University of South Australia

Verification of the WAP Transaction Layer using Coloured Petri nets

Steven Donald Gordon

B.Eng. (Hons)

Institute for Telecommunications Research
and Computer Systems Engineering Centre,
School of Electrical and Information Engineering
University of South Australia

A thesis submitted for the degree of
Doctor of Philosphy in Telecommunications

24 November 2001

Contents

List of Figures

List of Tables

Listings

Acronyms and Abbreviations

AEC	Acknowledgment Expiration Counter
AMPS	Advanced Mobile Phone Service
ANSI	American National Standards Institute
ATM	Asynchronous Transfer Mode
BE	Binding Element
BNF	Backus Naur Form
CDMA	Code Division Multiple Access
CDPD	Cellular Digital Packet Data
CFFP	Chaos-Free Failures Divergence
cHTML	Compact HTML
CLNP	Connection-less Network Protocol
CNF	Confirmed
cnf	confirm
CON	Continue
CPN	Coloured Petri Net
CSD	Circuit Switched Data
DECT	Digital Enhanced Cordless Telecommunications
DES	Data Encryption Standard
DFSA	Deterministic FSA
DL	Deadlock
ECDH	Elliptic Curve Diffie-Hellman
ETSI	European Telecommunications Standards Institute
FDDI	Fibre Distributed Data Interconnect
FIFO	First-In First-Out
FSA	Finite State Automata
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communication
GTR	Group Trailer
GUTS	General UDP Transport Service
HTML	Hypertext Markup Language

HTTP Hypertext Transfer Protocol
I2R Upper integer bound of InitToResp
ind indication
IOTP Internet Open Trading Protocol
IP Internet Protocol
ISDN Integrated Services Digital Network
ISO International Organization for Standardization
ITU International Telecommunications Union
ITU-T ITU Telecommunications Standardization Sector
IWS Integrated Weapons Simulator
LAN Local Area Network
LL Livelock
ML Meta Language
MPL Maximum Packet Lifetime
MSISDN Mobile Station/Subscriber ISDN Number
NIP Not In TR-Protocol language
NIS Not In TR-Service language
OSI Open Systems Interconnection
PDA Personal Digital Assistant
PDC Personal Digital Cellular
PDU Protocol Data Unit
PHS Personal Handyphone System
PN Petri Net
PSN Packet Sequence Number
R2I Upper integer bound of RespToInit
RCR Re-transmission Counter
res response
RES Reserved
RID Re-transmission Indicator
req request
RSA Rivest, Shamir, and Adleman public-key cryptosystem
SAP Service Access Point
SAR Segmentation and Re-assembly
SCC Strongly Connected Component
SDU Service Data Unit
SGSN Serving GPRS Support Node
SIP Session Initiation Protocol
SMS Short Message Service

TCL	Transaction Class
TCP	Transmission Control Protocol
TETRA	Terrestrial Trunked Radio Access
TID	Transaction Identifier
TIDok	Transaction Okay
TIDve	Transaction Verify
TM	Terminal Marking
Tok	Transaction Okay
TPI	Transport Information Items
TR	Transaction
TSD	Time Sequence Diagram
T/TCP	Transaction/TCP
TTR	Transmission Trailer
Tve	Transaction Verify
UDP	User Datagram Protocol
UserAck	User Acknowledgment
USSD	Unstructured Supplementary Services Data
WAE	Wireless Application Environment
WAN	Wide Area Network
WAP	Wireless Application Protocol
WDP	Wireless Datagram Protocol
WML	Wireless Markup Language
WSP	Wireless Session Protocol
WTA	Wireless Telephony Application
WTLS	Wireless Transport Layer Security
WTP	Wireless Transaction Protocol
WWW	World Wide Web
XTP	Xpress Transport Protocol

Summary

The rapid growth of the Internet and its use has inevitably led to its entrance into the wireless world. However, the characteristics of wireless networks and the devices typically used in them, result in a significantly different operating environment than that provided by the existing Internet infrastructure. A range of protocols and architectures, of which the Wireless Application Protocol (WAP) is one, have been proposed to overcome the difficulties of providing Internet services in wireless networks. The Transaction layer in WAP defines a protocol to support transactions where an Initiator makes a request to a Responder, which returns the requested information. This thesis applies a protocol engineering methodology to analyse the functional behaviour of the Transaction layer in WAP. Coloured Petri net (CPN) models of the Class 2 Transaction layer service (TR-Service) and protocol (TR-Protocol) are created based on an existing specification. The aim is to verify that the TR-Protocol is a faithful refinement of the TR-Service, in terms of sequences of primitives seen by the users.

The WAP Transaction layer specification defines the TR-Service using narrative descriptions and a table specifying the possible primitives that may follow one another, as seen by a user. The CPN modelling and analysis process has revealed the existing specification is ambiguous and incomplete. A set of rules is defined to remove the ambiguities. These rules, and the existing specification, are formalised in the TR-Service CPN. State space generation and automata reduction are used to produce the TR-Service language, the set of possible global sequences of primitives, as opposed to the local sequences only available in the WAP specification.

The WAP Transaction layer specification also describes the TR-Protocol. The dynamic behaviour of the TR-Protocol is described mainly using state tables. These state tables are formalised by the creation of the TR-Protocol CPN. Without loss of generality, only a single transaction between one Initiator and one Responder is modelled. As the initial focus is on the core operation of the TR-Protocol, the version handling and segmentation and re-assembly features are omitted from the model. The underlying service (the Transport layer in WAP) is assumed to be error free (no corruption, loss or duplication), but allows overtaking of protocol data units (PDUs). A set of assumptions are also made that remove certain errors from the WAP specification. As a result, suggested

changes to the specification have been input to the WAP Forum, and the majority of them have been accepted.

State space and language analysis are used to investigate the following four properties of the TR-Protocol: refinement of the TR-Service, in terms of sequences of primitives; successful termination (which includes absence of deadlocks); absence of livelocks; and absence of redundant state table entries (dead transitions in the CPN). The analysis reveals the following errors in the existing specification: ambiguous semantics of two PDUs; the possibility of a re-transmitted Invoke PDU starting a new transaction from the Responder's point of view; and the possibility of the Responder misinterpreting a re-transmitted Ack PDU. A set of changes are proposed, leading to a Revised TR-Protocol. Further analysis proves the four desired properties hold for the Revised TR-Protocol for all permutations of the parameters of the CPN (i.e. a toggle of the user acknowledgment feature, and the maximum values of the re-transmission counters at both Initiator and Responder) when the counters are less than 5. This includes the suggested configuration to be used in several GSM (Global System for Mobile communication) networks.

Finally, from the results obtained from the set of parameter values analysed, observations are made on the relationship between the counters used in the TR-Protocol CPN and the state space size. These relationships may be used in the future to prove properties independently of the particular parameter values, significantly increasing the power of analysis.

Declaration

I declare that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge it does not contain any material previously published or written by another person except where due reference is made in the text.

Steven Donald Gordon

Acknowledgments

My supervisor, Professor Jonathan Billington, provided suggestions for directions of my research, and has continued throughout my candidature to be a source of technical expertise. He has also provided substantial feedback on the writing of this thesis, for which he is gratefully acknowledged. Finally, I would like to thank Professor Billington for his advice and support for all other matters related to undertaking postgraduate study, such as funding and employment opportunities.

Participation in the Computer Systems Engineering Centre (CSEC) and the TNS Group of the Institute for Telecommunications Research (ITR) has been of enormous benefit, both for my research and in advancing my general knowledge and communication skills. I would like to thank all the members for this. Several people deserve a special mention: Maria Villapol for the thought provoking discussions over the 3 years; Lars Kristensen for the assistance with Design/CPN and analysis techniques; and Lin Liu, Chun Ouyang, Bing Han and Lars for reviewing a draft of this thesis.

The financial support from the following organisations has been vital for the successful completion of my research: the Australia Government for an Australian Postgraduate Award; the Cooperative Research Centre for Satellite Systems for the DSpace Scholarship; and the School of Electrical and Information Engineering, ITR and CSEC for funding to attend conferences and general support.

I am indebted to my family and friends for their support during my candidature. In particular, I am grateful to my parents for giving me the opportunity to undertake my undergraduate and postgraduate studies. Finally, thanks go to Sarah Williams for her continuing love and understanding, especially during the difficult phases of writing this thesis.

Chapter 1

Introduction

1.1 Background

The Internet [34] has spread into our everyday lives via the proliferation of desktop computers and software for distributed applications, such as World Wide Web (WWW) [12] browsers. At the same time, mobile telephony has allowed people to contact the rest of the world from almost anywhere and at anytime. Providing data services offered on the Internet to mobile users, often referred to as *mobile data* [146], is an obvious step forward. However, moving Internet services to a mobile medium is not straightforward, due to two main reasons.

The existing Internet infrastructure has been designed for fixed users communicating via a wired network. Mobile users communicate via a wireless network, which exhibits many different characteristics from a wired network. These include: relatively lower bandwidth, higher probability of data loss and greater transmission delays. These characteristics detrimentally affect (sometimes severely) the performance of Internet protocols. The second difficulty for mobile data services is adapting existing Internet applications written for desktop computers, to applications suitable for mobile devices (e.g. mobile telephones have a significantly smaller screen size and a numeric keypad as an input device).

As a result of these problems, a range of communication protocols and architectures have been proposed [37, 190, 21] and/or implemented [171, 143, 144, 123, 124, 24] for mobile data services. The Wireless Application Protocol (WAP) [171] is one architecture proposed by an industry consortium, called the WAP Forum [170], that has been taken up by many mobile phone manufacturers. It has therefore been exposed to a wide consumer market. WAP includes several protocols and an application environment, each optimized towards operating in a wireless environment. The Transaction layer [183] in WAP defines a protocol for performing short request/response transactions between two entities. This Transaction Protocol is an integral component of browsing-type applica-

tions, where mobile users request information from a server which returns a response. The Transaction layer specification is publicly available [183], and comprises a definition of the service it offers to the upper layers (Transaction Service) and the mechanisms for providing the service (Transaction Protocol).

Although performance is an important issue in the design of the WAP architecture, it is also necessary to ensure the communication protocols are unambiguous, complete and functionally correct. One approach to ensuring the correctness of the Transaction layer design is to create formal models of the Transaction Service and Transaction Protocol, and then compare the two to determine if indeed the protocol provides the defined service. Coloured Petri nets (CPNs) [86] are a suitable modelling language for this verification task, as they can conveniently express non-determinism, concurrency and different levels of abstraction that are inherent in communication protocols. They are also supported by the computer tool Design/CPN [109].

1.2 Research Aims

The overall aim of this thesis is to verify the design of the Transaction layer in the WAP architecture. Three objectives must be reached in order to achieve this aim.

The verification involves comparing the Transaction Protocol to the Transaction Service. The first objective, therefore, is to provide a complete and unambiguous definition of the Transaction Service. A CPN forms the major part of this definition.

The second objective is to develop a CPN model of the Transaction Protocol that is consistent with the specification [183], but also optimized for verification purposes.

The third objective is to show that the Transaction Protocol satisfies a set of properties, including that it is a faithful refinement of the Transaction Service, for a set of initial configurations, that depend on protocol parameters.

1.3 Scope

The research undertaken for this thesis has involved investigating the applicability of Coloured Petri nets for the analysis of distributed systems. More specifically, Coloured Petri nets were applied to two case studies with the objective of verifying that the system at one level of abstraction (e.g. protocol/design specification) faithfully refined the system at a higher level of abstraction (e.g. service/requirements specification). The two case studies were the WAP Transaction layer and a distributed missile simulator, called the Integrated Weapons Simulator (IWS) [33]. To meet practical limits on the size of this thesis the IWS case study has been omitted. Several results from the analysis of IWS are reported in [51, 52, 53, 58]. Appendix H also includes summaries of these publications.

The focus of this thesis is the verification of the WAP Transaction layer. Therefore, the CPN model of the Transaction Protocol is optimized towards the verification procedure. A considerable amount of effort was spent on developing a Transaction Protocol CPN that was purely for specification purposes [57]. This CPN is not included. Chapter 3 discusses the differences between specification and verification oriented models.

1.4 Structure of the Thesis

The remainder of this thesis consists of the following chapters.

Chapter 2 provides motivation for investigating mobile data services and, more specifically, the WAP Transaction layer. The problems with adapting the existing Internet infrastructure to a mobile environment are presented, and a selection of proposed and/or implemented solutions summarized. Of these solutions, a more detailed treatment is given to the WAP architecture. A survey of work related to the analysis of the Transaction layer is also presented.

Chapter 3 describes a general protocol engineering methodology, which includes several key concepts that arose from the development of the Reference Model for Open Systems Interconnection (OSI) [80]. The tailoring of the methodology to the Transaction layer is also described.

Chapter 4 introduces Coloured Petri nets, the formal technique that is used to model the Transaction Service and Protocol. The analysis techniques used, state space, language/automata and sweep-line, are also described.

Chapter 5 summarizes the specification of the Transaction layer given in [183]. This chapter is meant to accurately reflect what is given in [183], with no attempt to justify the design decisions. In the following chapters, deficiencies in the specification are revealed and discussed.

Chapter 6 discusses where the Transaction Service in [183] is ambiguous or incomplete, and provides a set of modifications for improving the Transaction Service. A CPN model of the Transaction Service is created and used to generate the Transaction Service language, which defines the set of possible global sequences of service primitives. This chapter refines the Transaction Service CPN given in [54].

Chapter 7 gives a detailed description of the Transaction Protocol CPN. This includes discussion of a set of restrictions, simplifications and assumptions made to tailor the CPN for verification, while still accurately representing the Transaction Protocol [183]. An earlier model of the Transaction Protocol was published in [56].

Chapter 8 analyzes the CPN from Chapter 7, revealing errors in the Transaction Protocol given in [183]. Changes to the Transaction Protocol are suggested, leading to a Revised Transaction Protocol.

Chapter 9 presents the state space and language analysis results of a set of initial configurations of the Revised Transaction Protocol CPN. The language analysis compares the Revised Transaction Protocol and Transaction Service languages, showing that the protocol does provide the service. The results, in addition with sweep-line analysis results, are used to conjecture about the dependence of the state space size on the Transaction Protocol parameters.

Chapter 10 concludes this thesis with a summary of the contributions of the research and a list of areas for future work.

Appendix A describes the practical steps in obtaining a service primitive language from a CPN model.

Appendix B gives the state tables from the WTP Specification [183] in full, so that the CPN in Chapter 7 can be validated.

Appendix C lists a summary of statistics obtained from analysing the Transaction Service CPN of Chapter 6.

Appendix D contains supporting material, including analysis results, for the Transaction Protocol CPN of Chapter 7.

Appendix E gives the CPN model of the Revised Transaction Protocol in full, and presents the results for all configurations analysed.

Appendix F lists the modifications to the Revised Transaction Protocol CPN that enable the efficient calculation of results for all configurations.

Appendix G discusses the applicability of the tools and techniques to the verification task, as a guide for other potential and existing users.

Appendix H lists the publications, with abstracts, resulting from the research undertaken during the candidature.

Chapter 2

Mobile Data and WAP

The rapid growth of the Internet and its use has inevitably led to its entrance into the wireless world. Users are (or will be) no longer content with desktop access to the wealth of information provided by computer networks such as the Internet—many desire connectivity while mobile. Some typical mobile data applications include:

- sales people retrieving customer information as they make visits;
- locating nearby services such as restaurants and petrol stations;
- performing secure transactions with your bank; and
- receiving updates of important news or stock prices.

These and other applications represent a large market place. This is illustrated by the use of existing mobile data services:

- *i-mode* [123], a proprietary mobile Internet service in Japan, had 18 million users (15% of the population) in February 2001 [44].
- 50 billion Short Message Service (SMS) text messages were sent on GSM (Global System for Mobile communications [116]) networks worldwide in the first three months of 2001 [63].

This chapter examines the need for new mobile data services, in particular the Wireless Application Protocol (WAP), and why the existing Internet infrastructure is inadequate. Section 2.1 describes the characteristics of the wireless networks and devices used for mobile data, and the problems they present for the Internet protocol suite. Several prominent mobile data services are also described. Section 2.2 introduces WAP and the Wireless Transaction Protocol (WTP) which we investigate in this thesis. Section 2.3 concludes this chapter with a discussion of our motivation for analysing WTP, including a survey of other research into WTP and similar protocols.

2.1 Mobile Data Services

Mobile data services differ from existing Internet services because users communicate with small, often hand-held, devices across wireless networks. Section 2.1.1 lists the characteristics of wireless networks and devices, and Section 2.1.2 describes their impact on the suitability of mobile data services using the existing Internet infrastructure. Section 2.1.3 gives some general solutions and surveys several mobile data services already in use.

2.1.1 Wireless Networks and Devices

The Internet protocols were designed as a method for communicating across different physical networks. These include: Wide Area Networks (WANs) (e.g. Asynchronous Transfer Model (ATM)), Local Area Networks (LANs) (e.g. Ethernet, Fibre Distributed Data Interconnect (FDDI)) and point-to-point connections (such as dial-up over a telephone line). The common assumption that the Internet protocols make about the links in the physical networks is that they are error free. This, and other design assumptions of the Internet protocols, do not hold for wireless networks. The following is a list of characteristics that differentiate wireless networks from their fixed counterparts:

High error rates: Error rates are typically higher in wireless links because of the power loss in transmission and interference (e.g. signals received from multiple paths, rain attenuation).

Low bandwidth: The bandwidth available (link capacity) is lower when compared with wired communications (due to regulations on frequency assignment and tradeoffs with terminal power and error rates). For example, GSM mobile phones can transmit/receive data at a maximum rate of 9.6 kb/s, whereas dial-up Internet connections over a phone line are now mainly 56 kb/s. Although wireless bandwidth is increasing, this resource will remain a limiting factor in comparison with wired connections.

Disconnections: Due to variability of signal strength, there may be times when the wireless device is not connected to the network.

Hand-over: When devices are mobile they may have to move through different cells in a mobile communication network. Each cell has one base station, with which devices communicate. As a device moves from one cell to another, a hand-over of base stations occurs, which should be transparent to the user.

In addition to the network characteristics, the types of devices used in a wireless environment are significantly different from those in fixed networks (i.e. PCs, workstations). They may include mobile phones, personal digital assistants (PDAs), and laptop or palm-top computers. We can typically classify wireless devices as having:

- Short battery life (e.g. usage times of minutes to several hours, standby times of days)
- Small memory
- Less CPU power
- Different input devices (e.g. numeric keypad, stylus)
- Small displays (e.g. several lines, with or without colour/graphics)

The characteristics of wireless devices will tend to limit the types of applications that are developed for mobile data services.

2.1.2 Problems with Existing Internet Infrastructure

The network and device characteristics listed in the preceding section impose limitations on the use of existing Internet protocols and applications [126, 1, 7]. Here we summarize just a few of the problems.

- Transmission Control Protocol (TCP) [133], the reliable transport protocol used in the Internet, assumes that packets lost in the network are due to congestion (a fair assumption in wired networks). Therefore, when errors occur TCP invokes its congestion control mechanism. This effectively reduces the transmission rate. Errors in a wireless link, however, are likely to be due to transmission errors, rather than congestion. When this is the case, reducing the transmission rate will reduce the overall throughput.
- TCP uses a three-way handshake to synchronize the two parties of a connection. For applications making frequent, small connections, this can be a significant and unnecessary overhead in wireless networks. Overheads should be kept to a minimum in wireless networks so the amount of information transferred is reduced.
- TCP and the Internet Protocol (IP) [132] were not designed to handle hand-overs and disconnections. A transport protocol aware of these phenomena may be able to detect, for example, a hand-over and stop transmitting until it is complete.

- The Internet addressing scheme, using IP, allocates addresses to each host, based on the network it is in. Packets are sent to the network and then forwarded to the host. When a mobile host leaves the network, IP has no standard way of forwarding packets to the host in its new (de-facto) network.

- The HyperText Markup Language (HTML) [189] has a large set of tags for describing Web page content. Mobile devices will require a significant amount of processing power and memory to interpret all of these tags, even when many will not be used for mobile data services.

2.1.3 Current and Future Solutions

There are several systems in use that provide mobile data services, either independently of the Internet protocol suite or using part of the Internet infrastructure enhanced for wireless access. The former approach has the disadvantage that the existing Internet resources cannot be utilized. There are a range of techniques for the latter, including:

- Increase TCP efficiency when transmission errors occur by either notifying TCP of the types of errors (e.g. [37]) or hiding the errors from TCP end-points (e.g. [21]). (Surveys of these techniques can be found in [8, 1].)

- Add mobility support to IP so that mobile hosts can be reached while outside their normal (home) networks (e.g. [125, 25, 26]).

- Apply compression techniques and remove redundant information from headers for transmission over the wireless link (e.g. [40]).

- Define new languages for the display of content on wireless devices (e.g. [180, 190, 93, 95]).

The majority of the systems that already provide mobile data services have low data rates and only specify the functionality of the physical, data link and network layers (or equivalent) in the protocol stack (see [145] for a survey of several services). These include:

MOBITEX: A packet data technology with a data rate of up to 8 kb/s [143, 94]. MOBITEX uses a proprietary network layer protocol (called MPAK) on top of existing data link and physical layer protocols. A transport protocol, MTP/1, is also defined, but applications may bypass this protocol and directly use MPAK. Typical applications of MOBITEX networks include [160, 153]: monitoring of vending machines, alarms and personnel movements; transmission of financial information, such as credit card authorizations; and messaging for field sales or emergency situations.

CDPD: (Cellular Digital Packet Data) [144] A cellular based, packet switching overlay network of the Advanced Mobile Phone Service (AMPS) in the United States. IP [132] or the OSI connection-less network protocol (CLNP) [83] can be used at the networking layer to provide data rates up to 19.2 kb/s. CDPD networks have similar applications to MOBITEX.

SMS: (Short Message Service) [124] A messaging service that utilizes the GSM control channel (allowing messages to be transmitted while the voice channel is being used). Messages, which can contain up to 160 characters, can be broadcast to all users in a cell or delivered point-to-point between users. Although the performance of SMS is relatively low (delivery times may be in the order of 10's of seconds or longer), it is extremely popular due mainly to its ease of use and the penetration of GSM phones [63]. Personal communications, content delivery and subscription applications are also common (e.g. delivery of news bulletins).

GPRS: (General Packet Radio Service) [24] A packet switched bearer service of GSM which is destined to take-over from SMS. It provides data rates up to 170 kb/s that is shared based on user demand. GPRS packets from a mobile terminal are switched through a (serving) GPRS support node (SGSN), which can utilize existing network protocols such as IP and CLNP for further transmission. Although few applications utilize GPRS directly (unlike SMS, which applications use directly), other architectures, as we will see shortly for WAP, use GPRS as a network technology.

Two systems that attempt to do more than just enhancing the network layers and layers below are i-mode [123] and WAP [171]. These systems specify enhancements to the transport protocols and the mechanisms for developing mobile data applications. The following brief descriptions of each conclude this section.

i-mode

i-mode [123, 121] (meaning *information mode*) is a mobile data service created and provided by NTT DoCoMo [123], Japan's leading mobile phone operator. Users can access Internet content via their i-mode mobile phones, which are widely available in Japan. In practice, most of the content is restricted to that developed specifically for i-mode using a subset of HTML, called Compact HTML (cHTML) [93]. cHTML allows developers to take advantage of existing HTML applications, but restricts the use of memory and data intensive tags (e.g. frames, tables and image maps cannot be used). There are also limitations on the size of cHTML pages and images.

i-mode uses a packet based overlay of Personal Digital Cellular (PDC) [117], Japan's mobile phone network technology [121]. From the mobile phone to a gateway on the edge

of the mobile phone network and the Internet, a transport protocol called TL, which is standardized by the Association of Radio Industries and Businesses [3], is used. TCP [133] is used from the gateway to web servers on the Internet. Content on a web server is transferred to the mobile phone using the HyperText Transfer Protocol (HTTP) [48]. The maximum data rate offered to users is 9.6 kb/s.

NTT DoCoMo launched i-mode in February 1999. Two years later there were 18 million users in Japan, which comprises 15% of the population [44]. The types of applications specifically designed for i-mode include: bank transactions, telephone and restaurant directories, airline and entertainment reservations, email, news updates, games, karaoke and character downloads. The introduction of a Java [59] programming environment for i-mode handsets will allow further, more interactive applications to be created [112].

WAP

WAP [171] is an architecture that attempts to alleviate the problems of wireless Internet access at different levels of the protocol architecture, from transport layer up to the application layer. WAP applications are tailored towards mobile phones and other hand-held devices. Like i-mode, WAP uses a markup language and transport protocols optimized for the link between the mobile terminal and a WAP gateway situated in the fixed network. The existing Internet infrastructure can be used from the fixed network to the WAP gateway. Although the design principles and targeted applications are similar, different protocols are used in i-mode and WAP. Another major difference is that WAP is more open than i-mode, in that specifications are developed by an industry consortium [170] and made publicly available. WAP has been designed to be a global standard, operating over GSM, CDMA (Code Division Multiple Access) and even PDC mobile phone networks, whereas i-mode is mainly a proprietary Japanese solution[1]. Because of its world-wide applicability, and use in Australia, we have chosen to begin further investigations into WAP. The following section provides more details on the motivation and design of WAP.

2.2 Wireless Application Protocol

2.2.1 Background

The WAP architecture[2] [171] was designed by the WAP Forum [170] to provide Internet and similar services to mobile users. The initial focus was on the delivery of services

[1] This may, however, change with proposed introductions of i-mode into Europe [155] and the United States [75].

[2] Although named "Wireless Application Protocol", WAP is actually an *architecture* that comprises protocols and applications.

over existing wireless networks, particularly GSM mobile phone networks. The types of applications include:

Location-based: traffic reporting, finding events or restaurants.

Financial: stock quotes, banking.

Travel: airline or public transport schedules, hotel bookings.

Enterprise: email, database, information updates.

Commerce: price comparisons, impulse buying.

Informational: news, weather, sports.

The objectives of the design are to take advantage of the existing knowledge and infrastructure in developing Web applications, while using protocols optimized to run over wireless links. The former is achieved by using a programming model similar to that used in the WWW [12]. Figure 2.1, based on Figure 2 in [171], is a good indication of this.

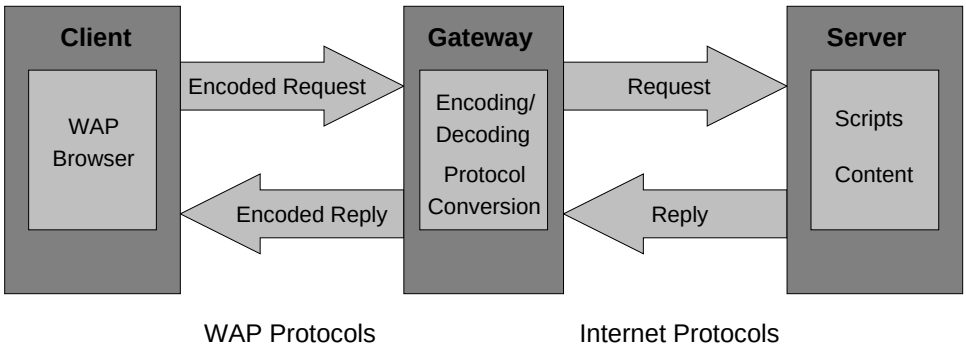

Figure 2.1: WAP programming model

The programming model comprises a client, a gateway and a server. A typical application on the client might submit a request to a server for information. In the Web programming model the request goes directly to the server, which responds to the client with some content. In the WAP programming model the request and response go via a gateway. The gateway is the interface between the wireless Internet and the wired Internet. It may perform encoding/decoding and protocol conversion so that the request and response can be efficiently sent over the wireless link. The scenario in Figure 2.1 shows the WAP protocols operating between the client and gateway, and the standard Internet protocol suite between the gateway and server.

The WAP Forum was founded by Ericsson, Motorola, Nokia, and Unwired Planet (which became Phone.com, and is now Openwave Systems) and released the first public copy of the WAP specifications in April 1998. Since then more than 500 organizations have joined the consortium. They include: device manufacturers, wireless service

providers, software companies, infrastructure providers and content providers. To participate in WAP Forum meetings and have access to the design documents, a membership fee is required. The final specifications are available publicly (via `www.wapforum.org`), and non-members can submit input documents to the WAP Forum with comments or corrections. The remainder of this section will provide further details on the WAP architecture. Throughout the thesis, unless otherwise stated, we refer to an updated version of WAP 1.2, called the *June 2000 Conformance Release* [171]. Recently, a new version of WAP has been released, Version 2.0 [186]. Chapter 10 discusses the impact of this release on the contributions of this thesis.

2.2.2 Architecture

To provide a scalable and extensible architecture, WAP is designed in layers as shown (shaded grey) in Figure 2.2. There is an application layer and four protocol layers: session, transaction, security and transport. The relationship with the OSI layers [80], which are discussed further in Chapter 3, is shown to the left of the architecture. The layers below the transport layer are not part of WAP.

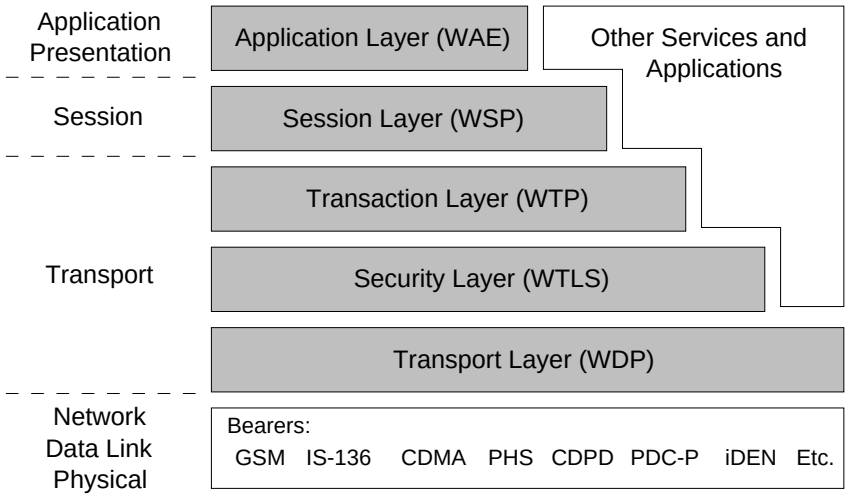

Figure 2.2: WAP architecture

The WAP layers are designed so that other applications and services can also use them. For example, a user application may be written directly on top of the Transport layer. Alternatively, it could use the reliability and integrity provided by the Transaction and Security layers as well.

Below the WAP stack are the wireless bearer services. As WAP has been designed to be a global standard for mobile data services, it must support different bearer services. The bearer services are discussed further in Section 2.2.7.

2.2.3 Application Layer

The application layer is called the Wireless Application Environment (WAE) [178]. It provides a suite of mechanisms and protocols for developing and running WAP applications. These include:

- Wireless Markup Language (WML) [180]: a mark-up language for describing the structure and layout of content. This is analogous to HTML [189] in the WWW. WML content is encoded in a binary tokenized form for transmission over the wireless link. This, and the cut-down but specialized tag set, can improve the efficiency of transmitting to, and displaying on, small wireless devices.

- WMLScript [185]: a scripting language for creating advanced behaviour on the user interface. This is similar to JavaScript [49] in the WWW. WMLScript allows functionality to be implemented on the wireless device, bypassing expensive transmissions to the server.

- Wireless Telephony Applications (WTA) [182]: a framework for making use of the telephony features in the device and network infrastructure. This includes WML and WMLScript interfaces to telephony functions (e.g. dialling numbers in the phonebook), mechanisms for the WAP client detecting telephony events and procedures for ensuring the secure access of the telephony infrastructure.

- User Agent Profiles [175]: mechanisms for describing the features and profiles of the client device (e.g. screen size, colour capabilities), the user (e.g. the browser supported) and the network (e.g. latency, reliability). The information can be used to customize content at the server for different clients.

- Push Architecture [173]: provides a means to transmit information to a client device without an explicit user request. The architecture comprises the client, a push initiator and a push proxy gateway that: may provide client discovery services; efficient encoding and protocol conversion; and ensures the push initiator is authorized to deliver content to the client.

2.2.4 Session Layer

The Wireless Session Protocol (WSP) [181] provides client/server communication in two modes: connection-less and connection-oriented.

Connection-less: a non-confirmed request to, response from, or push from, the server. This uses the Transport service (Section 2.2.7) and is useful for applications that sometimes require communication without the overhead of session setup and tear-down.

Connection-oriented: a session is created between a client and a server, and within it reliable exchange of content can occur. The Class 2 Transaction service (Section 2.2.5) is often used in this mode. Features include: the ability to negotiate capabilities on connection setup; a push service within the context of a session; and a lightweight session tear-down (suspend) and setup (resume) that allows a session to be maintained over different bearer networks.

The connection-oriented service design is based on HTTP/1.1 [48]. Method requests and responses are made between the client and server, content types can be specified and negotiated and composite objects can be transmitted.

2.2.5 Transaction Layer

The Wireless Transaction Protocol (WTP) [183] provides communication between an entity that initiates a transaction by making a request (the Initiator) and an entity that receives, and optionally responds to the request (the Responder). There are three classes of service:

Class 0: an unreliable one-way request from Initiator to Responder. This is intended to be used within the context of a session when a push is required. The direct use of the Transport protocol (Section 2.2.7) by a user is more efficient and should be used instead if unreliable one-way requests are common.

Class 1: a reliable one-way request from Initiator to Responder. The receipt of the request by the Responder must be acknowledged to the Initiator. This realizes the reliable push service in WSP.

Class 2: a reliable request and reliable response between Initiator and Responder. The request is acknowledged by the Responder and, likewise, the receipt of the response at the Initiator is acknowledged to the Responder. This provides reliable transaction-oriented communication between two endpoints.

The focus of this thesis is WTP Class 2. More details on the service and protocol features of WTP Class 2 are given in Chapter 5.

2.2.6 Security Layer

The aim of the Security layer (Wireless Transport Layer Security (WTLS) [184]) is to provide applications with privacy, data integrity and authentication. To do this, WTLS specifies procedures for setting up a secure transport connection and guidelines for the types of security algorithms that can be used (e.g. Data Encryption Standard (DES) [11]

for bulk encryption, Rivest, Shamir, and Adleman public-key cryptosystem (RSA) [139] or Elliptic Curve Diffie-Hellman (ECDH) [108] for authentication). WTLS is an optional layer and has a one-to-one mapping of service primitives to Transport service primitives. This allows the users (e.g. WTP) to operate in the same manner, whether the security layer is used or not.

2.2.7 Transport Layer

The Transport layer aims to quarantine the upper layers from the differences inherent in the supported bearer services. The bearers include:

Cellular systems: GSM [116], ANSI-136 [156], PDC [117], CDMA [50], CDPD [144]

Cordless phones: Digital Enhanced Cordless Telecommunications (DECT) [45], Personal Handyphone System (Japan) (PHS) [129, 130]

Trunked radio: Terrestrial Trunked Radio Access (TETRA) [46]

Paging: Flex/ReFlex [114]

Proprietary: DataTAC [38], iDEN [115]

These bearers have different characteristics. For example, the maximum round trip time for the GSM SMS maybe 40 seconds, whereas for bearers supporting IP the round trip time could be 3 seconds [183]. The WAP layers have to be designed to cope with these different characteristics. The Transport layer protocol (called Wireless Datagram Protocol (WDP) [179]) provides an unreliable datagram service. For bearers that support IP (e.g. GPRS and CDPD), the User Datagram Protocol (UDP) [131] is used. For all other bearers (including GSM SMS and CDMA Circuit Switched Data (CSD)), WDP specifies a mapping from the bearer to the datagram service.

2.3 Wireless Transaction Protocol

The fundamental reason for verifying WTP is to gain confidence that the design specification is correct and unambiguous. This is a good step towards ensuring different implementations of WTP can inter-operate and behave as expected. The correct operation of WTP is vital given that implementations are already available and being used for many applications [3]. This section surveys previous work on analysing WTP and similar

[3] An argument may be made that since implementations are available there is no point in verifying the design. This can be refuted by the fact that the design/implementation is an iterative process, illustrated by the regular updates made of the WAP standards, e.g. versions 1.0, 1.1, 1.2 and so on [172, 174, 187].

protocols.

The only published research on formally analysing WTP (or analysis by any means) we are aware of is [74]. Transaction Class 1 is modelled as a timed automaton and state-graph manipulators [73] (reduction and composition techniques that can be applied on state spaces) are used to investigate the real-time properties of WTP. The focus of [74] is on how different manipulators reduce the state space when the values of time-out intervals are varied. There are no results presented regarding the comparison of the protocol to the service or deadlocks and livelocks in the protocol.

Within the WAP Forum, there was an attempt at formally analysing WTP using SPIN [71], although it was incomplete [2]. There were no results reported.

Although WTP is a relatively new protocol (first released in 1998 [172]), it has similarities with other transaction and transport protocols, the most notable being that they provide end-to-end reliability. There are several attempts at formally analysing various transport protocols, which are oriented towards data transfer as opposed to short transactions. They include: TCP [133] using high-level Petri nets [39, 64, 107]; the OSI Transport Service [82] and Protocol [81] using Petri nets [91, 10, 13]; and the Xpress Transport Protocol (XTP) [191] using Estelle [22, 29], Temporal Logic of Actions [68], and systems of communicating machines [105].

Transaction/TCP (T/TCP) [20] is an extension of TCP that provides an efficient transaction service in the Internet. T/TCP has been specified using timed and untimed automaton models [149], and demonstrated to not provide the same service as TCP. Follow on work [150] has shown the dependence of T/TCP on accurate clocks for transaction protocols to provide efficient, reliable transactions.

Although some features of other transport and transaction protocols are similar to those of WTP, there are substantial differences between them and the Transaction Service and Protocol to warrant the formal verification of WTP.

We have chosen to analyse Transaction Class 2 because it is the basis of many applications (e.g. WSP uses it for browsing functionality) and it is significantly more complex than Class 0 or 1.

Finally, transactions are a common form of communication in computer networks. There are other architectures and protocols that make use of transactions (e.g. the Internet Open Trading Protocol (IOTP) [23], Session Initiation Protocol (SIP) [65], T/TCP [20]). Analysing WTP may lead to results that can be applied to transaction mechanisms in general. The generalisation of our results is discussed in Chapter 10.

Chapter 3

Protocol Engineering

The basic systems engineering approach of incrementally refining the most abstract representation of a system (e.g. user requirements) until a concrete or target implementation is obtained, can be applied in the design of communication protocols (e.g. the Wireless Transaction Protocol (WTP)). When formal methods (i.e. those based on mathematics) are used in the design, the approach is called *protocol engineering* [103, 14]. This chapter describes a general protocol engineering methodology [18], which is the basis for the methodology used in verifying the WAP Transaction layer.

The basics of designing communication systems, that is building a layered architecture, are described in Section 3.1. Section 3.2 explains the process and activities involved in protocol engineering. The role of formal methods is also discussed. Section 3.3 concludes this chapter by outlining the approach we have used in the remainder of this thesis.

3.1 Layered Communication Architectures

The tasks involved for communicating between entities in a distributed system are, like most complex systems, organised into a layered (or hierarchical) structure. This is called a *communication architecture*. Each layer uses the services of the layers beneath to perform functions, and together the functions provide a service to the layer above. The functions of a layer may be defined by a communication protocol. The two main advantages of using a layered structure are [154, 70]: the complex system can be partitioned into smaller subsystems that are easier to comprehend and design (i.e. divide and conquer); and layers can be designed and built independently of other layers, which simplifies maintenance, allows competitive solutions, and can simplify the standardization process.

A well known communications architecture is the Reference Model for Open Systems Interconnection (OSI) [80]. OSI, for which the layered architecture is shown in Figure 3.1 (based on Figure 11 of [80]), was developed jointly by the International Organization for

Standardization (ISO) and the Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T). This section introduces the central concepts of OSI.

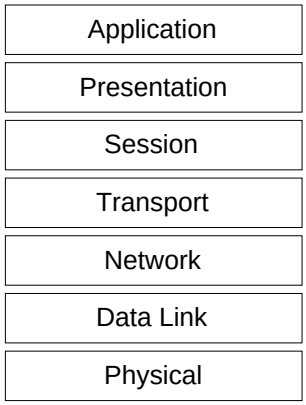

Figure 3.1: Reference Model for Open Systems Interconnection (OSI)

3.1.1 Open Systems Interconnection

OSI is a seven-layer architecture, ranging from the physical layer up to the application layer (Figure 3.1). As seen in Chapter 2 (Figure 2.2), WAP is a five-layer communications architecture. The functions performed by each layer in the OSI Reference Model include:

Application: Manage the transfer of documents, messages and files between applications.

Presentation: Negotiate the syntax and semantics of information sent by applications, including the encoding of data into standardized forms.

Session: Allow end hosts to establish a session between each other for ordinary transport of data and enhanced services such as dialogue and synchronization control (e.g. re-starting a transport connection after a crash).

Transport: Ensure end-to-end delivery of packets from the source to destination.

Network: Establish mechanisms (including addressing, routing and congestion control) for transmitting packets from the source to destination.

Data Link: Define frames of data, and control their transmission (including rate of flow and errors) across a single link.

Physical: Define mechanical and electrical interfaces to the physical network, including the modulation and coding schemes to be used.

The basic structure of a layered communications architecture is shown in Figure 3.2 (which is based partly on Figures 4, 5 and 7 in [80]). Each layer comprises entities that perform functions within the layer. The capabilities provided by the entities in the (N)-layer (and all layers below) at the boundary between the (N)-layer and (N+1)-layer is called the *(N)-service*. The (N+1)-entities access the (N)-service by *(N)-service-access-points*. Therefore, the logical path for exchange of information is vertically, via service-access-points (SAPs). However, virtual communication occurs between peer (N)-entities via a (N)-protocol. The (N)-protocol is discussed in Section 3.1.3.

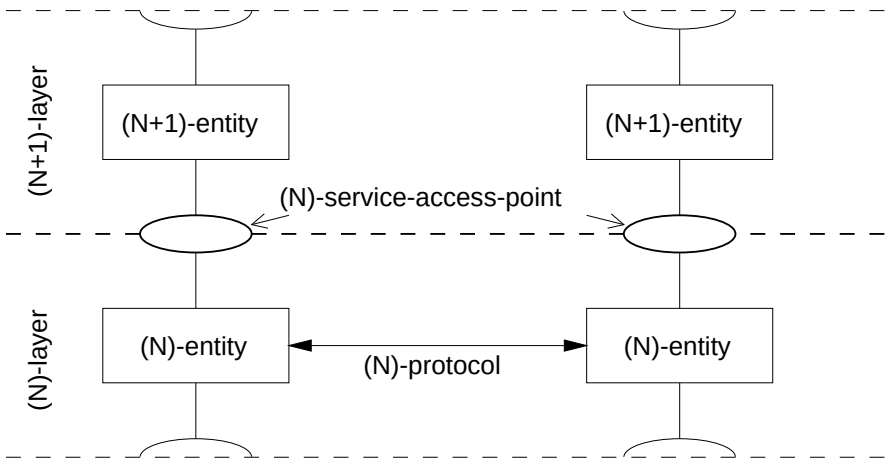

Figure 3.2: Layers in a communications architecture

3.1.2 Service Definition

The advantage of designing and building layers independently of other layers is only possible when the boundaries between layers (i.e. the services) are well defined [166]. In defining the (N)-service, an abstraction of the layered communication architecture, as illustrated in Figure 3.3 (which is based partly on Figure 1 in [79]), can be made.

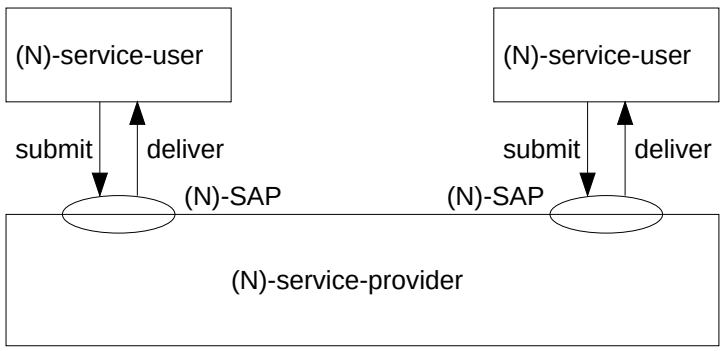

Figure 3.3: Abstraction of the (N)-service in a communications architecture

In Figure 3.3, the (N+1)-entities are designated (N)-service-users. The (N)-service-provider represents all entities in the (N)-layer and all layers below that provide capabilities to the (N)-service-users. Therefore, the definition of the (N)-service requires the interactions possible between the (N)-service-users and (N)-service-provider to be defined. The conventions for defining OSI services are given in [79]. The interactions between users and provider are described using *service primitives*.

A service primitive defines data that is issued by either the user or provider. The information conveyed by the primitive (i.e. the purpose, the conditions the user or provider has met to issue the primitive, the expectations on the receiver) is given by a primitive name, a primitive type and the parameters of the primitive. The primitive name identifies the capability the primitive is providing. For example, the Transaction Service in Chapter 5 includes the primitive name *Invoke* which is used to invoke a transaction. The user that initiates the service facility is called the *requestor*, and the peer user for that facility is called the *acceptor*. There are four primitive types:

1. *request*: submitted by the requestor user, initiating the use of the service facility;
2. *indication*: delivered to the acceptor user, conveying information from the request;
3. *response*: submitted by the acceptor user, responding to the indication; and
4. *confirm*: delivered to the requestor user, conveying information from the response.

A sequence of primitive types may be [request,indication,response,confirm], or a non-confirmed service may only use the [request,indication] sequence. In special circumstances (e.g. an abort or disconnect issued by the service provider) only an indication primitive type may be used. The four primitive types may be abbreviated to *req*, *ind*, *res* and *cnf*, respectively.

The primitive parameters represent user data or control information, and may be passed between primitive types (e.g. the values of the parameters of an indication primitive are set to the same values of those in the corresponding request primitive). The Transaction Service in Chapter 5 includes parameters such as Source and Destination Addresses, Abort Codes and User Data. Primitives are structured as:

$$< servicename > - < primitivename > . < primitivetype >$$

For example, the Transaction Service has a primitive denoted as: *TR-Invoke.request*, where *TR* is the abbreviation of the Transaction Service used in WAP. Parameters may be given as a tuple following the primitive when necessary.

