On integrity of telecommunications networks when controlled via the Parlay API

Master's thesis

Peter Ebben

Lucent Technologies
Bell Labs Innovations
Forward Looking Work EMEA

Supervisor: *dr. ir. Frans Panken*

Katholieke Universiteit Nijmegen
Informatics for Technical Applications

Supervisor: *prof. dr. Frits Vaandrager*
Consultant: *dr. Jozef Hooman*

Thesis number: *483*
Date: *August 1, 2001*

Lucent Technologies
Bell Labs Innovations

Katholieke *Universiteit* Nijmegen

Title:	On integrity of telecommunications networks when controlled via the Parlay API
Author:	Peter Ebben
Document type:	Master's thesis
Document number:	483
Date:	August 1, 2001
Status:	Final
Supervisors:	dr. ir. Frans Panken (Lucent Technologies) prof. dr. Frits Vaandrager (Katholieke Universiteit Nijmegen)
Consultant:	dr. Jozef Hooman (Katholieke Universiteit Nijmegen)
Keywords:	Telecommunications networks, Network integrity, Parlay API, Formal verification, SPIN, Promela, Intelligent Networks, Telephony, Model checking

The research reported upon in this thesis has been carried out under the auspices of the Informatics for Technical Applications (ITA) group of the University of Nijmegen (Katholieke Universiteit Nijmegen), the Netherlands and the Bell Labs Forward Looking Work EMEA department of Lucent Technologies, Bell Labs Innovations, the Netherlands.

Lucent Technologies
Bell Labs Forward Looking Work EMEA secretariat
Botterstraat 45
1270 AA Huizen
The Netherlands

On integrity of telecommunications networks when controlled via the Parlay API

by

Petrus Wilhelmus Gerardus Ebben

A thesis
presented to the University of Nijmegen
in fulfilment of the
thesis requirements for the degree of
Master of Science
in
Computer Science

Nijmegen, the Netherlands, 2001

Preface

Although writing a thesis is a solitary affair, this thesis would not have existed in its current form without the help of many people that either directly or indirectly contributed to it. I would like to take this opportunity to thank everyone for his or her help.

First of all, I want to thank my supervisors, dr. ir. Frans Panken (Lucent Technologies) and prof. dr. Frits Vaandrager (University of Nijmegen), for their excellent support along the road and for their valuable comments on draft versions of this manuscript. Frits made me aware of the existence of ResearchIndex[1] (formerly known as CiteSeer), an enormous citation index and document database on the area of Computer Science freely available on the Internet. This database has been extensively used to find relevant literature on the subject of integrity. Frits also proposed to use SPIN for our model checking purposes, since this seemed to be the best tool for the job. Frans is an expert on the area of the Parlay API. He gave me a lot of valuable hints while studying the API, and came with more than enough suggestions to keep me occupied all the time. Dr. Jozef Hooman, who acted as consultant, also assisted in finding a suitable model checker, for which I would like to thank him.

I would also like to thank Ognjen Prnjat (University College London) for making postscript versions of some of his papers available via the World Wide Web upon my request. Special thanks goes to Keith Ward, visiting professor of telecommunications business at University College London, for sending me a copy of an article of his hand and for including another paper on the same topic as well.

I am greatly indebted to dr. ir. Theo Ruys of Twente University for all his help with SPIN. He helped me to resolve several issues regarding the usage of SPIN, and although I have never met him (yet), he kept me motivated to look into the problems with SPIN in more detail. He was very enthusiastic in his responses to my questions. I also want to thank him for sending me a copy of his Ph. D. thesis. His dissertation proved to be very useful in reducing the complexity of our model. I would also like to thank Gerard Holzmann, the creator of SPIN, for looking into my bug report. I thank professor Ed Brinksma (University of Nijmegen / University of Twente) for putting me in touch with Theo Ruys.

The research that is reported upon in this thesis has been conducted within the Forward Looking Work department of Lucent Technologies, located in Huizen, the Netherlands. I am very grateful to Lucent for allowing me the use of their computer and communications facilities. The financial support by Lucent was also greatly appreciated. I would like to thank all employees of the Forward Looking Work department for the nice time I had. Special thanks goes to Harold Batteram and Bram van Driel, with whom I shared an office.

Since the distance between my home in Haalderen and Lucent Technologies in Huizen was too large to travel within reasonable time and finding a room for a period of less than six months in the vicinity of Huizen turned out to be either very problematic or very expensive, I am greatly indebted to Rob Arntz, who has been kind enough to let me stay in his apartment in Amersfoort for the duration of my research activities with Lucent Technologies. This saved me up to four hours of travel time by public transport every single day. And that only holds on days that the *Nederlandse Spoorwegen* (the Dutch railroad company) does not have to suffer from delays, which became a rare occasion since the beginning of June.

[1] See http://citeseer.nj.nec.com.

I wish to thank Rob Arntz, Stefan Ermens and Micha Streppel for their interest in the progress of my research. Bianca de Vree, although very busy with writing a thesis herself, found some time to provide some moral support via e-mail, which was greatly appreciated. Bianca also commented on a draft version of this manuscript, for which I would like to thank her.

After a week of hard work at Lucent, it was time for the necessary relaxation. I wish to thank Rob Arntz, Helga Bosch, Jolanda Knuman, Chantal van Kuil, Bas Maas, Hester de la Parra and Micha Streppel for the nice and pleasant *Pieterpad*-walks we had during the weekends. They effectively succeeded in getting my mind of the research for a while, which, I know, has not always been easy. The Pieterpad is a 488 kilometre long trajectory from Pieterburen in the north of the Netherlands to the Sint Pietersberg near Maastricht in the south of the Netherlands that we have planned (for some unknown reason) to cover by foot this year.

Last, but certainly not least, I wish to thank my parents and other relatives for their support.

Peter Ebben
August 1, 2001

Table of contents

List of figures

List of tables

Chapter 1

Introduction

This chapter describes important background information about telecommunications networks. We start with a brief description of the history of the telephone network, since it plays such an important role in fulfilling today's telecommunications needs. Second, we review the situation in which most telecommunications networks find themselves today and pay special attention to some of the 'problems' that exist in these networks. Finally, a solution for these problems is presented in terms of API-driven control over the network. One of these APIs, the Parlay API, will be the subject of further study in this Master's thesis. We will look into the question whether the Parlay API poses threats to the integrity of the telecommunications network it controls.

1.1 The early days of telecommunications

This section describes the evolution of (switched) telephone and telecommunications networks over the years. We start with the invention of the telephone, and subsequently discuss switching and signalling, two important parts in modern day telecommunications networks.

1.1.1 Invention of the telephone

The development of the telephone and telephone network goes back to the second half of the nineteenth century. In the early 1870s, two American inventors, Elisha Gray and Alexander Graham Bell, independent of each other, designed devices that could transmit speech by means of an electrical current.

In the summer of 1874, both Gray and Bell had conceived membrane receivers, but they both had trouble finding a suitable transmitter at first. Following earlier experiments, Bell postulated that, if two membrane receivers were connected electronically, a sound wave that caused one membrane to vibrate would induce a voltage in the electromagnetic coil that would, in turn, cause the other membrane to vibrate. Such a device would thus be able to transmit human speech. Working with his assistant, the young machinist Thomas Augustus Watson, Alexander Graham Bell had two such instruments constructed in June 1875. The device was put to the test on June 3, 1875 and although no intelligible words were transmitted, sounds resembling speech were heard at the receiving end.

On February 14, 1876, Alexander Graham Bell filed an application for an U.S. patent on his work. Mere hours later that same day, Elisha Gray filed a caveat[2] on the concept of a telephone transmitter and receiver. Since Bell's patent application was filed earlier, it was allowed over Gray's caveat and thus, on March 7, 1876, Alexander Graham Bell was awarded U.S. patent 174,465. Bell's patent is often referred to as the most valuable ever issued by the U.S. Patent Office, as it described not only the telephone instrument but also the concept of a telephone system [EncBr]. Note that at that time, neither Bell nor Gray had successfully constructed devices that were able to transmit human speech in an intelligible way.

The first transmission of speech took place between Bell and Watson on March 10, 1876, using a transmitter similar to one constructed by Gray: two conductive rods immersed in an acidic solution. Work on improving the transmitter continued over the next months. Six months after the first transmission of speech, on October 9, 1876, a two-way test of the Bell telephone system was conducted over a 5-kilometre distance between

[2] A caveat is a formal declaration by an inventor to the U.S. Patent office of an intent to file a patent on an idea yet to be perfected; it is intended to prevent an idea from being used by other inventors.

Boston and Cambridgeport, Massachusetts. The first commercial application of the telephone took place in May 1877. Since then, continuous improvements have been made to the telephone, which eventually resulted in the telephones that are in use today.

1.1.2 Switching

The performance of the early telephone transmitters was poor compared to today's instruments. However, there was an enormous demand for Bell's invention [Tane96]. The initial market was for the sale of telephones. It was up to the customers to string a wire between two telephones, so they could communicate with each other. If a telephone owner wanted to talk to *n* other telephone owners, *n* separate wires had to be strung. Within a year, the cities were covered with wires passing over houses and trees. Alexander Graham Bell recognized that connecting every telephone to every other telephone was not going to work, due to the large number of connections involved[3], and formed the Bell Telephone Company, which opened its first switching office in 1878. The Bell Telephone Company ran a wire from this switching office (located in New Haven, Connecticut) to the homes or offices of their customers. The switching office, which was operated by hand, permitted up to 21 customers to reach one another, but was soon upgraded to handle hundreds of lines. To make a telephone call, the telephone owner would crank the telephone, herewith inducing a voltage that rang a bell in the switching office, attracting the operator's attention. The operator would then connect the two lines (which were terminated in the switchboard) of the caller and the callee to each other by manually inserting a jumper cable between the two terminating sockets in the switchboard. More manually operated switching offices soon followed. Pretty soon, people wanted to make long-distance calls between cities, so the Bell system began to interconnect the switches. The original problem, the wild jumble of cabling, soon returned. As a solution to this problem, second-level switching offices were invented to interconnect the switches. As such, a switching hierarchy arose. In the United States, this hierarchy eventually grew to five levels [Tane96].

The invention of automatic switching equipment did not take long, since the 19th century undertaker Almon B. Strowger invented the first automatic switch as early as 1879. History records his motivation to invent automatic switching equipment as follows. Shortly after someone died, one of the relatives would call the town operator and ask to be connected through to an undertaker. Unfortunately for Mr. Strowger, there were two undertakers in his town and it happens to be that the other one's wife was the town telephone operator…

The Strowger switch, or its more common name, step-by-step switch, had a capacity of 100 terminals, arranged 10 rows high and 10 columns wide. This electromechanical switch contained a movable 'brush' that could be brought to the position of any of the 100 terminals. Pulses generated by the telephone instrument operated the brush, one step at a time. A telephone owner could thus select the telephone of the callee by generating pulses with his telephone. No operator invention was needed. Later, the rotary dial replaced the special buttons on the telephone that were used to generate the necessary pulses.

By the 1890's all three major components of a telephone system were in place: the local loops (i.e., the copper wires that are used to connect one's telephone to the nearby switching office), switches and the interconnection trunks that provided the connections between switches. However, the telephone system still continued to evolve, with the most notable inventions in the switching systems.

In 1913, J.N. Reynolds, an engineer with the manufacturing division of the American Telephone and Telegraph Company (AT&T) patented a new type of electromechanical switch, which became known as the crossbar switch. It could serve up to 10 simultaneous voice connections. Six years after the invention, Televerket, the Swedish government-owned telephone company, demonstrated the first crossbar system. The first commercially successful crossbar system was AT&T's No. 1 crossbar system, deployed in Brooklyn, New York, in 1938. This crossbar system was continuously improved. The No. 5 crossbar switch, which was first deployed in 1948, was the most notable improvement of the No. 1 crossbar switch, and became the workhorse of the Bell telephone system [EncBr]. In the end, the No. 5 crossbar switch was able to handle 35,000 voice circuits.

Since telephone traffic continued to grow through the years, the need for more and larger switches arose. There were also plans for providing new services via the telephone network, thus creating a demand for

[3] To create a fully interconnected network with *n* points (i.e., a network in which every point is connected to every other point via a direct connection) a total number of $n (n – 1) / 2$ wires are needed. For n = 1,000 this amounts to 499,500 wires, and for n = 5,000,000 this amounts to no less than 12,499,997,500,000 wires.

innovative switching design. Due to the advent of the electronic transistor in 1947 and subsequent advances in other electronic devices (e.g. memory devices), it became possible to design a telephone switch that was fundamentally based on electronic components, instead of electromechanical technology.

In 1965, AT&T placed the No. 1 ESS (Electronic Switching System) in service. This switch contained both a read-only and a random access memory device, permitting hundreds of new features to be handled by the switching equipment. In essence, this switch was a computer with a large amount of highly specialised I/O equipment. The No. 1 ESS served 65,000 two-way voice circuits. In 1976, the AT&T No. 4 ESS was placed into service. This switch was capable of handling 53,760 two-way *trunk* circuits (it was thus meant to interconnect switches). The No. 5 ESS, providing an even larger capacity of up to 100,000 lines soon followed the No. 4 ESS. This switch is installed in today's telephony network in the Netherlands and other countries.

1.1.3 Signalling

Closely related with the ongoing development of switching equipment was the development of signalling systems. Signalling is a major component of any telephone system. In the signalling system, electric pulses or audible tones are used for alerting (requesting service), addressing, supervision (e.g., monitoring idle lines) and information provision (e.g., providing a busy signal). At first, all signalling that took place was in-band signalling, meaning that control signals that activate and deactivate (switch) internal control functions lie within the band accessible to the user. An example of in-band signalling (on analogue telephone lines) are the *Dual Tone Multi Frequency* (DTMF) tones that are transmitted by the telephone instrument and gathered by the switch to determine the destination address of the call. As telephone traffic continued to grow and demand for new services arose, signalling among switches evolved from in-band signalling to out-of-band signalling, in which a separate (often fully digitised) channel is used for signalling purposes, employing its own routing system. The signalling system in use in North America is Signalling System 7, also known as SS7 (see e.g. [SS7]), which supports ISDN and includes features to form the foundation of a future intelligent network.

In the *Integrated Services Digital Network* (ISDN), out-of-band signalling is also used in the local loop, i.e. the connection between the telephone and the switching office. The same two copper wires are used, but by using multiplexing techniques, the users effectively gains access to two data channels (B-channels) and one signalling channel (D-channel) in the case of narrowband ISDN (N-ISDN). The ISDN network is meant to provide integration of voice and non-voice services. Example services that are possible via the ISDN network are number display, instant call setup, and medical, burglar and smoke alarms, alerting the hospital, police station and fire department, respectively.

The *Intelligent Network* (IN) is 'an architectural concept that provides for the real-time execution of network services and customer applications in a distributed environment consisting of interconnected computers and switching systems' [FGKS97]. To put it in other words, an Intelligent Network *is* a telephony network, enriched with computers that are able to perform specific call-related services. The most simple example of an IN service is perhaps the freephone service. In freephone services, a logical[4] (800) number is associated with one or more physical numbers. When a user calls such an 800 number (free of charge), the switch is presented with a problem, since it does not know how to route calls to a logical number (switches, in their purest form, can only route calls to physical (i.e., existing) numbers). To solve this problem, it launches a *query* via the signalling network to a database. This database will have to respond with a physical telephone number, so that the switch can resume call processing. Depending on the application, the database can always return the same number (e.g., if a company only has one number but wants customers to reach them free of charge) or it can be made to vary, depending on e.g. the time of day, day of week, or any other factor.

1.1.4 Modern telephony

In Figure 1, a simplified view of the Intelligent Network architecture is shown. It also summarises the most important parts of a modern *Public Switched Telephone Network* (PSTN, sometimes also referred to as POTS, *Plain Old Telephone System*). In conventional telephone networks, the network centric database and associated *Service Control Point* (SCP) are not present. It is this Intelligent Network architecture, in which highly specialised computer equipment is used in combination with high-performance switches, that allows

[4] A logical number is a virtual number that is not directly assigned to a subscriber line, but points to a real (physical) telephone number.

for the huge amount of services that is available in modern day telephone networks. Note that also without an Intelligent Network, quite a lot of features can be realised in other ways (see section 1.2 below).

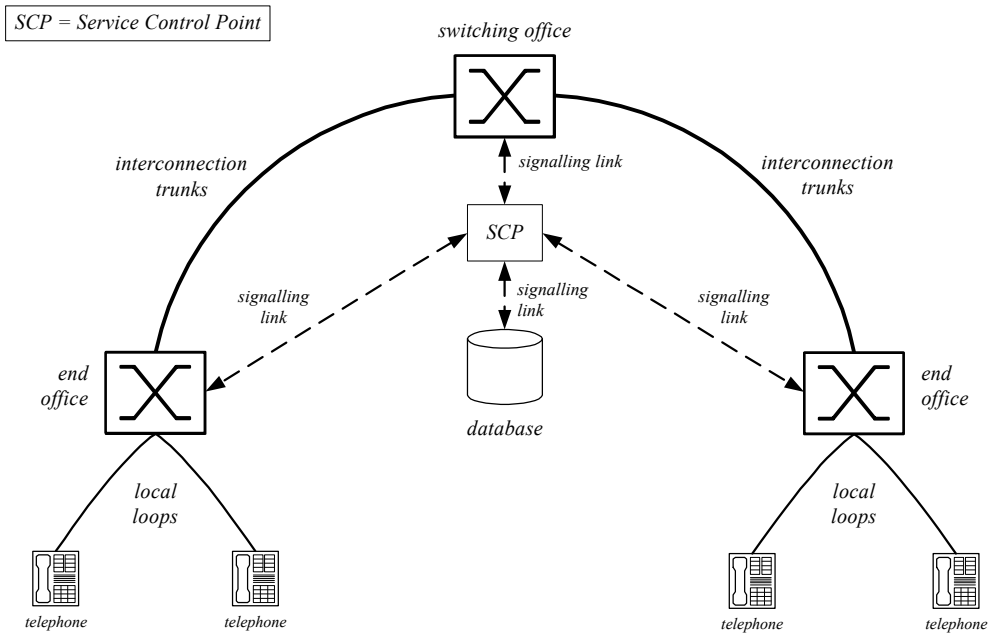

Figure 1: Intelligent Network Architecture

In modern telephony, call setup works as follows. If a subscriber goes off hook, the nearby end office, which has been continuously monitoring the line, provides a dial tone to the subscriber's telephone extension, indicating that the switch is ready to collect DTMF input. The user enters the telephone number, which is collected by the switch. If the whole address is collected, it is translated to a route in the telephone network. If it turns out that the same end office serves the destination address, the switch connects the two lines directly to each other. If this is not the case, the switch uses the signalling network to find out if the destination is busy. If not, a call path is set up from caller to callee (in reverse order, so the first trunk that is allocated for use with this call is the trunk between the last and the one but last switch). The end office serving the destination address subsequently plays a ringing tone, that can be heard by the caller once the complete call path is set up, until the destination party answers the call. A voice connection is then established automatically.

1.2 Two service delivery mechanisms in use today

Intelligent capabilities in conventional switched networks (e.g. PSTN) and data networks have generated opportunities to expose control of many telecommunications network capabilities to enterprises outside of the network operator's domain [Parl99]. However, this is not the current practice. Intelligent capabilities have been exclusively under the control and exploitation of the network operators, mainly because network integrity and security had to be ensured, but another reason for this can be found in incompatible standards. As a consequence, there are two ways in which services can be delivered (service delivery mechanisms) to users:

- *Network (centric)* service delivery, which means that services are realised in the network domain. These services (actually: the 'programs' implementing these services) can access information that is available in the telecommunications network (e.g., information presented by the signalling network), but generally cannot access data in the enterprise domain for critical decision-making.
- *Edge of network* service delivery, which means that services are realised outside the network domain (i.e., in the enterprise domain). Such services can, by their nature, access information in the (associated) enterprise domain, but they cannot access information and capabilities within the network, other than those that are available to any terminal.

An example of a network centric service is a voice mailbox, which is activated when the dialled number is busy or not answered after, say, five rings. Network services can access the data available in the network domain, such as, for example, the place the call originated from or the billing method currently in use. These

network centric services provide little flexibility, since they run in the network domain and are not customisable from the outside. They are well suited for mass-market applications and are usually very robust due to the large amount of testing that has been done (by the network operator) before the new service was launched.

To edge of network services, such network-specific information is generally not available. An edge of network service can, however, access data in the enterprise domain, since it runs in the enterprise domain. With such a service, a call is delivered to a terminating party at the edge of the network. This party can then decide what to do with the call.

Consider the network topology shown in Figure 2, in which a company's private telecommunications network is interconnected with the public telecommunications network via trunk *IV* (e.g., 30 telephone lines). The core of the company's network is a switch, which handles all incoming calls. Suppose that this company allows their employees to work at home and, if employees choose to do so, provides call-forwarding services for all calls that are made to the employee's number at work to their home number.

Now consider the situation that subscriber *A* (associating users with their telephone) wants to call employee *G* at a moment that he is working at home. Subscriber *A* goes off hook, and dials *G*'s telephone number. As soon as the end office, serving subscriber *A*, has collected the address information, it determines that the call has to be routed via trunk *II* and subsequently via trunk *IV* to the company. As seen from a public network operator's point-of-view, the company's switch terminates the call. In reality, this switch will continue call processing. It determines that the call has to be routed to *G*, but knows that *G* is working at home and that the call thus has to be redirected to *G*'s home number, *D*.

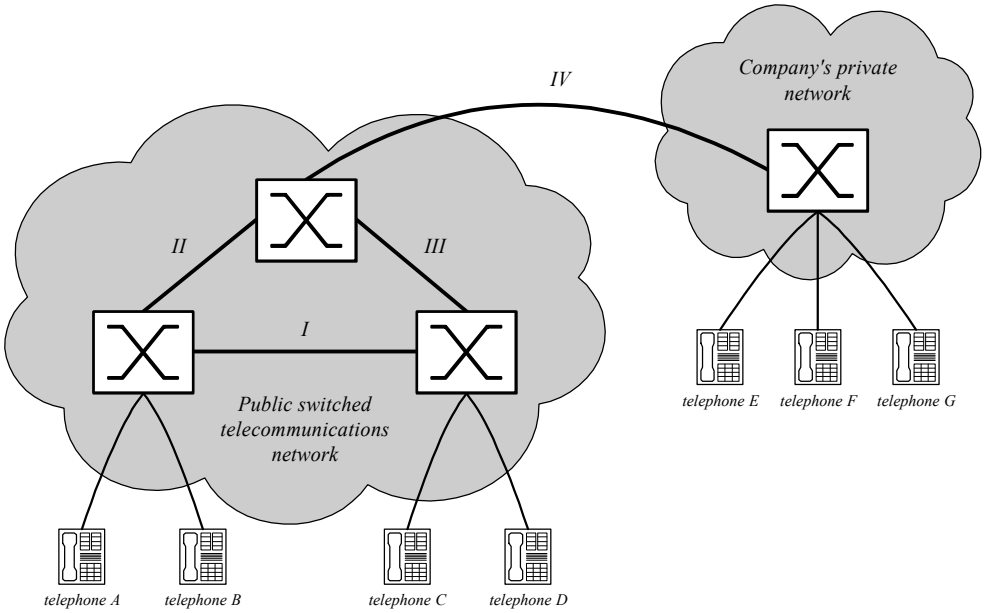

Figure 2: Example network topology with public and private telecommunications network

Since the company's switch does not have control over the public telecommunications network, it cannot 'ask' the public network to reroute the call to a different destination. Instead, all it can do is to set up a *new* call to *D*, and connect those calls through to each other. The communications path that is now in use, starts at telephone *A*, and goes via the local loop to the nearby end office. There, the call is placed on trunk *II* to the next switching office, which in turn places the call on trunk *IV* to the company[5]. The company's switch routes the call back to the public network, via trunk *IV*, and subsequently via trunk *III* and *D*'s end office to *D*'s telephone. In this way, network and edge resources are wasted and potential bottlenecks introduced. For example, the company now has two telephone lines in use for a call whose terminating party does not even lie within the company's private network. It would be more efficient to directly connect the caller to the callee, since they are both part of the public network, e.g. from *A*, via trunk *I* to *D*. This is at present not possible, since it would require means to tell the public telecommunications network that the ongoing call has

[5] Recall that the communications path is set up in the reverse order; what we show here is the path *A*'s voice has to travel to reach the person he called.

to be routed to another number. The information from the enterprise domain (i.e., that the employee is working at home) is not available in the public network operator's domain.

1.3 Merging the enterprise and network domain

The two service delivery mechanisms in use today are thus not optimised for those situations where both access to the enterprise domain and control over the public telecommunications network is needed, such as in the example above. A solution is needed in which the benefits of both types of service delivery mechanisms are combined. In other words, services must be able to access both information in the enterprise domain and information in the network domain.

To facilitate the merging of these two domains, access to network information and control of network capabilities must be made available to applications outside the network operator's domain. This can be achieved by creating an *Application Programming Interface* (API) that resides between the application layer and the service component layer[6] in the network. Such an API can provide controlled access to the network, so that information from the network can be used in critical decision-making and more efficient routing decisions can be made. There are four groups of players on the telecommunications market that can benefit from such an API: network operators, service providers, application developers and end users.

Network operators can benefit from the market growth as new service provider applications increase the demand for network traffic. New revenue can be generated by charges for access to intelligent network capabilities via the API.

Service providers are able to improve or extend existing services and can address niche market requirements by deploying applications that exploit the API. For service providers, the revenue lies in the exploitation of API-based services.

Application developers benefit from the fact that telecommunications capabilities are now more easy to integrate in applications. They are thus presented with a whole range of new business opportunities that previously were not either technically or financially feasible (because the network operator could not deliver the functionality that was needed to implement the application, or the cost of integration testing was too high to make telecommunications application development financially viable). If the API ensures integrity, the testing period (especially integrity testing) can be shortened.

The last group, end users, can benefit from the reduced time between identifying a requirement for a new communications related application and the deployment of such an application and the expected increase in the number of services that are made available.

There is however one potential problem with API-based solutions that open-up the network. It is of the utmost importance that such an API encapsulates the network capabilities that are useful to the third party service provider domain in a way that *maintains* the integrity, performance and security of the telecommunications networks. Applications that use the API must be prevented from impacting existing network hosted applications. If the API does not provide protection against these situations, the network operator will never deploy the API in his network, since by doing so he will endanger his network's integrity.

Since the network operator must be able to control which applications are using the public telecommunications network and must have control over the resources that are being used by the applications (to avoid network congestion), anyone who wants to deploy API-based services has to get a fiat from the public network operator first. Such a fiat can e.g. be in the form of a Service Level Agreement, which has to be signed by both the network operator and the service provider.

The past years, work on designing such APIs has been done, and it still continues. Sun Microsystems leads a community of companies called JAIN that is developing standard, open, published Java APIs for next-generation telecommunications systems consisting of integrated Internet Protocol (IP) or asynchronous transport mode (ATM), public switched telephone network (PSTN), and wireless networks [JAMS00].

Another API that is currently being designed is the Parlay API, developed by the Parlay Group, a consortium of companies (mainly) in the telecommunications sector. The Parlay API aims to provide ways to intimately

[6] Example service components in a telecommunications network are components for routing, billing and authentication.

link IT applications with the capabilities of the communications world, and is as such not restricted to telephony alone. The Parlay API can be used in combination with e.g., PSTN, mobile networks and IP-based networks.

1.4 Problem definition and outline

As we have outlined in the previous section, it is important that APIs that expose control of telecommunications capabilities to users outside of the network operator's domain maintain the integrity of the telecommunications network in which they are deployed. In this thesis, we are trying to find out whether this is the case with the *Parlay API*. To put it in other words, we are looking for an answer to the following question:

> *Is it possible to use (part of) the Parlay API in such a way that the integrity of the telecommunications network in which it is deployed, is harmed?*

Note that deploying the Parlay API in a telecommunications network automatically means that part of the control over that telecommunications network is made available via the Parlay API. We are thus referring to the *same* telecommunications network with the words 'the telecommunications network in which the Parlay API is deployed' and 'the telecommunications network that is controlled via the Parlay API'.

The research reported upon in this thesis is thus aimed at finding situations in which telecommunications network's integrity is harmed, as a consequence of a series of Parlay API calls. If such situations are found, we will also try to find ways to prevent these problems from occurring. We are thus also looking for an answer to the following question:

> *What modifications have to be made to the Parlay API in order to avoid integrity-related problems from occurring once it is deployed?*

The remainder of this thesis is structured as follows. Chapter 2 describes the Parlay API in more detail, and describes which parts of the Parlay API are subject of study in this thesis. Chapter 3 is dedicated to the meaning of the word 'integrity', since a clear notion of what exactly is meant by 'the integrity is harmed' is needed in order to be able to answer the questions stated above. Chapter 4 describes our efforts to formally verify the absence of integrity-related problems in a small part of the Parlay API using the model checker SPIN. Chapter 5 summarises (other) threats to integrity that have been found while studying the Parlay API. Chapter 6 concludes this thesis, by summarising the most important conclusions and by indicating areas in which further research is necessary.

Chapter 2

The Parlay API

In this chapter, the Parlay API is described briefly. At the time of writing, the latest version of the Parlay API is version 2.1, dated June 26, 2000. The complete documentation of this API encompasses 26 different documents and two sets of files, defining interfaces. Therefore, it falls out of the scope of this thesis to present the Parlay API in all detail. Instead, some context information about the API is given and we will subsequently shift focus to a small but important part of the API, namely call control.

2.1 The Parlay Group

In April 1998, the Parlay working group was formed to produce an API specification that would provide enterprises access to network information and allow them to control a range of network capabilities. This API, which is still under development, is called the Parlay API and the latest version is available to the general public via the Parlay Group's website at http://www.parlay.org.

The first version of the Parlay API was published in December 1998. At that time, the Parlay Group consisted of British Telecom (BT), Ulticom, Microsoft, Nortel Networks and Siemens. In May of 1999, six more members joined the Parlay Group: AT&T, Cegetel, Cisco Systems, Ericsson, IBM and Lucent Technologies, bringing the total number of members to eleven. In January 2000 version 2.0 of the API was published and merely six months later Parlay API version 2.1 saw the light. In November 2000 and January and March 2001, revised versions of (parts of) the 2.1 API specification were brought out by the Parlay Group. These revisions did not include major changes. In fact, they were mostly correcting typing errors in previous versions of the specification.

Since June 2000 the Parlay Group is encouraging new members to join, already with considerable effect. In February 2001, the Parlay Group's member list showed 24 full members and 13 more affiliate members. At the moment, the group (a non-profit organisation) is working on version 3.0 of the API.

The goals of the Parlay Group are perhaps best described by the group itself [Parl00c]:

> The Parlay Group aims to create an explosion in the number of communication applications by specifying and promoting open Application Programming Interfaces (APIs) that intimately link IT applications with the capabilities of the communications world.

The key focus of the Parlay Group is thus the realisation of the API. Future deliverables of the Parlay Group are aimed to assist the realisation. These deliverables need not only be documents that describe the API in an abstract form, but could also include Software Developer's Kits (SDK) and/or reference implementations of the Parlay API in specific technologies. It is however not the intention of the group to produce a (commercial) product. Whether and when specific products and applications are available, is thus completely dependent of companies or individuals willing to realise the benefits of the Parlay API. The Parlay Group also intends to launch a number of supporting programs, including the creation of training materials and providing diverse documentation about the Parlay API and the surrounding concepts for people at different levels of understanding (ranging from Chief Technical Officer and marketing people to engineers) [Parl00d, Parl01a].

2.2 Characteristics of the Parlay API

The Parlay API specification is open (i.e., freely available at the Parlay Group's website) and technology independent, meaning that it is not tailored to one specific programming language or architecture. The specification itself is defined in the technology-independent Unified Modelling Language UML [UML97]. Based on the UML specification, the Parlay Group has generated two sets of *Interface Definition Language* (IDL) files, one for Microsoft IDL and another one for CORBA IDL, describing all Parlay operations and their signatures.

Since the API specification is open and technology independent, it enables the widest possible range of players (e.g., independent software vendors) on the telecommunications market to develop and offer advanced telecommunications services. It should be possible for the applications to be built, tested and operated by enterprises outside the network domain.

The Parlay API itself abstracts from network protocols. A network operator has to implement a *Parlay Gateway* in his network, in order to enable others (and himself) to use the Parlay API. This gateway subsequently translates Parlay API calls into low-level operations that are understood by the underlying network. For application developers, this has the advantage that Parlay API-applications are portable between networks.

The Parlay Group has chosen for an object oriented design, in order to achieve these goals. The Parlay API can be deployed via distributed technologies such as COM or CORBA.

A few key characteristics of the Parlay API are mentioned below:

- The API supports services with different media types, not just telephony. For example, Parlay could be used to realise a conference call between several parties, providing both audio and video or just audio to the parties in the call, depending on the capabilities of their terminals.
- The API is supposed to be manageable, which is an essential characteristic if network operators are going to deploy the API. The Parlay API provides ways to present the client application with a *Service Level Agreement* (e.g. defining the maximum allowed resource usage) that has to be digitally signed by the client application. In this way, the network operator can determine how many of his resources he wishes to supply to the third party service provider and take corrective action if more resources are requested than allowed. Later versions of the Parlay API are also expected to provide administrative interfaces.
- The API is designed to be secure. This is an essential characteristic, since both the service providers and the network operator must be convinced that the security and integrity of their respective domains (enterprise domain and network domain) is maintained before the API will ever be adopted. Unfortunately, this area needs some work, as will be shown further on in this thesis.
- The API supports discovery. Discovery is a way for a client application to get informed about the services offered via the Parlay gateway, without knowing the addresses of the services on beforehand.

The most important characteristic of the Parlay API is perhaps that it affects calls *before* they are delivered. Upon arrival of a call, it is not directly routed to the edge of the network where an application (implementing the desired service) will further process the call, but the application is *informed* about the call attempt. The application then has control over the call and makes decisions about the future lifecycle of the call. If it decides, for example, that the call has to be forwarded to another number, it asks the *network* to reroute the call. Only then a connection is established, saving resources in the network when compared to the traditional approach, described in Chapter 1. Note that, in the case of edge of network service delivery, forwarding a call to a new destination in the public telecommunications network meant that *two* calls had to be set up in the network.

Figure 3 depicts the location of the Parlay API with respect to the public switched telephone network. The Parlay Client application (which runs outside of the network operator's domain) decides what to do with calls in the network. Inside the network, in the rectangle drawn on top of a switch, resides the Parlay gateway. The dotted arrow between the Parlay application and the Parlay gateway represents the communications path over which the Parlay API is used. Note that *no* telephone connection has to be used for this communication, but a

medium supporting DCOM or CORBA. The view in Figure 3 is simplified, in that the Parlay Gateway does not necessarily have to be implemented on top of a switch. It can also be implemented on a specialised computer that communicates with the network equipment via the signalling network.

