Modelling Legacy Telecommunications Switching Systems for Interaction Analysis

Muffy Calder and Stephan Reiff
Department of Computing Science
University of Glasgow
Glasgow

email: muffy, sreiff@dcs.gla.ac.uk

1 Introduction

Telecommunications switching systems have evolved rapidly over the past two decades, from networks which supported only *POTS* (the plain old telephony service), to ones which support services such as call forwarding, 3-way calling, personal numbers, alternative charging, and virtual private networks. Forces behind this evolution include deregulated, global markets and changing social and business practices, as well as advances in the underlying software and hardware technologies.

Like most processes of change, this evolution can be a difficult process to manage, not least because it has continually thrown up a number of interworking problems. A fundamental source of these interworking problems derives from rapidly changing system requirements. These requirements, particularly those which involve network functionality, must respond rapidly to changing technological capabilities and social contexts. In turn, the implementation of these new requirements enable further technological, business and social changes, and so on. Thus, the scope for introducing new network functionality is never ending.

At each stage in the evolution, new additional functionality, i.e. new *features*, may *interact* with pre-existing features, in unanticipated ways. When one feature influences the behaviour of another, we refer to this situation as a *feature interaction*. To manage interworking, we must be able to *detect* any such feature interactions, and *resolve* them, in a suitable manner.

The telecommunications industry has invested heavily in developing software switching systems, many of which are extremely fragile and were developed without the benefit of modern software methods and technologies. These legacy systems cannot in many cases, for economic or technical reasons, be re-engineered. Interworking with them is clearly a challenge, as is interworking with any third party system – where access to system specifications or intentions is not possible.

The SEBPC-funded project *Hybrid Techniques for Detecting and Resolving Feature Interactions in Telecommunications Services*[1] aims to meet this challenge by developing a number of off-line (at the design stage) and on-line (at run-time) approaches to interworking problems. One main objective is to develop a *hybrid feature manager* which can mediate between new features and a legacy switching system. The manager is hybrid in that it will run in real-time, and be able to adapt to new services, yet in order to do so effectively it must be informed by an off-line analysis of features. The analysis will provide the knowledge and theories about the potential causes and resolutions of interactions.

In this chapter we concentrate on describing the overall hybrid approach and preliminary results from an off-line analysis. The latter involves modelling switching systems in such a way that we can reason about their *observable* behaviour and postulate and test theories of interactions: detection *and* resolution. Here, we give an overview of how we model the system components, both at an informal and formal level, and how we employ mathematical reasoning techniques to analyse behaviour.

In the next section we give a very brief overview of telecommunications services and feature interactions. In section 3 we discuss the special case of legacy/proprietary systems and in section 4 and we present our hybrid solution. Section 5 contains an overview of the feature manager and is followed by further details of specific aspects of our formal model. Section 7 briefly describes the model-checker SPIN and a temporal logic and in section 8 we discuss how we have employed them for automated reasoning about our legacy system model. The final section contains our conclusions and directions for further work.

2 Background: Telecommunications Services

In modern telecommunications systems, control of the progress of calls and connections is provided by software at an exchange, this is referred to as a *stored program control* exchange. This software must respond (or react) to events such as lifting a handset or entering digits, as well as sending control signals to handsets and lines such as ringing tone or line engaged.

A *service* is the collection of functionality provided by a network operator and is usually self-sustaining; the basic service is known as *POTS*.

A *feature* is additional functionality; examples of features are a *call forwarding capability* and *ring back when free*. Services offer a variety of features which are said to *interact* when one affects the functionality of another(s). When interactions are not benign, both the expectations of users and the quality of services may be compromised. The feature interaction problem is diverse and complex, and has been recognised by both the industrial and academic communities as urgent – as evidenced by the International Feature Interaction in Telecommunications workshop which was established in 1992 and has since been held at

[1]This is a joint research project with the Department of Electrical and Electronic Engineering at the University of Strathclyde.