The definition of a service must specify all possible interactions of primitives. For example, a possible interaction in the Transaction Service may be for one user to submit a

TR-Invoke.req primitive, resulting in a TR-Invoke.ind primitive (with the same parameter values as the TR-Invoke.req primitive) being delivered to the peer user. As a result, a service definition specifies the possible set of global primitive sequences.

Once the (N)-service is defined, the (N+1)-entities can make use if it, without any knowledge of the details of the (N)-protocol.

3.1.3 Protocol Definition

The (N)-protocol is used for communicating between peer (N)-entities (Figure 3.2). Figure 3.4, based partly on Figures 4, 5 and 7 in [80], shows the virtual path of information is directly between peer (N)-entities (referred to as (N)-protocol-entities). The information is conveyed between peer (N)-protocol-entities in *protocol data units* (PDUs). As the (N)-protocol-entities use the (N-1)-service, the logical path of information passes through the (N-1)-service-provider.

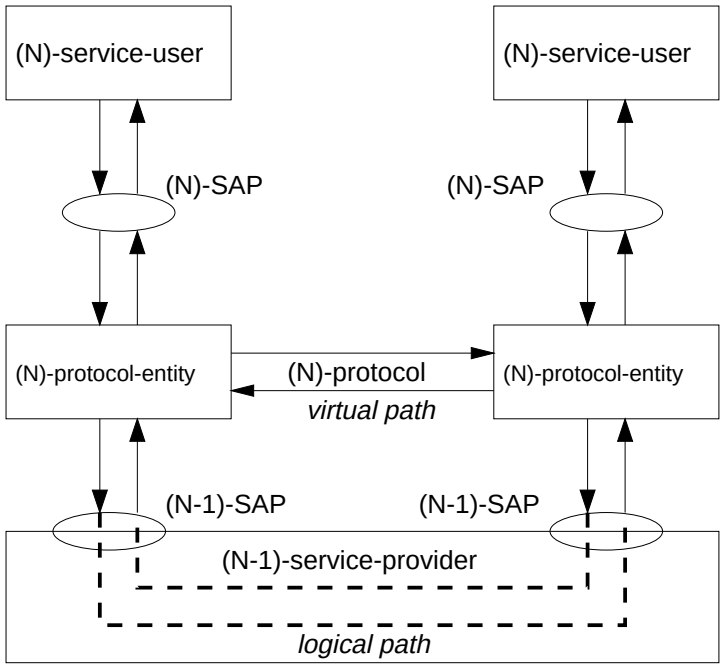

Figure 3.4: Logical and virtual paths used by the (N)-protocol in a communications architecture

A PDU is encapsulated in a *service data unit* (SDU) when transferred along the logical information path. In Figure 3.4, PDUs used by the (N)-protocol are encapsulated in SDUs that are transferred, but not interpreted, by the (N-1)-service-provider. The (N)-protocol-entities, therefore, communicate by sending and receiving PDUs to each other, and together provide a service to the (N+1)-layer (or (N)-service-users).

The definition of a protocol must specify the encoding of PDUs and the rules for exchanging the PDUs. The encoding of a PDU comprises the definition of a header, which conveys control information between (N)-protocol-entities, and the user data. The

rules define the procedures for providing the desired service of the protocol. An aim of protocol engineering is to design unambiguous protocols, which involves verifying that the protocol provides the intended service. The next section details the process and activities involved in protocol engineering.

3.2 Protocol Engineering Methodology

Protocol engineering involves the application of formal methods to the design of communication protocols. Section 3.2.1 describes the general process, and Section 3.2.2 points out the major design activities in the process. Section 3.2.3 gives an overview of formal methods and their application to protocols. This section is only a brief introduction to the field of protocol engineering. For further information we refer the reader to other survey papers [103, 14], articles in special issues of journals [141, 152, 104] and proceedings of conferences [135] that focus on protocol engineering.

3.2.1 Protocol Engineering Process

The protocol engineering process follows the top down methodology shown in Figure 3.5.

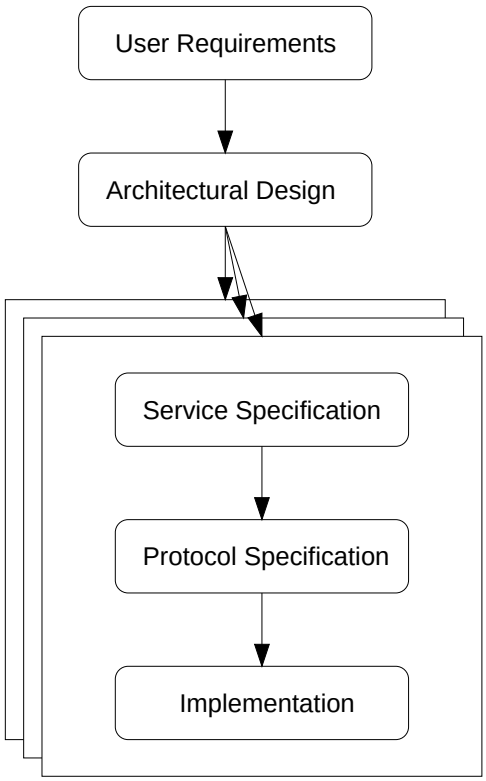

Figure 3.5: Steps of the protocol engineering process

As with any system, the requirements of the user drive its design and implementation. For communication architectures, user requirements come from the types of applications

that are intended for use in the architecture. The first step is, therefore, to gather and document user requirements.

Once the user requirements have been identified, the communications architecture is designed. As discussed in Section 3.1, the architecture is partitioned into layers so that the complex system for providing the distributed applications required by the users is easier to design and build.

Each layer in the communications architecture is then designed separately. Figure 3.5 shows this by the multiple boxes (one for each layer). The process applied to each layer is: define the service to be provided by the layer (Section 3.1.2); design a protocol (or sometimes a class of protocols) that will provide the service (Section 3.1.3); and generate a target implementation from the service and protocol specifications. Once implementations of each layer are obtained, the communications architecture has been built and designed. Section 3.2.2 describes the activities that are performed during the protocol engineering process to ensure the communications architecture implemented meets the requirements of the user.

3.2.2 Protocol Engineering Activities

The first activity in the protocol engineering process is gathering and documenting the user requirements. There are no well-defined rules or procedures for this step that particularly apply for protocol engineering. However, techniques for capturing and analysing user requirements from other domains (e.g. [151]) can be applied.

The design of the communications architecture is based on the environment that applications are to be used in. In many cases, existing architectures or protocols can be re-used (or at least the principles behind their design can be). For example, the OSI Reference Model [80] provides a well-defined partition of functionality which an architecture can be based upon, even if the OSI protocols are not used. Similarly, the Internet architecture [35, 154], although not defined as clearly as OSI, is a starting point for new communication architectures.

The activities applied for each layer are shown in Figure 3.6, which is based on Figure 1 in [14].

The steps for defining the service are given in Section 3.1.2. A definition of the protocol to provide the service is then specified (Section 3.1.3). In protocol engineering these service and protocol specifications are formally defined. Section 3.2.3 introduces suitable formal techniques. An important protocol engineering activity is to ensure the protocol does indeed provide the specified service. There are two approaches to doing this:

1. Apply a synthesis technique on the service, automatically generating a conformant

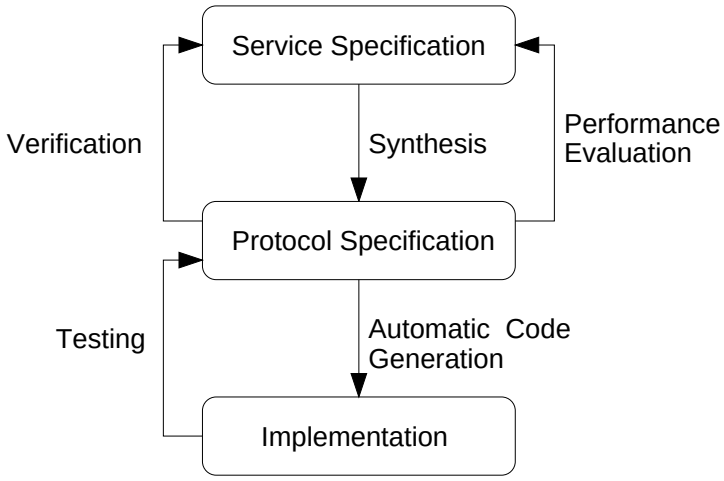

Figure 3.6: Protocol engineering design activities for each layer

protocol (or components of the protocol). A survey of different protocol synthesis techniques can be found in [142]. Several examples of their application are in [91, 62, 119, 92]. Protocol synthesis is not applicable for existing protocol specifications.

2. Verify the protocol provides the service, in terms of the set of global primitive sequences, by comparing the sequences generated by the two specifications [70]. The advantage of this approach is that the protocol can be designed without the need for complex formal rules that refine the service. It can also be applied on existing protocols, that have not yet been verified. The major disadvantage is the computational complexity in verifying the protocol against the service. This is discussed more in Chapter 4.

These two approaches are complimentary: synthesis can be used to obtain some components of the protocol (or a skeleton of the protocol design), and then after the remainder of the protocol is designed, it can be verified against the service.

It is insufficient for only the functional behaviour of the protocol design to be investigated. The protocol must also meet a set of performance criteria (usually specified in the service). Performance evaluation of the protocol design can be performed using analytical, experimental or simulation techniques [140, 67].

The design of the service and protocol is an iterative process. For example, verification may identify errors in the service or protocol that need fixing. Similarly, performance evaluation may result in changes to the protocol, and hence verification must be performed again. It is not expected that iterations will not be required in the protocol engineering methodology—the purpose of the methodology is to minimize the number of iterations, especially those which occur across multiple steps in the process in Figure 3.5 (e.g. errors in the architecture or service identified as a result of testing the implementation).

Once the protocol has met its behavioral and performance requirements, an implemen-

tation is produced, preferably automatically from the protocol specification [98, 28, 61]. When the code is not automatically generated, conformance testing must be performed [122, 102, 168]. Test sequences may be generated from the protocol specification. The testing process should also evaluate the performance of the implementation against the specification.

Another activity that is part of the protocol engineering methodology, but not shown in Figures 3.5 and 3.6, is maintenance. Once the communication protocols are in use, undetected errors and the changing environment will require steps of the protocol engineering process to be re-visited. This is expected, and where possible, specifications should be designed with maintenance in mind.

3.2.3 Formal Methods

To obtain complete and unambiguous specifications in the design stages of protocol engineering, formal methods can be used. Formal methods [27, 167] are those which are based on mathematics. This means there is no room for mis-interpretation of the formal specifications. Different interpretations can lead to implementations that do not inter-operate, thereby not providing the intended service to the distributed applications.

A major advantage of using formal methods in protocol engineering is the ability to formally reason about the properties of a protocol, including verifying it against the service. A set of formal analysis techniques are available that take advantage of the mathematical foundation of the model. These include state space analysis, invariants analysis, language theory and proofs [87, 163, 72]. As we will show in Section 3.3, state space analysis, where basically all states of the system are explored (see Chapter 4), and language theory (for comparing models at different levels of abstraction) are the analysis techniques of choice. Formal methods also assist in the other protocol engineering activities.

There are several other features that formal methods must possess, in order for them to be suitable for protocol engineering [14]. They include:

- intuitive modelling of concepts inherent in communication protocols, most notably concurrency and non-determinism,
- the ability to specify systems at different levels of abstraction (e.g. service and protocol), and
- adequate support from computer tools.

A range of formal methods have been proposed as suitable for protocol engineering, including: finite state machines [84, 76, 97], net theory [136, 15, 188], programming

languages [120], process algebras [110, 69, 77], temporal logic [32, 60] and predicate logic [148]. A comparison of the benefits of these and other formal methods is out of the scope of this thesis and can be found elsewhere [14, 135]. We use Petri nets [118, 136, 127], and more specifically, Coloured Petri nets [86, 87, 88], for the modelling and analysis of the Transaction layer. The choice of Petri nets is made because:

- They support the desirable features previously described.
- They have been used to model and analyse a number of communication protocols including signalling protocols [165], link-layer protocols [147, 66] and transport protocols [13, 64, 107, 91, 99]. (Further examples can be found in [16] and several conference proceedings [135, 134].)
- Our research group [36] has considerable experience with Petri nets (e.g. [17, 47, 165, 51, 85, 158]).

Coloured Petri nets are particularly applicable for the verification methodology we use in this thesis.

3.3 Verification Methodology

The protocol engineering methodology in Section 3.2 applies to the whole life cycle of communication protocols. The objective of this thesis is to verify that the WAP Transaction Protocol refines the Transaction Service. This section, therefore, details the process and activities applied in this thesis. The verification methodology used comes from [18].

The first two steps in Figure 3.5, gathering the user requirements and designing the communications architecture, have been performed by the WAP Forum. The WAP architecture [171] is described in Chapter 2. We assume the architectural design is adequate.

The Transaction layer is specified in [183]. A detailed description is given in Chapter 5. In summary, the service and protocol are informally specified in [183] using narrative descriptions and state tables. (Although state tables, with a definition of their interpretation, may be a formal model, we illustrate in Chapter 7 that the state tables given are ambiguous and incomplete, and therefore not a formal model.) The first step is, therefore, to create formal models of the service and protocol. The verification of the protocol is then required. All other activities shown in Figure 3.6 are out of the scope of this thesis. Performance evaluation of the Transaction Protocol is a vital part of future work. Although implementations are in use, generating an implementation from the formal specification would be of use, especially in conformance testing of other implementations [176, 157].

The verification process begins by creating a formal model of the Transaction Service. Figure 3.7 shows the process for modelling and analysing the service. The existing information provided in the WTP Specification [183] is used to develop the Service Model. As a part of future work, the Service Model may be used to generate programming interfaces for the implementation. The Service Model is used to calculate the Service State Space, which is in turn used to calculate the Service Language. The Service Language specifies the set of global possible primitive sequences, and is used as the basis for comparison with the Protocol Language. Chapter 6 describes the modelling and analysis of the Transaction Service.

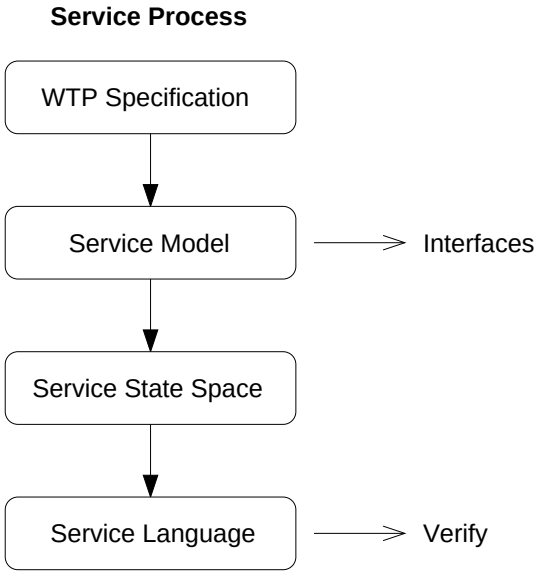

Figure 3.7: Modelling and analysis steps for the Transaction Service

Figure 3.8 shows the process for modelling and analysing the Transaction Protocol. The WTP Specification [183] is used to create the Protocol Model. There are two types of Protocol Model shown: Specification and Analysis. The ideal approach is, firstly, to create a Specification Model which defines the complete protocol. This model gives a concise description of the protocol and can be used to generate implementations, either manually or automatically. For analysis purposes, the Specification Model often contains redundant information that will increase the complexity of the analysis. Therefore, an Analysis Model is created from the Specification Model. The Analysis Model used depends on the properties being investigated. For example, different Analysis Models would be used for verification purposes and performance evaluation. Also, to cope with the inherent complexity of the protocol, the analysis may be performed incrementally, resulting in several Analysis Models (e.g. first omitting transmission errors, and then including them). In this thesis we do not describe a Specification Model (although considerable effort has been spent on developing one), but instead focus on the Analysis Model. There are two reasons for this: the state tables [183], with several modifications, provide a good basis

for the specification; and to keep this thesis within practical size limits. The Analysis Model is created by defining a set of restrictions and simplifications of the specification. These are justified in Chapter 7.

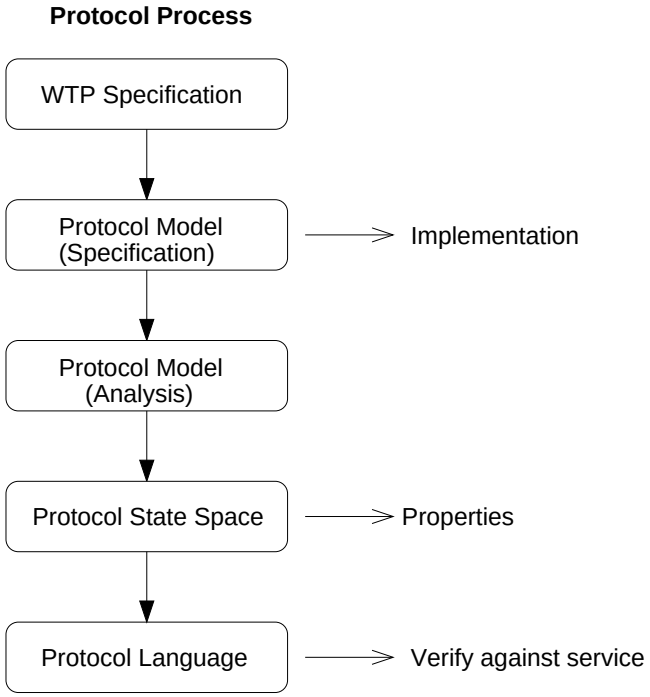

Figure 3.8: Modelling and analysis steps for the Transaction Protocol

From the Analysis Protocol Model, the Protocol State Space can be calculated. Several logical properties (e.g. successful termination, absence of live-locks) can be investigated from the Protocol State Space. The Protocol State Space can then be used to calculate the Protocol Language which is compared with the Service Language to determine if the Transaction Protocol provides the set of sequences defined by the Transaction Service. Chapter 4 describes the properties and verification techniques in detail.

The steps shown in Figures 3.7 and 3.8 are repeated in several iterations. For the Transaction Service, we only describe the final model and results (Chapter 6). An initial model of the Transaction Protocol is presented in Chapter 7. Analysis of this model reveals several errors in the protocol. These errors, and suggested fixes, are discussed in Chapter 8. Chapter 9 presents the verification results of a revised model of the Transaction Protocol.

Chapter 4

Coloured Petri Nets

In Chapter 3 we explained how formal methods can be used to perform rigorous analysis of systems. Petri nets are a formal method that are well suited to the analysis of distributed systems because of their ability to express concurrency, non-determinism and system concepts at different levels of abstraction [14]. In this chapter, a tutorial style introduction is given to Coloured Petri nets which are an extended version of Petri nets. The aim is to provide enough practical details to understand the Coloured Petri net modelling and analysis in the remainder of this thesis. Section 4.1 provides pointers to background material on Petri nets. Section 4.2 describes, using a simple example, the main features of Coloured Petri nets. Relevant analysis methods and computer tools are described in Sections 4.3 and 4.4, respectively.

4.1 Petri Nets

Petri nets (PNs) were devised by Carl Adam Petri in the early sixties [128]. Since then they have evolved into different forms, which have been used to model and analyse a variety of systems. Introductory publications on PNs (e.g. [136, 127, 41, 118]), their variants (e.g. [86, 106, 169, 137]), for which a standardisation effort is in progress [78], and the many applications (e.g. [134, 42, 88, 16, 138]) are widely available. Coloured Petri nets (CPNs) [86, 87, 88] are one variant of PNs aimed at making it easier to model complex systems.

Jensen [86] likened the connection between PNs and CPNs to that between an assembly language (i.e. machine instructions) and high-level programming languages. The computational power is the same but the high-level language is usually easier to use and provides a more compact representation. In theory, CPNs can be transformed into (infinite) PNs (although, in practice, this step is usually not required), giving them the same computational power. However with CPNs, a system can be modelled in a manner that is much more convenient for a human user.

4.2 Coloured Petri Nets

Coloured Petri nets (CPNs), like ordinary Petri nets, have a graphical form that is based on an underlying mathematical definition (see [86]). For our case study, it is sufficient that we describe only the graphical form. Section 4.2.1 describes an example system, which is then used to demonstrate a CPN's structure (Section 4.2.2) and dynamic behaviour (Section 4.2.3).

4.2.1 Example System

We will use the example given in Figure 4.1 throughout this chapter. This CPN models a simplified book borrowing process with the following characteristics:

- library users, identified by a name and the number of books they have already borrowed, can borrow books or place them on hold;
- to borrow a book, there must be a librarian present, and the user must not go over the borrowing limit (e.g. four books);
- the user can place a book on hold (if its available) without assistance from the librarian and there are no limits on the number of books they can have on hold; and
- we keep track of the books that are available for borrowing, have been borrowed or are on hold.

Other features, such as collecting a book on hold, returning books etc. are not modelled as the purpose of the model is just to demonstrate the major features and concepts of CPNs.

4.2.2 Structure of a CPN

CPNs are directed graphs with two types of nodes: *places* and *transitions*. Directed arcs can only be between nodes of different types. We refer to an arc from a place to a transition as an *input arc* and from transition to a place as an *output arc*. In Figure 4.1, and in all CPNs, nodes are shown as ellipses and transitions as rectangles. In the example of Figure 4.1 there are five places and two transitions. All nodes have a name. In our example the name is given inside the node (e.g. place Employees, transition Borrow).

Places are typed by a *colour set*. We show the colour sets in italics next to the place. For example, the place Out has a colour set Books. The colour set is defined in the declarations, which are given in the box below the CPN in Figure 4.1. The colour set determines the type of values that can mark a place (*type* and *colour set* are used interchangeably).

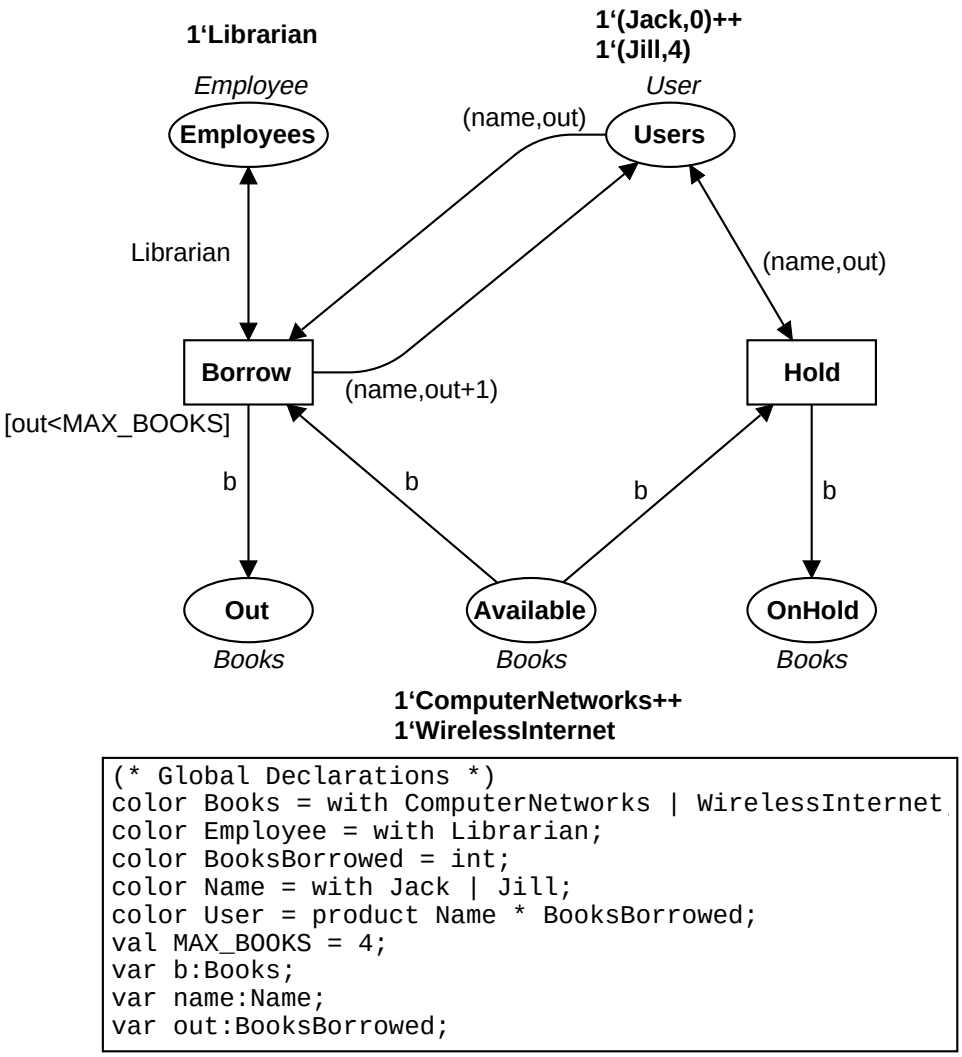

Figure 4.1: Example CPN of a book borrowing procedure

These values are called *tokens*. The collection of tokens on a place is called its *marking* (e.g. the marking of Available is 1‘ComputerNetworks++1‘WirelessInternet, which indicates that one ComputerNetworks and one WirelessInternet token is in the place), and the marking of the CPN comprises the markings of all places. We often use M to denote a marking of a CPN, and M(p) for the marking of a particular place p.

There are five colour (color) sets in the declarations. The first three, Books, Employee and Name, are enumerated types of different coloured tokens. For example, the colour set Books consists of two colours: ComputerNetworks and WirelessInternet. Places typed as Books can contain tokens of either of these colours. Place Available has one token of each in Figure 4.1.

The fourth colour set, BooksBorrowed, is the set of integers (type int), used to indicate the number of books borrowed. The fifth and final colour set, User, is a pair where the first element must be a value from the colour set Name and the second from the colour set BooksBorrowed. The marking of place Users, i.e. 1‘(Jack,0)++1‘(Jill,4), indicates

that there is a user named Jack who has not borrowed a book, and a user named Jill who has borrowed 4 books. The remaining declarations in Figure 4.1 are a constant (MAX_BOOKS) and three variables (b, name and out). Other components of a CPN are the inscriptions on arcs (given next to the corresponding arc, e.g. (name,out) on the arc from Users to Borrow) and transitions (given in square brackets next to the transition, e.g. [out<MAX_BOOKS] for transition Borrow). These are best explained by examining the dynamic behaviour of a CPN.

4.2.3 Dynamic Behaviour of a CPN

The execution of a CPN consists of occurrence (firings) of transitions. A transition can occur if it is *enabled*, and it is enabled when the following conditions are true:

1. for all the input places, enough tokens exist that satisfy the input arc inscription, and

2. the transition inscription, or *guard*, evaluates to true.

For example, transition Borrow has three input places, Employees, Users and Available. (Note that the double headed arc with one inscription is identical to an input arc and an output arc with the same inscription.) The input arc from Employees requires a Librarian token in the place, which is true (given by 1‘Librarian above the place).

The input arc from Users has a pair with two variables as the inscription. This requires a token in Users such that the variables can be bound to values. There are two bindings of the variables that satisfy the inscription: name bound to Jack and out bound to 0; and name bound to Jill and out bound to 4. The input arc from Available has the variable b as an inscription, which can be bound to ComputerNetworks or WirelessInternet, as it is typed by Books (see the declarations in Figure 4.1). The marking of place Available satisfies this requirement.

Variables are local to a transition, in that, if we have the same variable, a, on any input arcs to a transition, then a must be bound to the same value in each arc expression. The variable must also take the same value in the guard and output arcs.

The first item of the enabling condition has been satisfied. To be enabled, the guard on Borrow, out<MAX_BOOKS, must also evaluate to true. The input arc from Users can be evaluated as (Jack,0) or (Jill,4). But only (Jack,0) will satisfy the guard because for (Jill,4), out is bound to 4 which is not less than MAX_BOOKS (also 4). Therefore, the transition Borrow is enabled with the following values bound to the variables, which from herein we refer to as *binding elements* (i.e. the variables are bound to values) when associated with the transition:

BE1: Borrow: {name=Jack,out=0,b=ComputerNetworks}

BE2: Borrow: {name=Jack,out=0,b=WirelessInternet}

It can also be seen that in the initial marking the CPN transition Hold is enabled for the following binding elements:

BE3: Hold: {name=Jack,out=0,b=ComputerNetworks}

BE4: Hold: {name=Jill,out=4,b=ComputerNetworks}

BE5: Hold: {name=Jack,out=0,b=WirelessInternet}

BE6: Hold: {name=Jill,out=4,b=WirelessInternet}

The enabling of these transitions illustrates several important concepts of CPNs. Firstly, there is *non-determinism* present in the model. For example, it is not specified whether b should be bound to ComputerNetworks or WirelessInternet. Either case is possible. Also, as we have six enabled binding elements for the initial marking, they are either enabled *concurrently* or in *conflict*. Concurrently enabled binding elements are those that can occur at the same time. Two binding elements in conflict are enabled at the same time but are not concurrently enabled. We will shortly see examples of each.

When a transition occurs, tokens required by the input arcs are removed from the input places, and the evaluation of the expression on the output arcs (for a given binding element) give the tokens to be added to the output places. If the first binding element for Borrow occurred, then we have the following changes:

- one Librarian token removed from Employees;
- one (Jack,0) token removed from Users;
- one ComputerNetworks token removed from Available;
- one Librarian token added to Employees;
- one (Jack,1) token added to Users; and
- one ComputerNetworks token added to Out.

Figure 4.2 shows this second marking. The markings of the places are now shown in boxes (this is the convention used by Design/CPN, a tool which is introduced in Section 4.4.1). The encircled number indicates the total number of tokens on that place.

The binding elements concurrently enabled in the initial marking are: (1,6); (2,4); (3,6); and (4,5). The binding elements in conflict are: (1,2); (1,3); (1,4); (1,5); (2,3); (2,5); (2,6); (3,4); (3,5); (4,6); and (5,6).

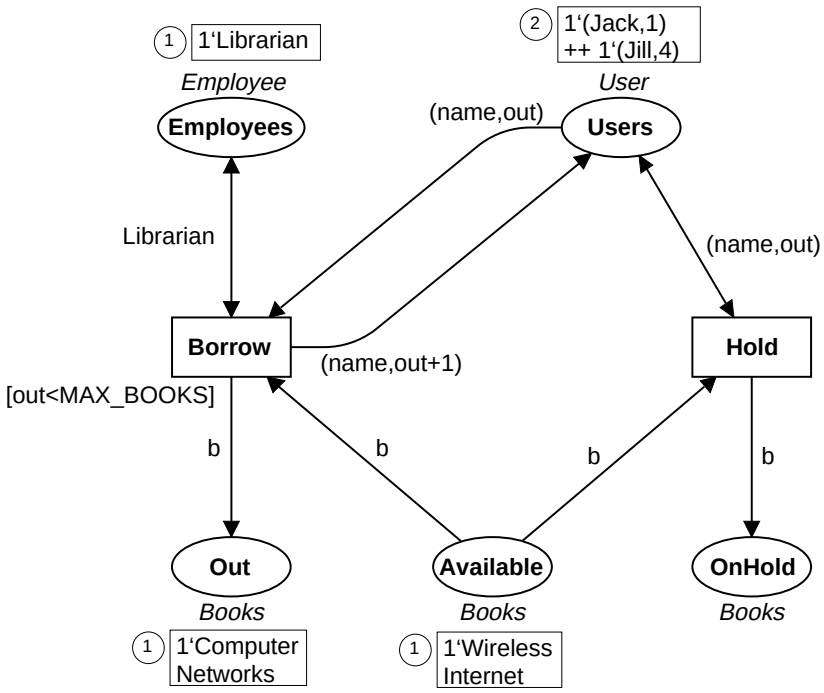

Figure 4.2: Second marking of the example CPN in Fig. 4.1

The important features of CPNs have now been introduced by stepping through the occurrence of one transition. The following section describes the analysis methods that will be used in the thesis to investigate the behaviour of the CPN models. The practicalities of CPN modelling and analysis are discussed in Section 4.4 on computer tools.

4.3 Analysis Methods

4.3.1 Simulation

The example in the previous section showed the CPN changing state from the initial marking to a new marking when a specific binding element occurred. Another binding element that was enabled in the new marking could occur resulting in a third marking. We refer to this process of firing a sequence of transitions as simulating (or executing) the CPN model. Simulation has three main purposes:

1. To gain confidence in the accuracy of the model. Simulating the model can identify errors in its design and give confidence that the model is an accurate representation of the system being modelled. This is the debugging stage of the modelling and analysis process.

2. To investigate specific sequences. Once we are confident in our model, sequences can be simulated to investigate the behaviour of the system under specific scenarios.

3. To formally analyse the model. By simulating all possible sequences of binding elements (and hence, calculating all possible states of the model) we can formally reason about properties of the model. This is known as *state space analysis* and is covered in the following sub-section.

4.3.2 State Space Analysis

The main benefit of creating formal models is that certain properties of the models can be proved. State space analysis is one method for doing this formal analysis.

A state space (of a CPN) is a directed graph with nodes representing the marking or state of the model and arcs representing the binding elements or state changes[1]. From a given initial marking, a state space can be generated by executing enabled binding elements until all reachable markings have been generated. Hence, the state space represents all possible sequences of binding element occurrences or state changes from the initial marking and also all possible states of the model. With this information a range of properties can be proved (or disproved) for the model.

The state space for the example CPN (Figure 4.1) is shown in Figure 4.3. There are nine nodes and 18 arcs. The markings and binding elements are shown in boxes (dashed for binding elements) alongside the state space. The state change from node 1 to node 3 (via arc 2) corresponds to the binding element and resulting marking (Figure 4.2) given as an example in Section 4.2. The marking shows, for all places in the CPN of Figure 4.1, the place name followed by the marking of that place. The binding elements show the transition name, followed by a record specifying the values that each variable is bound.

The following are some interesting properties of a model that can be proven from its state space. The formal definition of these properties can be found in [86].

Reachability: From some marking (e.g. the initial marking) we can determine if another marking is reachable. In the book borrowing example we could ask the question: From the initial marking, can we reach a marking where both books are out? The answer is yes—node 6 has place Out marked with 1‘ComputerNetworks++1‘WirelessInternet.

Dead marking: If a marking has no successors then this state is a *dead marking*, i.e. the system being modelled has terminated. An undesirable dead marking is called a *deadlock*. Deadlocks indicate errors (in either the model or the system being modelled), so functionally correct systems should be free of them. We call a desirable dead marking a *terminal marking*. There are four dead markings in

[1] Throughout the thesis we use the terms *node*, *state* and *marking* interchangeably when referring to the state space. Similarly, the terms *arc*, *event* and *binding element* are used interchangeably.

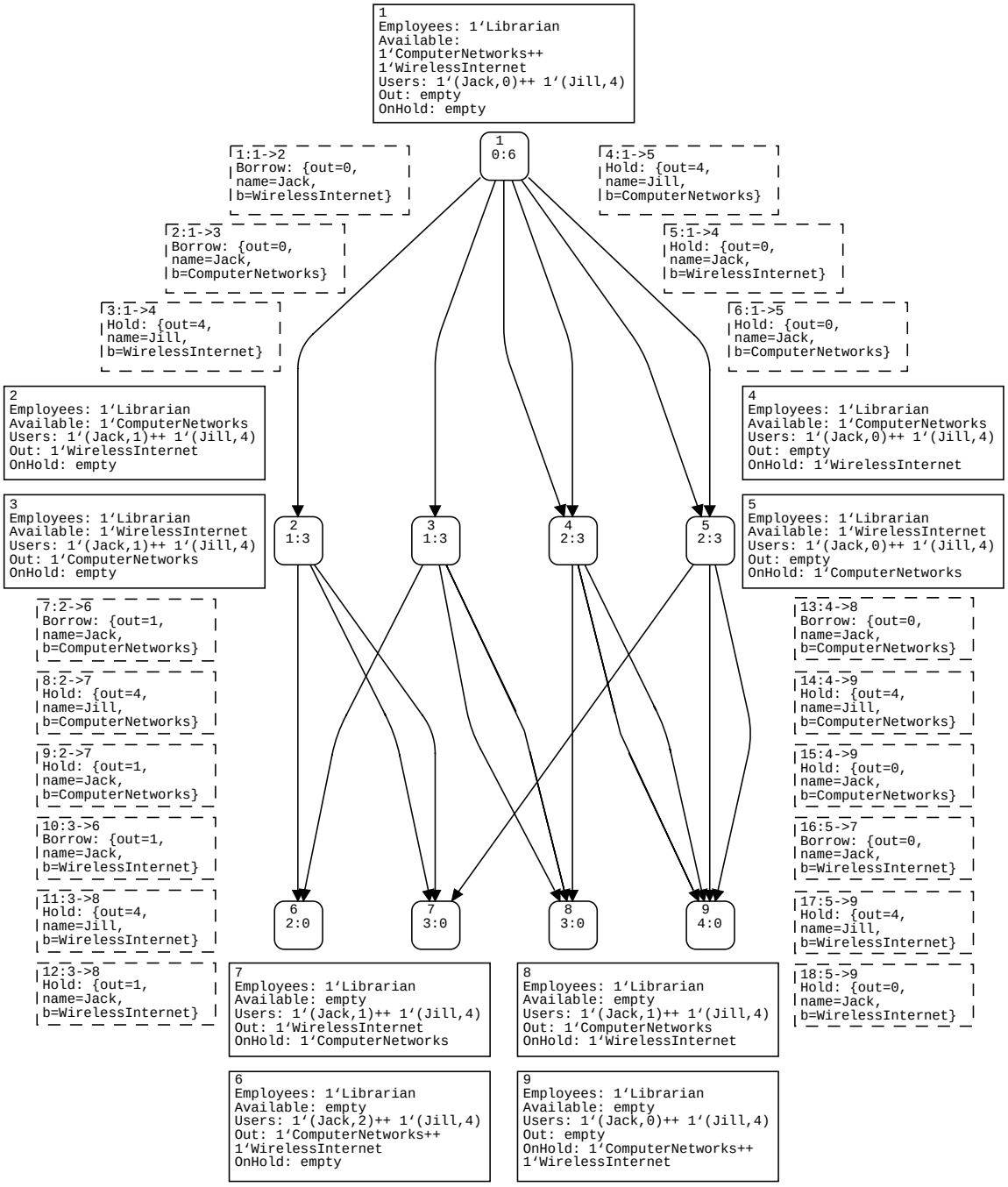

Figure 4.3: State space of the example CPN in Fig. 4.1

Figure 4.3 (nodes 6, 7, 8 and 9). If we require for successful termination of the book borrowing system that no books are both on hold and borrowed, and no users have more than their maximum number of books borrowed, then all of the dead markings are terminal markings (i.e. no deadlocks).

Livelock: A *livelock* is a part of the state space that once entered, will repeat forever. That is, a cycle is entered that leads to no markings outside of the cycle. Livelocks are typically undesirable, and can indicate errors in the system. Figure 4.3 has no livelocks.

Bounds: The markings of places over the full state space determine the bounds on those places. The upper bound of a place is the largest multi-set value it can be marked with at any time. Similarly, a place has a lower bound. Integer bounds (counting the number of tokens on a place) also exist. In the example, the upper multi-set bound on Out is 1‘ComputerNetworks++1‘WirelessInternet and the upper integer bound is 2.

These properties are important when attempting to verify the correctness of a model against a set of criteria. In Chapter 9, some of these properties will be investigated for the Transaction Protocol.

From the state space we can also calculate and draw a graph of the strongly connected components (SCC) [86]. A strongly connected component is a maximal set of nodes where, from each node in the set, all other nodes in the set are reachable. The SCC graph is helpful in determining absence of livelocks, because if the SCC graph and state space are isomorphic, and the state space contains no self loops (a self loop occurs when the source and destination nodes of an arc are identical), then there are no livelocks.

Proving properties automatically from a state space is an important advantage of this analysis technique for CPNs. However, state space analysis has two practical limitations, especially when tackling complex systems:

1. the state space becomes too large to reasonably store in memory of conventional computers (known as *state explosion*), and

2. the state space is dependent on the initial parameters of the model (e.g. maximum values of counters).

One obvious approach to alleviate these problems, which is employed in this thesis, is to model systems with state space analysis in mind. Abstractions of the actual system should be made when the details are not necessary to prove the particular properties of interest. Such abstractions are used in Chapters 6 and 7.

Many techniques for alleviating the state explosion problem have been proposed [163]. An approach that is used in this thesis, which can reduce the amount of memory consumed at any one time to store the state space, is the *sweep-line method* [31].

4.3.3 The Sweep-Line Method

The sweep-line method for state space exploration [31] takes advantage of progress properties inherent in some systems to delete some states after they have been generated. States can be deleted if we know for sure they cannot be reached again. This is known if each state has a *progress measure*, which has the property that a state cannot by occurrence of transitions lead to a state with a lower progress measure. This analysis technique is best explained by illustrating the steps it uses to construct the state space:

1. States are generated via breadth first traversal of successor markings of the initial marking.

2. At any point in the state space the set of states reached can be divided into two: processed (their successors calculated) and unprocessed (successors not calculated).

3. When the number of states processed reaches a user defined value, garbage collection occurs. This involves:

   (a) finding the state among the unprocessed set with the minimum progress measure,

   (b) deleting all states from the processed set that have a lower progress measure than the state found in step (a).

4. The above step is repeated until all states have been processed (or a user defined stop criteria has been satisfied).

The sweep-line method traverses all the states of the full state space, but does not store all the states simultaneously. Properties can be proven using on-the-fly verification. For example, if checking whether a particular state is reachable from the initial marking, a predicate specifying the reachability property is applied on states as they are calculated. If the predicate returns true, the traversal can be stopped, returning the state to the user. The properties that can be verified on-the-fly include those mentioned in Section 4.3.2. A drawback of this method is that once, for example, an erroneous state has been found, a path leading to it (which is useful for debugging purposes) may not have been stored. It is expensive (compared to full state spaces) to find such a path.

Like most techniques that alleviate the state explosion problem, the sweep-line method is suited to systems that exhibit a particular property, in this case, progress. A progress

measure must be defined for each state that gives it a value greater than or equal to its predecessor state. Some example applications that may exhibit this are: communication protocols that use counters or sequence numbers, systems with time that use a global clock (e.g. timed CPNs [86]) and systems that have an iteration counter. In Chapter 9, the sweep-line method is applied to the Transaction Protocol CPN. There we will see in more detail, how a progress measure can be defined.

4.3.4 Language Analysis

As well as proving properties from it, a state space can be used as a means of comparing different models with respect to sequences of events (binding elements). A state space can be treated as a finite state automaton (FSA) [72], where the binding elements represent the alphabet accepted by the FSA. Hence, the language of the FSA defines all possible sequences of events. Theorems and algorithms developed for the analysis of FSAs can be applied to determine if the sequences of one state space/FSA is preserved in another state space/FSA [9, 72] (language equivalence). The approach is based on the fact that any non-deterministic FSA with empty moves (ϵ-transitions) can be converted into a canonical form, a minimized deterministic FSA. If the canonical forms are isomorphic then the two initial FSAs have the same language. This is useful when comparing models at different levels of abstraction, e.g. a design and its requirements, or a protocol and its service. In Chapters 8 and 9 we will use language analysis to determine if the Transaction Protocol correctly implements the Transaction Service.

There are other techniques that can be used for proving the language equivalence of two systems [163]. For example, the Chaos-Free Failures Divergence (CFFD) model [164] not only preserves sequences, but can also be used to check for deadlocks and livelocks. However, as the state space is already calculated from the CPN (from which deadlocks and livelocks can be calculated), CFFD offers no significant advantages. The main advantage of the approach used in this thesis is that there is adequate tool support, as shown in the next section.

4.4 Computer Tools

The previous two sections introduced the features and concepts of CPNs and accompanying analysis techniques. For practical use, computer tools are necessary to support the modelling and analysis process. This section describes Design/CPN [109, 161] and several other pieces of software used in the thesis.

4.4.1 Design/CPN

Design/CPN is a suite of tools for editing, simulating and analysing CPNs. We briefly describe the components that are used in this thesis. For more detailed descriptions the reader is referred to the literature [109, 89, 90, 101, 30].

Editor

Design/CPN has a graphical editor that allows the user to create and layout the different net components. Auxiliary graphics (i.e. non-CPN components) can also be added to enhance the model. The example CPN of Figure 4.1 was created in Design/CPN.

Design/CPN uses *pages* to visually divide the model into components. The pages do not affect the execution or analysis of the model, but can enhance its maintainability and readability. A hierarchy page must be present in all models and defines the pages in the model and their relationships. Figure 4.4 shows the hierarchy page for Figure 4.1. Along with the CPN page (called Borrow), there is a page with the state space analysis results.

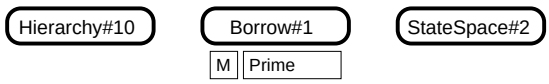

Figure 4.4: Hierarchy page for the example CPN in Fig. 4.1

CPNs on different pages can be connected in two ways:

1. Fusion places: Copies of places can be created on different pages and, if defined to be part of the same *fusion set*, they act as the same place. Therefore, when a *fusion place* on page A is marked with a token, then the corresponding fusion place on page B is marked with the same token.

2. Substitution transitions: Hierarchy can be introduced into the model by representing a CPN on sub-page by a single *substitution transition* on a higher level page. The input and output places of the substitution transition are called input and output *socket places*, respectively (a place that is both input and output to the substitution transition is an I/O socket place), and are also present on the CPN page the substitution transition represents (the places on this sub-page are called *port* places, and typically are assigned to socket places of the same name). A non-hierarchical view of the Design/CPN net would involve replacing the substitution transitions with their corresponding sub-pages.

Examples of fusion places and substitution transitions will be seen in Chapters 6 and 7.

Design/CPN uses CPN ML [109], a dialect of Standard ML [111, 159], for the net inscriptions. The declarations in Figure 4.1 illustrate the use of several CPN ML constructs (colour sets, values, variables). Others (e.g. functions, more complex colour sets) will be introduced when used in Chapters 6 and 7.

Simulator

The graphical CPN model in Design/CPN can be converted into Standard ML code. The Design/CPN Simulator allows the user to execute this code in various ways:

Selecting binding elements: In a particular marking Design/CPN shows the enabled transitions. The user can select one, choose the values the variables are bound to, and fire the transition.

Random simulation with interactive feedback: A random sequence of binding elements is fired. The changing of markings and enabled transitions are displayed to the user as the sequence occurs. Different feedback options can be set as to the speed of the interaction.

Automatic simulation: A random sequence of binding elements is fired. No feedback is given to the user except for the final marking.

The different simulation options are used to debug the model and view specific sequences of events.

