconnecttone, 0, 0, nil )))
   []
   [(s eq 4) and (p_event(msg) eq i_alert)] ->
      exit (4, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
   []
   [(s eq 4) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_disconnect)
                 or (p_event(msg) neq i_alert))] ->
      exit (4, emptyseq)
   []
   [(s eq 5) and (p_event(msg) eq onhook)] ->
      exit (0, add(emptyseq, message(snd, billing_onhook, 0, 0, time)))
   []
   [(s eq 5) and (p_event(msg) eq i_alert)] ->
      exit (5, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
   []
   [(s eq 5) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_alert))] ->
      exit (5, emptyseq)
   []
   [(s eq 6) and (p_event(msg) eq i_stopalert)] ->
      exit (0, add(emptyseq, message(snd, stopalert , 0, 0,  nil )))
   []
   [(s eq 6) and (p_event(msg) eq offhook)] ->
      exit (7, add(add(emptyseq, message(snd, billing_offhook , 0, 0, time)),
                                  message(snd, o_connect , 0, 0,  nil )))
   []
   [(s eq 6) and (p_event(msg) eq i_alert)] ->
      exit (6, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
   []
   [(s eq 6) and ((p_event(msg) neq i_stopalert) or (p_event(msg) neq offhook)
                 or (p_event(msg) neq i_alert))] ->
      exit (6, emptyseq)
   []
   [(s eq 7) and (p_event(msg) eq onhook)] ->
      exit (0, add(add(add(emptyseq, message(snd, billing_stop, 0, 0, time)),
                                         message(snd, billing_onhook , 0, 0,  time)),
                                         message(snd, o_disconnect , 0, 0,  nil )))
   []
   [(s eq 7) and (p_event(msg) eq i_disconnect)] ->
      exit (8, add(emptyseq, message(snd, disconnecttone, 0, 0, nil )))
   []
   [(s eq 7) and (p_event(msg) eq i_alert)] ->
      exit (7, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
   []
   [(s eq 7) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_disconnect)
                 or (p_event(msg) neq i_alert))] ->
      exit (7, emptyseq)
   []
   [(s eq 8) and (p_event(msg) eq onhook)] ->
      exit (0, add(emptyseq, message(snd, billing_onhook, 0, 0, time)))
   []
   [(s eq 8) and (p_event(msg) eq i_alert)] ->
      exit (8, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
   []
   [(s eq 8) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_alert))] ->
      exit (8, emptyseq)
)
[]
[fid eq 10] -> (* Calling Number Display *)
(
   [(s eq 0) and (p_event(msg) eq i_alert)] ->
      exit (1, add(emptyseq, message(snd, o_request, 0, 0,  callerid )))
   []
   [(s eq 0) and (p_event(msg) neq i_alert)] ->
      exit (0, emptyseq)
   []
   [(s eq 1) and (p_event(msg) eq i_alert)] ->
      exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
   []
   [(s eq 1) and (p_event(msg) eq i_inform)] ->
      exit (0, add(emptyseq, message(snd, display , 0, 0, id )))
   []
   [(s eq 1) and ((p_event(msg) neq i_inform) or (p_event(msg) neq i_alert))] ->
      exit (1, emptyseq)
)
[]
[fid eq 11] -> (* Calling Number Display Everyone *)
(
   [(s eq 0) and (p_event(msg) eq i_request)] ->
      exit (0, add(emptyseq, message(snd, o_inform, 0, 0, id )))
   []
   [(s eq 0) and (p_event(msg) neq i_request)] ->
      exit (0, emptyseq)
)
[]
[fid eq 12] -> (* Calling Number Delivery Blocking *)
(
   [(s eq 0) and (p_event(msg) eq i_request)] ->
      exit (0, add(emptyseq, message(snd, o_inform, 0, 0, barred)))
   []
   [(s eq 0) and (p_event(msg) neq i_request)] ->
```

```
            exit (0, emptyseq)
      )
      []
      [fid eq 13] -> (* Call Transfer *)
      (
        [(s eq 0) and (p_event(msg) eq i_connect)] ->
            exit (1, emptyseq)
        []
        [(s eq 0) and (p_event(msg) eq o_connect)] ->
            exit (2, emptyseq)
        []
        [(s eq 0) and ((p_event(msg) neq o_connect) or (p_event(msg) neq i_connect))] ->
            exit (0, emptyseq)
        []
        [(s eq 1) and (p_event(msg) eq i_disconnect)] ->
            exit (0, emptyseq)
        []
        [(s eq 1) and (p_event(msg) eq onhook)] ->
            exit (0, emptyseq)
        []
        [(s eq 1) and (p_event(msg) eq i_alert)] ->
            exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 1) and (p_event(msg) eq flash)] ->
            exit (2, add(emptyseq, message(snd, dialtone, 0, 0, nil)))
        []
        [(s eq 1) and ((p_event(msg) neq i_alert) or (p_event(msg) neq i_disconnect)
                      or (p_event(msg) neq onhook) or (p_event(msg) neq flash))] ->
            exit (1, emptyseq)
        []
        [(s eq 2) and (p_event(msg) eq flash)] ->
            exit (1, emptyseq)
        []
        [(s eq 2) and (p_event(msg) eq i_disconnect)] ->
            exit (2, emptyseq)
        []
        [(s eq 2) and (p_event(msg) eq i_alert)] ->
            exit (2, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 2) and (p_event(msg) eq dial)] ->
            exit (3, add(emptyseq, message(snd, o_alert, 0, 0, nil)))
        []
        [(s eq 2) and ((p_event(msg) neq i_alert) or (p_event(msg) neq i_disconnect)
                      or (p_event(msg) neq flash) or (p_event(msg) neq dial))] ->
            exit (2, emptyseq)
        []
        [(s eq 3) and (p_event(msg) eq flash)] ->
            exit (1, add(emptyseq, message(snd, o_stopalert, 0, 0, nil)))
        []
        [(s eq 3) and (p_event(msg) eq i_disconnect)] ->
            exit (3, emptyseq)
        []
        [(s eq 3) and (p_event(msg) eq i_alert)] ->
            exit (3, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 3) and (p_event(msg) eq i_free)] ->
            exit (5, add(emptyseq, message(snd, ringtone, 0, 0, nil)))
        []
        [(s eq 3) and (p_event(msg) eq i_busy)] ->
            exit (4, add(emptyseq, message(snd, busytone, 0, 0, nil)))
        []
        [(s eq 3) and ((p_event(msg) neq i_alert) or (p_event(msg) neq i_disconnect)
                      or (p_event(msg) neq flash) or (p_event(msg) neq i_free)
                      or (p_event(msg) neq i_busy))] ->
            exit (3, emptyseq)
        []
        [(s eq 4) and (p_event(msg) eq flash)] ->
            exit (1, emptyseq)
        []
        [(s eq 4) and (p_event(msg) eq i_disconnect)] ->
            exit (0, emptyseq)
        []
        [(s eq 4) and (p_event(msg) eq onhook)] ->
            exit (0, emptyseq)
        []
        [(s eq 4) and (p_event(msg) eq i_alert)] ->
            exit (4, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 4) and ((p_event(msg) neq i_alert) or (p_event(msg) neq onhook)
                      or (p_event(msg) neq flash) or (p_event(msg) neq i_disconnect))] ->
            exit (4, emptyseq)
        []
        [(s eq 5) and (p_event(msg) eq flash)] ->
            exit (1, add(emptyseq, message(snd, o_stopalert, 0, 0, nil)))
        []
        [(s eq 5) and (p_event(msg) eq i_disconnect)] ->
            exit (5, emptyseq)
        []
        [(s eq 5) and (p_event(msg) eq onhook)] ->
            exit (0, add(emptyseq, message(snd, o_stopalert, 0, 0, nil)))
        []
        [(s eq 5) and (p_event(msg) eq i_alert)] ->
            exit (5, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
```