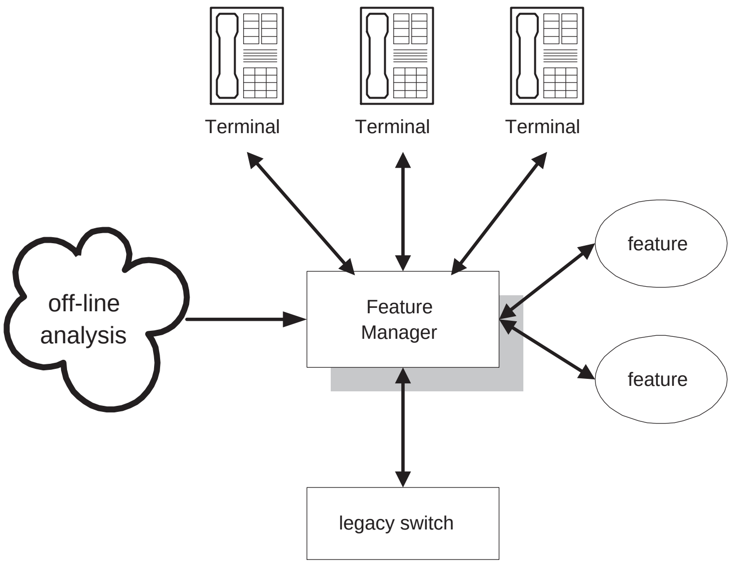

Figure 1: An On-line Feature Manager – System Architecture

regular intervals [FIW92, FIW94, FIW95, FIW97, FIW98].

Feature interaction detection and resolution techniques are broadly characterised as *off-line*, at design stage or service conception, or *on-line*, as services are executing. While off-line interaction analysis techniques are perhaps most aesthetically appealing, they may not always be possible. Specifically, they are predicated upon the existence of specifications of features – at the very least descriptions of functional behaviour and/or requirements. Moreover, they suffer from a combinatorial explosion of cases to consider as the number of features involved grows.

3 Feature Interaction and Legacy Services

Although one would normally assume a satisfactory resolution of feature interactions within a given legacy and/or proprietary system, the issue arises again when that system is required to work with further, additional, features and/or services, or simply to interwork with another system e.g. a PBX (private branch exchange) and a PSTN (public switching telecommunications network).

In both cases, a purely off-line approach is not going to be feasible, an on-line approach is required which does not depend on knowledge of the internal behaviour of the system components.

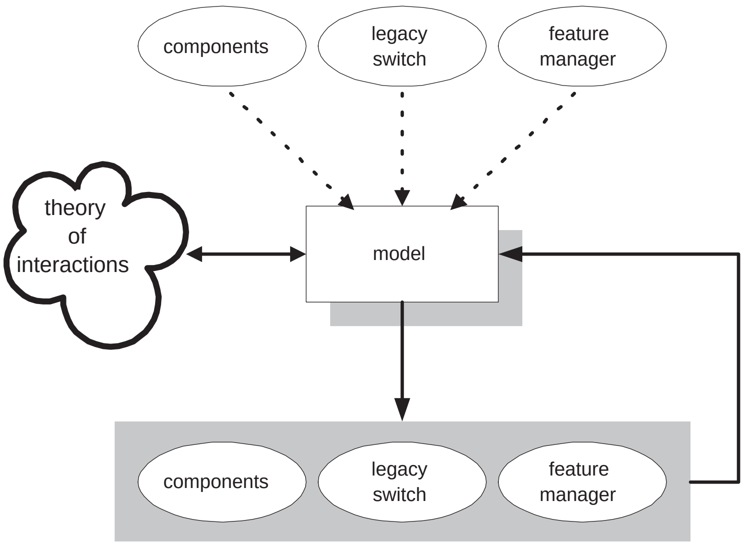

Figure 2: Role of Formal Modelling

4 A Hybrid Approach

The role of fur on-line feature manager is to mediate between new features, the legacy system, and the other components of the system (e.g. terminal devices). Figure 1 illustrates the role of the feature manager; for simplicity, we label all other system components as ‘terminal’ , for terminal devices, etc.

The feature manager must both detect potential interactions, between new features and the legacy system, and between new features themselves, *and* resolve them in a satisfactory way. Sometimes, the only resolution is to surpress one feature, then the question is which one; other times, it may be possible to interleave features or to run them sequentially, in a particular order. In all cases, the feature manager must be able to make that decision.