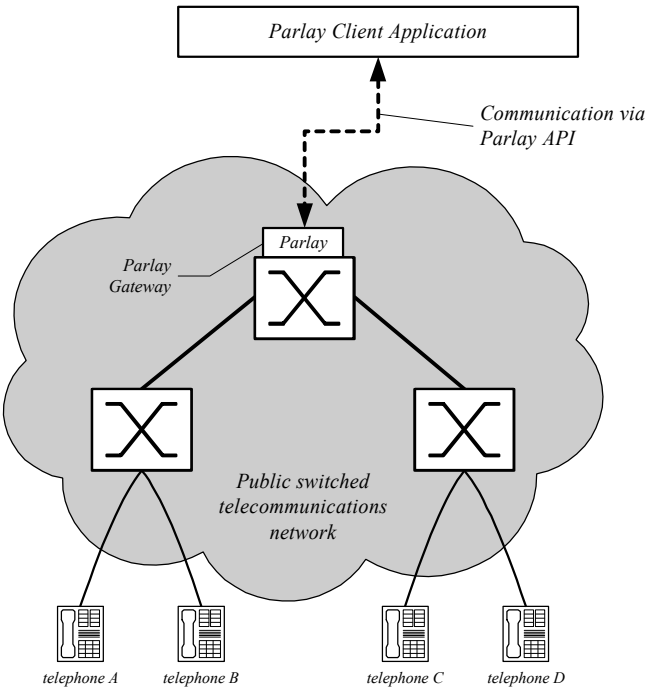

Figure 3: Call routing via the Parlay API

It is worth mentioning how calls in the network topology of Figure 3 can be affected *before* they are delivered. Consider the situation that subscriber *A* dials a freephone (800) number, and that the freephone service is implemented by the Parlay application. Subscriber *A* goes off hook and the end office collects the address information. The switch is programmed in such a way as to redirect freephone calls to the topmost switch. Call processing is now suspended, and the Parlay application is informed (by the gateway) of the call attempt. The Parlay application inspects the information that is presented to it, consisting of, among other things, the originating and destination address and subsequently decides that the call has to be routed to e.g. telephone *D*. This decision can be purely based on the destination address (in the case one freephone number always has to be routed to the same physical number) or on any other information. The Parlay API now asks the network to reroute the call to its destination *D*. Call processing is resumed, and the call is delivered to *D* in the usual way. Note that there is only *one* telephone connection in use, since there is only one terminating party.

If the Parlay application implements the 'call forwarding when working at home' feature of our previous example, no telephone connection to the company is established, since the Parlay application first inspects whether the person in question is working at home, before the call is (physically) routed to the company, saving the company up to *two* active telephone connections.

2.3 Architecture of the Parlay API

In version 2.1 of the Parlay API, three entities play an important role: the *Client Application*, the *Parlay Framework* and the *Parlay Services* (we refer to the latter two when using the name 'Parlay Gateway'). These are graphically depicted in Figure 4, which is a simplified version of the architectural overview that is given in the Parlay API documentation.

The Parlay Framework and Parlay Services expose their behaviour to the Client Application by means of the Service Interfaces and Framework Interfaces. This is also depicted in Figure 4.

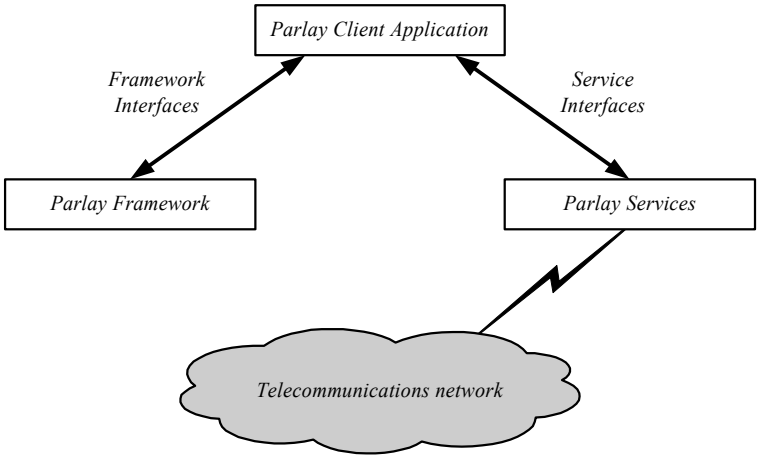

Figure 4: Parlay API architecture

The service interfaces offer applications access to a range of network capabilities and network related information, whereas the framework interfaces provide the supporting capabilities necessary for the service interfaces to be secure, located and managed. For example, to use Parlay services, the client application must first authenticate itself with the framework and then discover an appropriate service. The framework also provides fault management, load management and integrity management.

In version 2.1 of the Parlay API, the following four groups of service interfaces are distinguished:

- *Call processing*. The call processing service interfaces encompass both call control and user interaction. These interfaces allow e.g. for the routing and billing of calls (ranging from normal telephone calls to multimedia videoconference calls), as well as user interaction (playing messages to the user, gathering DTMF or voice input from the user).
- *Generic messaging*. This service interfaces allows for the handling of e-mail and voice messages. It includes operation to be notified upon the arrival of new mail and to send and receive messages.
- *Mobility*. This is a set of interfaces that is used to implement applications with a close relationship to mobility. It has the functionality to allow applications to obtain the geographical location and status of fixed, mobile and IP telephony users. It could e.g. be used to automatically receive geographical location information in case of an emergency call.
- *Connectivity manager*. This set of interfaces can be used for quality of service purposes, e.g. to set up a connection with a guaranteed bandwidth.

In this thesis, we will mainly deal with the *Call Control Service*, which is part of the call processing services. The Call Control Service is described in detail in the next section. Section 2.5 concludes this chapter with some examples of applications that can be realised using the Parlay API.

2.4 Call Control service capabilities

The Call Control services, as described in [Parl00a, Parl00b], are perhaps the most important services offered by the Parlay API, since Call Control deals with the routing, billing and management of calls. There are four different types of Call Control services[7]:

- Generic
- Multiparty
- Multimedia
- Conference

[7] The terminology used by the Parlay API designers is not consistent. They use the term ‘call processing’ to refer to both the generic call control service and user interaction service. The ‘generic call control service’ is then divided into four other services, containing, among others, the generic call control service (again). In this thesis, we use the term ‘generic call control service’ in this last meaning. The set of four services, containing the generic call control service, is termed ‘call control services’ (thus without the ‘generic’) and the combination of both call control and user interaction services is referred to as ‘call processing services’.

The *Generic Call Control Service* (GCCS) deals with 'normal' calls, that is: calls between two parties. The *Multiparty Call Control Service* allows multiparty calls to be established and managed. The *Multimedia Call Control Service* enables control over the media used (for example, it allows for monitoring of incoming video conference calls with a specified bandwidth), and thus enhances the Multiparty Call Control Service. The *Conference Call Control Service*, in turn, enhances the possibilities of the Multimedia Call Control Service by allowing the creation of subconferences in a larger conference [Parl00a]. Before we go into these call control services in more detail in subsections 2.4.2 through 2.4.6, we will first describe the call model that is used in the Parlay API.

2.4.1 Call model

The call model adopted by Parlay distinguishes the following objects [Parl00a]:

- A *call* object. A call is a relation between a number of parties. It is the application's view of a physical call in the network. Note that different applications can have different views on the same (physical) call, since they are not aware of one another.
- A *call leg* object. A call leg object represents a logical relation between a call and an address. In a regular two-party call, there are always two call legs involved: one for the calling party and another one for the called party. The Generic Call Control Service of Parlay 'hides' this to the user. In a regular two-party call one cannot have access to the call legs itself. In a multiparty call, such access is possible and even necessary.
- An *address* object. An address logically represents the party in a call (e.g. by means of a telephone number in case of a PSTN or an IP number in case of an IP-based network).

A call leg is a logical (signalling) relationship, and not a physical one. However, a bearer connection (in voice-only networks) or zero or more media channels (in multimedia networks) can be associated with a call leg. Legs can be attached to a call, or detached. If a leg is attached to a call, it means that all associated physical media are connected to the associated physical media of the other attached legs in the call. If a leg is detached, such a physical connection is not present. So, only legs that are attached have a media path.

Since the capacity of the network is limited, there is usually an upper limit on the number of legs that are being routed or the number of legs that can be simultaneously attached to the same call. It is also possible (in fact very likely) that such a maximum is defined in the Service Level Agreement a service provider has to sign with the network operator prior to gaining access to the network-side of the Parlay Services.

There are two ways in which a Parlay application can get control of a call. The first one is when an application is notified of call events that match previously specified criteria. The second way, in which a Parlay application can get control of a call, is to create a new call from the application.

2.4.2 Call Control Services overview

In Figure 5, all Parlay Call Control Interfaces and their inheritance relationships are shown in a (rotated) UML class diagram. An arrow from class *A* to class *B* means that class *A* inherits *B*'s operations. The dotted and dashed lines in this figure divide the interfaces into categories. All interfaces left of the dotted line, representing the network boundary, are located on the network side. All interfaces to the right are located on the enterprise side[8]. In other words, the interface that are classified as being in the network domain have to be implemented by the Parlay Gateway, the interfaces in the enterprise domain have to be implemented by the Client Application.

The horizontal, dashed lines divide the interfaces into the different call types: single party (generic), multiparty, multimedia and conference calls. All interfaces shown, except the two topmost interfaces, `IParlayInterface` and `IpService`, will be described briefly in the subsections 2.4.3 through 2.4.6. The `IParlayInterface` and `IpService` interfaces are general Parlay Interfaces and have been included to complete the picture. To save space, the operations provided by each of the interfaces are not shown.

[8] In this thesis, we assume that Parlay applications are run by some enterprise. Therefore, we use the term 'enterprise domain' as complementary to network or network operator's domain (the last two are synonyms). By 'enterprise domain' we thus mean a domain that is not controlled by the network operator.

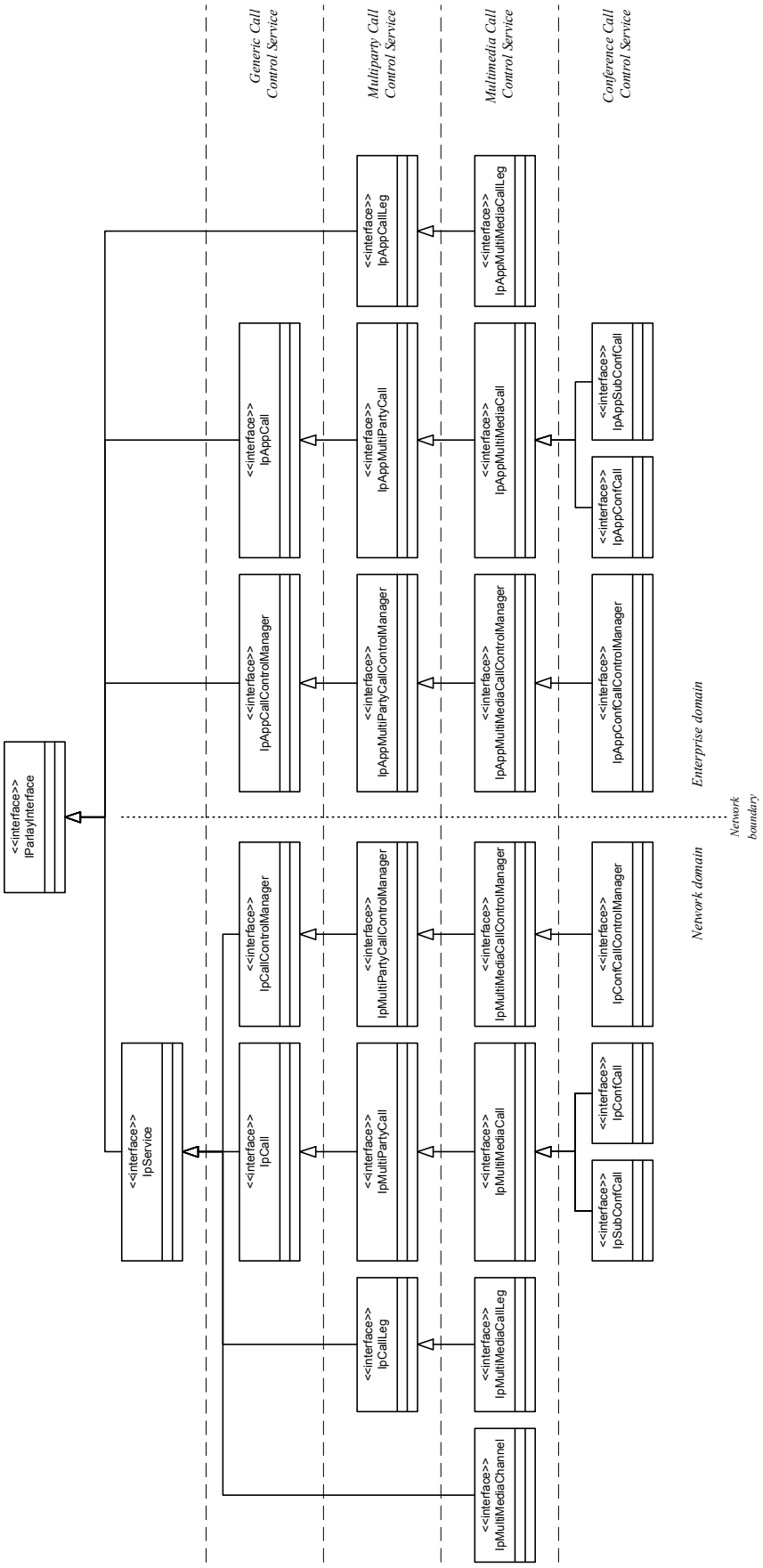

Figure 5: Call Control Interfaces and their inheritance relationships

2.4.3 Generic Call Control Service

As mentioned before, the Generic Call Control Service uses only a subset of the entire call model. Calls are limited to two parties and access to the legs is impossible. Since the Generic Call Control Service also does not deal with multimedia connections, control over the media channels is not possible.

The Generic Call Control Service (GCCS) is represented by two interfaces on the network domain side, namely `IpCallControlManager` and `IpCall`, and their counterparts on the side of the enterprise domain, `IpAppCallControlManager` and `IpAppCall`.

The operations provided by these interfaces are described below, omitting a large amount of detail. For more detail, the reader is referred to the Generic Call Control Service Interfaces specification [Parl00a].

2.4.3.1 IpCallControlManager

The `IpCallControlManager` interface provides, as the name suggests, operations to manage calls. The `createCall()` operation that can be invoked on this interface serves to create a new call object (i.e., an object implementing the `IpCall` interface). It also provides operations that can be invoked by the client application to request notifications of call events. For example, the client application can request, using the `IpCallControlManager` interface, to be notified about every call that is made to a specific telephone number or a range of telephone numbers. If, due to some error, call notification is not possible, the client application is not allowed to ask for call notifications.

Once a call notification request has been made, it can be changed or deleted via this interface. The interface also provides operations that can be invoked to impose load control on a series of calls, or to remove a previously set load restriction.

2.4.3.2 IpCall

The `IpCall` interface provides operations to route a call to a destination party and to monitor its state. For example, the client application can invoke an operation on this interface, requesting call-related information (to calculate charging, for example) to be sent when the call ends. Using the `IpCall` interface, the client application can also request to supervise a call, which means that, after a specified period of time, a status report about the call is sent to the application and control over the call is given to the application. This could be useful in pre-paid applications, to prevent users from continuing the call when the balance on their pre-paid account has reached zero.

The `IpCall` interface also provides operations to set the charging of the call. Another two operations provided by the `IpCall` interface are operations to request the user for more DTMF-input and an ‘advice of charge’ operation, which informs the user about the charging of the call (i.e., a message is sent to his terminal, if it is capable of displaying this information).

2.4.3.3 IpAppCallControlManager

The `IpAppCallControlManager` is the enterprise-sided counterpart of the `IpCallControlManager` interface. This interface provides an operation that is to be invoked upon the arrival of a call event (requested via the `IpCallControlManager` interface). It also provides an operation that allows it to receive information about the status of call notifications in the underlying network (enabled or disabled) and is informed by the Parlay Gateway (using the corresponding API call) when a call overload is encountered in the network. An operation indicating that a call was aborted in the network is also provided by this interface. It is to be called by the Parlay Gateway as soon as the network detects that a call (in which the application was interested) has terminated.

2.4.3.4 IpAppCall

The fourth and last interface involved in the Generic Call Control Service is the `IpAppCall` interface, the enterprise-sided counterpart of the `IpCall` interface. This interface provides operations that are used to handle call request responses and state reports. For example, the `IpAppCall` interface is informed about the status of routing requests: was routing successful and did the called party answer the call, or was the call

refused due to busy? The IpAppCall interface receives all status reports that correspond to the requests made via the IpCall interface. For example, if requested, the Parlay Gateway has to send call-related information to the client application, using an operation provided via this interface, upon termination of the call. Also, additional DTMF-input from the user is sent to this interface.

2.4.4 Multiparty Call Control Service

In the Multiparty Call Control Service six interfaces play an important role:

- IpMultiPartyCallControlManager
- IpAppMultiPartyCallControlManager
- IpMultiPartyCall
- IpAppMultiPartyCall
- IpCallLeg
- IpAppCallLeg

The interfaces IpMultiPartyCallControlManager, IpAppMultiPartyCall and IpAppMultiPartyCallControlManager inherit the operations from their GCCS-colleagues and introduce no new behaviour. The IpMultiPartyCall interface is an extension of IpCall. It contains operations that gain explicit access to the call legs involved. Note that in multiparty calls, a call can contain more than two call legs. The interface also provides a method to create an object implementing the IpCallLeg interface.

The IpCallLeg interface provides methods to route the call leg to the specified destination and to attach or detach the associated media to/from the call. It also provides methods for leg-specific information-requests to be sent to the application upon termination of the call leg. It does *not* provide an operation to supervise the leg continuously, although leg-specific event reports can be requested (example events are 'answer from called party' and 'disconnect'). Furthermore, several access-functions are provided via the IpCallLeg interface, for example for retrieving the ID of the call a leg belongs to. Note that a call can have multiple legs, but a leg can only be part of one call at a time.

The IpAppCallLeg interface is the enterprise-sided counterpart of the IpCallLeg interface. It receives the responses to the requests made via the IpCallLeg interface.

2.4.5 Multimedia Call Control Service

In multimedia calls, the following seven interfaces are to be considered:

- IpMultiMediaCallControlManager
- IpAppMultiMediaCallControlManager
- IpMultiMediaCall
- IpAppMultiMediaCall
- IpMultiMediaCallLeg
- IpAppMultiMediaCallLeg
- IpMultiMediaChannel

The IpMultiMediaCallControlManager interface inherits all the operations that are provided by the IpMultiPartyCallControlManager interface and extends its behaviour by providing two new operations. With these operations it is possible to enable or disable media channel notification. When media channel notification is enabled, two sets of criteria are involved. Both sets of criteria have to be fulfilled before the channel is reported to the client application. First, the multimedia call must match with the call criteria (e.g., the destination number must be in the specified range) and second, the media channel specifications must match.

The enterprise-sided counterpart of this interface is called IpAppMultiMediaCallControlManager, which inherits the operations from the IpAppMultiPartyCallControlManager interface. It introduces one new operation, namely the media channel event notification.

To represent the multimedia call on the network side, an object implementing the IpMultiMediaCall interface is used. This interface inherits the operations from the IpMultiPartyCall interface, introducing one new operation to set a granted data volume for the call (measured in milliseconds). After the granted data volume has expired, the enterprise-sided counterpart called IpAppMultiMediaCall receives a notification of this event. No further new behaviour is introduced.

The fifth interface involved is the IpMultiMediaCallLeg interface, which inherits all operations from the IpCallLeg interface. The client application can monitor and influence media channels by invoking operations on this interface. Three new operations are provided. The first one enables monitoring for the opening and closing of media channels. Two type of monitoring are possible: general monitoring (any media type will suffice), or specific monitoring (notification is sent only when a channel uses the specified media type). If the monitoring is set to interrupt mode, the application must explicitly grant access for the opening of this media channel. The IpMultiMediaCallLeg interface provides an operation to do so. The last new operation the IpMultiMediaCallLeg introduces, is another access function to obtain a list of all opened media channels associated with this call leg (note that several media channels can be associated with one call leg).

On the side of the enterprise domain, the IpAppMultiMediaCallLeg interface has to be implemented. This interface inherits the operations of its ancestor IpAppCallLeg. One new operation is introduced, namely the media channel notification, which is to be invoked by the Parlay Gateway when the opening or closing of a media channel fulfilling the requested criteria is detected.

The last interface involved in multimedia call control is the IpMultiMediaChannel interface. This interface represents a unidirectional data stream associated with a call leg. In the current design of the Parlay API, there is only one operation available, and that is the closing of the channel.

2.4.6 Conference Call Control Service

The most advanced form of call control defined by the Parlay API, is conference call control. The Conference Call Control Service enhances the Multimedia Call Control Service with the concepts of conference and subconference. A conference is comparable to a multimedia, multiparty call, with the difference that resources for conferences can be reserved in advance. A subconference is a subset of the legs involved in the conference, satisfying the property that only the legs in the same subconference can ‘communicate’ with each other (i.e., only legs that are in the same subconference have a media path with one another).

There are two ways in which a conference can be started:

- *From scratch*, i.e. without prior reservation of resources for the conference. In this case, conference setup is application-initiated and the availability of resources is not guaranteed.
- *Automatically* by the telecommunications network, as a result of a prior reservation. At the start time indicated in the reservation, the network creates the conference and the application is notified about the start of the conference, so that it can add parties to it. It is also possible for external parties to join the conference, using the address provided during resource reservation (if the conference policy allows this).

In Conference Call Control Service, six interfaces play an important role. These interfaces are described in short in the following paragraphs.

2.4.6.1 IpConfCallControlManager

The IpConfCallControlManager interface is the interface that has to be used for creating conferences and resource management. The interface inherits from its ancestor IpMultiMediaCallControlManager, but also adds significant new behaviour. This interface provides operations to create a new conference, to

check whether there are enough resources available for a conference, to reserve resources for a conference some time in the future and to free allocated resources.

Upon creation of a new conference, the application has to specify the number of subconferences to use. This must be at least one. Also, the number of participants in the conference and the (estimated) duration of the conference should be provided upon the request to create a conference. The network subsequently tries to allocate the necessary resources for the specified time. If the resources are available, the conference is started. When the duration of the conference exceeds the time for which the resources were reserved, the resources are no longer guaranteed, but the conference can continue. Note however that parties can be dropped or rejected by the network, if the resources are needed to fulfil other requests.

The `IpConfCallControlManager` interface also provides an operation to check whether enough resources are available at a specified time in the (near) future. As input for this operation, the start and stop date and time must be specified (this represents the search interval), and optionally the expected number of participants and the conference duration. The search algorithm returns the actual number of available resources, the conference duration and actual start time. Note that this operation does not yet reserve resources.

To reserve resources, another operation is provided by the `IpConfCallControlManager` interface. As parameters for this operation the start date and time, the number of participants and the conference duration have to be specified. Also, the conference policy has to be specified, containing, among others, whether parties are allowed to join ('call into') the conference after it has been created.

The last operation provided by the `IpConfCallControlManager` interface serves to free previously reserved resources. This operation effectively cancels a reservation.

2.4.6.2 IpAppConfCallControlManager

The `IpAppConfCallControlManager` interface provides an operation that is to be invoked by the Parlay Gateway as soon as a conference is created by the network as a result of an earlier reservation. This interface inherits from the `IpAppMultiMediaCallControlManager` interface, and introduces no further new behaviour, apart form the operation described above.

2.4.6.3 IpConfCall

The `IpConfCall` interface inherits from the `IpMultiMediaCall` interface. It is used to manage the subconferences, but is also able to hide these subconferences from the applications that do not need to know that there are multiple subconferences involved.

Apart from the inherited operations, this interface provides new operations to request a list of all the subconferences in the conference call, to create a new subconference and to request monitoring for 'party leaves'-events.

2.4.6.4 IpAppConfCall

The `IpAppConfCall` interface is the enterprise-sided counterpart of the `IpConfCall` interface. The `IpAppConfCall` interface receives the responses to the requests made via the `IpConfCall` interface. It inherits the operations of its ancestor, `IpAppMultiMediaCall` and provides two new operations.

First of all, the `IpAppConfCall` interface provides an operation via which it is notified on the arrival of a new party (and thus a call leg) in the conference. This can for example be used in a scheduled conference (a conference for which resources have been reserved in advance), where participants can 'dial in' to the conference using an address provided during the resource reservation (if the conference policy allows dial in).

The second new operation is meant to notify the application about the leave of a party, and should be invoked by the Parlay Gateway when a party leaves the conference and a 'monitor party leave' request has been made earlier.

2.4.6.5 IpSubConfCall

The `IpSubConfCall` interface provides operations to manage subconferences. Recall that a subconference is a subset of legs in an entire conference, so that only the legs in the same subconference can communicate with one another. It inherits from `IpMultiMediaCall`. Several new operations are provided by this interface.

One of the operations this interface provides is the splitting of a subconference. In this case, some of the legs in this subconference are moved to a newly created subconference. The inverse operation, merging of subconferences, is also provided. In this case, all the legs in this subconference are moved to an (already existing) subconference, and the 'empty' subconference is released. The interface also provides an operation to move one call leg to another subconference.

The policy for an ongoing multimedia (sub)conference can be changed via the `IpSubConfCall` interface. Some example settings that are part of the conference policy are:

- Are the video streams chair controlled or voice switched? In a chair-controlled videoconference, the chair decides which video is broadcast to the other parties. If the policy is set to voice switched, starting to talk is sufficient to have your video broadcasted to the other parties in the conference.
- Is the conference opened or closed (i.e. are parties allowed to join)?
- What sort of video mode is used? Is only the speaker's video shown, or is some sort of composite video being used?

In a chair-controlled conference, it is possible (for the chair) to inspect the video stream of one of the parties involved. It is – of course – also possible to cancel this inspection, upon which the video that currently is being broadcast is once again displayed on the chair's terminal.

The interface also provides operations to appoint a specified party as speaker and to select the chair.

2.4.6.6 IpAppSubConfCall

The `IpAppSubConfCall` interface allows the application to be informed about subconference-related events. It inherits from the `IpAppMultiMediaCall` interface and introduces two new operations. Both operations are to be called to inform the application of the requests made in the network. One of the operations is used to provide information on chair requests, the other for floor requests.

2.5 Sample Parlay applications

In this section we describe an example of a simple application that could be realised using the Parlay API. It is intended to get a better view of the possibilities of the Parlay API. Therefore, we will not include technical detail about *how* (i.e., with what Parlay API calls) the application can be realised. The application described here provides advanced services for conference calls.

Consider the situation in which five people, located on different places around the globe, have to discuss an important subject together. Since getting together in real life is not possible on such short notice, and they all own videoconferencing equipment, it is decided that the matter will be discussed in a videoconference call. One person is given the task to coordinate this. This person (call him the chair) invokes the Parlay application via a World Wide Web interface, to find a suitable time for the conference call. It is expected that the conference call will take two hours, and it has to take place within the next three days. After supplying these criteria, via the web interface, to the Parlay application, the application asks the network to find a suitable time for the conference, so that for a consecutive two-hour period enough resources are available. The result of this search is presented to the user, via the web interface. The user then decides that the time is appropriate, and thus decides to reserve the resources for the conference, indicating that the conference will be chair-controlled and that no parties are allowed to dial into the conference (the application will set up connections with the participants). After the reservation has been confirmed by the application, an e-mail or voice message is sent to all participants, informing them about the time of the conference. This can be fully automated by the Parlay application, as soon as the addresses of the participants are known. The application user thus selects the persons that are to participate in the conference call from the application's address book (which includes the e-mail addresses and voice mailbox numbers of the participants) via the web interface.

Also, to make it possible to automatically call all participants on the agreed time, their (videoconferencing) addresses have to be known by the application. This is also something that can be stored in the application's address book.

On the agreed time, the network starts the conference, and all participants are called automatically by the application. By default, their media are attached and they all end up in the same subconference, thus able to communicate with one another. The chair (which by now has been selected by the application based on information that was presented to it during reservation) welcomes all parties. Via a secure web page, the chair continuously receives updated information on the conference state, such as the number of subconferences that are involved, and which participants are placed in which subconferences. This web page also provides hyperlinks that can be used to create subconferences, to move parties between existing subconferences, and to start or stop the inspection of video streams. The conference itself can now be controlled by either this web page, or via specialised equipment (if the protocol that is used for videoconferencing supports this).

So, if the need arises for two or more people to discuss something apart from the others, the chair can create a subconference and move the two participants to it. Once they are ready, they are moved back to the subconference containing the other participants, and the now empty second subconference is removed.

If a party leaves the conference, and the chair has requested notification of this event, this information is displayed on his web page. He can now e.g. decide to add a new party to the conference, by asking the application to create a new call leg and route this to a new participant.

Once the conference is finished (i.e., all parties have left), the chair receives some status information about the conference, such as the start and end time of the conference and the parties that were involved. This information can be sent to the chair by e-mail, but the Parlay application can also present this information on the web page.

The example Parlay application described above only uses a small part of the functionality offered by the Parlay API. Yet, it suffices to illustrate the intimate linking of IT-applications with the possibilities of the telecommunications world.

A more advanced example of a Parlay API application can be found in [Parl99]. In that example, all the four groups of service interfaces (call control, messaging, mobility and connectivity manager) are combined in an application that is used to provide helpdesk and repair services to customers. If a customer calls with problems during office hours, a helpdesk agent answers the call. The customer's record is brought on screen by the application (using the calling number to identify the customer) and the helpdesk agent can help the customer. If needed, the helpdesk agent can run tests on the customer's equipment, using an IP-connection with guaranteed bandwidth, created by the Parlay API. If repairs seem necessary, the nearest mobile repair vehicle (that can be located since they are equipped with mobile devices) is sent to the customer's location.

If a customer calls outside of normal office hours, the Parlay application looks up information regarding the service contract that has been agreed upon with the customer and, dependent on this information, either plays a message that the company can only be reached during office hours, or forwards the call to a home-based expert.

Chapter 3

Network integrity

Public telecommunications networks, like the Public Switched Telephone Network (PSTN) have historically been designed with high levels of integrity in mind. Public network architects and designers have recognized the critical role that public telecommunications play in today's society and the consequences of network failure are well understood. That is why techniques such as duplication, alternate routing facilities and separate power supplies have been built into networks to enhance integrity. Integrity, in this case, is solely concerned with the availability of the network, since all techniques applied are fail-safe mechanisms to protect the network from going down in case of a failure.

But is integrity just a matter of availability? Or are there other factors involved that, together with availability, determine the integrity of a network? What exactly *is* network integrity? How should it be defined? And how can network integrity be enforced? These are important questions, especially when seen in the light of on-going developments in the telecommunications market, that are mostly aimed on opening up the network to third parties.

In this chapter, a survey on the subject of network integrity is presented. We will start by stressing the importance of network integrity in section 3.1. In section 3.2, the concept of integrity is explored in more detail. Section 3.3 describes several factors that can influence the integrity of a system. Section 3.4 describes several ways to maintain and/or enforce integrity, to some extent. This chapter is concluded with section 3.5, in which a definition of integrity is given that will be used in the remainder of this thesis.

3.1 The importance of network integrity

This section discusses the importance of network integrity by giving some examples of major network failures that have occurred in the past and the huge impact they had on day-to-day life. Also, it is described why integrity management is so important if telecommunications networks are opened up to other licensed operators, which is required by law in parts of Europe and the United States.

3.1.1 Background

Telecommunications systems are becoming increasingly complex, both in terms of hardware and software. The more complex a system, the more likely it is to fail. To prevent the network from going down in case of a hardware failure, redundant hardware can be added to the network, so the network can recover from failures and continue operation. In the late 1980's and early 1990's, it became painfully clear that these techniques do not always work as planned. A series of massive network failures occurred in the United States. John C. McDonald, among others, has described several such incidents in an invited paper for an edition of the IEEE Journal on Selected Areas in Communications, entirely devoted to the subject of network integrity [McDo94].

At May 8, 1988, the Network Control Center in Springfield, Illinois, indicated a power outage and fire at the Hinsdale central office. Attempts to call the fire department failed, since the fire had burnt through the telephone lines. The community immediately felt the impact of the fire. Telephone lines were down and emergency calls went unanswered. Air traffic controllers at the nearby O'Hare International Airport could not properly schedule flights, since they lost most of their voice and data circuits. Retailers could not verify

credit cards numbers. In total, the fire impacted 500,000 residential and business customers, who made 3.5 million telephone calls per day. Full service was not restored until a month later [McDo94].

The above is an example of a situation where the redundancy in the network could not prevent the network from going down. The fire resulted in loss of service for 35,000 residential telephones, 37,000 trunks, 13,500 special circuits, 118,000 fiber optic circuits and 50 percent of the cellular phones in Chicago. In this case, there was no single point of failure. The fire damaged too much network (and backup) equipment and telephone lines, so even if part of the redundant hardware was still working, there were too many points of failure in the network to restore full operation. Due to the costs involved, redundant hardware such as backup switches cannot always be placed on different geographical locations. It would be very expensive to e.g. connect all residential telephones to *two* end offices, instead of one, to make sure that failure in the local loop does not result in loss of service.

Telecommunications switches are essentially highly reliable software-controlled computers. As such, they are also vulnerable against bugs in software. This was illustrated by an incident in the AT&T switching system in January 1990: a single misplaced statement, inserted as part of a three line software fix propagated through the signalling network causing degradation of the operation and ending in a total shutdown. The entire eastern seaboard of the US lost telephone connections for several hours. The financial loss was estimated at 1 billion US dollar [McDo94, Hatt97].

In this case, a software error in a switch resulted in a huge financial loss. It becomes even more painful when one realises that this event was entirely avoidable, had the software been tested better. At least, in this case, testing was possible.

3.1.2 Opening up the network

Regulatory initiatives in parts of Europe and the United States require established network operators to open up their networks to other licensed operators, third party retailers and service providers. By doing so, a restricted part of control over the network is made available for use by other parties. Network operators will still have to live up to customer's expectations and thus guarantee network integrity. In other words, the network operator has to be sure that a third party retailer or service provider does not harm the integrity of his network. But the big question *how* (or even *if*) this can be realised, is not answered in a satisfactory way. We will come back to that in section 3.4.

Today's networks are (to a large extent) software controlled. This provides great flexibility, since in order to introduce new services one only has to perform a software upgrade. Due to this great flexibility, it is expected that all future telecommunications networks will be software controlled. As a consequence, software integrity is going to play an even bigger role, since a software failure automatically induces a network failure. Furthermore, since new ways of providing third-party services are emerging due to industry initiatives like JAIN and Parlay, more integrity issues arise. The reason for this is that network operators usually will have no access to the source code or the design of the application developed by the third-party service provider but still have to guarantee their network's integrity. So, the telecom operator has to be confident that some unknown application does not jeopardise his network's integrity. That is almost like granting someone access to your bankcard and PIN-code, with the request to withdraw some money for you. In this case, you just have to trust this person and hope he does not withdraw more money from your account than you asked for!

Of course, just as one is not expected to give away ones PIN-code, the network operator is not expected to grant access to his network without protecting himself against misuse. That is, he will apply some integrity control where possible, but has to do so without having access to the design or source code of the deployed application. How this can be done safely is the subject of section 3.4.

3.2 What is integrity?

In this section we will review some of the meanings of the word integrity that can be found in the literature. In section 3.5, we will present our definition of the concept integrity.

If one is interested in the meaning of a word, the first step one usually takes is to look up the word in a dictionary. According to Webster's College Dictionary, integrity means 'to have a sound or unimpaired condition' [Webs91].