State Space Tool

The executable Standard ML code is also used to generate a full or partial state space by using the state space tool [89]. All or part of the state space can be drawn in the state space tool, showing the markings and binding elements (see Figure 4.3). However, visually inspecting the state space is not an adequate technique for formal analysis, especially when the number of states is non-trivial (e.g. more than 50). The state space tool provides a set of functions to assist in finding properties of the model. These include functions to:

- refer to different parts of the state space (e.g. markings, binding elements, successor nodes of a node);
- search the state space (the user can, for example, set the search area and predicates specifying the stop conditions);
- obtain some standard properties of the state space (including the dead markings, bounds on places, liveness properties).

The standard properties, as well as the statistics (number of nodes, arcs etc.), can be written to a file as a report by Design/CPN. As well as obtaining the standard properties, the above set of functions can be used to define queries to prove more complex properties. Examples of such functions will be given in Chapter 8.

Another feature of the state space tool is to allow the user to define the information displayed for the markings and binding elements. This is useful when treating the state space as a FSA. Rather than printing the full binding element to a text file, it can be mapped to an integer. As we will see in Section 4.4.2, integers are used to represent the alphabet of a FSA in the FSM tool.

There are other components of Design/CPN that can be used for analysis. These include: performance analysis [101]; state spaces with equivalence classes [90]; state spaces with symmetries [90]; and the sweep-line method [30]. Of these, we have applied the sweep-line method in Chapter 9.

Sweep-line Library

The sweep-line library is in its early stages of development, and for the analysis in this thesis we used a prototype implementation [30]. The major function of the prototype tool is to install the sweep-line algorithm for state space generation instead of the standard algorithm. The functions available once a full state space is calculated are not available in the sweep-line library because they no longer apply for on-the-fly verification. A progress measure must be written as a function that takes a marking and returns an integer. To prove specific properties (such as deadlocks, bounds) the algorithm is modified to evaluate the necessary functions while calculating the state space with the sweep-line method. The use of the sweep-line library will be demonstrated in Chapter 9.

4.4.2 FSM, LexTools and GraphViz

Design/CPN does not have any built in tools for analysing FSAs. Therefore, our approach is to save the state space from Design/CPN as a text file that can be used as input to other FSA analysis tools. The most successful for our requirements have been the set of libraries from AT&T: FSM [4], LexTools [6] and GraphViz [5]. Several parts of these libraries are regularly used in Chapters 8 and 9.

FSM

FSM is a set of programs for analysing FSAs[2]. First, a text file of the input FSA is compiled into a binary form using `fsmcompile`. The input FSA is usually of the form:

[2]Throughout this thesis FSM is used to refer to the tool [4], not finite state machines.

```
1 2 1
2 3 0
3 4 2
2
3
```

where lines with three integers correspond to *source state*, *destination state* and *transition*, lines with one integer give the halt states, and transitions which are 0 correspond to empty or ϵ-transitions.

The FSM programs regularly used are:

- `fsmrmepsilon`: Removes ϵ-transitions from the FSA.
- `fsmdeterminize`: Convert the non-deterministic FSA into a deterministic FSA (DFSA).
- `fsmminimize`: Convert a DFSA into its minimized form.
- `fsmdifference`: Returns the intersection of one FSA with the complement of a second FSA.
- `fsminfo`: Return information about the FSA (including number of states, transitions and halt states).
- `fsmdraw`: Specifies, in a text form readable by GraphViz, the FSA, with optional layout features, labels and colours.

Appendix A provides more details on the language analysis procedure, and gives an example of the use of the tools.

LexTools

FSAs can be converted to and from languages, and analysis of the language can be performed using the set of programs called LexTools. We make use of the program `lexfsmstrings` which, given an FSA and a file mapping transition numbers to labels, writes the language of an acyclic FSA to a text file.

GraphViz

GraphViz is a set of graph drawing tools that attempts to intelligently layout a graph (from example, so as few as possible arcs cross). The program `dot` is used to take the output from `fsmdraw` and create a PostScript version of a FSA. This is only useful when small FSAs are to be drawn.

Chapter 5

Definition of the Wireless Transaction Protocol

The Wireless Transaction Protocol (WTP) can be seen as a reliable transport protocol in the WAP architecture. Our objective is to investigate the functional behaviour of WTP and verify certain properties. The purpose of this chapter is to provide a summary of the WTP Specification [183]. No attempt is made to discuss or justify any of the decisions made in the design of WTP, as this chapter is meant to accurately reflect the WTP Specification. A critical appraisal of WTP, including further explanation and interpretations, is provided in Chapters 6 and 7, as a prelude to modelling WTP.

Section 5.1 outlines the structure of the WTP Specification. Sections 5.2 and 5.3 respectively describe those sections in the WTP Specification that mainly describe the service and protocol. Parts of the WTP Specification that are not relevant to the CPN modelling and analysis are not included. Forward references to the modelling assumptions and decisions in Chapters 6 and 7 that justify the exclusion of these parts are given instead.

5.1 Structure of the WTP Specification

The specification of WTP is made publicly available by the WAP Forum [183]. This document describes the purpose of WTP, and specifies the service and protocol. Hereafter, we will refer to the whole document as the *WTP Specification*, to the service specification as the *Transaction Service* and the protocol specification as the *Transaction Protocol*. As discussed in Chapter 3, making the distinction between service and protocol is an important step in the design of communication protocols. The WTP Specification comprises eleven sections and three appendices, totaling 67 pages.

Section 1 (Scope)[1] summarizes in three paragraphs the purpose of WTP, namely to provide a reliable request response service.

Section 2 (Document Status) provides pointers to the WAP Forum web site [170] where the document can be downloaded and comments submitted.

Section 3 (References) lists normative and informative references.

Section 4 (Definitions and Abbreviations) defines terms and abbreviations used in the WTP Specification.

Section 5 (Protocol Overview) summarizes WTP features and its relation to other elements in the WAP architecture. It includes descriptions relevant to the service and protocol. We outline this section in Section 5.2.

Sections 6 (Elements for Layer-To-Layer Communication) and 7 (Classes of Operation) describe the Transaction Service. A summary of these sections will be given in Section 5.2.

Sections 8 (Protocol Features), 9 (Structure and Encoding of Protocol Data Units), 10 (State Tables) and 11 (Examples of Protocol Operation) describe the Transaction Protocol. A summary of these sections will be given in Section 5.3.

In some parts of Sections 5 (Protocol Overview) to 11 (Examples of Protocol Operation) there is no clear distinction between describing the Transaction Service or the Transaction Protocol. This will be discussed further in Chapters 6 and 7.

There are three Appendices in the WTP Specification [183]. Appendix A (Default Timer and Counter Values) gives default timer and counter values for the following bearer services: GSM SMS, GSM Unstructured Supplementary Services Data (USSD), and Bearers supporting IP. The appendix states "The timers are initial estimates and have not yet been verified" (page 64, [183]). Appendix B (Implementation Note) provides guidance to implementers on extending timers when large messages are in use. Appendix C (History and Contact Information) lists the history and contact information for the WTP Specification.

For consistency throughout this thesis, we introduce the following definitions:

TR-Service: Transaction Service. Those parts of the WTP Specification that define the service.

TR-Protocol: Transaction Protocol. Those parts of the WTP Specification that define the protocol.

TR-Init-User: User of the TR-Service that initiates a transaction.

TR-Resp-User: User of the TR-Service that responds to a transaction.

[1] When referring to sections from the WTP Specification, we will include the section name in parentheses to distinguish references to sections in this thesis (which only include the number).

TR-User: Either the TR-Init-User or the TR-Resp-User.

TR-Service-Provider: A representation of the entities that provide the TR-Service.

TR-Init-PE: Protocol entity that initiates a transaction.

TR-Resp-PE: Protocol entity that responds to a transaction.

TR-PE: Either the TR-Init-PE or the TR-Resp-PE.

5.2 Transaction Service

The majority of the TR-Service is described in Section 6 (Elements for Layer-to-layer Communication) and Section 7 (Classes of Operation) of the WTP Specification [183]. However, Section 5 (Protocol Overview) presents some information on the TR-Service, and so we include it in this section. For the TR-Service the service primitives and parameters are defined and each class of service described. Along with textual descriptions, the classes of service also have tables defining sequences of service primitives. The primitive sequence tables specify which primitives may immediately follow other primitives. The details of these descriptions are given in this section. As described in Chapter 2, Class 0 provides an unreliable one-way request service, Class 1 provides a reliable one-way request service, and Class 2 provides a reliable two-way request (request/response) service. Where appropriate, we will limit the discussion to Transaction Class 2, as this is the focus of the CPN modelling and analysis. We use identical section headings so a direct correlation to the WTP Specification is provided.

5.2.1 Protocol Overview

Section 5 (Protocol Overview) of the WTP Specification [183] summarizes the protocol features and its relation to other elements in the WAP architecture. It is divided into seven sub-sections: Protocol Features, Transaction Classes, Relation to Other Protocols, Security Considerations, Management Entity, Static WTP Conformance Clause, and Other WTP Users.

The Protocol Features sub-section lists eleven features of the TR-Protocol. These features are specified in Section 5.3.1.

The Transaction Classes sub-section provides a narrative description of the basic behaviour of each of the TR-Protocol classes. The TR-Init-PE is defined as the WTP entity initiating a transaction, and the TR-Resp-PE is defined as the WTP entity responding to a transaction. The basic behaviour for Transaction Class 2 is specified in Section 5.3.1.

The Relation to Other Protocols sub-section describes the relationship between the TR-Protocol and other components in the WAP architecture. Table 5.1 (taken from Section 5.3 (Relation to Other Protocols) of [183]) illustrates the relationship between the TR-User and protocols providing it services (some indirectly).

WTP User (e.g. WSP)	
WTP	Transaction Handling Re-transmissions, duplicate removal, acknowledgments Concatenation and separation
[WTLS]	Optionally compression Optionally encryption Optionally authentication
Datagram Transport (e.g. WDP)	Port number addressing Segmentation and re-assembly (if provided) Error detection (if provided)
Bearer Network (e.g. IP, GSM SMS/USSD, IS-136 GUTS)	Routing Device addressing (IP address, MSISDN) Segmentation and re-assembly (if provided) Error detection (if provided)

Table 5.1: Services provided to TR-User

The Security Considerations sub-section states there are no security mechanisms in WTP.

The Management Entity sub-section describes the role of the WTP management entity. It states the mobile device needs to satisfy the following conditions (page 15, [183]):

- *the mobile is within a coverage area applicable to the bearer service being invoked;*
- *the mobile having sufficient power and the power being on;*
- *sufficient resources (processing and memory) within the mobile are available to WTP;*
- *the WTP protocol is correctly configured, and;*
- *the user is willing to receive/transmit data.*

The management entity monitors the state of the mobile device and notifies the WTP layer if any of the above capabilities are not met. The management entity may also be used by the TR-User to configure parameters of the WTP layer. There is no specification of what the management entity must do, nor how it interacts with the TR-Protocol (except that it must interact with the mobile device) because it is considered implementation specific.

The Static WTP Conformance Clause defines the features that are mandatory or optional when using WTP as a client or server. When describing the features in Section 5.3.1 all are assumed to be mandatory, unless otherwise specified.

The Other WTP Users sub-section states the intended user of WTP is the Wireless Session Protocol [181], but other applications may also use it.

5.2.2 Elements for Layer-to-Layer Communication

Section 6 (Elements for Layer-to-Layer Communication) of the WTP Specification [183] describes the service primitives and parameters used in the Transaction Service. It is divided into three sub-sections: Notations Used, Requirements of the Underlying Layer, and Services Provided to the Upper Layer.

The Notations Used sub-section describes how service primitives and parameters are used, in a similar manner to OSI [79] (see Chapter 3). These will become apparent in the following paragraphs.

The Requirements on the Underlying Layer sub-section states that WTP operates over a datagram service that must provide (page 19, [183]): "Port numbers to route the incoming datagram to the WTP layer; [and] Length information for the SDU passed up to the WTP layer." The datagram service may optionally provide error detection. Lower layers, such as the network layer, are expected to provide segmentation and re-assembly functions.

The Services Provided to the Upper Layer sub-section describes the primitives used in the TR-Service and their parameters. The service primitives used are:

TR-Invoke: Initiates a new transaction. Types: request (req), indication (ind), response (res), confirm (cnf).

TR-Result: Sends back a result of a previously initiated transaction. Types: req, ind, res, cnf.

TR-Abort: Aborts an existing transaction. Types: req, ind.

Each primitive has a set of parameters and a corresponding table that specifies whether the parameter is mandatory, conditional or optional (discussed shortly).

The TR-Invoke parameters are:

Source Address: A unique address of the device at the TR-Init-User.

Source Port: The port number of the application at the TR-Init-User.

Destination Address: A unique address of the device at the TR-Resp-User.

Destination Port: The port number of the application at the TR-Resp-User.

Ack-Type: User acknowledgment - On or Off. If On, then the TR-Users must acknowledge each indication primitive with a response primitive. If Off, then the TR-PEs may optionally acknowledge PDUs without explicit acknowledgment from the TR-Users. This feature is discussed further in Section 5.3.1.

User Data: The data carried by the protocol.

Class Type: 0, 1 or 2.

Handle: An alias that identifies the transaction to the higher layer. This is only used locally, i.e. by the TR-Init-User or the TR-Resp-User.

Table 5.2 (taken from Section 6.3 (Services Provided to the Upper Layer) of [183]) specifies the parameters included in each TR-Invoke primitive. A mandatory parameter (M) must be present in the service primitive. An equals sign (=) for a deliver primitive (indication or confirm) denotes the parameter must be equal to the parameter of the same name in the corresponding submit primitive (request or response). An optional parameter (O) either may or may not be present in the service primitive. A conditional parameter (C) in a deliver primitive must (must not) be present if the optional parameter of the same name in the corresponding submit primitive is (is not) present.

Parameter	**req**	**ind**	**res**	**cnf**
Source Address	M	M (=)		
Source Port	M	M (=)		
Destination Address	M	M (=)		
Destination Port	M	M (=)		
Ack-Type	M	M (=)		
User Data	O	C (=)		
Class Type	M	M (=)		
Handle	M	M	M	M

Table 5.2: Parameters for TR-Invoke primitive

We have omitted the ExitInfo parameter from Table 5.2 because it cannot be used in the Class 2 transactions (see Section 8.4 (Information in Last Acknowledgment) of the WTP Specification [183]).

There is one new parameter used by TR-Result primitives:

Exit Info: Additional user data sent to the TR-User on completion of the transaction.

Table 5.3 (taken from Section 6.3 (Services Provided to the Upper Layer) of [183]) specifies the parameters included in each TR-Result primitive.

There is one new parameter used by TR-Abort primitives:

Parameter	req	ind	res	cnf
User Data	O	C (=)		
Exit Info			O	C (=)
Handle	M	M	M	M

Table 5.3: Parameters for TR-Result primitive

Abort Code: The reason for the abort. It can be either one specified by the TR-Service-Provider or a reason defined by the TR-User. The TR-User defined reasons are not specified in [183], however WSP [181] does specify a set of abort codes.

Table 5.4 (taken from Section 6.3 (Services Provided to the Upper Layer) of [183]) specifies the parameters included in each TR-Abort primitive.

Parameter	req	ind
Abort Code	O	C (=)
Handle	M	M

Table 5.4: Parameters for TR-Abort primitive

5.2.3 Classes of Operation

Section 7 (Classes of Operation) of the WTP Specification [183] describes the legal service primitive sequences seen by each TR-User. It is divided into three sub-sections: Class 0 Transaction, Class 1 Transaction, and Class 2 Transaction.

Table 5.5 (taken from Section 7.3 (Class 2 Transaction) of [183]) defines the legal primitive sequences for the Class 2 Transaction. The sequences are started by a TR-Init-User submitting a TR-Invoke.req primitive, and as a result, the TR-Resp-User being delivered the TR-Invoke.ind primitive.

	TR-Invoke				TR-Result				TR-Abort	
	req	ind	res	cnf	req	ind	res	cnf	req	ind
TR-Invoke.req										
TR-Invoke.ind										
TR-Invoke.res		X								
TR-Invoke.cnf	X									
TR-Result.req		X*	X							
TR-Result.ind	X*			X						
TR-Result.res						X				
TR-Result.cnf					X					
TR-Abort.req	X	X	X	X	X	X	X			
TR-Abort.ind	X	X	X	X	X	X	X			

Note: A primitive listed in the column header may only be followed by primitives listed in the row header that are marked with an X. Those marked with an X* are not possible if the User Acknowledgment option is used.

Table 5.5: Legal primitive sequences for the Transaction Service

Also included in the Class 2 Transaction sub-section is a list of the PDUs used and the general procedure. These will be covered in Section 5.3.

5.3 Transaction Protocol

The majority of the TR-Protocol is described in Sections 8 (Protocol Features), 9 (Structure and Encoding of Protocol Data Units), 10 (State Tables) and 11 (Examples of Protocol Operation) of the WTP Specification [183]. The details of these descriptions (except the examples, which provide no additional information) are given in this section.

5.3.1 Protocol Features

Section 8 (Protocol Features) of the WTP Specification [183] describes, in detail, the different protocol features. It is divided into 14 sub-sections[2], one for each protocol feature. Each sub-section describes one or more of the following components: motivation, service primitives, PDUs, timer intervals and counters, and procedure. The following summarizes the 14 protocol features.

Message Transfer

The two TR-PEs communicate using four primary PDUs: Invoke, Result, Ack and Abort. Other PDUs are used for the optional Segmentation and Re-assembly (SAR) feature. Section 5.3.2 describes the structure of the PDUs. The procedure for normal message transfer (without aborts) in a single transaction involves five steps:

1. Upon the submission of a TR-Invoke.req from the TR-Init-User, the TR-Init-PE sends an Invoke PDU to the TR-Resp-PE. The TR-Init-PE starts a re-transmission timer and waits for a response. If a timeout occurs before receiving a response, then the TR-Init-PE may either re-transmit the Invoke PDU (up to a fixed number of times) or abort the transaction.

2. Upon receipt of the Invoke PDU, the TR-Resp-PE delivers the request to the TR-Resp-User (TR-Invoke.ind) and waits for a result.

3. While waiting for the result (from the TR-Resp-User), the TR-Resp-PE may send a "hold on" Ack PDU to the TR-Init-PE if the TR-Resp-User is taking too long to acknowledge the Invoke PDU. Then the TR-Init-PE knows not to retransmit the Invoke PDU. In this case, a TR-Invoke.cnf is delivered to the TR-Init-User.

4. A TR-Result.req primitive submitted by the TR-Resp-User triggers the TR-Resp-PE to send the Result PDU. Upon receipt of the Result PDU by the TR-Init-PE,

[2] In fact only 13 sub-sections are given, but it is believed that a typographical error has lead to one sub-section (Transmission of Parameters) not receiving a number. In previous versions of the WTP Specification [172, 174], and in the latest version [187], Transmission of Parameters is a sub-section.

a TR-Invoke.cnf (if not already delivered) and TR-Result.ind are delivered to the TR-Init-User.

5. The TR-Init-PE acknowledges the Result PDU by sending an Ack PDU to the TR-Resp-PE. If the TR-Init-PE receives a re-transmitted Result PDU, then it assumes the TR-Resp-PE has not received the Ack PDU (e.g. due to a loss) and so re-transmits the PDU. Upon a time-out, the TR-Init-PE removes any transaction information and re-enters its initial state, after which it cannot re-transmit the Ack PDU.

Re-transmission Until Acknowledgment

This feature is used to provide reliable transfer of PDUs when losses occur. A re-transmission counter (which has a maximum value) and re-transmission timer (which runs over a given interval) is required.

When a PDU is sent by a TR-PE the re-transmission timer is started. If the sending TR-PE receives no acknowledgment of the receipt of the PDU at the peer TR-PE by the time the re-transmission timer reaches the end of its interval (i.e. a time-out occurs), then the PDU is re-transmitted. The re-transmission timer is re-started from 0. The re-transmission counter is incremented by 1. The sender TR-PE again waits for an acknowledgment. This procedure is repeated until either an acknowledgment is received, the transaction is aborted, or a time-out occurs when the re-transmission counter is at its maximum value. If the latter occurs, the transaction is aborted and the local TR-User is notified using a TR-Abort.ind primitive.

The first PDU sent has its re-transmission indicator (RID) field in the header (see Section 5.3.2) set to 0. All re-transmitted PDUs have their RID set to 1. When RID is set to 0, the receiving TR-PE can distinguish between re-transmitted PDUs and PDUs duplicated by the network. If the receiver has already received a PDU with RID set to 0, then when it receives a second PDU with RID set to 0, it can ignore the second PDU (it assumes it has been duplicated in the network). On receipt of a PDU with RID=1, the receiving TR-PE cannot assume the PDU is duplicated because it may have been a deliberate re-transmission by the peer TR-PE. The receiving TR-PE may act on this (e.g. by re-transmitting a PDU). For example, if the TR-Resp-PE has sent an Ack PDU acknowledging receipt of the Invoke PDU, then, upon receipt of a re-transmitted Invoke PDU, the TR-Resp-PE may re-send the Ack PDU (it assumes the first Ack PDU was lost).

The RID field is not sufficient to ensure a re-transmitted Invoke PDU does not start a transaction that has already been started (e.g. by the original Invoke PDU). The Transaction Identifier Verification feature, described shortly, is used to avoid this situation.

User Acknowledgment

This feature provides two options for the requirement on the TR-User to acknowledge messages. When User Acknowledgment (UserAck) is On, a TR-PE cannot acknowledge receipt of a PDU until the TR-User has acknowledged the receipt of the corresponding primitive. This guarantees that when a confirmation primitive is received by a TR-User, the corresponding response primitive had been submitted by the peer TR-User. Figure 5.1, based on Figure 2 in the WTP Specification [183], shows an example sequence of primitives when UserAck is On. Further explanation of this diagram will be given in Chapter 6.

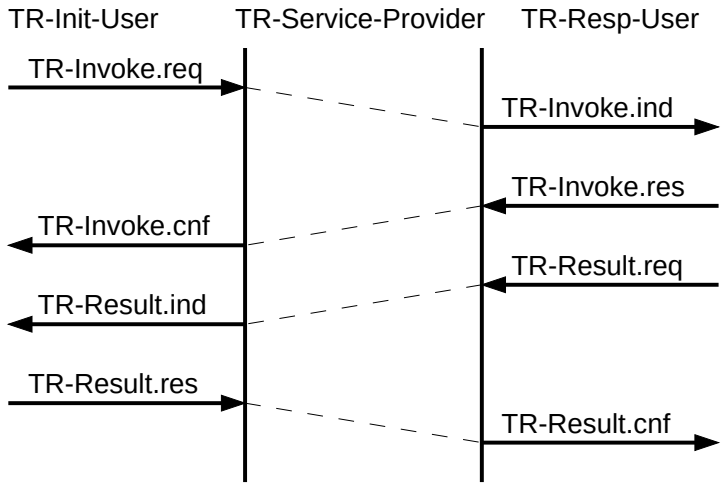

Figure 5.1: Legal service primitive sequence when UserAck is On

The TR-Init-User sets the `Ack-Type` flag in the TR-Invoke.req primitive to 1 if they want UserAck On. When set, UserAck must be On for the complete transaction. The submission of the TR-Invoke.req primitive with Ack-Type=1 results in the `U/P` (User/Protocol entity) flag in the Invoke PDU being set to 1. Upon receipt of the Invoke PDU, the TR-Resp-PE delivers the TR-Invoke.ind primitive (with Ack-Type=1) to the TR-Resp-User and starts an acknowledgment timer. The TR-Resp-PE only sends an Ack PDU after the TR-User has submitted a TR-Invoke.res primitive. If the TR-Resp-PE does not receive a response from the TR-User after a specified time, then the timer expires and an Abort PDU is sent to the TR-Init-PE. A TR-Abort.ind primitive is also delivered to the TR-Resp-User.

UserAck On is an optional feature, but WSP [181] makes use of it, and therefore it is recommended to be implemented. When UserAck is Off, the protocol may acknowledge the receipt of a PDU without a response from the TR-User. For example, in Figure 5.1, the response primitives are omitted. Chapter 6 examines the different possible sequences when UserAck is On and Off.

Information in Last Acknowledgment

This feature allows the TR-User to attach information (e.g. performance measures) to the last acknowledgment of a transaction. This information is given by the TR-User as the ExitInfo parameter of the TR-Result.res primitive. The information is transported as a Transport Information Item (TPI) in the variable part of the Ack PDU header. The Information TPI cannot be included in the Ack PDU used to acknowledge the Invoke PDU.

Concatenation and Separation

This feature allows multiple PDUs to be transferred in one SDU by the datagram layer. The mechanisms for concatenating multiple PDUs into one SDU and separating one SDU into multiple PDUs are not specified. This is implementation dependent. However, the structure of the PDUs used by this feature is specified (see Section 5.3.2).

Asynchronous Transactions

This feature allows multiple transactions to be performed concurrently. A TR-Init-PE may initiate a second transaction before a response to its first transaction is received. The TR-Resp-PE should process transactions independently of other transactions. It does not have to wait until a previous transaction is complete. If a TR-Resp-PE reaches the maximum number of transactions it can simultaneously handle (this number is implementation dependent) and receives an Invoke PDU, the TR-Resp-PE should ignore and discard the Invoke PDU. An Abort PDU is not sent to the TR-Init-PE because the TR-Resp-PE may accept re-transmitted Invoke PDUs at a later stage.

Transaction Abort

This feature allows the TR-User (by submitting a TR-Abort.req) to abort any outstanding transaction (i.e. a transaction that is in progress). Also the TR-Service-Provider may abort an outstanding transaction if there is an error in the TR-Service-Provider. An abort reason is used to indicate who initiated the abort: TR-User or TR-Service-Provider.

There are three special cases that must be considered when aborts occur (page 32, [183]):

1. *The sending WTP provider has not yet sent the message: the provider MUST discard the message from its memory.*

2. *The sending WTP provider has sent the message to the peer, or is in the process of sending the message: the provider MUST send the Abort PDU to the remote peer to discard all data associated with the transaction.*

3. *The receiving provider receives the Abort PDU: it generates the TR-Abort indication primitive and discards all transaction data.*

Note that the reference to the "WTP provider" in the WTP Specification is equivalent to the TR-PE. Similarly, a "message" is equivalent to a PDU.

Transaction Identifier

Each transaction started by a single TR-Init-PE has a transaction identifier (TID) associated with it. The TID, along with the source and destination addresses and ports, uniquely identifies a transaction. The TR-Init-PE increments the TID by one for every new transaction it starts (no matter whether the TR-Resp-PE is the same or not). The same TID is used in all PDUs sent within the transaction. The TR-Resp-PE, when receiving an Invoke PDU, uses the TID to detect old Invoke PDUs. The TR-Resp-PE should be expecting an Invoke PDU with a TID higher than the previous transaction initiated by the same TR-Init-PE. If it doesn't receive one, then TID verification is performed (discussed in the next protocol feature).

The TID is represented as a 16-bit integer. The highest order bit indicates the direction of the PDU sent (0 for TR-Init-PE to TR-Resp-PE, 1 for TR-Resp-PE to TR-Init-PE). This is necessary when an application acts as both a TR-Init-User and TR-Resp-User and uses the same port and address. If a received PDU on that port has the highest bit set to 1, then the PDU is destined for the TR-Init-PE. Excluding the highest order bit, there are effectively 2^{15} unique TID values (0 to 32767). The TR-Init-PE cannot increment the TID more than 2^{14} times in a period of 2MPL. MPL (Maximum Packet Lifetime) is the maximum time that packets are assumed to be in the network. The restriction on incrementing the TID is designed to ensure that when the TR-Init-PE starts a transaction, old duplicate Invoke PDUs of the previous incarnation of that transaction (e.g. 32768 transactions ago) are no longer in the network. There are two valid cases where the current transaction TID may be less than the previous transaction TID: a failure in the TR-Init-PE has led to it re-starting with a TID value less than the previous one, and the TID values have wrapped (i.e. ..., 32766, 32767, 0, 1, ...). These cases lead to TID verification being performed.

The TR-Resp-PE determines whether TID verification should be performed by caching the TID values of previous transactions. There is no requirement for the TR-Resp-PE to perform caching (nor requirement on the type of caching mechanism), although it is recommended to increase efficiency of the protocol. If the TR-Resp-PE does not use a cache, then every Invoke PDU received results in TID verification being performed. If a cache is being used, then the TID of the received Invoke PDU is compared to the TID of the last transaction initiated by the same TR-Init-PE. The WTP Specification gives an

example test algorithm, but other algorithms may be used. In general, the test should fail (therefore, initiating TID verification) when the received TID is less than or equal to the last TID. If the test succeeds, then the last TID becomes the received TID and the transaction proceeds (without TID verification). An exception occurs when the `TIDnew` flag (see Section 5.3.2) in the Invoke PDU is set. This empties the cache and forces TID verification to be performed. Chapter 7 provides further insight into appropriate testing algorithms, and their requirements to ensure old Invoke PDUs do not initiate transactions that have already been started.

Transaction Identifier Verification

TID verification is performed when the TR-Resp-PE is uncertain if the Invoke PDU it receives is valid (in which case, the TR-Init-PE wants the transaction performed) or an old duplicate (the transaction should not be re-started). TID verification is used to ask the TR-Init-PE to confirm or reject the Invoke PDU.

When the TR-Resp-PE detects that TID verification is required (in general, the TID of the PDU received is less than or equal to the TID of the last Invoke PDU received for the same TR-Init-PE) an Ack PDU, with the `Tve/Tok` flag set (see Section 5.3.2), is sent to the TR-Init-PE. Upon receipt of this Ack PDU, the TR-Init-PE responds based on whether it wishes the TR-Resp-PE to proceed with this transaction or not:

1. If the TID of the Ack PDU is an outstanding TID of the TR-Init-PE, then the TR-Resp-PE should proceed. The TR-Init-PE sends an Ack PDU, with the `Tve/Tok` flag set.

2. If the TID of the Ack PDU is not an outstanding TID of the TR-Init-PE, then the TR-Resp-PE should not proceed. The TR-Init-PE sends an Abort PDU with the Abort Reason field set to INVALIDTID (see Section 5.3.2).

The TID verification procedure amounts to a three-way handshake: an Invoke PDU is sent from TR-Init-PE to TR-Resp-PE; the TR-Resp replies with an Ack PDU (Tve/Tok); and the TR-Init-PE replies with an Ack PDU (Tve/Tok) if the TR-Resp-PE should proceed, or an Abort PDU (INVALIDTID) if the TR-Resp-PE should abort the transaction.

Transport Information Items

This feature allows additional information to be sent with PDUs in the variable part of the header. This additional information is referred to as TPIs. There are four TPIs: Error, Info, Option and Packet Sequence Number. The Error TPI is sent when an erroneous TPI type is received (unless the erroneous TPI was received in the last message of the

transaction, when the receiver cannot notify the sender of the error). The latter three TPIs are used by, and described with, other protocol features.

Transmission of Parameters

This feature allows protocol parameters to be transmitted between TR-PEs. This is achieved using the Option TPI. There are no mandatory parameters. The SAR feature defines several optional parameters.

Error Handling

When an error occurs in a TR-PE during a transaction the local TR-User must be notified and the transaction aborted (with an appropriate Abort reason).

Version Handling

If an Invoke PDU, with a version number not supported by the TR-Resp-PE, is received, then the TR-Resp-PE must abort the transaction. The version number is defined in Section 5.3.2.

Segmentation and Re-assembly

This feature allows PDUs that will be too large to be sent as one packet on the bearer network to be segmented into smaller packets. The procedure for re-assembling the received packets into one PDU is also defined. SAR is an optional feature and is not modelled or analysed in the following chapters (see Chapter 7 for the justification). Therefore, no further details are given here, and the reader is referred to the WTP Specification [183] for further explanation (if desired).

5.3.2 Structure and Encoding of Protocol Data Units

Section 9 (Structure and Encoding of Protocol Data Units) of the WTP Specification [183] defines the formatting of the PDUs. It is divided into five sub-sections: General, Common Header Fields, Fixed Header Structure, Transport Information Items, and Structure of Concatenated PDUs. The following summarizes the five sub-sections.

PDUs in the TR-Protocol are formed as an integral number of octets, and contain the following parts: fixed header; variable header; and data (optional). The structure of the header parts depends on the type of PDU: Invoke, Result, Ack, Abort, Segmented Invoke, Segmented Result and Segmented Ack. The type is defined by a PDU code, always given as 4 bits in the first octet of the fixed header.

The following header fields are present in two or more types of PDUs:

Continue Flag (CON): When set, indicates one or more TPIs are present in the variable part of the header. If clear, indicates no TPIs are present, i.e. the variable part of the header has a length of 0 octets. Also used in the TPI header.

Group Trailer Flag (GTR): When set, indicates the last packet of a group when SAR is used. Used in combination with `TTR` (see the next field)—if both `GTR` and `TTR` are set, then SAR is not supported.

Transmission Trailer Flag (TTR): When set, indicates the last packet of a message when SAR is used. Used in combination with `GTR`—if both `GTR` and `TTR` are set, then SAR is not supported.

Packet Sequence Number (PSN): Indicates the position of the packet in a segmented message when using SAR.

PDU Type: Indicates the type of PDU: Invoke, Result, Ack, Abort, Segmented Invoke, Segmented Result and Segmented Ack.

Reserved (RES): Bits reserved for future use.

Re-transmission Indicator (RID): Indicates whether the PDU is the first one sent (0) or if it is a re-transmitted PDU (1).

Transaction Identifier (TID): Associates the PDU with a transaction.

The Invoke PDU (the complete header of which is given in Figure 5.2, which is taken from Section 9.3 (Fixed Header Structure) of the WTP Specification [183]) has the following additional header fields:

Transaction Class (TCL): Indicates the class of the transaction: 0, 1 or 2.

TIDnew Flag: Indicates that the TID of the current transaction is lower than the TID of the preceding transaction. This occurs, for example when the TID has wrapped around.

Version: Indicates the version of WTP being used. The version in the WTP Specification [183] is 00x0 (or *Version Zero*).

User/Protocol Entity Flag (U/P): Indicates whether UserAck is On (1) (acknowledgment provided by TR-User) or Off (0) (acknowledgment provided by TR-PE).

The Result PDU header is given in Figure 5.3, which is taken from Section 9.3 (Fixed Header Structure) of the WTP Specification [183]. It has no additional header fields.

Octet/Bit	0	1	2	3	4	5	6	7
1	CON	PDU Type = Invoke				GTR	TTR	RID
2 3	TID							
4	Version		TIDnew	U/P	RES	RES	TCL	

Figure 5.2: Invoke PDU header structure

Octet/Bit	0	1	2	3	4	5	6	7
1	CON	PDU Type = Result				GTR	TTR	RID
2 3	TID							

Figure 5.3: Result PDU header structure

The Ack PDU header is given in Figure 5.4, which is taken from Section 9.3 (Fixed Header Structure) of the WTP Specification [183]. It has the following additional header fields:

Tve/Tok Flag: Indicates if the Ack PDU is part of the TID verification procedure (see the Transaction Identifier Verification feature in Section 5.3.1). Tve indicates the TR-Resp-PE is performing a verification, and Tok (transaction ok) indicates the TR-Init-PE wants the TR-Resp-PE to proceed with the transaction.

Octet/Bit	0	1	2	3	4	5	6	7
1	CON	PDU Type = Ack				Tve/Tok	RES	RID
2 3	TID							

Figure 5.4: Ack PDU header structure

The Abort PDU header is given in Figure 5.5, which is taken from Section 9.3 (Fixed Header Structure) of the WTP Specification [183]. It has the following additional header fields:

Abort Type: Indicates whether the abort has been initiated by the TR-User or the TR-Service-Provider.

Abort Reason: Indicates the reason for the abort. There are TR-Service-Provider reasons (Unknown, Protocol Error, Invalid TID, Not Implemented Class 2, Not Implemented SAR, Not Implemented User Acknowledgment, WTP Version Zero, Capacity Temporarily Exceeded, No Response, Message Too Large) and TR-User reasons (there are no reasons defined—these are specific to the local TR-User (e.g. WSP [181])). Refer to the WTP Specification [183] for descriptions of the abort reasons.

Octet/Bit	0	1	2	3	4	5	6	7
1	CON	PDU Type = Abort				Abort type		
2 3	TID							
4	Abort reason							

Figure 5.5: Abort PDU header structure

Further details on the SAR PDUs (Segmented Invoke, Segmented Result, Segmented (or Negative) Ack) and the structure of the Transport Information Items and concatenated PDUs is omitted because these features are not modelled or analysed (see Chapter 7 for the justification).

5.3.3 State Tables

Section 10 (State Tables) of the WTP Specification [183] describes the timer, variables and counters used in the TR-Protocol, and gives the state tables. It is divided into six sub-sections: General (which simply provides an overview); Event Processing; Actions; Timers, Counters and Variables; WTP Initiator; and WTP Responder.

Event Processing

The Event Processing sub-section describes how events are validated before they are processed according to the state tables. Table 5.6 (taken from Section 10.2 (Event Processing) of the WTP Specification [183]) describes the tests on incoming events. The incoming events are the receipt of request and response primitives from the TR-User, the receipt of an indication primitive from the datagram layer (e.g. T-DUnitData.ind) or internal events (e.g. time-outs) in the TR-PEs. If no action is taken, then the events are processed according to the state tables. We have added Entry numbers to the left of the table so we can clearly refer to individually entries.

Entry 1 in Table 5.6 indicates an Invoke PDU received by the TR-Resp-PE causes a new transaction to be created and the Invoke PDU to be processed according to the state tables.

Entry 2 refers to the `TIDve` flag (as do several entries in the state tables, as will be seen shortly). This is an abbreviation for the `TveTok` flag in the Ack PDU header when the Ack PDU is in the direction from TR-Resp-PE to TR-Init-PE. Similarly, in the opposite direction the abbreviation `TIDok` is used. We also refer to the Ack PDU with `TIDve` and `TIDok` set as the Ack(Tve) and Ack(Tok) PDUs, respectively.

Actions

The following actions are used in the state tables:

	Test	Action
1	UnitData.ind on the Responder: Invoke PDU	Create a new transaction
2	UnitData.ind on the Initiator: Ack PDU with the TIDve flag set, no matching outstanding transaction	Send Abort PDU (INVALIDTID)
3	UnitData.ind: Ack PDU, Result PDU or Abort PDU, no matching outstanding transaction	Ignore
4	Illegal PDU type or erroneous header structure	Refer to entry 'RcvErrorPDU' in state tables
5	Buffer overflow or out-of-memory errors	Send Abort PDU (CAPTEMPEXCEED)
6	UnitData.ind on the Responder: Invoke PDU requesting Class 2 transaction and Class 2 is not supported	Send Abort PDU (NOTIMPLEMENTEDCL2)
7	UnitData.ind on the Responder: Invoke PDU using SAR and SAR is not supported	Send Abort PDU (NOTIMPLEMENTEDSAR)
8	UnitData.ind on the Responder: Invoke PDU requesting User Acknowledgment and User Acknowledgment is not supported	Send Abort PDU (NOTIMPLEMENTEDUACK)
9	UnitData.ind on the Responder: Invoke PDU with Version $\neq$ 00x0	Send Abort PDU (WTPVERSIONONE)
10	UnitData.ind on the Responder: Invoke PDU when no more transaction requests can be accepted	Ignore

Table 5.6: Test of incoming events

Start timer, <interval>: Start the timer to run over the interval specified. If the timer is already running, it is re-started to run over the interval specified.

Stop timer: Stop the timer.

Reset counter: Set the counter to 0.

Increment counter: Increment the counter by 1.

Queue (Time T): Queue a PDU for eventual delivery. The PDU is sent when either the timer with interval T expires, or a Send action occurs (see below).

Send: Send the PDU and any PDUs queued to be sent.

Timers, Counter and Variables

Each transaction has a single timer associated with it. The timer is given an interval, and a time-out occurs if the end of the interval is reached. The following timer intervals are used by TR-PEs:

Acknowledgment interval (A): The time a TR-PE will wait for a response from the TR-User before generating a time-out.

Re-transmission interval (R): The time a TR-PE will wait for an acknowledgment from the peer TR-PE before generating a time-out.

Wait time-out interval (W): The time the TR-Init-PE will wait before deleting all transaction state information. This interval does not apply to the TR-Resp-PE.

Although there is only one timer used, we refer to, for example, the timer given the acknowledgment interval as the *acknowledgment timer*.

There are two counters used by the TR-PEs:

Acknowledgment Expiration Counter (AEC): Counts the number of times a time-out occurs due to the acknowledgment timer expiring. If the counter reaches its maximum value, `AEC_MAX`, and another acknowledgment time-out occurs, the transaction is aborted.

Re-transmission Counter (RCR): Counts the number of times a time-out occurs due to the re-transmission timer expiring. If the counter reaches its maximum value, `RCR_MAX`, and another re-transmission time-out occurs, the transaction is aborted.

Default values for the timer intervals and maximum counter values are given in Appendix A of the WTP Specification [183].

Table 5.7 (taken from Section 10.4 (Timers, Counters and Variables) in the WTP Specification [183]) describes the variables used in the TR-Protocol by both TR-PEs.

Variable	Type	Description	Comment
GenTID	Uint16	The TID to use for the next transaction. Incremented by one for every initiated transaction.	Global Only Initiator
SendTID	Uint16	The TID value to send in all PDUs in this transaction	One per transaction
RcvTID	Uint16	The TID values expected to receive in every PDU in this transaction. RcvTID = SendTID XOR 0x8000	One per transaction
LastTID	Uint16	The last received TID from a certain remote host	One per remote host Only Responder
HoldOn	BOOL	True if HoldOn acknowledgment has been received	One per Class 2 transaction
Uack	BOOL	True if User Acknowledgment has been requested for this transaction	One per transaction

Table 5.7: Variables used by the Transaction Protocol

The two types used are `Uint16` (a 16 bit unsigned integer) and `BOOL` (boolean). The comment column specifies where and when the variables are used. `GenTID` is a global variable (i.e. applies across all transactions) that is used only by the TR-Init-PE. A new set of the four variables `SendTID`, `RcvTID`, `HoldOn` and `Uack` must be defined for each transaction. They are used by both TR-PEs. `LastTID` is a global variable used only by the TR-Resp-PE, except in this case a new variable must be used for each different TR-Init-PE that starts a transaction with the TR-Resp-PE.

Note that the variable `RcvTID` is equivalent to `SendTID` except the highest order bit is complemented. In hexadecimal, `RcvTID` = `SendTID XOR 0x8000`.

WTP Initiator and Responder

The WTP Initiator and WTP Responder sub-sections give the state tables for the TR-PEs. There are four state tables for the TR-Init-PE and six state tables for the TR-Resp-PE (one is for Class 1 transactions only). Each state table specifies the major state of one of the TR-PEs. (As we will see in Chapter 7, the complete state of a TR-PE comprises both the major state it is in, or *state name*, and a set of values of variables and counters used by the TR-PE.) The meaning of each state is given in Table 5.8 (although this information is not given directly in the WTP Specification, it is easily derived from the state tables that are given).

State Name	*Significance*
NULL	TR-Init-PE has not begun a transaction and is waiting for submission of TR-Invoke.req from TR-Init-User.
RESULT WAIT	TR-Init-PE has sent an Invoke PDU to the TR-Resp-PE and is waiting for the result from TR-Resp-PE.
RESULT RESP WAIT	TR-Init-PE has delivered TR-Result.ind to the TR-Init-User and is waiting for acknowledgment of result from TR-Init-User.
WAIT TIMEOUT	TR-Init-PE saves transaction information in case acknowledgment of result needs to be re-transmitted to TR-Resp-PE.
LISTEN	TR-Resp-PE is ready to accept transactions.
TIDOK WAIT	TR-Resp-PE has initiated TID verification and is waiting for response from TR-Init-PE.
INVOKE RESP WAIT	TR-Resp-PE has delivered TR-Invoke.ind to the TR-Resp-User and is waiting for acknowledgment of the invoke from TR-Resp-User.
RESULT WAIT	TR-Resp-PE is waiting for submission of TR-Result.req from TR-Resp-User.
RESULT RESP WAIT	TR-Resp-PE has sent Result PDU to TR-Init-PE and is waiting for acknowledgment of the result from TR-Init-PE.

Table 5.8: Significance of TR-PE state names

One scenario for a successful transaction would have the TR-Init-PE start in the NULL state and traverse through the other states (in the order they are given in Table 5.8) until it returned to the NULL state when the transaction is finished. Similarly, the TR-Resp-PE would start in the LISTEN state and traverse through the other (bypassing TIDOK WAIT if verification of the TID is not required), returning to LISTEN on completion.

The general structure of the state tables are described in the remainder of this section, using the TR-Resp-PE LISTEN state table (Table 5.9) from the WTP Specification [183] as an example. All of the state tables from the WTP Specification are given in Appendix B. Again we have numbered the entries, but only those that apply for Transaction Class 2.

A state table is given for each possible state of the two TR-PEs. Table 5.9 defines the procedures to follow when the TR-Resp-PE is in the LISTEN state. The state table has four columns:

	Event	Condition	Action	Next State
1	RcvInvoke	Class == 2 | 1 Valid TID U/P flag	Generate TR-Invoke.ind Start timer, A Uack = True	INVOKE RESP WAIT
2		Class = 2 | 1 Valid TID	Generate TR-Invoke.ind Start timer, A Uack = False	
		Class == 0	Generate TR-Invoke.ind	LISTEN
3		Class == 2 | 1 Invalid TID	Send Ack(TIDve)	TIDOK WAIT
4	RcvErrorPDU		Send Abort PDU (PROTOERR)	LISTEN

Table 5.9: TR-Protocol state table: TR-Resp-PE LISTEN

Event: The incoming event. The incoming events are the receipt of request and response primitives from the TR-User, the receipt of an indication primitive from the datagram layer (i.e. the receipt of PDUs) or internal events (e.g. time-outs) in the TR-PE.

Condition: The conditions, if any, of the incoming event. The conditions are predicates on: the parameters of service primitives, header fields of PDUs, counters or variables.

Action: The action to be taken when the event occurs and the conditions are met. The actions may be: sending PDUs; delivery of primitives to the TR-User; or modifying variables, timers and counters.

Next State: Specifies the next state for the TR-PE to enter after the action has been taken. The next state must be one of the states defined by the state tables (including the current state).

Each tuple of (event, condition, action, next state) can be referred to as a state table entry. For example, there are five state table entries in the TR-Resp-PE LISTEN state table, of which only four apply for Transaction Class 2. Chapter 7 provides further explanations on interpreting the state tables.