```
[(s eq 5) and (p_event(msg) eq i_connect)] ->
   exit (6, add(add(emptyseq, message(snd, billing_start , 0, 0, time)),
                              message(snd, connect , 0, 0, nil )))
[]
[(s eq 5) and (p_event(msg) eq i_timeout)] ->
   exit (4, add(emptyseq, message(snd, timeouttone, 0, 0, nil)))
[]
[(s eq 5) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq onhook)
                or (p_event(msg) neq flash) or (p_event(msg) neq i_disconnect)
                or (p_event(msg) neq i_connect) or (p_event(msg) neq i_timeout))] ->
   exit (5, emptyseq)
[]
[(s eq 6) and (p_event(msg) eq i_disconnect)] ->
   exit (6, emptyseq)
[]
[(s eq 6) and (p_event(msg) eq onhook)] ->
   exit (0, add(add(add(emptyseq, message(snd, o_inform, 0, 0, originator)),
                                  message(snd, o_notify , 0, 0,  nil )),
                                  message(snd, o_notify , 0, 0,  nil )))
[]
[(s eq 6) and (p_event(msg) eq i_alert )] ->
   exit (6, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
[]
[(s eq 6) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq onhook)
                or (p_event(msg) neq i_disconnect))] ->
   exit (6, emptyseq)
[]
[(s eq 7) and (p_event(msg) eq i_disconnect)] ->
   exit (0, emptyseq)
[]
[(s eq 7) and (p_event(msg) eq onhook)] ->
   exit (0, emptyseq)
[]
[(s eq 7) and (p_event(msg) eq i_alert )] ->
   exit (7, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
[]
[(s eq 7) and (p_event(msg) eq flash)] ->
   exit (8, add(emptyseq, message(snd, dialtone, 0, 0, nil )))
[]
[(s eq 7) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq i_disconnect)
                or (p_event(msg) neq onhook) or (p_event(msg) neq flash))] ->
   exit (7, emptyseq)
[]
[(s eq 8) and (p_event(msg) eq flash)] ->
   exit (7, emptyseq)
[]
[(s eq 8) and (p_event(msg) eq i_disconnect)] ->
   exit (8, emptyseq)
[]
[(s eq 8) and (p_event(msg) eq i_alert )] ->
   exit (8, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
[]
[(s eq 8) and (p_event(msg) eq dial)] ->
   exit (9, add(emptyseq, message(snd, o_alert , 0, 0, nil )))
[]
[(s eq 8) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq i_disconnect)
                or (p_event(msg) neq flash) or (p_event(msg) neq dial))] ->
   exit (8, emptyseq)
[]
[(s eq 9) and (p_event(msg) eq flash)] ->
   exit (7, add(emptyseq, message(snd, o_stopalert , 0, 0, nil )))
[]
[(s eq 9) and (p_event(msg) eq i_disconnect)] ->
   exit (9, emptyseq)
[]
[(s eq 9) and (p_event(msg) eq i_alert )] ->
   exit (9, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
[]
[(s eq 9) and (p_event(msg) eq i_free)] ->
   exit (11, add(emptyseq, message(snd, ringtone, 0, 0, nil )))
[]
[(s eq 9) and (p_event(msg) eq i_busy)] ->
   exit (10, add(emptyseq, message(snd, busytone, 0, 0, nil)))
[]
[(s eq 9) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq i_disconnect)
                or (p_event(msg) neq flash) or (p_event(msg) neq i_free )
                or (p_event(msg) neq i_busy))] ->
   exit (9, emptyseq)
[]
[(s eq 10) and (p_event(msg) eq flash)] ->
   exit (7, emptyseq)
[]
[(s eq 10) and (p_event(msg) eq i_disconnect)] ->
   exit (0, emptyseq)
[]
[(s eq 10) and (p_event(msg) eq onhook)] ->
   exit (0, emptyseq)
[]
[(s eq 10) and (p_event(msg) eq i_alert )] ->
   exit (10, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
[]
[(s eq 10) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq onhook)
                or (p_event(msg) neq flash) or (p_event(msg) neq i_disconnect))] ->
   exit (10, emptyseq)
```

```
      []
      [(s eq 11) and (p_event(msg) eq flash)] ->
         exit (7, add(emptyseq, message(snd, o_stopalert , 0, 0, nil )))
      []
      [(s eq 11) and (p_event(msg) eq i_disconnect)] ->
         exit (11, emptyseq)
      []
      [(s eq 11) and (p_event(msg) eq onhook)] ->
         exit (0, add(emptyseq, message(snd, o_stopalert , 0, 0, nil )))
      []
      [(s eq 11) and (p_event(msg) eq i_alert )] ->
         exit (11, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
      []
      [(s eq 11) and (p_event(msg) eq i_connect)] ->
         exit (12, add(add(emptyseq, message(snd, billing_start , 0, 0, time)),
                                     message(snd, connect , 0, 0, nil )))
      []
      [(s eq 11) and (p_event(msg) eq i_timeout)] ->
         exit (10, add(emptyseq, message(snd, timeouttone, 0, 0, nil )))
      []
      [(s eq 11) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq onhook)
                   or (p_event(msg) neq flash) or (p_event(msg) neq i_disconnect)
                   or (p_event(msg) neq i_connect) or (p_event(msg) neq i_timeout))] ->
         exit (11, emptyseq)
      []
      [(s eq 12) and (p_event(msg) eq i_disconnect)] ->
         exit (12, emptyseq)
      []
      [(s eq 12) and (p_event(msg) eq onhook)] ->
         exit (0, add(add(emptyseq, message(snd, o_notify, 0, 0, nil )),
                                     message(snd, o_notify , 0, 0, nil )))
      []
      [(s eq 12) and (p_event(msg) eq i_alert )] ->
         exit (12, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
      []
      [(s eq 12) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq onhook)
                   or (p_event(msg) neq i_disconnect))] ->
         exit (12, emptyseq)
    )
    []
    [fid eq 14] -> (* Call Transfer Everyone *)
    (
      [(s eq 0) and (p_event(msg) eq i_connect)] ->
         exit (2, emptyseq)
      []
      [(s eq 0) and (p_event(msg) eq o_connect)] ->
         exit (1, emptyseq)
      []
      [(s eq 0) and ((p_event(msg) neq o_connect) or (p_event(msg) neq i_connect))] ->
         exit (0, emptyseq)
      []
      [(s eq 1) and (p_event(msg) eq i_disconnect)] ->
         exit (0, emptyseq)
      []
      [(s eq 1) and (p_event(msg) eq onhook)] ->
         exit (0, emptyseq)
      []
      [(s eq 1) and (p_event(msg) eq i_alert )] ->
         exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
      []
      [(s eq 1) and (p_event(msg) eq i_inform)] ->
         exit (2, emptyseq)
      []
      [(s eq 1) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq i_disconnect)
                   or (p_event(msg) neq onhook) or (p_event(msg) neq i_inform))] ->
         exit (1, emptyseq)
      []
      [(s eq 2) and (p_event(msg) eq i_disconnect)] ->
         exit (0, emptyseq)
      []
      [(s eq 2) and (p_event(msg) eq onhook)] ->
         exit (0, emptyseq)
      []
      [(s eq 2) and (p_event(msg) eq i_alert )] ->
         exit (2, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
      []
      [(s eq 1) and (p_event(msg) eq i_notify )] ->
         exit (4, emptyseq)
      []
      [(s eq 2) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq i_disconnect)
                   or (p_event(msg) neq onhook) or (p_event(msg) neq i_notify))] ->
         exit (2, emptyseq)
      []
      [(s eq 3) and (p_event(msg) eq i_alert )] ->
         exit (3, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
      []
      [(s eq 3) and (p_event(msg) eq i_notify )] ->
         exit (4, emptyseq)
      []
      [(s eq 3) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq i_notify ))] ->
         exit (3, emptyseq)
      []
      [(s eq 4) and (p_event(msg) eq i_disconnect)] ->
         exit (0, emptyseq)
```