Another definition is given by Scott Alexander *et al.* [AAKS98]:

> We define system integrity to mean that the system (hardware and software configuration) is not altered from some known (and presumably correct) state.

This definition is presented in the context of a secure bootstrapping mechanism, where the system's state (secure) was not supposed to be able to change (to insecure) during the bootstrapping process. It must be noted that 'state' must be seen as a rather high level notion, such as e.g. secure or insecure. When state is interpreted as a more detailed notion, the definition is rather contra-intuitive. Take for example a public switched telephone network, and assume that subscriber A has just dialled subscriber B's number. The state of the network could now be seen as 'the network is routing a call from A to B', but in that case, for the network to fulfil the integrity property, its state is not allowed to change, so a connection between A and B will never be established.

William Stallings [Stal98] has the following to say on the subject of integrity, in relation to operating systems:

> Integrity requires that computer system assets can be modified only by authorised parties. Modification includes writing, changing, changing status, deleting and creating.

McDonald [McDo94] uses the following definition of public network integrity:

> The ability of a network provider to deliver high quality, continuous service while gracefully absorbing, with little or no customer impact, failures of or intrusions into the hardware or software of network elements.

This definition, by itself, is rather strange, since it defines network integrity to be a property of the network provider instead of the network itself. So, in case the provider's network suffers from an outage, but he is able to hide this completely by e.g. routing all calls via a competitor's network, which has sufficient capacity for this purpose, the public network integrity would not be harmed, according to this definition. Indeed, when shown from a user's point-of-view, nothing seems wrong with the network (users can use it in the normal way), but from a technical point-of-view, network integrity is severely harmed, since the whole network is rendered inoperable. That in this case all calls could be routed via a competitor's network was just a happy coincidence.

Another definition, proposed by Keith Ward [Ward95] is the following:

> Integrity is the ability of the system to retain its specified attributes in terms of performance and functionality.

This definition is used extensively in the literature, especially by Ognjen Prnjat and Lionel Sacks [PSH98, Prnj99a, Prnj99b, Prnj99c, Prnj00, KPS00]. The definition applies to systems as stand alone components (e.g. switches) as well as 'integrated in the compound mesh of interconnected components which take the form of a vast distributed system' [Prnj99a]. That is, the definition also applies to telecommunication networks. Any decrease in performance or diminishing functionality (below a specified minimum) corresponds to a lower level of integrity. One minor disadvantage of this definition is that it must be known exactly what the specified attributes in terms of performance and functionality are, before it can be exactly said whether integrity is harmed. So, when we are talking about *minor* integrity issues, an average user might not be in the position to discuss whether the integrity of the nation's public telephone network is harmed, since this requires the complete specification of the performance and functionality of the nation's telephone network to be available to that user. With *major* integrity issues, that is, in those cases where it is clear beyond a doubt that the network does not deliver the services that it is supposed to deliver (for example, when it does not deliver any service whatsoever), this disadvantage does not apply. Moreover, since telephone networks are so important in today's society, it is likely for regulatory organisations to define minimum requirements on – for example – the subject of availability.

Based on Ward's definition, [MWWM97] defines the meaning of the words 'network integrity' as follows:

> Network integrity is the ability of a network to maintain a safe state in which it is invulnerable to unexpected perturbations.

Understanding the basic area covered by the concept of integrity is not a simple task, according to Prnjat *et al.* [Prnj99a], since integrity is a broad term, encompassing a variety of issues concerning system structure, functionality and behaviour. Ognjen Prnjat & Lionel Sacks mention several concepts that are closely related to integrity [Prnj99a, Prnj00]. They are listed below and are described in short in the following subsections.

- Robustness
- Resilience
- Availability
- Performance
- Scalability
- Data coherence
- Liveness
- Safety
- Feature interaction
- Complexity
- Reliability
- Security

3.2.1 Robustness

Robustness is defined as the ability of a system to handle unexpected events. For example, if a certain system is in a state in which event *X* is not meant to occur, the system is called robust if the system continues to operate in a 'normal' way in spite of the fact that event *X* occurred anyhow. Another typical example of a robust system is a system that provides protection against division by zero, or a 'paranoid' system that checks every input-parameter of a procedure, even when the caller is supposed to implement this check.

Robustness is proportional to integrity, in the sense that a robust system is more likely to retain a high level of integrity than a less robust system. A robust system can thus cope with all eventualities and continue operation.

A non-robust system is called brittle. A system is brittle if it is likely to fail when the operational environment (requested commands, timing constraints, but one could even think of e.g. the humidity level) is more narrowly defined than is likely to be true under all circumstances. A non-robust system is thus less likely to maintain a high level of integrity.

3.2.2 Resilience

A system is called resilient if it is able to recover from faults. This term is often used in the context of networks; viz. a resilient network is a network that can recover from link faults.

For example, the Internet is not resilient since it cannot recover from IP-packet loss. Terminals need to ask for a retransmission of the lost packet; packet loss can be identified via the sequence numbers in the header.

A resilient system can recover from faults and can thus continue operation. A resilient system is thus better able to obtain a high lever of integrity than a non-resilient system. Or, to put it in another way, a system that has to maintain a high level of integrity has to be resilient.

3.2.3 Availability

The availability of a system is defined as the percentage of time that the system is operational and conforms to its specification:

$$\text{availability} = \frac{\text{time the system is operational and conforms to its specification}}{\text{total observation time}}$$

In [ReVe91], this definition is referred to long-term, or steady-state availability and the term 'availability' itself is defined as the probability that the system will be operational at a given time, and is thus a predictive measure.

A high availability rate typically means that the system is likely to respond to the requests made to it in good time.

A more elaborate definition is found in [McDo94]:

> The ability of an item to be in a state to perform its required function at a given instant of time or at any instant of time within a given time interval, assuming that the external resources, if required, are available.

If a system is not available, it cannot maintain its level of integrity. A system that is required to maintain a certain level of integrity is thus required to be available, at least for the time that the integrity level has to be maintained.

3.2.4 Performance

Performance is a measure to indicate how well a given system performs. One aspect of performance is the throughput of a system. Throughput is measured in (successful) transactions per time unit, where 'transactions' is an arbitrarily chosen measure like the number of database records that can be sorted, and 'time unit' an arbitrarily chosen amount of time, e.g. a second.

Performance is often traded off against functionality, since the more a system tries to do in one transaction, the longer that transaction will take and the lower the system's throughput.

There is a clear relation between performance and integrity. A system that is to maintain a certain level of integrity requires that performance does not decrease below a specified minimum. Integrity is thus also concerned with maintaining a certain performance level.

3.2.5 Scalability

Scalability of a system is the impact on the performance of the system as a whole as more entities (hardware devices, but also processes) are added to it. The degree of scalability of a system is mainly determined by its architecture, but also by its communication, computational, data and time complexity (see subsection 3.2.10).

Although a non-scalable system can operate at a high level of integrity within certain operating conditions, it is likely that performance and integrity deteriorates when e.g. the demand for the services delivered by the system increases. If the system is scalable, this deterioration of the performance can be 'undone' by adding extra processing power, so that the old level of integrity can be reached again.

3.2.6 Data coherence

In a distributed system, where information is copied or distributed over several locations, this information has to remain consistent through time and change of circumstances. For example, if all processors in a distributed system locally maintain the state of the system as a whole, all processors involved should have the same information stored.

If differences in state information occur, the integrity of the system is at stake. This would make it e.g. possible to base decisions on information that is no longer correct and thus situations that are not meant to occur in the system can arise. A system that fails to keep the data coherent is thus more likely to fail, and therefore is a risk.

3.2.7 Liveness

Liveness means that something 'good' will, eventually, happen. If a system does not remain live (i.e., it enters a state where nothing 'good' will eventually happen), one of the two following conditions hold [Prnj99a]:

- The system is in *deadlock*. In this case, the system is blocked in a state, awaiting a message or event that never will or can occur. A real life example of deadlock would be four cars approaching a four-way intersection, each from a different direction, at the same time. Respecting the normal rules of the

road, the cars refrain from entering the intersection, since the car coming from the right is allowed to go first. A circular wait condition now occurs: each car refrains from entering the intersection until the car to his right has entered the intersection. Thus, a deadlock situation occurs, since no one can enter the intersection.

- The system is in *livelock*. In this case, the system oscillates between a (closed) set of states, which it cannot leave, and despite ongoing work, never makes progress. A human example of livelock would be two people who meet face-to-face in a narrow corridor, and each moves aside to let the other pass, but they end up swaying from one side of the corridor to the other, because they always move the same way at the same time.

If liveness is not preserved among all possible states the system can be in, chances are that the system cannot provide the functionality it is supposed to provide, since it has entered a state of deadlock or livelock, thereby reducing availability and directly impacting network integrity. A system that has to maintain a certain level of integrity is thus required to preserve liveness.

3.2.8 Safety

A system is safe if nothing 'bad' will occur [Prnj99a]. This relates to the direct results produced by any procedure, as well as the side effects that occurred while executing the procedure. To put it in other words, a system that maintains the safety property assures that the system will not end up in a state it does not belong.

If a system is in such a bad state (and thus does not satisfy the safety property), normal operation of the system cannot continue and thus the integrity of the system is at stake.

3.2.9 Feature interaction

A feature (in the context of telephone systems) is an addition of functionality to provide new behaviour to the users or the administration of the telephone system [ABR99]. Feature interaction (see also [CaVe93]) occurs when two or more systems (features), each with well-defined and understood behaviour, result in unforeseen behaviour when operated together.

An example of feature interaction, taken from [ABR99], is interaction between the features Call Forwarding Unconditional (CFU) and Terminal Call Screening (TCS). CFU is a feature that implements the unconditional forwarding of all calls that are made to a specified number, to another number. For example, if subscriber *B* enables Call Forwarding Unconditional so that all calls are transferred to subscriber *C*'s telephone number and subscriber *A* dials subscribers *B* number, *A* is transferred to *C* instead of *B*.

TCS is a feature that allows users to put telephone numbers on a black list. If the telephone number of a certain subscriber *A* is on *B*'s black list, subscriber *A* will not be able to connect to *B*'s telephone. The feature interaction occurs in the following scenario: Suppose that *B* forwards its phone to *C*. Let us also suppose that *A* is on *C*'s black list. If *A* dials *B*'s number, *A*'s call is forwarded to *C*, and a successful connection is established, since for the network the call is originating from *B*, and *B* is not on *C*'s black list.

So, when TCS and CFU are deployed together, this could result in feature interaction. In the scenario above, TCS does not work as expected due to the feature interaction with CFU, impacting network integrity (the functionality of TCS is harmed).

3.2.10 Complexity

Several notions of (structural) complexity exist:

- *Communications* complexity. This is an indication of the amount of operations or messages that have to be generated to perform some action;
- *Time* complexity. This is an indication of the amount of time (usually measured in iterations or CPU-cycles) needed for an operation to complete;
- *Computational* complexity. This is an indication of the amount of computation (and their costs) that have to be performed in order to perform some action;

- *Data* complexity. This refers to the complexity of the data structures involved and their interdependencies.

The way in which complexity relates to integrity is the following. The complexity of a system determines, to a large extent, the scalability of a system. For example, if the communications complexity of a system is very high, the system might be able to perform within the specified bounds under normal operating conditions, but a slight increase in the system load might cause the system to start malfunctioning, because the processor is waiting for the I/O operations to complete, thus missing (hard) deadlines. Another example: If a system has a high data complexity, with many interdependencies, it might not be possible to scale the system, since the data complexity requires that all data is stored on the same location (otherwise, synchronisation is needed, which can be a severe bottleneck with large amounts of data).

Coupling, a measure of the interdependence between objects or systems, is sometimes also considered as a form of complexity. Systems that are dependent on others often fail when the system they are depending on fails. In this way, an integrity breach in one component can ripple through the entire system.

There is another relation between complexity and integrity, which does not primarily have to do with the forms of complexity mentioned above, but more with 'how difficult the system is to understand'. If a system's behaviour is hard to describe, there are higher chances that errors are introduced into the system without being detected. Thus, the more complex a system is, the more likely it is that the system contains errors. After all, systems are designed by humans, and humans are not entirely fool-proof. Formal verification techniques can be used to detect these errors, but only in those situations where the run-time or memory requirements are acceptable.

It is also possible to introduce complexity in a system to increase robustness. So, complexity can also have a positive effect on overall system integrity. One could think of e.g. enhancing data security by adding a checksum to a message that is to be transmitted. Such a checksum can for example be calculated using a hash-function. The introduction of a hash-function into the system increases complexity, but is meant to increase overall system stability, since the checksum can be used to check whether the message arrived in the correct condition, before the contents of the message are used for further processing.

3.2.11 Reliability

Reliability is defined as the probability of a system performing its purpose adequately for the period of time intended under the operating conditions encountered [ReVe91]. It is important to note that system reliability is dependent on the length of time considered. For example, it might be just as hard to design a system with a reliability of 99 % during its operating life of 20 years (e.g. a spaceship) than to design a system with 99.9999 % reliability during several hours (a rocket that is used to put a satellite in place).

A system that is highly reliable is less likely to fail. A system that is required to maintain a specified level of integrity is required to be reliable over the total period of time that the system has to be operational.

3.2.12 Security

Security deals with protection of the system against attacks from untrusted users. A secure system is more likely to stay in its operational state, since it is able to detect and/or avoid any intentional attack. Security can be enforced by means of encryption and authentication and by employing e.g. non-repudiation techniques.

Security directly relates to integrity, since any system that is required to maintain a high level of integrity has to provide protection against attacks from the outside. If such protection is not provided, untrusted users (e.g. hackers) might succeed in bringing the system down, thereby reducing its integrity.

3.3 How network integrity can be affected

Network integrity can be affected by many factors. The purpose of this section is to stress some of these factors, without the intention to be complete.

One source of integrity breaches is defective hardware. If, for example, a switch fails, the integrity of the telecommunications network that uses this switch is at stake, since in the best case performance just

decreases. These single points of failure can be dealt with efficiently by adding redundant hardware to the telecommunications system, to ensure that backup resources are always available. Example backup resources range from extra power supplies, via unused fibers in fiber optical cables (to use in case a fiber brakes), to complete switches that are always standby to replace an existing switch in the case of a failure. By adding redundant hardware, availability of the system as a whole can be increased. Such backup equipment inherently costs money, so there is a trade-off between the costs of the network and the level of availability.

Another source of failure is software. If software fails, having an identical back-up switch present is of no use, since both switches contain the same software and will thus both fail. To help to avoid these problems, other, non-identical software equipped devices (performing the same function), should be present, or the software should be proven correct.

It is important to stress that designing and writing highly reliable software is not easy. This is especially so, since conflicts can arise between the criteria that together determine integrity, as described in subsections 3.2.1 through 3.2.12. For example, providing security usually means that computationally intensive cryptographic techniques have to be implemented, possibly influencing the timing-behaviour and performance of the software. Several papers about the influence of security on integrity have been written (e.g., [PSH98] and [Prnj99c]). The most important conclusion to be drawn from these papers is that it does not work to add security afterwards. It might have a significant impact on the timing-behaviour of the system. So, if security is an issue, it has to be identified in the first stage of the software development life cycle and taken into account all along the way to the implementation and testing of the program.

Another potential conflict is the performance/functionality trade-off. By increasing the functionality of a system, performance usually decreases, since performing a task becomes more complex (we assume here that ‘increasing functionality’ means that extra instructions are added to an existing procedure). The total number of these procedures that can be executed within one time unit thus decreases as a result of the extra computations/actions that have to be performed.

To summarise the above paragraphs: The required level of integrity influences the architecture of the system being developed and therefore has to be taken into account in the design phase. Integrity is usually not something that can be added afterwards.

Apart from failures due to incorrect hardware or software design, there are more sources of network failure. McDonald [McDo94] describes several of them, like ‘acts of God’ (hurricane, lightning, earthquake or flood), accidents (railroad derailment, car crashes or any other incidents damaging telecommunications equipment) or sabotage. The network operator cannot protect himself against these events for 100 percent.

In this thesis, we will not consider integrity breaches due to hardware defects, extreme weather, accidents or sabotage et cetera. Instead, we are concerned with software failures. So, in the remainder of this thesis we assume that all network hardware is working perfectly (and if not: that adequate back-ups are present) and that external events damaging the telecommunications infrastructure do not occur. So no fires at Hinsdale any more. Also, we will refrain from looking at procedural errors or inadequate maintenance. We are solely concerned with software causing integrity breaches and will thus not describe integrity problems that are caused by other factors.

3.4 Enforcing integrity

In the previous section we have eliminated many sources of integrity breaches, so that only one remains: software. However, since software errors that cause integrity breaches are still a too large area to consider, we will further narrow our view on integrity breaches to breaches that are caused by third party Parlay applications.

We are thus concerned with software that interacts with the telecommunications network and by doing so realises telecommunications features. Several outcries for a structured approach of the telecommunications software development process have been published the last few years. Prnjat and Sacks, for example, describe an approach for minimising the integrity risk during the early stages of telecommunications systems development [Prnj99a, Prnj99b]. The approach is based on system complexity, where several integrity analyses are being performed. If a component poses, according to the analyses, a too high threat to system integrity, it has to be redesigned. A design methodology for a formal and systematic framework to assess risks to network integrity is described by Mónton *et al.* [MWWM97].

Also, the use of formal languages for the design of telecommunications applications has been advocated. Dr. Audun Jøsang, for example, describes two formal languages, ANISE and CRESS, for defining telecommunications features [Josa99]. The main advantage of this approach is that it results in well-structured programmes, making life of the developer easier, since integrity risks can be assessed more quickly.

An approach using UML and SDL for developing telecommunications features, followed by a phase of integrity analysis, has been described by Koltsidas *et al.* [KPS00]. In this approach, UML is used for the specification of the requirements of the system under development. The application is then designed using SDL, the *Specification and Description Language* defined by the *International Telecommunication Union* (ITU[9] in short). To complete the design phase, integrity analysis (especially liveness analysis) is conducted using state-reachability analysis based on the SDL specification.

Although the approaches described above probably help with designing better applications, they suffer from one major disadvantage: access to the specification and/or design of the application is needed! So, these techniques cannot be used by the network operator to guarantee the integrity of his domain, when a third-party application is deployed. That would require access to the design and specification, which we consider is not the case. Moreover, these approaches are all concerned with the development phase, thereby excluding the possibilities of implementation errors.

So, the above-mentioned approaches are of no use to the network operator how wishes to safeguard his network against failures that are introduced by third party applications controlling network resources via the Parlay API. The only thing a network operator can influence regarding these third party applications, are the Parlay API calls that are made by that application. That is, the network operator has the possibility to restrict the usage of several API calls (dependent on the parameters provided), when he expects (or knows) that these calls will put the network in an unsafe state. To do so, the network operator's side of the Parlay API implementation should include an 'integrity gateway' function. This has been proposed in several papers [AAKS98, Josa99, Prnj00], although not always in the (specific) context of the Parlay API.

The question remains what this integrity gateway function should do in order to prevent integrity breaches from occurring. This will be, to some extent and in the context of the Parlay API, the subject of the remainder of this thesis. However, we still have not defined what exactly is to be considered as an integrity breach. Therefore, we will provide a definition, in the context of the Parlay API, of what we consider to be an integrity breach in the next section.

3.5 Definition of integrity, regarding the Parlay API

In the context of this thesis, we will define network integrity as follows:

> *Network integrity is the ability of the network to maintain a safe state, while performing within bounds that are acceptable to the network users, and delivering the functionality that is expected by the network users.*

This definition was chosen, since it reflects the majority of the definitions found in the literature and suits best to the intuitive meaning of integrity.

However, this definition is too vague to be generally usable, since it requires us to know what exactly is a safe state in the context of the Parlay API, and what is acceptable performance. To solve these problems, at least in part, we will give a list of situations that we consider as integrity breaches shortly. We will however refrain from saying something about the performance that is acceptable to the end users. The performance issue is considered to be handled at middleware (CORBA/COM) and signalling level and is therefore beyond the scope of this thesis.

If any of the following conditions hold, we state that integrity of the network, in the context of the Parlay API, is harmed. Note that it is the user's perception of the network state that determines whether network

[9] The ITU operates in three main sectors: radiocommunications, telecommunications standardisation and development. The telecommunications standardisation sector ITU-T was formerly known as CCITT, *Comité Consultatif International Télégraphique et Téléphonique*.

integrity is harmed. This does not necessarily imply that, technically speaking, something is wrong with the network.

Network integrity is said to be harmed if either of the following conditions hold:

- A requested service is not delivered to the end-user (either due to feature interaction with existing features or for some other reason).
- The total number of physical calls and/or call legs, created as result of a Parlay API invocation, does not correspond to the number of Parlay objects (objects implementing Parlay interfaces) that is in memory and still is being used, at any moment in time. So, if a situation occurs in which there is one physical call in the network, but there are two objects attempting to control that call, this is considered to be an integrity issue.
- Resources are used unnecessary. Resources can be either of:
    - Software resources (e.g. Parlay objects);
    - Communications links (telephone trunks, network connections);
    - Processing time / memory[10]
- There is a difference in state information as seen by the client application and the real network state.
- A service that is requested by the application is not delivered (e.g. a call is not routed to a specified number).
- Unauthorized access to network resources is gained.
- The network is in a state of deadlock.
- The network is in a state of livelock.
- The Gateway's or Client Application's load is too high, so that services are temporarily not available or the Parlay API operates outside acceptable performance bounds. It can be argued that a too high load on the Client Application is not a problem when seen from the public network operator's point of view (from the service provider's point of view, it *is* a problem), but network integrity can be harmed if the Client Application, as a consequence of this high load, does not respond to the events that are sent to it in due time.
- The network shuts down in total or in part.

The list above is somewhat arbitrarily chosen. Especially the fact that a difference in state information is to be considered as an integrity breach can give rise to discussion. However, it is the opinion of the author that all situations, in which the application has a different view of the network state than the real network state (according to the call model), are potentially dangerous. For example, if the application thinks that a call is still active, but in fact it has been terminated a while ago, this is potentially dangerous. Good design should prevent these situations from occurring, and therefore we term a difference in state information, whether or not decisions are actually based on it, as an integrity breach.

[10] With this item we do not intend to say that network integrity is harmed if the implementation of a service is not optimal in terms of CPU-cycles or memory usage (i.e., a more efficient implementation would be possible), but we mean that integrity is harmed when processing time and memory are wasted, in the current implementation, without performing any useful function.

Chapter 4

Verification of the Parlay API using SPIN/Promela

In this chapter we report on our efforts to formally verify whether (part of) the Parlay API poses threats to the integrity of the underlying telecommunications network, using the model checker SPIN. Section 4.1 provides some general information about the verification tool SPIN and its input language Promela. In section 4.2 the model that has been used for our verification purposes is described in detail. Section 4.3 summarises the results that have been obtained during the verification runs. During the verification process some problems were encountered regarding the use of SPIN. These are described in section 4.4. Finally, section 4.5 gives several hints on how to improve the model.

4.1 About SPIN/Promela

SPIN is a verification system that supports on-the-fly formal verification of distributed systems. It has been developed by Gerard J. Holzmann at Bell Labs (the research division of Lucent Technologies), Murray Hill, USA. Work leading to the development of SPIN began in the 1980's with the construction of the verifier *pan*. SPIN version 1.0 was released in 1991, accompanying Gerard Holzmann's book on SPIN [Holz91]. Several technical improvements have been made to SPIN, since. Currently, the most recent version is 3.4.7, which dates from April 2001.

4.1.1 General description

SPIN verification models are focused on proving the correctness of process interactions. In SPIN, processes can communicate with one another by sending messages over typed channels, or via access to shared memory.

Message passing can either be synchronous (i.e., rendezvous) or asynchronous (i.e., buffered). Rendezvous communication can be seen as consisting of two steps, a rendezvous offer (send) and a rendezvous accept (receive). For a rendezvous communication to succeed, it is required that, once a rendezvous offer has been made, an active process is able to perform the matching rendezvous accept immediately, that is, without the execution of intervening steps by other processes.

In SPIN, every statement is either executable or not. A statement that is not executable blocks until it becomes executable. A rendezvous send will thus block if there is no active process willing to match the rendezvous send with a corresponding rendezvous receive. Likewise, a rendezvous receive will block if there is no matching send.

In asynchronous communication, messages are sent across buffered channels. The size of the buffer is bounded. Asynchronous send operations are executable if and only if the buffer is not full. If the buffer is full, the asynchronous send operation will block[11]. A buffer behaves as a *first-in-first-out* (FIFO) queue by default, adding messages to the tail of the queue, and removing them from the head of the queue. SPIN also supports sorted send operations, in which the buffer no longer forms a FIFO queue but messages are sorted based on their content.

[11] This is the default behaviour of SPIN. SPIN can, however, be instructed to lose messages if the buffer is full. In that case, asynchronous send operations are *always* executable.

An asynchronous receive is executable if and only if the buffer contains at least one message, matching the (possibly empty) receive criteria of the receiving process. These 'receive criteria' are constraints on the values of the different fields in the message, e.g. message type.

Promela (an acronym for Process Meta Language) is SPIN's input language. Its syntax somewhat resembles the programming language C. It supports booleans, bits, bytes, shorts and integers as data types, which can be composed into (fixed size) arrays or user-defined types using a `typedef` construct similar to the one in C. Promela also supports a special type `mtype`, to define message types (although usage is not restricted to this purpose). Once a symbolic constant is declared as part of an `mtype`, the same name cannot be used for any other purpose.

Especially user-defined types are 'syntactic sugar' in Promela, since it is not possible to assign the value of one user-defined type to another, without explicitly copying every field by itself. User-defined types can be used as parameters in messages, but are broken up into separate fields when sending the message. So, while it is possible to introduce user-defined types, Promela does not provide operations that work on user-defined types, such as assignment. Usage of such types is therefore limited to enhance readability and understanding of the model. Promela supports do-loops and if-statements. As opposed to C, Promela is a non-deterministic language, meaning that alternatives in if-clauses (and in Promela also in do-loops) do not have to exclude each other and thus more than one alternative can be executable at any moment in time.

It is out of the scope of this thesis to discuss Promela in all detail. However, we will be referring to several Promela language constructs in the remainder of this chapter. The purpose of these constructs will be explained when needed. A detailed description of the Promela language can be found in the language reference [PROM].

4.1.2 Using SPIN

Given a Promela model as input, SPIN can be used to perform a syntax check on the Promela model. If the model satisfies the syntax requirements, SPIN can be used in either of the following modes:

a) As a *simulator*. In this mode SPIN performs a random, a guided or an interactive simulation of the input model. In a random simulation, SPIN uses a pseudo random generator to resolve the non-determinism in the model. That is, if there is more than one statement executable at a time, SPIN will choose one at random. The pseudo random number generator that is used in SPIN takes a seed value as input. If the user changes this seed value, a different execution is obtained. A random simulation can be used to see if the model behaves as expected. Guided simulations are used to play back error trails that have been generated by the verifier (see below). During interactive simulation, SPIN displays a list of enabled transitions, letting the user choose the one to be executed. In interactive simulation, the non-determinism is thus resolved by the user. This mode can for example be used to quickly lead the model into an error scenario.

b) As an *exhaustive state space analyser (verifier)*. In this mode, SPIN attempts to prove[12] the validity of user-specified correctness requirements.

c) As a *bit state space analyser (approximate verifier)*. In this mode, SPIN performs an approximate search, reducing the memory requirements by usually several orders of magnitude to just a few bits per reachable state. This mode is used to verify large systems, in those cases where an exhaustive search is not possible, due to the memory requirements involved. This mode can be used to find errors in the model, although such an approximate search cannot guarantee the absence of errors, since it usually does not cover the complete state space. More on this so-called bitstate hashing technique can be found in [Holz98].

The user can specify correctness requirements in several ways. The first way is to express correctness requirements as assertions, using Promela's `assert` statement. These then serve as system or process invariants. Another way is by means of *Linear-time Temporal Logic* (LTL) formulae. Although these cannot be entered directly into a Promela model, SPIN provides a command line parameter to translate an LTL-formula into a so-called *never* claim, which does satisfy the Promela syntax requirements. In verification

[12] As with almost every model checker, we must keep in mind that SPIN is a research tool and that, although it has been used in a wide range of applications and is perhaps one of the most popular and powerful model checkers around, no absolute guarantees on the correctness of the output of the verifier are given.

runs, the never claim (which is in essence a separate process) is executed interleaved with all other processes in the Promela model, to check its validity.

SPIN can also be used to check for the absence of deadlocks, or to perform liveness analysis on the model. For the latter purpose, SPIN uses either acceptance or progress labels. An acceptance label in the model defines the correctness requirement that the labelled global system state can only be visited finitely many times in any infinite system execution[13]. A progress label defines the correctness requirement that the labeled global system state is required to be visited infinitely often in any infinite system execution.

4.1.3 Obtaining and installing SPIN

SPIN is distributed free of charge for research and educational purposes only and can be downloaded via anonymous FTP from ftp://netlib.bell-labs.com/netlib/spin/. Download via HTTP is also possible, from site http://netlib.bell-labs.com/netlib/spin/index.html.

SPIN runs on Unix-based systems, as well as on PC's under Windows 95/98/ME or Windows NT. For Windows, SGI and Linux machines, there are binary executables available for download. The full source, which is written in Ansi/Posix C and Yacc, is available for download on Unix machines. There is no binary executable available for Unix (one has to compile it oneself), but the installation process on Unix is documented reasonably well.

SPIN also comes with a graphical user interface called XSPIN, which has been written in Tcl/Tk. On most Unix-like systems, all necessary files to run SPIN and its graphical user interface (C-compiler, preprocessor, Tcl/Tk and optionally the program *dot* to beautify the layout of the state diagrams that can be produced with SPIN) will be readily available. On Windows systems, these however are not. SPIN's `README.html` file provides links to Windows versions of the required software. The installation process on Windows machines is rather complicated, since it is not automated and requires several files to be edited, moved or renamed by the user. It also requires the modification of several environment settings in `AUTOEXEC.BAT` (on Windows 95/98/ME) or in the Control Panel (Windows NT).

The installation instructions for Windows-based PC's provide enough detail to get SPIN and XSPIN working. The author succeeded in installing SPIN and XSPIN under Windows NT 4 with Service Pack 4 as well as under Windows ME, both within the hour. To get the optional *dot* working one must not forget to copy the file `dot.exe` into SPIN's `bin` directory (this is not mentioned in the installation instructions, but *is* necessary to get *dot* working).

More information about SPIN can be found in [Holz91, Holz97]. The SPIN homepage also contains a lot of valuable information regarding SPIN, its application domains, and detailed installation instructions. The SPIN homepage is located at http://netlib.bell-labs.com/netlib/spin/whatispin.html (note the single s!).

4.2 Description of the model

For our verification purposes, we decided to model (part of) Parlay's Conference Call Control Service (CCCS). It was a deliberate choice to model this part of the call processing services offered by the Parlay API and not e.g. the Generic Call Control Service (GCCS), since some prototyping experience with the Parlay API, and specifically the GCCS, had already been gathered within Lucent Technologies. In the first few months of 2000, a project took place within the Forward Looking Work department in the EMEA region, which aimed at prototyping two Parlay applications to test the feasibility of the Parlay API. Although this project was not primarily concerned with integrity-related issues, lots of knowledge about the GCCS had been gathered [HPW01]. Since little is known about the Conference Call Control Service, this provided a nice challenge to model (part of) the CCCS.

4.2.1 Scope

Given the fact that the Conference Call Control Service is rather complex, it was certainly not possible to model the complete Conference Call Control Service, given the time available. In order to be able to extend

[13] Since in SPIN all message fields, queues and data types are bounded, just as the maximum search depth is, infinite executions can only occur if there are cycles.

the model in a later stage, a structured approach to modelling the API has been taken. The idea was to add one operation at a time, leaving out as many parameters as possible without harming the basic semantics of the operations. To illustrate this, consider the `reserveResources()` operation that is provided by the `IpConfCallControlManager` interface. According to the Parlay API specification, it returns a field of type `TpAddress` that specifies the address that can be used to dial into the conference. In our model we use a different way to identify the conference (namely its ID), since we abstract from the underlying network and so this field has been omitted in our model.

Where possible, every time a new Parlay operation was added to the model, it was fed into SPIN to check for syntax errors and to perform some random or interactive simulations, to see whether the newly added operation behaved as intended.

During the modelling phase, some serious problems were encountered. These will be described in more detail in sections 4.3 and 4.4. As a consequence, the modelling process was delayed significantly. In the end, we managed to model only the following Parlay operations:

- `IpConfCallControlManager.reserveResources()`
- `IpConfCall.leaveMonitorReq()`
- `IpCallLeg.attachMedia()`
- `IpCallLeg.detachMedia()`
- `IpAppConfCallControlManager.conferenceCreated()`
- `IpAppConfCall.partyJoined()`
- `IpAppConfCall.leaveMonitorRes()`

With this set of operations, conferences can be created by the network, in response to an earlier reservation request. Once a conference is created, parties can dial into the conference, or – if they are in the conference – leave the conference. The Parlay client application can be informed about these events. The client application can also decide to attach or detach the media channels associated with a party's call leg.

Although this is a relatively small subset of the complete set of Parlay operations, it serves to get the basic idea and even suffices to show some integrity risks. In the following four subsections, we describe the architecture of the model in more detail and explain its functionality briefly. Also, we describe the preconditions and invariants that are supposed to hold in the model, so that the API calls do not put the network in an unsafe state.

4.2.2 Architecture

The core of the model consists of two processes, called `NetworkApplication` (which serves as an alternative name for the Parlay Gateway) and `ClientApplication`[14]. These two processes communicate with each other across a synchronous (= rendezvous) channel. This channel is named `Parlay` and all messages that are sent via this channel are designated to be Parlay API calls. The model's architecture is graphically depicted in Figure 6.

The process `NetworkApplication` serves as an abstraction of the Parlay Gateway that would be run by the network operator in his telecommunications network. Apart from implementing the behaviour of the Parlay Gateway, this process also serves to simulate events that occur in the network. That is to say, it models network events that could have been generated by network users had the Parlay API been deployed in a real telecommunications network, such as for example a party dialling into a conference.

The process `ClientApplication` serves as an abstraction of a Parlay Client Application that would be run by a value-added service provider and also serves to abstract from the users of that application. That is to say, the process `ClientApplication` also models those decisions that, had the API been deployed in a real telecommunications network, would have been made by the users of the application or the application itself.

[14] Strictly speaking, this is not true. In Promela, processes cannot be named. Every process in the system is an instance of a so-called process type. This process type defines the behaviour of the processes that are instantiated from it. A process is thus defined by its process type and instantiation number. In this thesis, we will refer to the processes with the name of the process type if there is only one instance (as is the case with `ClientApplication` and `NetworkApplication`).

For example, in a real Parlay application, an `attachMedia()` API call might be issued by the client application in response to an event triggered (e.g. via a web interface) by the chairperson of the conference call.