Chapter 6

Transaction Service Specification

The Transaction layer in the WAP architecture comprises a definition of the Transaction Service (TR-Service) and the Transaction Protocol (TR-Protocol). As a first step towards verifying that the Transaction layer possesses certain properties, the TR-Service must be formally modelled and analysed. With a well defined service specification, this step should be straightforward. However, this chapter will show that the WTP Specification [183] contains an inadequate service specification, leading to difficulties with the CPN modelling and analysis.

Most of the discussion in this chapter is a result of performing an initial CPN analysis of the TR-Service and TR-Protocol. Ambiguity in the TR-Service allowed several interpretations of its meaning to be made. The initial models were based on different interpretations. Comparing the TR-Service CPN and TR-Protocol CPN models identified inconsistencies between the two. This gave insights into the appropriateness of the interpretations made, and they could be refined to obtain an unambiguous and accurate TR-Service. For example, in [54] we presented a model of the TR-Service that didn't enforce end-to-end behaviour of the service, and allowed TR-Abort primitives to occur after a successful transaction. An early model of the TR-Protocol [56] was then compared to the TR-Service, identifying inconsistencies in both models. Rather than present the initial models of the TR-Service, a discussion of its drawbacks is given, and only the final TR-Service model is presented.

In Section 6.1, the current service specification is discussed. This leads to a set of modelling assumptions and decisions for creating an adequate TR-Service CPN. Section 6.2 describes the resulting CPN model. Section 6.3 presents the analysis results, which include the TR-Service language for each value of UserAck (On and Off). This chapter concludes with a discussion of the impact of the results on the remaining steps in the verification process.

Comments and suggestions from Professor Jonathan Billington regarding the definitions of successful and aborted transactions were received throughout the candidature.

The initial idea of modifying the service in [54] to require end-to-end behaviour of the primitives is also due to Professor Billington.

6.1 Discussion of the Transaction Service

The description of a communication architecture layer using both a service specification and a protocol specification is a fundamental part of distributed systems design (as discussed in Chapter 3). Although there is no explicit division in [183], different sections of the WTP Specification were identified as defining the TR-Service (see Section 5.2) and the TR-Protocol (see Section 5.3). The division was based on the majority of the content of each section, as some TR-Service sections contained references to protocol features (for example, sending and receiving PDUs), and vice versa. This is a shortcoming of the WTP Specification as the service and protocol should be described independently. This section discusses the TR-Service given in the WTP Specification [183]. The basic behaviour of the TR-Service (i.e. when aborts do not occur) is discussed first, and then the impact of aborts on the TR-Service are considered.

6.1.1 Structure of the TR-Service

The two core elements of the TR-Service definition are: the primitives and their parameters; and the set of possible primitive sequences. Figure 6.1 shows an idealized model of the TR-Service. The primitives that can be submitted by, and delivered to, each TR-User are shown. For clarity, the parameters are omitted. If the primitives are executed in the correct sequence and the correct parameters are delivered, then the desired service will be provided to the TR-Users.

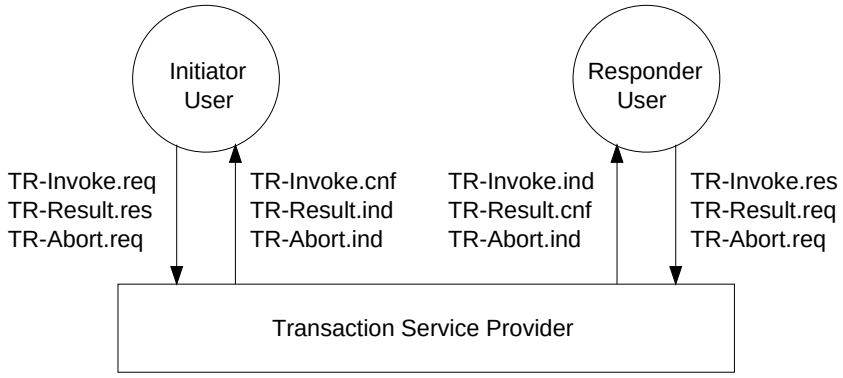

Figure 6.1: Block diagram of the TR-Service

Only one TR-Init-User and one TR-Resp-User are considered in the system. This is consistent with the definition of the TR-Service, where the primitive sequence table (Table 5.5) specifies the primitives for only one transaction. The TR-Users view the

TR-Service-Provider as a single entity. An example primitive sequence may comprise of the TR-Init-User submitting a TR-Invoke.req primitive, resulting in the TR-Service-Provider delivering a TR-Invoke.ind primitive to the TR-Resp-User. When describing sequences of primitives, we categorize them into describing three types of transactions: a *successful transaction*; an *aborted transaction*; or an *incomplete transaction*. The first two types of transaction comprise sequences of primitives where the transaction has been completed. The sequences may either be legal (the sequence is defined in the TR-Service, e.g. Table 5.5) or illegal. An incomplete transaction, although possibly defining part of a legal sequence, always consists of illegal sequences (i.e. a legal sequence must define a completed transaction). The remainder of this section uses these categories of transactions to describe the TR-Service.

6.1.2 Basic Behaviour

The TR-Service can be separated into two types: UserAck Off and UserAck On. In the primitive sequence table (Table 5.5) the two types are differentiated by the exclusion of two possible primitive sequences (marked with a X*) when UserAck is On. This suggests the set of primitive sequences with UserAck On is a subset of the set of primitive sequences when UserAck is Off. In the following we describe the basic behaviour of each of the types (i.e. when aborts do not occur). However, we first define the conventions used in the diagrams that illustrate service primitive sequences.

Time Sequence Diagram Conventions

Figure 6.2 is a time sequence diagram (TSD), whose conventions are based on those defined in [79]. The vertical lines represent the interface between a TR-User and TR-Service-Provider. They also represent the passage of time increasing from top to bottom. The TR-Init-User is shown on the left and the TR-Resp-User on the right. The TR-Service-Provider is shown between the two vertical lines. An arrow towards the vertical line represents a service primitive being submitted by the TR-User. An arrow away from the vertical line represents a service primitive being delivered to the TR-User. Two horizontally aligned primitives (e.g. TR-Invoke.cnf and TR-Result.req) can occur in any order (e.g. TR-Invoke.cnf then TR-Result.req, or TR-Result.req then TR-Invoke.cnf), otherwise the ordering is determined from top to bottom. A (dashed) line in the TR-Service-Provider is used to correlate a submitted primitive type (request or response) with a delivered primitive type (indication or confirmation). The absence of a line between a submitted and delivered primitive indicates there is no time relationship between the two. A (dashed) wave between two primitives (see Figure 6.5) indicates both options are possible, i.e. the two primitives are related in time or the two primitives are not related

in time. The conventions for horizontally aligned primitives and lines or waves between primitives differ from those used in [79]. We have introduced them because they are beneficial for compactly describing different scenarios in the TR-Service. Several of the conventions are illustrated later in this chapter.

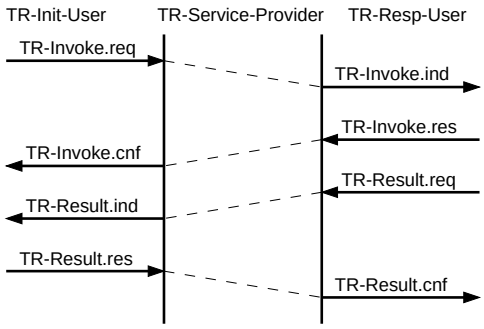

Figure 6.2: Basic primitive sequence for the TR-Service with UserAck On

UserAck On

When UserAck is On, a primitive sequence indicative of the basic behaviour of the TR-Service is shown in Figure 6.2. This sequence shows the TR-Init-User making a request (TR-Invoke.req) which is delivered to the TR-Resp-User (TR-Invoke.ind). The TR-Resp-User confirms the request was received (TR-Invoke.res) which is in turn delivered to the TR-Init-User (TR-Invoke.cnf). After confirming the receipt of the request, the TR-Resp-User sends the result (TR-Result.req), which is delivered to the TR-Init-User (TR-Result.ind). Finally, the TR-Init-User confirms the receipt of the result (TR-Result.res), resulting in a TR-Result.cnf primitive being delivered to the TR-Resp-User. From the primitive sequence table (Table 5.5), no primitive can follow a TR-Result.cnf primitive at the TR-Resp-User. The question arises as to whether the sequence in Figure 6.2 constitutes a successful transaction. We believe it does, but the WTP Specification contains no statements regarding the end of a transaction sequence. Therefore, we make the following assumption about the successful completion of a transaction primitive sequence:

Assumption 6.1 (Successful Transaction (UserAck On)). *A successful transaction in the TR-Service (with UserAck On) is any legal sequence defined in Table 5.5 that is complete when the TR-Init-User has submitted the TR-Result.res primitive and the corresponding TR-Result.cnf primitive has been delivered to the TR-Resp-User.* □

From our explanations of complete and incomplete transactions in Section 6.1.1, Assumption 6.1 does not include the scenario where a TR-Abort.req or TR-Abort.ind primitive at the TR-Init-User follows a TR-Result.res primitive. These TR-Abort primitives are discussed in Section 6.1.3.

UserAck Off

When UserAck is Off, the UserAck On set of primitive sequences is possible. There are other sequences that are possible, due to the inclusion of the entries in the primitive sequence table (Table 5.5) marked with X*. Therefore, we only discuss those sequences that are possible when UserAck is Off, but not when UserAck is On.

Again, there is no indication of transaction completion for the primitive sequences. If Assumption 6.1 is followed, two possible primitive sequences are shown in Figure 6.3.

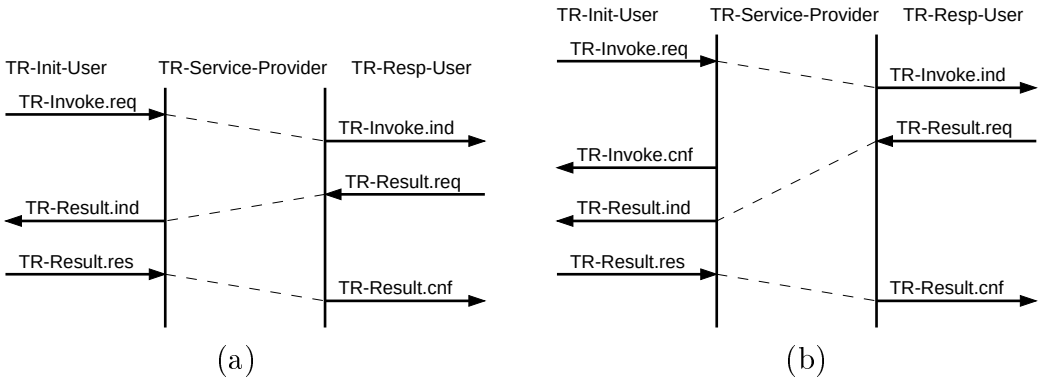

Figure 6.3: Basic primitive sequence for TR-Service with UserAck Off. (a) TR-Result.req implicitly acknowledges receipt of Invoke. (b) TR-Invoke.cnf delivered without explicit TR-Invoke.res from TR-Resp-User.

The sequence in Figure 6.3(a) differs from that in Figure 6.2 because the TR-Resp-User does not explicitly acknowledge the receipt of an invocation. Instead, the receipt of the result by the TR-Init-User implicitly acknowledges the invoke has been received.

The sequence in Figure 6.3(b) also represents a valid sequence from the primitive sequence table (Table 5.5). Here, a TR-Invoke.cnf primitive is delivered to the TR-Init-User before the TR-Result.ind primitive. However, the TR-Resp-User has not submitted a TR-Invoke.res primitive, and therefore the TR-Invoke.cnf is incorrect—it should confirm the response from the TR-Resp-User, i.e. the primitives should be end-to-end. This end-to-end principle is often used for service definitions. The OSI conventions [79] for using service primitives states in clause 6.1(b):

> *Only service primitives which relate to some element of the service involving service-users in the environment need to be considered. The interactions which are related only to local conventions between the service-user and service-provider do not have to be considered in a service definition. For example, strictly local functions could be provided in some implementations. As they do not involve other service-users, such functions are not visible outside the local system.* (page 8, [79])

This implies that the service definition is at a higher level of abstraction than, for example, just the interactions between a TR-User and TR-PE. Interactions between the

TR-User and TR-Service-Provider that are not end-to-end are called *local user events*. This convention is important in both the TR-Service and TR-Protocol (Chapter 7). We now make the following assumption:

Assumption 6.2 (End-to-End Behaviour). *The TR-Invoke and TR-Result primitives exhibit end-to-end behaviour. That is, the global sequence of primitive types must occur in the order: request, indication, response, then confirm. (A primitive type cannot occur before any of its predecessors have occurred.) However, it is not necessary for all of the primitive types to occur in a legal primitive sequence.* □

So far, Assumption 6.1 has been used to indicate the completion of a successful transaction. With UserAck Off, however, the TR-Users are not required to explicitly acknowledge the receipt of the invoke or the result. Therefore, the primitive sequence in Figure 6.4 is also possible.

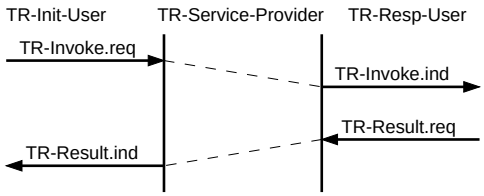

Figure 6.4: Basic primitive sequence for TR-Service with UserAck Off. TR-Users are not involved in acknowledgment of the result.

The successful completion of a transaction when UserAck is Off is now defined as:

Assumption 6.3 (Successful Transaction (UserAck Off)). *A successful transaction in the TR-Service (with UserAck Off) is any legal sequence defined in Table 5.5 that is complete when either:*

1. *the TR-Init-User has submitted the TR-Result.res primitive and the corresponding TR-Result.cnf primitive has been delivered to the TR-Resp-User, or*

2. *the TR-Resp-User has submitted the TR-Result.req primitive and the corresponding TR-Result.ind primitive has been delivered to the TR-Init-User.* □

Assumption 6.3 means that if the TR-Init-User has been delivered the TR-Result.ind primitive, and follows up with the submission of the TR-Result.res primitive, then the transaction is *not* successful if the TR-Result.cnf primitive is not delivered to the TR-Resp-User. Section 6.1.3 discusses the significance of such a transaction.

6.1.3 Aborted Transactions

The basic behaviour of the TR-Service does not consider transactions that are aborted. A transaction abort can be initiated by the TR-User (submitting a TR-Abort.req) or the TR-Service-Provider (delivering a TR-Abort.ind). The primitive sequence table (Table 5.5) specifies that a TR-Abort.req or TR-Abort.ind primitive can follow any primitive except themselves and a TR-Result.cnf primitive. In this sub-section, we discuss when it is appropriate for an abort to occur, and the impact that this has on the transaction.

An abort cannot occur before a TR-Invoke.req primitive because the transaction is not in progress. After a TR-Invoke.req primitive has been submitted by the TR-Init-User, aborts may occur.

In general, there are four cases when both TR-Users are aware that an abort has occurred:

1. A TR-User submits a TR-Abort.req and the peer TR-User is delivered a TR-Abort.ind (Figure 6.5(a)).

2. A TR-User submits a TR-Abort.req and the TR-Service-Provider initiates an abort by delivering a TR-Abort.ind to the peer TR-User (Figure 6.5(b)—Section 6.1.2 explains the wave between primitives).

3. Both users submit TR-Abort.req primitives (Figure 6.5(c)).

4. The TR-Service-Provider initiates an abort, delivering TR-Abort.ind primitives to both users (Figure 6.5(d)).

The above four abort scenarios represent an aborted transaction.

There are two special cases when aborts occur that also need consideration. The first special case is when an abort occurs after the TR-Init-User has submitted a TR-Invoke.req primitive, but before the TR-Resp-User has been notified of the invoke (TR-Invoke.ind has not yet occurred). Since the TR-Resp-User has no knowledge the transaction has started, the abort only needs to be known by the TR-Init-User. Figures 6.6(a) and 6.6(b) illustrate the two possible primitive sequences for this special case.

The second special case deals with aborts occurring after the TR-Init-User has submitted a TR-Result.res primitive. The primitive sequence table (Table 5.5) states a TR-Abort.req or TR-Abort.ind primitive may follow the TR-Result.res primitive at the TR-Init-User. This should not be possible because after the TR-Init-User has received the result (TR-Result.ind) and acknowledged it (TR-Result.res), then the transaction has been successfully completed. The TR-Init-User should not have to wait to see if the transaction is aborted after it has acknowledged the result. For example, we can consider

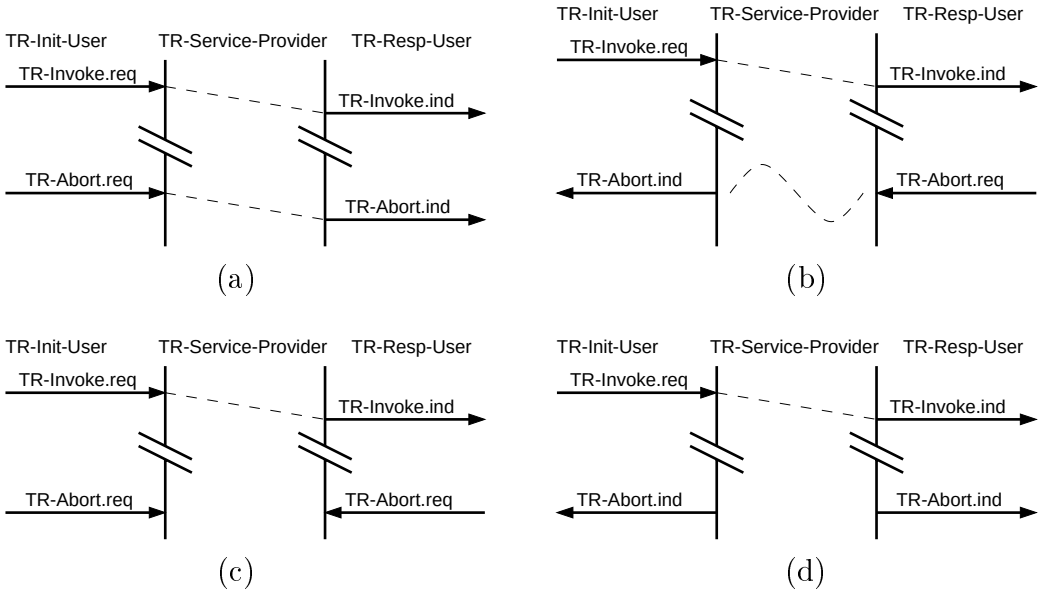

Figure 6.5: Abort primitive sequences for TR-Service. (a) TR-Abort.req followed by TR-User initiated TR-Abort.ind. (b) TR-Abort.req and TR-Service-Provider initiated TR-Abort.ind. (c) TR-Abort.req primitives from TR-Users. (d) TR-Service-Provider initiated TR-Abort.ind primitives.

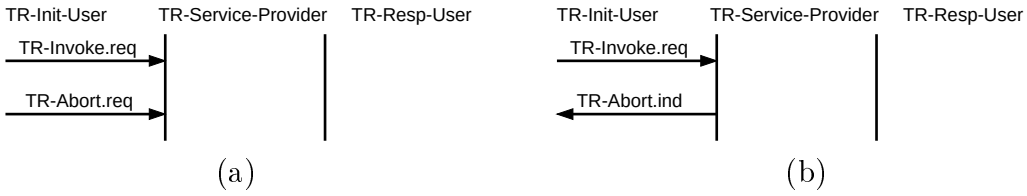

Figure 6.6: Abort primitive sequences for TR-Service when there is no interaction with the TR-Resp-User. (a) TR-Invoke.req followed by TR-Abort.req from TR-Init-User. (b) TR-Invoke.req followed by TR-Abort.ind initiated by TR-Service-Provider.

the WSP [181] which uses the TR-Service (with UserAck On) for the method invocation feature. The client (TR-Init-User) makes a method request (using the TR-Invoke primitives) and a result is returned (using the TR-Result primitives). After the client has submitted the TR-Result.res primitive, the client protocol entity in WSP returns to the NULL state. This means the TR-User does not allow any aborts to occur after the submission of the TR-Result.res primitive.

Although the TR-Init-User cannot submit a TR-Abort.req primitive after the TR-Result.res, local user events can still occur. For example, the TR-Init-PE maintains state information after the TR-Result.res is submitted. The TR-Init-User may indicate to the TR-Init-PE to delete this state information. However, this does not correspond to a TR-Abort.req being submitted, as primitives have end-to-end intent, rather than local intent.

When UserAck is Off, a successful transaction may be completed after the TR-Init-User has been delivered the TR-Result.ind primitive, or the TR-Resp-User has been delivered the TR-Result.cnf primitive (Assumption 6.3). However, Table 5.5 allows TR-Abort primitives after the TR-Result.req and TR-Result.ind primitives. This is satisfactory because the TR-Resp-User may attempt to abort the transaction after submitting the TR-Result.req primitive, to notify the TR-Init-User that it is not expecting acknowledgment of the result. Similarly, the TR-Init-User may submit an TR-Abort.req primitive after being delivered the TR-Result.ind primitive to notify the TR-Resp-User the transaction has been aborted and an acknowledgment of the result (TR-Result.cnf) should not be expected.

The discussion on aborts is summarized in Assumptions 6.4 to 6.8. Assumptions 6.4 and 6.5 include the possibilities of the TR-Resp-User submitting TR-Abort.req primitives after the TR-Result.req primitive. These override Assumptions 6.3 and 6.1, respectively.

Assumption 6.4 (Successful Transaction (UserAck Off)). *A successful transaction in the TR-Service (with UserAck Off) is any legal sequence defined in Table 5.5 that is complete when either:*

1. *the TR-Init-User has submitted the TR-Result.res primitive and the corresponding TR-Result.cnf primitive has been delivered to the TR-Resp-User;*

2. *the TR-Resp-User has submitted the TR-Result.req primitive and the corresponding TR-Result.ind primitive has been delivered to the TR-Init-User;*

3. *the TR-Init-User has submitted the TR-Result.res primitive, and the TR-Resp-User has submitted the TR-Result.req primitive followed by the submission of a TR-Abort.req primitive or delivery of a TR-Abort.ind primitive; or*

4. *the TR-Resp-User has submitted the TR-Result.req primitive followed by the submission of a TR-Abort.req primitive or delivery of a TR-Abort.ind primitive, and the TR-Init-User has been delivered the TR-Result.ind primitive.* □

Assumption 6.5 (Successful Transaction (UserAck On)). *A successful transaction in the TR-Service (with UserAck On) is any legal sequence defined in Table 5.5 that is complete when either:*

1. *the TR-Init-User has submitted the TR-Result.res primitive and the corresponding TR-Result.cnf primitive has been delivered to the TR-Resp-User; or*

2. *the TR-Init-User has submitted the TR-Result.res primitive, and the TR-Resp-User has submitted the TR-Result.req primitive followed by the submission of a TR-Abort.req primitive or delivery of a TR-Abort.ind primitive.* □

Assumption 6.6 (Aborted Transaction). *An aborted transaction in the TR-Service is any legal sequence defined in Table 5.5 that is complete when either:*

1. *a TR-User has submitted a TR-Abort.req primitive and the peer TR-User has been notified of it by being delivered a TR-Abort.ind primitive;*

2. *a TR-User has submitted a TR-Abort.req primitive and the peer TR-User has been delivered a TR-Abort.ind primitive that was initiated by the TR-Service-Provider;*

3. *both TR-Users have submitted a TR-Abort.req primitive;*

4. *both TR-Users have been delivered a TR-Abort.ind primitive that was initiated by the TR-Service-Provider; or*

5. *the TR-Init-User has submitted a TR-Invoke.req primitive which is immediately followed by the submission of a TR-Abort.req primitive by the TR-Init-User, or the delivery of a TR-Abort.ind primitive by the TR-Service-Provider to the TR-Init-User, and the TR-Service-Provider has not delivered the TR-Invoke.ind primitive to the TR-Resp-User.* □

Assumption 6.7 (TR-Aborts at the TR-Init-User). *TR-Abort primitives (TR-Abort.req and TR-Abort.ind) cannot occur before the TR-Init-User has submitted a TR-Invoke.req primitive, nor after the TR-Init-User has submitted a TR-Result.res primitive.* □

Assumption 6.8 (TR-Aborts at the TR-Resp-User). *TR-Abort primitives (TR-Abort.req and TR-Abort.ind) cannot occur before the TR-Resp-User has been delivered a TR-Invoke.ind primitive, nor after the TR-Resp-User has been delivered the TR-Result.cnf primitive.* □

6.2 Transaction Service CPN

The objective of modelling the TR-Service using CPNs is to obtain all possible global sequences of primitives that constitute a transaction, i.e. the TR-Service language. Ideally, any formalism that allows sequences to be specified could be used (e.g. regular languages and expressions, FSA, Backus Naur Form (BNF)). However, it is difficult to determine the sequences *a priori*, and have confidence that all sequences have been obtained and they are indeed correct. Therefore, we use CPNs to model the TR-Service as the TR-Users communicating in a distributed system. The model allows us to automatically generate all possible sequences (i.e. obtain a FSA), and to validate that they accurately represent the behaviour in practice. This section describes the TR-Service CPN. The approach taken, and structure of the model, is based on a model of the OSI Transport Service in [13].

6.2.1 Scope of the TR-Service CPN

Only one transaction (from the TR-Init-User's point of view) is modelled by the TR-Service CPN. In terms of sequences of primitives, most transactions are independent of each other. Exceptions may occur when the TID space wraps, but, as we discuss in Chapter 7, it is still sufficient to consider one transaction.

Reducing the TR-Service CPN to model just one transaction allows us to ignore the Source and Destination Addresses and Ports in the primitive parameters. Similarly, the following parameters are not modelled:

- User Data is the data submitted to WTP for transmission. The content is not manipulated or used in WTP. Similarly for Exit Info, which is data intended only for the user.
- The Class Type is always 2 for our CPN models.
- The Handle is used by the higher layer to identify the transaction. Again, considering only one transaction, this has no effect on the primitive sequences.
- The Abort Code indicates the reason for aborting. Within one transaction, no further action is taken based on the reason for an abort.

The only parameter modelled is Ack-Type in the TR-Invoke.req and TR-Invoke.ind primitives. Section 6.2.4 describes how this is done. The remainder of the information conveyed to the peer TR-User by the primitive and its parameters (e.g. TR-Invoke.req{SrcAdr,DestAdr,...}) can be modelled as a message (e.g. Invoke). This is also described in Section 6.2.4.

The assumptions given in Section 6.1 are also used in creating the CPN model of the TR-Service. Recall that Assumptions 6.3 and 6.1 are not used—they are overwritten by Assumptions 6.4 and 6.5, respectively.

6.2.2 Structure of the TR-Service CPN

Figure 6.7 shows the hierarchy page of the TR-Service CPN. There are four other pages in the model: the declarations (Section 6.2.3), the CPN pages InvokeResult and Abort (Section 6.2), and Standard ML code for analysing the state space (discussed in Section 6.3).

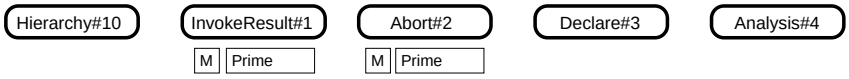

Figure 6.7: Hierarchy page for the TR-Service CPN

The model is divided into two CPN pages: InvokeResult models the basic behaviour of the TR-Service (i.e. the TR-Invoke and TR-Result primitives); and Abort models the TR-Abort primitives. This structure is to improve clarity of the model (rather than having all 14 transitions on one page). The pages are connected by fusion places. This structure has been chosen because we believe it is the clearest approach for our purpose of modelling the TR-Service, namely generating the global sequences of primitives. A top-level CPN page may show how the pages are related, but would add no benefit in showing the sequences of primitives, and hence, is omitted.

6.2.3 Declarations

The declarations for the TR-Service CPN are given in Listing 6.1.

Listing 6.1: Declarations for the TR-Service CPN

```
1 color Istate = with Uinvoke | Pinvokeack | Presult | Uresultack | Icomplete;
2 color Rstate = with Pinvoke | Uinvokeack | Uresult | Presultack | Rcomplete;
3 color Message = with Invoke | Result | Ack | NoAck | Abort;
4 color UserAck = with On | Off;
5 var i : Istate;
6 var r : Rstate;
7 var u : UserAck;
```

There are four types used in the CPN. Istate and Rstate define the state of the interfaces between the TR-User and the TR-Service-Provider at the initiating and responding sides. Message defines the messages that are used for communication within the TR-Service-Provider. UserAck defines the parameter Ack-Type used in the TR-Invoke primitive for determining whether UserAck is On or Off. The variables i and r are used to non-deterministically select states. The variable u is used to specify the value of the

UserAck parameter. Further details on the declarations will become apparent when the CPN model is described.

6.2.4 Page Structure

The pages in the TR-Service CPN model have similar structures. The InvokeResult page, given in Figure 6.8, is used to illustrate this structure. There are:

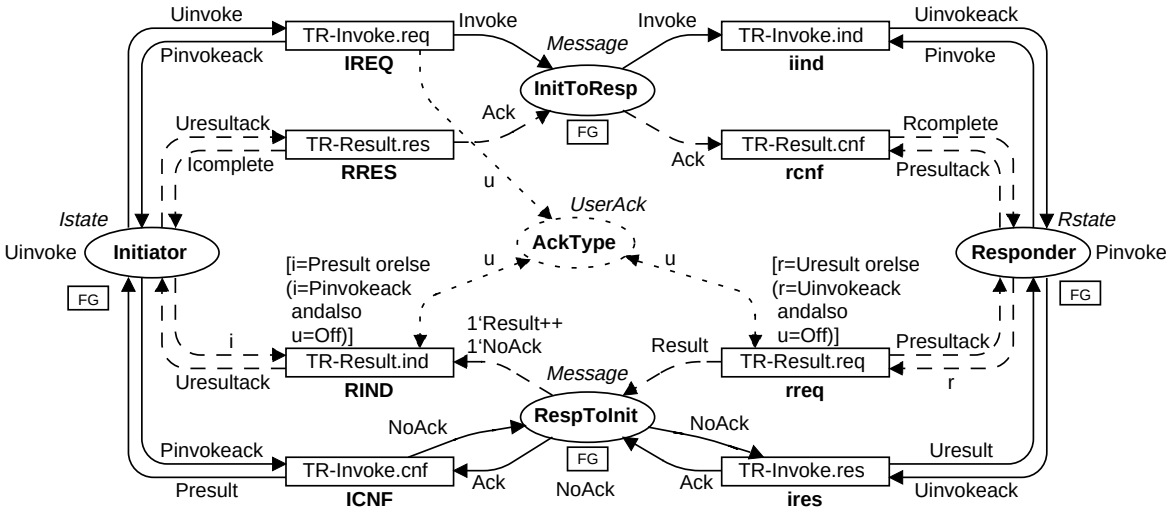

Figure 6.8: InvokeResult page for TR-Service CPN

- Two fusion places called Initiator and Responder that control the sequences of primitives at the interface between the TR-User and the TR-Service-Provider. The marking of these places can be thought of as the state of the interface. The Initiator and Responder places are typed by the colour sets Istate and Rstate, respectively. Table 6.1 lists the significance of each state. In general, the names of the states indicate which entity a message is expected from (U for TR-User or P for TR-Service-Provider), and the type of message (e.g. invoke, result, or acknowledgment). The two exceptions are Icomplete and Rcomplete that indicate the sequence is complete at the Initiator or Responder interface, respectively.

- Two fusion places called InitToResp and RespToInit representing the TR-Service-Provider storage. These places are typed by the colour set Message.

- Transitions that represent the submitting and delivering of different primitives from and to a TR-User. These are referred to as *primitive transitions*. The primitive names are given as the labels inside the transitions. The transition names are mnemonics given in bold face below the transitions. They follow the conventions used in [13]. The first letter in the mnemonic is the first letter of the primitive name (*i*nvoke, *r*esult, *a*bort). The following three letters indicate the primitive type (*req*,

State	*Side*	*Significance*
Uinvoke	Initiator	Waiting on the TR-Init-User to submit an invoke (TR-Invoke.req).
Pinvokeack	Initiator	Waiting on the TR-Service-Provider to deliver an acknowledgment of the invoke (TR-Invoke.cnf). If UserAck is Off, the delivery of the result is also acceptable (TR-Result.ind).
Presult	Initiator	Waiting on the TR-Service-Provider to deliver the result (TR-Result.ind).
Uresultack	Initiator	Waiting on the TR-Init-User to submit an acknowledgment of the result (TR-Result.res).
Icomplete	Initiator	The transaction primitive sequence is complete from the TR-Init-User's point of view.
Pinvoke	Responder	Waiting on the TR-Service-Provider to deliver an invoke (TR-Invoke.ind).
Uinvokeack	Responder	Waiting on the TR-Resp-User to submit an acknowledgment of the invoke (TR-Invoke.res). If UserAck is Off, the submission of the result is also acceptable (TR-Result.req).
Uresult	Responder	Waiting on the TR-Resp-User to submit the result (TR-Result.req).
Presultack	Responder	Waiting on the TR-Service-Provider to deliver an acknowledgment of the result (TR-Result.cnf).
Rcomplete	Responder	The transaction primitive sequence is complete from the TR-Resp-User's point of view.

Table 6.1: Significance of interface states in the TR-Service CPN

ind, *res*, *cnf*). The mnemonics of the primitives seen by the TR-Init-User are in uppercase, while those seen by the TR-Resp-User are in lowercase. Table 6.2 gives the significance of each transition, including those on the Abort page which is described in Section 6.2.6. The guards on some of the transitions will be discussed in Sections 6.2.5 and 6.2.6.

- Arcs from the interface places (Initiator and Responder) to the transitions, with inscriptions specifying the required state of the interface for the transition to occur.

- Arcs from the transitions to the interface places, with inscriptions specifying the next state of the interface after the transition occurs.

- Arcs between the transitions and TR-Service-Provider places (InitToResp and RespToInit), with inscriptions specifying the information to be transferred via the TR-Service-Provider. The information is that which is conveyed by the service primitive and its parameters, and is referred to as a *message*. Input arcs to the TR-Service-Provider places show the message that is stored by the TR-Service-Provider. Output arcs show the message that is delivered by the TR-Service-Provider to the TR-User. The two arcs inscribed with NoAck are discussed in Section 6.2.5. Table 6.3 gives the significance of each message.

- A place called AckType (on the InvokeResult page only) that models the TR-Invoke

Transition	*Primitive Type*	*Significance*
IREQ	TR-Invoke.req	Invoke request submitted by the TR-Init-User
iind	TR-Invoke.ind	Invoke indication delivered to the TR-Resp-User
ires	TR-Invoke.res	Invoke response submitted by the TR-Resp-User
ICNF	TR-Invoke.cnf	Invoke confirm delivered to the TR-Init-User
rreq	TR-Result.req	Result request submitted by the TR-Resp-User
RIND	TR-Result.ind	Result indication delivered to the TR-Init-User
RRES	TR-Result.res	Result response submitted by the TR-Init-User
rcnf	TR-Result.cnf	Result confirm delivered to the TR-Resp-User
AREQ	TR-Abort.req	Abort request submitted by the TR-Init-User
AINDP	TR-Abort.ind	TR-Service-Provider initiated Abort indication delivered to the TR-Init-User
AIND	TR-Abort.ind	TR-Resp-User initiated Abort indication delivered to the TR-Init-User
areq	TR-Abort.req	Abort request submitted by the TR-Resp-User
aindp	TR-Abort.ind	TR-Service-Provider initiated Abort indication delivered to the TR-Resp-User
aind	TR-Abort.ind	TR-Init-User initiated Abort indication delivered to the TR-Resp-User

Table 6.2: Significance of transitions in the TR-Service CPN

Message	*Significance*
Invoke	Communicates the TR-Invoke.req parameters from the initiating side to the responding side.
Result	Communicates the TR-Result.req parameters from the responding side to the initiating side.
Ack	Communicates the TR-Invoke.res parameters from the responding side to the initiating side, and communicates the TR-Result.res parameters from the initiating side to the responding side.
Abort	Communicates the TR-Abort.req parameters between the two sides of the TR-Service-Provider.

Table 6.3: Significance of TR-Service-Provider messages in the TR-Service CPN

parameter AckType. AckType is typed by the UserAck colour set—it can be marked with an On token or an Off token. The AckType parameter is set in the TR-Invoke.req primitive by the TR-Init-User and notified to the TR-Resp-User in the TR-Invoke.ind primitive. The AckType (UserAck On or UserAck Off) remains the same for the complete transaction. Rather than modelling the parameter being passed from the TR-Init-User to the TR-Resp-User, we use the place AckType, which has global significance, to indicate the status of UserAck. Where transitions depend on the status of UserAck, there is an arc from AckType to that transition. AckType is marked with either an On or Off token when IREQ occurs. In the analysis we deal with the UserAck On and Off cases separately. Therefore, we change the inscription of the arc from IREQ to AckType to either On or Off, depending on the scenario we are interested in.

The CPN has an initial marking of:

- Initiator: 1‘Uinvoke
- Responder: 1‘Pinvoke
- InitToResp: the empty multi-set
- RespToInit: 1‘NoAck
- AckType: the empty multi-set

The marking of RespToInit is explained in Section 6.2.5 which describes the InvokeResult page. The Abort page is described in Section 6.2.6. We assume UserAck is Off (i.e. the arc inscription from IREQ into AckType is Off, not u). The differences when UserAck is On will be noted.

6.2.5 InvokeResult Page

The InvokeResult page models the TR-Invoke and TR-Result primitives (Figure 6.8). The TR-Invoke primitives are modelled by the four outer transitions (referred to as TR-Invoke transitions) and the TR-Result primitives are modelled by the four inner transitions (referred to as TR-Result transitions). In the initial marking, IREQ is the only transition enabled (Initiator is marked with Uinvoke). On the occurrence of IREQ the interface enters the Pinvokeack state, and an Invoke message is added to InitToResp. The place AckType is also marked with 1‘Off. The transition iind is now enabled because Responder is marked with a Pinvoke token and InitToResp is marked with an Invoke token. After the occurrence of iind, Responder is marked with Uinvokeack.

With UserAck Off, the TR-Resp-User can either submit a TR-Invoke.res primitive or a TR-Result.req primitive after being delivered a TR-Invoke.ind primitive. When UserAck is On however, a TR-Result.req primitive cannot be submitted by the TR-Resp-User before the TR-Invoke.res primitive has been submitted. This difference in behaviour is modelled using the place AckType and the guard on transition rreq. If ires occurs first, then rreq is enabled with r bound to Uresult. This is independent of the status of UserAck. If UserAck is Off, then rreq (r bound to Uinvokeack and u bound to Off) is in conflict with ires. If rreq occurs, then Responder is marked with the token Presultack, disabling ires. If ires and then rreq occur, then the place RespToInit has the marking 1`Ack++1`Result. The receipt of these messages at the initiating side is modelled in a similar way to ires and rreq. ICNF is enabled in the Pinvokeack interface state, and RIND is enabled in the Presult or (if u is bound to Off) Pinvokeack interface state.

Note that the NoAck is necessary in RespToInit to ensure that if both ires and rreq have occurred, then ICNF must occur before RIND. That is, RIND can only occur if there is a Result token in RespToInit, but no Ack token. We model the absence of Ack by NoAck.

After the RIND transition has occurred, the only transition enabled (for both UserAck On and Off) is RRES. The occurrence of RRES deposits an Ack in InitToResp. This enables rcnf. TR-Result.cnf is the final transition on the InvokeResult page that can occur, after which the Initiator place is marked with a Icomplete token and the Responder place is marked with a Rcomplete token.

When UserAck is Off a transaction can also be successfully completed after the TR-Init-User has received the TR-Result.ind primitive (Assumption 6.4). The completion of this sequence is not modelled explicitly in the CPN (i.e. the interface states do not become Icomplete and Rcomplete), but is considered in the FSA analysis. This is discussed further in Section 6.3.

6.2.6 Abort Page

The TR-Abort primitives are modelled on the Abort page (Figure 6.9). There are six transitions (referred to as TR-Abort transitions), all with input arcs from the interface places (Initiator and Responder) inscribed with the variable i or r. The two transitions AINDP and aindp have a fifth letter (p) in their name to indicate the TR-Abort.ind primitive is only initiated by the TR-Service-Provider. Each of the variables i and r can be bound to any token in the colour set of its type (Istate or Rstate). The enabling of the transitions are however restricted by guards. At the initiating side, the guards specify the TR-Abort transitions are not enabled when the interface is in the Uinvoke or Icomplete states. Similarly, at the responding side, the TR-Abort transitions are not enabled in the Pinvoke and Rcomplete states. This means a TR-Abort transition cannot occur before the

transaction has begun (TR-Invoke.req submitted by the TR-Init-User or TR-Invoke.ind delivered to the TR-Resp-User), nor after the transaction has been aborted or successfully completed (TR-Result.res submitted by the TR-Init-User or TR-Result.cnf delivered to the TR-Resp-User). This corresponds with Assumptions 6.7 and 6.8.

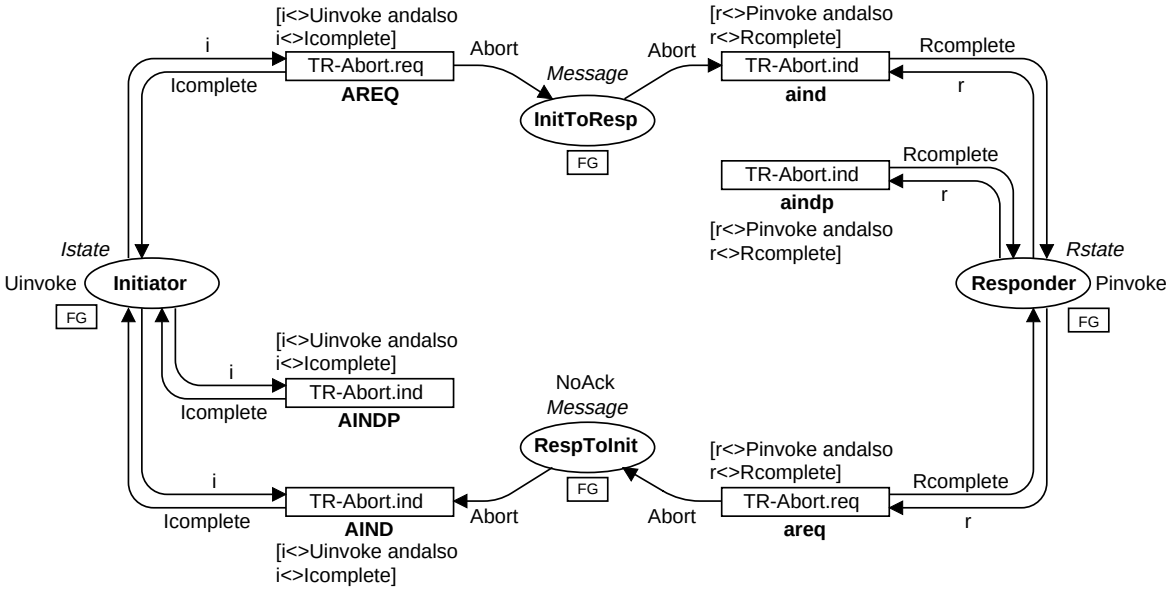

Figure 6.9: Abort page in TR-Service CPN

The TR-Abort transitions are symmetrical. Therefore, we only describe the TR-Abort transitions on the initiating side.

The TR-Init-User can submit a TR-Abort.req primitive when a transaction is in progress. This is modelled by the transition AREQ, which delivers an Abort message to the TR-Service-Provider (place InitToResp). After the TR-Init-User has aborted, the transaction is complete (from the TR-Init-User's point of view) and the interface enters the Icomplete state.

The TR-Init-User can be delivered a TR-Abort.ind primitive (transition AIND) as a result of the TR-Resp-User submitting a TR-Abort.req primitive which creates an Abort message in RespToInit. The Abort message enables AIND, and on its occurrence the TR-Abort.ind primitive is delivered to the TR-Init-User. The interface enters the Icomplete state indicating the completion of the transaction.

Finally, the TR-Service-Provider can initiate an abort, delivering a TR-Abort.ind primitive to the TR-Init-User. This is modelled by the transition AINDP. After the TR-Abort.ind primitive is delivered to the TR-Init-User, the interface enters the Icomplete state indicating the completion of the transaction.

6.3 Transaction Service Analysis

The TR-Service CPN has been analysed using the methodology outlined in Section 3.3. In this section we present the state space and language analysis results.

6.3.1 State Space Analysis

State space analysis of the TR-Service CPN was performed separately for the UserAck Off and UserAck On scenarios. We chose to separate the scenarios so the TR-Protocol could be compared to each individually. That is, separate state spaces for the TR-Protocol when UserAck is Off and On are also calculated. This reduced the state space size for the TR-Protocol CPN (Chapter 7). Table 6.4 gives the statistics for each TR-Service state space. Design/CPN reports are included in Appendix C.

UserAck	*Nodes*	*Arcs*	*Terminal Markings*
Off	60	129	22
On	57	114	22

Table 6.4: State space statistics for the TR-Service CPN

The terminal markings for the two state spaces are shown in Figures 6.10 and 6.11. The multiple terminal markings are comprised of different combinations of the markings of places InitToResp and RespToInit. The markings are a result of different sequences of primitives leading to the completion of a transaction. For example, if both TR-Users submit TR-Abort.req primitives, then the transaction is complete, but Abort messages remain in the places InitToResp and RespToInit. However, if the Abort message was received from InitToResp and a TR-Abort.ind primitive delivered to the TR-Resp-User, then the places InitToResp and RespToInit would be empty. Although the many different terminal markings result in a cumbersome state space, the language analysis does not depend on the specific markings, and therefore the results will be more compact.

6.3.2 Language Analysis

The TR-Service state space was treated as a FSA to obtain the TR-Service language. The process described in Chapter 4 is used. The first step involves mapping binding elements in the Design/CPN state space to the alphabet of the language. The functions that performs the mapping are given in Listing 6.2.

The function be2str() takes a binding element as input and returns an integer[1]. The binding elements in the TR-Service CPN represent service primitives. Each primitive is mapped to the integer according to Table 6.5. Note that the TR-Abort primitives at

[1]The integer is given in Standard ML string format so that is can be printed to a file.