```
        []
        [(s eq 4) and (p_event(msg) eq onhook)] ->
           exit (0, emptyseq)
        []
        [(s eq 4) and (p_event(msg) eq i_alert)] ->
           exit (4, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 4) and ((p_event(msg) neq i_alert) or (p_event(msg) neq i_disconnect)
                       or (p_event(msg) neq onhook))] ->
           exit (4, emptyseq)
      )
      []
      [fid eq 15] -> (* Call Waiting *)
      (
        [(s eq 0) and (p_event(msg) eq i_connect)] ->
           exit (1, emptyseq)
        []
        [(s eq 0) and (p_event(msg) eq o_connect)] ->
           exit (1, emptyseq)
        []
        [(s eq 0) and ((p_event(msg) neq o_connect) or (p_event(msg) neq i_connect))] ->
           exit (0, emptyseq)
        []
        [(s eq 1) and (p_event(msg) eq i_disconnect)] ->
           exit (0, emptyseq)
        []
        [(s eq 1) and (p_event(msg) eq onhook)] ->
           exit (0, emptyseq)
        []
(*        [(s eq 1) and (p_event(msg) eq i_alert)] -> *)
(*           exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil))) *)
(*        [] *)
        [(s eq 1) and (p_event(msg) eq i_alert)] ->
           exit (2, add(emptyseq, message(snd, o_inform, 0, 0, cwhold)))
        []
        [(s eq 1) and ((p_event(msg) neq i_alert) or (p_event(msg) neq i_disconnect)
                       or (p_event(msg) neq onhook))] ->
           exit (1, emptyseq)
        []
        [(s eq 2) and (p_event(msg) eq i_alert)] ->
           exit (2, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 2) and (p_event(msg) eq flash)] ->
           exit (3, add(add(add(add(emptyseq, message(snd, billing_stop, 0, 0, time)),
                                              message(snd, o_inform, 0, 0, cwhold)),
                                              message(snd, billing_start , 0, 0, time)),
                                              message(snd, o_connect, 0, 0, nil)))
        []
        [(s eq 2) and (p_event(msg) eq i_disconnect)] ->
           exit (0, emptyseq)
        []
        [(s eq 2) and (p_event(msg) eq onhook)] ->
           exit (0, add(emptyseq, message(snd, o_disconnect, 0, 0, nil)))
        []
        [(s eq 2) and ((p_event(msg) neq i_alert) or (p_event(msg) neq i_disconnect)
                       or (p_event(msg) neq onhook) or (p_event(msg) neq flash))] ->
           exit (2, emptyseq)
        []
        [(s eq 3) and (p_event(msg) eq i_alert)] ->
           exit (3, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 3) and (p_event(msg) eq flash)] ->
           exit (2, add(add(add(add(emptyseq, message(snd, billing_stop, 0, 0, time)),
                                              message(snd, o_inform, 0, 0, cwhold)),
                                              message(snd, billing_start , 0, 0, time)),
                                              message(snd, o_connect, 0, 0, nil)))
        []
        [(s eq 3) and (p_event(msg) eq i_disconnect)] ->
           exit (0, emptyseq)
        []
        [(s eq 3) and (p_event(msg) eq onhook)] ->
           exit (0, add(emptyseq, message(snd, o_disconnect, 0, 0, nil)))
        []
        [(s eq 3) and ((p_event(msg) neq i_alert) or (p_event(msg) neq i_disconnect)
                       or (p_event(msg) neq onhook) or (p_event(msg) neq flash))] ->
           exit (3, emptyseq)

      )
      []
      [fid eq 16] -> (* Call Waiting Everyone *)
      (
        [(s eq 0) and (p_event(msg) eq i_inform)] ->
           exit (1, add(emptyseq, message(snd, cwtone, 0, 0, nil)))
        []
        [(s eq 0) and (p_event(msg) neq i_inform)] ->
           exit (0, emptyseq)
        []
        [(s eq 1) and (p_event(msg) eq i_alert)] ->
           exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 1) and (p_event(msg) eq i_connect)] ->
           exit (1, add(emptyseq, message(snd, connect, 0, 0, nil)))
        []
        [(s eq 1) and (p_event(msg) eq onhook)] ->
```

```
            exit (0, emptyseq)
          []
          [(s eq 1) and (p_event(msg) eq i_disconnect)] ->
            exit (0, emptyseq)
          []
          [(s eq 1) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq i_disconnect)
                        or (p_event(msg) neq onhook) or (p_event(msg) neq i_connect))] ->
            exit (1, emptyseq)
        )
        []
        [fid eq 19] -> (* Group Ringing *)
        (
          [(s eq 0) and (p_event(msg) eq i_alert)] ->
            exit (1, add(add(emptyseq, message(snd, o_alert, 0, 0, nil )),
                                    message(snd, o_alert , 0, 0, nil )))
          []
          [(s eq 0) and (p_event(msg) neq i_alert)] ->
            exit (0, emptyseq)
          []
          [(s eq 1) and (p_event(msg) eq i_stopalert)] ->
            exit (0, add(add(emptyseq, message(snd, o_stopalert, 0, 0, nil )),
                                    message(snd, o_stopalert , 0, 0, nil )))
          []
          [(s eq 1) and (p_event(msg) eq i_alert)] ->
            exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
          []
          [(s eq 1) and (p_event(msg) eq i_busy)] ->
            exit (2, emptyseq)
          []
          [(s eq 1) and (p_event(msg) eq i_free)] ->
            exit (3, emptyseq)
          []
          [(s eq 1) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq i_stopalert)
                        or (p_event(msg) neq i_busy) or (p_event(msg) neq i_free))] ->
            exit (1, emptyseq)
          []
          [(s eq 2) and (p_event(msg) eq i_alert)] ->
            exit (2, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
          []
          [(s eq 2) and (p_event(msg) eq i_busy)] ->
            exit (0, emptyseq)
          []
          [(s eq 2) and (p_event(msg) eq i_free)] ->
            exit (3, emptyseq)
          []
          [(s eq 2) and (p_event(msg) eq i_stopalert)] ->
            exit (0, add(emptyseq, message(snd, o_stopalert , 0, 0, nil )))
          []
          [(s eq 2) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq i_stopalert)
                        or (p_event(msg) neq i_busy) or (p_event(msg) neq i_free))] ->
            exit (2, emptyseq)
          []
          [(s eq 3) and (p_event(msg) eq i_alert)] ->
            exit (3, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
          []
          [(s eq 3) and (p_event(msg) eq i_free)] ->
            exit (3, emptyseq)
          []
          [(s eq 3) and (p_event(msg) eq i_stopalert)] ->
            exit (0, add(emptyseq, message(snd, o_stopalert , 0, 0, nil )))
          []
          [(s eq 3) and (p_event(msg) eq i_connect)] ->
            exit (0, add(add(add(add(add(add(add(emptyseq, message(snd, o_stopalert, 0, 0, nil)),
                                                  message(snd, stopalert , 0, 0, nil )),
                                                  message(snd, billing_forwarded , 0, 0, nil )),
                                                  message(snd, o_free , 0, 0, nil )),
                                                  message(snd, o_notify , 0, 0, nil )),
                                                  message(snd, o_notify , 0, 0, nil )),
                                                  message(snd, o_connect , 0, 0, nil )))
          []
          [(s eq 3) and ((p_event(msg) neq i_alert ) or (p_event(msg) neq i_stopalert)
                        or (p_event(msg) neq i_connect) or (p_event(msg) neq i_free))] ->
            exit (3, emptyseq)
        )
        []
        [fid eq 18] -> (* Group Ringing Everyone *)
        (
          [(s eq 0) and (p_event(msg) eq o_connect)] ->
            exit (1, emptyseq)
          []
          [(s eq 0) and (p_event(msg) eq i_connect)] ->
            exit (2, emptyseq)
          []
          [(s eq 0) and ((p_event(msg) neq o_connect) or (p_event(msg) neq i_connect))] ->
            exit (0, emptyseq)
          []
          [(s eq 1) and (p_event(msg) eq onhook)] ->
            exit (0, emptyseq)
          []
          [(s eq 1) and (p_event(msg) eq i_disconnect)] ->
            exit (0, emptyseq)
          []
          [(s eq 1) and (p_event(msg) eq i_alert)] ->
            exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil )))
```