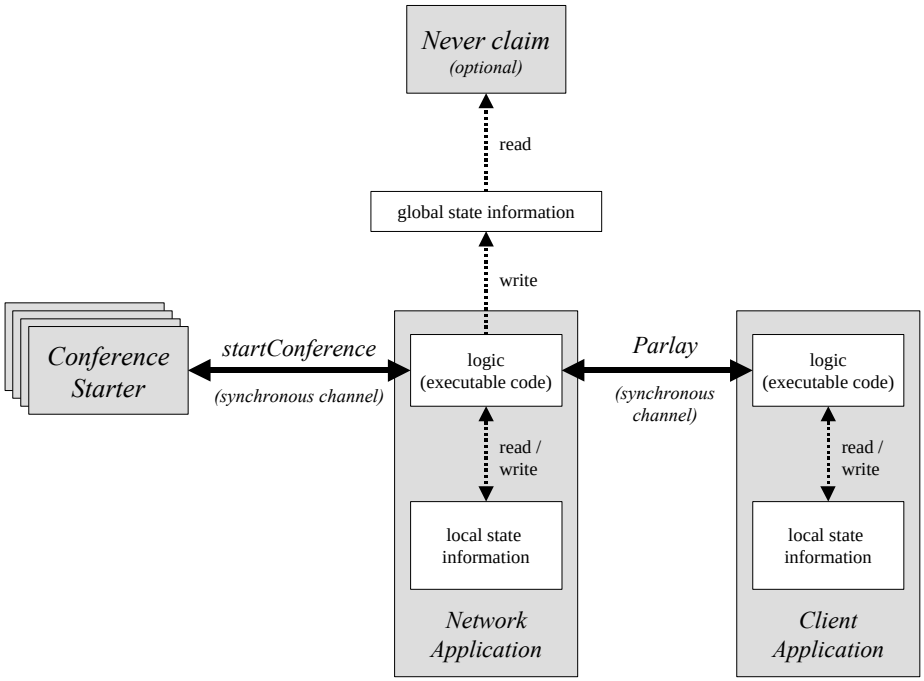

Figure 6: Model architecture

Both the `NetworkApplication` and `ClientApplication` maintain their own, local state. The primary component of this local state is the view on the state of the underlying network. The local state thus contains information about the status of the conferences and call legs, such as the number of conferences that are in use, the number of call legs in each conference, whether the media channels are attached or not, et cetera. Since the `NetworkApplication` runs in the network domain, it knows the *real* network state. The `ClientApplication` only has a *view* on the network state, which it can update depending on the API calls issued. Apart from this conference and call leg related status information, the local state includes some more variables that are used for bookkeeping purposes.

The process `NetworkApplication` uses auxiliary processes (instances of the `ConferenceStarter` process type) to start a reserved conference 'at the appropriate time' after it has been reserved. The text 'at the appropriate time' is enclosed in quotes here, since our model abstracts from the concept of time. During a verification run, SPIN will take into account all possible relative speeds of the different processes and thus abstract from 'real' time. For the communication between the processes `NetworkApplication` and `ConferenceStarter`, a separate, synchronous channel is used.

All processes together share a global state, which contains, in our model, only the variable `nrOfErrors`, which is prefixed by the Promela keyword `local`, to declare that the variable is part of the global state, but is used by only one process. By doing so, SPIN can treat the variable as being local, so that local optimisations can be applied, but it can also be used within LTL claims (which is only possible with variables that are part of the global state).

4.2.3 Implementation

In this subsection we will describe in short how the model is implemented. There are two topics worth mentioning, namely the usage of Parlay interfaces, and the constructs that have been used to model the API calls. Some knowledge about the latter is needed in order to understand how and why the model works.

4.2.3.1 Parlay interfaces

The Parlay API specification prescribes that the network application implements the following five interfaces:

- `IpMultiMediaChannel`
- `IpConfCallControlManager`
- `IpConfCall`
- `IpSubConfCall`
- `IpMultiMediaCallLeg`

The interfaces `IpSubConfCall` and `IpMultiMediaChannel` are not incorporated in the model, since the model does not support any operations offered by these interfaces. The other three interfaces are present, but are not recognisable as such. For reasons of preserving the message order, the operations that are offered by these interfaces (if implemented) have been grouped together. In our model, Parlay operations are thus not invoked on any specific interface.

According to the Parlay specification, the operations `attachMedia()` and `detachMedia()` require one parameter, namely the session ID of the call leg. This means that either the session ID of the call leg has to be unique among all other call leg session IDs provided by the network application, or a separate `IpMultiMediaCallLeg` interface is to be used for every conference. In the former case, the session ID suffices to determine which call leg in which conference is meant. In the latter case, the call leg is uniquely determined by session ID and conference ID (which can be obtained since the interface on which the message is received, is known). Since both options were not feasible for our model (due to the structure chosen to represent network state information), this problem was solved by adding an extra parameter to the `attachMedia()` and `detachMedia()` operations: the conference ID. A call leg is in our model thus uniquely identified by the combination of call leg session ID and conference session ID.

The Parlay API specification prescribes that the client application implements the following four interfaces:

- `IpAppMultiMediaCallControlManager`
- `IpAppConfCall`
- `IpAppSubConfCall`
- `IpAppMultiMediaCallLeg`

The last two interfaces are not present in our model, since it contains no operations requiring the presence of any of these interfaces. The remaining two interfaces are present, but are not recognisable as such. Just as has been done with the interfaces implemented by the `NetworkApplication`, these are also grouped together.

Since all operations are grouped together, the references to the Parlay interfaces that are present in various Parlay API calls, carry no meaning any longer. Therefore, to reduce the size of the state-vector, these fields have been removed from the operations' parameters. Due to the fact that most Parlay operations are given a unique name, grouping them together poses no problems. There are, however, some operations that are not named uniquely. One example is `release()`, which can be invoked on almost any interface. If any such operation is incorporated in the model, the names have to be made distinct or another way to distinguish the various types of operations has to be used.

4.2.3.2 Control flow

In this sub-subsection, we describe how the model works in detail. The full Promela source of our model, with a total of 2 conferences of 4 call legs each, is included in Appendix B. The reason why in this model the values 2 and 4 are hard-coded is described later, in section 4.3.1.

Both the `ClientApplication` and the `NetworkApplication` are modelled using a Promela `do`-construct. This construct is a combination of a well-known do-loop and a non-deterministic if-clause. It is denoted in Promela as follows:

```
do
::  /* alternative 1
       (code fragment) */

::  /* alternative 2
       (code fragment) */

::  /* alternative 3
       (code fragment) */

…
…

::  /* alternative n
       (code fragment) */
od
```

Either of the alternatives may be executable or not (depending on the first statement). In every iteration through the do-loop, SPIN chooses one of the executable alternatives (delimited by the double colon on the beginning of the line) and executes it. If none of the alternatives is executable, the do-construct as a whole blocks until at least one of its alternatives becomes executable again.

In our model, all the alternatives inside the do-loops have the following form:

```
atomic {
  /* guard condition */
  ->  /* statement 1 */
      /* statement 2 */
      …
      …
      /* statement m */
}
```

The Promela keyword `atomic` instructs SPIN to treat all code between the opening brace following the `atomic` keyword and the matching closing brace as one indivisible block (a so called atomic sequence). That is, once its first statement is executed, the execution must continue with the next statement in the code block and cannot be interrupted by intervening steps made by other processes (which would have been the case if the statements were not located inside an atomic sequence). There is one exception to this rule: non-deterministic execution of executable statements is resumed once a statement inside an atomic sequence blocks.

The first statement inside this atomic sequence is its guard condition, and the atomic sequence can thus only be entered if the guard condition is executable. If it is executable, and SPIN chooses to execute this statement, the execution must continue with the statements *1* through *m*, until one of these statements block. After statement *m* has been executed, the atomic sequence is completed and normal, non-deterministic process interleaving en execution is continued.

In our model we used two different kinds of guard conditions, each corresponding with a different type of action. As has been mentioned before, the processes `ClientApplication` and `NetworkApplication` both handle incoming Parlay calls, but are also used to respond to events originating from the network or decisions that could have been made by application users.

The guard condition for an incoming Parlay call consists of a rendezvous accept. For example, the guard condition for an `attachMedia()` call is as follows:

```
Parlay?attachMedia(confSessionID, callLegSessionID)
```

It is executable if and only if another active process is willing to send an `attachMedia()` message over the channel `Parlay`. Since both `confSessionID` and `callLegSessionID` are variables, no additional constraints on the parameters passed via that message are set. If either of them would have been constants, the guard condition would only be executable if the parameter in the corresponding send statement matches the value of the constant in the receive statement.

The second type of guard condition we have used in our model is a logical constraint on the network state, as viewed by the applications. Such a guard condition is executable if and only if the logical formula evaluates to true. As guard condition for the event 'a party joins the conference with conference session ID 1' we need to express that the conference has already been started by the network and that the party was allowed to join this conference. In Promela:

```
network.conference[1].used && network.conference[1].joinAllowed
```

The preconditions that have to be satisfied before API calls can be issued, are described in more detail in section 4.2.4.

If an incoming Parlay call is received, the typical thing the receiver has to do is process the API call, which results in a change of the network state, as viewed by the receiver. If needed, a reply[15] has to be sent to the application issuing the Parlay call. Therefore, all alternatives regarding the processing of incoming Parlay calls are structured as follows:

```
atomic {
  Parlay?messagetype(param1, param2)
  ->  /* update local state information here */
      ...

      /* if needed, send reply: */
      Parlay!messagetype(param1, param2)
}
```

The words in Italics are to be replaced by the proper values. As a result of a network event or a decision made in the application domain, one typically has to issue a Parlay API call, and, optionally, wait for the reply. This kind of alternative is thus structured as follows:

```
atomic {
  /* guard condition on network state as viewed by application */
  ->  /* update local state information here */

      /* send Parlay API call: */
      Parlay!messagetype(param1, param2)

      /* if needed, wait for reply: */
      Parlay?messagetype(param1, param2)
}
```

Synchronous send and receive statements within atomic sequences have a special meaning in Promela, which is exploited in our model to avoid deadlocks. The Promela Language Reference [PROM] states the following:

> "If an atomic sequence contains a rendezvous send statement, control will pass from sender to receiver the moment that the rendezvous handshake completes. Control can return to the sender at a later time, under the normal rules of non-deterministic process interleaving, to allow it to continue the atomic execution of the remainder of the sequence. In the special case where the recipient of the rendezvous handshake is also inside an atomic sequence, atomicity will flow with the rendezvous and is not interrupted (except that another process now holds the exclusive privilege to execute)."

Due to this property of synchronous send and receive statements within an atomic sequence, we have accomplished the following situation (provided that no statement blocks): In response to a network event or decision, the local state information is updated first. Then, a Parlay API call is sent, and atomicity and control will flow with the API call to the other process, since the receiving statement is within an atomic sequence. At this stage, the receiver holds the exclusive privilege to execute statements and thus continues by updating its local state. Then, a reply is sent, and atomicity and control will flow with this message. The receiver (which is the sender of the first Parlay call) can now resume its execution. Meanwhile, no other steps can be

[15] Separate reply messages are not part of the Parlay API specification. They have been introduced in our model since, in Promela, parameters can only be passed in one way at a time. Reply messages are thus the Promela-counterpart of output parameters in the Parlay API specification.

executed by any process. We have thus created a kind of *channel polling* mechanism: messages can only be processed when both applications are ready to do so. This automatically implies that messages are received in the same order as they were sent. While processing the messages (i.e., API calls), no other events can occur. Note that first issuing a Parlay call and then updating local state information results in a deadlock if a reply message is to be received.

4.2.4 Preconditions

In this subsection, we describe the preconditions that have to be satisfied before the Parlay operations that are contained in our model can be invoked. By doing so, we describe the operational environment of both the network and client applications. For clarity, the preconditions are shown per API call. They are described in Table 1. The last column contains line numbers, referring to Appendix A. On the lines specified, the Promela constructs that express the precondition can be found.

API call	*Precondition*	*Line*
`conferenceCreated()`	The conference has been started by the network, as a result of prior reservation.	262
`partyJoined()`	The conference on whose behalf this operation is invoked, is in use (that is, a conference with the correct conference ID exists);	281
	Parties are allowed to join the conference;	282
	The number of parties that already have joined the conference is still below the maximum number of parties that is allowed to join.	283
`leaveMonitorRes()`	The conference is in use;	303
	The call leg is in use;	304
	A `leaveMonitorReq()` has been issued for the conference.	311
`reserveResources()`	There are fewer conferences reserved than is allowed according to the Service Level Agreement.	391
`leaveMonitorReq()`	The conference is in use;	405
	A `leaveMonitorReq()` has not yet been issued for this conference (the Parlay API does not require this. It has been added as precondition to limit the number of non-deterministic choices that can be made in the model).	406
`attachMedia()`	The conference is in use;	422
	The call leg is in use;	423
	The call leg is not yet attached (the Parlay API does not require this. It has been added as precondition to limit the number of non-deterministic choices that can be made in the model);	424
	An `attachMedia()` operation has not yet been issued for this call leg (included to limit the number of non-deterministic choices);	425
`detachMedia()`	The conference is in use;	441
	The call leg is in use;	442
	The call leg is not yet detached (the Parlay API does not require this. It has been added as precondition to limit the number of non-deterministic choices that can be made in the model);	443
	A `detachMedia()` operation has not yet been issued for this call leg (included to limit the number of non-deterministic choices).	444

Table 1: Preconditions for the various API calls

As can be deduced from the description of the preconditions shown above, a few extra preconditions have been added to our model, that are not required by the Parlay API. These preconditions serve only one goal, and that is to reduce the state space. Theoretically, the media associated with a call leg can be attached or detached arbitrarily many times. However, since SPIN requires that the system that is being verified is bounded (that is: the state space is finite, and the model can thus be verified within finite time, given enough resources), we could not allow this to happen in our model. Therefore, the number of operations that is invoked by the client application, and the number of events that is reported by the network, has been constrained using the preconditions shown in the table above.

4.2.5 Invariants

In this subsection, we describe the invariants that have to be satisfied just *after* the invocation of a Parlay API call. The invariants express the fact that the Parlay API call is allowed in the current situation. If, during verification, an invariant fails, there was a misalignment between the precondition and the invariant, or the execution of the statements in the model was interleaved in a way that was not meant to happen (e.g., because atomicity was broken while executing an atomic sequence).

The invariants are described in Table 2 below. The line numbers in the last column once again refer to Appendix A.

API call	*Invariant*	*Line*
`conferenceCreated()`	A conference with the specified ID does not already exist.	346
`partyJoined()`	A conference with the corresponding conference ID is in use.	361
`leaveMonitorRes()`	A conference with the corresponding conference ID is in use;	376
	The application must have asked for the monitoring of party leave events for that conference;	377
	The party exists (i.e., its call leg is in use).	378
`reserveResources()`	This reservation request is still allowed, i.e., the Service Level Agreement still allows for the reservation of conferences (the number of reservations is still below the specified maximum).	200
`leaveMonitorReq()`	The conference is in use.	221
`attachMedia()`	The conference is in use;	232
	The call leg is in use.	233
`detachMedia()`	The conference is in use;	249
	The call leg is in use.	250

Table 2: Invariants for the various API calls

In the current model, parties are always allowed to join, therefore, the fact that parties must be allowed to join the conference before the `partyJoined()` API call may be invoked, is not expressed as an invariant, since in the current model there are no possibilities to create a conference in which parties are not allowed to join.

Apart from these invariants, several other `assert` statements can be found in the model. They serve to check the overall sanity of the model and, should there be an error in the model, prevent run-time errors from occurring due to, for example, array subscription with a wrong index. Instead, the error is reported as an assertion violation, making it easier to locate the problem (error trails can be generated and examined).

4.3 Verifying the model

Our verification experiments were mainly focussed on two aspects:

a) Proving that no deadlock could occur

b) Proving that all Parlay calls that were made, were valid (that is, that all invariants described in section 4.2.5 were satisfied)

As we have outlined before, in our model, API calls are only made when they are valid according to the local view on the network state. For example, the `ClientApplication` will not issue an `attachMedia()` request for a specific call leg in a specific conference before, according to its own vision on the network state, that call leg is created. The invariants are thus included in the model to find out if, and how, *differences* in the various views on the network state can occur. If such differences arise, either one of the applications must have a faulty vision on the network state. Since the `NetworkApplication` runs in the network, its state information is presumed to be correct. Therefore, the state information that is maintained by the `ClientApplication` must be wrong.

In the next subsection we will describe how the verification experiments were conducted. In subsection 4.3.2 we will summarise the most important results obtained with our verification experiments.

4.3.1 Verification setup

We have conducted a series of verification experiments, varying the total number of conferences and the maximum allowed number of call legs per conference. The total number of conferences and the maximum number of call legs per conference determine what network events can occur and how many Parlay calls can be issued. In fact, they could be seen as constraints that are placed on the resource usage, which typically are defined in a Service Level Agreement. For example, if there are only two call legs per conference, a third `partyJoined()` API call will never be invoked. Since, to the best of our knowledge, Promela provides no efficient way to encode this, parameterised with the maximum number of call legs and conferences allowed, we used the powerful macro processor *m4* [Sein94] to solve this problem (see also [Ruys01]).

An *m4* source file has been written, which is to be parameterised with the total number of conferences and the maximum number of call legs per conference. Depending on these parameters, *m4* generates a Promela source file that is specifically tailored to verification runs for that number of conferences and call legs. The full *m4* source file that can be used to produce a tailored Promela model is included in Appendix A. The output of *m4* for two conferences with four call legs each is included in Appendix B. This file has been generated with the command

```
m4 -DCONFS=2 -DCALLLEGS=4 cccs.m4 > cccs_2_4.pml
```

The default value for the number of conferences is set to 2, and the default number of call legs per conference is 4. The two `-D` parameters in the command above could thus have been omitted. The command

```
m4 -DCALLLEGS=3 cccs.m4 > cccs_2_3.pml
```

generates the Promela source `cccs_2_3.pml`, tailored to two conferences with three call legs each, and

```
m4 -DCONFS=1 cccs.m4 > cccs_1_4.pml
```

produces the Promela source for one conference, with four call legs.

Although it is not required to supply any of these parameters, it is advised for reasons of clarity to always provide both the `-DCONF=`*X* and `-DCALLLEGS=`*Y* parameters. Note the three consecutive L's in `CALLLEGS`. The header of the generated Promela file contains the number of conferences and call legs the model is generated for.

Generating the full Promela source using *m4* is only the first step of the largely automated verification process. The next step in the verification process is to use SPIN to create the C source for the verifier. This source is then compiled, using the C compiler, into an executable verifier. The compilation process is provided with several parameters about the sort of verifier that is required (exhaustive/bitstate), the aspects to verify (assertion violations, invalid endstates, acceptance cycles, never claim, et cetera), techniques used (better compression, partial order reduction) and the maximum amount of memory to use.

The last step in the verification process is to run the verifier, providing it with some parameters to indicate the maximum search depth and the size of the hashtable. The hashtable is used by the verifier to provide efficient memory storage.

Almost all verification runs took place overnight or during weekends on a Sun UltraSparc-IIi processor running at 440 Mhz under SunOS release 5.6. Several simple *Korn* shell scripts controlled the whole process of compiling and executing the verifier with the correct parameters and logging the results and the run-time consumed, as reported by the Unix command *time*, to file. The whole process is graphically depicted in Figure 7. With the 'equal' sign between the two boxes labelled *pan*, we intend to express that first the verifier is generated by the C compiler, and subsequently executed.

For our verification runs, we used SPIN version 3.4.7, dated April 23, 2001, GNU *m4* version 1.4 and the standard C compiler *cc* (WorkShop Compilers 4.2 30 Oct 1996 C 4.2).

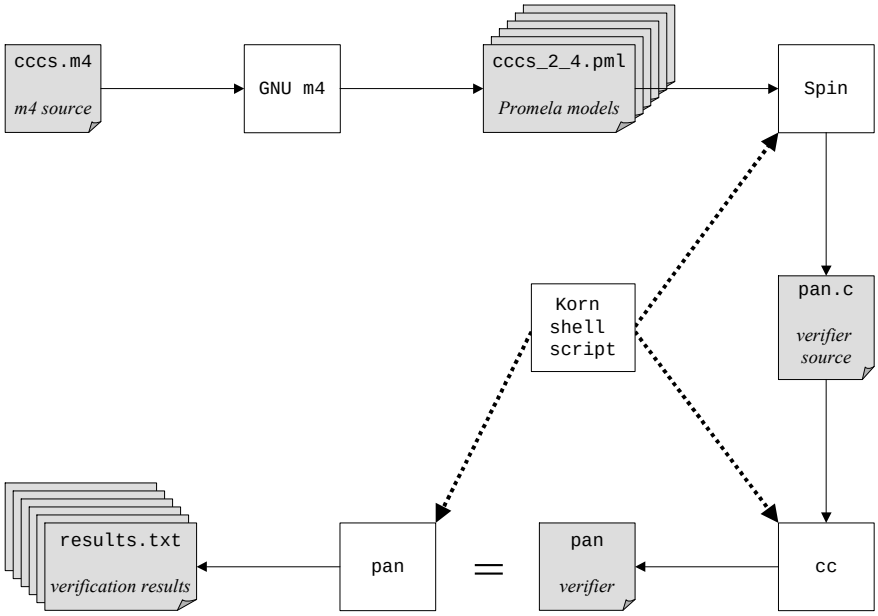

Figure 7: Verification setup

4.3.2 Verification results

In this subsection we report on the verification results that have been obtained while verifying the model against the invariants that are supposed to express the proper functioning of the model and the Parlay API, regarding integrity issues.

4.3.2.1 The first error

Already during one of our first verification experiments, it became clear that it is very easy for the `NetworkApplication` and `ClientApplication` to obtain a *different* view on the network state. According to the Parlay API specification, the event that a party leaves a conference is only to be communicated to the client application (via the API call `leaveMonitorRes()`) if this was previously requested by the application (via the API call `leaveMonitorReq()`). Only when the client application is informed about the event that a party has left the conference, it can update its state accordingly, thus preventing the invocation of erroneous operations. With the term erroneous operations in the previous sentence, we mean operations that are allowed according to the client application's local state, but whose call leg ID no longer matches an existing call leg. According to the real network state, the operation is thus not allowed.

In one of our earliest models, every API call was guarded by an `assert` statement, expressing the fact that the conference or call leg that the operation was called upon, still existed. During verification, this thus led to an assertion violation every time an operation was called with a non-existing call leg or conference as parameter. Limiting ourselves to the seven Parlay API calls that are part of our model, such a situation can occur if (and only if) an `attachMedia()` or `detachMedia()` call is made, prior to a `leaveMonitorReq()` call. In that case, the party for which the call is being made may already have left, without the application knowing about it. Apart from these API calls, there are other calls that have to suffer from this inconsistency, such as the `release()` operation. In general, all Parlay API calls that require that a specified party is available before the API call can be invoked, suffer from this error.

So, we have found our first integrity risk. In order to be able to search for other errors, the `assert` statement from our previous model was replaced by an increment of the variable `nrOfErrors` in the current model. The verifier will thus no longer halt on every such situation, but continue its search for other errors.

It is still possible in the current model to obtain an error trail of a situation as mentioned above. The error trails will not be produced in normal verification runs, checking for the absence of deadlock and assertion

violations, but we can 'force' the generation of such an error trail by using SPIN's capability to verify LTL formulae.

The LTL formula

```
<> p
```

with `p` defined as

```
(nrOfErrors == 1)
```

states that eventually `p` must hold, and thus describes error behaviour (it is meant to be invalid in all executions). We can now use SPIN to translate this claim into a never claim and subsequently use SPIN to prove that this property is not satisfied. The shortest possible error trail for our model with one conference and two call legs consists of 33 process steps. The sequence of Parlay API calls that is issued in this error trail is shown in Figure 8. Note that arrow numbers 2 and 4 do not depict Parlay API calls; arrow number two is not even a message that is passed between processes in our model. It is just included for clarity, so that it becomes clear when the process `ConferenceStarter` is instantiated.

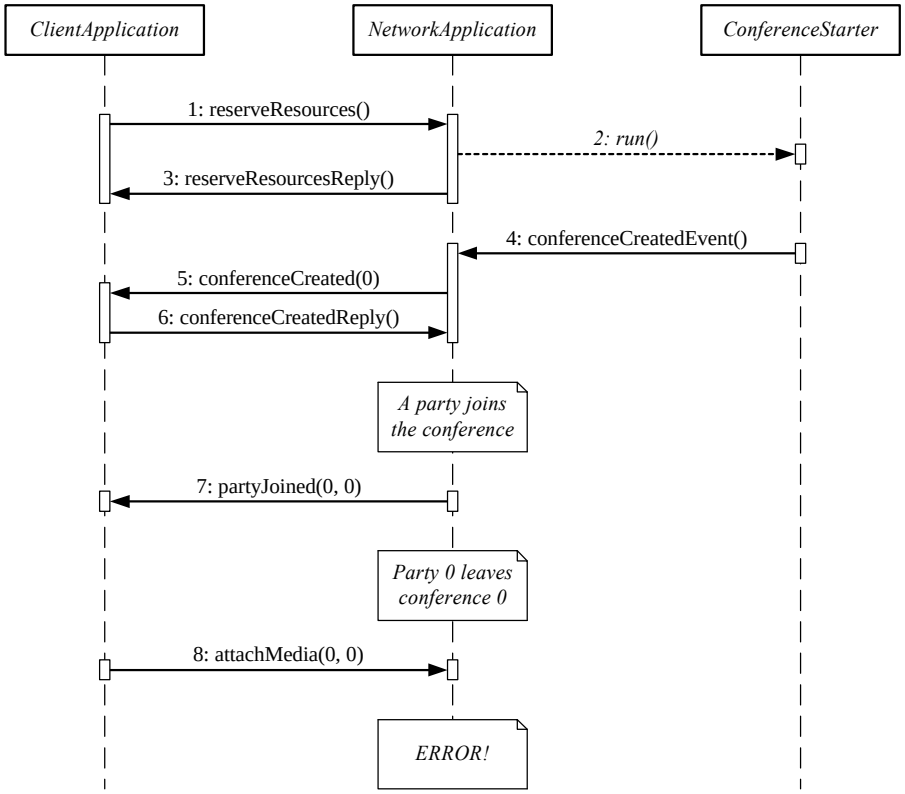

Figure 8: Error trail for the LTL property `<> (nrOfErrors == 1)`

The first six arrows depict the process of conference reservation and the initiation of the conference by the network. Its conference session ID is 0 (shown in message 5). Arrow 7 indicates that a party joins the previously created conference and is awarded session ID 0 by the network. Following the join, the party leaves immediately. However, this is not communicated to the Parlay application, since a `leaveMonitorReq()` was not issued yet. Therefore, the invocation of the `attachMedia()` operation on conference 0, call leg 0, leads to an error, since the guard

```
network.conference[confSessionID].callLeg[callLegSessionID].used
```

is no longer fulfilled.

This integrity issue is easy to resolve by changing the Parlay API specification in such a way that 'party leave' events are *always* reported to the application. The `leaveMonitorReq()` operation then becomes superfluous.

In the rest of our verification experiments, we paid no further attention to these scenarios and focussed on finding other erroneous situations instead.

4.3.2.2 Exhaustive verification

It soon became clear that even a small part of the Parlay API (i.e., the seven operations in our model) is hard to verify using exhaustive search techniques, since the state-space is too large. We have conducted a range of experiments with six versions of our model:

- 1 conference, 2 call legs per conference (1 – 2)
- 1 conference, 3 call legs per conference (1 – 3)
- 1 conference, 4 call legs per conference (1 – 4)
- 2 conferences, 2 call legs per conference (2 – 2)
- 2 conferences, 3 call legs per conference (2 – 3)
- 2 conferences, 4 call legs per conference (2 – 4)

With 500 MB of main memory and employing SPIN's *compression* technique[16], we were only able to exhaustively verify the first four models. Some details of these verification runs are shown in Table 3.

Model	*State-vector (bytes)*	*Nr. of stored states*	*Nr. of matched states*	*Nr. of transitions*	*Memory usage for states (Megabytes)*	*User time (seconds)*
1 – 2	92	475	671	1146	< 0.2	0.62
1 – 3	104	6307	12299	18606	0.205	1.21
1 – 4	116	80665	201596	282261	1.946	13.03
2 – 2	140	517409	1938190	2455609	11.059	118.90

Table 3: Verification results (exhaustive search)

The first column contains the model being used. The number before the dash indicates the number of conferences, the number behind the dash indicates the number of call legs per conference. The second column contains the size of the state-vector (i.e., the number of bytes required to store one system state). One system state corresponds to the concatenation of the state descriptors for all variables, channels and processes. Column three contains the number of stored states, i.e. the total number of different states that has been stored in main memory. The fourth column contains the number of matched states. These states, which have been traversed during the verification run, are not stored in main memory, since they are identical to a state that is already in main memory (every reachable state is only stored once in memory). Column five contains the number of transitions, that is, the total number of stored and matched states (this column thus contains the sum of the two previous ones). The sixth column contains the total memory that was needed to store the states, after compression. This is indicated in Megabytes. In this thesis, we use the term Megabyte to indicate a unit of 1,000,000 bytes. The term MB is used to denote units of 2^{20} bytes. The last column contains the user time that was needed to perform an exhaustive verification on the Sun UltraSparc-IIi.

Note the alarming increase in the number of stored and matched states when an additional call leg is being added. Adding a conference is even more alarming. The number of stored and matched states for the various models is plotted in the stacked column graph in Figure 9 (on a linear scale) and once again in Figure 10 (on a logarithmic scale). Studying these graphs, it becomes clear that our verification problem scales exponentially with the number of call legs and conferences involved. We are thus facing the state-space explosion problem.

To show why we are dealing with exponential increase here, consider the following situation[17]. Assume that there are (in total) *n* call legs in our model. At a given moment in time, they might all be in use, i.e., there is a party associated with every call leg. The application can now choose to attach all the media associated with the call legs. Since there are *n* different call legs, there are *n!* different orders in which the application can

[16] For the runs shown here, compression was not necessary, since the memory requirements were small enough to store all information in main memory. These runs have been conducted together with other runs, of which the results are shown later. In order to be able to compare them, we chose to use compression for the runs shown here, also.
[17] This is not intended as a formal proof, but mere a reasoning that explains why the increase is exponential instead of polynomial.

choose to issue the `attachMedia()` API calls. In practice, every call leg will be associated with a different party, and thus a different address, so that the order, in which the call legs are attached, matters.

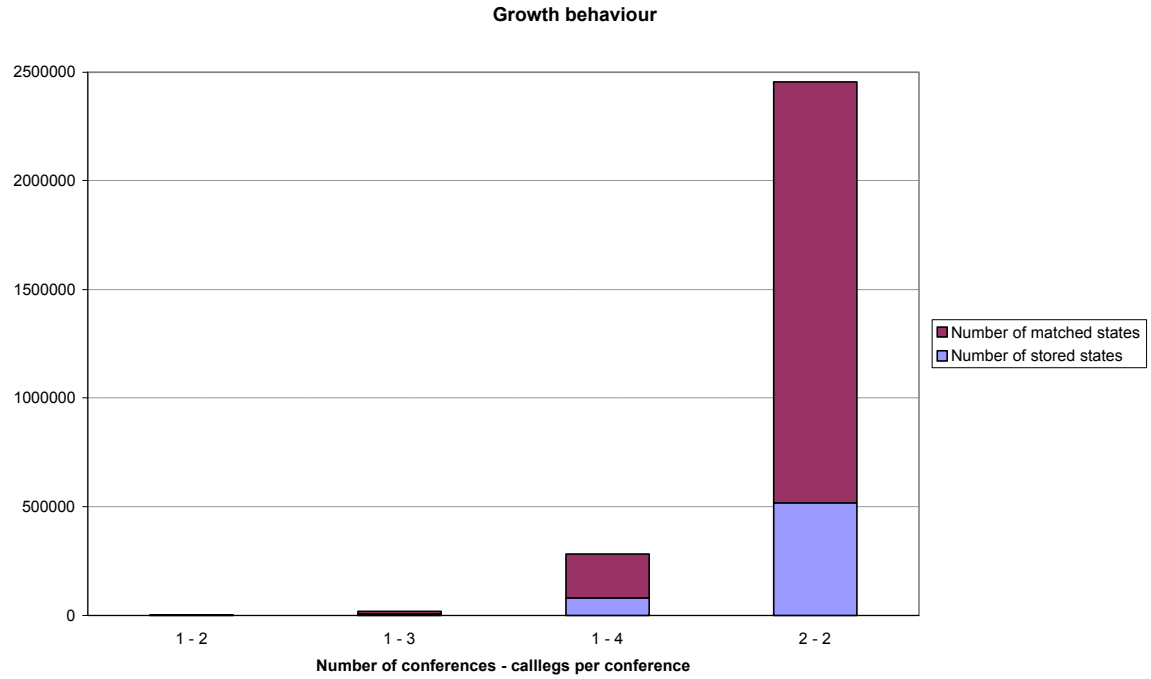

Figure 9: A comparison of the total number of states in various models (linear scale)

Since each call leg maintains its own state information, there are at least *O(n!)* different states that will be stored in memory. Since the verifier takes into account *all* possible executions, it will also consider the execution (fragment) mentioned above, so the total number of states that is to be stored is at least in the order of *n!*. Since for $n \geq 1$, $n! \geq 2^{n-1}$ and we are thus dealing with an exponential increase. The reason for the exponential behaviour is thus to be found in the non-determinism in the models.

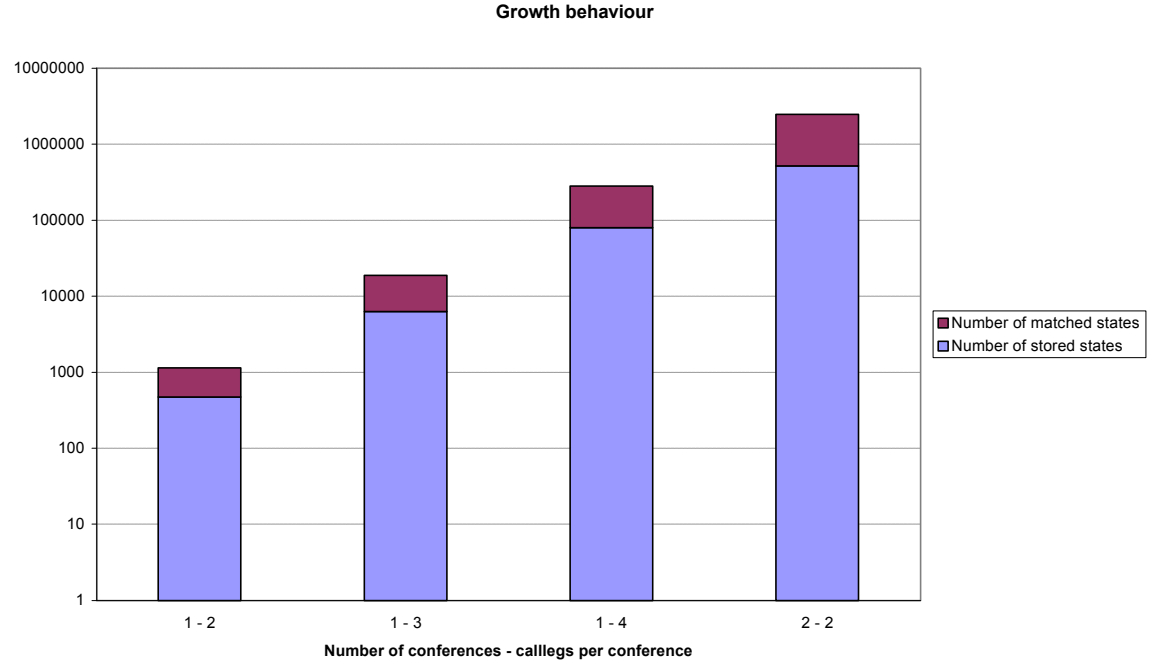

Figure 10: A comparison of the total number of states in various models (logarithmic scale)

Although we were not able to verify the models with two conferences and three or four call legs each in an exhaustive way, the partial verification results are shown in Figure 11 and Figure 12. At the time the

verification was terminated due to the fact that it exhausted its available memory, the verifier had not found errors in the model yet.