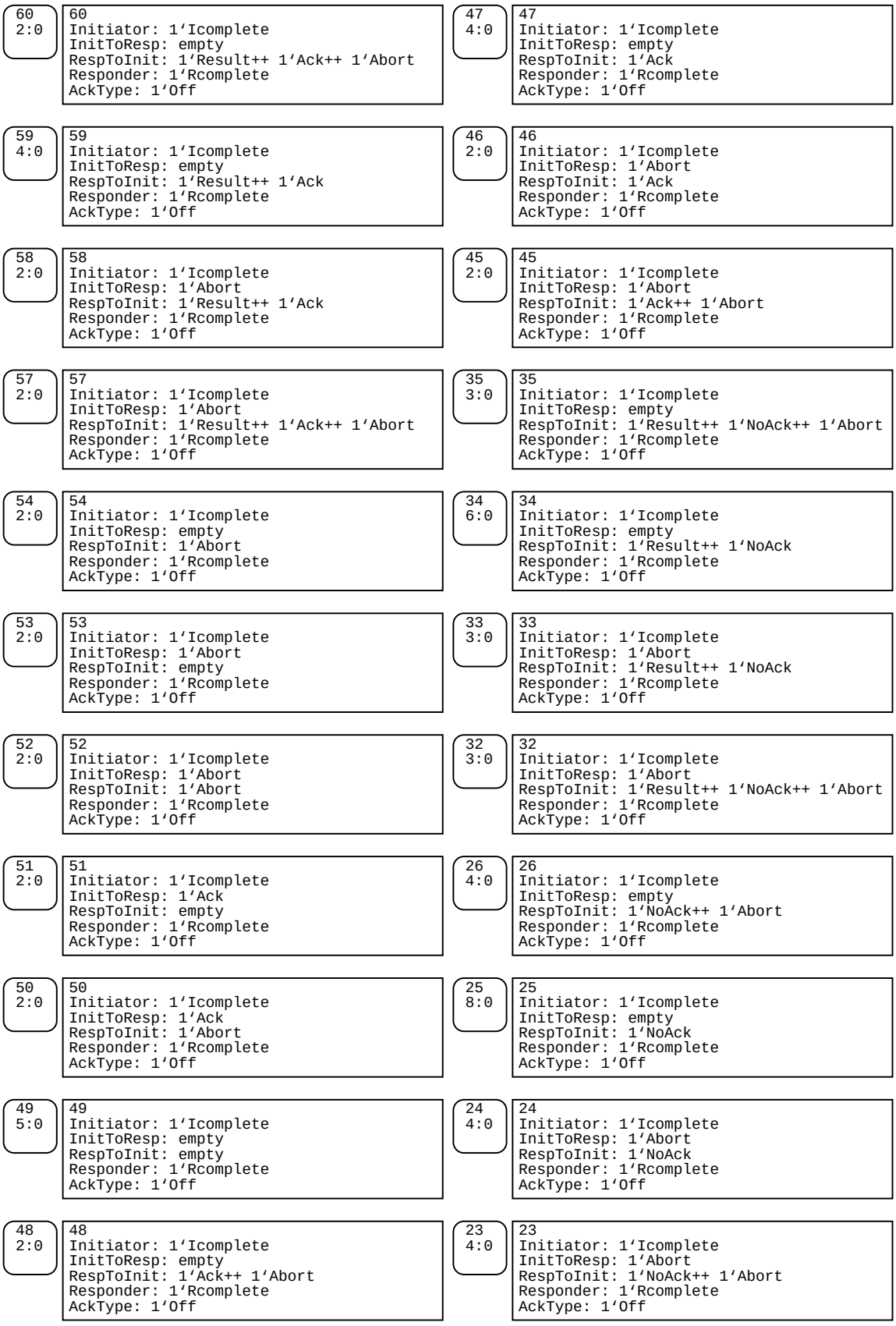

Figure 6.10: Terminal markings for TR-Service CPN with UserAck Off

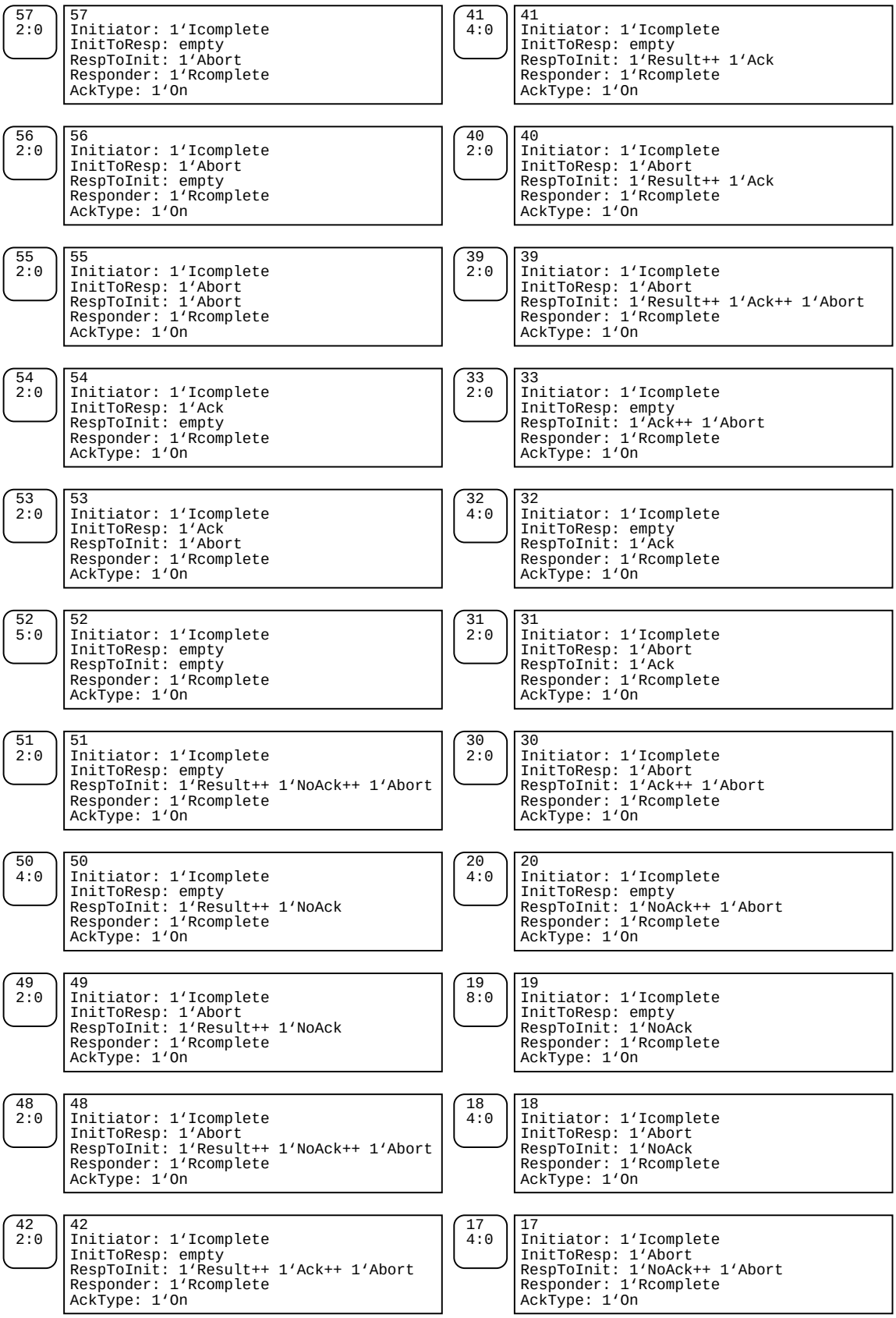

Figure 6.11: Terminal markings for TR-Service CPN with UserAck On

Listing 6.2: Mapping function used in the TR-Service

```
1  fun be2str (Bind.InvokeResult'IREQ (1,_))      = "1"
2    | be2str (Bind.InvokeResult'iind  (1,_))     = "2"
3    | be2str (Bind.InvokeResult'ires  (1,_))     = "3"
4    | be2str (Bind.InvokeResult'ICNF (1,_))      = "4"
5    | be2str (Bind.InvokeResult'rreq  (1,_))     = "5"
6    | be2str (Bind.InvokeResult'RIND (1,_))      = "6"
7    | be2str (Bind.InvokeResult'RRES (1,_))      = "7"
8    | be2str (Bind.InvokeResult'rcnf  (1,_))     = "8"
9    | be2str (Bind.Abort'AREQ (1,_))             = "9"
10   | be2str (Bind.Abort'areq  (1,_))            = "10"
11   | be2str (Bind.Abort'AIND (1,_))             = "11"
12   | be2str (Bind.Abort'aind  (1,_))            = "12"
13   | be2str (Bind.Abort'AINDP (1,_))            = "11"
14   | be2str (Bind.Abort'aindp (1,_))            = "12"
15   | be2str (_) = "ERROR";
16
17 fun ArcToFSM a = be2str(ArcToBE(a));
18
19 fun FindHalts n =
20         (((Mark.InvokeResult' Initiator   1 n) == 1`Icomplete    ) andalso
21          ((Mark.InvokeResult'Responder  1 n) == 1`Pinvoke       ))
22         orelse
23         (((Mark.InvokeResult' Initiator   1 n) == 1`Icomplete    ) andalso
24          ((Mark.InvokeResult'Responder  1 n) == 1`Rcomplete    ))
25         orelse
26         (((Mark.InvokeResult' Initiator   1 n) == 1`Uresultack   ) andalso
27          ((Mark.InvokeResult'Responder  1 n) == 1`Presultack   ) andalso
28          ((Mark.InvokeResult'AckType    1 n) == 1`Off          ))
29         orelse
30         (((Mark.InvokeResult' Initiator   1 n) == 1`Uresultack   ) andalso
31          ((Mark.InvokeResult'Responder  1 n) == 1`Rcomplete    ) andalso
32          ((Mark.InvokeResult'AckType    1 n) == 1`Off          ));
```

the initiating side correspond to different integers than the TR-Abort primitives at the responding sides.

Service Primitive	*TR-User*	*Number*
TR-Invoke.req	TR-Init-User	1
TR-Invoke.ind	TR-Resp-User	2
TR-Invoke.res	TR-Resp-User	3
TR-Invoke.cnf	TR-Init-User	4
TR-Result.req	TR-Resp-User	5
TR-Result.ind	TR-Init-User	6
TR-Result.res	TR-Init-User	7
TR-Result.cnf	TR-Resp-User	8
TR-Abort.req	TR-Init-User	9
TR-Abort.req	TR-Resp-User	10
TR-Abort.ind	TR-Init-User	11
TR-Abort.ind	TR-Resp-User	12

Table 6.5: Correspondence of service primitives to numbers in the FSA

A FSA must contain at least one halt state, which defines the end of a sequence. A halt state may or may not lead to other states. When treating the state space as a FSA, all the terminal markings correspond to halt states. We introduce additional halt states to include the sequences that complete before a dead marking in the state space is reached. Nodes in the state space are defined as halt states if they satisfy any of the following conditions:

1. M(Initiator) = 1‘Icomplete and M(Responder) = 1‘Pinvoke: A TR-Invoke.req primitive is immediately followed by a TR-Abort.req or TR-Abort.ind (at the initiating side interface). This corresponds to Item 5 in Assumption 6.6.

2. M(Initiator) = 1‘Icomplete and M(Responder) = 1‘Rcomplete: A transaction has been successfully completed or aborted. The successful transaction corresponds with Items 1 and 3 of Assumption 6.4 when UserAck is Off, and Assumption 6.5 when UserAck is On. The aborted transaction corresponds to Items 1 to 4 of Assumption 6.6.

3. M(Initiator) = 1‘Uresultack and M(Responder) = 1‘Presultack and M(AckType) = 1‘Off: With UserAck Off, a transaction has been successfully completed, without the TR-Init-User acknowledging the result. This corresponds to Item 2 of Assumption 6.4.

4. M(Initiator) = 1‘Uresultack and M(Responder) = 1‘Rcomplete and M(AckType) = 1‘Off: With UserAck Off, a transaction has been successfully completed, without the TR-Init-User acknowledging the result, but with a TR-Abort primitive occurring at the TR-Resp-User. This corresponds to Item 4 of Assumption 6.4.

The minimized FSA for the UserAck Off and UserAck On state spaces are shown in Figures 6.12 and 6.13, respectively. The convention of double circle halt states is used [9]. The FSA and language statistics are given in Table 6.6. Each language has two sequences that have a length of 2: TR-Invoke.req followed by TR-Abort.req (by TR-Init-User); and TR-Invoke.req followed by TR-Abort.ind (to TR-Init-User). The longest sequences are of length 8. Closer investigation of the languages reveals the language with UserAck On is a subset of the UserAck Off language. This is expected because when UserAck is On, the TR-Users must acknowledge the invocation and result. When UserAck is Off, the TR-Users may either send acknowledgments or not send acknowledgments.

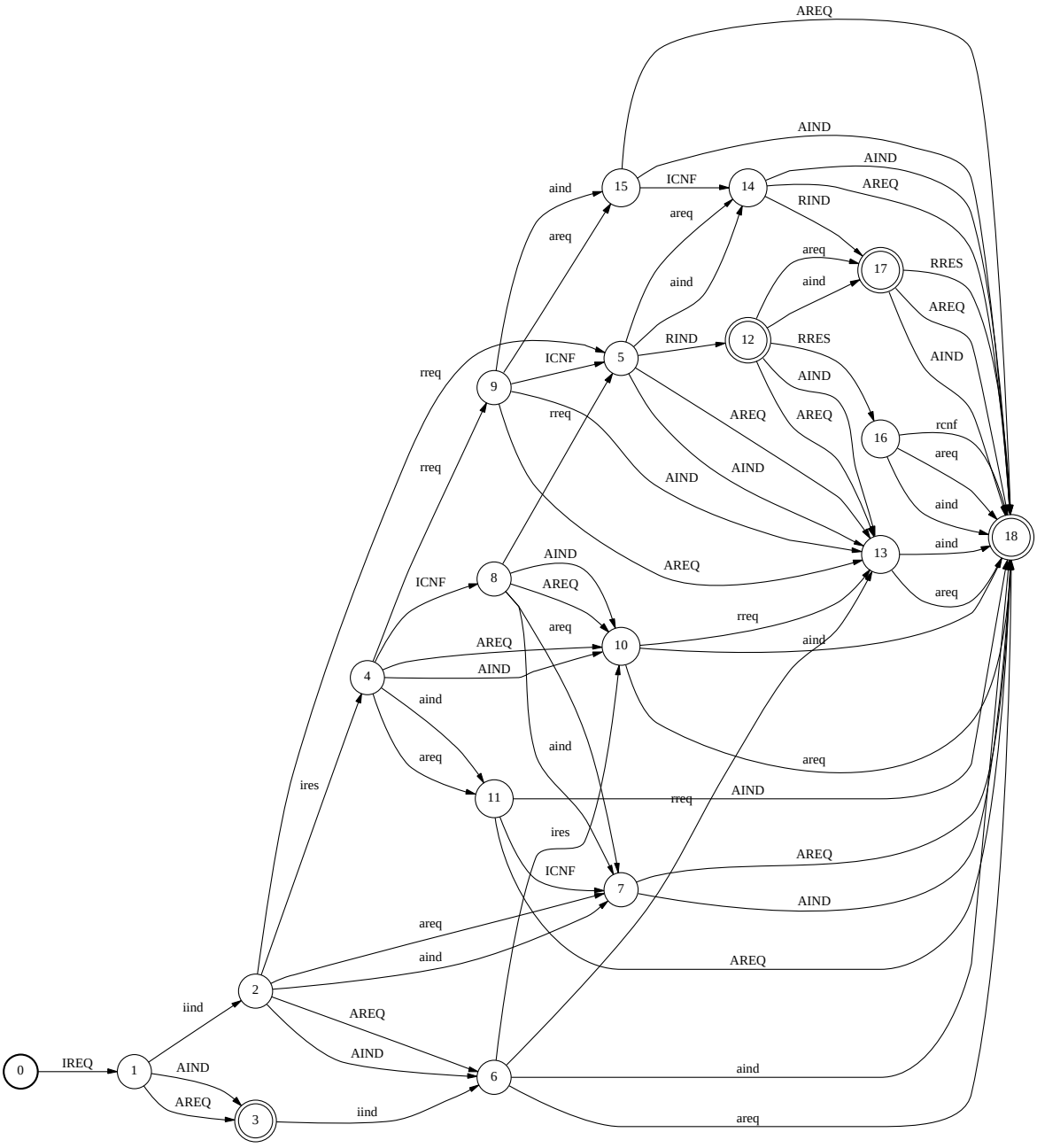

Figure 6.12: FSA of the TR-Service CPN with UserAck Off

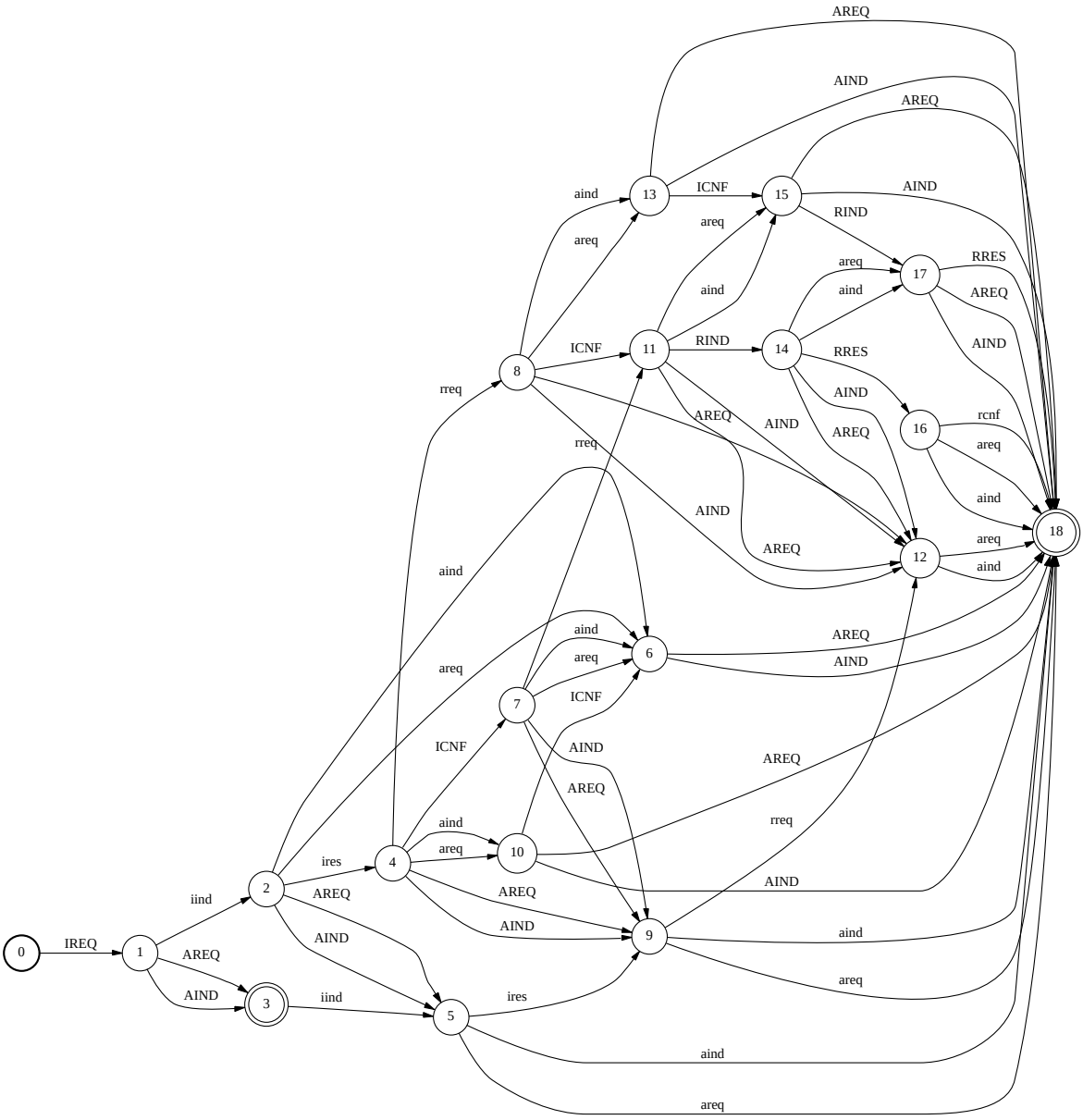

Figure 6.13: FSA of the TR-Service CPN with UserAck On

UserAck	*Nodes*	*Arcs*	*Halts*	*Sequences*	*Longest*	*Shortest*
Off	19	63	4	182	8	2
On	19	61	2	130	8	2

Table 6.6: Statistics for both TR-Service languages

6.4 Summary

The TR-Service in the WTP Specification [183] comprises the definition of the primitives and their parameters, and the set of possible primitive sequences as seen by both TR-Users. We have shown in this Chapter that the TR-Service definition in the WTP Specification inadequately describes the primitive sequences. We have presented a set of assumptions that we make about the TR-Service so that an unambiguous and (we believe) correct definition is obtained. In Section 6.2, a CPN model of the TR-Service is given, which formalizes the definition of the TR-Service. Section 6.3 has shown the results from analysing the CPN model. The FSAs of the TR-Service with UserAck On (Figure 6.13) and Off (Figure 6.12) give the set of possible global primitive sequences.

The main limitation of the TR-Service definition in the WTP Specification is that the primitive sequence table (Table 5.5) only describes the sequences from the point of view of one TR-User. The lack of information on the global sequences (e.g. what constitutes the end of a transaction, when can aborts occur?) allows different interpretations of the TR-Service to be made. In Section 6.1, we have made assumptions about the end-to-end behaviour of the TR-Service and the definition of successful and aborted transactions (including when aborts can occur). These assumptions are based on the TR-Service given in the WTP Specification, the behaviour of the TR-Protocol, existing conventions for defining services (e.g. [79]), and our understanding of the purpose of the Transaction layer in the WAP architecture.

A CPN model of the TR-Service has been created. The objective of the CPN modelling was to generate the set of possible primitive sequences, and so, with this in mind, several assumptions about the primitive parameters were made to simplify the CPN. The CPN uses transitions to represent primitives being submitted by, or delivered to, the TR-User. Therefore, the state spaces for the two initial markings of the CPN model (UserAck On and UserAck Off) gave all possible sequences of the primitives. By treating the state space as a FSA, using the assumptions about successful and aborted transactions to define halt states, and then minimizing that FSA, the canonical set of possible primitive sequences, or the TR-Service language, was obtained. The FSA of the TR-Service with UserAck Off (Figure 6.12) has 19 nodes and 63 arcs, giving a language with 182 different sequences of primitives. With UserAck On there are 19 nodes, 61 arcs and 130 sequences of primitives (Figure 6.13). The FSAs are an integral part of the remaining verification steps. In the next chapter, the Transaction Protocol will be modelled using CPNs. In Chapter 8 the TR-Protocol FSAs, obtained from the analysis of the CPN, will be compared with the TR-Service FSAs to determine if the TR-Protocol provides the defined TR-Service.

Chapter 7

Transaction Protocol CPN

The Transaction Protocol [183] (TR-Protocol) described in Chapter 5 gives details of the procedures the two protocol entities, Initiator and Responder, use to communicate with each other. This should implement the Transaction Service (TR-Service) given in Chapter 6. This chapter presents the CPN model of the TR-Protocol.

The CPN modelling and analysis of the TR-Protocol, like the TR-Service, went through several iterations. Firstly, we created an initial CPN that was based mainly on the state tables in the WTP Specification [183]. Analysis of this initial CPN revealed inconsistencies between the TR-Protocol and TR-Service. The TR-Protocol CPN was modified and then analysed again. This process was repeated until we arrived at a final model of the protocol (called the *Revised* TR-Protocol) where no errors were present. This chapter presents the initial TR-Protocol CPN (with several modifications). Chapter 8 presents the analysis of, and changes to, this initial CPN that led to the Revised TR-Protocol CPN. Chapter 9 presents a detailed analysis of the Revised TR-Protocol CPN.

The modifications included in the TR-Protocol CPN presented in this Chapter are a result of errors found in the protocol from analysing previous CPN models. However, we present them up-front in this chapter (as opposed to presenting the initial TR-Protocol CPN and then the analysis results) to save space and because the errors and solutions, once identified using the CPN model, are easily described by examining the state tables. Other errors, with more involved solutions, are presented in Chapter 8 where the analysis results are used in the explanations.

The TR-Protocol CPN presented in this chapter is an updated version of that given in [56]. Improvements have been made to produce a simpler and clearer model that better reflects the WTP Specification. The main change is the current TR-Protocol CPN models just one transaction, whereas the TR-Protocol CPN in [56] modelled multiple transactions. We have generalized the TID verification procedure, allowing the removal of the specific caching mechanism. The counter `AEC` is also modelled non-deterministically,

significantly reducing the number of configurations to be analysed. Chapter 8 discusses how the results obtained in [56], which were submitted to the WAP Forum [55], differ from the analysis of the current TR-Protocol CPN.

This chapter begins by giving an overview of the TR-Protocol structure in Section 7.1. Section 7.2 lists the scope and assumptions of the TR-Protocol CPN. Also included are the errors in the TR-Protocol that this CPN has fixed. Section 7.3 describes the hierarchical structure of the TR-Protocol CPN. The four layers of the TR-Protocol CPN are presented from highest to lowest in Sections 7.4 to 7.7.

The model of the Transaction Protocol, especially its scope and the assumptions, has benefited from discussions with Professor Jonathan Billington. The approach to modelling the aborted transactions for which the Responder is not aware the transaction has started (Section 7.6.3) is partly due to Professor Billington.

7.1 Structure of the Transaction Protocol

The TR-Service in Chapter 6 is defined by the primitives and their parameters, and the set of possible primitive sequences between the TR-Users and the TR-Service-Provider. Figure 6.1 shows a simple model of the TR-Service. The TR-Protocol is a refinement of the TR-Service. It defines the procedures of the two protocol entities (TR-PEs), Initiator (TR-Init-PE) and Responder (TR-Resp-PE), for providing the TR-Service to the TR-Users. Figure 7.1 shows the model of the TR-Protocol and its relationship with the TR-Users and the Transport Service Provider (T-Service-Provider).

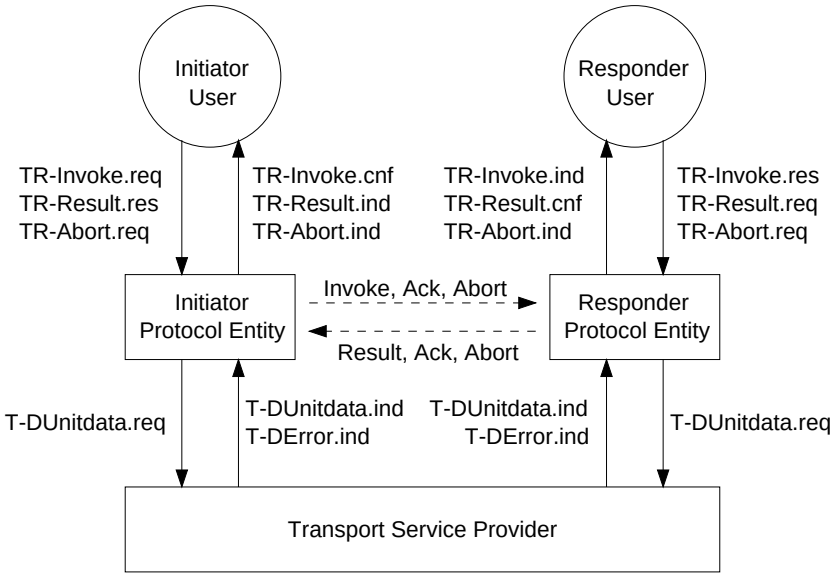

Figure 7.1: Block diagram of the TR-Protocol

The TR-Service primitives are submitted by, or delivered to, the TR-Users. For clarity, the primitive parameters are omitted from Figure 7.1. The TR-Init-PE and TR-Resp-

PE use the primitives of the Transport Service (e.g. T-DUnitData.req) to send and receive PDUs (Invoke, Ack, Result and Abort). In the WAP architecture, the Transport Service is provided by WDP [179]. The Transport Service primitives are T-DUnitData (with a request and indication type) and T-DError (with an indication type). T-DUnitData is used to deliver data (in the TR-Protocols case, the four PDUs) between the users. The T-DError primitive is only used when the T-Service-Provider initiates an abort. The following section scopes the TR-Protocol features and components of Figure 7.1 to be modelled and lists the assumptions made in the modelling process.

7.2 Scope and Assumptions of the TR-Protocol CPN

The aim of creating the TR-Protocol CPN is to verify the design of the TR-Protocol. For the verification to be successful, we must ensure the TR-Protocol CPN is a valid representation of the TR-Protocol and that our analysis tools and techniques can cope with the verification. Another minor requirement is that the TR-Protocol CPN should be easy to maintain to reflect changes in the TR-Protocol. With these objectives in mind, we limit parts of the TR-Protocol that are modelled by the CPN. Note that we are not interested in the performance aspects of the TR-Protocol, only its functional behaviour.

Section 7.2.1 defines the scope of our TR-Protocol CPN. This comprises a set of limitations that we impose so that our objectives can be met. Either the limitations may have an impact on the analysis results (in which case they are referred to as *restrictions*), or the results are independent of them (called *simplifications*).

Section 7.2.2 gives the assumptions made when modelling the TR-Protocol. The assumptions are necessary when the description of the TR-Protocol in the WTP Specification is ambiguous or incomplete. Therefore, the assumptions are also related to errors found in the WTP Specification.

7.2.1 Scope of the TR-Protocol CPN

Configuration of TR-Users

The first restriction we introduce limits the investigation to Transaction Class 2 of the TR-Protocol in the WTP Specification [183]:

Restriction 7.1 (Transaction Class 2). *Only Class 2 Transactions and version 1.2.1 (as described in [183]) are modelled.* □

A consequence of Restriction 7.1 is that the `TCL` and `Version` header fields (see Section 5.3.2) are not modelled.

Transactions take place between a TR-Init-User and TR-Resp-User. Each transaction is identified by the address of the device TR-Users are operating on, and the port of the TR-Users' application (see Section 5.2.2). In addition, each TR-Init-PE uses a TID to identify the transactions outstanding. If the same port is being used by a TR-Init-PE and TR-Resp-PE, then the destination of PDUs is identified by the highest order bit of the TID (see Section 5.3.1). There is, however, no interaction between the TR-Init-PE and TR-Resp-PE using the same port that affects their functional behaviour. Therefore, we can simplify the TR-Protocol CPN in two ways without impacting the analysis results:

Simplification 7.1 (Configuration of the TR-Users). *Transactions occur between one TR-Init-User and one TR-Resp-User.* □

Simplification 7.2 (Direction of PDUs). *The highest order bit in the TID field of PDU headers is not modelled because there is no interaction between a TR-Init-PE and TR-Resp-PE using the same port.* □

From Figure 7.1 the TR-Service primitives submitted by and delivered to the TR-Users must be modelled. However, most of the parameters of the primitives may be omitted from the TR-Protocol CPN:

Simplification 7.3 (TR-Service Primitive Parameters). *For the three TR-Service primitives (TR-Invoke, TR-Result and TR-Abort), only the Ack-Type parameter in the TR-Invoke.req primitive is modelled.* □

Simplification 7.1 means the Source Address, Source Port, Destination Address and Destination Port do not change and, therefore, are unnecessary in the TR-Protocol CPN. Similarly for the Class Type, which is always 2. The operation of the TR-Protocol is independent of the User Data, Handle and Abort Code, therefore they do not need to be modelled. As we will see in Simplification 7.5, TPIs, which are the sole purpose of Exit Info, are not modelled.

Multiple Transactions

For the TR-Protocol to operate correctly, multiple transactions between a TR-Init-User and TR-Resp-User must not interact. Although differentiated by the TID, when the TID numbers wrap, there may be a possibility confusion occurs as to which transactions the PDUs belong. Figure 7.2 illustrates how the confusion may occur.

The TR-Init-User initiates a transaction using the TID value of 0. The TID value is incremented by one for each new transaction initiated. The effective range of the TIDs is 0 to 32767 (see the Transaction Identifier protocol feature in Section 5.3.1). Figure 7.2 shows that an Invoke PDU is sent initiating each transaction. For clarity, the other PDUs

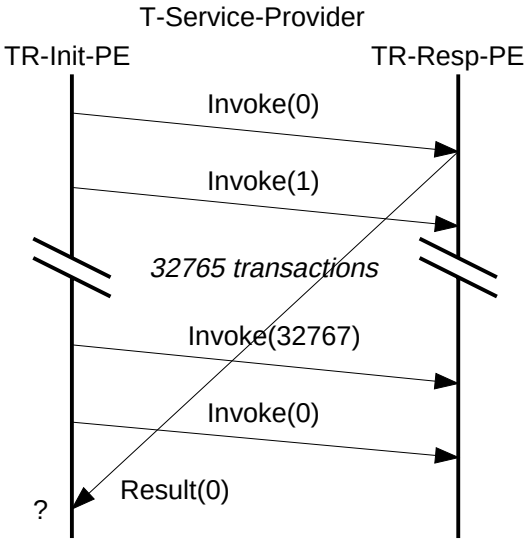

Figure 7.2: Example of PDUs with different TID incarnations overlapping

within the transaction are hidden. After the TID value of 32767 is used, the TR-Init-User re-uses 0, i.e. there is a new incarnation of TID 0. Confusion occurs if PDUs are received by a TR-PE when two (or more) incarnations of a TID value are in use. In Figure 7.2 the TR-Init-PE has no way to determine which transaction the Result PDU with TID 0 belongs to.

The possibility of confusion of PDUs is a problem encountered by many communication protocols that use sequence numbers. The solution used by the TR-Protocol (which is similar to that for other protocols, e.g. T/TCP [20]) is to make the following assumptions:

1. Each PDU has a maximum lifetime for being in the network (Maximum Packet Lifetime (MPL)). When this lifetime expires, it is assumed the PDU or any of its duplicates will have been removed from the network. The value of MPL depends on the bearer service.

2. The rate the TR-Init-PE can initiate transactions is limited to 2^{14}, or 16384, transactions in a time of 2MPL (see Section 5.3.1).

3. The transaction time, (i.e. the time from when the first PDU is sent, until the last PDU is received) is insignificant compared to MPL.

If these assumptions hold, then the TR-Protocol guarantees confusion of PDUs will not occur. Suppose transaction (with TID) 0 is initiated at time 0, followed by $2^{14} - 1$ more transactions. As 2^{14} transactions have been initiated, according to the second assumption, the next transaction cannot be initiated until time 2MPL. After this time, all PDUs associated with transaction 0 will have been removed from the network. Therefore,

there will be no possibility of confusing PDUs of the next incarnation of transaction 0, with the preceding transaction.

Using the assumptions made in the WTP Specification, we can simplify our model:

Simplification 7.4 (Single Transaction). *Only a single transaction is modelled since there is no interaction between transactions in the TR-Protocol.* □

Simplification 7.4 is made because our objective is to verify the functional behaviour of the TR-Protocol. If performance aspects of the TR-Protocol were the main objective, then investigating the assumptions made about the MPL and transfer rate would be important. These aspects have, to some extent, been investigated in [19, 20, 149]. For example, if MPL was assumed to be 2000 seconds (as in [19]), the maximum rate at which transactions could be initiated by the TR-Init-PE would be approximately 4 per second. Investigating the transaction rate required by intended applications of the TR-Protocol, and the appropriate MPL for different bearer services is discussed as an area for future work in Chapter 10.

Protocol Features

We use an incremental approach to verifying the TR-Protocol. Therefore, only those features that are representative of the core behaviour of the TR-Protocol are modelled and analysed. As confidence is gained in the these features, other features can be included in the CPN. Chapter 10 elaborates on the features that can be added as a part of future work. Table 7.1 lists which protocol features have been modelled and which have been omitted.

Feature	*Modelled*	*Omitted*
Message transfer	X	
Re-transmission until acknowledgment	X	
User acknowledgment	X	
Transaction abort	X	
Transaction identifier verification	X	
Error handling	X	
Transport information items		X
Transmission of parameters		X
Information in last acknowledgment		X
Asynchronous transactions		X
Transaction identifier		X
Version handling		X
Segmentation and re-assembly		X

Table 7.1: Protocol features modelled or omitted from the TR-Protocol CPN

Simplification 7.5 (Omission of Transport Information Items). *Transport Information Items are not modelled in the TR-Protocol CPN. This includes the Transmission of Parameters and Information in Last Acknowledgment features, which use TPIs.* □

TPIs are used to transfer optional information (typically to do with the protocol's performance) to TR-PEs or TR-Users within existing PDUs. As these existing PDUs are modelled, and the objective of the analysis is investigating the functional behaviour (not the performance), TPIs do not have any impact on the core behaviour of the TR-Protocol. As TPIs are not modelled, the `CON` header field in the PDUs (see Section 5.3.2) is not modelled.

The Asynchronous Transactions and Transaction Identifier protocol features are not modelled because only one transaction is considered (Simplification 7.4).

The Version Handling protocol feature is not modelled because we assume the correct versions are being used by both TR-PEs (see Restriction 7.1).

Restriction 7.2 (Omission of Segmentation and Re-Assembly). *The Segmentation and Re-assembly (SAR) protocol feature is not modelled in the TR-Protocol CPN.* □

SAR is an optional feature and, therefore, is not a core part of the TR-Protocol. As a result of Restriction 7.2, the `GTR`, `TTR` and `PSN` header fields (see Section 5.3.2) are not modelled.

Transport Service Provider

The Transport Service is used by the TR-PEs to communicate with each other. (The optional Security layer only adds security to the Transport Service which is transparent to the TR-Protocol and, therefore, can be ignored.) The characteristics of the T-Service-Provider must be known so that an abstract model of it can be created. The WTP Specification suggests PDUs can be lost (e.g. the Re-transmission Until Acknowledgment feature in Chapter 5), re-ordered (e.g. Section 8.8.2.4 (Reception of out-of-order Invoke messages) of the WTP Specification [183]) or duplicated (e.g. the Transaction Identifier feature in Chapter 5). These assumptions about the T-Service-Provider are justified by the fact that WDP [179], a connection-less protocol, is used in the Transport layer. WDP (which specifies UDP to be used over bearer services supporting IP) only provides addressing via ports. Error-detection and SAR are optional. There is no guarantee of eventual, in-order or unique delivery of PDUs.

Restriction 7.3 states the characteristics of the T-Service-Provider that are assumed in the TR-Protocol CPN:

Restriction 7.3 (T-Service-Provider Characteristics). *The Transport Service Provider provides reliable delivery of datagrams (i.e. no corruption, losses or duplicates), but with no guarantee of ordering. The capacity of the Transport Service Provider is infinite.* □

As discussed in Chapter 3, an incremental approach to the verification process is being used. Restriction 7.3 places the current TR-Protocol CPN in an intermediate stage of

this process. Modelling the TR-Protocol without the possibility of any errors is the first stage, but the introduction of overtaking is a more interesting scenario. However, loss and duplication of PDUs is not modelled because it introduces significant complexity into the state space, potentially limiting the different configurations to be analysed. (Section 7.5 discusses how errors increase the state space size). We are, therefore, limited by the approach we have chosen.

Note that although errors in the channel are not modelled, the Error Handling feature is modelled, as that comprises provider initiated aborts, and the abort procedures included in the state tables.

TID Verification

The TID verification protocol feature (discussed in Section 5.3.1) allows the TR-Resp-PE to verify if the Invoke PDU it received is for an outstanding transaction at the TR-Init-PE. We can generalize the procedure for determining whether TID verification is necessary by stating that an Invoke PDU with TID x can either be expected (no TID verification) or unexpected (initiate TID verification). Therefore, a specific caching mechanism is not required to be modelled. Similarly, the `TIDnew` field in the Invoke PDU header can be ignored as its use is to invalidate the cache at the TR-Resp-PE when the TID space wraps.

Simplification 7.6 (TID Verification). *The decision on receiving a new Invoke PDU is an arbitrary choice between initiating a TID verification or not. A specific caching mechanism and the `TIDnew` field in the Invoke PDU are not modelled.* □

Protocol Data Units

Section 5.3.2 defined the header fields for the Invoke, Ack, Result and Abort PDUs. As well as these mentioned in the preceding simplifications and restrictions, the TR-Protocol CPN does not model the `TID`, `RES`, `AbortType` and `AbortReason` fields because they have no impact on the behaviour of a single transaction.

Buffers of the TR-PEs

Each TR-PE sends PDUs to an output buffer which is delivered, via the T-Service-Provider, to the input buffer of the peer TR-PE. The buffers are modelled as part of the T-Service-Provider. They can have arbitrary capacity.

7.2.2 Modelling Assumptions

The TR-Protocol CPN is based mainly on the state tables in the WTP Specification. These are described in Section 5.3.3 and repeated in full in Appendix B. There are several assumptions we must make about how to read the state tables and also how to treat several ambiguities in the WTP Specification.

Assumption 7.1 (Atomic Events). *Each state table entry describes an atomic event of a TR-PE.* □

Assumption 7.1 allows each entry in the state tables to be modelled as a transition in the TR-Protocol CPN. There is one exception (i.e. a state table entry that is *not* an atomic event) which is described in Section 7.7.

Assumption 7.2 (State Table Conditions). *Where state table entry conditions do not specify values for variables, parameters or header fields, the variables, parameters and header fields can take any value. Where state table entries in one state table have identical events, the conditions must be mutually exclusive.* □

The first sentence of Assumption 7.2 is an obvious way to read the state tables. The second sentence of Assumption 7.2 is necessary because several state tables in the WTP Specification specify entries that, if the conditions were satisfied, would be inclusive of other entries. For example, Entry 1 in the TR-Init-PE NULL state table (Table B.1) requires the transaction class to be 1 or 2. Entry 2 in this table requires the transaction class to be 1 or 2, and `UserAck` to be On. Both of these entries have the same event (TR-Invoke.req). If Assumption 7.2 was not made, then Entry 1 could also include Entry 2. This is obviously not the intention of the state tables because, from Entry 1, the variable `Uack` could be set to false if `UserAck` was On. Therefore, we assume that the condition of Entry 1 includes the requirement that `UserAck` is Off, making the entries with the same event mutually exclusive. Thus, in this case, the value of the variable `UserAck` is not arbitrary.

Assumption 7.3 (Ordering of Actions). *The actions in state table entries are ordered, from top to bottom.* □

Being atomic events, we could assume the actions in state table entries could occur in any order. However, we make Assumption 7.3 because some actions both start and stop a timer, e.g. Event 10 in the TR-Init-PE RESULT WAIT state table (Table B.2). Executing the actions in any order would result in different behaviors of the TR-Protocol. Assumption 7.3 also ensures the TR-Invoke.cnf primitive is delivered to the TR-Init-User before the TR-Result.ind primitive for Event 10 in the TR-Init-PE RESULT WAIT state table.

Assumption 7.4 (PDUs Without State Table Entries). *If a state table does not have an event for receiving a PDU, then the PDU is ignored upon its receipt.* □

There are state tables in the WTP Specification that do not have events for the receipt of all possible PDUs. (Note that an ErrorPDU is defined as an "Illegal PDU type or erroneous header structure" (page 50, [183]), and hence the RcvErrorPDU entry does not cover the receipt of legal PDUs that are not specified in the state tables.) Table 7.2 shows for all states of both TR-PEs, whether the state tables specify if a PDU can be received (Y) or not (N). The entries marked *n.a.* are not applicable because the TR-Init-PE cannot receive Invoke PDUs and the TR-Resp-PE cannot receive Result PDUs.

TR-PE and State Name	*Invoke*	*Result*	*Abort*	*Ack*	*Ack (Tve/Tok)*
I NULL	n.a.	Y	Y	Y	Y
I RESULT WAIT	n.a.	Y	Y	Y	Y
I RESULT RESP WAIT	n.a.	Y	Y	N	N
I WAIT TIMEOUT	n.a.	Y	Y	N	N
R LISTEN	Y	n.a.	Y	Y	Y
R TIDOK WAIT	Y	n.a.	Y	N	Y
R INVOKE RESP WAIT	Y	n.a.	Y	N	N
R RESULT WAIT	Y	n.a.	Y	N	N
R RESULT RESP WAIT	N	n.a.	Y	Y	Y

Table 7.2: State tables that do not specify the receipt of all PDUs

We have assumed that for all those entries marked with an N in Table 7.2, if a PDU *is* received, then it is ignored (discarded). This is a sensible interpretation because the nature of the TR-Protocol, in that PDUs are re-transmitted and overtaking can occur, means that it is likely that PDUs will be received in states when they are not required. For example, suppose the TR-Resp-PE sends an Ack PDU (Entry 9, Table B.7) and a Result PDU (Entry 1, Table B.8) after receiving the Invoke PDU. As overtaking of PDUs is allowed in the communication channel, the Result PDU may be received by the TR-Init-PE (Entry 10, Table B.2) before the Ack PDU. The TR-Init-PE enters the RESULT RESP WAIT state (Table B.3), which does not specify the event of receiving an Ack PDU. In this case, it would be sensible to ignore the Ack PDU. Assumption 7.4 signifies that all PDUs may be received in any state (although no action is taken on the receipt of some PDUs).

Assumption 7.5 (TveTok Flag). *The conditions* TIDve *and* TIDok *in a RcvAck state table entry indicate the received Ack PDU must have the* `TveTok` *flag set to 1. The absence of such a condition in a RcvAck state table entry indicates the received Ack PDU must have the* `TveTok` *flag set to 0.* □

Assumption 7.2 stated the absence of header fields in the conditions indicates they can take any value. Assumption 7.5 is an exception to this. If the `TveTok` flag is absent, then we assume `TveTok` must be set to 0 (i.e. an ordinary Ack PDU is received). This

is necessary because while in the RESULT RESP WAIT state (see Table B.9), if the TR-Resp-PE receives a re-transmitted and delayed Ack(Tok) PDU (Entry 3), it may be confused as an acknowledgment of the Result PDU. Using Assumption 7.5 and 7.4, the receipt of an Ack(Tok) PDU in this state will be ignored. Table 7.3 shows Entry 3 of the TR-Resp-PE RESULT RESP WAIT state table updated to include the appropriate condition. The addition to the state table is shown in italics.