```
        []
        [(s eq 1) and (p_event(msg) eq i_notify)] ->
           exit (3, emptyseq)
        []
        [(s eq 1) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_disconnect)
                       or (p_event(msg) neq i_notify) or (p_event(msg) neq i_alert))] ->
           exit (1, emptyseq)
        []
        [(s eq 2) and (p_event(msg) eq onhook)] ->
           exit (0, emptyseq)
        []
        [(s eq 2) and (p_event(msg) eq i_disconnect)] ->
           exit (0, emptyseq)
        []
        [(s eq 2) and (p_event(msg) eq i_alert)] ->
           exit (2, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 2) and (p_event(msg) eq i_notify)] ->
           exit (4, emptyseq)
        []
        [(s eq 2) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_disconnect)
                       or (p_event(msg) neq i_notify) or (p_event(msg) neq i_alert))] ->
           exit (2, emptyseq)
        []
        [(s eq 3) and (p_event(msg) eq onhook)] ->
           exit (0, add(emptyseq, message(snd, o_disconnect, 0, 0, nil)))
        []
        [(s eq 3) and (p_event(msg) eq i_disconnect)] ->
           exit (0, emptyseq)
        []
        [(s eq 3) and (p_event(msg) eq i_alert)] ->
           exit (3, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 3) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_disconnect)
                       or (p_event(msg) neq i_alert))] ->
           exit (3, emptyseq)
        []
        [(s eq 4) and (p_event(msg) eq onhook)] ->
           exit (0, add(emptyseq, message(snd, o_disconnect, 0, 0, nil)))
        []
        [(s eq 4) and (p_event(msg) eq i_connect)] ->
           exit (0, emptyseq)
        []
        [(s eq 4) and (p_event(msg) eq i_alert)] ->
           exit (4, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 4) and (p_event(msg) eq i_timeout)] ->
           exit (0, emptyseq)
        []
        [(s eq 4) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_disconnect)
                       or (p_event(msg) neq i_alert) or (p_event(msg) neq i_timeout))] ->
           exit (4, emptyseq)
     )
     []
     [fid eq 19] -> (* Ringback *)
     (
        [(s eq 0) and (p_event(msg) eq offhook)] ->
           exit (1, emptyseq)
        []
        [(s eq 0) and (p_event(msg) neq offhook)] ->
           exit (0, emptyseq)
        []
(*        [(s eq 1) and (p_event(msg) eq onhook)] -> *) (* cases for no message stored *)
(*           exit (0, emptyseq)            *)
(*        []                               *)
(*        [(s eq 1) and (p_event(msg) eq i_stopalert)] ->
(*           exit (0, emptyseq)            *)
(*        []                               *)
(*        [(s eq 1) and (p_event(msg) eq o_timeout)] ->
(*           exit (0, emptyseq)            *)
(*        []                               *)
        [(s eq 1) and (p_event(msg) eq i_alert)] ->
           exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 1) and (p_event(msg) eq i_stopalert)] ->
           exit (2, add(emptyseq, message(snd, alert, 0, 0, fastalert)))
        []

        [(s eq 1) and (p_event(msg) eq onhook)] ->
           exit (2, add(emptyseq, message(snd, alert, 0, 0, fastalert)))
        []
        [(s eq 1) and (p_event(msg) eq o_timeout)] ->
           exit (2, add(emptyseq, message(snd, alert, 0, 0, fastalert)))
        []
        [(s eq 1) and ((p_event(msg) neq o_timeout) or (p_event(msg) neq i_stopalert)
                       or (p_event(msg) neq onhook))] ->
           exit (1, emptyseq)
        []
        [(s eq 2) and (p_event(msg) eq offhook)] ->
           exit (0, add(emptyseq, message(snd, o_alert, 0, 0, nil)))
        []
        [(s eq 2) and (p_event(msg) eq i_alert)] ->
           exit (2, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
```

```
      [(s eq 2) and ((p_event(msg) neq offhook) or (p_event(msg) neq i_alert)] ->
         exit (2, emptyseq)
    )
    []
    [fid eq 20] -> (* Ringback Everyone 1 *)
    (
      [(s eq 0) and (p_event(msg) eq i_inform)] ->
         exit (0, add(emptyseq, message(snd, announce, 0, 0, ringback)))
      []
      [(s eq 0) and (p_event(msg) neq i_inform)] ->
         exit (0, emptyseq)
    )
    []
    [fid eq 21] -> (* Ringback Everyone 2 *)
    (
      [(s eq 0) and (p_event(msg) eq i_alert)] ->
         exit (0, add(emptyseq, message(snd, o_inform, 0, 0, ringback)))
      []
      [(s eq 0) and (p_event(msg) neq i_alert)] ->
         exit (0, emptyseq)
    )
    []
    [fid eq 22] -> (* Three Way Calling *)
    (
      [(s eq 0) and (p_event(msg) eq i_connect)] ->
         exit (1, emptyseq)
      []
      [(s eq 0) and (p_event(msg) eq o_connect)] ->
         exit (1, emptyseq)
      []
      [(s eq 0) and ((p_event(msg) neq o_connect) or (p_event(msg) neq i_connect))] ->
         exit (0, emptyseq)
      []
      [(s eq 1) and (p_event(msg) eq i_disconnect)] ->
         exit (0, emptyseq)
      []
      [(s eq 1) and (p_event(msg) eq onhook)] ->
         exit (0, emptyseq)
      []
      [(s eq 1) and (p_event(msg) eq i_alert)] ->
         exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
      []
      [(s eq 1) and (p_event(msg) eq flash)] ->
         exit (2, add(emptyseq, message(snd, dialtone, 0, 0, nil)))
      []
      [(s eq 1) and ((p_event(msg) neq i_alert) or (p_event(msg) neq i_disconnect)
                    or (p_event(msg) neq onhook) or (p_event(msg) neq flash))] ->
         exit (1, emptyseq)
      []
      [(s eq 2) and (p_event(msg) eq flash)] ->
         exit (1, emptyseq)
      []
      [(s eq 2) and (p_event(msg) eq i_disconnect)] ->
         exit (2, emptyseq)
      []
      [(s eq 2) and (p_event(msg) eq i_alert)] ->
         exit (2, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
      []
      [(s eq 2) and (p_event(msg) eq dial)] ->
         exit (3, add(emptyseq, message(snd, o_alert, 0, 0, nil)))
      []
      [(s eq 2) and ((p_event(msg) neq i_alert) or (p_event(msg) neq i_disconnect)
                    or (p_event(msg) neq flash) or (p_event(msg) neq dial))] ->
         exit (2, emptyseq)
      []
      [(s eq 3) and (p_event(msg) eq flash)] ->
         exit (1, add(emptyseq, message(snd, o_stopalert, 0, 0, nil)))
      []
      [(s eq 3) and (p_event(msg) eq i_disconnect)] ->
         exit (3, emptyseq)
      []
      [(s eq 3) and (p_event(msg) eq i_alert)] ->
         exit (3, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
      []
      [(s eq 3) and (p_event(msg) eq i_free)] ->
         exit (5, add(emptyseq, message(snd, ringtone, 0, 0, nil)))
      []
      [(s eq 3) and (p_event(msg) eq i_busy)] ->
         exit (4, add(emptyseq, message(snd, busytone, 0, 0, nil)))
      []
      [(s eq 3) and ((p_event(msg) neq i_alert) or (p_event(msg) neq i_disconnect)
                    or (p_event(msg) neq flash) or (p_event(msg) neq i_free)
                    or (p_event(msg) neq i_busy))] ->
         exit (3, emptyseq)
      []
      [(s eq 4) and (p_event(msg) eq flash)] ->
         exit (1, emptyseq)
      []
      [(s eq 4) and (p_event(msg) eq i_disconnect)] ->
         exit (0, emptyseq)
      []
      [(s eq 4) and (p_event(msg) eq onhook)] ->
         exit (0, emptyseq)
      []
```