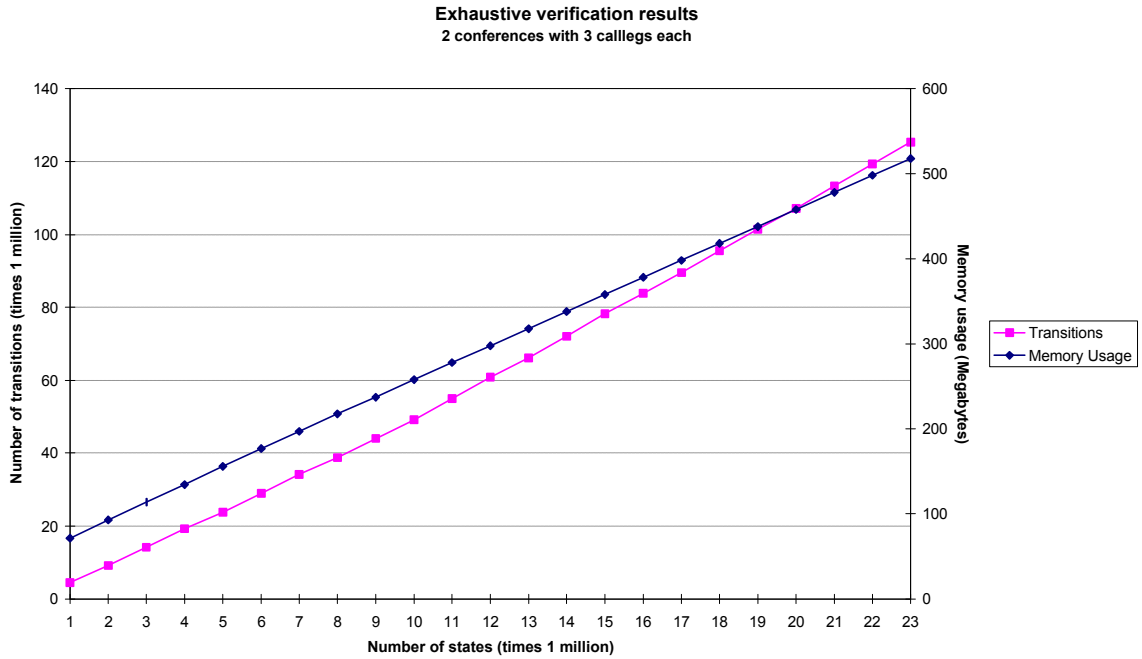

Figure 11: Verification results for exhaustive search of model 2 – 3 (search not completed)

In these two graphs, the memory requirements and the number of transitions are plotted. The number of transitions is plotted on the left axis, the memory usage is plotted on the right axis. Both verification runs were conducted with an upper memory limit of 500 MB (= 524,288,000 bytes). Note that the unit on the right axis is Megabytes instead of MB.

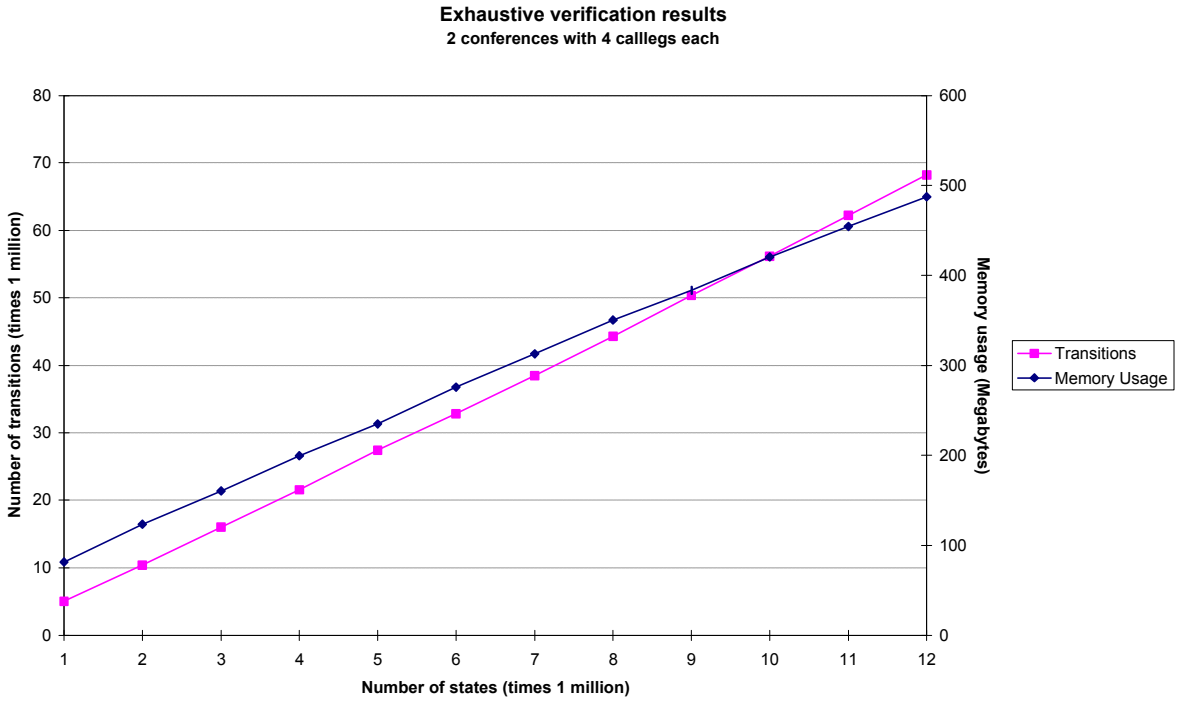

Figure 12: Verification results for exhaustive search of model 2 – 4 (search not completed)

Neither of the two searches could be completed within this memory limit. The graph thus ends on the last multiple of 1 million states for which SPIN printed the memory usage and the number of transitions, before

running out of memory. The size of the full state-space cannot be determined with these graphs (but we will come back to that later in section 4.3.2.3.

In Table 4, some more details about the last two verification runs are grouped together.

Model	*State-vector (bytes)*	*Average nr. of transitions per million states*	*Average memory usage per million states (Megabytes)*	*Compression ratio (%)*
2 – 3	156	5,491,954	20.309	87.47
2 – 4	180	5,745,435	36.902	80.25

Table 4: Statistics on exhaustive search for models 2 – 3 and 2 – 4

The first column contains the model involved, using the same notation as in Table 3. Column two gives the sizes of the state-vectors. Column three contains the average number of transitions that takes place with every million states that are stored in memory. The average number of matched states is thus 1 million less. The average memory usage, in Megabytes, is shown in column four. The last column contains the compression ratio, in percent.

From this table it becomes clear that using SPIN's built in compression technique (by providing the compiler with the parameter `-DCOLLAPSE`) severely reduces the memory that is needed to store 1 million states in memory. SPIN managed to reduce memory requirements with 87.47 percent, to 20,309,000 bytes per million states in the case of 3 call legs per conference. Without any compression, 162,083,000 bytes (154.6 MB) would have been needed to store 1 million states.

In the case of four call legs, the compression algorithm performed less well, but still managed to reduce the memory requirements to about 20 percent. Without compression, 178.2 MB would have been needed to store 1 million states. The penalty one has to pay for this reduction in memory usage is, of course, a prolonged execution time.

4.3.2.3 Approximate verification

Since exhaustive verification of the models with two conferences and three or four call legs was not possible with 500 MB of main memory, we used SPIN's bitstate hashing technique to perform an approximate search (also called *supertrace*). With the technique we used, two hash-functions are performed on the state vector of a reachable state. The outcome of the hash-functions, two numbers, are used as indices in a bit-array. The two bits addressed in this way are set to 1 and together represent this state. If *both* bits are already set to 1, a *hash collision* occurs. In that case, the state is matched on an already stored state. This is possibly incorrect, since it is not clear whether the states were the same and thus produced the same hash values, or the states were different but produced the same hash values by coincidence. As a consequence, all transitions originating from this state are no longer taken into account since, due to the depth first search, this already has been done. It can thus occur that parts of the state-space are not traversed, since they are incorrectly matched on an existing state. That is why the bitstate hashing technique is an approximation. Once the search is completed, SPIN *guesses* the coverage, i.e., what part of the state-space it has traversed. Regrettably, this is no more than a guess, as has been illustrated in [Ruys01] where a case was shown where SPIN predicted to have seen about 99.9 percent of the state-space, whereas in fact less than one percent was traversed. SPIN only predicts the coverage if the *hash factor* is sufficiently high. During our verification runs, we noted that a hash factor of about twelve was sufficient for SPIN to predict coverage at 98 percent, and as far as we have seen, this is the lowest prediction that SPIN makes.

The hash factor is defined as

$$Hf = \frac{m}{N}$$

where m is the size of the hash array in bits and N equals the number of reached states.

To get a better approximation, the size of the hash-array has to be increased. In SPIN, the size of the hash-array is always a power of two.

For models 2 – 3 and 2 – 4, a supertrace with a hash-array of 128 MB has been conducted. The results of these supertraces are summarised in Table 5.

Model	*Nr. of stored states*	*Nr. of matched states*	*Nr. of transitions*	*Equivalent memory usage (Megabytes)*	*Hash factor*	*User time (hh:mm:ss)*
2 – 3	$5.46042 \cdot 10^7$	$2.73124 \cdot 10^8$	$3.27728 \cdot 10^8$	9173.5	9.83205	02:45:47.44
2 – 4	$3.26875 \cdot 10^8$	$1.71943 \cdot 10^9$	$2.04631 \cdot 10^9$	62760.024	1.64243	18:30:27.08

Table 5: Supertrace verification results with a hash-array of 128 MB

The first column in this table describes the model that was used, in the standard form. Column two shows the number of states that have been stored in memory. Column three contains the number of matched states. Column four contains the number of transitions, i.e., the sum of the two previous columns. Column five contains the equivalent memory usage for states, that is: the product of the total number of states and the size of the state-vector plus overhead, and thus equals the minimum amount of memory that is needed to store all states uncompressed. Column six contains the hash factor and the last column contains the total user-time that was needed to 'complete' the search.

Figure 13 shows the relation between the number of stored states and the total number of transitions in the model with three call legs. On average, more than 6 million transitions took place for each million states that has been stored.

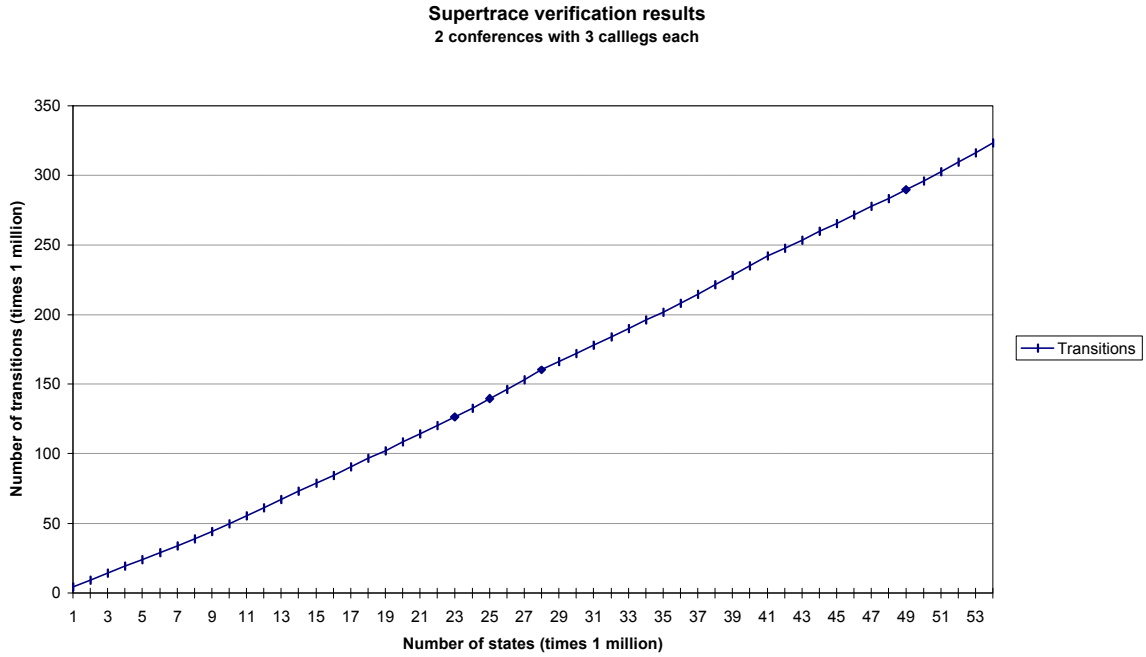

Figure 13: Supertrace verification results for the model with three call legs, with a hashtable of 128 MB

The hash factor of 9.83205 for the model with three call legs is too low for SPIN to give a coverage estimate. But even if we assume that this search has a coverage of 100 percent we would (using the compression ratios from Table 4) need 1,149,439,550 bytes (1096.2 MB) to store the states only. Additionally, SPIN allocates memory for a hashtable to rapidly find the location of the stored states. The larger the hashtable, the lower the number of hash conflicts, and the higher the execution speed. When using a hashtable of 256 MB, we would need around 1352 MB of main memory to perform an exhaustive verification, provided that the coverage *is* 100 percent (which it is most likely not) and that the same compression ratio is maintained. We did not have access to a machine with enough main memory to validate this theory.

In Figure 14, the relation between the number of stored states and the number of transitions is shown for the model with two conferences with 4 call legs each. On average, a little over 6.26 million transitions took place for each million stored states.

The hash factor of 1.64243 is once again too low for SPIN to predict coverage. To say something about the minimum memory requirements needed, we will once again assume that the supertrace has a coverage of 100

percent. In that case, the total amount of memory needed to store all states uncompressed, is 62,760.024 Megabytes. Using compression, and assuming the compression ratio shown in Table 4, a total of 12,395,104,740 bytes (11.54 GB) are needed to store the states in compressed form. Note that this is a minimum requirement, needed to store the states only. The actual memory that is needed for an exhaustive verification can only be determined if the coverage of the supertrace is known.

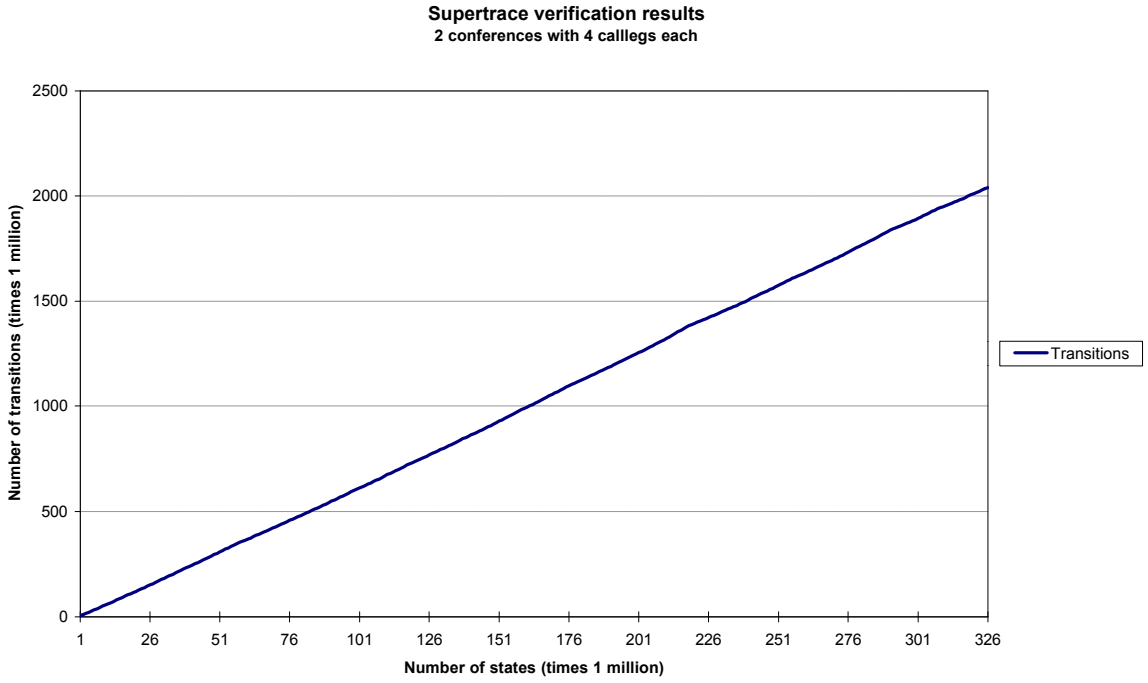

Figure 14: Supertrace verification results for the model with four call legs, with a hashtable of 128 MB

There is one other point to note regarding this supertrace: During the 18 and a half hour long verification run, we saw a slow increase in the search depth. Details are summarised in Table 6. Note that there are gaps between the intervals. This is due to the fact that SPIN only reports its progress every million states, so it is not exactly known when the search depth increased. The table thus has to be interpreted as follows: Somewhere between 62 and 63 million stored states, the depth reached increased from 153 to 154.

Number of stored states (in millions)	*Search depth*
1 - 62	153
63 - 308	154
309 - 311	155
312 - 326	156

Table 6: Search depths

SPIN uses a (nested) depth first search strategy to traverse the state space [HPY96]. Apparently, either our model is ordered in such a way that the search algorithm comes across the 'long' execution sequences at a later point in time, or a hash collision in the early phase of the search process prevented a long execution sequence to be examined. Since the state-space that was examined in the exhaustive search was so small when compared to the number of states in this supertrace, the only way to find out which one is the case (apart from using computers with more memory) is to repeat the supertrace with different hash functions. Due to the severe run-time requirements involved, such experiments have not been conducted.

To improve coverage in supertraces the size of the hash-array has to be increased, or one has to use another hash function. SPIN comes with 32 different hash functions, and the best coverage – when fixing the size of the hash-array – is obtained by performing a supertrace with all those 32 hash-functions (in fact, 32 different supertraces are conducted). Due to the severe run-time requirements, such extensive experiments have not been conducted. In our earlier experiments it became clear that doubling the size of the hash-array increased the runtime by about one order of magnitude, although we do not have sufficient details on that, and once the

hash table is large enough, this no longer holds (this also implies that the coverage is expected to be close to 100 percent).

No further verification experiments with this model have been conducted. Section 4.5 contains several possible tips to improve the performance, so that this model, or more important: this part of the Parlay API, can be verified with less memory or within shorter time.

4.4 Problems encountered with SPIN

From the previous section it became clear that one of the major problems we had to cope with is the state-space explosion problem. This was, however, not the only problem. We came across some strange things, which we believe are errors in SPIN. We will discuss them in short below.

4.4.1 Atomic sequence followed by do-loop

As it turns out, SPIN seems to have a problem with atomic sequences that are immediately followed by a do-loop (such a construct is not present in the current version of our model, but it was in one of the earlier versions). During our modelling efforts, we came across the following cryptic error message while performing a syntax check on a reasonable-sized model using SPIN and XSPIN version 3.4.7 with Tcl/Tk 8.3:

```
spin: unexpected: seqno=6 ntyp=46 (5 ntyp=275)
```

Since the error message does not include a line number and does not stop you from simulating or verifying the model (which a real syntax error should do) it was rather hard to pinpoint the exact location of the error. The error message could eventually be reproduced with the following trivial Promela model:

```
active proctype A()
{
  atomic {
    run B();
    run B()
  }

  do
  ::  skip
  od
}

proctype B()
{
  skip
}
```

It turns out that the error message does not occur when a dummy `skip` statement is inserted between the closing brace of the atomic sequence in process type `A` and the `do`-keyword. However, there is no reason to think of why an atomic sequence could not be followed immediately by a do-loop.

This error has been reported to Gerard Holzmann, the creator and maintainer of SPIN. He confirmed that the error message was spurious and the message is said to be removed in the next release of SPIN.

4.4.2 Erroneous message sequence charts

XSPIN's Help window advises us to run a syntax check on a model first and then to debug the model further by running some random simulations, before it is verified. While performing a random simulation, XSPIN draws a message sequence chart, showing the process interleaving and message passing. For our model, the advice to run random simulations before verification turned out to be useless, since the message sequence charts contained errors.

It looks like SPIN's random simulator does not work particularly well with models containing atomic sequences. During several of our step-by-step random simulation runs, it turned out that atomicity was lost,

while it should not have been. As has been discussed earlier, atomicity can be lost only when a statement in an atomic sequence blocks. However, in an earlier version of our model, we came across an atomic sequence that apparently blocked on an `assert` statement, since execution did not continue inside the atomic sequence, but transferred to a different process. This is an error, since assert statements are always executable by definition and there was thus no reason for the statement to block.

The same Help window advises us to run a guided simulation to examine error trails found by the verifier. XSPIN displays some strange behaviour here, since synchronous send and receive statements seem to change into asynchronous statements. When performing a step-by-step guided simulation, the following occurs. If, somewhere along an execution path, in a given process step *n*, a synchronous send occurs, the message is first stored in a buffer, to be received by the receiver one step later (in step *n + 1*). This is also an error, since the semantics of the synchronous send require that another active process can perform the matching synchronous receive immediately (which would mean 'in the same process step'). The same numbering scheme seems to be used in verification runs, since the search depths of all reported errors by the verifier matched the depths displayed in the message sequence charts. This numbering scheme is rather contra-intuitive (we are talking about a *rendezvous* here!) and is certainly strange compared to the numbering scheme used in random simulations, where synchronous send and receive do take place in the *same* process step.

There is another error regarding the guided simulation, which is not so much an error in SPIN but possibly in the XSPIN implementation on Windows. Although we were not able to exactly reproduce the error, it sometimes seemed that the verifier was not able to overwrite an existing error trail (XSPIN always uses the same file to store an error trail), so that, when running a guided simulation, we were presented with the wrong error trail. Note that the command line version of SPIN gives a warning if the error trail is older than the model, but we have not been able to find that message in XSPIN (but perhaps it is there). Without such a message, only by thoroughly examining the error trail one can be sure about looking at the right trail. It is very frustrating to debug a model against an error that no longer exists…

The last disadvantage of the message sequence charts is that they are not always readable because two or more processes overlap. Zooming in on the diagram usually does not work since the maximum width of the diagram is determined by the graphical interface and it is our experience that this maximum is usually to small to comfortably view the diagram. When using large zoom-factors, parts of the message sequence chart are drawn outside areas that can be made visible on the screen.

4.4.3 Erroneous behaviour without safe statement merging

When generating a verifier, SPIN by default applies a set of optimisations such as dead variables elimination and statement merging. These are claimed to have no effect on the verification result, only on the run-time or memory usage. This is, however, not true.

Below are the verification results for an exhaustive verification of our model with one conference and two call legs (as generated out of the *m4* source included in Appendix A). SPIN is instructed to continue with the verification once an error is found. When the optimisation is applied, i.e. in the default condition, SPIN produces the following output[18]:

```
State-vector 88 byte, depth reached 47, errors: 0
     475 states, stored
     671 states, matched
    1146 transitions (= stored+matched)
    1847 atomic steps
```

SPIN was thus not able to find any errors in our model. But when the optimisation 'statement merging' is disabled by providing SPIN with the `-o3` parameter while generating the verifier source, the following output is returned:

```
State-vector 88 byte, depth reached 117, errors: 19
     851 states, stored
```

[18] The output shown here is for Windows NT, but the Unix version only differs in the size of the state vector it occupies: 4 extra bytes, which can be explained by the usage of different compilers and hardware platforms.

```
  935 states, matched
 1786 transitions (= stored+matched)
 7232 atomic steps
```

Apparently SPIN is now able to find no less then 19 errors in our model. The increase in the number of states, transitions and atomic steps can be explained, since SPIN by default merges several so-called safe statements into one statement. In effect, in the first run, SPIN first generated a different model by merging some statements together and performed a verification on that.

It is claimed that this 'minimised model' encompasses the whole behaviour of the original model. However, this is clearly not the case, regarding the number of errors. Or is it? When studying the error trails generated by the verifier, the suspicion arose that the model checker erroneously rejected the model, since the error trails displayed some strange behaviour. Take for example the following fragment of code, corresponding to the event of a party leaving a conference in the process `NetworkApplication`:

```
atomic {
  network.conference[0].used &&
  network.conference[0].callLeg[0].used
  ->  /* remove the call leg */
      network.conference[0].callLeg[0].used = false;
      network.conference[0].callLeg[0].attached = false;

      /* if requested (via leaveMonitorReq): inform the client */
      if
      ::  network.conference[0].monitorLeave
          ->  Parlay!leaveMonitorRes(0, 0)
      ::  else
          ->  skip
      fi
}
```

In one of the error trails, the following situation occurred. The last statement executed in the process `NetworkApplication` was the line `Parlay!leaveMonitorRes(0, 0)` in the if-clause above. Upon execution of this statement, the atomicity token was passed to the process `ClientApplication`, since the matching receive took place, which was also in an atomic sequence. The code in that atomic sequence ran to completion. So far, nothing strange has happened. However, after that atomic sequence had run to completion, a new atomic sequence was entered in process `ClientApplication`, which subsequently required synchronisation with the `NetworkApplication`. However, this synchronisation was, according to SPIN, not possible. Apparently, `NetworkApplication` was still located inside the 'atomic' sequence above (which was by now no longer atomic), although there were no remaining statements to execute.

Thanks to dr. ir. Theo Ruys of the Formal Methods and Tools group of Twente University, the Netherlands, the error could be reproduced with a simplified model, based on one of the earlier models. It turns out that SPIN, when provided with the `-o3` flag shows erroneous behaviour. In short, the following is the case: After having send the `Parlay!leaveMonitorRes(0, 0)` operation above, SPIN is not yet ready to synchronise on another Parlay call. This seems strange, since the atomic sequence should have ran to completion. However, SPIN inserts an 'implicit goto' with as destination the beginning of the do-loop in which the code fragment was included, just behind the before mentioned operation. In `-o3` mode, this dummy instruction (it is not part of the model, nor the Promela semantics, but a means introduced by SPIN to implement the Promela semantics) is not removed (it should have been, though) and thus the code fragment above is *not* finished after executing the Parlay call.

The problem is analysed in more detail, in Appendix C, using a small model which is loosely based on the model we have been using for our verification purposes. It differs from the model made by Theo Ruys in that it only uses two processes instead of four.

4.4.4 Memory usage statistics

After each verification run, SPIN reports on memory used for the verification, for example

```
Stats on memory usage (in Megabytes):
10.325  equivalent memory usage for states
        (stored*(State-vector + overhead))
1.946   actual memory usage for states (compression: 18.84%)
        State-vector as stored = 12 byte + 12 byte overhead
134.218 memory used for hash-table (-w25)
0.024   memory used for DFS stack (-m1000)
136.266 total actual memory usage
```

There are two things to remark about this listing. First of all, the unit 'Megabytes' corresponds to 1,000,000 bytes, instead of the usual 2^{20} = 1,048,576 bytes. This is somewhat strange, since, while compiling the verifier, an upper memory limit has to be provided. This value is to be given in 'real' Megabytes (MB). That is why it can occur that SPIN reports to have used 536 Megabytes of main memory, while the upper memory limit was set to 512 MB.

Second, SPIN reports a compression ratio. In the example above, this equals 18.84 percent. This is not, as the word 'compression' suggests, the rate of compression, but the remaining size after compression. In the example above, the memory needed to store the states has thus been reduced to 18.84 percent of the original size of, in this case, 10.325 Megabytes. The compression rate thus equals 81.16 percent.

If the verification requires more main memory than set in the upper memory limit or than physically available in the system (whichever one comes first), the verifier *pan* runs out of memory, e.g. displaying the following error message (the upper memory limit was set to 500 MB):

```
pan: out of memory
        5.24275e+08 bytes used
        102404 bytes more needed
        5.24288e+08 bytes limit (2^MEMCNT)
```

Pan thus happily reports the extra amount of memory that is needed to complete the verification. At least, that is what one expects when seeing a message like this. To test the validity of this claim, a verification run was performed on an almost identical machine (same CPU and operating system, only now equipped with a main memory of 1 GB instead of 512 MB), allowing SPIN to use the double amount of memory. Again, the verifier ran out of memory and again, 102,404 bytes more were needed according to SPIN.

In all our verification runs, with this model and with earlier versions, whenever *pan* ran out of memory, it reported an additional memory need of 102,404 bytes, so we can safely conclude that the amount of extra memory needed, as reported by *pan*, is not correct.

4.5 Improving the model

Theo Ruys describes several tips on how to reduce the complexity of SPIN models in his Ph. D. thesis on effective model checking [Ruys01]. Several of these tips have been incorporated in our model, since an earlier version of our model showed an even worse behaviour regarding the memory and time requirements for verification. The reduction achieved was quite significant: the state-vector in the case of two conference calls with four call legs each has been reduced from 220 bytes to 180 bytes. Before the model can be verified within reasonable time and memory requirements, a lot still has to be done, however. In this subsection we will mention several ways to, perhaps, improve the model.

In our model, the main source of complexity stems from the size of the state-vector, closely followed by the large number of transitions. Our first point of concern should thus be the reduction of the size of the state-vector. Theo Ruys remarks that SPIN does not use bit-arrays in an efficient way. Although our model does not contain arrays of bits, his solution to this problem, i.e., implementing the array as bitvector by grouping several bits together in a byte, short or int, could prove useful.

Instead of arrays of bits, our model contains many booleans. To be precise, we store seven booleans per conference (`used`, `reserved`, `joinAllowed` and `monitorLeave` in structure `network`, `used` and `monitorLeave` in structure `client` and `leaveMonitorReqIssued` in structure `status` in process `ClientApplication`). These could be grouped together using a 1-byte bit-vector. An additional six booleans are stored per call leg, which could also be grouped together in a bit-vector. To be precise, we could store all our booleans in a bit-vector with a size of CONFS(7 + 6 CALLLEGS) where CONFS denotes the total number of conferences and CALLLEGS denotes the total number of call legs per conference. For 2 conferences, with 4 call legs each, this leads to a bit-vector the size of 62 bits, i.e. two 32 bits integers (8 bytes). We also need a few additional bits to encode the remaining variables (storing them as efficiently as possible). Perhaps we can thus obtain a smaller state-vector, which even can be compressed a little further, but only implementing this suggestion will tell.

However, there are some disadvantages to this approach, namely:

- If all variables are grouped into one vector, this must be declared global since more than one process requires access to them. The internal optimisations by SPIN are predicted to have less effect on global variables than on local variables. It is however an idea to group the variables per process or conference. There is also an advantage, in that global variables can be used in LTL formulae, whereas local variables cannot.
- The complexity of the model increases, since extra operations have to be added for working with the bit-vectors.
- It might be harder to generate a model for a different number of call legs, since the implementation of the state-vector may have to change from byte, to short, int, or arrays of the before mentioned data types to provide efficient storage.

Note also that the performance of SPIN's compression algorithms is likely to decrease once a more optimal implementation has been found, since this results in the same information in less bits, i.e. a higher information density.

Another possibility to reduce the size of the state-vector as stored, not as such, is to use a better compression algorithm. Although SPIN's compression is good, reducing the memory requirements to about 1 tenth of the normal usage, it might be possible to find a better algorithm. SPIN itself advises us to use a minimised automaton, which is said to improve the compression ratio even further. Instead of `-DCOLLAPSE`, one has to pass the parameter `-DMA=`*`xx`* to the C compiler when compiling the verifier source, where *`xx`* stands for the size of the state-vector. SPIN also warns us for the increased run-time, so this might not prove helpful in all situations.

An entirely different approach, is to find a better abstraction of the Parlay API. It appears that our implementation, with two interacting processes, is too much to ask SPIN to verify. Although it might not be possible to reduce the number of processes even further, it can be a good idea to get rid of the instantiations of the process type `ConferenceStarter`. This could e.g. be done by starting a conference immediately after its reservation. Although this is not the normal behaviour in the case of a real Parlay application, it might prove powerful enough to verify selected parts of the Parlay API. Using a different model checker, that encodes the state-vector in a more efficient way, might also help.

The last strategy, and perhaps the most promising one, that we describe here, is to employ symmetry reduction. Symmetry reduction is focussed on reducing the state space, as follows. Often, it can be proved that it is sufficient to verify only a small percentage of all the possible executions, in order to verify the system as a whole. Symmetry reduction mechanisms are based on this observation. By reasoning over the model, they determine which part of the state space has to be explored in order to verify the system as a whole. Currently, SPIN does not include techniques for symmetry reduction, although work in this area is being done (see e.g. [BDH00]). If one is able to prove that verifying part of the total state space is sufficient, and one is able to determine exactly what execution sequences have to be verified, one can use SPIN to verify (part of) the Parlay API with presumably much less memory.

Only after the state space that has to be traversed in verification runs has decreased to acceptable levels, it makes sense to further extend the model. Therefore, this thesis does not describe how the model can be extended.

Chapter 5

Integrity-related issues concerning the Parlay API

In the previous chapter we have shown, using a formal verification technique, how two different views on the network state can occur by not invoking the `leaveMonitorReq()` operation. Although such a difference is in principle not a problem when seen from an integrity point of view, it does become so when decisions are based on this incorrect information. The resulting API call is either not allowed, or contains invalid parameters. If such a situation occurs, the network application has to detect these errors, prevent them from harming network integrity and inform the application about what has happened.

In this chapter we will describe several other possible Parlay-related 'points of failure' regarding telecommunications network integrity. These problem cases are described in short. Where possible, a solution to the problem is presented. Those solutions should be seen as proposed changes to the Parlay API.

It is the opinion of the author that, in order to assess whether an operation poses threats to network integrity, the semantics of that operation must be described in detail and without contradictions. While studying the Parlay API, we came across several topics that were, in our opinion, described in too little detail to assess whether telecommunications network integrity is at stake. Also, several contradictions have been found in the Parlay API documents. In this chapter, we will mention those issues in short.

The information given here is purely based on our experience with the Parlay API and not the result of a formal verification process. In this chapter, the focus is no longer only on the Conference Call Control Service, but on the whole of call processing services delivered by the Parlay API.

The remainder of this chapter is divided into five sections. Section 5.1 describes integrity-related issues regarding the Generic Call Control Service. Section 5.2 describes issues related to the Multiparty Call Control Service. Section 5.3 describes issues related to the Multimedia Call Control Service and section 5.4 describes the CCCS related issues. Section 5.5 concludes this chapter with some integrity-related remarks that could not be placed in earlier sections.

5.1 Generic Call Control Service

In this section we describe the integrity-related issues concerning the Generic Call Control Service, that have been found while studying the API.