We have followed the the *transactional* approach proposed by Magill and Marples in [11] where the legacy switching system is treated as black-box, i.e. it receives inputs and responds with outputs, embedded in a transactional “cocoon”. The cocoon permits a rollback and commit facility, allowing us to experiment with sequences of possible inputs and responses and thereby choose the best possible resolution to a feature interaction (assuming one has been detected). The precise details of the transactional approach are not relevant here and they are described in more detail in section 5 and in [2]. What is relevant is to note that the information required in order to choose a best resolution is derived from an off-line analysis of feature behaviour. Crucial to this analysis is a formal model of the system.

The role of the formal model is more than just an abstraction, or specification, of intended and/or required behaviour; it is actually part of the incremental process of developing those intentions/requirements. That is, it is an part of a *adaptive*, experimental process, as we have advocated in [1]. Figure 2 illustrates this process.

The initial step in the process is to develop an (initial) formal model of the on-line system, i.e. the legacy switch, a relatively "uninformed" feature manager, and the system components. Dotted lines denote this step. The (initial) model provides us with a platform, or test-bed, in which to experiment with and reason about observable behaviour of the legacy system and the new features. Properties of this system will enable us to postulate theories of features and system events such as "feature x should have precedence over feature y", "event x should never be followed by event y, otherwise deadlock will occur", "event x and event y cannot be offered to a user simultaneously (e.g. a spoken announcement and a busy tone)".

The next step is to use these theories to guide the feature manager, which will consequently alter system behaviour. We can then derive further properties of the altered system, postulate further theories and further enhance the feature manager, observe new system behaviour, and so on.

We are still in the preliminary stages of this project, and therefore in the initial modelling step. The remainder of this paper outlines progress so far in this regard.

5 Overview of the Feature Manager

Overview of the Model

There are three principal categories of components in our system: the users, the operator and the network (Figure 3). Components communicate by sending and receiving messages along the connecting channels.

The term 'user' is used to describe points of interaction between the network and the outside world, such as terminal devices, trunks (connections to other networks) or any other communication equipment.

The (network) operator maintains subscriber and call records, although these are important to correct functioning and billing within the system, we do not dwell on these aspects in more detail here. Hence, all the associated connections are not drawn in Figure 3.

The network consists of the database components, the switching hardware process and a number of processes, instantiated during the lifetime of the system. Figure 3 illustrates a configuration for n calls, i.e. n feature manager processes, each with a legacy software process and m feature processes.

The switching hardware delivers messages between the users and the feature managers (and combinations thereof). The legacy software processes represent the existing software, which is to be enhanced by the feature processes. Note that the number of features connected to each feature manager is arbitrary (and can be zero).