```
    [(s eq 4) and (p_event(msg) eq i_alert)] ->
      exit (4, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
    []
    [(s eq 4) and ((p_event(msg) neq i_alert) or (p_event(msg) neq onhook)
                  or (p_event(msg) neq flash) or (p_event(msg) neq i_disconnect))] ->
      exit (4, emptyseq)
    []
    [(s eq 5) and (p_event(msg) eq flash)] ->
      exit (1, add(emptyseq, message(snd, o_stopalert, 0, 0, nil)))
    []
    [(s eq 5) and (p_event(msg) eq i_disconnect)] ->
      exit (5, emptyseq)
    []
    [(s eq 5) and (p_event(msg) eq onhook)] ->
      exit (0, add(emptyseq, message(snd, o_stopalert, 0, 0, nil)))
    []
    [(s eq 5) and (p_event(msg) eq i_alert)] ->
      exit (5, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
    []
    [(s eq 5) and (p_event(msg) eq i_connect)] ->
      exit (6, add(add(emptyseq, message(snd, billing_start, 0, 0, time)),
                              message(snd, connect, 0, 0, nil)))
    []
    [(s eq 5) and (p_event(msg) eq i_timeout)] ->
      exit (4, add(emptyseq, message(snd, timeouttone, 0, 0, nil)))
    []
    [(s eq 5) and ((p_event(msg) neq i_alert) or (p_event(msg) neq onhook)
                  or (p_event(msg) neq flash) or (p_event(msg) neq i_disconnect)
                  or (p_event(msg) neq i_connect) or (p_event(msg) neq i_timeout))] ->
      exit (5, emptyseq)
    []
    [(s eq 6) and (p_event(msg) eq i_disconnect)] ->
      exit (0, emptyseq)
    []
    [(s eq 6) and (p_event(msg) eq onhook)] ->
      exit (0, add(emptyseq, message(snd, o_disconnect, 0, 0, nil)))
    []
    [(s eq 6) and (p_event(msg) eq i_alert)] ->
      exit (6, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
    []
    [(s eq 6) and (p_event(msg) eq flash)] ->
      exit (7, emptyseq)
    []
    [(s eq 6) and ((p_event(msg) neq i_alert) or (p_event(msg) neq onhook)
                  or (p_event(msg) neq flash) or (p_event(msg) neq i_disconnect))] ->
      exit (6, emptyseq)
    [(s eq 7) and (p_event(msg) eq i_disconnect)] ->
      exit (0, emptyseq)
    []
    [(s eq 7) and (p_event(msg) eq onhook)] ->
      exit (0, add(emptyseq, message(snd, o_disconnect, 0, 0, nil)))
    []
    [(s eq 7) and (p_event(msg) eq i_alert)] ->
      exit (7, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
    []
    [(s eq 7) and ((p_event(msg) neq i_alert) or (p_event(msg) neq onhook)
                  or (p_event(msg) neq i_disconnect))] ->
      exit (7, emptyseq)
  )
  []
  [fid eq 23] -> (* Voicemail Recording *)
  (
    [(s eq 0) and (p_event(msg) eq o_timeout)] -> *) (* no msg *)
      exit (1, add(emptyseq, message(snd, o_inform, 0, 0, callminder)))
    []
    [(s eq 0) and (p_event(msg) neq o_timeout)] ->
      exit (0, emptyseq)
    []
    [(s eq 1) and (p_event(msg) eq i_alert)] ->
      exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
    []
    [(s eq 1) and (p_event(msg) eq i_disconnect)] ->
      exit (0, emptyseq)
    []
    [(s eq 1) and (p_event(msg) eq i_msg)] ->
      exit (0, add(emptyseq, message(snd, storemsg, 0, 0, nil)))
    []
    [(s eq 1) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_alert)
                  or (p_event(msg) neq i_msg))] ->
      exit (1, emptyseq)
  )
  []
  [fid eq 24] -> (* Voicemail Everyone *)
  (
    [(s eq 0) and (p_event(msg) eq i_inform)] -> *) (* no msg *)
      exit (1, add(emptyseq, message(snd, announce, 0, 0, callminder)))
    []
    [(s eq 0) and (p_event(msg) neq i_inform)] ->
      exit (0, emptyseq)
    []
    [(s eq 1) and (p_event(msg) eq i_alert)] ->
      exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
    []
    [(s eq 1) and (p_event(msg) eq onhook)] ->
```

```
          exit (0, emptyseq)
        []
        [(s eq 1) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_alert))] ->
          exit (1, emptyseq)
      )
      []
      [fid eq 25] -> (* Voicemail Playback *)
      (
(*        [(s eq 0) and (p_event(msg) eq dial)] -> *) (* no msg *)
(*          exit (1, add(emptyseq, message(snd, announce, 0, 0, nomsg)))
(*        []*)
        [(s eq 0) and (p_event(msg) eq dial)] -> *) (* no msg *)
          exit (1, add(add(add(emptyseq, message(snd, storeread, 0, 0, msgtext)),
                                          message(snd, announce, 0, 0, msgtext)),
                                          message(snd, storeclear , 0, 0,  nil)))
        []
        [(s eq 0) and (p_event(msg) neq dial)] ->
          exit (0, emptyseq)
        []
        [(s eq 1) and (p_event(msg) eq i_alert)] ->
          exit (1, add(emptyseq, message(snd, o_busy, 0, 0, nil)))
        []
        [(s eq 1) and (p_event(msg) eq onhook)] ->
          exit (0, emptyseq)
        []
        [(s eq 1) and ((p_event(msg) neq onhook) or (p_event(msg) neq i_alert))] ->
          exit (1, emptyseq)
      )
    )
  endproc (* FeatBehave *)

(*
 Transaction Control
*)
  process TransControl (s: nat, rbstack: t_natstack, msg: t_msg): exit (t_natstack, nat, bool) :=
  (
    [(p_event(msg) eq abort_transaction)] ->
       exit (emptystack, bottom (rbstack), false)
        (* clear the stack, return the bootom element as new state *)
    []
    [(p_event(msg) eq commit_transaction)] ->
       exit (emptystack, bottom (rbstack), true)
        (* same as abort, get back to state before transaction because msgs are played back *)
    []
    [(p_event(msg) eq rollback_transaction)] ->
       exit (pop (rbstack), top (rbstack), false)
        (* pop the topmost state, this is the new state *)
    []
    [(p_event(msg) eq start_transaction)] ->
       exit (push (rbstack, s), s, false)
        (* we need to save the current state onto the stack *)
  )
  endproc (* TransControl *)