5.1.1 Routing without monitoring

When issuing a `callEventNotification()` request, one has to specify the conditions for which to monitor. Also, one has to indicate whether monitoring is to be done in notify mode or in interrupt mode. In the `routeReq()` operation, one also has to provide a parameter indicating which progress reports are requested. Now, the monitor mode can be supplied for each progress report (busy, no answer, et cetera). Apart from the values 'interrupt' and 'notify' a third option is now available: 'do not monitor'. Why this value has been included in the Parlay application is not clear to the author, since one is not required to explicitly specify for which events monitoring is not needed.

5.1.2 Refused call event notification requests

The operation `enableCallNotification()` in class `IpCallControlManager` enables applications to set criteria for the call events they are interested in, so that events can be sent to an application for further call processing. The field `eventCriteria` (of type `TpCallEventCriteria`) is used to specify in what event(s) the application is interested. This field contains an originating and terminating address range, the names of the events to monitor for, the call notification type and the monitor mode [Parl01c]. The call notification type can be either 'originating' or 'terminating', defining on whose behalf the event is triggered. In telephony, an event with originating call notification type would be triggered on the first switch that lies on the path from route to destination. With terminating call notification type, the event would be triggered on the last switch. The number plan describes the type of address being used, e.g. telephone number[19]. The monitor mode indicates whether the application is to be notified of this event only, while call processing continues, or whether call processing has to be suspended, awaiting further instructions from the application.

According to [Parl00a], an `enableCallNotification()` request is refused if some other application already requested notifications with criteria that overlap the specified criteria. Criteria are said to overlap if both the originating and terminating ranges overlap, the same number plan is used (e.g., both addresses are telephone numbers) and the same call notification type is used. Addresses can be specified in the Parlay API as formatted strings. The format of the string is prescribed by the number plan being used. For example, telephony addresses have to be specified in the Parlay API without the international access code, including the country code and excluding the leading zero of the area code [Parl01b].

In order to specify address ranges the same structure is used, but wildcards are allowed. The wildcard '*' may be used to match zero or more characters and '?' can be used to match exactly one character.

The use of wildcards is restricted, depending on the number plan being used. In general, wildcards are only allowed at either the beginning or the end of the string, not both. It is therefore not entirely clear whether the address range '*', to match all addresses belonging to the specified number plan, is allowed by the Parlay API, since one could argue that the wildcard occurs at both the beginning and the end of the address string. However, since the address string '*' has great practical value, e.g. when creating an unconditional call forwarding service to forward all calls, we will assume that the address string consisting of only the wildcard '*' is valid.

For telephony applications, an extra constraint applies: wildcards can only occur at the end of the address string. The address range '*1', to denote all telephone numbers with 1 as last digit, is invalid, whereas '31*', to denote all telephone numbers in the Netherlands, is valid.

One must be careful with using wildcards. If the first `enableCallNotification()` request ever issued specified E.164 (telephony) as number plan, the address range '*' as both originating and terminating address, monitor mode 'interrupt' and call notification type 'originating', the Parlay application is informed about *all* telephone calls in the operator's network, and all telephone calls are suspended until the application indicates how call processing has to be resumed. This poses a severe threat to network integrity, since the application probably cannot keep up with the pace of incoming call events, reducing network availability and thus integrity. Apart from this potential performance problem, another problem arises: all other invocations of the `enableCallNotification()` operation with number plan set to telephony and notification type 'originating' are refused by the Parlay network application, so it is not possible for third party service providers to introduce new telephony-services that require originating notification, using Parlay, on the same operator's network.

Although the above situation is not very likely to occur, protection against it must be provided. It is thus either the task of the Parlay API designers to define additional constraints on the allowed address ranges, or the network operator has to specify these constraints in the Service Level Agreement that has to be signed by the value-added service provider who wishes to exploit Parlay applications. In either case, the network operator has to safeguard the network against the misuse of these wildcards.

[19] Remember that the Parlay API has not only been designed for telephony applications, but for various types of communication. To specify telephone numbers and e.g. e-mail addresses, the same `TpAddress` structure is used. This contains a field of type `TpAddressPlan`, defining the actual number plan that is being used. Example number plans supported by the Parlay API are IP (Internet Protocol), multicast, unicast, E.164 (telephony), AESA (ATM End System Address), Uniform Resource Locators (Internet URLs) and SMTP (e-mail). In our discussion, we will restrict ourselves mostly to telephony applications.

Similar problems can also occur on a much smaller scale, as described below. Consider the situation where a call event notification request for all calls from *A* to *B* was issued (interrupt mode, originating notification). Such a request would be issued for example to implement call blocking for all calls from *A* to *B*. This is a valid Parlay application, but due to the way Parlay threats overlapping call notification requests it introduces integrity problems. Say *B* is going away for a while, but wishes to forward all his calls to a different number, in order to stay reachable. Suppose he wants to use the services provided by a Parlay application for this purpose. In order to realise the service, the Parlay application has to issue an `enableCallNotification()` request (originating notification, interrupt mode) with the originating range set to '*' and the destination range set to '*B*' (all calls to *B*). This is a valid request, but it is refused by the Parlay application due to the fact that both the originating and terminating ranges overlap with the already issued request, the same number plan is used, and the same call notification type is used.

When seen from the user's point of view, this is a severe integrity problem, since one person who wishes to block all outgoing calls to a certain user effectively prohibits that user from e.g. forwarding all his calls. This is a situation you cannot explain to people, since it is just as absurd as a travel agent denying you to sell a trip to Greece since the previous customer went to the United States. Both things are totally independent, but for some (to the user unknown) reason they apparently cannot exist together. Although nothing is wrong with the telecommunications network, this has to be considered as an integrity issue.

The problem with this whole issue is that it cannot be solved with the current Parlay API specification. A possible solution would be to issue a `callEventNotification()` request for calls originating from all numbers, excluding *A*, to *B*. However, for such a solution to succeed, one must know exactly why the request was refused, i.e. one must know that there is a call event notification active for all calls from *A* to *B*. The interface class `IpCallControlManager` does provide an operation `getCriteria()` that can be called by an application to request a list of active call event notifications, but this only reports the events of which the application is notified, i.e. only those call event notification requests that have been issued by the application itself. In the case that the call event notification for the call blocking service was issued by a different Parlay application (a likely scenario), there is thus no way to solve this problem with the current Parlay API.

In order to solve this problem, some changes to the Parlay API thus have to be made. A possible change is to include more detailed error information specifying the overlap between already active call event notifications, to be sent in the `P_GCCS_INVALID_CRITERIA` error message. In addition, a more flexible way to specify address ranges has to be introduced. Using wildcards, the smallest range (being not a single address) possible contains ten numbers (from 0 to 9). To specify e.g. the fact that an application has to be notified of a call attempt originating from a number in the range'31241234???', excluding the number 31241234567, one needs 26 strings to denote the numbers that are allowed, assuming that the digit on the place of the first question mark cannot be zero. So, to denote the two intervals from 31241234100 to 31241234566 and from 31241234568 to 31241234999, one needs 26 different call event notification requests. By using a more powerful way to express addresses, these problems can be circumvented. For example, in addition to the already existing wildcards, there seems to be no reason why the addresses cannot be specified also as lists, or by using a range operator that describes an upper and lower bound, or combinations thereof. All telephone numbers in the range '31241234???', excluding the number 31241234567, could then for example be denoted by the string '31241234100 – 31241234566; 31241234568 – 31241234999'. Such a notation does work well with numerical addresses, but perhaps not with alphanumerical addresses such as e-mail addresses.

Another possible change that can be made to the Parlay API, is to no longer refuse call event notification requests if they contain overlap with already existing requests. If a call event occurs that matches several call event notification requests, the application that corresponds with the most *narrowly defined* range of all matching requests should be informed. With the most narrowly defined range we mean the range (i.e., originating and terminating address range) that matches with the event, and also denotes the smallest interval of all matching notification requests. So, if a call from *A* to *B* occurs, and there is a call notification from *A* to *B* and a call notification from '*' to *B* (which would then be allowed), the application that issued the call notification request from *A* to *B* has to be informed, since this is the most narrowly defined range.

Such a situation would allow user *B* in the example above to forward his calls, regardless of the fact that user *A* blocks all outgoing calls to *B* (as it should be). However, such a change possibly introduces new integrity issues that have to be solved. For example, one has to specify exactly when one range is more narrowly defined than the other. Should the origination or the destination number be the determining factor? In other

words, is an event notification request from '3124123456?' to 31241234567 more narrowly defined than an event notification request from 31241234567 to '3124123456?' or not?

In addition, the use of wildcards must perhaps be more restricted. Should it be allowed for an application that is interested in calls originating from the numbers 31241234567 and 31241234568 to issue a call event notification request with '3124123456?' as originating address? If so, the application can be notified of events it is not really interested in. However, a different application can be interested in these events, but the events cannot be signalled to that application. Perhaps it would be a good suggestion to extend the Parlay API with a sort of query function, so that the Parlay framework can signal all interested applications of the event, querying which application really wants to process this call further (under the assumption that there can be only one application really interested in processing the call).

As shown in this subsection, Parlay's decision to deny event notification requests if they contain overlap with already existing notifications introduces some undesired behaviour. Several possible solutions have been mentioned, of which the last one (do not refuse) is the most powerful, but there is certainly a point of further research here, to assure that no other integrity issues arise. To perform this analysis is outside the scope of this thesis.

5.1.3 Undefined (combinations of) call events

The type `TpCallEventName` defines the names of the events upon which an application has to be notified of call events. This type is used in e.g. the `enableCallNotification()` API call. One of the possible values of the `TpCallEventName` is `P_EVENT_NAME_UNDEFINED`, suggesting that one can monitor for undefined events. This is of course not the case, since there is no way to recognise undefined events. The value `P_EVENT_NAME_UNDEFINED` is thus a dummy value and it is unclear to the author why it is included in the Parlay API.

It is possible to specify multiple events for which to monitor. However, the Parlay API does not describe which events can be combined together. For example, according to the Parlay API specification it is possible to combine both the 'address analysed' and the 'called party busy' events in interrupt mode, meaning that the application must indicate how to continue with call processing once a requested event occurs. However, since the application has limited ways to continue call processing, the above combination cannot occur in real life. The event 'address analysed' occurs as soon as the user has entered a complete telephone number. The application now has to indicate how call processing is to be resumed. The only appropriate call that can be made here is `routeReq()`, requesting the routing of the call to a destination. By doing so, the event 'called party busy' will not occur, at least: not this one. When issuing the `routeReq()`, one has to provide a new set of event criteria. So, the Parlay API allows combinations of values that in practice cannot occur. Although this is not really an integrity issue, it lacks the clarity that would be needed to implement the API in an integrity-preserving way.

Therefore it is recommended to change the Parlay API on this point as follows: combinations of events that necessarily occur after each other in time are only to be allowed in notify mode. Combinations of events that exclude each other (e.g. party busy and party answered) are to be allowed in both notify and interrupt mode.

Note that the monitor mode has to be specified for all events at the same time. It is not possible to e.g. request notification of the event 'address analysed' in notify mode and 'called party busy' in interrupt mode. The reason why the Parlay API designers imposed this restriction is unclear to the author.

5.1.4 Processing calls more than once

Once a Parlay client application is notified about a call event in interrupt mode, it has to give further instructions on how to continue with call processing. One possible way in which the client application can react is by issuing a `routeReq()`, to ask the network to route the call to a destination. It is then the task of the telecommunications network to route the call to the specified destination.

Consider the following situation, in which persons (and telecommunications network users) *A* and *B* both work for the same company. *A* is away on holiday, and *B* is given the responsibility to handle *A*'s unfinished business. To facilitate this, person *A* enables call forwarding for all his incoming telephone calls, and forwards them to the telephone number of user *B*, using a Parlay application. Since *B* is not always in the

office, but sometimes also working at home, *B* forwards all his incoming calls, using a Parlay application, to his home number, *C*.

As a consequence, it is likely that there are two call event notifications active:

- A call event notification for all calls made to *A* (* → *A*), issued by *A*'s call forwarding application
- A call event notification for all calls made to *B* (* → *B*), issued by *B*'s call forwarding application

Note that the Parlay applications that are used can be the same, but this need not be.

The Parlay API does not describe what should happen in the above situation when some other user (say *D*) calls *A*'s number. Of course, the call forwarding application is notified of the event of *D* calling *A*, since it matches its call event notification criteria. Being a call forwarding application, the application decides to route the call to *B*. The network thus delivers the call to *B*, were it remains unanswered since *B* is not in the office. From a user's point of view, this is erroneous behaviour. User *B* knew that he was going to work at home, but wanted to stay reachable for telephone calls, and thus invoked the call forwarding application to reroute the calls to his home number. However, the calls to *A*, that are to be forwarded to *B*, do not reach him on his home number.

Should not the call forwarding application used by user *A* trigger the call event notification of the call forwarding application that is used by user *B*, so that the call is processed by two Parlay applications and eventually reaches its (new) destination *C*? The Parlay API specification does not provide any detail on this subject. However, the Parlay API also does not provide any operations to pass already created objects[20] associated with the call to another application for further processing.

Out of an integrity point of view, it cannot be allowed for one call to be controlled by two different Parlay client applications at the same time. However, if an application only translates a dialled number into another number, and subsequently does not issue any other Parlay calls to control the call, there seems to be no reason why the call cannot be processed by another Parlay application that is interested in call attempts that are made to the translated number.

It would thus be possible to change the semantics of the Parlay API to facilitate the situation above. By changing the semantics of the `routeReq()` operation, so that it does not automatically create a new call leg object and by removing the by then already existing call object, there are no no-longer used objects any more and there is thus no reason why the call cannot be processed by another Parlay application. It would be even better to indicate the fact that the application will only provide a new destination address once it is informed about call events in the `enableCallNotification()` request, so that no objects have to be instantiated at all.

One has to take care, however, that such a number translation / call forwarding application does not enter an infinite loop if the destination address of the `routeReq()` is the same as the original destination address. So, a means to detect cycles must be included in the Parlay API implementation on the network side.

Other loops are also possible. Think e.g. of two call forwarding applications. The first application forwards all calls made to *A*, to *B*, and the second one forwards all calls that are made to *B*, to *A*. If a user now dials either of the numbers *A* or *B*, an infinite loop is entered (when not protected by a time-out mechanism), since the applications are continuously invoked to route the call. It is not possible to employ static loop detection, since the network does not know the destination address that is going to be provided by the client application until it is provided. How (and if) this looping problem can be solved in a way that does not harm the integrity of the network is not clear. Once the network detects that a call event is reported for the second time to the same application, it can decide to prevent this from occurring, herewith cutting the loop, but this invariantly leads to loss of service, since the requested service cannot be provided.

So, either when calls can only be processed by one application, or when they can be processed by more than one application, integrity issues arise.

The question thus arises if (and which) combinations of applications are to be allowed. As long as no application wishes to control the call (i.e. the applications only will provide the network with a new

[20] The `routeReq()` operation implicitly creates a new call leg object. Also, a call object is associated with every call.

destination address), arbitrary many can be combined, as long as no cycles are introduced. However, as soon as one application wishes to control the call, the call objects have to be instantiated. A second Parlay application that wishes to control the call is thus not allowed. But what about the situation that an application that wants to control the call (say from *A* to *B*) issues a `routeReq()` to a destination *C* and there is a number translation application interested in calls made to *C*. Should that application be invoked to provide a new destination address? This is a somewhat different question, since the controlling Parlay application can also be used for billing purposes, e.g. as part of a special service that makes it possible to call certain numbers at lower cost. If the number translation application provides a different number, which is perhaps more expensive to call (but one cannot know until the new destination is provided by that application), should that application be invoked? Or does the called party pay for the extra expenses, since he was the one that enabled the forwarding / translation function?

To summarise: Currently, the Parlay API does not allow one call to be processed by more than one application. However, this gives rise to unexpected situations, especially when seen from a users point of view, and thus, at least from a user's point of view, the network does not behave as expected and we are thus facing an integrity issue. It is subject to further research to find the most efficient solution for this problem.

5.2 Multiparty Call Control Service

The Multiparty Call Control Service inherits all operations of the Generic Call Control Service. However, the API specification does not describe the semantics of the inherited operations. The semantics of most operations, such as `routeReq()` and `release()` are inherently clear, but for example the semantics of `getCallInfoReq()` are not. This operation is used to collect information about the call, such as e.g. the time the call started, the time it was connected to the destination and the time the call ended. The reports can be requested once a party disconnects or when the call ends. In the Multiparty Call Control Service, a similar operation `getInfoReq()` is available on a call leg-basis. If both operations can be used together, it is possible to receive the information twice. It is also not clear when the responses to the `getCallInfoReq()` operation can be expected. The API specification does not mention whether the reports are sent upon call leg disconnection or only when the call ends (i.e. when there are no parties left, or the controlling party disconnects). This is another example of a situation in which the Parlay API description does not provide enough detail about the semantics of the operations.

5.3 Multimedia Call Control Service

The same remarks as in section 5.2, regarding the semantics of inherited operations, apply to the MultiMedia Call Control Service.

The operation `enableMediaNotification()` in class `IpMultiMediaCallControlManager` is in fact an enhanced version of the `enableCallNotification()` operation. It differs from the operation `enableCallNotification()` in that additional constraints on media channels can be specified. If monitoring with `enableMediaNotification()` is done in interrupt mode, the application has to explicitly attach the media channels. The Parlay API does not describe whether both the operations `enableMediaNotification()` and `enableCallNotification()` can be used together (that is, whether the rules for overlapping address ranges also apply here). If this is allowed, it can occur that a multimedia call attempt (matching the channel requirements) is reported, in notify mode. This is an undesired situation, since the Parlay API then requires two replies, but only one event has really occurred.

5.4 Conference Call Control Service

In the Conference Call Control Service, call legs are grouped together in subconferences. If a new party joins a conference, the corresponding call leg will be assigned to the default subconference object [Parl00a]. It can be moved to a different subconference before attaching the media. The Parlay API specification does however not define *which* subconference is the default subconference. In fact, the API contradicts itself on the question whether there is always a subconference. Upon creation of a conference, one has to specify the number of subconferences that need to be created automatically. The number of subconferences should be at least 1 [Parl00a, p. 64]. In the Conference Call Control Service, the conference call object is used to manage the subconferences. It also includes some methods (mostly inherited) to hide the subconferences from the applications that do not need it. On page 68 of the Generic Call Control Service Interfaces description

[Parl00a], it is noted that the conference call always contains one subconference. Although this sentence implies otherwise, the total number of subconferences can be larger than one.

The Parlay API specification also contains several class diagrams, showing the cardinality between classes. The class diagrams for the call processing services can be found in [Parl00e]. The class diagrams for the Conference Call Control interfaces show that one `IpConfCall` object relates to *zero* or more `IpSubConfCall` objects and that `IpAppConfCall` relates to *zero* or more `IpAppSubConfCall` objects. The accompanying description however, states that each conference call can have *one* or more subconference associated with it. The Parlay API description is thus not very clear on the subject of the minimum number of subconferences that is to be allowed.

More contradictions can be found in the description of the Conference Call Control Service. The semantics of the `partyJoined()` operation, which is called when a party joins a conference by dialing into the number that was provided during reservation, describe that the corresponding call leg will be assigned to the default subconference object and will be in detached state. The application may move the call leg to a different subconference before attaching the media [Parl00a]. A new party joining the conference by default does not have a speech connection with other parties in the conference. Such a connection is to be established using the `attachMedia()` API call. The operation `moveCallLeg()`, which serves to move a call leg from one subconference to another, receives three input parameters: the source and target subconference IDs and the session ID of the call leg itself. Another Parlay API document [Parl00e], describing the same API, however states on page 21 that detached call legs are not associated with any specific subconference, but are instead associated with the conference call itself. If this would be the case, however, the operation `moveCallLeg()` would not be applicable, or the conference call itself should be treated as a subconference.

The description of the Conference Call Control Service thus lacks clarity and flatly contradicts itself. It is not clear whether detached legs are to be associated with any subconference and if there always is a default subconference. And even if there is always one default subconference, it must be documented which subconference becomes the new default when removing the current default subconference, an operation supported by Parlay.

5.5 General remarks

We will conclude this chapter with some general remarks regarding the Parlay API that are related to integrity.

5.5.1 More on invalid API calls

During our verification attempts, we tried to determine whether invoking Parlay calls could give rise to dangerous situations, in which the integrity of the network was harmed. We limited ourselves to only a few API calls, and only invoked them when they were valid according to the Parlay API semantics.

If a Parlay client application issues API calls whose precondition is not satisfied, the call is – by definition – invalid. It is the Parlay API implementation in the operator's network that has to check whether the calls that are made, are valid. The network application cannot assume that API calls will not be invoked if their precondition is not satisfied, since no assumptions about the correct functioning of the client application can be made.

Another responsibility of the network operator is to thoroughly verify whether the parameters that are provided in the API calls are valid. For example, if the Parlay Gateway does not check if the IDs that are present in the various API calls correspond to call and call leg objects that have been created on behalf of the application issuing the call, a malicious client application might be able to control calls that have not been created on its behalf (especially if the Parlay Gateway uses the same interfaces to receive API calls from more than one client application) by just guessing the IDs. It needs no further explanation that such behaviour is highly undesired, and that protection against such situations is thus needed.

The last form of protection that has to be provided by the network operator (i.e., in the Parlay Gateway) and that is discussed here, is that the network operator needs to protect his network against excessive resource usage by client applications. To be able to do so, the Service Level Agreement must describe how many resources are available to the client application. Since Service Level Agreements are to be digitally signed by

the client application, they must be in the form of structured text, so that the application is able to determine whether enough resources are allowed for it to deliver the desired services to the application users.

5.5.2 Timeout

By using the Parlay API, control over call processing is transferred to third party applications. Therefore, the network depends on these applications in order to be able to e.g. route the calls to a destination address. If the application responds to the events that are reported to it in due time, nothing is wrong, but if the application does not reply in time, integrity is at stake.

If a situation occurs where the network times out, waiting for a reply, the network has to decide what to do with the call. Since it does not know how the application would have responded, the only logical action it can take, is to terminate the call attempt. Otherwise, resources remain allocated in the network for an unknown period of time. The termination of the call attempt however implies that the services that were offered by the client application cannot be delivered. This is a problem, and even an integrity problem, but it is a problem that cannot be solved by the network operator. In this case, the third party service provider is responsible for the fact that services cannot be delivered, and the network operator cannot do anything about it. So, this is an integrity problem, but it is not a problem regarding network integrity in the sense of the definition from section 3.5.

5.5.3 Lack of clarity

As has been mentioned earlier, it is the opinion of the author that, in order to assess whether an operation poses treats to network integrity, its behaviour has to be fully defined. It must be clear in exactly what situations the operation can be used and what effect the invocation of that operation has on the overall network and application state.

The Parlay API specification version 2.1 that has been used for this thesis does not fulfil all these requirements. In the previous sections some of the ‘unclear’ aspects of the Parlay API have been mentioned. In general, the Parlay API is described in the following four ways:

- UML Class diagrams, describing the relations between classes.
- UML Statecharts, describing some of the interface classes in more detail.
- UML Sequence diagrams, describing some typical event sequences and example applications.
- Text

It is the opinion of this author that this is in essence a good way to describe the API, although a little more care should have been taken by the Parlay API designers to keep the documentation consistent. Some inconsistencies between the text and the diagrams have been mentioned in earlier sections.

The sequence diagrams have proven to be very useful when studying the Parlay API for the first time. Upon closer study it turned out that the sequence diagrams where too much simplified to get a good view on how the Parlay API really works. Several events, such as the creation of new objects, have been omitted from the diagrams. Also, in the examples, almost all ‘intelligence’ is placed inside the interface classes, and not in an application that provides these interfaces to the outside world to allow communication. When studying the message flow in detail, one then finds out that the classes internally also have to communicate with each other, since the class upon which operations are called does not always know the state information that is needed to complete the operation. The sequence diagrams thus suggest that the interface classes themselves contain the intelligence needed to make decisions, although this need not be the case (implementation detail).

As an exercise while studying the Parlay API, the author redesigned some of the sequence diagrams to get a good view on how the Parlay API really works. All intelligence was put in a network application and a client application. Those applications provided the Parlay interfaces to the outside world for communication. The purpose of these interfaces was only to provide a convenient way to invoke operations. Every invocation of an operation was ‘forwarded’ to the application in question and it was this application that kept all state information and was responsible for taking the appropriate actions. This helped a great deal in understanding the Parlay API, but suffered from one major disadvantage: The resulting sequence diagrams were to large to include in any A4-sized document in a readable form. Therefore, the author recommends that a different

specification technique is used, that is both able to express all the constraints that are needed, and does not result in diagrams that cannot fit on one page. In the context of thesis, no further research has been conducted to find a suitable specification language for this purpose.

The UML statecharts are incomplete. To start with, not all classes come with a statechart. For the MultiMedia Call Control Service and Conference Call Control Service, no statecharts have been included in the Parlay API documentation. The statecharts accompanying the description of the Generic Call Control Service and MultiParty Call Control Service sometimes lack detail: preconditions that have to be satisfied before an operation can be invoked are sometimes omitted, and in some cases it is not clear from the diagrams which transition texts[21] belong to which transitions. Due to the way the arrows are drawn, the diagrams are often hardly readable. As an exercise, this author redesigned some statecharts using Rational Rose, and suffered from the same problem as above: the resulting statecharts became too large to include in any A4-sized document in a readable way. Again, a more space-efficient specification language could be used to prevent these problems from occurring.

The textual descriptions are often accurate, disregarding those cases in which they contradict themselves or any of the UML diagrams. It is however a disadvantage that the textual descriptions are spread out over multiple documents. For example, all types are documented in two separate documents [Parl01b, Parl01c], so that one has to consult these documents often to see e.g. what events can be monitored for, since the document describing the semantics of e.g. the `routeReq()` operation does not provide those details. A HTML-version of the complete documentation set, including hyperlinks to the definitions of all types, can greatly speed up the process of studying the API.

5.5.4 Integrity management by the Parlay framework

The Parlay framework includes ‘integrity’ management functions. However, the concept of integrity that is used in the Parlay Framework differs from the definition used in this thesis. The integrity management functions provided by the Parlay framework consist of load management, activity tests, service failure detection and heartbeat supervision [Parl00f, Parl00g].

Activity tests can be used to check whether the Parlay Framework and Parlay Services are operational. In turn, the Parlay Framework can request an activity test being carried out by the client application as well. Heartbeat supervision conceptually corresponds to two interacting processes, of which the first asks the second process whether it is alive. Upon reception of this message, the other process has to respond with ‘I am alive’ on regular time intervals, until heartbeat supervision is disabled. Heartbeat supervision can occur for both the framework and the client application.

These functions are thus primarily concerned with the proper operation of the Parlay Framework and the Parlay Services and with informing the client application of errors in the framework. The integrity management functions in the Parlay framework do not provide any protection whatsoever against putting the network in an unsafe situation, although they are needed for the proper operation of the Parlay Gateway.

5.5.5 Privacy

Although the topic of this subsection is not directly related with network integrity, it is strongly related with the usage of the Parlay API and therefore we will mention it here in short. As has been seen throughout this thesis, the Parlay API can be a powerful API to realise telecommunications features. It is this aspect of the API that requires additional attention, since misuse of the API must be prevented.

In more traditional telecommunications networks, such as the telephone network, features such as call forwarding are also accessible to the user. For example, the Dutch national telecommunications provider KPN provides call forwarding features that can be enabled by dialling the number ‘*21’[22]. By using this service, all calls that are made to the telephone extension on which the ‘*21’ number was dialled, are forwarded to a different number. The danger of using the Parlay API to realise the same features lies in the fact that such a close link between a telephone extension and a person enabling a service need not be present. One could think of e.g. a World Wide Web interface that allows users to manage their telecommunication

[21] By transition texts, we mean the textual description that is to accompany each arrow in UML statecharts, describing the precondition, event, action and operations that have to be invoked.
[22] Note that in this case the asterisk is not a wildcard, but a button on a telephone extension.

needs. A user could then access this interface to activate e.g. call forwarding features. However, enabling such a feature should be restricted to his own telephone numbers only.

If such a restriction is not enforced, it is very easy to use the Parlay API as a means to terrorise others. It would be no problem to forward e.g. all calls made to a pizza delivery to one of its competitors, effectively taking customers away from the pizza delivery and perhaps putting it out of business. It would also be easy to e.g. block all outgoing calls of somebody you do not like, or to 'stalk' people by monitoring their telephone traffic.

Enforcing such restrictions is not a task of the Parlay API designers, but rather a task for telecom operators or Parlay API service providers, closely supervised by the government or other regulatory organisations. One must not underestimate the impact that misuse of the Parlay API can have once it is operational on a nation-wide basis: It would be not so much of a problem to effectively block all telephone calls made to and from police stations, fire departments, hospitals and ambulance services, if no protection is provided against such situations.

What is interesting to note in this context is that the Parlay API specification contains a huge list of error codes that can be returned upon failure of an operation, including the codes `P_USER_NOT_SUBSCRIBED`, `P_APPLICATION_NOT_ACTIVATED` and `P_USER_PRIVACY`, all three indicating that the operation was not allowed by the telecommunications network user [Parl01b]. Apparently, the Parlay API designers are planning to include some protection against misuse of the API, since the presence of these error codes suggests that users can specify whether applications are allowed to request information or enable services regarding themselves.

Chapter 6

Conclusions & further research

The research that is presented in this thesis has been concerned with the question whether the Parlay API can pose threats to the integrity of the telecommunications network that it controls. Network integrity has been defined as the ability of the network to maintain a safe state, while performing within bounds that are acceptable to the network users, and delivering the functionality that is expected by the network users. To concretise the definition, a list of situations in which network integrity is said to be harmed, has been given (see section 3.5).

In this chapter, we sum up the most important results, conclusions and recommendations following our research, as well as the areas in which further research is necessary. For clarity, these are grouped into four categories: Integrity issues, recommendations regarding the Parlay API, formal verification issues and future work. Each category is the subject of one of the sections below.

6.1 Integrity issues

We have reported on several integrity issues. Most integrity issues do not classify as very serious issues that result in total or partial shutdown of the telecommunications network, but they nevertheless require attention when implementing and deploying the Parlay API. The following integrity issues have been found:

- It is easy for a Parlay Client Application to get a faulty vision on the real network state, which is dangerous since the application uses this state information to determine whether it can safely issue Parlay API calls. The situation can thus occur in which the client application expects to invoke a valid API call, but the Parlay Framework refuses the invocation. In the Conference Call Control Service, such a situation can occur since the client application is not automatically informed about the event that a party leaves a conference.
- In the current version of the Parlay API, call event notification requests made by applications are refused if they overlap with already issued requests (perhaps by other applications). This can lead to a situation where subscriber *A*, who wishes to block outgoing calls to the telephone number of user *B*, effectively prevents user *B* from forwarding his telephone calls. So, the fact that a feature is realised using the Parlay API can prevent another, conceptually different, feature from being realised at all, using the API.
- The current API specification only allows calls to be processed by one Parlay Client Application. There are situations in which this leads to services not being delivered, for example if user *A* forwards all calls to *B*, and *B* forwards all calls to *C* (using the Parlay API in both cases), calls that are made to *A* end up at *B*, instead of *C*.
- There is a serious privacy issue with the current version of the Parlay API, since there need not be a close link between the application issuing call event notification requests and the address ranges for which it is issued. A malicious Parlay application can thus take control over calls that are not supposed to be controlled by any application. For example, if no protection is provided against misuse, it is possible to impact the telephone (and other communications) traffic of people you do not like.

Although these integrity issues do not harm the telecommunications network by e.g. congesting trunks, they do impact the functionality that can be delivered by the network, and are therefore important issues to resolve, before the Parlay API can ever be used on a large scale.

6.2 Recommendations regarding the Parlay API

There are a few important changes that are recommended to make to the Parlay API and its specification documents. These are outlined below:

- If a call event notification request is refused due to overlap with a previously issued request, the Client Application is informed about this situation, but has no ways to find out what the exact overlap is. Therefore, the application is not able to circumvent the problem by issuing a request that does not overlap with already issued requests. This problem can easily be solved by either specifying the overlap in the error message or by adding an operation to the API to request the overlap, which is therefore recommended.
- The API denotes source and destination addresses using strings, allowing the usage of wildcards to describe ranges. The usage of this wildcards is limited to such an extent that it is not possible to denote e.g. a range of three telephone numbers with one address string. For telephony, the number of addresses in an address range is always a power of ten, i.e., 1, 10, 100, 1000, et cetera). Especially due to the fact that call event notifications are refused by the Parlay Gateway if they overlap with other requests, it is important to keep the address range that can be specified for the notification request as closely as possible to the real range the application is interested in. Therefore, it is recommended to allow more freedom in the address string representation, to facilitate the above.
- The Parlay API specification omits detail that is needed to fully assess whether operations poses threats to the integrity of the underlying telecommunications network. Especially the semantics of inherited operations are often not defined, and preconditions sometimes are unclear. The statecharts are often hard to read and sometimes they are missing. Since network operators must be convinced of the fact that the Parlay API poses no threats to their network's integrity, it is recommended that the API specification is improved, so that the semantics of the operations, and the preconditions that have to be satisfied before the operations can be invoked, are clear.

6.3 Formal verification issues

The following conclusions have to be drawn following the model checking efforts that have been conducted:

- Exhaustive verification of our model, containing seven Parlay operations, is possible in SPIN, but only if the number of call legs does not exceed two in a two-conference situation. For three or more call legs, the verifier is not able to exhaustively verify the model, using 512 MB of main memory.
- Supertraces that have been conducted on the two-conference model with three and four call legs showed that the state space contains millions of states. The hash factor that is obtained using a 128 MB hash-array, is too low for SPIN to predict the coverage. Assuming a coverage of 100 percent, we have shown that an exhaustive verification for the 4 call leg model requires at the very least several GB of main memory.
- The main source of complexity in our model stems from the number of non-deterministic choices, although they are already limited to a great extent by adding extra preconditions.
- Formal verification of a small part of the Parlay API, using our model, is hard due to the state-space explosion problem. Techniques that are aimed at the reduction of the state space or the number of bytes that is needed to store one state are needed. Example techniques that might have the desired effect are symmetry reduction or techniques like manually encoding the state variables in a highly efficient way.
- Although SPIN is a widely used model checker, it still contains several errors that do not make the life of the user any easier.

6.4 Future work

Several areas in which further research has to be carried out have been identified in this thesis. We mention them in short below:

- In the current Parlay API, it is not possible for a call to be processed by more than one Parlay application, although we have shown that this can be desirable. It is however not immediately clear how (and if) this can be implemented in a safe way, without introducing more threats to the integrity of the telecommunications network. This is therefore an area in which further research has to be conducted.
- We have shown that Parlay's rules for refusing event notification requests are too strict. Further research is necessary to identify whether it is possible to loosen these rules, so that the integrity issues that are caused by these rules no longer occur, but also no new issues are introduced.
- Formal verification of part of the Parlay API was not possible with our model, due to the state-space explosion problem. Further research is necessary to reduce the state-space.

Based on our experiences, it becomes clear that there is still a lot of work to be done before it is possible to convince network operators of the fact that the Parlay API does not pose threats to the integrity of their networks and before the API can be deployed in the real world in a safe way.

Appendix A

GNU m4 input for generating Promela models

Below is the full input file that has to be processed by the macroprocessor GNU *m4* in order to generate Promela models of the Parlay Conference Call Control Service. The line numbers are not part of the file.