	Event	Condition	Action	Next State
3	RcvAck	*TveTok=0*	Generate TR-Result.cnf	LISTEN

Table 7.3: Entry 3 of the TR-Resp-PE RESULT RESP WAIT state table (Table B.9) modified to include TveTok set to 0

Assumption 7.5 is included to overcome the error of misinterpreting an Ack(Tok) PDU for an Ack PDU. This was discovered while analysing a previous TR-Protocol CPN. Since its discovery, the WAP Forum have published a Specification Information Note [177] for the WTP Specification [183] that also identifies the problem.

Assumption 7.6 (Acknowledgment Sent Variable). *A variable called* `AckSent` *is used by both TR-PEs. There is one variable per transaction. The type of* `AckSent` *is BOOL.* `AckSent` *is True at the TR-Init-PE if the TR-Init-PE has sent an Ack(Tok) PDU.* `AckSent` *is True at the TR-Resp-PE if the TR-Resp-PE has sent an Ack PDU.* □

Assumption 7.6 is necessary because the state tables in the WTP Specification include conditions that check if an Ack PDU has already been sent. For the TR-Init-PE, Entry 7 of the RESULT WAIT state table (Table B.2) has a condition "Ack(TIDok) already sent". Similarly, Entry 4 of the TR-Resp-PE RESULT WAIT state table (Table B.8) has a condition "Ack PDU already sent". Therefore, we introduce the variable `AckSent` at each TR-PE that is set at the times specified in Assumption 7.6 so the conditions can be applied.

Assumption 7.7 (Limiting the Counters). *Any counters used by the TR-PEs (*`AEC` *and* `RCR`*) cannot have a value higher than its specified maximum (*`AEC_MAX` *and* `RCR_MAX`*). Therefore, whenever an action in a state table entry increments a counter (by 1), there must be a condition in the state table entry that requires the counter to be less than its specified maximum.* □

Assumption 7.7 is required to ensure the `RCR` counter used by the TR-Init-PE is not increased above `RCR_MAX` when the TR-Init-PE sends the Ack(Tok) PDU. Entry 3 of the TR-Init-PE RESULT WAIT state table (Table B.2) increments `RCR` without checking if it is less than the maximum. This behaviour results in ambiguity in the WTP Specification. It may, for example, create confusion in implementations. The counter may be implemented as an array populated by time-out periods [183]. The index into the array

is the counter. If the array is defined to have `RCR_MAX` elements, then the index (`RCR`) may be larger than the number of elements.

Table 7.4 shows Entry 3 of the TR-Init-PE RESULT WAIT state table updated to limit `RCR`. A new entry is also created so that when an Ack(Tve) PDU is received while `RCR` is equal to `RCR_MAX`, it is ignored.

	Event	Condition	Action	Next State
3	RcvAck	TIDve Class == 2 | 1 *RCR<RCR_MAX*	Send Ack(TIDok) Increment RCR Start timer, R[RCR]	RESULT WAIT
3a	*RcvAck*	*TIDve* *Class == 2 | 1* *RCR==RCR_MAX*	*Ignore*	*RESULT WAIT*

Table 7.4: Entries 3 and 3a of the TR-Init-PE RESULT WAIT state table (Table B.2) modified to limit RCR

Assumption 7.8 (Notifying TR-Users of Aborts). *When a transaction is aborted by a TR-PE as a result of a time-out, a TR-Abort.ind primitive must be delivered to the TR-User.* □

Assumption 7.8 ensures the TR-User is notified that the transaction has been aborted. This assumption is necessary because, from the state tables in the WTP Specification, upon expiration of the acknowledgment timer, the TR-Init-PE and TR-Resp-PE abort the transaction (i.e. return to their respective initial states) without delivering a TR-Abort.ind primitive to the TR-User. The two erroneous state table entries are Entry 8 of the TR-Init-PE RESULT RESP WAIT state table (Table B.3) and Entry 8 of the TR-Resp-PE INVOKE RESP WAIT state table (Table B.7). Figure 7.3 shows a possible scenario where the TR-Init-PE aborts and the TR-Resp-PE receives the Abort PDU and notifies its user with a TR-Abort.ind primitive. The TR-Init-User is not aware the transaction has been aborted. The TSD is an extension of the service TSDs used in Chapter 6. Service primitives are shown in the same way as the service TSDs, but the vertical lines now represent the TR-PEs. The area between the two vertical lines (TR-PEs) represents the T-Service-Provider. The delivery of PDUs between the TR-PEs, with a delay, are shown there. The expiration of a timer is shown as a black square at the TR-PE (on the vertical line). Alongside the service primitives, an abbreviation of the state of the TR-PE is shown (e.g. NU = NULL, RW = RESULT WAIT). Vertical, double headed arrows show the timer interval (A, R or W) that was in use when the time-out occurs.

Tables 7.5 and 7.6 show the updated entries for the TR-Init-PE RESULT RESP WAIT (Entry 8) and TR-Resp-PE INVOKE RESP WAIT (Entry 8) state tables, respectively, to ensure the TR-Abort.ind primitive is delivered to the TR-User.

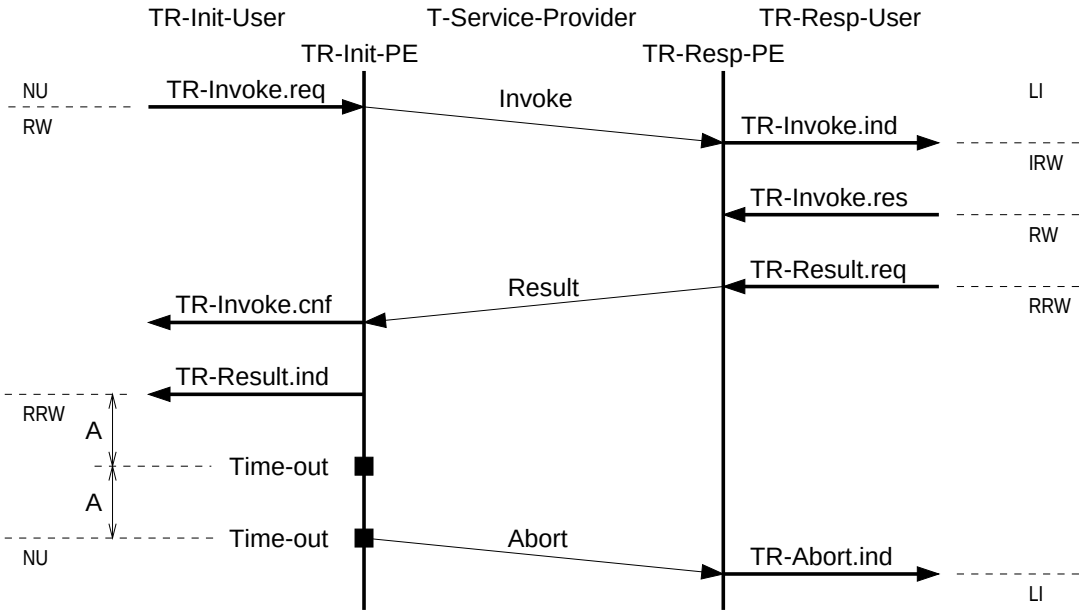

Figure 7.3: Error in the TR-Protocol where both TR-PEs have aborted the transaction, but the TR-Init-User is not notified of the abort

	Event	Condition	Action	Next State
8	TimerTO_A	AEC == AEC_MAX	Abort transaction *Generate TR-Abort.ind* Send Abort PDU (NORESPONSE)	NULL

Table 7.5: Entry 8 of the TR-Init-PE RESULT RESP WAIT state table (Table B.3) modified to deliver the TR-Abort.ind primitive when aborting due to a time-out

	Event	Condition	Action	Next State
8	TimerTO_A	AEC == AEC_MAX	Abort transaction *Generate TR-Abort.ind* Send Abort PDU (NORESPONSE) Start timer, W	LISTEN

Table 7.6: Entry 8 of the TR-Resp-PE INVOKE RESP WAIT state table (Table B.7) modified to deliver the TR-Abort.ind primitive when aborting due to a time-out

Assumption 7.9 (Restricting TR-Result.req when UserAck On). *When UserAck is On, the TR-Resp-User cannot submit the TR-Result.req primitive before it has submitted the TR-Invoke.res primitive.* □

Assumption 7.9 is necessary because after the TR-Resp-PE delivers the TR-Invoke.ind primitive to the TR-Resp-User, it enters the INVOKE RESP WAIT state which allows the submission of the TR-Result.req primitive (Entry 2 of Table B.7). When UserAck is On, the TR-Result.req primitive should only be possible while the TR-Resp-PE is in the RESULT WAIT state (Table B.8).

Table 7.7 shows Entry 2 of the TR-Resp-PE INVOKE RESP WAIT state table updated by placing a condition that restricts the submission of TR-Result.req to the case when UserAck is Off.

	Event	**Condition**	**Action**	**Next State**
2	TR-Result.req	*Uack==False*	Reset RCR Start timer, R[RCR] Send Result PDU	RESULT RESP WAIT

Table 7.7: Entry 2 of the TR-Resp-PE INVOKE RESP WAIT state table (Table B.7) modified to prevent the submission of TR-Result.req when UserAck is On

Assumption 7.10 (Restricting TR-Aborts after TR-Result.res). *The TR-Init-User cannot submit a TR-Abort.req primitive, nor be delivered a TR-Abort.ind primitive after the TR-Init-PE has sent an Ack PDU acknowledging the Result PDU.* □

Assumption 7.10 is included so the TR-Protocol reflects the behaviour of the TR-Service described in Assumption 6.7.

After the TR-Init-User has submitted the TR-Result.res primitive, the TR-Init-PE enters the WAIT TIMEOUT state (Entries 1 and 2 of the TR-Init-PE RESULT RESP WAIT state table (Table B.3)). The TR-Abort primitives should not be possible in this scenario. However, the WAIT TIMEOUT state may also be entered if only the TR-Init-PE acknowledges the Result PDU (i.e. the TR-Init-User does not submit a TR-Result.res primitive) as given by Entry 9 of the TR-Init-PE RESULT RESP WAIT state table. Although the TR-Service allows the TR-Abort primitives to occur in this scenario, we restrict the TR-Protocol so that they cannot. We assume the purpose of the WAIT TIMEOUT state is to only re-transmit the Ack PDU, if necessary. Therefore, our suggested change to the TR-Protocol is to disallow TR-Abort primitives in the WAIT TIMEOUT state.

The event corresponding to the TR-Init-User submitting the TR-Abort.req primitive remains, but it now corresponds to a local user event (Entry 7 of Table B.4). We call this event *Clear*. The TR-Init-User notifies the TR-Init-PE to clear the transaction state information. This is not an end-to-end event.

The receipt of an Abort PDU while in the WAIT TIMEOUT state (Entry 4) is possible, but it does not initiate the delivery of a TR-Abort.ind primitive to the TR-Init-User. Only the transaction state information is cleared upon receipt of the Abort PDU.

Table 7.8 shows Entries 4 and 7 of the TR-Init-PE WAIT TIMEOUT state table updated to restrict the TR-Abort primitives in this state. Text to be deleted from state tables is indicated with a line through it.

	Event	**Condition**	**Action**	**Next State**
4	RcvAbort		Abort transaction ~~Generate TR-Abort.ind~~	NULL
7	~~TR-Abort.req~~ *Clear*		Abort transaction ~~Send Abort PDU (USER)~~	NULL

Table 7.8: Entries 4 and 7 of the TR-Init-PE WAIT TIMEOUT state table (Table B.4) modified to remove TR-Abort primitives

A discussion of Assumptions 7.7, 7.8 and 7.9 have been submitted to the WAP Forum [55]. A response indicated that these suggested changes would be acted upon, which is evident in Version 2.0 of WTP [187].

7.3 Structure of the TR-Protocol CPN

The hierarchy page for the TR-Protocol CPN is shown in Figure 7.4. The TR-Protocol CPN consists of 15 CPN pages, one page for declarations and several pages for analysis code and results. The CPN pages are divided into four hierarchical levels. The first level presents an abstract view of the TR-Protocol on one page (called TR_Protocol). The second level shows the two TR-PEs (pages TR_Init_PE and TR_Resp_PE), which are further decomposed in the third level where each state table of a TR-PE is modelled on a separate page. The fourth level comprises just one page, called I_RW_RcvResult_Cnf, which models a special case of two primitives being delivered to the TR-Init-User as part of one action. The pages for each of the levels will be described in detail in Sections 7.4 to 7.7.

The Declarations page defines all types, constants, variables and functions used in the TR-Protocol CPN. The declarations will be introduced incrementally as we progress through the descriptions of the CPN pages. The complete set of declarations are shown in Appendix D (Listing D.1). The code and results on the analysis pages will be discussed in Chapter 8.

The hierarchical structure of the TR-Protocol CPN aids in the development and maintenance of the model by providing a logical structure that can be validated against the WTP Specification [183]. Section 7.6 describes how the validation is achieved. Another

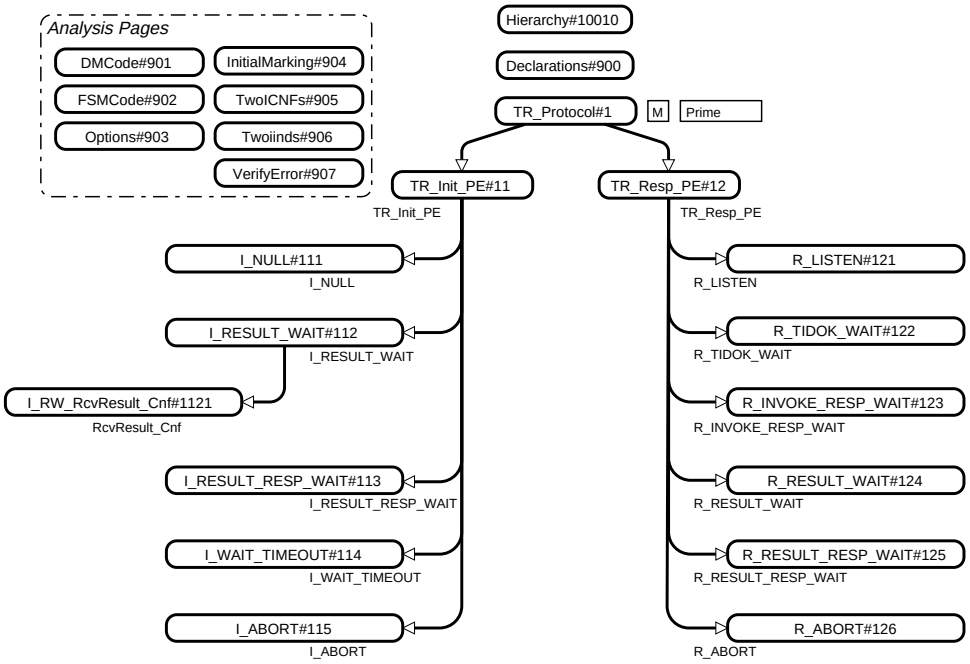

Figure 7.4: Hierarchy page for the TR-Protocol CPN

advantage of the structure is the choice of different types of visualization when examining the dynamic behaviour of the TR-Protocol CPN. For example, from the second level pages, the current state of a TR-PE during an interactive simulation can be clearly seen, without any concern for the details of what is happening in that state. However, as all the state tables are significantly different, we cannot re-use any pages (i.e. using page instances in Design/CPN) to improve maintainability of the model.

7.4 Overview Page

The TR_Protocol page, shown in Figure 7.5, presents an abstract view of the TR-Protocol. This view is similar to that shown in Figure 7.1 where the two TR-PEs, modelled by transitions TR_Init_PE and TR_Resp_PE, communicate with each other by sending and receiving PDUs. The PDUs are delivered via the T-Service-Provider (which includes the input and output buffers). The places InitToResp and RespToInit, which are collectively referred to as *communication places*, model the T-Service-Provider. (The dashed arc from InitToResp to TR_Init_PE models a special case where the transaction is aborted before a PDU is delivered to the TR-Resp-PE. This, and the reason for removing a token from InitToResp is discussed in Section 7.6.3.)

The communication places are typed by the colour set PDU (shown in italics above the place). The details of the PDU colour set will be introduced in Section 7.5, but for now it is sufficient to know that it defines the different types of PDUs (e.g. Invoke, Ack, ...) and any relevant header fields. Two places are used to clearly visualize the direction

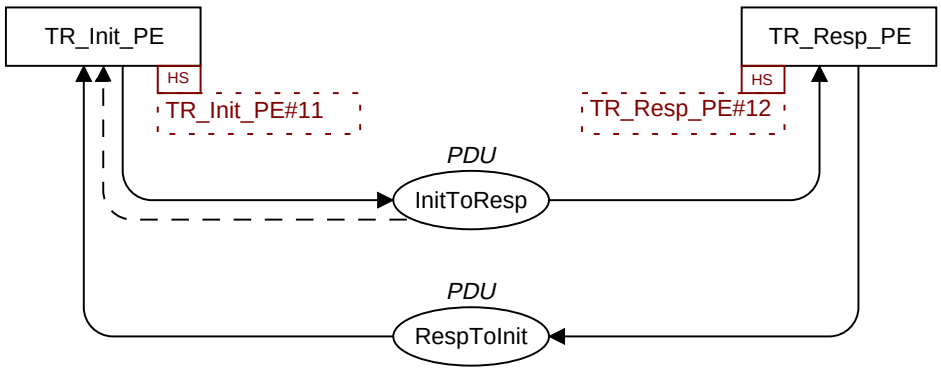

Figure 7.5: TR_Protocol page in the TR-Protocol CPN

of flow of PDUs between the TR-PEs.

Each place realizes the infinite capacity, and possibility of re-ordering of PDUs in the T-Service-Provider (Restriction 7.3). Because of the re-ordering, the communication places conveniently model the buffers, independently of their queuing discipline, as the TR-PEs simply view the communication channel as allowing re-ordering. The mechanism that queues PDUs before sending them (see Section 5.3.3) is, therefore, also modelled by the communication places.

The TR_Protocol page does not show the TR-Users (cf. Figure 7.1). This is because we chose to model the TR-Users implicitly via the occurrence of transitions representing the submission and delivery of primitives. These transitions are in the state table pages described in Section 7.6.

Each TR-PE transition is decomposed into a sub-page that provides further details on the TR-PEs operation. The boxed HS below the transitions on the TR_Protocol page designates them as hierarchical substitution transitions. The name of the corresponding sub-page is shown in the dashed box. Arc inscriptions are not necessary as their details become apparent on the sub-pages.

7.5 Protocol Entity Pages

The second level pages in the TR-Protocol CPN show the different states of the TR-PEs. The TR_Init_PE page is shown in Figure 7.6 and the TR_Resp_PE page in Figure 7.7.

The TR-PE pages each introduce a new place. On the TR_Init_PE page there is the place named Initiator (typed by the colour set InitState), and for the TR_Resp_PE page a place named Responder (typed as RespState). Together, these two places are referred to as the *state places*. They model the current state of their respective TR-PEs.

The TR_Init_PE page has five (substitution) transitions. Each of the top four tran-

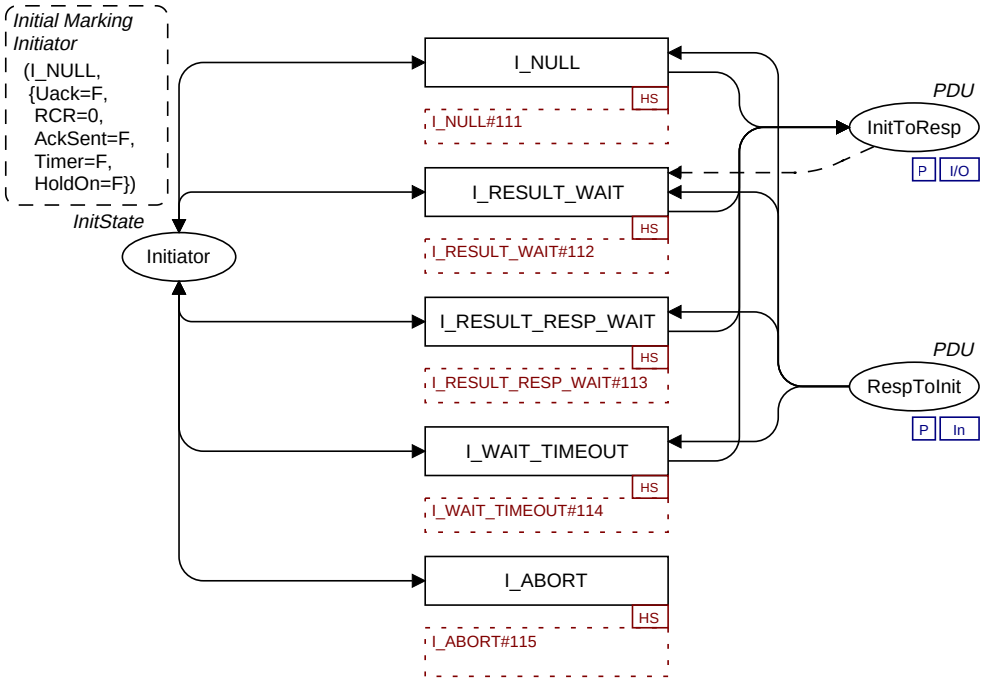

Figure 7.6: TR_Init_PE page in the TR-Protocol CPN

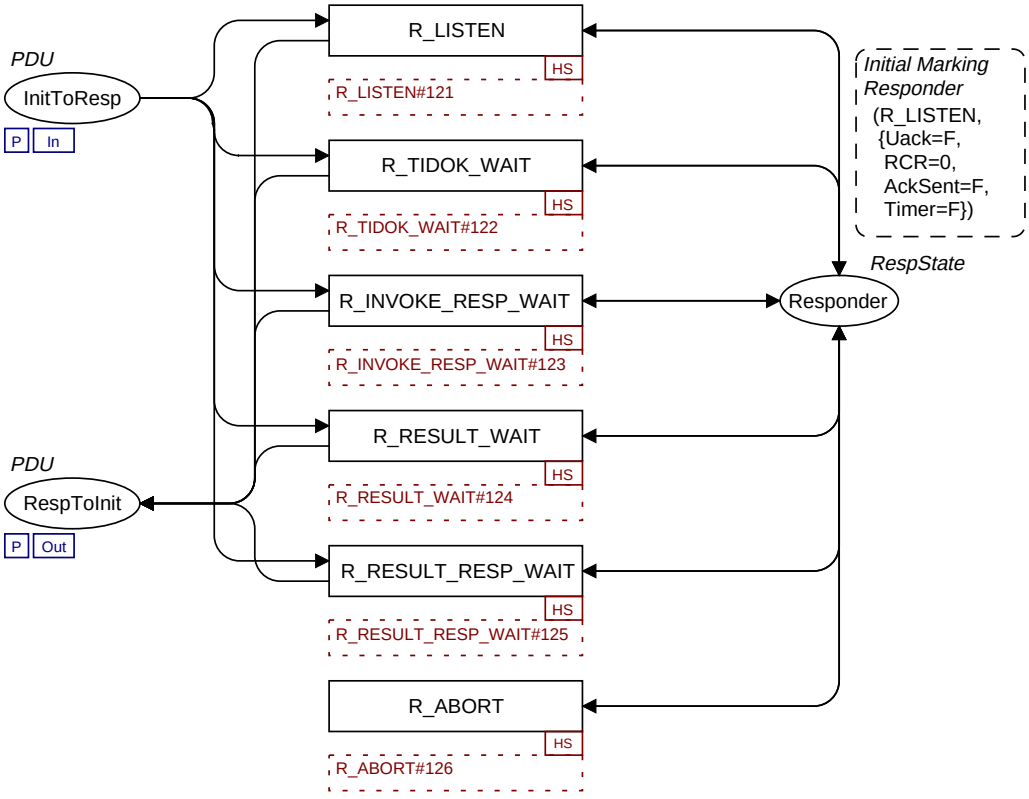

Figure 7.7: TR_Resp_PE page in the TR-Protocol CPN

sitions model a state of the TR-Init-PE. The bottom transition, I_ABORT models a T-Service-Provider initiated abort. The letter I prefixes the state names to distinguish from the states of the TR-Resp-PE. The meaning of the states is described in Table 5.8. Each transition is decomposed into a sub-page that models the state table (or actions, for I_ABORT) for the corresponding state of the TR-Init-PE. Section 7.6 describes the sub-pages.

To gain a better understanding of how the TR-PE pages are structured, we introduce the PDU, InitState and RespState colour sets. Listing 7.1 defines two basic colour sets from which the others are derived.

Listing 7.1: Basic colour sets used in the TR-Protocol declarations

```
1 color Flag = bool with (F,T);
2 color RCR_c = int;
```

The colour set Flag is a boolean type that is used to model boolean variables stored by the TR-PEs or 1-bit header fields in PDUs. By default, the colour set includes the colours false and true. We can also use the colours F and T (for brevity). The colour set RCR_c is defined as the integers. It models the values of the TR-PE counter `RCR`. The suffix _c (for colour) is used on this and other colour sets to distinguish them from variables when necessary.

Listing 7.2 defines the colour sets and variables that represent PDUs in the TR-Protocol CPN.

Listing 7.2: PDU colour sets used in the TR-Protocol declarations

```
1  color InvokePDU_c = record
2     RID:Flag      *          (* Retransmission Indicator *)
3     UP:Flag;                 (* User or PE acknowledgment *)
4  color ResultPDU_c = Flag;   (* RID *)
5  color AckPDU_c = record
6     RID:Flag *               (* Retransmission Indicator *)
7     TveTok:Flag;             (* TID Verification /TID Ok *)
8  color AbortPDU_c = with abort;  (* No fields *)
9  color PDU = union InvokePDU:InvokePDU_c +
10                   ResultPDU:ResultPDU_c +
11                   AckPDU:AckPDU_c    +
12                   AbortPDU:AbortPDU_c;
13 var invoke:InvokePDU_c;
14 var result :ResultPDU_c;
15 var ack:AckPDU_c;
```

Each PDU type is modelled as a different colour set. The Invoke and Ack PDUs are modelled as records, with entries corresponding to the necessary header fields as defined in Section 7.2.1. Records are used (as opposed to products) as they allow meaningful names to be given to the entries. As only the `RID` header field is modelled for the Result

PDU, we define its colour set as Flag. The Abort PDU has no header fields to be modelled, and so we only need a token (abort) to represent the PDU.

The final colour set, PDU, is a union of all the PDU types. The variables invoke, result and ack are used to represent tokens of the respective PDU colour sets.

The definition of the PDU colour sets has been chosen so that the inscriptions in the TR-Protocol CPN (which are written in Standard ML) are clear and consistent. The state table pages in Section 7.6 will illustrate this.

Listing 7.3 defines the colour sets and variables that represent the state of the TR-PEs.

Listing 7.3: TR-PE state definitions used in the TR-Protocol declarations

```
(* Initiator & Responder State Names *)
color IStateName = with
   I_NULL              |
   I_RESULT_WAIT       |
   I_RESULT_RESP_WAIT  |
   I_WAIT_TIMEOUT ;

color RStateName = with
   R_LISTEN            |
   R_TIDOK_WAIT        |
   R_INVOKE_RESP_WAIT  |
   R_RESULT_WAIT       |
   R_RESULT_RESP_WAIT ;

(* Transaction Data - Initiator *)
color ITransData = record
   Uack:Flag    *   (* True if UserAck On *)
   RCR:RCR_c    *   (* Retransmission Counter *)
   AckSent:Flag*    (* True if Ack(TIDok) PDU sent *)
   HoldOn:Flag  *   (* True if Ack has been received *)
   Timer:Flag   ;   (* True if Timer on *)

(* Transaction Data - Responder *)
color RTransData = record
   Uack:Flag    *   (* True if UserAck On *)
   RCR:RCR_c    *   (* Retransmission Counter *)
   AckSent:Flag*    (* True if Ack PDU sent *)
   Timer:Flag   ;   (* True if Timer on *)

color InitState = product IStateName * ITransData;
color RespState = product RStateName * RTransData;

var isn : IStateName;
var rsn : RStateName;
var it : ITransData;
var rt : RTransData;
```

The state of a TR-PE for a single transaction comprises the name of the state and a set of values stored for that state. The colour sets IStateName and RStateName define

the set of state names for the TR-Init-PE and TR-Resp-PE, respectively. These state names are identical to the transition names on the TR-PE pages and the states defined in Table 5.8. The colour sets ITransData and RTransData define the values of the variables that are stored by the TR-PE (see Section 5.3.3). We refer to these sets as the *transaction data*.

The TR-Init-PE and TR-Resp-PE store some variables used for the same purpose. Uack and AckSent represent the variables of the same name, and RCR represents the counter of the same name described in Chapter 5. Timer models the transaction timer used by the TR-PEs as either on (T) or off (F). When a timer is off, a time-out of any sort cannot occur. When the timer is on, a time-out may occur. Modelling the timer in this way, as opposed to an actual timer, simplifies the TR-Protocol CPN and state space analysis, while maintaining the necessary functionality of the TR-Protocol timers (i.e. the possibility of a time-out is modelled, but not the actual timer).

The TR-Init-PE also uses the HoldOn variable. As only one transaction is modelled (Simplification 7.4), it is not necessary to include the variables `SendTID` and `RcvTID` for either of the TR-PEs .

The colour sets InitState and RespState are a product of their respective state names and transaction data colour sets. Variables are also defined for accessing each component of the product. Note that although there is some commonality between the transaction data colour sets of the TR-PEs, Standard ML does not provide any inheritance constructs and, therefore, we have defined the colour sets separately. This limits the maintainability of the TR-Protocol CPN.

Referring back to the TR_Init_PE page shown in Figure 7.6, we see that the state place Initiator has an initial marking that defines the initial state of the TR-Init-PE. The TR-Init-PE is in the state named I_NULL, all the boolean variables are initially false (F) and RCR is initially 0. These are the default values for the transaction data used by the TR-Init-PE. The arcs between the state place and transitions are all bi-directional. Again, the inscriptions are not necessary as they become apparent on the sub-pages presented in Section 7.6. However, in general the TR-Init-PE is modelled so that transitions on a sub-page are only enabled when the state name in the place Initiator matches the name of the sub-page. The I_ABORT page is one exception which is explained in Section 7.6.11.

Every transition on the TR_Init_PE page except, I_ABORT, has an output arc to the InitToResp communication place and an input arc from the RespToInit communication place. The output arcs indicate that PDUs are sent and the input arcs that PDUs are received. (Note that the transition I_RESULT_WAIT also has a dashed input arc *from* InitToResp. This models a special case where a transaction is aborted before a PDU is delivered to the TR-Resp-PE. This is discussed in Section 7.6.3.) The communication places have an annotation specifying they are *port places* (given by the boxed P). As

discussed in Chapter 4, port places indicate that it is an input (In), output (Out) or input/output (I/O) place on a sub-page, and they are assigned to *socket* places of the same name on the page where the sub-page is represented as a substitution transition (Figure 7.5).

The TR_Resp_PE page shown in Figure 7.7 has a similar structure to the TR_Init_PE page. There are five substitution transitions modelling the states of the TR-Resp-PE, and one modelling the T-Service-Provider initiated abort (R_ABORT). The TR-Resp-PE is initially in the state named R_LISTEN with all boolean variables set to false and RCR set to 0. Again, these are the default values for the transaction data used by the TR-Resp-PE.

The structure of the TR_Init_PE and TR_Resp_PE pages aids in visualizing each TR-PE from an abstract view. The substitution transitions (except I_ABORT and R_ABORT) are identical to the states shown in Table 5.8. During simulations and state space analysis, we can also view the current state of the TR-PE and the PDUs in the communication channel. However, the communication with the TR-Users is not shown, nor the procedures occurring in the TR-PE states. The sub-pages described in the next section show these details.

7.6 State Table Pages

There are 11 pages on the third level of the TR-Protocol CPN. Nine of them are sub-pages of the transitions on the TR-PE pages (Figure 7.6 and 7.7) and model the state tables described in Chapter 5. The other two pages (I_ABORT and R_ABORT) model the provider initiated aborts. All of the pages have a similar structure, which we outline in Section 7.6.1 using the page I_NULL as an example. Sections 7.6.2 to 7.6.11 then describe each state table page, with focus on the design decisions and any features not covered by the general page structure. We continue to introduce declarations when necessary.

7.6.1 General Page Structure

The state tables (Appendix B) used in the WTP Specification [183] to describe the TR-Protocol comprise, for each state of the TR-PE: an event; a set of conditions; a set of actions; and the next state of the TR-PE. In addition, before an incoming event is processed, a test is performed as specified by Table 5.6. This test table and the state tables are the main source of information in the WTP Specification for our TR-Protocol CPN. We have chosen to model each state table as a separate page (called a *state table page*). Several events in the test table are also included on these state table pages. These will be discussed in Sections 7.6.2 and 7.6.6. The advantages of modelling the

TR-Protocol CPN as state table pages are:

- the repetitive structure aids in understanding of the TR-Protocol CPN—once the general structure of one page is understood, it should be easy to grasp the meaning of the other pages; and
- the relationship between the TR-Protocol CPN and the TR-Protocol state tables simplifies validation of the model.

One disadvantage of the structure chosen is that we have several transitions that model the same actions (although in different states). It may be possible to combine such transitions so that they model a common action that can be applied in multiple states. However, the ability to quickly validate the transitions with the state tables outweighs this disadvantage.

The remainder of this sub-section describes the purpose and conventions for the state places, arc inscriptions and transitions used in the TR-Protocol CPN. Also presented is a modelling decision not to include entries of the state tables that ignore PDUs. The I_NULL page shown in Figure 7.8 is used as an example.

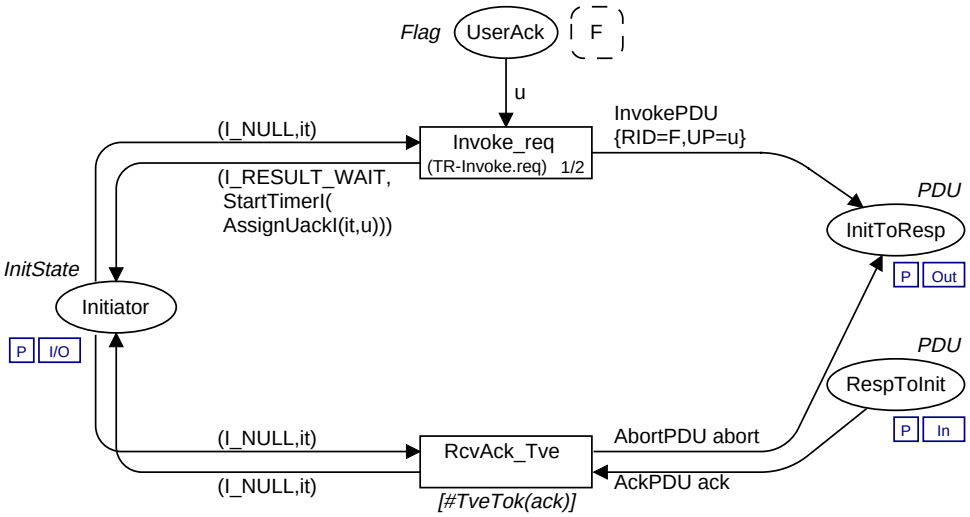

Figure 7.8: I_NULL page in the TR-Protocol CPN

State Places and Arc Inscriptions

Each state table page contains a state place, depending on which TR-PE the page is describing. The state place stores the current state name and transaction data. Every transition has an input arc and output arc between it and the state place. The input arc is always shown higher on the page than the output arc. The inscription of the input and output arcs are shown above and below the arcs, respectively.

An input arc from the state place to a transition puts a condition on the enabling of the transition that it must be in the state being modelled by the state table page. For example, on the I_NULL page (Figure 7.8), the input arc inscriptions is (I_NULL,it). One condition for these transitions being enabled is that the TR-Init-PE is in the I_NULL state. The variable it (and rt on the TR-Resp-PE state table pages) indicates the transaction data can take any values. Further enabling conditions are put on the transitions using guards (discussed shortly). The output arcs from the transitions to a state place specify the next state of the TR-PE. The state name and the transaction data may or may not be changed. As Standard ML records are used to model the transaction data, all entries must be defined when changing the transaction data. Therefore, we have defined a set of functions that modify the transaction data. Using these functions on the output arc inscriptions, as opposed to showing the complete entry, increases the clarity of the pages. For example, Listing 7.4 defines the two functions used on the output arc of transition Invoke_req to Initiator.

Listing 7.4: Selected functions used in the TR-Protocol declarations

```
1  fun AssignUackI ( t:ITransData, u:Flag ):ITransData =
2          {Uack          =       u,
3           RCR           =       #RCR(t),
4           AckSent       =       #AckSent(t),
5           HoldOn        =       #HoldOn(t),
6           Timer         =       #Timer(t)}
7  fun StartTimerI  ( t:ITransData ):ITransData =
8          {Uack          =       #Uack(t),
9           RCR           =       #RCR(t),
10          AckSent       =       #AckSent(t),
11          HoldOn        =       #HoldOn(t),
12          Timer         =       T}
```

AssignUackI() has two parameters: t representing the transaction data at the TR-Init-PE and u representing the value of `UserAck` to be used for the transaction. The function assigns the Uack entry of the record in the transaction data the value u and assigns the remaining record entries to those given in t. The result is the transaction data given as input is updated with the value u for the Uack entry. StartTimerI() performs a similar update, but sets Timer to true (T). From the output arc in Figure 7.8, we see that the occurrence of Invoke_req changes the state name of the TR-Init-PE from I_NULL to I_RESULT_WAIT, assigns the variable Uack to u in the transaction data and starts the timer.

Functions similar to StartTimerI() are defined for updating other variables and counters in the transaction data for the TR-PEs. The function names are based on the actions in the state tables (Appendix B). This gives strong correlation between the actions and the arc inscriptions, simplifying validation of the model. The complete set of functions is

given Listing D.1. Note that separate functions are required for the TR-Init-PE and TR-Resp-PE because the transaction data records (ITransData and RTransData) are different (cf. Standard ML doesn't support inheritance). This further limits the maintainability of the TR-Protocol CPN.

Transitions

In general, transitions on state table pages model entries in the state tables. The following two conventions are used for the transitions that model entries in the state tables:

1. Transition names are given based on the entry event. In some cases the names are modified to obtain unique and meaningful names for each state table page.

2. A number in the lower right hand corner of the transition indicates the entry that the transition is modelling. Recall that we have numbered the Transaction Class 2 events in the state tables from the WTP Specification (Appendix B). In some cases there are multiple numbers (separated by a forward slash (/)) meaning the transition models multiple entries. The numbers simplify the comparisons with the state tables when validating the model.

For example, the I_NULL page has a transition named Invoke_req which models both Entry 1 and Entry 2 in the NULL state table for the TR-Init-PE (Table B.1).

The purpose of the transitions can be classified into six groups:

Primitive Submission: A TR-User submits a TR-Service primitive to the TR-PE. These transitions model the event of the TR-PE receiving the primitive from the TR-User. This also implicitly models the TR-User submitting the primitive. The primitive submitted by the TR-User is shown in parenthesis below the transition name. For example, transition Invoke_req models the submission of the TR-Invoke.req primitive by the TR-Init-User.

Primitive Delivery: A TR-PE delivers a TR-Service primitive to the TR-User. These transitions model the event of the TR-PE sending the primitive to the TR-User. This also implicitly models the TR-User being delivered the primitive. The primitive delivered to the TR-User is shown in parenthesis below the transition name.

Sending a PDU: A TR-PE sends a PDU to the peer TR-PE. An arc from the transition to an appropriate communication place is present. The inscription on the arc indicates the type of PDU and the values set in the header fields. For example, transition Invoke_req indicates an Invoke PDU is sent, with RID set to F and UP set to the value of u (u is discussed in Section 7.6.2).

Receiving a PDU: A TR-PE receives a PDU from the peer TR-PE. An arc from an appropriate communication place to the transition is present. The inscription on the arc indicates the type of PDU that may be received. For example, transition RcvAck_Tve indicates an Ack PDU is received from the TR-Resp-PE.

Time-out: A timer reaches its maximum value. If the timer has an associated counter (e.g. RCR), and that counter is less than its maximum value, then the counter is increased. If the counter is equal to the maximum value, when a time-out transition occurs, then the transaction is aborted. Examples of time-out transitions will be seen in other state table pages (e.g. Section 7.6.3).

Provider Abort: The T-Service-Provider initiates an abort. These transitions are discussed in Section 7.6.11.

A transition will have one or more of the above purposes. For example, the transition Invoke_req signifies both a submission of the TR-Invoke.req primitive and sending of the Invoke PDU. Guards on the transitions (shown in italics and inside square brackets) restrict their enabling based on the transaction data values and the header fields of the received PDUs. For example, the guard on transition RcvAck_Tve indicates the received PDU is an Ack(Tve) PDU (i.e. TveTok=T).

Entries that Ignore PDUs

From the state tables and Assumption 7.4, there are entries that ignore PDUs upon receipt. We model these *ignore entries* by leaving the PDUs in the communication channel. A transition could be introduced which: a) discards the PDU and b) leaves the TR-PE in the same state. However, we argue that the same result occurs if the PDU is left in the channel, because the channel allows for re-ordering of PDUs, and hence following PDUs will not be blocked by the PDU that is ignored. (Preliminary analysis of a model that included transitions modelling the ignore entries did indeed give the same result, in terms of the four desired properties defined in Chapter 8. However, the extra transitions resulted in increases in the numbers of nodes (by a factor of approximately 1.3) and number of arcs (factor of 2.5), and the processing times doubled.) The only difference is that the marking of the communication places will include the PDUs that are ignored, and thus our definition of a successful terminal state (see Chapter 8, Property 8.2) needs to take this into account. This has the beneficial effect of reducing the size of the state space, while preserving the essential behaviour of the protocol.

7.6.2 I_NULL

The I_NULL page is shown in Figure 7.8. This page models the TR-Init-PE while there is no transaction in progress (see Table B.1). The TR-Init-PE is initially in the state named I_NULL. As well as the components described in Section 7.6.1, the I_NULL page contains a place called UserAck. This place is typed by the colour set Flag and its initial marking models the selection of the UserAck feature by the TR-Init-User. In Figure 7.8 the initial marking is F, indicating that UserAck is Off. The analysis in Chapters 8 and 9 consider scenarios where the initial marking is T as well. We chose to model UserAck as a parameter to the TR-Protocol CPN (via the initial marking) so we can investigate the behaviour when UserAck is Off and On separately, in the same manner as was done for the TR-Service CPN in Chapter 6. Modelling UserAck non-deterministically is possible, but it will increase the state space size.

The Invoke_req transition models Entries 1 and 2 of Table B.1. The only difference between the entries is the variable `Uack` is set to true when the parameter UserAck in the TR-Invoke.req primitive is set. Therefore, upon occurrence of Invoke_req, the variable Uack in the transaction data is assigned the value of u (which is a variable of type Flag), obtained from the place UserAck, using the function AssignUackI. Invoke_req is the only transition enabled in the initial marking of the TR-Protocol CPN. Upon its occurrence, an Invoke PDU is sent to the TR-Resp-PE and the TR-Init-PE enters the I_RESULT_WAIT state.

Transition RcvAck_Tve models Entry 2 of the tests on incoming events given in Table 5.6. If the TR-Init-PE has no transaction outstanding (i.e. in the state named I_NULL), then, on receipt of an Ack(Tve) PDU, the TR-Init-PE sends an Abort PDU to the TR-Resp-PE. The Abort PDU signifies a negative response to a TID verification.

We have only modelled Entries 1 and 2 of Table 5.6 (the tests on incoming events). Entry 3 states that Ack, Result and Abort PDUs must be ignored if received for a transaction that is not outstanding, and therefore correspond to *ignore entries* that are not explicitly modelled. The remaining entries of Table 5.6 are not modelled because the corresponding features are outside the scope of the TR-Protocol CPN (see Section 7.2).

7.6.3 I_RESULT_WAIT

The I_RESULT_WAIT page is shown in Figure 7.9. This page models the TR-Init-PE waiting for the Result PDU (and Ack PDUs) after it has sent an Invoke PDU (see Table B.2). There are 11 transitions, one of which is a substitution transition. The first two transitions at the top of the page, UserAbort and ProviderAbort, model the special cases of a transaction being aborted before the first Invoke PDU has delivered to the TR-Resp-PE (i.e. the transaction is complete without the TR-Resp-PE being aware that

the TR-Init-User started it). We will explain these two transitions, which do not model specific entries in the state tables, in further detail shortly.

The Abort_req transition models the TR-Init-User submitting a TR-Abort.req primitive which aborts the transaction. The TR-Init-PE sends an Abort PDU to notify the TR-Resp-PE that the transaction has been aborted, and re-enters the I_NULL state. Every transition modelling the TR-Init-PE aborting a transaction causes the TR-Init-PE to re-enter the I_NULL state and re-set the transaction data to its default values (using the function ClearInitI()).

Returning to the top two transitions on this page, we see that UserAbort, like the transition Abort_req, models the TR-Init-User submitting a TR-Abort.req primitive. However, UserAbort models the special case where an Invoke PDU has not yet been sent by the local host of the TR-Init-PE. Therefore, instead of the local host sending the Invoke PDU and Abort PDU, neither PDUs are sent, and the transaction is ended (the TR-Init-PE enters the I_NULL state). However, an InvokePDU token as been added to the place InitToResp. In this special case, the marking of InitToResp with an InvokePDU token does not mean the Invoke PDU has been sent by the local host (i.e. it cannot yet be received by the TR-Resp-PE)—it is only ready to be sent. UserAbort has a dashed input arc *from* InitToResp that removes the InvokePDU so it cannot be received by the TR-Resp-PE. The guard on UserAbort specifies this special case can only occur if there have been no re-transmissions of the Invoke PDU (once a re-transmission occurs, the original Invoke PDU must have been sent).

The transition ProviderAbort models the special case where the T-Service-Provider initiates an abort after the submission of the TR-Invoke.req primitive, but before the Invoke PDU has been delivered to the TR-Resp-PE. (The general case of a T-Service-Provider initiated abort is modelled on the I_ABORT page—see Section 7.6.11.) Like UserAbort, no re-transmissions can have occurred, and the InvokePDU token is removed from the InitToResp place. The TR-Init-PE enters the I_NULL state. Although in the CPN UserAbort and ProviderAbort perform the same operations, they are modelled as separate transitions because they correspond to different primitives occurring. For the language analysis in Chapter 8 it is necessary that there is a one-to-one mapping of service primitives to transitions.

The next three transitions, TimerTO_R_Max, TimerTO_R and TimerTO_R_Tve, model the re-transmission timer expiring. The transition guards specify different conditions on the time-outs. All transitions that model a time-out must have the Timer entry in the transaction data set to T, i.e. the timer must be on for a time-out to occur. We now introduce the final set of declarations for the TR-Protocol CPN that define the constants used for the maximum counter values (Listing 7.5).