(*
 Prelimary Trigger Generator
*)
  process TriggerGen[Out, In0]: noexit :=
  (
      (* Triggers from user *)
      Out ! message(rcv, dial , 0, 0,  nil ) (*| C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
        TriggerGen[Out, In0]
    []
      Out ! message(rcv, offhook , 0, 0,  nil ) (*| C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
        TriggerGen[Out, In0]
    []
      Out ! message(rcv, onhook, 0, 0, nil ) (*| C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
        TriggerGen[Out, In0]
    []
      Out ! message(rcv, flash , 0, 0,  nil ) (*| C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
        TriggerGen[Out, In0]
    []
      (* Triggers from remote end of call *)
      Out ! message(rcv, i_notify , 0, 0,  nil ) (*| C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
        TriggerGen[Out, In0]
    []
      Out ! message(rcv, i_inform , 0, 0,  nil ) (*| C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
        TriggerGen[Out, In0]
    []
      Out ! message(rcv, i_alert  , 0, 0,  nil ) (*| C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
        TriggerGen[Out, In0]
    []
      Out ! message(rcv, i_free  , 0, 0,  nil ) (*| C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
        TriggerGen[Out, In0]
    []
      Out ! message(rcv, i_busy , 0, 0,  nil ) (*| C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
        TriggerGen[Out, In0]
    []
      Out ! message(rcv, i_stopalert , 0, 0,  nil ) (*| C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
        TriggerGen[Out, In0]
    []
      Out ! message(rcv, i_timeout , 0, 0,  nil ) (*| C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
        TriggerGen[Out, In0]
    []
```

```
        Out ! message(rcv, i_disconnect , 0, 0,  nil ) (*|  C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
          TriggerGen[Out, In0]
      []
        Out ! message(rcv, i_connect , 0, 0,  nil ) (*|  C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
          TriggerGen[Out, In0]
      []
        Out ! message(rcv, i_msg , 0, 0,  nil ) (*|  C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
          TriggerGen[Out, In0]
      [] (*  to create  closed  system *)
        Out ! message(rcv, i_request , 0, 0,  nil ) (*|  C printf ("TGen ! %s \n", kd_draw ($!1)); |*);
          TriggerGen[Out, In0]
      [] (*  to create  closed  system *)
        In0 ? m: t_msg;
          TriggerGen[Out, In0]
      [] (*  to create  closed  system *)
        In0 ? m: t_tree;
          TriggerGen[Out, In0]
    )
  endproc (* TriggerGen *)

endspec
```

B Perl Code Customiser

```
#!/usr/bin/perl

$fname_start = "fm_impl";
$fname_end  = "feat.lot";

# How many features will there be? No checking of input correctness ...
$ok = 0;
while ($ok != 1) {
  print STDOUT "Enter the number of features (2..8): ";
  chop($nooffeat = <STDIN>); # need to remove the trailing nl
  if  (($nooffeat > 1) && ($nooffeat < 26)) {
    $ok = 1;
  }
}
$fname = $fname_start . $nooffeat . $fname_end;
print $fname;


# for 1 .. nooffeat ask which one?
for ($count = 0; $count < $nooffeat; $count++) {
  print STDOUT "Please chose which feature you want <number>: \n";
  print STDOUT "1. TL\n";
  print STDOUT "2. CFB\n";
  print STDOUT "3. CFB_E\n";
  print STDOUT "4. TCS\n";
  print STDOUT "5. TCS_E1\n";
  print STDOUT "6. TCS_E2\n";
  print STDOUT "7. RC\n";
  print STDOUT "8. SB\n";
  print STDOUT "9. BCS\n";
  print STDOUT "10: CND\n";
  print STDOUT "11: CND_Everyone\n";
  print STDOUT "12: CNDB\n";
  print STDOUT "13: Call_Transfer\n";
  print STDOUT "14: Call_Transfer_Everyone\n";
  print STDOUT "15: Call_Waiting\n";
  print STDOUT "16: Call_Waiting_Everyone\n";
  print STDOUT "17: Group_Ringing\n";
  print STDOUT "18: Group_Ringing_Everyone\n";
  print STDOUT "19: Ringback\n";
  print STDOUT "20: Ringback_Everyone1\n";
  print STDOUT "21: Ringback_Everyone2\n";
  print STDOUT "22: Three_Way_Calling\n";
  print STDOUT "23: Voice_Mail - record\n";
  print STDOUT "24: Voice_Mail_Everyone\n";
  print STDOUT "25: Voice_Mail - playback\n";
  chop($tempfid = <STDIN>);
  @fid{$count} = $tempfid;
}

# open files for template and destination
open(TEMPLATE, "fm_template.lot");
$fname = $fname_start . $nooffeat . $fname_end;
open(MODEL, ">$fname");

# print header
print MODEL "(* \n" .
            "  Version 1.3 of the telephone switch with feature manager \n".
            "  GENERATED FROM TEMPLATE \n" .
            "  Stephan Reiff\n" .
            "  Started:     17-07-2000\n" .
            "  Last Change: 06-10-2000 (to template)\n\n" .
            "  MODEL WITH $nooffeat FEATURES \n \n";
print MODEL "  The features are:\n";
for ($count = 0; $count < $nooffeat; $count++) {
  print MODEL "    *  @fid{$count}";
  switch: {
```

```
      if (@fid{$count} == 1) {print MODEL "␣␣␣Teenline\n"; last switch;}
      if (@fid{$count} == 2) {print MODEL "␣␣␣Call␣Forwarding␣Busy\n"; last switch;}
      if (@fid{$count} == 3) {print MODEL "␣␣␣Call␣Forwarding␣Busy␣Everyone\n"; last switch;}
      if (@fid{$count} == 4) {print MODEL "␣␣␣Terminating␣Call␣Screening\n"; last switch;}
      if (@fid{$count} == 5) {print MODEL "␣␣␣Terminating␣Call␣Screening␣Everyone\n"; last switch;}
      if (@fid{$count} == 6) {print MODEL "␣␣␣TCS_E2\n"; last switch;}
      if (@fid{$count} == 7) {print MODEL "␣␣␣RC\n"; last switch;}
      if (@fid{$count} == 8) {print MODEL "␣␣␣SB\n"; last switch;}
      if (@fid{$count} == 9) {print MODEL "␣␣␣BCS\n"; last switch;}
      if (@fid{$count} == 10) {print MODEL "␣␣␣CND\n"; last switch;}
      if (@fid{$count} == 11) {print MODEL "␣␣␣CND_Everyone\n"; last switch;}
      if (@fid{$count} == 12) {print MODEL "␣␣␣CNDB\n"; last switch;}
      if (@fid{$count} == 13) {print MODEL "␣␣␣Call␣Transfer\n"; last switch;}
      if (@fid{$count} == 14) {print MODEL "␣␣␣Call␣Transfer␣Everyone\n"; last switch;}
      if (@fid{$count} == 15) {print MODEL "␣␣␣Call␣Waiting\n"; last switch;}
      if (@fid{$count} == 16) {print MODEL "␣␣␣Call␣Waiting␣Everyone\n"; last switch;}
      if (@fid{$count} == 17) {print MODEL "␣␣␣Group␣Ringing\n"; last switch;}
      if (@fid{$count} == 18) {print MODEL "␣␣␣Group␣Ringing␣Everyone\n"; last switch;}
      if (@fid{$count} == 19) {print MODEL "␣␣␣Ringback\n"; last switch;}
      if (@fid{$count} == 20) {print MODEL "␣␣␣Ringback␣Everyone1\n"; last switch;}
      if (@fid{$count} == 21) {print MODEL "␣␣␣Ringback␣Everyone2\n"; last switch;}
      if (@fid{$count} == 22) {print MODEL "␣␣␣Three␣Way␣Calling\n"; last switch;}
      if (@fid{$count} == 23) {print MODEL "␣␣␣Voice␣Mail␣-␣record\n"; last switch;}
      if (@fid{$count} == 24) {print MODEL "␣␣␣Voice␣Mail_Everyone\n"; last switch;}
      if (@fid{$count} == 25) {print MODEL "␣␣␣Voice␣Mail␣-␣playback\n"; last switch;}
      $nothing = 1;
    }
  }
  print MODEL "*)\n";