To generate Promela models, use

```
m4 -DCONFS=X -DCALLLEGS=Y cccs.m4 > filename.pml
```

where *X* is the number of conferences and *Y* is the number of call legs per conference.

```
1  divert(-1)
2  /*****************************************************************************
3  * CODE TEMPLATE for SPIN/Promela model of Parlay Conference Call Control       *
4  *               Service. Needs GNU m4 as preprocessor.                        *
5  * ------------------------------------------------------------------------- *
6  *                                                                             *
7  * Filename:     cccs.m4                                                       *
8  *                                                                             *
9  * Author:       Peter Ebben                                                   *
10 *                                                                             *
11 * Date:         June 15, 2001                                                 *
12 *                                                                             *
13 * Copyright (c) 2001 by Lucent Technologies                                   *
14 *                                                                             *
15 * ------------------------------------------------------------------------- *
16 *                                                                             *
17 * Usage:                                                                      *
18 * ------                                                                      *
19 *                                                                             *
20 * This file needs to be preprocessed by the macro processor m4 before it can  *
21 * be used in simulations and/or verifications using SPIN or its graphical     *
22 * front end XSPIN. To do so, run                                              *
23 *                                                                             *
24 *   m4 -DCONFS=X -DCALLLEGS=Y cccs.m4 > filename.pml                          *
25 *                                                                             *
26 * where X equals the number of conferences and Y equals the number of calllegs *
27 * per conference, or run                                                      *
28 *                                                                             *
29 *   m4 cccs.m4 > filename.pml                                                 *
30 *                                                                             *
31 * to use the default values (2 conferences with 4 calllegs each)              *
32 *                                                                             *
33 \*****************************************************************************/
34
35 changecom(,)
36
37 /* set number of conferences */
38 ifdef(`CONFS', `define(`NR_OF_CONFERENCES', CONFS)',
39                `define(`NR_OF_CONFERENCES', 2)')
40
41 /* set number of calllegs */
42 ifdef(`CALLLEGS', `define(`NR_OF_CALLLEGS_PER_CONFERENCE', CALLLEGS)',
43                   `define(`NR_OF_CALLLEGS_PER_CONFERENCE', 4)')
44
45 /* macro definition for use in model */
46 define(`none', 255)
47
48 /* for-loop for text generation using m4 */
```

```
define(`m4_forloop',
       `pushdef(`$1', `$2')_forloop(`$1', `$2', `$3', `$4')popdef(`$1')')
define(`_forloop',
       `$4``'ifelse($1, `$3', ,
                    `define(`$1', incr($1))_forloop(`$1', `$2', `$3', `$4')')')

/* macro used in header generation */
define(`show', `format(`%4d                                                *
', $1)')

divert``'dnl
/*****************************************************************************\
* SPIN/Promela model of Parlay Conference Call Control Service                *
* --------------------------------------------------------------------------- *
*                                                                             *
* This Promela source file has been automatically generated out of the code   *
* template cccs.m4, using the GNU m4 preprocessor.                            *
*                                                                             *
* The settings used to generate this source file are listed below:            *
*                                                                             *
*   Maximum number of                                                         *
*    conference calls:         show(`NR_OF_CONFERENCES')``'dnl                *
*    calllegs per conference:  show(`NR_OF_CALLLEGS_PER_CONFERENCE')``'dnl    *
*                                                                             *
* --------------------------------------------------------------------------- *
* ---- Author: Peter Ebben ---- Copyright (c) 2001 by Lucent Technologies ---- *
* --------------------------------------------------------------------------- *
*                                                                             *
* Remarks:                                                                    *
* --------                                                                    *
*                                                                             *
* This Promela model contains the following Parlay API operations:            *
*                                                                             *
* - IpConfCallControlManager.reserveResources()                               *
* - IpConfCall.leaveMonitorReq()                                              *
* - IpCallLeg.attachMedia()                                                   *
* - IpCallLeg.detachMedia()                                                   *
* - IpAppConfCallControlManager.conferenceCreated()                           *
* - IpAppConfCall.partyJoined()                                               *
* - IpAppConfCall.leaveMonitorRes()                                           *
*                                                                             *
* The number of parameters and the type of parameters may differ from the     *
* Parlay API specification. Only the parameters that are needed to obtain     *
* results regarding network integrity have been included.                     *
*                                                                             *
\*****************************************************************************/

/*****************************************************************************\
* TYPE DECLARATIONS                                                           *
\*****************************************************************************/
mtype = {reserveResources, reserveResourcesReply, conferenceCreated,
         conferenceCreatedReply, partyJoined, partyJoinedReply,
         leaveMonitorReq, leaveMonitorRes, attachMedia, detachMedia,
         conferenceCreatedEvent}

typedef NetworkCallLegState {
  bool used;
  bool attached;
}

typedef NetworkConferenceState {
  bool used;
  bool reserved;
  bool joinAllowed;
  bool monitorLeave;
  NetworkCallLegState callLeg[NR_OF_CALLLEGS_PER_CONFERENCE]
}

typedef NetworkState {
  NetworkConferenceState conference[NR_OF_CONFERENCES]
}

typedef ClientCallLegState {
  bool used;
  bool attached
}

typedef ClientConferenceState {
  bool used;
```

```
  bool monitorLeave;
  ClientCallLegState callLeg[NR_OF_CALLLEGS_PER_CONFERENCE]
}

typedef ClientState {
  ClientConferenceState conference[NR_OF_CONFERENCES]
}

typedef CallLegDecisionStatus {
  bool attachMediaIssued;
  bool detachMediaIssued
}

typedef ConferenceDecisionStatus {
  bool leaveMonitorReqIssued;
  CallLegDecisionStatus callLeg[NR_OF_CALLLEGS_PER_CONFERENCE]
}

typedef DecisionStatus {
  ConferenceDecisionStatus conference[NR_OF_CONFERENCES];
  byte nrOfReservations
}

typedef ConferenceEventStatus {
  byte nrOfPartyJoins
}

typedef EventStatus {
  ConferenceEventStatus conference[NR_OF_CONFERENCES]
}


/*****************************************************************************\
* GLOBAL VARIABLES                                                            *
\*****************************************************************************/
chan Parlay = [0] of {mtype, byte, byte};             /* for Parlay API calls */

local byte nrOfErrors       /* the number of erroneous requests that are made */


/*****************************************************************************\
* PROCESS TYPE NetworkApplication                                             *
\*****************************************************************************/
active proctype NetworkApplication()
{
  NetworkState network;              /* state according to NetworkApplication */

  /* channel for internal use */
  chan startConference = [0] of {mtype, byte};

  /* information about already fired events, to keep things bounded... */
  EventStatus status;

  /* variables containing the next ConfCall/ConfCallLeg IDs */
  byte nextConfSessionID = 0;
  byte nextCallLegSessionID[NR_OF_CONFERENCES] = 0;

  /* temporary variables */
  bool joinAllowed;
  byte confSessionID;
  byte callLegSessionID;

  run ClientApplication();

  /* begin Parlay API call / event loop */

end:
  do
  ::  /* Parlay API call: reserveResources */
      atomic {
        Parlay?reserveResources(joinAllowed, _)
        ->  /* make sure an extra conference is allowed */
            assert(nextConfSessionID < NR_OF_CONFERENCES);

            /* mark conference as being reserved */
            network.conference[nextConfSessionID].reserved = true;
            network.conference[nextConfSessionID].joinAllowed = joinAllowed;

            /* make sure the conference is started in due time */
```

```
207             run ConferenceStarter(nextConfSessionID, startConference);
208
209             /* update ID for the next conference */
210             nextConfSessionID++;
211
212             /* inform the client that resources have been reserved */
213             Parlay!reserveResourcesReply(none, none)
214       }
215
216   ::  /* Parlay API call: leaveMonitorReq */
217       atomic {
218         Parlay?leaveMonitorReq(confSessionID, _)
219         ->  /* make sure the request is valid */
220             assert(confSessionID < NR_OF_CONFERENCES);
221             assert(network.conference[confSessionID].used);
222             network.conference[confSessionID].monitorLeave = true
223       }
224
225   ::  /* Parlay API call: attachMedia */
226       atomic {
227         Parlay?attachMedia(confSessionID, callLegSessionID)
228         ->  /* make sure the API call relates to an existing call leg */
229             assert(confSessionID < NR_OF_CONFERENCES &&
230                    callLegSessionID < NR_OF_CALLLEGS_PER_CONFERENCE);
231             if
232             ::  network.conference[confSessionID].used &&
233                 network.conference[confSessionID].callLeg[callLegSessionID].used
234                 ->  network.conference[confSessionID].callLeg[callLegSessionID].
235                     attached = true
236             ::  else
237                 ->  printf("MSC: Error!\n");
238                     nrOfErrors++
239             fi
240       }
241
242   ::  /* Parlay API call: detachMedia */
243       atomic {
244         Parlay?detachMedia(confSessionID, callLegSessionID)
245         ->  /* make sure the API call relates to an existing call leg */
246             assert(confSessionID < NR_OF_CONFERENCES &&
247                    callLegSessionID < NR_OF_CALLLEGS_PER_CONFERENCE);
248             if
249             ::  network.conference[confSessionID].used &&
250                   network.conference[confSessionID].callLeg[callLegSessionID].used
251                 ->  /* mark the leg as being detached */
252                     network.conference[confSessionID].callLeg[callLegSessionID].
253                       attached = false
254             ::  else
255                 ->  printf("MSC: Error!\n");
256                     nrOfErrors++
257             fi
258       }
259
260   ::  /* Network event: it is time to start a reserved conference */
261       atomic {
262         startConference?conferenceCreatedEvent(confSessionID)
263         ->  /* make sure the right conference is started */
264             assert(confSessionID < NR_OF_CONFERENCES);
265             assert(network.conference[confSessionID].reserved &&
266                    !network.conference[confSessionID].used);
267
268             /* update state information */
269             network.conference[confSessionID].used = true;
270
271             /* inform the client that the conference was started */
272             Parlay!conferenceCreated(confSessionID, none);
273
274             /* wait for the reply */
275             Parlay?conferenceCreatedReply(_, _)
276       }
277
278 m4_forloop(CONF_ID, 0, eval(NR_OF_CONFERENCES - 1), `
279   ::  /* Network event: a party joins conference CONF_ID */
280         atomic {
281           network.conference[CONF_ID].used &&
282             network.conference[CONF_ID].joinAllowed &&
283               status.conference[CONF_ID].nrOfPartyJoins < NR_OF_CALLLEGS_PER_CONFERENCE
284           ->  /* update event status information */
285               status.conference[CONF_ID].nrOfPartyJoins++;
```

```
286
287               /* update state */
288               callLegSessionID = nextCallLegSessionID[CONF_ID];
289               assert(callLegSessionID < NR_OF_CALLLEGS_PER_CONFERENCE);
290               nextCallLegSessionID[CONF_ID]++;
291               network.conference[CONF_ID].callLeg[callLegSessionID].used = true;
292               network.conference[CONF_ID].callLeg[callLegSessionID].attached = false;
293
294               /* make API call */
295               Parlay!partyJoined(CONF_ID, callLegSessionID)
296         }
297 ')
298
299 m4_forloop(CONF_ID, 0, eval(NR_OF_CONFERENCES - 1), `
300 m4_forloop(CALLLEG_ID, 0, eval(NR_OF_CALLLEGS_PER_CONFERENCE - 1), `
301   ::  /* Network event: party CALLLEG_ID leaves conference CONF_ID */
302       atomic {
303         network.conference[CONF_ID].used &&
304           network.conference[CONF_ID].callLeg[CALLLEG_ID].used
305         ->  /* remove the call leg */
306             network.conference[CONF_ID].callLeg[CALLLEG_ID].used = false;
307             network.conference[CONF_ID].callLeg[CALLLEG_ID].attached = false;
308
309             /* if requested (via leaveMonitorReq): inform the client */
310             if
311             ::  network.conference[CONF_ID].monitorLeave
312                 ->  Parlay!leaveMonitorRes(CONF_ID, CALLLEG_ID)
313             ::  else
314                 ->  skip
315             fi
316       }
317 ')')dnl
318
319   od
320 }
321
322
323 /*****************************************************************************\
324 * PROCESS TYPE ClientApplication                                              *
325 \*****************************************************************************/
326 proctype ClientApplication()
327 {
328   ClientState  client;               /*  state according to ClientApplication */
329
330   DecisionStatus status;              /* used to track already made decisions */
331
332   /* temporary variables */
333   bool joinAllowed;
334   byte confSessionID;
335   byte callLegSessionID;
336
337   /* process incoming decisions / Parlay calls */
338
339 end:
340   do
341   ::  /* Parlay API call: conferenceCreated */
342       atomic {
343         Parlay?conferenceCreated(confSessionID, none)
344         ->  /* make sure the API call is valid */
345             assert(confSessionID < NR_OF_CONFERENCES);
346             assert(!client.conference[confSessionID].used);
347
348             /* update client state */
349             client.conference[confSessionID].used = true;
350
351             /* send reply */
352             Parlay!conferenceCreatedReply(none, none)
353       }
354
355   ::  /* Parlay API call: partyJoined */
356       atomic {
357         Parlay?partyJoined(confSessionID, callLegSessionID)
358         ->  /* make sure the request is valid */
359             assert(confSessionID < NR_OF_CONFERENCES &&
360                    callLegSessionID < NR_OF_CALLLEGS_PER_CONFERENCE);
361             assert(client.conference[confSessionID].used);
362
363             /* create a call leg */
364             client.conference[confSessionID].callLeg[callLegSessionID].
```

```
365               used = true;
366             client.conference[confSessionID].callLeg[callLegSessionID].
367               attached = false;
368       }
369
370   ::  /* Parlay API call: leaveMonitorRes */
371       atomic {
372         Parlay?leaveMonitorRes(confSessionID, callLegSessionID)
373         ->  /* make sure the call is valid */
374             assert(confSessionID < NR_OF_CONFERENCES &&
375                    callLegSessionID < NR_OF_CALLLEGS_PER_CONFERENCE);
376             assert(client.conference[confSessionID].used &&
377                    client.conference[confSessionID].monitorLeave &&
378                    client.conference[confSessionID].callLeg[callLegSessionID].
379                      used);
380
381             /* remove the call leg */
382             client.conference[confSessionID].callLeg[callLegSessionID].
383               used = false;
384             client.conference[confSessionID].callLeg[callLegSessionID].
385               attached = false;
386       }
387
388
389   ::  /* Application decision: reserve resources for a conference */
390       atomic {
391         status.nrOfReservations < NR_OF_CONFERENCES
392         ->  /* update status */
393             status.nrOfReservations++;
394
395             /* issue reserveResources Parlay call */
396             Parlay!reserveResources(true, none);
397
398             /* wait for reply */
399             Parlay?reserveResourcesReply(_, _)
400       }
401
402 m4_forloop(CONF_ID, 0, eval(NR_OF_CONFERENCES - 1), `
403   ::  /* Application decision: monitor for leave events on conference CONF_ID */
404       atomic {
405         client.conference[CONF_ID].used &&
406           !status.conference[CONF_ID].leaveMonitorReqIssued
407         ->  /* update event status */
408             status.conference[CONF_ID].leaveMonitorReqIssued = true;
409
410             /* update client conference state */
411             client.conference[CONF_ID].monitorLeave = true;
412
413             /* issue leaveMonitorReq Parlay call */
414             Parlay!leaveMonitorReq(CONF_ID, none)
415       }
416 ')dnl
417
418 m4_forloop(CONF_ID, 0, eval(NR_OF_CONFERENCES - 1), `
419 m4_forloop(CALLLEG_ID, 0, eval(NR_OF_CALLLEGS_PER_CONFERENCE - 1), `
420   ::  /* Application decision: attach the media of call leg CALLLEG_ID, conference CONF_ID */
421       atomic {
422         client.conference[CONF_ID].used &&
423           client.conference[CONF_ID].callLeg[CALLLEG_ID].used &&
424           !client.conference[CONF_ID].callLeg[CALLLEG_ID].attached &&
425             !status.conference[CONF_ID].callLeg[CALLLEG_ID].attachMediaIssued
426         ->  /* update status information */
427             status.conference[CONF_ID].callLeg[CALLLEG_ID].attachMediaIssued = true;
428
429             /* mark the media as attached */
430             client.conference[CONF_ID].callLeg[CALLLEG_ID].attached = true;
431
432             /* ask network to attach the media */
433             Parlay!attachMedia(CONF_ID, CALLLEG_ID)
434       }
435 ')')dnl
436
437 m4_forloop(CONF_ID, 0, eval(NR_OF_CONFERENCES - 1), `
438 m4_forloop(CALLLEG_ID, 0, eval(NR_OF_CALLLEGS_PER_CONFERENCE - 1), `
439   ::  /* Application decision: detach the media of call leg CALLLEG_ID, conference CONF_ID */
440       atomic {
441         client.conference[CONF_ID].used &&
442           client.conference[CONF_ID].callLeg[CALLLEG_ID].used &&
443             client.conference[CONF_ID].callLeg[CALLLEG_ID].attached &&
```

```
444                !status.conference[CONF_ID].callLeg[CALLLEG_ID].detachMediaIssued
445          ->  /* update event status */
446              status.conference[CONF_ID].callLeg[CALLLEG_ID].detachMediaIssued = true;
447
448              /* mark the media as detached */
449              client.conference[CONF_ID].callLeg[CALLLEG_ID].attached = false;
450
451              /* ask network to detach the media */
452              Parlay!detachMedia(CONF_ID, CALLLEG_ID)
453        }
454 ')')
455   od
456 }
457
458
459 /*****************************************************************************\
460 * PROCESS TYPE ConferenceStarter                                              *
461 \*****************************************************************************/
462 proctype ConferenceStarter(byte confSessionID; chan startConference)
463 {
464   /* wait for an arbitrary time, then create the conference */
465   startConference!conferenceCreatedEvent(confSessionID)
466 }
```

Appendix B

Promela model for 2 conferences with 4 calllegs

Below is a Promela model for the Parlay Conference Call Control Service, with the number of conferences set to 2, and the number of calllegs per conference set to 4. The model has been automatically generated out of the GNU *m4* input file (included in Appendix A). The line numbers are not part of the Promela model.

```
/*************************************************************************\
* SPIN/Promela model of Parlay Conference Call Control Service            *
* ----------------------------------------------------------------------- *
*                                                                         *
* This Promela source file has been automatically generated out of the code *
* template cccs.m4, using the GNU m4 preprocessor.                        *
*                                                                         *
* The settings used to generate this source file are listed below:        *
*                                                                         *
*   Maximum number of                                                     *
*    conference calls:            2                                       *
*    calllegs per conference:     4                                       *
*                                                                         *
* ----------------------------------------------------------------------- *
* ---- Author: Peter Ebben ---- Copyright (c) 2001 by Lucent Technologies ---- *
* ----------------------------------------------------------------------- *
*                                                                         *
* Remarks:                                                                *
* --------                                                                *
*                                                                         *
* This Promela model contains the following Parlay API operations:        *
*                                                                         *
* - IpConfCallControlManager.reserveResources()                           *
* - IpConfCall.leaveMonitorReq()                                          *
* - IpCallLeg.attachMedia()                                               *
* - IpCallLeg.detachMedia()                                               *
* - IpAppConfCallControlManager.conferenceCreated()                       *
* - IpAppConfCall.partyJoined()                                           *
* - IpAppConfCall.leaveMonitorRes()                                       *
*                                                                         *
* The number of parameters and the type of parameters may differ from the *
* Parlay API specification. Only the parameters that are needed to obtain *
* results regarding network integrity have been included.                 *
*                                                                         *
\*************************************************************************/

/*************************************************************************\
* TYPE DECLARATIONS                                                       *
\*************************************************************************/
mtype = {reserveResources, reserveResourcesReply, conferenceCreated,
         conferenceCreatedReply, partyJoined, partyJoinedReply,
         leaveMonitorReq, leaveMonitorRes, attachMedia, detachMedia,
         conferenceCreatedEvent}

typedef NetworkCallLegState {
  bool used;
  bool attached;
}

typedef NetworkConferenceState {
  bool used;
  bool reserved;
  bool joinAllowed;
  bool monitorLeave;
```

```
  NetworkCallLegState callLeg[4]
}

typedef NetworkState {
  NetworkConferenceState conference[2]
}

typedef ClientCallLegState {
  bool used;
  bool attached
}

typedef ClientConferenceState {
  bool used;
  bool monitorLeave;
  ClientCallLegState callLeg[4]
}

typedef ClientState {
  ClientConferenceState conference[2]
}

typedef CallLegDecisionStatus {
  bool attachMediaIssued;
  bool detachMediaIssued
}

typedef ConferenceDecisionStatus {
  bool leaveMonitorReqIssued;
  CallLegDecisionStatus callLeg[4]
}

typedef DecisionStatus {
  ConferenceDecisionStatus conference[2];
  byte nrOfReservations
}

typedef ConferenceEventStatus {
  byte nrOfPartyJoins
}

typedef EventStatus {
  ConferenceEventStatus conference[2]
}


/*****************************************************************************\
* GLOBAL VARIABLES                                                           *
\*****************************************************************************/
chan Parlay = [0] of {mtype, byte, byte};             /* for Parlay API calls */

local byte nrOfErrors       /* the number of erroneous requests that are made */


/*****************************************************************************\
* PROCESS TYPE NetworkApplication                                            *
\*****************************************************************************/
active proctype NetworkApplication()
{
  NetworkState network;              /* state according to NetworkApplication */

  /* channel for internal use */
  chan startConference = [0] of {mtype, byte};

  /* information about already fired events, to keep things bounded... */
  EventStatus status;

  /* variables containing the next ConfCall/ConfCallLeg IDs */
  byte nextConfSessionID = 0;
  byte nextCallLegSessionID[2] = 0;

  /* temporary variables */
  bool joinAllowed;
  byte confSessionID;
  byte callLegSessionID;

  run ClientApplication();

  /* begin Parlay API call / event loop */
```

```
134
135 end:
136   do
137   ::  /* Parlay API call: reserveResources */
138       atomic {
139         Parlay?reserveResources(joinAllowed, _)
140         ->  /* make sure an extra conference is allowed */
141             assert(nextConfSessionID < 2);
142
143             /* mark conference as being reserved */
144             network.conference[nextConfSessionID].reserved = true;
145             network.conference[nextConfSessionID].joinAllowed = joinAllowed;
146
147             /* make sure the conference is started in due time */
148             run ConferenceStarter(nextConfSessionID, startConference);
149
150             /* update ID for the next conference */
151             nextConfSessionID++;
152
153             /* inform the client that resources have been reserved */
154             Parlay!reserveResourcesReply(255, 255)
155       }
156
157   ::  /* Parlay API call: leaveMonitorReq */
158       atomic {
159         Parlay?leaveMonitorReq(confSessionID, _)
160         ->  /* make sure the request is valid */
161             assert(confSessionID < 2);
162             assert(network.conference[confSessionID].used);
163             network.conference[confSessionID].monitorLeave = true
164       }
165
166   ::  /* Parlay API call: attachMedia */
167       atomic {
168         Parlay?attachMedia(confSessionID, callLegSessionID)
169         ->  /* make sure the API call relates to an existing call leg */
170             assert(confSessionID < 2 &&
171                    callLegSessionID < 4);
172             if
173             ::  network.conference[confSessionID].used &&
174                 network.conference[confSessionID].callLeg[callLegSessionID].used
175                 ->  network.conference[confSessionID].callLeg[callLegSessionID].
176                     attached = true
177             ::  else
178                 ->  printf("MSC: Error!\n");
179                     nrOfErrors++
180             fi
181       }
182
183   ::  /* Parlay API call: detachMedia */
184       atomic {
185         Parlay?detachMedia(confSessionID, callLegSessionID)
186         ->  /* make sure the API call relates to an existing call leg */
187             assert(confSessionID < 2 &&
188                    callLegSessionID < 4);
189             if
190             ::  network.conference[confSessionID].used &&
191                   network.conference[confSessionID].callLeg[callLegSessionID].used
192                 ->  /* mark the leg as being detached */
193                     network.conference[confSessionID].callLeg[callLegSessionID].
194                       attached = false
195             ::  else
196                 ->  printf("MSC: Error!\n");
197                     nrOfErrors++
198             fi
199       }
200
201   ::  /* Network event: it is time to start a reserved conference */
202       atomic {
203         startConference?conferenceCreatedEvent(confSessionID)
204         ->  /* make sure the right conference is started */
205             assert(confSessionID < 2);
206             assert(network.conference[confSessionID].reserved &&
207                    !network.conference[confSessionID].used);
208
209             /* update state information */
210             network.conference[confSessionID].used = true;
211
212             /* inform the client that the conference was started */
```

```
213             Parlay!conferenceCreated(confSessionID, 255);
214
215             /* wait for the reply */
216             Parlay?conferenceCreatedReply(_, _)
217         }
218
219     ::  /* Network event: a party joins conference 0 */
220         atomic {
221           network.conference[0].used &&
222             network.conference[0].joinAllowed &&
223               status.conference[0].nrOfPartyJoins < 4
224           ->  /* update event status information */
225               status.conference[0].nrOfPartyJoins++;
226
227               /* update state */
228               callLegSessionID = nextCallLegSessionID[0];
229               assert(callLegSessionID < 4);
230               nextCallLegSessionID[0]++;
231               network.conference[0].callLeg[callLegSessionID].used = true;
232               network.conference[0].callLeg[callLegSessionID].attached = false;
233
234               /* make API call */
235               Parlay!partyJoined(0, callLegSessionID)
236         }
237
238     ::  /* Network event: a party joins conference 1 */
239         atomic {
240           network.conference[1].used &&
241             network.conference[1].joinAllowed &&
242               status.conference[1].nrOfPartyJoins < 4
243           ->  /* update event status information */
244               status.conference[1].nrOfPartyJoins++;
245
246               /* update state */
247               callLegSessionID = nextCallLegSessionID[1];
248               assert(callLegSessionID < 4);
249               nextCallLegSessionID[1]++;
250               network.conference[1].callLeg[callLegSessionID].used = true;
251               network.conference[1].callLeg[callLegSessionID].attached = false;
252
253               /* make API call */
254               Parlay!partyJoined(1, callLegSessionID)
255         }
256
257     ::  /* Network event: party 0 leaves conference 0 */
258         atomic {
259           network.conference[0].used &&
260             network.conference[0].callLeg[0].used
261           ->  /* remove the call leg */
262               network.conference[0].callLeg[0].used = false;
263               network.conference[0].callLeg[0].attached = false;
264
265               /* if requested (via leaveMonitorReq): inform the client */
266               if
267               ::  network.conference[0].monitorLeave
268                   ->  Parlay!leaveMonitorRes(0, 0)
269               ::  else
270                   ->  skip
271               fi
272         }
273
274     ::  /* Network event: party 1 leaves conference 0 */
275         atomic {
276           network.conference[0].used &&
277             network.conference[0].callLeg[1].used
278           ->  /* remove the call leg */
279               network.conference[0].callLeg[1].used = false;
280               network.conference[0].callLeg[1].attached = false;
281
282               /* if requested (via leaveMonitorReq): inform the client */
283               if
284               ::  network.conference[0].monitorLeave
285                   ->  Parlay!leaveMonitorRes(0, 1)
286               ::  else
287                   ->  skip
288               fi
289         }
290
291     ::  /* Network event: party 2 leaves conference 0 */
```

```
292         atomic {
293           network.conference[0].used &&
294             network.conference[0].callLeg[2].used
295           ->  /* remove the call leg */
296               network.conference[0].callLeg[2].used = false;
297               network.conference[0].callLeg[2].attached = false;
298
299               /* if requested (via leaveMonitorReq): inform the client */
300               if
301               ::  network.conference[0].monitorLeave
302                   ->  Parlay!leaveMonitorRes(0, 2)
303               ::  else
304                   ->  skip
305               fi
306         }
307
308     ::  /* Network event: party 3 leaves conference 0 */
309         atomic {
310           network.conference[0].used &&
311             network.conference[0].callLeg[3].used
312           ->  /* remove the call leg */
313               network.conference[0].callLeg[3].used = false;
314               network.conference[0].callLeg[3].attached = false;
315
316               /* if requested (via leaveMonitorReq): inform the client */
317               if
318               ::  network.conference[0].monitorLeave
319                   ->  Parlay!leaveMonitorRes(0, 3)
320               ::  else
321                   ->  skip
322               fi
323         }
324
325     ::  /* Network event: party 0 leaves conference 1 */
326         atomic {
327           network.conference[1].used &&
328             network.conference[1].callLeg[0].used
329           ->  /* remove the call leg */
330               network.conference[1].callLeg[0].used = false;
331               network.conference[1].callLeg[0].attached = false;
332
333               /* if requested (via leaveMonitorReq): inform the client */
334               if
335               ::  network.conference[1].monitorLeave
336                   ->  Parlay!leaveMonitorRes(1, 0)
337               ::  else
338                   ->  skip
339               fi
340         }
341
342     ::  /* Network event: party 1 leaves conference 1 */
343         atomic {
344           network.conference[1].used &&
345             network.conference[1].callLeg[1].used
346           ->  /* remove the call leg */
347               network.conference[1].callLeg[1].used = false;
348               network.conference[1].callLeg[1].attached = false;
349
350               /* if requested (via leaveMonitorReq): inform the client */
351               if
352               ::  network.conference[1].monitorLeave
353                   ->  Parlay!leaveMonitorRes(1, 1)
354               ::  else
355                   ->  skip
356               fi
357         }
358
359     ::  /* Network event: party 2 leaves conference 1 */
360         atomic {
361           network.conference[1].used &&
362             network.conference[1].callLeg[2].used
363           ->  /* remove the call leg */
364               network.conference[1].callLeg[2].used = false;
365               network.conference[1].callLeg[2].attached = false;
366
367               /* if requested (via leaveMonitorReq): inform the client */
368               if
369               ::  network.conference[1].monitorLeave
370                   ->  Parlay!leaveMonitorRes(1, 2)
```

```
371             ::  else
372                 ->  skip
373             fi
374       }
375
376   ::  /* Network event: party 3 leaves conference 1 */
377       atomic {
378         network.conference[1].used &&
379           network.conference[1].callLeg[3].used
380         ->  /* remove the call leg */
381             network.conference[1].callLeg[3].used = false;
382             network.conference[1].callLeg[3].attached = false;
383
384             /* if requested (via leaveMonitorReq): inform the client */
385             if
386             ::  network.conference[1].monitorLeave
387                 ->  Parlay!leaveMonitorRes(1, 3)
388             ::  else
389                 ->  skip
390             fi
391       }
392   od
393 }
394
395
396 /*****************************************************************************\
397 * PROCESS TYPE ClientApplication                                              *
398 \*****************************************************************************/
399 proctype ClientApplication()
400 {
401   ClientState  client;               /*  state according to ClientApplication */
402
403   DecisionStatus status;              /* used to track already made decisions */
404
405   /* temporary variables */
406   bool joinAllowed;
407   byte confSessionID;
408   byte callLegSessionID;
409
410   /* process incoming decisions / Parlay calls */
411
412 end:
413   do
414   ::  /* Parlay API call: conferenceCreated */
415       atomic {
416         Parlay?conferenceCreated(confSessionID, 255)
417         ->  /* make sure the API call is valid */
418             assert(confSessionID < 2);
419             assert(!client.conference[confSessionID].used);
420
421             /* update client state */
422             client.conference[confSessionID].used = true;
423
424             /* send reply */
425             Parlay!conferenceCreatedReply(255, 255)
426       }
427
428   ::  /* Parlay API call: partyJoined */
429       atomic {
430         Parlay?partyJoined(confSessionID, callLegSessionID)
431         ->  /* make sure the request is valid */
432             assert(confSessionID < 2 &&
433                    callLegSessionID < 4);
434             assert(client.conference[confSessionID].used);
435
436             /* create a call leg */
437             client.conference[confSessionID].callLeg[callLegSessionID].
438               used = true;
439             client.conference[confSessionID].callLeg[callLegSessionID].
440               attached = false;
441       }
442
443   ::  /* Parlay API call: leaveMonitorRes */
444       atomic {
445         Parlay?leaveMonitorRes(confSessionID, callLegSessionID)
446         ->  /* make sure the call is valid */
447             assert(confSessionID < 2 &&
448                    callLegSessionID < 4);
449             assert(client.conference[confSessionID].used &&
```

```
                    client.conference[confSessionID].monitorLeave &&
                    client.conference[confSessionID].callLeg[callLegSessionID].
                      used);

            /* remove the call leg */
            client.conference[confSessionID].callLeg[callLegSessionID].
              used = false;
            client.conference[confSessionID].callLeg[callLegSessionID].
              attached = false;
      }

    ::  /* Application decision: reserve resources for a conference */
      atomic {
        status.nrOfReservations < 2
        ->  /* update status */
            status.nrOfReservations++;

            /* issue reserveResources Parlay call */
            Parlay!reserveResources(true, 255);

            /* wait for reply */
            Parlay?reserveResourcesReply(_, _)
      }

    ::  /* Application decision: monitor for leave events on conference 0 */
      atomic {
        client.conference[0].used &&
          !status.conference[0].leaveMonitorReqIssued
        ->  /* update event status */
            status.conference[0].leaveMonitorReqIssued = true;

            /* update client conference state */
            client.conference[0].monitorLeave = true;

            /* issue leaveMonitorReq Parlay call */
            Parlay!leaveMonitorReq(0, 255)
      }

  ::  /* Application decision: monitor for leave events on conference 1 */
      atomic {
        client.conference[1].used &&
          !status.conference[1].leaveMonitorReqIssued
        ->  /* update event status */
            status.conference[1].leaveMonitorReqIssued = true;

            /* update client conference state */
            client.conference[1].monitorLeave = true;

            /* issue leaveMonitorReq Parlay call */
            Parlay!leaveMonitorReq(1, 255)
      }

  ::  /* Application decision: attach the media of call leg 0, conference 0 */
      atomic {
        client.conference[0].used &&
          client.conference[0].callLeg[0].used &&
          !client.conference[0].callLeg[0].attached &&
            !status.conference[0].callLeg[0].attachMediaIssued
        ->  /* update status information */
            status.conference[0].callLeg[0].attachMediaIssued = true;

            /* mark the media as attached */
            client.conference[0].callLeg[0].attached = true;

            /* ask network to attach the media */
            Parlay!attachMedia(0, 0)
      }

  ::  /* Application decision: attach the media of call leg 1, conference 0 */
      atomic {
        client.conference[0].used &&
          client.conference[0].callLeg[1].used &&
          !client.conference[0].callLeg[1].attached &&
            !status.conference[0].callLeg[1].attachMediaIssued
        ->  /* update status information */
            status.conference[0].callLeg[1].attachMediaIssued = true;

            /* mark the media as attached */
            client.conference[0].callLeg[1].attached = true;
```

```
529
530             /* ask network to attach the media */
531             Parlay!attachMedia(0, 1)
532       }
533
534   ::  /* Application decision: attach the media of call leg 2, conference 0 */
535       atomic {
536         client.conference[0].used &&
537           client.conference[0].callLeg[2].used &&
538           !client.conference[0].callLeg[2].attached &&
539             !status.conference[0].callLeg[2].attachMediaIssued
540         ->  /* update status information */
541             status.conference[0].callLeg[2].attachMediaIssued = true;
542
543             /* mark the media as attached */
544             client.conference[0].callLeg[2].attached = true;
545
546             /* ask network to attach the media */
547             Parlay!attachMedia(0, 2)
548       }
549
550   ::  /* Application decision: attach the media of call leg 3, conference 0 */
551       atomic {
552         client.conference[0].used &&
553           client.conference[0].callLeg[3].used &&
554           !client.conference[0].callLeg[3].attached &&
555             !status.conference[0].callLeg[3].attachMediaIssued
556         ->  /* update status information */
557             status.conference[0].callLeg[3].attachMediaIssued = true;
558
559             /* mark the media as attached */
560             client.conference[0].callLeg[3].attached = true;
561
562             /* ask network to attach the media */
563             Parlay!attachMedia(0, 3)
564       }
565
566   ::  /* Application decision: attach the media of call leg 0, conference 1 */
567       atomic {
568         client.conference[1].used &&
569           client.conference[1].callLeg[0].used &&
570           !client.conference[1].callLeg[0].attached &&
571             !status.conference[1].callLeg[0].attachMediaIssued
572         ->  /* update status information */
573             status.conference[1].callLeg[0].attachMediaIssued = true;
574
575             /* mark the media as attached */
576             client.conference[1].callLeg[0].attached = true;
577
578             /* ask network to attach the media */
579             Parlay!attachMedia(1, 0)
580       }
581
582   ::  /* Application decision: attach the media of call leg 1, conference 1 */
583       atomic {
584         client.conference[1].used &&
585           client.conference[1].callLeg[1].used &&
586           !client.conference[1].callLeg[1].attached &&
587             !status.conference[1].callLeg[1].attachMediaIssued
588         ->  /* update status information */
589             status.conference[1].callLeg[1].attachMediaIssued = true;
590
591             /* mark the media as attached */
592             client.conference[1].callLeg[1].attached = true;
593
594             /* ask network to attach the media */
595             Parlay!attachMedia(1, 1)
596       }
597
598   ::  /* Application decision: attach the media of call leg 2, conference 1 */
599       atomic {
600         client.conference[1].used &&
601           client.conference[1].callLeg[2].used &&
602           !client.conference[1].callLeg[2].attached &&
603             !status.conference[1].callLeg[2].attachMediaIssued
604         ->  /* update status information */
605             status.conference[1].callLeg[2].attachMediaIssued = true;
606
607             /* mark the media as attached */
```

```
608             client.conference[1].callLeg[2].attached = true;
609
610             /* ask network to attach the media */
611             Parlay!attachMedia(1, 2)
612         }
613
614     ::  /* Application decision: attach the media of call leg 3, conference 1 */
615         atomic {
616           client.conference[1].used &&
617             client.conference[1].callLeg[3].used &&
618             !client.conference[1].callLeg[3].attached &&
619               !status.conference[1].callLeg[3].attachMediaIssued
620           ->  /* update status information */
621               status.conference[1].callLeg[3].attachMediaIssued = true;
622
623               /* mark the media as attached */
624               client.conference[1].callLeg[3].attached = true;
625
626               /* ask network to attach the media */
627               Parlay!attachMedia(1, 3)
628         }
629
630     ::  /* Application decision: detach the media of call leg 0, conference 0 */
631         atomic {
632           client.conference[0].used &&
633             client.conference[0].callLeg[0].used &&
634               client.conference[0].callLeg[0].attached &&
635                 !status.conference[0].callLeg[0].detachMediaIssued
636           ->  /* update event status */
637               status.conference[0].callLeg[0].detachMediaIssued = true;
638
639               /* mark the media as detached */
640               client.conference[0].callLeg[0].attached = false;
641
642               /* ask network to detach the media */
643               Parlay!detachMedia(0, 0)
644         }
645
646     ::  /* Application decision: detach the media of call leg 1, conference 0 */
647         atomic {
648           client.conference[0].used &&
649             client.conference[0].callLeg[1].used &&
650               client.conference[0].callLeg[1].attached &&
651                 !status.conference[0].callLeg[1].detachMediaIssued
652           ->  /* update event status */
653               status.conference[0].callLeg[1].detachMediaIssued = true;
654
655               /* mark the media as detached */
656               client.conference[0].callLeg[1].attached = false;
657
658               /* ask network to detach the media */
659               Parlay!detachMedia(0, 1)
660         }
661
662     ::  /* Application decision: detach the media of call leg 2, conference 0 */
663         atomic {
664           client.conference[0].used &&
665             client.conference[0].callLeg[2].used &&
666               client.conference[0].callLeg[2].attached &&
667                 !status.conference[0].callLeg[2].detachMediaIssued
668           ->  /* update event status */
669               status.conference[0].callLeg[2].detachMediaIssued = true;
670
671               /* mark the media as detached */
672               client.conference[0].callLeg[2].attached = false;
673
674               /* ask network to detach the media */
675               Parlay!detachMedia(0, 2)
676         }
677
678     ::  /* Application decision: detach the media of call leg 3, conference 0 */
679         atomic {
680           client.conference[0].used &&
681             client.conference[0].callLeg[3].used &&
682               client.conference[0].callLeg[3].attached &&
683                 !status.conference[0].callLeg[3].detachMediaIssued
684           ->  /* update event status */
685               status.conference[0].callLeg[3].detachMediaIssued = true;
686
```

```
687             /* mark the media as detached */
688             client.conference[0].callLeg[3].attached = false;
689
690             /* ask network to detach the media */
691             Parlay!detachMedia(0, 3)
692         }
693
694     ::  /* Application decision: detach the media of call leg 0, conference 1 */
695         atomic {
696           client.conference[1].used &&
697             client.conference[1].callLeg[0].used &&
698               client.conference[1].callLeg[0].attached &&
699                 !status.conference[1].callLeg[0].detachMediaIssued
700           ->  /* update event status */
701               status.conference[1].callLeg[0].detachMediaIssued = true;
702
703             /* mark the media as detached */
704             client.conference[1].callLeg[0].attached = false;
705
706             /* ask network to detach the media */
707             Parlay!detachMedia(1, 0)
708         }
709
710     ::  /* Application decision: detach the media of call leg 1, conference 1 */
711         atomic {
712           client.conference[1].used &&
713             client.conference[1].callLeg[1].used &&
714               client.conference[1].callLeg[1].attached &&
715                 !status.conference[1].callLeg[1].detachMediaIssued
716           ->  /* update event status */
717               status.conference[1].callLeg[1].detachMediaIssued = true;
718
719             /* mark the media as detached */
720             client.conference[1].callLeg[1].attached = false;
721
722             /* ask network to detach the media */
723             Parlay!detachMedia(1, 1)
724         }
725
726     ::  /* Application decision: detach the media of call leg 2, conference 1 */
727         atomic {
728           client.conference[1].used &&
729             client.conference[1].callLeg[2].used &&
730               client.conference[1].callLeg[2].attached &&
731                 !status.conference[1].callLeg[2].detachMediaIssued
732           ->  /* update event status */
733               status.conference[1].callLeg[2].detachMediaIssued = true;
734
735             /* mark the media as detached */
736             client.conference[1].callLeg[2].attached = false;
737
738             /* ask network to detach the media */
739             Parlay!detachMedia(1, 2)
740         }
741
742     ::  /* Application decision: detach the media of call leg 3, conference 1 */
743         atomic {
744           client.conference[1].used &&
745             client.conference[1].callLeg[3].used &&
746               client.conference[1].callLeg[3].attached &&
747                 !status.conference[1].callLeg[3].detachMediaIssued
748           ->  /* update event status */
749               status.conference[1].callLeg[3].detachMediaIssued = true;
750
751             /* mark the media as detached */
752             client.conference[1].callLeg[3].attached = false;
753
754             /* ask network to detach the media */
755             Parlay!detachMedia(1, 3)
756         }
757
758   od
759 }
760
761
762 /*****************************************************************************\
763 * PROCESS TYPE ConferenceStarter                                              *
764 \*****************************************************************************/
765 proctype ConferenceStarter(byte confSessionID; chan startConference)
```

```
766 {
767   /* wait for an arbitrary time, then create the conference */
768   startConference!conferenceCreatedEvent(confSessionID)
769 }
```