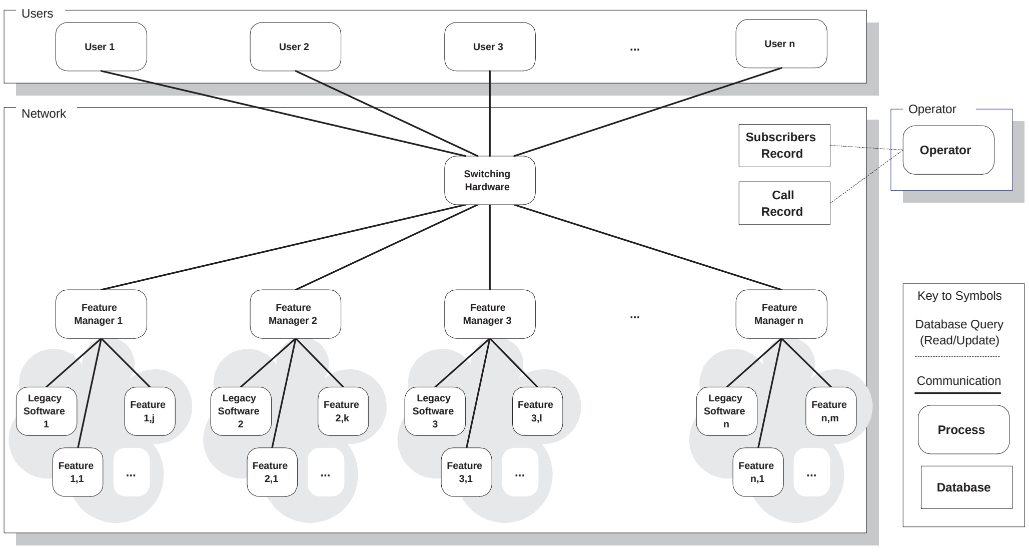

Figure 3: Overview of the System

The feature managers are central to our system, as they provide the ability to organise the co-working of the legacy software and the new features.

The Transactional Nature of the Feature Manager

The feature manager passes incoming messages from the switching hardware to the legacy software and all 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 [2].

The responses are collected by the feature manager, added to a list of responses, and once all responses are collected they are evaluated. There are three possible outcomes:

1. the feature manager did not receive any proper message,
2. the feature manager received exactly one proper message,

[2]Any message apart from the *transaction finished* message is a proper message.

3. the feature manager received multiple proper messages.

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 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; this choice will ultimately be guided by the underlying theory of features and events.

When the path is selected, it is committed by sending out the appropriate messages to the switching hardware.

The approach depends on certain assumptions about the system, namely that we can create copies of all the associated processes. As this might not always be possible, particularly 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 [2].

6 Modelling a Basic Legacy Switching System

A Simplified System

The system described in Section 5 turns out to be very complex, particularly since certain details of the feature manager will be refined during the modelling process. Therefore, here we concentrate on modelling the simplified system, i.e. the initial modelling step. However, we try to construct the (simplified) model in a way that will allow straightforward extension.

The system under consideration consists of users, the switching hardware and legacy software. The relation to the earlier proposed system can be seen easily: the number of features is zero. The feature manager is excluded, since in this simplified system its only purpose would be to pass messages from the legacy software to the switching hardware and vice versa.

Each process can be modelled by a non-deterministic finite state machine or automata (FSM/FSA); we can compose finite state machines, allowing communication via their respective input and output messages.

Switching Hardware Process

The switching hardware is the main control process, i.e. a reactive process which scans input channels for incoming messages and transmits appropriate outgoing messages. Our model incorporates this scanning as sequential polling. Apart from being realistic, this also provides a fair mechanism. Therefore, the general behaviour of the switching hardware process can be described as:

```
loop
  for each input channel
    scan for message
    if message exists
      read message
      process message
    fi
  end for
end loop
```

Processing a message basically consists of identifying the source and destination and sending it on the right output channel. There are three combinations of message source/destination to be considered:

1. a message from a user to the associated legacy software process,
2. a message from a legacy software process to the associated user,
3. a message from one legacy software process to another legacy software process.

In each case, the processing is quite straightforward.

Legacy Software

The legacy software is *POTS* (Plain Old Telephone System) basic call software. The POTS behaviour is described by the FSM in Figure 4 with input messages in italics; all other messages are output messages. A message consists of two parts: an event and an argument. The argument is a user identifier or 0 when no argument is required. Some arrows are labelled with two messages separated by a semicolon, we shall regard this as a sequential composition of messages.

State names are only given to increase the legibility: states do not contain any information; the behaviour is described by the input and output messages only.