   $isfirst  = 0;
  for ($count = 0; $count < $nooffeat; $count++) {
    $temp = $count + 1;
    if ( $isfirst  == 0){ # insert the first term ...
      $gatesline = "In" . $temp;
      $queuesline = "q" . $temp;
       $collectline  = "c" . $temp;
      $tmsgseqline = "t_msgseq";
       $falseline  = "false";
      $emptyseqline = "emptyseq";
      $tstateopns = "␣␣␣␣␣␣p_q" . $temp . "␣␣␣␣␣␣(*!␣implementedby␣ADT_STATE_PQ" . $temp .
                    "␣*)␣␣␣␣␣␣␣:␣t_state␣->␣t_msgseq";
      $behavecocoon = "␣␣␣␣␣␣Cocoon␣[In" . $temp . ",␣Out]␣(0,␣emptystack,␣" . $temp . ",␣" . @fid{$count} . ")";
      $condition = "c" . $temp;
       $featuresline  = "f" . $temp;
      $nqueuesline = "nq" . $temp;
      $tmpqueuesline = "tmpq" . $temp;
      $tempqueuesline = "tempq" . $temp;
      $lengthline = "length(q" . $temp . ")";
      $topline = "p_q" . $temp . "(top(stk))";
      $posschoicesline  = "possiblechoices(q" . $temp . ",␣0)";
      $concatline = "concat(nq" . $temp . ",␣tempq" . $temp . ")";
      $makeqlist = "add(emptyqlist,␣q" . $temp . ")";
      $helpdq1 = "ql2";
      $dequeuelist = "getfirst (" . $helpdq1 . ")";
       $isfirst  = 1;
    }
    else {
      $gatesline = $gatesline . ",␣" . "In" . $temp;
      $queuesline = $queuesline . ",␣" . "q" . $temp;
       $collectline  = $collectline  . ",␣" . "c" . $temp;
      $tmsgseqline = $tmsgseqline . ",␣" . "t_msgseq";
       $falseline  = $falseline  . ",␣false";
      $emptyseqline = $emptyseqline . ",␣emptyseq";
      $tstateopns = $tstateopns . "\n␣␣␣␣␣␣p_q" . $temp . "␣␣␣␣␣␣(*!␣implementedby␣ADT_STATE_PQ" . $temp .
                    "␣*)␣␣␣␣␣␣␣:␣t_state␣->␣t_msgseq";
      $behavecocoon = $behavecocoon . "\n␣␣␣␣␣␣|[Out]|\n␣␣␣␣␣␣Cocoon␣[In" . $temp . ",␣Out]␣(0,␣emptystack,␣" .
                      $temp . ",␣" . @fid{$count} . ")";
      $condition = "(" . $condition . "␣and␣c" . $temp . ")";
       $featuresline = $featuresline . ",␣" . "f" . $temp;
      $nqueuesline = $nqueuesline . ",␣" . "nq" . $temp;
      $tmpqueuesline = $tmpqueuesline . ",␣" . "tmpq" . $temp;
      $tempqueuesline = $tempqueuesline . ",␣" . "tempq" . $temp;
      $lengthline = "(" . $lengthline . "+␣length(q" . $temp . "))";
      $topline = $topline . ",␣" . "p_q" . $temp . "(top(stk))";
      $posschoicesline = "possiblechoices(q" . $temp . ",␣" . $posschoicesline . ")";
      $concatline = $concatline . ",␣" . "concat(nq" . $temp . ",␣tempq" . $temp . ")";
      $makeqlist = "add(" . $makeqlist . ",␣q" . $temp . ")";
      $helpdq1 = "remfirst(" . $helpdq1 . ")";
      $dequeuelist = $dequeuelist . ",␣getfirst (" . $helpdq1 . ")";
    }
  }

  # now the ones that use previous lines as subparts
   $isfirst  = 0;
  for ($count = 0; $count < $nooffeat; $count++) {
    $temp = $count + 1;
    # need some bits for collect
    $tempc = "";
    $tempq = "";
    for ($tcount = 0; $tcount < $nooffeat; $tcount++) {
      $temp2 = $tcount + 1;
      if ($temp2 == $temp){
        $tempc = $tempc . ",␣not(c" . $temp2 . ")";
```

```
          $tempq = $tempq . "add(q" . $temp2 . ",␣m),␣";
        }
        else{
          $tempc = $tempc . ",␣c" . $temp2;
          $tempq = $tempq . "q" . $temp2 . ",␣";
        }
      }
      # now do the proper source lines
      if ( $isfirst  == 0){ # insert the  first  term ...
        $collect  = "␣␣␣␣␣␣[not(c" . $temp . ")]␣−>␣␣␣(∗␣f" . $temp . "␣has␣not␣yet␣finished␣∗)\n" .
                    "␣␣␣␣␣␣(\n" .
                    "␣␣␣␣␣␣␣␣In" . $temp . "?m:␣t␣msg;␣(∗␣can␣read␣from␣f" . $temp . "␣∗)\n" .
                    "␣␣␣␣␣␣␣␣(\n" .
                    "␣␣␣␣␣␣␣␣[is␣rcomp(m)]␣−>␣(∗␣finished?␣set␣flag␣" . $temp . "␣∗)\n" .
                    "␣␣␣␣␣␣␣␣(\n" .
                    "␣␣␣␣␣␣␣␣␣␣CollectResp[" . $gatesline  . "](" . $queuesline .  $tempc . ")\n" .
                    "␣␣␣␣␣␣␣␣)\n" .
                    "␣␣␣␣␣␣␣␣[]\n" .
                    "␣␣␣␣␣␣␣␣[not(is␣rcomp(m))]␣−>␣(∗␣more␣actions␣required,␣add␣to␣queue␣" . $temp . "␣∗)\n" .
                    "␣␣␣␣␣␣␣␣(\n" .
                    "␣␣␣␣␣␣␣␣␣␣CollectResp[" . $gatesline  . "](" . $tempq . $collectline  . ")\n" .
                    "␣␣␣␣␣␣␣␣)\n" .
                    "␣␣␣␣␣␣␣␣)\n" .
                    "␣␣␣␣␣␣)␣␣(∗␣not␣c" . $temp . "␣∗)";
       $tstateeqns = "␣␣␣␣␣␣␣␣p␣q" . $temp . "(state(" . $queuesline . ",␣c))␣=␣q" . $temp . ";";
        $isfirst  = 1;
      }
      else {
        $collect  = $collect  . "\n" .
                    "␣␣␣␣␣␣[]\n" .
                    "␣␣␣␣␣␣[not(c" . $temp . ")]␣−>␣␣␣(∗␣f" . $temp . "␣has␣not␣yet␣finished␣∗)\n" .
                    "␣␣␣␣␣␣(\n" .
                    "␣␣␣␣␣␣␣␣In" . $temp . "?m:␣t␣msg;␣(∗␣can␣read␣from␣f" . $temp . "␣∗)\n" .
                    "␣␣␣␣␣␣␣␣(\n" .
                    "␣␣␣␣␣␣␣␣[is␣rcomp(m)]␣−>␣(∗␣finished?␣set␣flag␣" . $temp . "␣∗)\n" .
                    "␣␣␣␣␣␣␣␣(\n" .
                    "␣␣␣␣␣␣␣␣␣␣CollectResp[" . $gatesline  . "](" . $queuesline . $tempc . ")\n" .
                    "␣␣␣␣␣␣␣␣)\n" .
                    "␣␣␣␣␣␣␣␣[]\n" .
                    "␣␣␣␣␣␣␣␣[not(is␣rcomp(m))]␣−>␣(∗␣more␣actions␣required,␣add␣to␣queue␣" . $temp . "␣∗)\n" .
                    "␣␣␣␣␣␣␣␣(\n" .
                    "␣␣␣␣␣␣␣␣␣␣CollectResp[" . $gatesline  . "](" . $tempq . $collectline  . ")\n" .
                    "␣␣␣␣␣␣␣␣)\n" .
                    "␣␣␣␣␣␣␣␣)\n" .
                    "␣␣␣␣␣␣)␣␣(∗␣not␣c" . $temp . "␣∗)";
       $tstateeqns = $tstateeqns . "\n␣␣␣␣␣␣␣␣p␣q" . $temp . "(state(" . $queuesline . ",␣c))␣=␣q" . $temp . ";";
      }
    }