Appendix C

On SPIN's incorrect –o3 behaviour

In this appendix the incorrect behaviour of SPIN, when provided with the `-o3` parameter to disable safe statement merging, is analysed in detail.

Consider the following model, with two interacting processes `Client` and `Server`.

```
mtype = {leave, attach};

chan channel = [0] of {mtype, byte};

byte used[2] = 1;
byte attached[2] = 0;

active proctype Server()
{
   byte ID;

end:
  do
  ::  atomic {
        channel?attach(ID);
        if
        ::  used[ID] ->  attached[ID] = true;
        ::  else     ->  assert(false);
        fi
      }
  ::  atomic {
        used[0];
        if
        ::  true -> channel!leave(0)
        fi
      }
  ::  atomic {
        used[1];
        if
        ::  true -> channel!leave(1)
        fi
      }
  od
}


active proctype Client()
{
  byte ID;

end:
  do
  ::  atomic {
        channel?leave(ID) -> used[ID] = false;
      }
  ::  atomic {
        used[0] -> channel!attach(0)
      }
  ::  atomic {
        used[1] -> channel!attach(1)
      }
  od
}
```

The processes `Client` and `Server` communicate with each other by exchanging messages over a synchronous channel, which is named `channel`. There are two types of messages, `leave` and `attach`, each to be provided with a parameter, representing an ID. In this model, the ID can be either 0 or 1. It serves as an index into the global byte arrays `used` and `attached`.

The process `Server` consists of a do-loop, containing three alternatives, each within an atomic sequence. The first alternative is guarded by a synchronous receive of an `attach` message. Depending on the value of `used` for the specific ID (which must be thought of as representing a call leg), it either becomes attached, or it is considered an error.

The other two alternatives are guarded by the condition `used[0]` and `used[1]`, respectively. In the start state, these transitions are both enabled, since the value of both fields is greater than zero. Once the guard has been passed, a synchronous send of message `leave`, with the corresponding ID, is attempted. The if-clause contains only one alternative (`true`) and thus cannot block.

The process `Client` also consists of a do-loop containing three alternatives, each within an atomic sequence. The first alternative is guarded by a synchronous receive of message `leave`. Upon reception of this message, the value of `used` for that ID is set to `false`.

The other two alternatives are guarded by a constraint on the value of the variable `used`. In the begin state, both guards can be passed, since the value of all fields in the used array is `true`. Following the guard, an `attach` message is sent.

Note the resemblance with the model in Appendix B.

In this model, the assertion cannot be validated and no deadlock should occur, according to Promela semantics.

Consider an execution of this model. In the begin state, four transitions are enabled, namely those that are guarded by `used[0]` and `used[1]`. For the sake of argument, assume that the transition on line 22 is taken first. By doing so, an atomic sequence is entered and process `Server` has the exclusive privilege to execute. The rendezvous send of the message `channel!leave(0)` succeeds, since `Client` is at the beginning of the do-loop. Thus, line 25 of process `Server` synchronizes with line 47 of process `Client`. Since this is contained within an atomic sequence, atomicity flows with the send, and thus process `Client` now holds the exclusive privilige to execute. It will react by setting the value of `used` for the corresponding ID to `false`. This atomic sequence is now completed. There is no longer any process holding the exclusive privilege to execute, so normal, non-deterministic execution is resumed. Since the last statement executed in process `Server` was the last statement of an atomic sequence, that process is no longer located within one of its alternatives.

It is left as an exercise to the reader to study the execution flow within the other alternatives. Due to the passing of the atomicity token in synchronous sends, control flow is such that no intervening event can take place.

When verifiying this model with SPIN, checking for assertion violations and invalid endstates[23], SPIN produces the following output:

```
State-vector 24 byte, depth reached 16, errors: 0
      10 states, stored
       7 states, matched
      17 transitions (= stored+matched)
      32 atomic steps
```

SPIN was thus not able to find any deadlock situations. The `assert` statement in process `Server` is reported as being unreachable.

[23] The labels named `end` in the two processes define valid endstates. Due to the placement of the labels, they express that, if both processes are at the beginning of the do-loop and no further transitions are possible, the endstates are valid. However, if one process is still located within any of its alternatives, the endstate is invalid and we are thus facing a real deadlock situation.

However, when the safe statement merging option is disabled, by invoking SPIN with the `-o3` parameter, different results are obtained:

```
State-vector 24 byte, depth reached 24, errors: 4
      30 states, stored
      11 states, matched
      41 transitions (= stored+matched)
      80 atomic steps
```

During this verification run, SPIN was able to locate 4 errors. Moreover, the `assert` statement in process `Server` was reported as unreachable code, so there are no assertion violations, but four deadlocks.

Upon closer examination, it turns out that these errors occur because SPIN inserts a 'hidden goto' statement, immediately following the `fi` keywords, but before the closing brace of the atomic sequences. This hidden goto jumps to the beginning of the do-loop. However, since this statement is inserted inside the atomic sequence, the synchronous send statements are no longer the last statements in the atomic sequence, and deadlocks can occur.

Table 7 contains an example error trace, to show why deadlocks can occur in the model above.

Step	*Process*	*Line*	*Instruction*	*Comments*	*Exclusive*
1	`Server`	22	`used[0]`	check succeeds	`Server`
2	`Server`	24	`true`	guard in if-construct succeeds	`Server`
3	`Server`	24	`channel!leave(0)`	synchronises with `Client`	`Client`
3	`Client`	44	`channel?leave(ID)`	`ID == 0`	`Client`
4	`Client`	44	`used[ID] = false`	`used[0]` is now false	*none*
5	`Client`	50	`used[1]`	check succeeds	`Client`
			`channel!attach(1)` blocks		*none*
6	`Server`	25	*`hidden goto`*	`Server` at beginning of do-loop	*none*
7	`Server`	28	`used[1]`	check succeeds	`Server`
8	`Server`	30	`true`	guard in if-construct succeeds	`Server`
			`channel!leave(1)` blocks		*none*

Table 7: Error trace for deadlock situation

The execution starts by entering the atomic sequence at line 21. The server now holds the exclusive privilege to execute statements. The guard succeeds, the condition in the if-clause is fulfilled and the synchronous send takes place, since the `Client` process is able to perform the matching receive. In doing so, control passes from `Server` to `Client` and subsequently `Client` holds the exclusive privilege to execute statements. The atomic sequence runs to completion. Since no process holds the exclusive privilege to execute, non-deterministic execution is resumed. The guard `used[1]` is fulfilled, so the `Client` enters the atomic sequence at line 49. The subsequent synchronous send *blocks*. If the `Server` process was at the beginning of the do-loop, this would have not occurred, since synchronisation would then be possible. The crux is that the Server is *not* at the beginning of the do-loop, but still inside the alternative guarded by `used[0]`. Following the error trail it becomes clear that the send operation was blocked since the *hidden goto* had not been executed yet. Once the `Server` executes this 'statement' it is again back at the beginning of the do-loop. Now, execution *can* resume with the statement in the `Client` process that was blocked previously, but it *need* not be. In the error trail above, instead of continuing execution with the previously blocked, but now executable statement, an atomic sequence in the `Server` process is entered. The resulting synchronous send blocks, since `Client` is expecting a different message. The system is thus in a state of deadlock.

This is clearly an error in SPIN, and not in our model, since the hidden goto is inserted by SPIN, although it is not part of the semantics. When disabling safe statement merging, the verifier can thus return an incorrect answer, due to the way SPIN deals with these hidden gotos. When all optimisations are enabled – the default condition – SPIN handles these self-created statements correctly. It is also interesting to note that SPIN apparently inserts these hidden statements only following an if-clause, and not as final statement following each of the alternatives within a do-loop.

When the statements of the form

```
if
::  true -> channel!leave(x)
fi
```

are replaced by the statement

```
channel!leave(x)
```

which are supposed to be semantically equivalent[24] since the condition of the if-clause always succeeds, SPIN is not able to find errors in the `-o3` case either.

[24] Not counting the number of transitions that is needed to obtain the effect.

Bibliography

[AAKS98] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. *Safety and Security of Programmable Network Infrastructures*, IEEE Communications Magazine, issue on Programmable Networks, vol. 36, no. 10, October 1998, pp. 84-92. http://www.cis.upenn.edu/~angelos/Papers/ieeecomms.ps.gz

[ABR99] Carlos Areces, Wiet Bouwma and Maarten de Rijke, *Feature Interaction as a Satisfiability Problem*, Proceedings of MASCOTS'99, Maryland, USA, 1999. http://turing.wins.uva.nl/~carlos/Papers/mascots-fi.ps.Z

[BDH00] Dragan Bošnaèki, Dennis Dams and Leszek Holenderski, *Symmetric SPIN*, Proceedings of the 7th SPIN Workshop, Stanford University, California; Lecture notes in Computer Science, volume 1885, Springer Verlag, September 2000, pp. 1-19. http://netlib.bell-labs.com/netlib/spin/ws00/18850001.pdf

[CaVe93] E. Jane Cameron and Hugo Velthuijsen, *Feature interactions in telecommunications systems*, IEEE Communications Magazine, volume 31, issue 8, August 1993, pp. 18-23. http://ieeexplore.ieee.org/iel1/35/5925/00229532.pdf

[EncBr] *Telephone and telephone system*, Encyclopædia Britannica, Internet edition. http://www.britannica.com/eb/article?eu=119001

[FGKS97] Igor Faynberg, Lawrence R. Gabuzda, Marc P. Kaplan & Nitin J. Shah, *The Intelligent Network Standards, Their application to Services*, McGraw-Hill, USA, 1997.

[Hanr99] H.E. Hanrahan, *Evolution of Standards for Advanced Telecommunications Services and Network Management*, Proceedings of the 2nd Annual South African Telecommunications Networks Applications Conference, Durban, September 1999, pp. 12-17. http://www.ee.wits.ac.za/~comms/output/satnac99/hanrahan_servev.ps

[Hatt97] Les Hatton, *Software failures - follies and fallacies*, IEE Review, volume 43, issue 2, March 20, 1997, pp. 49-52. http://ieeexplore.ieee.org/iel1/2188/12713/00586152.pdf

[Holz91] Gerard J. Holzmann, *Design and Validation of Computer Protocols,* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1991. http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html

[Holz97] Gerard J. Holzmann, *The SPIN Model Checker*, IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295. http://cm.bell-labs.com/cm/cs/who/gerard/gz/ieee97.ps.gz

[Holz98] Gerard J. Holzmann, *An analysis of bitstate hashing*, Formal Methods in Systems Design, Nov. 1998. http://cm.bell-labs.com/cm/cs/who/gerard/gz/fmsd98.ps.gz

[HPY96] Gerard Holzmann, Doron Peled and Mihalis Yannakakis, *On Nested Depth First Search*, Proceedings of the Second SPIN workshop, Rutgers University, New Jersey, USA, August 5, 1996. ftp://netlib.bell-labs.com/netlib/spin/ws96/Ho.ps.Z

[HPW01] J.W. Hellenthal, F.J.M. Panken & M. Wegdam, *Validation of the Parlay API through prototyping*, IEEE Intelligent Network 2001 Workshop (IN 2001), Boston, MA, USA, May 6-9, 2001, pp. 58-63.

[JAMS00] R. Jain, F.M. Anjum, P. Missier and S. Shastry, *Java Call Control, Coordination, and Transactions*, IEEE Communications Magazine, January 2000, pp. 108–114.

[Josa99] Audun Jøsang, *Open IN Service Provider Access*, Proceedings of the Conference on Integrating Intelligent Networks and Computer Telephony, IBC Global Conferences, London, 1999. http://www.item.ntnu.no/~ajos/papers/oinspa.ps

[KPS00] Markos Koltsidas, Ognjen Prnjat and Lionel Sacks, *Development of Parlay-based Applications Using UML and SDL,* Third IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MMNS) 2000, Fortaleza, Brazil, September 2000. http://www.ee.ucl.ac.uk/~oprnjat/Prnj00c.pdf

[McDo94] J.C. McDonald, *Public network integrity - avoiding a crisis in trust*, IEEE Journal on Selected Areas in Communications, vol. 12, issue 1, January 1994, pp. 5-12. http://ieeexplore.ieee.org/iel1/49/6663/00265698.pdf

[MWWM97] Victoria Mónton, Keith Ward, Mark Wilby and Ronnie Masson, *Risk assessment methodology for network integrity*, British Telecommunications Technology Journal, Volume 15, Number 1, January 1997.

[Parl99] The Parlay Organisation, *Parlay API Business Benefits White Paper*, draft version 1.0, June 9, 1999.

[Parl00a] The Parlay Technical Team, *Parlay APIs 2.1 – Generic Call Control Service Interfaces*, version 2.1, June 26, 2000. http://www.parlay.org. This document is part of the Call Processing Service Interface documents.

[Parl00b] The Parlay Technical Team, *Parlay APIs 2.1 – Call Processing Sequence Diagrams*, version 2.1, June 26, 2000. http://www.parlay.org. This document is part of the Call Processing Service Interface documents.

[Parl00c] The Parlay Group, *White paper: The Parlay APIs*, version 1.1, June 2, 2000. http://www.parlay.org/docs/Whatis.pdf

[Parl00d] The Parlay Group, *Overview: The Parlay Group Objectives*, version 0.81, May 30, 2000. http://www.parlay.org/docs/Objectives.pdf

[Parl00e] The Parlay Technical Team, *Parlay APIs 2.1 – Call Processing Class Diagrams*, version 2.1, June 26, 2000. http://www.parlay.org. This document is part of the Call Processing Service Interface documents.

[Parl00f] The Parlay Technical Team, *Parlay APIs 2.1 – Framework Sequence Diagrams*, version 2.1.1, November 20, 2000. http://www.parlay.org. This document is part of the Framework Interface documents.

[Parl00g] The Parlay Technical Team, *Parlay APIs 2.1 – Framework Interfaces, Client Application View*, version 2.1.1, November 20, 2000. http://www.parlay.org. This document is part of the Framework Interface documents.

[Parl01a] The Parlay Group, *Frequently Asked Questions*, 2001. http://www.parlay.org/about/faqs.asp

[Parl01b] The Parlay Technical Team, *Parlay APIs 2.1 – Common Data Definitions*, version 2.1.1, January 3, 2001. http://www.parlay.org. This document is part of the Common Interface documents.

[Parl01c] The Parlay Technical Team, *Parlay APIs 2.1 – Generic Call Control Service Data Definitions*, version 2.1.1, January 3, 2001. http://www.parlay.org. This document is part of the Call Processing Service Interface Documents.

[Prnj99a] Ognjen Prnjat, Lionel Sacks, *Integrity Methodology for Interoperable Environments*, IEEE Communications Magazine, Vol. 37, No. 5, pp. 126-139, May 1999. http://www.ee.ucl.ac.uk/~pants/papers/ieee.pdf

[Prnj99b] Ognjen Prnjat, Lionel Sacks, *Telecommunications System Design Complexity and Risk Reduction based on System Metrics*, 10th European Workshop on Dependable Computing (EWDC10), Vienna, Austria, May 1999. http://www.ee.ucl.ac.uk/~pants/papers/metrics2.pdf

[Prnj99c] Ognjen Prjnat, Lionel Sacks, *Impact of Security Policies on the TMN X Interface Integrity and Performance*, IEEE Latin American Network Operations and Management Symposium LANOMS'99, Rio de Janeiro, Brazil, December 1999. http://www.ee.ucl.ac.uk/~oprnjat/testing2.pdf

[Prnj00] Ognjen Prnjat, Lionel Sacks, *Inter-domain Integrity Management for Programmable Network Interfaces*, Third IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MMNS) 2000, Fortaleza, Brazil, September 2000. http://www.ee.ucl.ac.uk/~oprnjat/Prnj00b.pdf

[PROM] *Promela Language Reference*. http://cm.bell-labs.com/cm/cs/what/spin/Man/promela.html

[PSH98] Ognjen Prnjat, Lionel Sacks and Håvard Hegna, *Testing the Integrity vs. Security Requirements on the TMN X Interface*, EUNICE '98 Network Management and Operation Summer School, Munich, Germany, September 1998. http://www.ee.ucl.ac.uk/~pants/papers/TestingXInterface.pdf

[ReVe91] Andrew L. Reibman and Malathi Veeraraghavan, *Reliability modeling: an overview for system designers*, Computer, IEEE, volume 24, issue 4, April 1991, pp. 49-57. http://ieeexplore.ieee.org/iel1/2/2541/00076262.pdf

[RMCL98] S. Rooney, J.E. van der Merwe, S.A. Crosby and I.M. Leslie, *The Tempest, a Framework for Safe, Resource Assured, Programmable Networks*, IEEE Communications Magazine, Vol. 36, No. 10, October 1998. http://www.cl.cam.ac.uk/Research/SRG/netos/ncam/docs/public/IEEE-Comms-98.ps.gz

[Ruys01] Theo C. Ruys, *Towards Effective Model Checking*, Ph.D. Thesis. University of Twente, Department of Computer Science. March 2001. CTIT PhD-Thesis Series 2001-34. http://wwwhome.cs.utwente.nl/~ruys/ruys-phd-thesis.pdf

[Sein94] René Seindal. *GNU m4, version 1.4 – A powerful macro processor*, Edition 1.4, November 1994. http://sunsite.ualberta.ca/Documentation/Gnu/m4-1.4/html_chapter/m4_toc.html

[SS7] Illuminet, *Signaling System 7 (SS7)*, Web ProForum Tutorials, The International Engineering Consortium. http://www.iec.org

[Stal98] William Stallings, *Operating Systems: Internals and design principles*, third edition, Prentice Hall, Upper Saddle River, New Jersey 07458, USA, 1998.

[Tane96] Andrew S. Tanenbaum, *Computer Networks*, third edition, Prentice Hall, Upper Saddle River, New Jersey 07458, USA, 1996.

[Turn00] Kenneth J. Turner, *Structuring Telecommunications Features*, Stephen Gilmore and Mark Ryan, editors, Language Constructs for Describing Features, Springer-Verlag, London Ltd, 2000/2001, pp. 1-10. http://www.dcs.ed.ac.uk/home/stg/fireworks/book/Turner/struc-feat.ps

[UML97] Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, I-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies & Softeam, *UML Notation Guide*, version 1.1, September 1, 1997. ftp://ftp.omg.org/pub/docs/ad/97-08-05.pdf

[Ward95] Keith Ward, *The Impact of Network Interconnection on Network Integrity*, British Telecommunications Engineering, Volume 13, January 1995.

[Webs91] Robert B. Costello, Editor in Chief, *Webster's College Dictionary*, Random House, Inc., New York, 1991.

Summary

This thesis describes research on the subject of integrity of telecommunications networks when controlled via the Parlay API.

The Parlay API is a technology-independent Application Programming Interface that aims to provide ways to intimately link IT applications with the possibilities of the communications world. The API provides controlled access over telecommunications network capabilities to applications out of the network operator's domain. This allows third parties to develop and deploy advanced telecommunications services, using the public telecommunications network. The API directly interacts with the underlying telecommunications network, so that e.g. efficient routing decisions can be made. The application can set triggers to be informed about certain call events. If such a call event occurs, the application is informed about the event and may indicate how call processing is to be continued.

History has shown that telecommunications network integrity breaches can have huge impact on day-to-day life. The network operator, who is responsible for the integrity of his network, has to be convinced of the fact that such third party applications cannot harm his network's integrity, before he is ever going to deploy the API. Since in most cases the network operator will not have access to the design or source code of the application, there are only limited ways in which the network operator can influence and limit the application's capabilities. The first is by defining exactly how many network resources may be used by the application at any time in a Service Level Agreement that has to be digitally signed by the application. The second, and more important, is monitoring the application's behaviour in terms of the API calls that are issued. Due to the nature of the Parlay API, a Parlay Gateway has to be implemented in the public telecommunications network before the API can be used. It is thus this gateway that has to monitor for and correct integrity-endangering API calls.

Based on the Parlay API specification version 2.1, we have modelled a small part of the (huge) API in the language Promela, which serves as input for the formal verification tool ('model checker') SPIN. SPIN is said to be one of the most powerful model checkers around, but we were nevertheless facing the state-space explosion problem. Even a very small part of the API turned out to be impossible to verify completely with the computing resources we had at our disposal, even after some improvements to the model had been made. Fortunately, SPIN provides ways to conduct approximate verifications, in which (large) parts of the state-space are verified, but 100 percent coverage is not guaranteed. We have managed to find one minor integrity issue during our formal verification experiments. Also, we came across some errors in the verification tool SPIN, which we looked further into.

As a result of further study of (parts of) the Parlay API, we managed to identify a few more risks to the integrity of the network if it is controlled via the Parlay API. In short, an integrity breach is defined as the user's perception of something being wrong with the telecommunications network. It does not necessarily imply that, from a technical point of view, something really is wrong with the network (e.g., congestion). Indeed, we were not able to find any situations in which parts of the network were rendered unusable (which does not mean that they cannot exist), but nevertheless the integrity issues require the necessary attention. For a Parlay application, it turns out to be fairly easy to get a faulty view on the status of the ongoing call in the network that can result in erroneous API calls. Also, we have shown how one application can effectively prohibit another one from functioning, even if they conceptually do not have anything to do with each other. The reason for this lies in the policy that is defined by the API for denying call notification requests, which is too strict in our opinion. Also, the API lacks ways to circumvent the problems that are caused by this policy.

Apart from these integrity issues, for which fixes have been proposed, some areas of further research have been identified. Especially the state-space explosion problem requires more attention in the scientific community, but there is also more work for the Parlay API designers. The documentation is not very clear on numerous points, and some contradictions have been identified.

Overall, we conclude that it will take some time before the Parlay API will be suitable for deployment on a large scale in the public telecommunications network.

Samenvatting

Deze scriptie beschrijft onderzoek betreffende de integriteit van telecommunicatienetwerken in het geval dat controle over deze netwerken uitgeoefend wordt middels de Parlay API.

De Parlay API is een technologie-onafhankelijke *Application Programming Interface*, met als doel om vergaande integratie tussen IT applicaties en de mogelijkheden die de communicatiewereld ons biedt mogelijk te maken. De API biedt gecontroleerde toegang over faciliteiten van het telecommunicatienetwerk aan applicaties die buiten het domein van de netwerk operator vallen. Dit maakt het voor derden mogelijk om geavanceerde telecommunicatiediensten te ontwikkelen en te exploiteren, hierbij gebruikmakend van het publieke telecommunicatienetwerk. De API heeft directe toegang tot het telecommunicatienetwerk, wat het mogelijk maakt om bijvoorbeeld efficiëntere routeringsbeslissingen te nemen. De applicatie kan in het netwerk triggers zetten om genotificeerd te worden als bepaalde *call*-gerelateerde gebeurtenissen zich voordoen. Als een dergelijke gebeurtenis zich daadwerkelijk voordoet, dan wordt de applicatie hiervan op de hoogte gebracht en kan deze vervolgens aangeven wat er verder met de *call* moet gebeuren.

De geschiedenis heeft ons geleerd dat integriteitgerelateerde problemen in telecommunicatienetwerken een vergaande impact kunnen hebben op het dagelijks leven. De netwerk operator, die verantwoordelijk is voor de integriteit van zijn netwerk, moet overtuigd zijn van het feit dat dergelijke Parlay-applicaties de integriteit van zijn netwerk niet in gevaar brengen, alvorens hij ooit Parlay-services aan zal gaan bieden. Aangezien de netwerk operator in de meeste gevallen geen toegang zal hebben tot het ontwerp of de implementatie van een dergelijke applicatie zijn er slechts beperkte mogelijkheden waarop de netwerk operator de mogelijkheden van dergelijke applicaties kan inperken en beï nvloeden. De eerste manier waarop dit plaats kan vinden is door middel van het opstellen van een Service Level Agreement, waarin precies omschreven wordt hoeveel netwerkbronnen er door de applicatie gebruikt mogen worden. Dit ‘contract’ moet voordat de applicatie gebruikt gaat worden, digitaal ondertekend worden. De tweede, en belangrijkere, manier waarop de netwerk operator controle uit kan oefenen op het gedrag van dergelijke applicaties, is door het monitoren van de API methoden die aangeroepen worden door de applicatie. Vanwege de aard van de Parlay API moet een Parlay gateway geï mplementeerd worden in het publieke telecommunicatienetwerk alvorens de API gebruikt kan worden. Het is dan ook deze gateway die de door de applicatie aangeroepen methoden zal moeten bewaken om te voorkomen dat ze de integriteit van het netwerk in gevaar brengen.

Gebruikmakend van de Parlay API specificatie versie 2.1 is een klein deel van de (omvangrijke) API gemodelleerd in de taal Promela, welke dient als invoer voor de formele verificatietool (‘model checker’) SPIN. Van SPIN wordt gezegd dat het een van de krachtigste model checkers van dit moment is. Ondanks dat zijn we tegen het state-space explosion probleem aangelopen. Het bleek niet mogelijk om zelfs maar een klein deel van de API formeel te verifiëren met de computerapparatuur die ons ter beschikking stond, zelfs niet nadat verschillende verbeteringen in het model aangebracht waren. SPIN biedt gelukkig ook mogelijkheden voor een benaderende verificatie, waarbij grote delen van de toestandsruimte doorzocht worden, maar geen 100 procent dekking gegarandeerd kan worden. Tijdens de verificatie-experimenten is een klein probleempje met de netwerk integriteit gevonden. Ook zijn we tegen een aantal fouten in de verificatietool SPIN aangelopen, die verder onderzocht zijn.

Als gevolg van verdere studie van de API zijn nog wat meer mogelijke integriteitsproblemen aan het licht gekomen. Een integriteitsprobleem wordt hierbij gedefinieerd als een situatie waarbij er, vanuit het standpunt van een gebruiker gezien, iets mis *lijkt* met het netwerk. Dit betekent niet dat er ook noodzakelijkerwijs iets mis moet zijn in het netwerk vanuit een technisch oogpunt (bijvoorbeeld congestie). We zijn er niet in geslaagd om situaties te vinden waarin delen van het netwerk onbruikbaar werden (wat overigens niet

betekent dat die niet voor zouden kunnen komen), maar hoe dan ook vereisen de integriteitsproblemen nog de nodige aandacht. Het blijkt dat het vrij eenvoudig is om als Parlay applicatie in een toestand te komen waarbij de status van de call, zoals die door de applicatie bijgehouden wordt, verschilt van de werkelijke situatie. In zo'n toestand is het gemakkelijk om foutieve API methoden aan te roepen. We hebben ook laten zien hoe een applicatie effectief de uitvoer van een andere applicatie kan verhinderen, ook al hebben deze conceptueel gezien niets met elkaar te maken. De oorzaak hiervoor ligt in het beleid dat de Parlay API hanteert betreffende het weigeren van *call event notification requests*, dat te strikt lijkt. De API biedt ook geen manieren om de hierdoor geï ntroduceerde problemen te omzeilen.

Afgezien van deze integriteitsproblemen, waarvoor oplossingen voorgesteld zijn, zijn een aantal gebieden voor verder onderzoek geï dentificeerd. Met name het state-space explosion probleem vereist nog de nodige aandacht vanuit de wetenschappelijke wereld, maar er is ook nog het nodige werk voor de ontwerpers van de Parlay API. De documentatie behorende bij de API is niet op alle punten even duidelijk en bevat zelfs tegenstrijdigheden.

Al met al kunnen we concluderen dat het nog wel even zal duren voordat de Parlay API geschikt is voor toepassing op grote schaal in het publieke telecommunicatienetwerk.