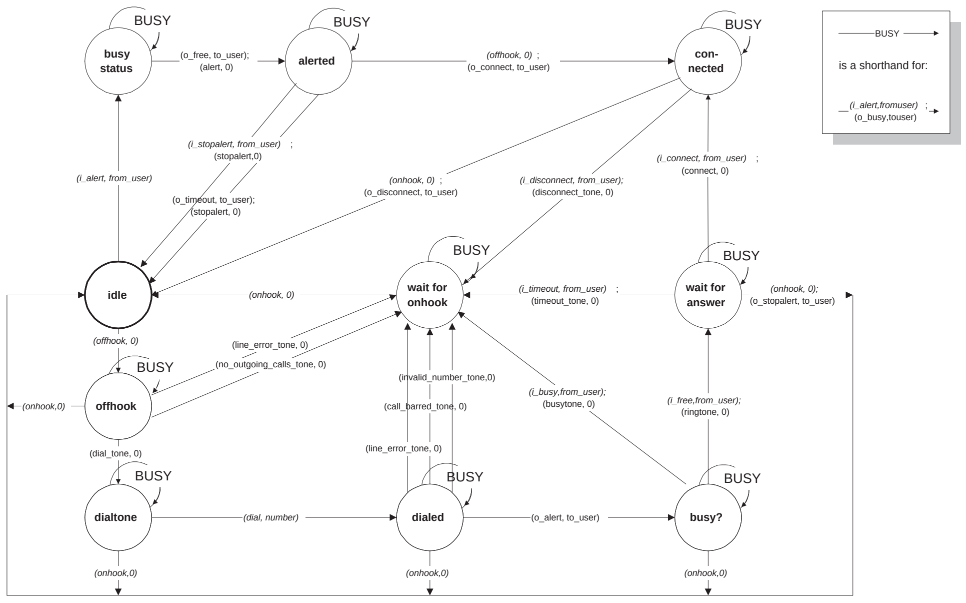
BUSY
busy status
(o_free, to_user); (alert, 0)
alerted
(offhook, 0) ; (o_connect, to_user)
con- nected
BUSY is a shorthand for:
(i_alert,fromuser) ; (o_busy,touser)
(i_stopalert, from_user) ; (stopalert,0)
(o_timeout, to_user); (stopalert, 0)
(i_alert, from_user)
(onhook, 0) ; (o_disconnect, to_user)
(i_disconnect, from_user); (disconnect_tone, 0)
(i_connect, from_user) ; (connect, 0)
idle
(onhook, 0)
wait for onhook
(i_timeout, from_user) ; (timeout_tone, 0)
wait for answer
(onhook, 0); (o_stopalert, to_user)
(offhook, 0)
(line_error_tone, 0)
(no_outgoing_calls_tone, 0)
(invalid_number_tone,0)
(call_barred_tone, 0)
(line_error_tone, 0)
(i_busy,from_user); (busytone, 0)
(i_free,from_user); (ringtone, 0)
(onhook,0)
offhook
(dial_tone, 0)
dialtone
(dial, number)
dialed
(o_alert, to_user)
busy?
(onhook,0)
(onhook,0)
(onhook,0)

Figure 4: The Basic Call Software Process

To illustrate the behaviour, consider an example of call set-up and clear-down for a call from user 0 to user 4. Assume that the basic call software is in the `idle` state and user 4 is able to engage in the call.

Example 1 *The basic call software receives an (offhook, 0) message to which it responds with (dial_tone, 0). Next, a (dial, 4) messages triggers the sending of (o_alert, 4). The fact that User 4 is free is signalled by receiving (i_free, 4). Assume user 4 goes offhook which leads to an (i_connect, 4) message being received, triggering (connect, 0) being sent. Now the two users are connected. The basic call software receives an (onhook, 0) message indicating that the connection is to be cleared down, triggering an (o_disconnect_request, 4) message being sent. At this point the basic call software is in its initial* `idle` *state and a new call can be made.*

Note that in the current BT (British Telecom) network, only the originator of a call can clear down the call (with exceptions like 999). However, many PBXs show symmetric behaviour, as modelled here.

When the initial model has been developed, and indeed during the process of development, the next step is to reason about (observable) behaviour. A number of formalisms and tools could be suitable for this purpose, but given the emphasis on communication between FSM's in our approach, we have chosen a formalism based on communicating FSM's.

7 Automated Reasoning

Promela and SPIN

Promela (*PROcess METa Language*) is the input language for the model checker Spin [9] which we use for formal, automated reasoning.

In Promela, processes run concurrently and communicate via channels, i.e. they can receive messages from and send messages to other processes. Communication may be synchronous or asynchronous. A process denotes a set of execution sequences, defined by a (Büchi) automaton. A (Büchi) automaton is simply an FSM/FSA which accepts infinite words, ideal as wen are dealing with non-terminating, i.e. infinite telecommunications processes. In our case, a program involves several concurrent processes, thereby denoting the interleaving product of the component process automata. Encoding our system model in Promela is relatively straightforward, since each major component is defined by an automata. The communication between the automata is similarly straightforward to define, employing both synchronous and asynchronous mechanisms, as appropriate.