    # and finally there are some nasty ones for the feedback process ...
     $isfrst  = 0;
     $isfirst  = 0;
    for ($count = 0; $count < 2∗∗$nooffeat − 1; $count++) {
      $isfrst2  = 0;
      # need to do some prep for each case
        if ( $isfrst  == 0){
          #nbit bin no ...  all  features  set to true
          for($i = 0; $i < $nooffeat; $i++){
            @bitarray[$i] = 1;
           }
          $isfrst  = 1;
        }
        else {
          #dec bin number ... some features must be set to false  now
          for($i = $nooffeat − 1; $i >=0; $i−−){
            if (@bitarray[$i] == 1) {
              @bitarray[$i] = 0;
              $i = −1;
            }
            else {
              @bitarray[$i] = 1;
            }
          }
        }
      for ($count2 = 0; $count2 < $nooffeat; $count2++) {
        $temp = $count2 + 1;
        if ( $isfrst2  == 0){
          if (@bitarray[$count2] == 1){ # this feature is "on"
            $part1 = "q" . $temp;
            $part2 = "true";
            $part3 = "f" . $temp;
          }
          else {
            $part1 = "emptyseq";
            $part2 = "false";
            $part3 = "not(f" . $temp . ")";
          }
          $isfrst2  = 1;
        }
        else {
          if (@bitarray[$count2] == 1){ # this feature is "on"
            $part1 = $part1 . ",␣q" . $temp;
            $part2 = $part2 . ",␣true";
            $part3 = $part3 . "␣and␣f" . $temp;
```

```
        }
        else {
          $part1 = $part1 . ", emptyseq";
          $part2 = $part2 . ", false";
          $part3 = $part3 . " and not(f" . $temp . ")";
        }

      }
    }
    # now create the code ...
    if ( $isfirst  == 0){ # insert the  first  term ...
      $fbcontrol = "      FeedBack[Out, " . $gatesline . "](" . $part1 . ", emptyststack, origt, 0, " . $part2 . ")\n" .
                   "      >> accept t: t tree in";
      $ignoreresp = "        [" . $part3 . "] -> exit(" . $part1 . ")";
       $isfirst  = 1;
    }
    else {
      $fbcontrol = $fbcontrol . "\n" .
                   "      FeedBack[Out, " . $gatesline . "](" . $part1 . ", emptyststack, t, 0, " . $part2 . ")\n" .
                   "      >> accept t: t tree in";
      $ignoreresp = $ignoreresp . "\n" .
                    "        []\n" .
                    "        [" . $part3 . "] -> exit(" . $part1 . ")";
    }
  }

  while ($line = <TEMPLATE>){
    if ( $line =~ s/\(\*\# GATES \*\)/$gatesline/){}
    if ( $line =~ s/\(\*\# QUEUES \*\)/$queuesline/){}
    if ( $line =~ s/\(\*\# COLLECT \*\)/$collectline/){}
    if ( $line =~ s/\(\*\# TMSGSEQ \*\)/$tmsgseqline/){}
    if ( $line =~ s/\(\*\# FALSE \*\)/$falseline/){}
    if ( $line =~ s/\(\*\# EMPTYSEQ \*\)/$emptyseqline/){}
    if ( $line =~ s/\(\*\# TSTATEOPNS \*\)/$tstateopns/){}
    if ( $line =~ s/\(\*\# TSTATEEQNS \*\)/$tstateeqns/){}
    if ( $line =~ s/\(\*\# BEHAVECOCOON \*\)/$behavecocoon/){}
    if ( $line =~ s/\(\*\# COLLECTBLOCK \*\)/$collect/){}
    if ( $line =~ s/\(\*\# CONDITION \*\)/$condition/){}
    if ( $line =~ s/\(\*\# FEATURES \*\)/$featuresline/){}
    if ( $line =~ s/\(\*\# NQUEUES \*\)/$nqueuesline/){}
    if ( $line =~ s/\(\*\# TMPQUEUES \*\)/$tmpqueuesline/){}
    if ( $line =~ s/\(\*\# TEMPQUEUES \*\)/$tempqueuesline/){}
    if ( $line =~ s/\(\*\# LENGTH \*\)/$lengthline/){}
    if ( $line =~ s/\(\*\# TOP \*\)/$topline/){}
    if ( $line =~ s/\(\*\# POSSCHOICES \*\)/$posschoicesline/){}
    if ( $line =~ s/\(\*\# CONCAT \*\)/$concatline/){}
    if ( $line =~ s/\(\*\# FEEDBACKCONTROL \*\)/$fbcontrol/){}
    if ( $line =~ s/\(\*\# IGNORERESP \*\)/$ignoreresp/){}
    if ( $line =~ s/\(\*\# MAKEQLIST \*\)/$makeqlist/){}
    if ( $line =~ s/\(\*\# DEQUEUELIST \*\)/$dequeuelist/){}
    if ( $line =~ s/\(\*\# NOOFFEAT \*\)/$nooffeat/){}
    if ( $line =~ s/\#TEMPLATE.*\n//){}
    print MODEL $line

  }

  # close the files ...
  close(TEMPLATE);
  close(MODEL);
```

C Compiling Prototypes

A main C file is required, for the current example the file is as follows:

```
#include <stdio.h>
#include "lotos.hh"

main (argc, argv)
  int argc;
  char* argv[];
{
  int count;
  void **param;

  long random();
  konfig.random= random;
  srandom (getpid());
  //  kaos.debug = 1;

  //  konfig.trace= 1;
   ldcinit  ();
  cmd2lbc (argc, argv, &count, &param);
   lbcinit  (count, param);
  exit  (sch  ());
}
```

Topo extracts two further C files from the LOTOS code, `ldc.c` and `lbc.c` for the data part and the behavioural part of the specification respectively. In addition two topo libraries are required (lotos and kaos), but they are part of the default Topo installation.

The executable prototype is generated in the following steps:

```
topo model.lot -data
topo model.lot -behaviour
gcc -I/somepath/topolib -c ldc.c
gcc -I/somepath/topolib -c lbc.c
gcc -I/somepath/topolib -c topomain.c
gcc -I/somepath/topolib -o prototype lbc.o ldc.o topomain.o \\
                          -L/somepath/topolib -lotos -lkaos
```