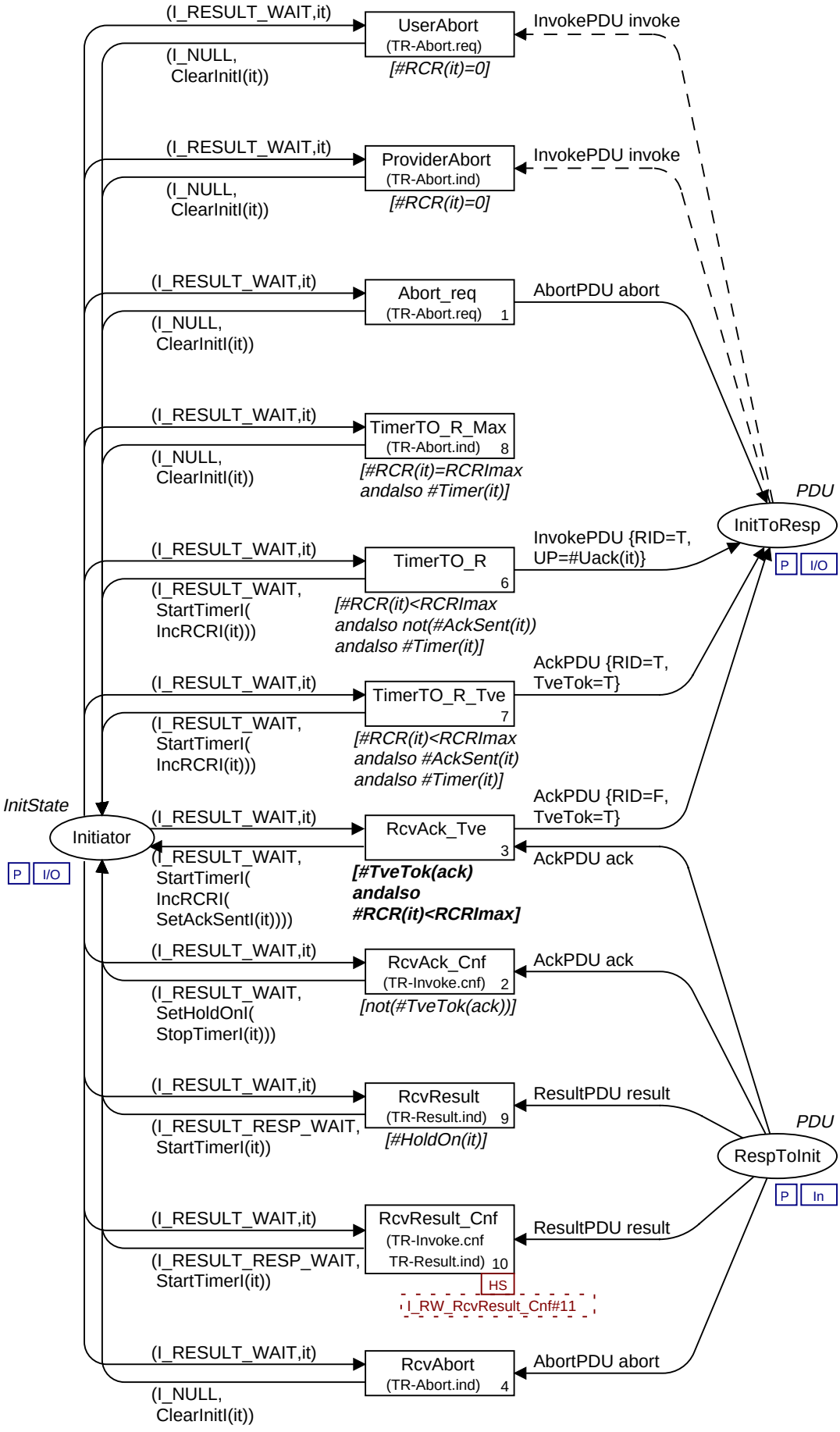

Figure 7.9: I_RESULT_WAIT page in the TR-Protocol CPN

Listing 7.5: Constants used in the TR-Protocol declarations

```
1 val RCRImax = 1;
2 val RCRRmax = 1;
```

Each TR-PE uses a constant representing the maximum value of the counter RCR. The constants are identified with the letter I for the TR-Init-PE (RCRImax) or letter R for the TR-Resp-PE (RCRRmax). As the maximum counter values do not change during a transaction, we chose to model them as constants, as opposed to using places with tokens representing the values. Chapter 8 discusses the analysis of the TR-Protocol CPN using different values of these constants.

The time-out transitions on the I_RESULT_WAIT page compare the value of RCR in the transaction data to the constant RCRImax. If the two are equal (transition TimerTO_R_Max), then the TR-Init-PE has already re-transmitted the maximum number of times, and must abort the transaction. If RCR is less than RCRImax then a re-transmission occurs upon a time-out. If the TR-Init-PE has previously received an Ack(Tve) PDU and subsequently sent an Ack(Tok) PDU, then the Ack(Tok) PDU is re-sent (transition TimerTO_R_Tve). The TR-Init-PE knows an Ack(Tok) PDU was previously sent because the AckSent flag is set. Upon re-sending the Ack(Tok) PDU, the TR-Init-PE increments RCR, starts the timer and remains in the I_RESULT_WAIT state. If AckSent was not set, then the TR-Init-PE re-transmits the Invoke PDU (transition TimerTO_R) that was initially sent when the TR-Init-User submitted the TR-Invoke.req primitive (Invoke_req on page I_NULL). Again, the TR-Init-PE increments RCR, starts the timer and remains in the I_RESULT_WAIT state. The RID header field of both re-transmitted PDUs is set to T.

It is not necessary to model the cases where RCR is greater than its maximum value because it is initially less than or equal to its maximum, and every transition that increases RCR (by 1) can only occur if RCR is less than RCRImax.

The remaining five transitions on the I_RESULT_WAIT page model the receipt of PDUs from the TR-Resp-PE. If the TR-Init-PE receives an Ack(Tve) PDU while in the I_RESULT_WAIT state, the transaction is outstanding and so an Ack(Tok) PDU is sent in reply to the TID verification. Transition RcvAck_Tve models this case. Note that the guard specifies RCR in the transaction data must be less than RCRImax. This models the fix introduced to limit RCR as discussed in Assumption 7.7 and accepted by the WAP Forum [55].

The receipt of an Ack PDU is modelled by transition RcvAck_Cnf. The TR-Init-PE remains in the I_RESULT_WAIT state, sets HoldOn to T and stops the timer. A TR-Invoke.cnf primitive is also delivered to the TR-Init-User.

The receipt of a Result PDU when HoldOn is T is modelled by transition RcvResult. A TR-Result.ind primitive is delivered to the TR-Init-User and the TR-Init-PE enters

the I_RESULT_RESP_WAIT state. Note that Entry 9 in Table B.2 states the timer is stopped, a TR-Result.ind primitive is generated and then the timer is started with the interval A. As we are modelling the entries as atomic events (Assumption 7.1) and the timer actions are ordered (Assumption 7.3), we only apply the function StartTimerI() on the transaction data.

Transition RcvResult_Cnf models Entry 10 in Table B.2. This is a substitution transition whose sub-page is I_RW_RcvResult_Cnf. Entry 10 is modelled differently from other entries because two primitives are delivered to the TR-Init-User. We leave the discussion of this until Section 7.7.

The final transition on the I_RESULT_WAIT page is RcvAbort. This models the receipt of an Abort PDU which signals the delivery of a TR-Abort.ind primitive to the TR-Init-User and returns the TR-Init-PE to the I_NULL state.

From the I_RESULT_WAIT page, we see how the choice of Standard ML constructs for the PDUs (see Listing 7.2) results in a clear and consistent approach for the inscriptions specifying the sending and receiving of PDUs.

The I_RESULT_WAIT page does not model Entry 5 of the TR-Init-PE RESULT WAIT state table (Table B.2). The event of this entry is the receipt of an erroneous PDU. This event is not modelled because we assume illegal PDUs are not possible (Restriction 7.3). This applies for all state tables that have similar entries.

7.6.4 I_RESULT_RESP_WAIT

The I_RESULT_RESP_WAIT page is shown in Figure 7.10. This page models the TR-Init-PE waiting for the TR-Init-User to acknowledge the receipt of the result (see Table B.3). There are five transitions, two of which model a time-out occurring when the timer interval A is used. This interval, and the corresponding counter `AEC`, is used (when UserAck is On) to determine the length of time a TR-PE waits for an acknowledgment from the TR-User before aborting a transaction. As a time-out with interval A (when `AEC` is less than its maximum value) only increments `AEC` (e.g. no PDUs are sent, the state name is not changed—see Entry 7 of Table B.3), `AEC_MAX` acts simply as a multiple of the interval A. Rather than modelling `AEC` explicitly, non-determinism is used. Either a time-out with interval A occurs when `AEC` is at its maximum (`AEC_MAX`), hence causing the transaction to be aborted, or the timer is still running (providing it hasn't been stopped by other events). Transition TimerTO_A_Max models a time-out when `AEC_MAX` is reached (Entry 8, Table B.3). Note that a TR-Abort.ind primitive is delivered to the TR-Init-User (see Assumption 7.8). There is no transition modelling Entry 7 of Table B.3. This design decision is made to simplify the analysis, so that the results will be independent of the value of `AEC_MAX`. (Such an abstraction is not used for `RCR_MAX` because a time-out on

interval R triggers PDUs to be sent. These PDUs impact other parts of the TR-Protocol. If these time-outs (e.g. transition TimerTO_R on page I_RESULT_WAIT (Figure 7.9)) were modelled non-deterministically, then there would be no limit on the number of PDUs sent, most likely resulting in an infinite state space. Modelling `RCR` explicitly, allows a practical bound to be placed on the state space. Chapter 9 discusses the impact of `RCR_MAX` on the size of the state space.)

When UserAck is Off, it is not necessary to wait for the TR-Init-User to respond. When the time-out occurs (transition TimerTO_A_Off), an Ack PDU is sent to the TR-Resp-PE to indicate the Result PDU has been received. The TR-Init-PE does not wait for the submission of the TR-Result.res primitive from the TR-Init-User, but enters the I_WAIT_TIMEOUT state and starts the timer (with interval W).

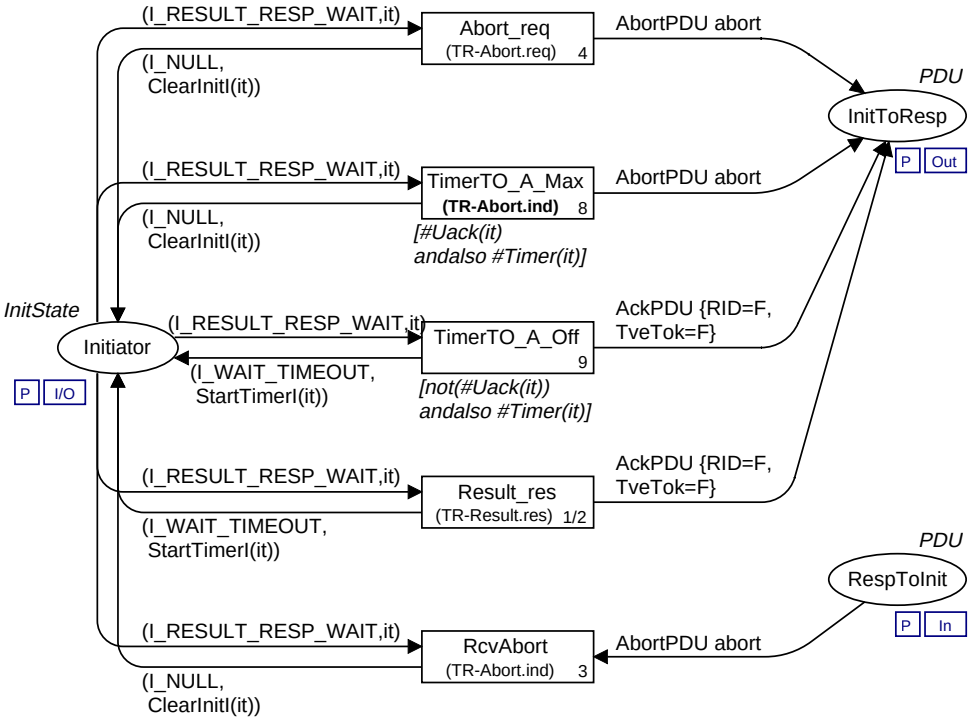

Figure 7.10: I_RESULT_RESP_WAIT page in the TR-Protocol CPN

The Abort_req and RcvAbort transitions model the same behaviour as those with the same name on the I_RESULT_WAIT page. Transition Result_res models the TR-Init-User submitting the TR-Result.res primitive, which signals an Ack PDU is to be sent to the TR-Resp-PE and the TR-Init-PE to enter the I_WAIT_TIMEOUT state. Both Entries 1 and 2 of Table B.3 are modelled by Result_res because the `ExitInfo` parameter of the TR-Result.res primitive is not modelled (Simplification 7.3).

The I_RESULT_RESP_WAIT state table does not model Entry 6 of the TR-Init-PE RESULT RESP WAIT state table (Table B.3), as it is an *ignore entry*.

7.6.5 I_WAIT_TIMEOUT

The I_WAIT_TIMEOUT page is shown in Figure 7.11. This page models the TR-Init-PE storing the transaction data in case the Ack PDU needs to be re-transmitted (see Table B.4). There are four transitions on this page.

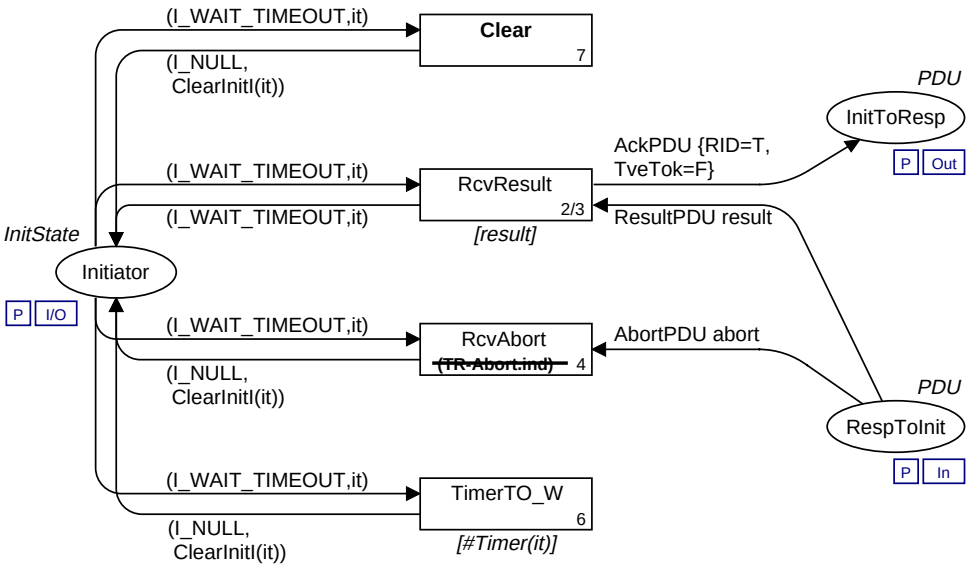

Figure 7.11: I_WAIT_TIMEOUT page in the TR-Protocol CPN

Transition RcvResult models the TR-Init-PE receiving a re-transmitted Result PDU (recall from Listing 7.2 that the variable result is used to identify the only header field modelled in the Result PDU, `RID`). An Ack PDU is re-transmitted and the TR-Init-PE remains in the I_WAIT_TIMEOUT state. Both Entries 2 and 3 of Table B.4 are modelled by RcvResult because the `ExitInfo` option is not modelled (Simplification 7.3).

Transition TimerTO_W models a time-out with interval W occurring. The action specified in the state table (Entry 6, Table B.4) is to clear the transaction. We interpret this as meaning clear all data for this transaction. This is performed using the function ClearInitI(), and the TR-Init-PE enters the I_NULL state.

Transitions Clear and RcvAbort model Entries 7 and 4 of Table B.4, respectively. However, they also incorporate the changes specified in Table 7.8. Neither of these transitions model service primitives occurring, as the original state table indicated.

Entry 1 in the WAIT TIMEOUT state table (Table B.4) is not modelled by a transition because it only specifies the TR-Init-PE to ignore the Result PDU. Entry 5 is not modelled by a transition because error PDUs cannot be received.

7.6.6 R_LISTEN

The R_LISTEN page is shown in Figure 7.12. This page models the TR-Resp-PE ready to accept transactions. There are two transitions and one new place named First. The

TR-Resp-PE is initially in the R_LISTEN state, so transitions on this page will be the first to occur in the TR-Resp-PE state table pages. The page plays a similar role to the I_NULL page for the TR-Init-PE. It models entries in the LISTEN state table (Table B.5).

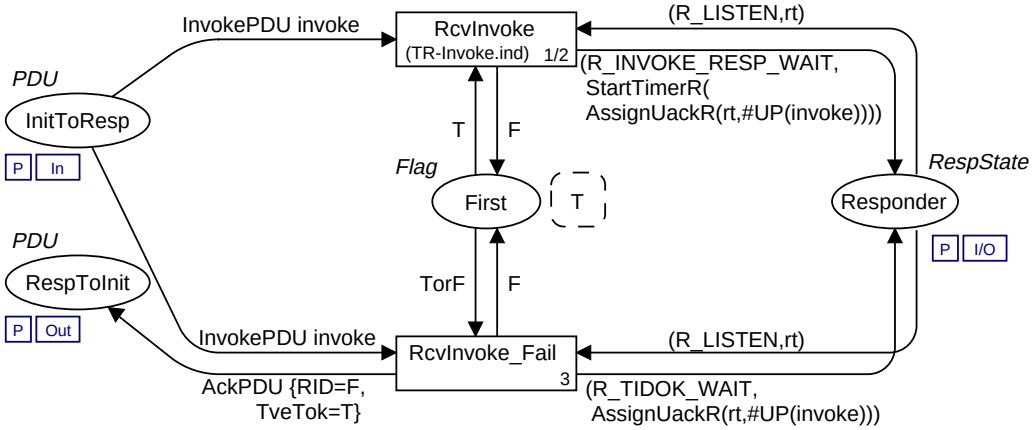

Figure 7.12: R_LISTEN page in the TR-Protocol CPN

Transition RcvInvoke models the receipt of an Invoke PDU with a valid TID (Entries 1 and 2). The two entries are modelled by one transition because they are only differentiated by the value of the U/P flag, which is set in the transaction data to the value of the UP header field in the Invoke PDU. A TR-Invoke.ind primitive is delivered to the TR-Resp-User and the TR-Resp-PE enters the R_INVOKE_RESP_WAIT state with the timer started (using interval A). The marking of First must also be 1‘T for RcvInvoke to be enabled. First always has the initial marking of 1‘T.

Transition RcvInvoke_Fail models the receipt of an Invoke PDU with an invalid TID (Entry 3). The TR-Resp-PE initiates TID verification by sending an Ack(Tve) PDU and entering the R_TIDOK_WAIT state. The place First also places a condition on the enabling of RcvInvoke_Fail.

As stated in Simplification 7.6, a specific mechanism for determining if a TID is valid is not modelled. Instead, on the receipt of the first Invoke PDU an arbitrary choice is made on the validity of the TID (i.e. RcvInvoke or RcvInvoke_Fail could occur). However, after the first Invoke PDU has been received, any re-transmitted Invoke PDUs received by the TR-Resp-PE while in the R_LISTEN state must have invalid TIDs (i.e. TID verification is initiated). Suppose a caching mechanism is used, then the TID of the first Invoke PDU would be stored in cache. A test on the following Invoke PDUs with the same TID must indicate an invalid TID[1]. If no caching mechanism is used, then every TID will be invalid. Place First ensures that, after the first Invoke PDU is received (transition RcvInvoke or RcvInvoke_Fail occur), all subsequent Invoke PDUs have invalid TIDs, therefore, initiating TID verification (only transition RcvInvoke_Fail is enabled). Note that the inscription on

[1] This assumes the caching mechanism and TID test implemented does ensure invalid TIDs are detected. This follows from the discussion in Section 7.2.1 and Simplification 7.4.

the arc from First to RcvInvoke_Fail, TorF, is a variable of type Flag. The marking of First can be T or F for RcvInvoke_Fail to be enabled (providing the other enabling conditions of the transition are satisfied).

The receipt of PDUs that are ignored (Ack and Abort) are again not modelled. Entry 4 in the LISTEN state table (Table B.5) is not modelled by a transition because error PDUs cannot be received.

7.6.7 R_TIDOK_WAIT

The R_TIDOK_WAIT page is shown in Figure 7.13. This page models the TR-Resp-PE waiting for a response from the TR-Init-PE after initiating TID verification (see Table B.6). There are three transitions on this page.

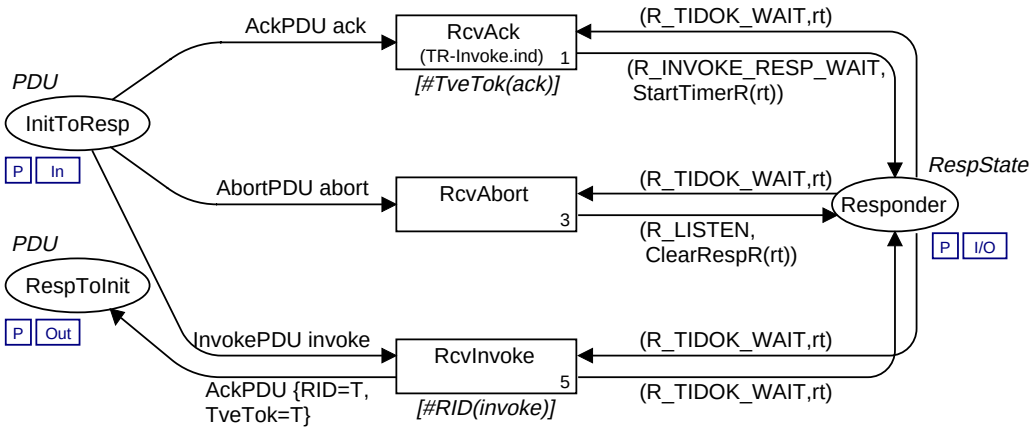

Figure 7.13: R_TIDOK_WAIT page in the TR-Protocol CPN

Transition RcvAck models the receipt of an Ack(Tok) PDU (i.e. a positive response to TID verification). This stimulates the delivery of the TR-Invoke.ind primitive to the TR-Resp-User. The TR-Resp-PE enters the R_INVOKE_RESP_WAIT state and proceeds with the transaction with the acknowledgment timer on.

Transition RcvAbort models the receipt of an Abort PDU which terminates the transaction. A TR-Abort.ind primitive is not delivered to the TR-Resp-User because a TR-Invoke.ind has not yet been delivered (i.e. the TR-Resp-User does not know a transaction was initiated).

Transition RcvInvoke models the receipt of a re-transmitted Invoke PDU. The TR-Resp-PE re-transmits the Ack(Tve) PDU and remains in the R_TIDOK_WAIT state.

As already discussed, Entry 4 in the TIDOK WAIT state table (Table B.6) is not modelled by a transition because it only specifies the TR-Resp-PE to ignore the Invoke PDU. Entry 2 is not modelled by a transition because error PDUs cannot be received.

7.6.8 R_INVOKE_RESP_WAIT

The R_INVOKE_RESP_WAIT page is shown in Figure 7.14. This page models the TR-Resp-PE waiting for the TR-Resp-User to acknowledge receipt of the Invoke PDU (see Table B.7). There are six transitions, two of which model the expiration of the acknowledgment timer. Transitions TimerTO_A_Max and TimerTO_A_Off model the time-outs in a similar way to the transitions of the same name on the I_RESULT_RESP_WAIT page. Again, `AEC` and `AEC_MAX` are not explicitly modelled, meaning Entry 7 of Table B.7 does not have a corresponding transition. Note that the occurrence of transition TimerTO_A_Max signals the delivery of a TR-Abort.ind primitive to the TR-Resp-User. This implements the change given by Table 7.6.

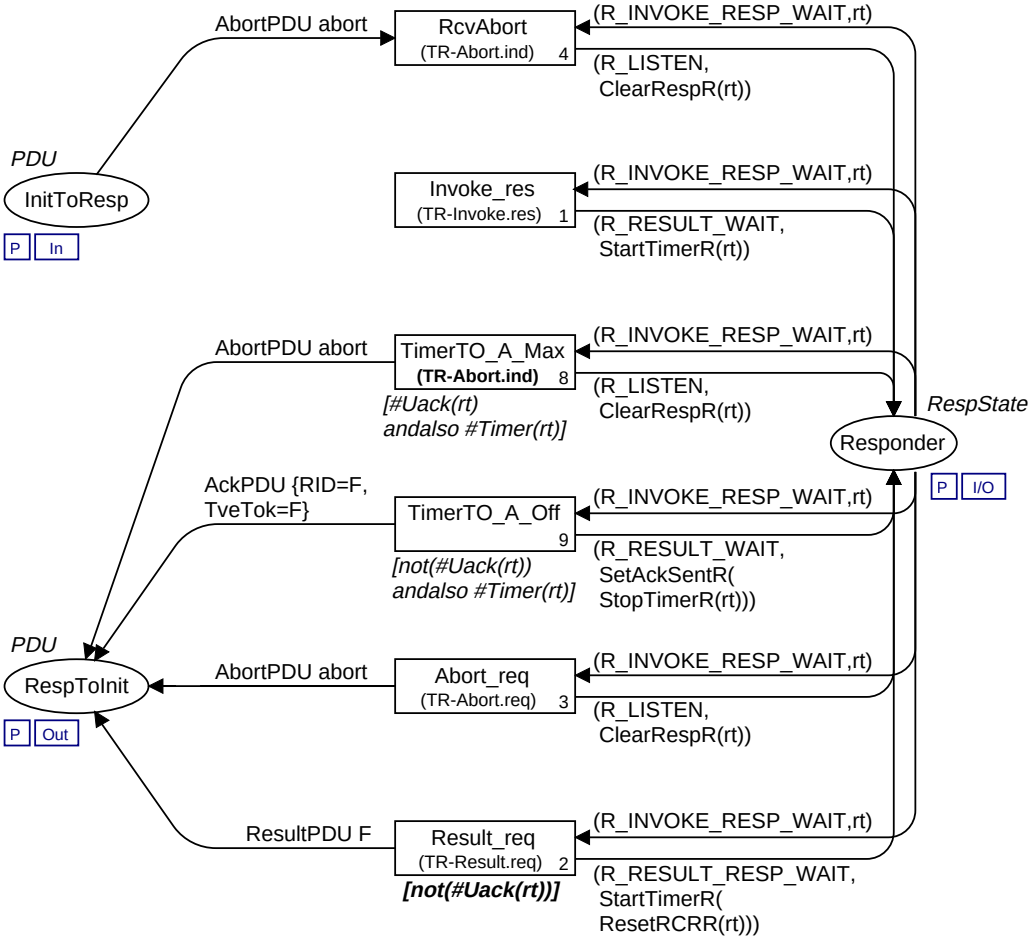

Figure 7.14: R_INVOKE_RESP_WAIT page in the TR-Protocol CPN

Transitions RcvAbort and Abort_req model behaviour in a similar way to the transitions of the same name on the I_RESULT_WAIT page. The TR-Resp-PE enters the R_LISTEN state and clears the transaction data using the function ClearRespR() when the transaction is aborted. Similar transitions are also included on the R_RESULT_WAIT and R_RESULT_RESP_WAIT pages.

Transition Invoke_res models the TR-Resp-User submitting the TR-Invoke.res primi-

tive. The timer is started and the TR-Resp-PE enters the R_RESULT_WAIT state.

Transition Result_req models the TR-Resp-User submitting the TR-Result.req primitive which causes a Result PDU (with `RID` set to F) to be sent to the TR-Init-PE, and the timer to be started. The TR-Resp-PE enters the R_RESULT_RESP_WAIT state. Note that this transition can only occur when UserAck is Off (see Assumption 7.9).

Entry 5 in the INVOKE RESP WAIT state table (Table B.7) is not modelled by a transition because it only specifies the TR-Resp-PE to ignore the Invoke PDU. Entry 6 is not modelled by a transition because error PDUs cannot be received.

7.6.9 R_RESULT_WAIT

The R_RESULT_WAIT page is shown in Figure 7.15. This page models the TR-Resp-PE waiting for the TR-Resp-User to submit the result (see Table B.8). There are five transitions, two of which model the same behaviour as RcvAbort and Abort_req on the R_INVOKE_RESP_WAIT page.

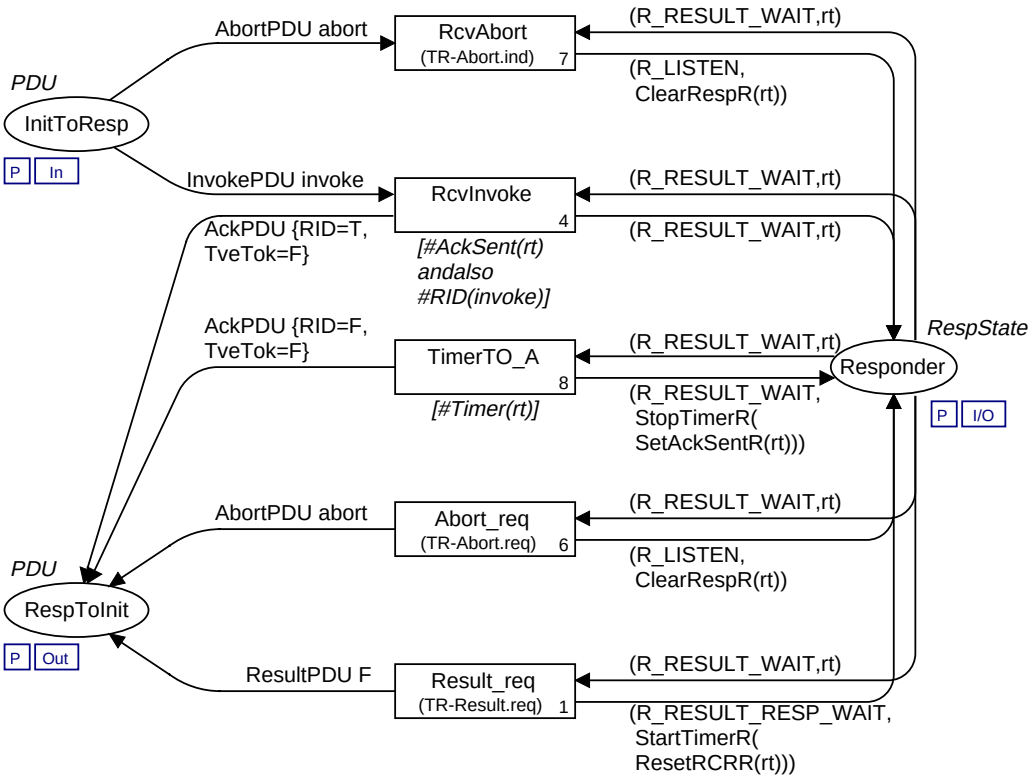

Figure 7.15: R_RESULT_WAIT page in the TR-Protocol CPN

Transition Result_req models the same behaviour as the transition with the same name on the R_INVOKE_RESP_WAIT page. However, UserAck can now also be Off.

Transition TimerTO_A models a time-out with interval A. An Ack PDU is sent acknowledging the Invoke PDU. If a re-transmitted Invoke PDU is received in the R_RESULT_WAIT state and an Ack PDU has already been sent (transition RcvInvoke),

then the Ack PDU is re-transmitted.

Entries 2 and 3 in the RESULT WAIT state table (Table B.8) are not modelled by a transition because they only specify the TR-Resp-PE to ignore the Invoke PDU. Entry 5 is not modelled by a transition because error PDUs cannot be received.

7.6.10 R_RESULT_RESP_WAIT

The R_RESULT_RESP_WAIT page is shown in Figure 7.16. This page models the TR-Resp-PE waiting for the TR-Init-PE to acknowledge the Result PDU (see Table B.9). There are five transitions, two of which model the same behaviour as RcvAbort and Abort_req on the R_INVOKE_RESP_WAIT page.

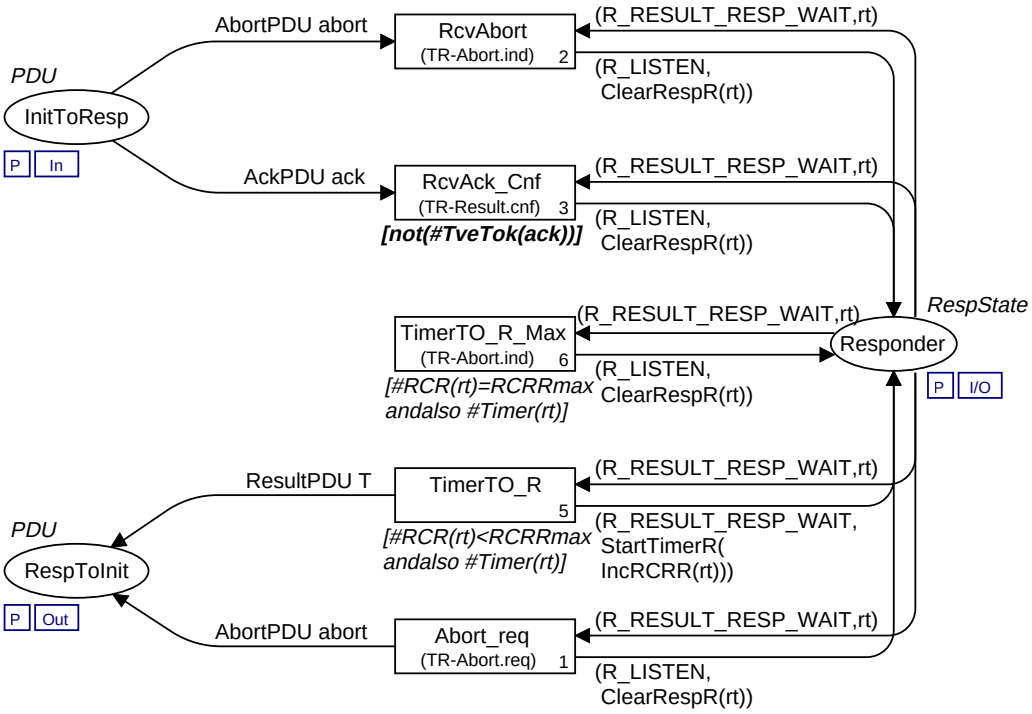

Figure 7.16: R_RESULT_RESP_WAIT page in the TR-Protocol CPN

Transitions TimerTO_R_Max and TimerTO_R model the expiration of the re-transmission timer in a similar manner to the transitions of the same name on page I_RESULT_WAIT. If RCR is less than RCRRmax, then the Result PDU is re-transmitted and RCR incremented (transition TimerTO_R). If RCR is equal to RCRRmax, then the transaction is aborted (transition TimerTO_R_Max).

Transition RcvAck_Cnf models the receipt of an Ack PDU from the TR-Init-PE and signals the delivery of the TR-Result.cnf primitive to the TR-Resp-User. (Note that an Ack(Tok) PDU is ignored—see Assumption 7.5.) The transaction is completed, and the TR-Resp-PE returns to the R_LISTEN state, with all transaction data cleared.

Entry 4 in the RESULT RESP WAIT state table (Table B.9) is not modelled by a transition because error PDUs cannot be received.

7.6.11 I_ABORT and R_ABORT

The I_ABORT and R_ABORT pages are shown respectively in Figures 7.17 and 7.18. These pages model the T-Service-Provider initiating aborts. They differ from the other state table pages, which model specific state tables from the WTP Specification [183].

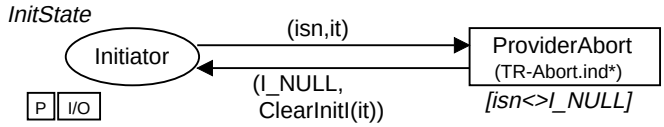

Figure 7.17: I_ABORT page in the TR-Protocol CPN

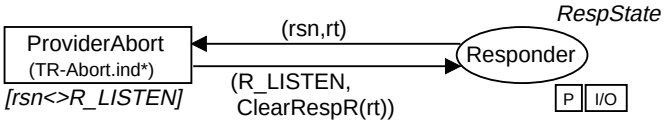

Figure 7.18: R_ABORT page in the TR-Protocol CPN

Transition ProviderAbort models an abort initiated by the T-Service-Provider, and being delivered to the TR-Init-User. The combination of the input arc and guard of this transition means it is enabled when the TR-Init-PE is in any state except I_NULL. A provider abort cannot occur while the TR-Init-PE is in the I_NULL state because either the transaction has not started, or the transaction has been completed. When ProviderAbort occurs, the TR-Init-PE aborts the transaction and enters the I_NULL state. The transaction data is reset to its default value by the function ClearInitI(). An asterisk is given inside the transition to indicate the delivery of the TR-Abort.ind primitive to the TR-Init-User is conditional. In this case, the TR-Abort.ind primitive will be delivered in all cases, except when the TR-Init-PE is in the I_WAIT_TIMEOUT state (recall from Section 7.6.5 that primitives are not delivered to the TR-Init-User in this state). As the delivery of primitives to TR-Users is not modelled explicitly, the conditions are not included in the TR-Protocol CPN. Instead, the condition is given when we specify the mapping of binding elements to service primitives. This information is used in the analysis when calculating the TR-Protocol language (Chapter 8).

We chose to model the ProviderAbort transition on a sub-page to maintain consistency with all other pages in the hierarchy. Alternatively, the actual transition could have been included on the TR_Init_PE page (instead of the substitution transition I_ABORT).

There is only one transition on the R_ABORT page. This transition, ProviderAbort, models a T-Service-Provider initiated abort being delivered to the TR-Resp-User. It is similar to the transition of the same name on the I_ABORT page. It is enabled in any state except R_LISTEN. The TR-Abort.ind primitive is not delivered to the TR-Resp-User in the R_TIDOK_WAIT state because the TR-Invoke.ind primitive has not yet been delivered. This will become apparent in Chapter 8.

7.7 Multiple Primitives Page

The state table entries have so far been modelled as transitions which represent atomic events (Assumption 7.1). When an entry event is the submission of a service primitive by a TR-User, then the transition is named with that service primitive. When an entry action is the delivery of a service primitive to a TR-User, then the transition is given a label (in parenthesis) with that service primitive. This allows each primitive to be directly associated with an individual transition, and hence an arc in the state space. This is an important requirement when performing language analysis, because arcs in the state space are mapped to primitives in the TR-Protocol language. Further details on the mapping process are given in Chapter 8.

Entry 10 of the TR-Init-PE RESULT WAIT state table (Table B.2) is the only entry that cannot be modelled as a single transition. This is because it specifies two primitives (TR-Invoke.cnf and TR-Result.ind) to be delivered to the TR-Init-User. If this was modelled as a single transition, then the two primitives could not be differentiated when selecting arcs in the state space. Therefore, this action is decomposed into a sub-page (the only fourth level page) of transition RcvResult_Cnf (Figure 7.9) that contains a transition for each primitive. This sub-page, called I_RW_RcvResult_Cnf, is shown in Figure 7.19.

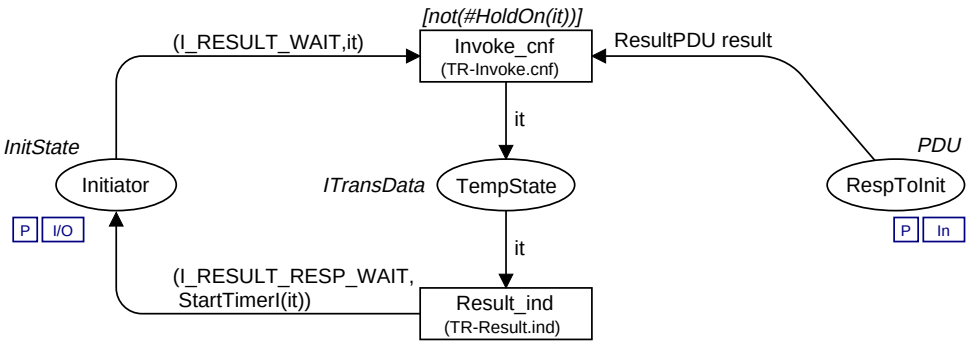

Figure 7.19: I_RW_RcvResult_Cnf page in the TR-Protocol CPN

On receipt of the Result PDU, a TR-Invoke.cnf primitive is delivered to the TR-Init-User (transition Invoke_Cnf). The transaction data taken from place Initiator is temporarily stored in place TempState. After Invoke_cnf occurs, there is no token in the place Initiator, therefore, disabling all transitions on the TR-Init-PE pages. This stops the TR-Init-PE from processing anything related to the state tables (although the TR-Resp-PE can still proceed). Transition Result_ind can occur, modelling the delivery of the TR-Result.ind primitive to the TR-Init-User. The output arc of this transition deposits a token in the Initiator place. This models the TR-Init-PE entering the I_RESULT_RESP_WAIT state with the acknowledgment timer started, allowing the TR-Init-PE to proceed as normal.

Chapter 8

Transaction Protocol Analysis

The Coloured Petri net in Chapter 7 models the Transaction Protocol described in the WTP Specification [183], with several small modifications. The CPN is now used as the basis for analysis. The state space of the Transaction Protocol (TR-Protocol) CPN can be calculated, allowing several properties of the TR-Protocol to be proved. The state space is also used to calculate the TR-Protocol language, the sequences of service primitives generated by the TR-Protocol. By comparing this language with the TR-Service language in Chapter 6, we can determine if the TR-Protocol provides the required service.

We divide the analysis of the TR-Protocol into two chapters. In this chapter we focus on the analysis of the TR-Protocol defined by the CPN in Chapter 7. This analysis reveals errors in the TR-Protocol to which we propose solutions. The set of modifications leads to a Revised TR-Protocol and corresponding CPN model. In Chapter 9 we analyse the Revised TR-Protocol, and show that it satisfies the desired properties.

The main purpose of the analysis is to verify that the TR-Protocol provides the TR-Service. State space analysis is also used to increase our confidence that the TR-Protocol operates correctly. Section 8.1 states the desired properties of the TR-Protocol CPN. The analysis environment is described in Section 8.2. Section 8.3 gives the parameter values chosen, and the corresponding state space and language results, for the configuration of the TR-Protocol that conveniently illustrates the errors found. The errors are described in Sections 8.4 to 8.6. For each error found, we propose a change to the TR-Protocol, and show how that change is reflected in our TR-Protocol CPN. Only a brief description of the changes to the TR-Protocol CPN is given, as the full Revised TR-Protocol CPN is shown in Appendix E.

8.1 Desired Properties of the Transaction Protocol

The main property of interest is that the TR-Protocol provides the TR-Service. That is, the sequences of service primitives that occur when the TR-Protocol is used, is identical

to the sequences of service primitives defined in the TR-Service. This requirement is stated in Property 8.1:

Property 8.1 (Refinement of TR-Service). *The TR-Protocol language must be identical to the TR-Service language.* □

Section 8.2 explains how the TR-Protocol language is obtained and compared with the TR-Service language given in Chapter 6.

The terminal markings of the TR-Protocol state space define the state of the TR-Protocol when a transaction is complete. Property 8.2 defines the required terminal markings.

Property 8.2 (Successful Termination). *The terminal markings of the TR-Protocol state space are dead markings of the form of the marking given in Table 8.1 or of the markings given in Table 8.2.* □

Place	*Marking*
UserAck	empty
First	1‘T
TempState	empty
Initiator	1‘(I_NULL,<default>)
Responder	1‘(R_LISTEN,<default>)
InitToResp	empty
RespToInit	empty

Table 8.1: Desired terminal marking indicating successful termination of the TR-Protocol when the original Invoke PDU has not been received by the TR-Resp-PE

Place	*Marking*
UserAck	empty
First	1‘F
TempState	empty
Initiator	1‘(I_NULL,<default>)
Responder	1‘(R_LISTEN,<default>)
InitToResp	<don’t care>
RespToInit	<don’t care>

Table 8.2: Desired set of terminal markings indicating successful termination of the TR-Protocol when the original Invoke PDU has been received by the TR-Resp-PE

The terminal marking in Table 8.1 is for the special case when the TR-Init-User submits a TR-Abort.req or the T-Service-Provider delivers a TR-Abort.ind (to the TR-Init-User) before the original Invoke PDU is sent by the local host of the TR-Init-PE (see Section 7.6.3). The set of terminal markings in Table 8.2 is for all cases when the Invoke PDU has been received by the TR-Resp-PE.

When Invoke_req on page I_NULL (Figure 7.8) occurs, the only token in UserAck is removed. No transitions deposit tokens in UserAck. Thus it will be empty on termination.

The place First (Figure 7.12) is initially marked with 1‘T indicating an Invoke PDU has not been received by the TR-Resp-PE. For the special case (Table 8.1) the marking of First must remain 1‘T. For the general case (Table 8.2) after an Invoke PDU is received, First will always be marked with 1‘F.

Both of the TR-PEs must return to their initial state upon completion of a transaction. The TR-Init-PE should be in the I_NULL state, and the place TempState (Figure 7.19) should be empty. The TR-Resp-PE must be in the R_LISTEN state. The transaction data of each TR-PE must be equal to the default values, which are specified in the initial markings (Figures 7.6 and 7.7).

For the special case (Table 8.1), there must be no tokens in the communication places, InitToResp and RespToInit (Figure 7.5). For the general case (Table 8.2), the marking of the communication places can take a range of values for the expected terminal markings. This is because we have modelled the TR-Protocol so that PDUs that are ignored during the transaction will remain in the communication channel. Section 7.6.1 discusses the justification for this modelling decision. Although we do not care about the terminal markings of the communication places, in Chapter 9 we measure and discuss the upper bounds on these places, i.e. the maximum number of PDUs that are in the communication channel.

It follows from Property 8.2 that any dead markings that are not terminal markings are undesirable, i.e. deadlocks.

As well as having no deadlocks, the TR-Protocol CPN should not be able to enter a sequence of occurrences that cannot end, i.e.:

Property 8.3 (Absence of Livelocks). *There must be no livelocks in the state space of the TR-Protocol.* □

As discussed in Chapter 4, if the state space is isomorphic to the SCC Graph and contains no self loops (which is true for the TR-Protocol, because there are no transitions that do not change the marking of the CPN), then it also has no livelocks. This property is used to prove there are no livelocks (in this chapter, and Chapter 9).