Reasoning about a Promela program in Spin involves reasoning over the underlying (Büchi) automaton; hence we do *model-checking*, rather than *theorem proving*. There are a number of reasoning mechanisms in Spin, we mention the two most relevant below.

In-Line Assertions

An assertion is a Boolean expression about the program state and can be inserted anywhere in a program. Assertions are typically program invariants, or they reflect our assumptions about the program state at a particular point. They are an important part of the verification process and allow us to perform an "behavioural audit" as a call progresses. Examples of assertions are `message.event == dial`, `len(buffer) == 0`, or `basiccall@idle`. The first two examples express an expectation about the value of the a variable; the last example asserts that the named process is at label `idle`.

Linear Temporal Logic

In-line assertions allow us to reason about particular program states, as we pass through them; to reason about program states over time, we need a temporal logic. Spin supports LTL, *Linear temporal logic*, a propositional logic with temporal operators including $\Box$ (always) and $\Diamond$ (eventually). The logic is *linear* because all formulae are implicitly quantified over *all* execution sequences.

The mechanism for checking satisfaction of linear temporal logic formulae is via *never* claims – processes which describe *undesirable* behaviour. Full details of never claims and Büchi automata are given in [9]; here we give only a brief, informal overview of the mechanisms involved.

A never claim is a process which is run in lock-step with the system under investigation. We say that a never claim is *matched* when either it completes, or, and it is in an acceptance cycle – part of an infinite cycle through an acceptance state. If the never claim expresses some *undesirable* behaviour, then matching the never claim means that the undesirable behaviour is possible. On the other hand, if the never claim is *not* matched, then the undesirable behaviour is not possible. Every LTL formula can be associated with a never claim; loosely speaking, the claim embodies the negation of the formula. Therefore, to prove a property, we demonstrate that its negation *never* holds.

8 Reasoning about the Legacy System Model

Observations

Reasoning about our system includes the validation (or otherwise) of expected behaviour vs operational behaviour. The expected behaviour is formulated by abstract properties, expressed in terms of observations of the system.

The fundamental question is: *what behaviour can we observe?*

Recalling that are reasoning about black-box systems, i.e. proprietary or legacy code, we cannot (in the worst case) observe any internal behaviour. This obviously excludes information describing states, e.g. the states themselves (i.e. `idle`, `dialling`, etc.) or local variables. So, we have to consider other possibilities, namely we need to rely on the messages that are passed on the communication channels. At first glance this does not seem to be too problematic, but the consequences are far reaching: any decision-making algorithm used in the online feature manager must be able to base decisions on exactly the same sparse information.

We cannot explore this in detail here, but we note that most properties of interest consist of a condition (maybe complex) and a consequence;n the following section we consider two examples.

Example Temporal Properties

Consider the following properties (desirable for POTS):

- If user A calls user A, then user A should perceive a busy-tone.
- If there are no faults with the lines and user A is allowed to originate calls, user A calls B and user B is not busy, then eventually users A and B can be connected.

Both properties can be expressed by reference to only messages on communication channels and an abstract notion of time (at least a temporal ordering events).

Consider expressing these two properties wrt our system model:

Example 2 *If (dial,1) is sent, then eventually a (busy_tone,0) is sent.*

Example 3 *If (dial,2) is sent, and neither of (line_error_tone, 0), (busy_tone, 0), (call_barred_tone, 0), (no_outgoing_calls_tone, 0) or (i_timeout_request, 2), is sent then eventually a (connect,0) is sent.*

The second property is particularly difficult to read, as the condition must be expressed by the conjunction of the (absence of) several messages. Note that these are not *complete* descriptions, which would suffer from the *frame problem.* That is, it is not a trivial to express properties which also describe possible events which are not supposed to happen need.

For completeness we show how the two examples above can be written in LTL. Note that we expect the first property to be true *always*, whereas the second one only holds for *some* computation sequence (e.g. sometimes, the called party might not answer). Other properties have much more expansive descriptions, these two give only a flavour of the approach.

Example 4

$$\Box((dial, 1) \rightarrow \Diamond(busy\_tone, 0)) \tag{1}$$

Example 5

$$\begin{aligned} \Diamond((dial, 2) \wedge \neg(line\_error\_tone, 0) \wedge \neg(busy\_tone, 0) \\ \wedge\neg(call\_barred\_tone, 0) \wedge \neg(no\_outgoing\_calls\_tone, 0) \\ \wedge\neg(i\_timeout\_request, 2)) \rightarrow \Diamond(connect, 0)) \end{aligned} \tag{2}$$

Using Spin, we are indeed able to demonstrate that both these properties hold, for the appropriate set of computation sequences. We note that the process of proving in-line assertions and temporal properties has been a valuable part of the development process, enabling us to uncover numerous errors, inconsistencies, and misconceptions about the system, as well as uncovering important properties.

9 Conclusions and Future Work

We have described a hybrid approach to developing an on-line feature manager for mediating between legacy and new telecommunications services. The approach involves developing a transactional system behaviour and modelling the switching systems, in such a way that we can reason about their *observable* behaviour and hypothesise about and test theories of interactions. The formal model is not just a specification, but it is part of an experimental, cyclical process in which we can both *prescribe* and *explore* behaviour, and derive, evaluate and test further properties.

We have described the system components, both at an informal and formal level, and briefly discussed how we employ mathematical reasoning techniques to analyse behaviour. Some examples illustrate our approach.

Our research is still at a preliminary stage. We have concentrated on the overall system architecture, the hybrid approach, and the feasability of automated reasoning using PROMELA and SPIN. We are satisfied with the outcomes so far, though we have some reservations about the ability of SPIN to handle the full-scale formal model, some further abstraction may be necessary.

Three major inter-related areas for further research are the design of specific algorithms for the construction of the behaviour trees, the selection of the "best" path within these trees, and the incorporation of feature and event theories into the feature manager. Last, but not least, we will implement the approach in the live switching environment provided by our industrial collaborators.

References

[1] M. Calder. What Use are Formal Design and Analysis Methods to Telecommunications Services? In [7], pp. 23- 31.

[2] M. Calder, E. Magill and D. Marples. A Hybrid Approach to Software Interworking Problems: Managing Interactions between Legacy and Evolving Telecommunications Software. To appear in *IEE Proceedings–Software*, 1999.

[3] Proceedings of International Workshop on Feature Interactions in Telecommunications Systems II, St. Petersburg, U.S.A., IEEE Communications Society, 1992.

[4] W.Bouma and H.Velthuijsen (eds.). *Feature Interactions in Telecommunications Systems II.* Proceedings of International Workshop, Amsterdam, IOS Press, 1994.

[5] K.E. Cheng and T. Ohta (eds.). *Feature Interactions in Telecommunications Systems III.* Tokyo, IOS Press, 1995.

[6] P. Dini, R. Boutaba, and L. Logrippo (eds.). *Feature Interactions in Telecommunications Systems IV.* Montreal, IOS Press, 1997.

[7] K. Kimbler and W. Bouma (eds.) *Feature Interactions in Telecommunications and Software Systems V.* IOS Press, 1998.

[8] *IN Distributed Functional Plane Architecture.* Recommendation Q.1204, ITU-T, March, 1992.

[9] Gerard J. Holzmann. *The Model Checker SPIN, IEEE Transactions on Software Engineering,* **23**,No.5, May (1997).

[10] Intelligent Network Recommendations. Q.1200 – Q.1229, CCITT, 1997.

[11] The Use of Rollback to Prevent Incorrect Operation of Features in Intelligent Network Based Systems. In [7].

[12] M. Thomas. Modelling and Analysing User Views of Telecommunications Services In P. Dini, R. Boutaba, and L. Logrippo (eds.). *Feature Interactions in Telecommunications Systems IV*. Montreal, IOS Press, 1997.