Dead transitions in the TR-Protocol state space may mean a state table entry is redundant or that unexpected behaviour has occurred, resulting in that entry not being utilized. However, we also expect transitions that model protocol features that are not activated for a particular set of initial parameter values to be dead.

Property 8.4 (Absence of Unexpected Dead Transitions). *There must be no dead transitions in the TR-Protocol state space, unless those transitions model features that are not activated because of the choice of initial parameters values (see Table 8.3).* □

The first four transitions listed in Table 8.3 model a time-out when either User-Ack is On or Off. Therefore, they will be dead when UserAck is Off (transitions 1

No.	*Condition*	*Dead Transition*	*Page*
1	UserAck Off	TimerTO_A_Max	I_RESULT_RESP_WAIT
2			R_INVOKE_RESP_WAIT
3	UserAck On	TimerTO_A_Off	I_RESULT_RESP_WAIT
4			R_INVOKE_RESP_WAIT
5		Result_req	R_INVOKE_RESP_WAIT
6	RCRRmax=0	TimerTO_R	R_RESULT_RESP_WAIT
7		RcvResult	I_WAIT_TIMEOUT
8	RCRImax=0	TimerTO_R	I_RESULT_WAIT
9		RcvAck_Tve	I_RESULT_WAIT
10		RcvAck	R_TIDOK_WAIT
11		RcvInvoke	R_TIDOK_WAIT
12			R_RESULT_WAIT
13	RCRImax≤1	TimerTO_R_Tve	I_RESULT_WAIT

Table 8.3: Conditions for a dead transition in the TR-Protocol CPN

and 2) or On (transitions 3 and 4), respectively. Likewise, transition Result_req on the R_INVOKE_RESP_WAIT page (Figure 7.14) can only occur when UserAck is Off.

When the maximum value of RCR is initially 0, the transitions that model the incrementing of this counter will be dead. These are transitions 6 and 8 (transition 7 cannot occur because it models the receipt of a re-transmitted Result PDU). Although the scenario of using an initial counter value of 0 is unlikely (the efficiency of the protocol will be poor), from the WTP Specification [183] it is possible.

Transitions 9 to 13 are all dead when RCRImax is 0. An Ack(Tok) (transition 9) or re-transmitted Invoke PDU (transition 8) cannot be sent when RCRImax is 0. Therefore, the TR-Resp-PE cannot receive these PDUs (transitions 10, 11 and 12). The TimerTO_R_Tve transition on page I_RESULT_WAIT can only occur after an Ack(Tok) has been sent, and when RCR<RCRImax. Therefore, it is dead when RCRImax=0, and also when RCRImax=1, since the sending of the original Ack(Tok) PDU increments RCR.

Proving these four properties of the TR-Protocol, and in particular Property 8.1, will provide a high level of confidence that the TR-Protocol operates correctly.

8.2 Analysis Parameters and Recording of Results

The analysis of the TR-Protocol involves selecting values for a set of parameters and recording the state space and language results. Section 8.2.1 lists the parameters of the TR-Protocol CPN. Section 8.2.2 describes the statistics recorded and Section 8.2.3 gives the setup of the hardware the analysis is performed on.

8.2.1 Parameters of the TR-Protocol CPN

The analysis results presented in this chapter relate to the TR-Protocol CPN described in Chapter 7. There are three parameters used in the CPN (the first two are from the

declarations in Listing 7.5):

1. RCRImax: The maximum value of the counter RCR at the TR-Init-PE.
2. RCRRmax: The maximum value of the counter RCR at the TR-Resp-PE.
3. UserAck: The initial marking of UserAck[1] (Figure 7.8). This indicates whether UserAck is Off (F) or On (T).

Hereafter, we will distinguish the parameters using italics. The values of the parameters define the *Configuration* of the current CPN model. For brevity, we may refer to a Configuration by the values the parameters take, ordered as in the above list, e.g. *Config 1-2-T*. As discussed in Chapter 4, a limitation of state space analysis is that it is dependent on the initial parameter values. For our TR-Protocol CPN, where there are an infinite number of configurations possible, the parameter values used must be chosen carefully so that there is a compromise between the number of configurations analysed and the level of confidence that can be obtained from the results. At a minimum, the parameter values used should test the basic features. For example, the first two parameters (the counters) test the re-transmission features of the TR-Protocol. The parameter *UserAck* tests the differences in behaviour when UserAck is On or Off. Section 8.3 discusses the selection of parameter values for analysis further.

8.2.2 State Space and Language Statistics

Design/CPN version 4.0.5 [161] was used to perform the state space analysis of the TR-Protocol CPN. Several standard properties [87] of the state space were calculated. Design/CPN conveniently writes these properties to a report. The statistics recorded for the state space include the number of nodes, arcs, dead markings and dead transitions (DT). The SCC Graph was also calculated and the number of nodes and arcs recorded. It is used to determine if any livelocks are present. A Standard ML query is written and executed in Design/CPN to determine if the dead markings are desired (terminal markings) or not (deadlocks). The query, given in Appendix D, returns true if all dead markings are of the form defined in Property 8.2.

The FSM libraries (see Chapter 4) are used to perform language analysis on the TR-Protocol CPN. Again, a number of statistics are recorded, including: the number of nodes, arcs and halt states in the FSA; the number of sequences in the language; the length of the longest and shortest sequences in the language; and the number of sequences in the TR-Protocol language, but not in the TR-Service language (NIS), and vice versa (NIP).

[1] We can omit references to the page that a place is on because the place names in the TR-Protocol CPN are unique.

8.2.3 Hardware and Software Setup

All analysis was performed on a computer with specifications as given in Table 8.4.

Item	*Value*
CPU	Intel Celeron
Clock Speed	366MHz
RAM	512MB
Operating System	Linux
OS version	Slackware 7
Design/CPN version	4.0.5
FSM version	3.6
LexTools version	3.0
GraphViz version	1.5

Table 8.4: Specification of hardware and software used for analysis

8.3 Configuration of the Transaction Protocol

8.3.1 General Approach to the Analysis

Our investigation of the TR-Protocol has consisted of calculating the state space and language for different configurations. The approach to selecting the configurations was to start with all counter parameters at 0, and then incrementing them. When it was discovered that the properties in Section 8.1 were not true for a particular configuration, similar configurations were analysed to see which parameters, if any, have an impact on the specific properties. From the inconsistencies identified in the languages and state space, and our understanding of the TR-Protocol, we suggested changes to the TR-Protocol and performed the analysis with the changes implemented in the TR-Protocol CPN. This process, which required a significant amount of ingenuity and effort, was continued until all properties of the TR-Protocol were true, and we were confident that the final set of changes made were correct. The selection of configurations to give us this confidence is discussed further in Chapter 9.

The remainder of this Chapter presents the errors found and the suggested changes. We use only one configuration, described in Section 8.3.2, which illustrates all of the errors. Chapter 9 presents the analysis results of the TR-Protocol with the suggested changes, proving the desired properties hold for the configurations analysed.

8.3.2 Parameter Values

The parameter values of the configuration analysed in the remainder of this chapter are shown in Table 8.5. We refer to the TR-Protocol with these values as Configuration 1. The counter parameter *RCRRmax* is set to 1 to allow a single re-transmission of

the Result PDU. *RCRImax* is 3 to allow for the scenario where one Invoke PDU is re-transmitted by the TR-Init-PE (e.g. RCR=1), and an Ack(Tok) PDU is sent (which increments RCR, i.e. RCR=2) and also re-transmitted (RCR=3). With UserAck Off, the TR-PEs may acknowledge PDUs without waiting for a response from the TR-Users.

Parameter	*Value*
RCRImax	3
RCRRmax	1
UserAck	F

Table 8.5: Parameter values for Configuration 1 of the TR-Protocol

Using the parameter values in Table 8.5, the initial marking of the TR-Protocol CPN is shown in Figure 8.1.

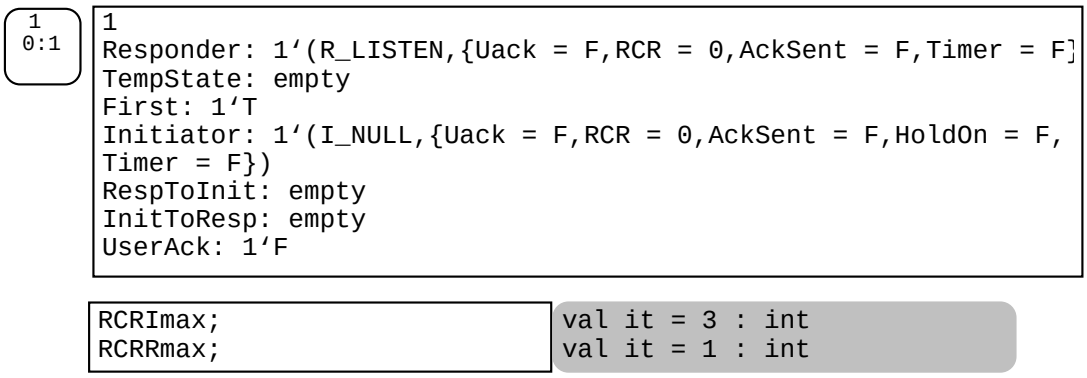

Figure 8.1: Initial marking of the TR-Protocol CPN in Configuration 1

8.3.3 State Space and Language Statistics

The state space and SCC Graph of Configuration 1 were calculated. The Design/CPN report and results of the query on the dead markings are given in Appendix D. Table 8.6 summarizes the statistics important for verifying the desired properties of the TR-Protocol.

Nodes	*Arcs*	*Terminal markings*	*Deadlocks*	*Livelocks*	*DT*
40386	182395	1884	0	0	2

Table 8.6: State space statistics of the TR-Protocol CPN in Configuration 1

The size of the state space is reasonable, in terms of Design/CPN processing time and memory consumption. Chapter 9 provides further discussion on the size of the state space in relation to the parameter values.

The query on the dead markings (Listing D.3) shows all 1884 are desired terminal markings (the result of the query is shown in Figure D.1). There are no livelocks in the TR-Protocol, since the size of the SCC Graph is equal to the size of the state space (i.e.

the number of nodes of each graph are identical, and the number of arcs for each graph are identical). The two dead transitions (see Listing D.2) are expected, as they are due to the UserAck On feature not being activated in Configuration 1 (transitions 1 and 2 in Table 8.3). Therefore, from the state space statistics measured, there is no indication of errors in the TR-Protocol (Properties 8.2, 8.3 and 8.4 are true).

To perform language analysis, binding elements of the TR-Protocol CPN are mapped to service primitives, and the state space is treated as a FSA. Chapter 7 described the correspondence of binding elements to service primitives. The functions that perform the mapping, and convert dead markings to halt states, are given in Appendix D. Table 8.7 shows the statistics of the minimized FSA and language of the TR-Protocol.

Nodes	*Arcs*	*Halts*	*Sequences*	*Long*	*Short*	*NIS*	*NIP*
61	278	6	59562	14	2	59406	26

Table 8.7: FSA and language statistics of the TR-Protocol CPN in Configuration 1

Table 8.7 shows that there are 59406 sequences in the TR-Protocol language that are not in the TR-Service language. These are illegal TR-Protocol primitive sequences. There are also 26 sequences in the TR-Service language, but not the TR-Protocol language. The following sections show that these 26 sequences are due to the same errors leading to the illegal sequences (NIS). Although we haven't examined all illegal sequences individually, from our inspection of some of the sequences and our experience from analysing other configurations, several *awk* scripts [43] have been written that categorize the sequences. The scripts, and results of their execution, are given in Appendix D. In summary, they show that:

- 57944 of the 59406 illegal primitive sequences in the TR-Protocol language contain two TR-Invoke.ind primitives;
- 1368 of the remaining 1462 sequences contain two TR-Invoke.cnf primitives;
- 92 of the remaining 94 sequences contain a TR-Invoke.cnf primitive, but no TR-Invoke.res primitive; and
- the remaining 2 sequences contain a TR-Result.cnf primitive, but no TR-Result.res primitive.

The analysis has involved investigating the TR-Protocol more closely (including inspecting specific parts of the state spaces), to see where and how these illegal sequences are generated. Sections 8.4, 8.5 and 8.6 describe the errors in the TR-Protocol that we have found, and suggest changes to the TR-Protocol (and corresponding CPN).

8.4 Ambiguous Ack and Result PDUs

The first error evident from the analysis of the TR-Protocol is the possibility of PDUs received by a TR-PE to be interpreted as an acknowledgment from the peer *TR-User*, when only the peer *TR-PE* has acknowledged. The illegal sequences in the TR-Protocol language that contain two TR-Invoke.cnf primitives lead to the identification of this error. Section 8.4.1 describes the error, using an occurrence sequence from the state space as an example. Section 8.4.2 gives the suggested changes to fix the error, and Section 8.4.3 briefly describes the corresponding changes to the CPN (a Revised TR-Protocol CPN that includes all suggested changes is given in Appendix E). This partition into three sub-sections will be continued in Sections 8.5 and 8.6.

8.4.1 Description and Example of the Error

Figure 8.2 shows a sequence possible in Configuration 1 where two TR-Invoke.cnf primitives are delivered to the TR-Init-User. A time sequence diagram of this occurrence sequence is given in Figure 8.3.

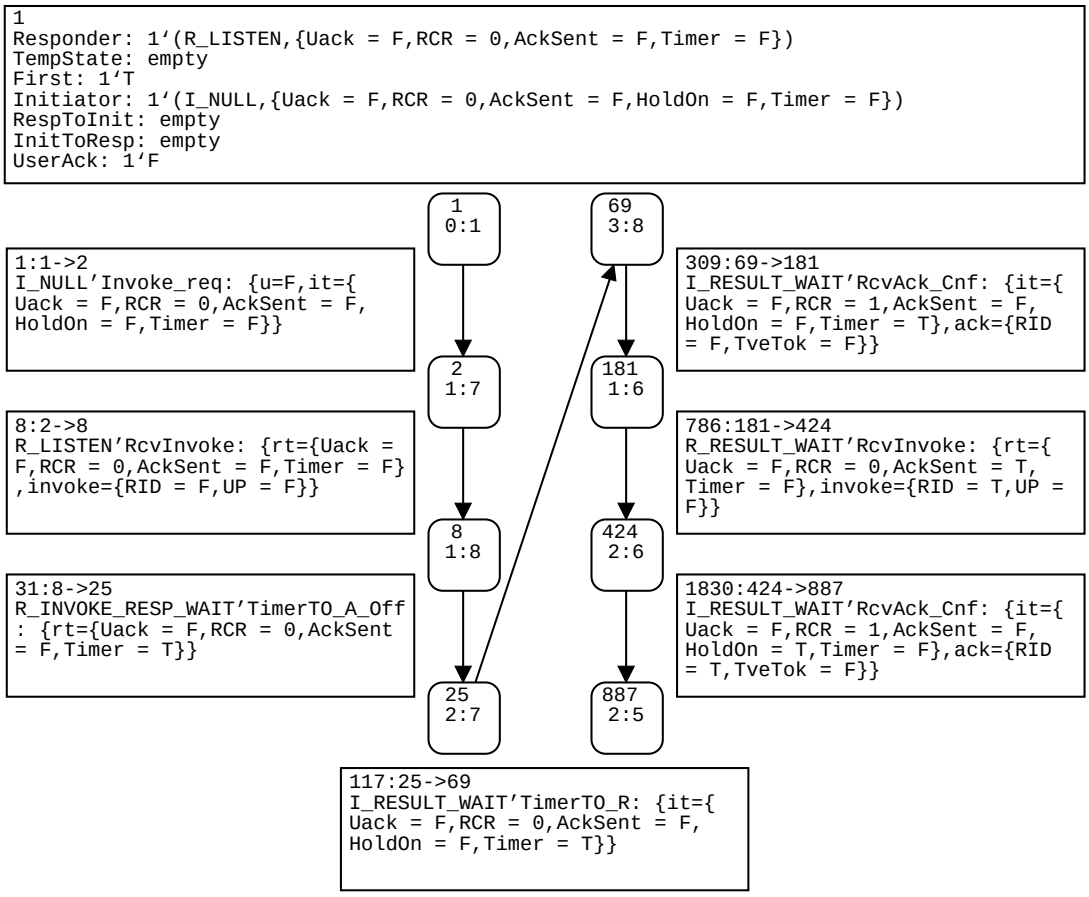

Figure 8.2: Path in the TR-Protocol (Configuration 1) state space showing two TR-Invoke.cnf primitives

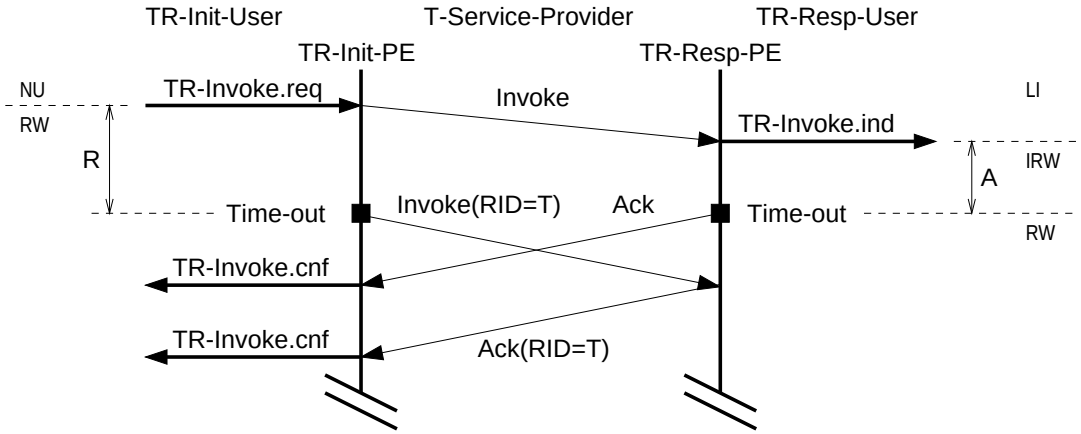

Figure 8.3: Time sequence diagram of the TR-Protocol in Configuration 1 showing two TR-Invoke.cnf primitives

The scenario shown by Figures 8.2 and 8.3 starts with the TR-Init-PE in the NULL (NU) state, and the TR-Resp-PE in the LISTEN (LI) state. A transaction is started by the TR-Init-User submitting the TR-Invoke.req primitive. The TR-Init-PE sends an Invoke PDU, whose receipt by the TR-Resp-PE triggers the delivery of the TR-Invoke.ind primitive to the TR-Resp-User. The TR-Resp-PE waits for a response from the TR-Resp-User in the INVOKE RESP WAIT (IRW) state after delivering the TR-Invoke.ind primitive. As the response is not received within the interval A, the TR-Resp-PE sends an Ack PDU to indicate to the TR-Init-PE that the Invoke PDU has been received. Following Figure 8.3, the TR-Init-PE has already re-transmitted the Invoke PDU due to the expiration of the re-transmission timer. Upon receipt of the re-transmitted Invoke PDU, the TR-Resp-PE re-transmits the Ack PDU to recover from the situation where, for example, the first Ack PDU was lost due to an error. However, the first Ack PDU is not lost, and the TR-Init-PE receives two Ack PDUs, triggering the delivery of two TR-Invoke.cnf primitives to the TR-Init-User. This is inconsistent with the TR-Service, where only one TR-Invoke.cnf primitive is delivered to the TR-Init-User.

We presented the error of delivering two TR-Invoke.cnf primitives to the TR-Init-User in [56]. The solution suggested was to ignore the second (or any subsequent) Ack PDUs received by the TR-Init-PE. However, the TR-Service language that we compared the TR-Protocol language with in [56] did not require end-to-end behaviour of the TR-Invoke and TR-Result primitives (e.g. Assumption 6.2 in Chapter 6). For the TR-Service presented in Chapter 6, Figure 8.3 reveals another error, where the first TR-Invoke.cnf primitive is delivered to the TR-Init-User without the TR-Resp-User previously having submitted the TR-Invoke.res primitive. This is the result of ambiguous semantics of the Ack PDU, which is explained shortly. Section 8.4.2 shows that re-defining the semantics of the Ack PDU will also solve the problem of two TR-Invoke.cnf primitives being delivered to the TR-Init-User. We will see shortly that the Result PDU sent by the TR-Resp-PE is also

ambiguous. A similar error can occur when the TR-Init-PE acknowledges the Result PDU. We first consider the acknowledgment of the Invoke PDU.

After the TR-Resp-PE has delivered the TR-Invoke.ind primitive to the TR-Resp-User, it waits for a response in the INVOKE RESP WAIT state. If no response is received before the Acknowledgment timer expires, then the TR-Resp-PE sends an Ack PDU and stops the timer (Entry 9 in Table B.7). The response from the TR-Resp-User may be the submission of a TR-Invoke.res primitive (Entry 1, Table B.7) or TR-Result.req primitive (Entry 2, Table B.7). The TR-Invoke.res primitive can only be submitted before the time-out occurs. The TR-Resp-PE re-starts the timer when TR-Invoke.res is received. The receipt of the TR-Result.req primitive causes the TR-Resp-PE to send the Result PDU. There are four different scenarios possible, which are shown in Figure 8.4. The actions of the TR-Init-PE shown are based on the current TR-Protocol.

If we ignore the actions taken by the TR-Init-PE on receipt of the PDUs, from the scenarios shown in Figure 8.4 we obtain the following semantics for the PDUs:

> **Figure 8.4(a):** The Result PDU indicates to the TR-Init-PE: the invocation has been received by the TR-Resp-User (TR-Invoke.ind), and the TR-Resp-User has returned the result (TR-Result.req).
>
> **Figure 8.4(b):** The Ack PDU indicates to the TR-Init-PE: the invocation has been received by the TR-Resp-User (TR-Invoke.ind). The Result PDU indicates to the TR-Init-PE: the TR-Resp-User has returned the result (TR-Result.req).
>
> **Figure 8.4(c):** The Ack PDU indicates to the TR-Init-PE: the invocation has been received by the TR-Resp-User (TR-Invoke.ind), and the TR-Resp-User has acknowledged the invocation (TR-Invoke.res). The Result PDU indicates to the TR-Init-PE: the TR-Resp-User has returned the result (TR-Result.req).
>
> **Figure 8.4(d):** The Result PDU indicates to the TR-Init-PE: the invocation has been received by the TR-Resp-User (TR-Invoke.ind), the TR-Resp-User has acknowledged the invocation (TR-Invoke.res), and the TR-Resp-User has returned the result (TR-Result.req).

These four different scenarios give rise to ambiguous semantics of the Ack PDU and Result PDU:

- Does the TR-Init-PE interpret the Ack PDU as an acknowledgment from the TR-Resp-User (in which case a TR-Invoke.cnf primitive should be delivered to the TR-Init-User) or just from the TR-Resp-PE?
- If an Ack PDU has not been received, does the TR-Init-PE interpret the Result PDU as an acknowledgment of the invocation from the TR-Resp-User (in which

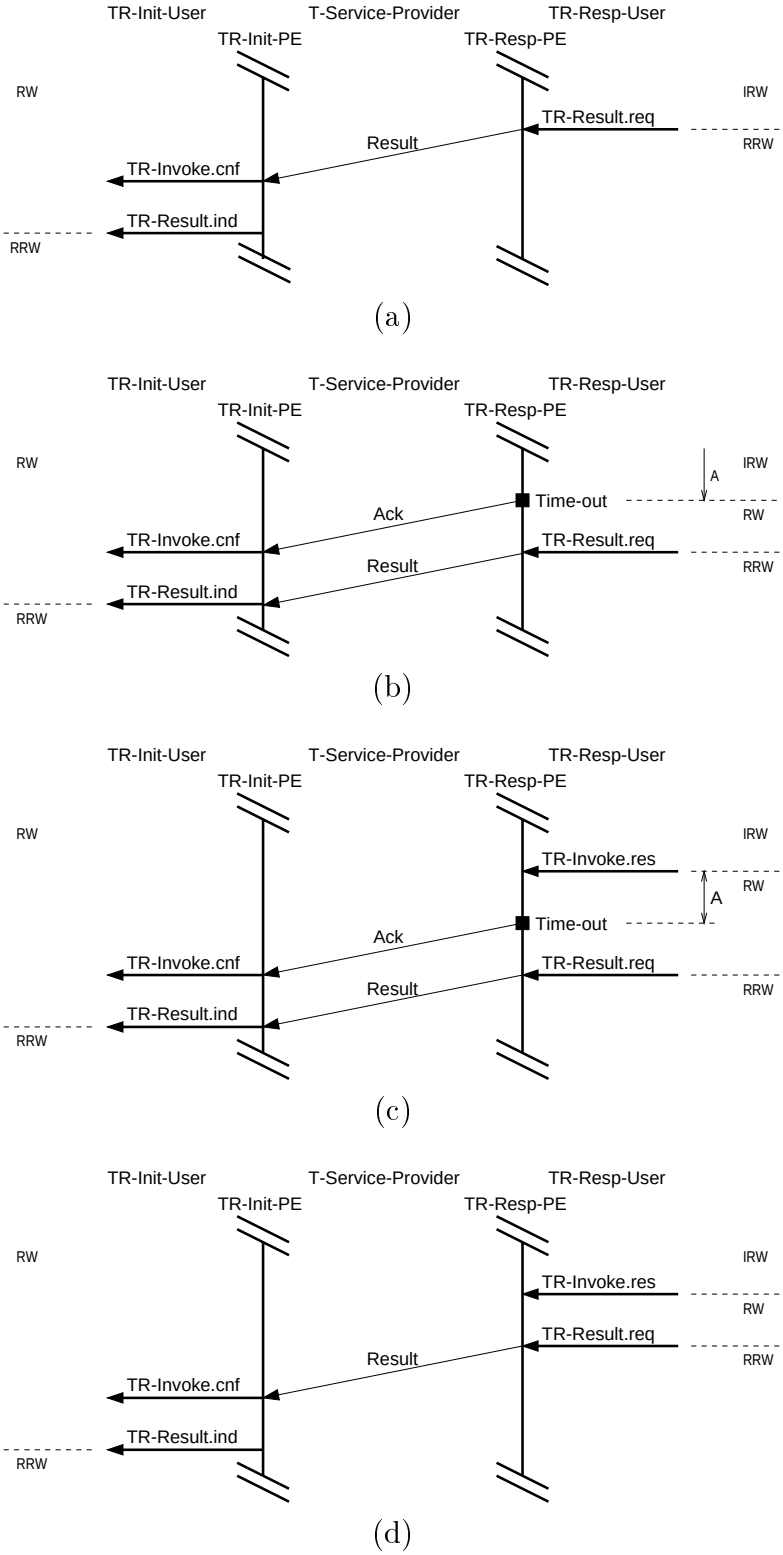

Figure 8.4: Acknowledgment of the Invoke PDU leads to illegal primitive sequences at the TR-Init-User when UserAck is Off. In (a) the TR-Invoke.cnf primitive is delivered without the TR-Invoke.res being submitted by the TR-Resp-User. This is the same for (b). The PDUs received are misinterpreted. In (c) and (d) the PDUs are correctly interpreted (i.e. the TR-Invoke.cnf primitive should be delivered).

case a TR-Invoke.cnf and TR-Result.ind primitive should be delivered to the TR-Init-User) or just as the result (only TR-Result.ind primitive delivered to the TR-Resp-User)?

There is a similar problem for the Ack PDU when used to acknowledge the Result PDU. A time-out by the TR-Init-PE while waiting for a response from the TR-Init-User (TR-Result.res) triggers the Ack PDU to be sent to the TR-Resp-PE (Entry 9, Table B.3). If the TR-Result.res is submitted before the time-out, then the same Ack PDU is sent (Entry 1/2, Table B.3). Does the TR-Resp-PE interpret the Ack PDU as an acknowledgment from the TR-Init-User (in which case a TR-Result.cnf primitive should be delivered to the TR-Resp-User) or just from the TR-Init-PE?

8.4.2 Suggested Changes to the TR-Protocol

To overcome these ambiguities, the purpose of each PDU must be defined so the receiving TR-PE can process it correctly. This can be achieved by using a new flag, called `Confirmed` or `CNF`, in each of the PDUs' headers. We first consider the case of the TR-Resp-PE sending PDUs to the TR-Init-PE. For the acknowledgment (or confirmation) of the Invoke PDU, if the `CNF` flag is set, then the PDUs indicate to the TR-Init-PE that the TR-Resp-User has acknowledged the invocation (with a TR-Invoke.res primitive). Therefore, in Figure 8.4(c) the Ack PDU would have its `CNF` flag set. In Figure 8.4(d) the Result PDU would have its `CNF` flag set. A new variable (of type boolean) is required at each TR-PE so it is known if the TR-Invoke.res primitive has been submitted (TR-Resp-PE) or the TR-Invoke.cnf primitive has been delivered (TR-Init-PE). We call this variable `Ucnf` for User confirm. Figure 8.5 shows the scenarios of Figure 8.4 updated to differentiate between the semantics of the Ack and Result PDUs.

An alternative solution to the ambiguous semantics of the Ack and Result PDU may be to define new PDUs. However, adding a single bit flag requires only a small change to the current TR-Protocol. For the Ack PDU the Reserved bit (`RES`) could be used for the flag. The Result PDU would require a fourth octet, of which one bit would be the `CNF` flag. The seven other bits would be reserved. Figures 8.6 and 8.7 show the new header structures for the Ack and Result PDUs, respectively.

Another alternative solution, in the UserAck Off case, could be to prevent the TR-Resp-User from submitting a TR-Invoke.res primitive. Then there would be no need to differentiate the PDUs at the TR-Init-PE because a TR-Invoke.cnf primitive must not be delivered to the TR-Init-User. Allowing the TR-Invoke.res primitive is however a better solution as it is more general.

The `CNF` flag is also used for the acknowledgment of the Result PDU. If set, the Ack PDU is indicating to the TR-Resp-PE that the TR-Init-User has submitted the TR-

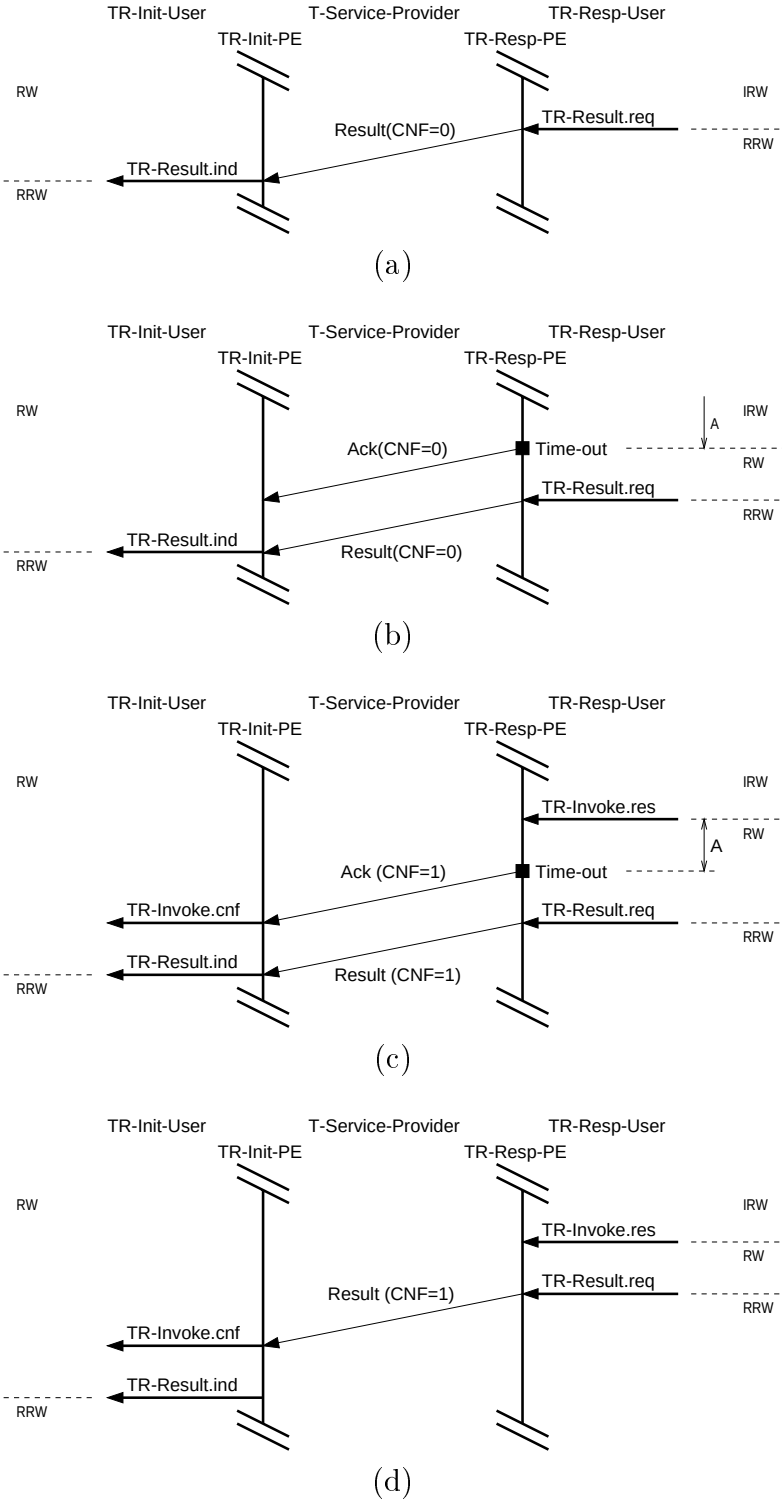

Figure 8.5: Adding a CNF bit to the Ack and Result PDUs enables the TR-Init-PE to correctly interpret the semantics of the PDU when UserAck is Off. It is assumed a TR-Invoke.ind primitive has just been delivered to the TR-Resp-User. (a) The Result PDU with `CNF`=0 indicates that the TR-Resp-User has submitted the TR-Result.req primitive, but not the TR-Invoke.res primitive. (b) The Ack PDU with `CNF`=0 indicates the TR-Resp-PE has received the Invoke PDU. (c) The Ack PDU with `CNF`=1 indicates the TR-Resp-User has submitted the TR-Invoke.res primitive. (d) The Result PDU with `CNF`=1 indicates the TR-Resp-User has submitted both the TR-Invoke.res and TR-Result.req primitives.

Bit/Octet	0	1	2	3	4	5	6	7
1	CON	PDU Type = Ack				Tve_Tok	CNF	RID
2 3	TID							

Figure 8.6: New header structure for Ack PDU including `CNF` bit

Bit/Octet	0	1	2	3	4	5	6	7
1	CON	PDU Type = Result				GTR	TTR	RID
2 3	TID							
4	CNF	RES						

Figure 8.7: New header structure for Result PDU including `CNF` bit

Result.res primitive. If the `CNF` flag is not set, then the Ack PDU is only indicating the TR-Init-PE has received the result.

Tables 8.8 to 8.13 show the updated state table entries to incorporate the CNF field in the TR-Protocol. To simplify the changes, we introduce the following conventions:

- The CNF field in the Ack PDU and Result PDU is set to the value specified in the Action column of the state tables. If no value is specified, the CNF field is 0.
- Additional text to the state tables is shown in *italics*.
- Text to be deleted from the state tables is indicated with a line through it.

	Event	Condition	Action	Next State
2	RcvAck	Class == 2 *CNF==1* *Ucnf==False*	Stop timer Generate TR-Invoke.cnf *Ucnf=True* ~~HoldOn=True~~	RESULT WAIT
2a		*Class == 2*	*Stop timer*	*RESULT WAIT*
9	RcvResult	Class == 2 ~~HoldOn==True~~	Stop timer Generate TR-Result.ind *Ucnf=False* Start timer, A	RESULT RESP WAIT
10		Class == 2 *CNF==1* *Ucnf==False* ~~HoldOn==False~~	Stop timer Generate TR-Invoke.cnf Generate TR-Result.ind *Ucnf=False* Start timer, A	

Table 8.8: Entries 2, 2a, 9 and 10 of the TR-Init-PE RESULT WAIT state table (Table B.2) modified to remove PDU ambiguities

The TR-Init-PE enters the RESULT WAIT state (Table 8.8) after sending the initial Invoke PDU. When an Ack PDU is received the timer is stopped (Entries 2 and 2a, Table 8.8). If the `CNF` field in the PDU is 1 and the TR-Invoke.cnf primitive hasn't already been delivered to the TR-Init-User, then it is now delivered (Entry 2, Table 8.8).

	Event	Condition	Action	Next State
1	TR-Result.res		Queue(A) Ack PDU *(CNF=1)* *Ucnf=True* Start timer, W	WAIT TIMEOUT
2		ExitInfo	Queue(A) Ack PDU*(CNF=1) with Info TPI* *Ucnf=True* Start timer, W	

Table 8.9: Entries 1 and 2 of the TR-Init-PE RESULT RESP WAIT state table (Table B.3) modified to remove PDU ambiguities

	Event	Condition	Action	Next State
2	RcvResult	RID=1	Send Ack PDU *(CNF=Ucnf)*	WAIT TIMEOUT
3	RcvResult	RID=1,ExitInfo	Send Ack PDU *(CNF=Ucnf) with info TPI*	WAIT TIMEOUT

Table 8.10: Entries 2 and 3 of the TR-Init-PE WAIT TIMEOUT state table (Table B.4) modified to remove PDU ambiguities

The same applies upon the receipt of the Result PDU (Entries 9 and 10, Table 8.8), except now `Ucnf` is set to `False` because it is now used to indicate if the TR-Init-User has confirmed (in the RESULT RESP WAIT state). The `HoldOn` variable is no longer needed.

The variable `Ucnf` at the TR-Init-PE is re-used for the acknowledgment of the Result PDU (Tables 8.9 and 8.10). It is set to 1 if the TR-Result.res primitive has been submitted (Entries 1 and 2, Table 8.9). If the TR-Init-PE has to re-transmit the Ack PDU, then it uses `Ucnf` to determine whether to set the `CNF` flag in the header (Entries 2 and 3, Table 8.10). When entering the RESULT RESP WAIT state from the RESULT WAIT state, the TR-Init-PE must ensure `Ucnf` is reset to 0 (Entries 9 and 10, Table 8.8).

	Event	Condition	Action	Next State
1	TR-Invoke.res	Class == 2	*Ucnf=True* Start timer, A	RESULT WAIT
2	TR-Result.req		Reset RCR Start timer, R[RCR] Send Result PDU *(CNF=0)*	RESULT RESP WAIT
9	TimerTO_A	Class == 2 Uack == False	Send Ack PDU *(CNF=0)*	RESULT WAIT

Table 8.11: Entry 1 of the TR-Resp-PE INVOKE RESP WAIT state table (Table B.7) modified to remove PDU ambiguities

`Ucnf` is initially set to `False` at the TR-Resp-PE. In the INVOKE RESP WAIT state, `Ucnf` is set to `True` if the TR-Resp-User submits a TR-Invoke.res primitive (Entry 1, Table 8.11). Otherwise, it remains `False`. Note that the TR-Resp-PE enters the RESULT WAIT state after the submission of the TR-Invoke.res primitive. Therefore, the events resulting in the sending of the Ack PDU or Result PDU in the INVOKE RESP WAIT state (Entries 2 and 9, Table 8.11) always set the `CNF` field in the PDUs to 0 (TR-Invoke.res hasn't been submitted). In the RESULT WAIT state, the `CNF` field in the PDUs is set to the value of `Ucnf` (Entries 1, 4 and 8, Table 8.12). Similarly, in the

	Event	Condition	Action	Next State
1	TR-Result.req		Reset RCR Start timer, R[RCR] Send Result PDU *(CNF=Ucnf)*	RESULT RESP WAIT
4	RcvInvoke	RID=1, Ack PDU already sent	Resend Ack PDU *(CNF=Ucnf)*	RESULT WAIT
8	TimerTO_A		Send Ack PDU *(CNF=Ucnf)*	RESULT WAIT

Table 8.12: Entries 1, 4 and 8 of the TR-Resp-PE RESULT WAIT state table (Table B.8) modified to remove PDU ambiguities

	Event	Condition	Action	Next State
3	RcvAck	*CNF==1*	Generate TR-Result.cnf	LISTEN
3a	*RcvAck*	*CNF==0*		*LISTEN*
5	TimerTO_R	RCR< RCR_MAX	Increment RCR Send Result PDU *(CNF=Ucnf)* Start timer, R[RCR]	RESULT RESP WAIT

Table 8.13: Entries 3, 3a and 5 of the TR-Resp-PE RESULT RESP WAIT state table (Table B.9) modified to remove PDU ambiguities

RESULT RESP WAIT state, the `CNF` field in the re-transmitted Result PDU is set to `Ucnf` (Entry 5, Table 8.13). Also in the RESULT RESP WAIT state, the TR-Resp-PE delivers a TR-Result.cnf primitive to the TR-Resp-User only if the `CNF` flag is set in the received Ack PDU (Entries 3 and 3a, Table 8.13).

8.4.3 Changes to the TR-Protocol CPN

The following are a set of changes to the TR-Protocol CPN (Chapter 7) to reflect the changes to the TR-Protocol in Section 8.4.2. The changes are only summarized as the complete Revised TR-Protocol CPN is given in Appendix E. The modified arc inscriptions and guards are shown in bold, while new transitions have thicker edges.

- The colour sets ResultPDU_c and AckPDU_c are modified to include the `CNF` flag. (Note that ResultPDU_c is now a record.)
- The `Ucnf` flag is added to the transaction data colour sets, ITransData and RTransData. Also, the flag HoldOn is removed from ITransData.
- The functions in the declarations are updated to reflect the changes to ITransData and RTransData.
- The Ack and Result PDUs on output arcs are updated to include the value of the CNF field in the record.
- The inscriptions of the output arcs from the following transitions to the Initiator or Responder place are updated to set Ucnf to T: RcvAck_Cnf on the page

I_RESULT_WAIT, Result_res on the page I_RESULT_RESP_WAIT, and Invoke_res on the page R_INVOKE_RESP_WAIT.

- The inscriptions of the output arcs from the following transitions to the Initiator or Responder place are updated to reset Ucnf to F: RcvResult_Cnf on page I_RESULT_WAIT, RcvResult on page I_RESULT_WAIT, and Result_ind on the page I_RW_RcvResult_Cnf.

- The guards of RcvAck_Cnf, RcvResult (both on page I_RESULT_WAIT), Invoke_cnf (page I_RW_RcvResult_Cnf) and RcvAck_Cnf (page R_RESULT_RESP_WAIT) are updated to include the conditions testing Ucnf and CNF.

- A new transition, called RcvAck, is added to the page I_RESULT_WAIT. This models the receipt of an Ack PDU that doesn't result in a TR-Invoke.cnf primitive being delivered to the TR-Init-User.

- A new transition, called RcvAck, is added to the page R_RESULT_RESP_WAIT. This models the receipt of an Ack PDU that doesn't result in a TR-Result.cnf primitive being delivered to the TR-Resp-User.

The desired properties must be updated to reflect these changes to the TR-Protocol. The transition RcvAck on the I_RESULT_WAIT page will be dead when *UserAck* is On and $RCRImax$=0, because:

- the TR-Resp-PE cannot send an Ack PDU with `CNF` set to `F` (the TR-Resp-User must confirm when *UserAck* is On), and

- `Ucnf` at the TR-Init-PE is only set to true after the first Ack PDU (`CNF=1`) sent by the TR-Resp-PE is received. The TR-Resp-PE can only send a second Ack PDU (`CNF=1`) upon receipt of a re-transmitted Invoke PDU, which isn't possible when $RCRImax$=0.

The transition RcvAck on the R_RESULT_RESP_WAIT page will be dead when *UserAck* is On, because in that case, an Ack PDU can only be sent by the TR-Init-PE after a TR-Result.res primitive has been submitted by the TR-Init-User.

Table 8.3 has two additional entries, as shown in Table 8.14, to include these new dead transitions.

8.5 Erroneous Re-start of the Transaction

The second error evident from the analysis of the TR-Protocol is the possibility for the TR-Resp-User to be delivered two TR-Invoke.ind primitives, within the context of one

No.	*Condition*	*Dead Transition*	*Page*
14	UserAck On ∧ RCRImax=0	RcvAck	I_RESULT_WAIT
15	UserAck On	RcvAck	R_RESULT_RESP_WAIT

Table 8.14: Conditions 14 & 15 for an expected dead transition in the TR-Protocol CPN (see Table 8.3)

transaction initiated by the TR-Init-User (i.e. only one TR-Invoke.req primitive submitted). This is in conflict with the TR-Service where the TR-Invoke primitives must be end-to-end (Assumption 6.2).

8.5.1 Description and Example of the Error

As we discussed for the TR-Service (Chapter 6), if the TR-Resp-User submits a TR-Abort.req primitive, then, from its point of view, the transaction is complete. The receipt of a re-transmitted (or duplicated) Invoke PDU after the TR-Abort.req must not be accepted. Figures 8.8 and 8.9 show an occurrence sequence and its TSD, respectively, where two TR-Invoke.ind primitives are delivered to the TR-Resp-User. After the TR-Resp-User is notified of the transaction (TR-Invoke.ind), it aborts (TR-Abort.req), and the TR-Resp-PE re-enters the LISTEN state. Note that the Abort PDU sent may be delayed, or even lost. Figure 8.9 shows the TR-Init-PE re-transmitting the Invoke PDU, due to a time-out on the re-transmit timer. The receipt of this re-transmitted Invoke PDU by the TR-Resp-PE initiates TID verification because the next TID value is not expected. The TR-Init-PE responds positively to the query, indicating to the TR-Resp-PE to proceed with the transaction. The receipt of the Ack(Tok) PDU by the TR-Resp-PE triggers a second TR-Invoke.ind primitive to be delivered to the TR-Resp-User.

8.5.2 Suggested Changes to the TR-Protocol

We modify the TR-Protocol so the TID verification is not initiated immediately after the transaction is aborted. This means the abort will end the transaction, and no more primitives are allowed. However, a TID verification *should* be possible for new incarnations of the TID and, therefore, the TID verification is not allowed for a certain time after the transaction is aborted (denoted by the interval W). A new incarnation of a TID is a result of the TID space wrapping and the TID being re-used. This will be discussed in more detail shortly. The changes to the TR-Protocol can be summarized as:

- A new timer interval, `W`, is used by the TR-Resp-PE.
- A new variable of type boolean, called `AbortSent`, is used by the TR-Resp-PE.
- Modifications are made to the TR-Resp-PE state tables (Tables 8.15 to 8.19) so that when the TR-Resp-User or TR-Resp-PE aborts a transaction, the timer is started

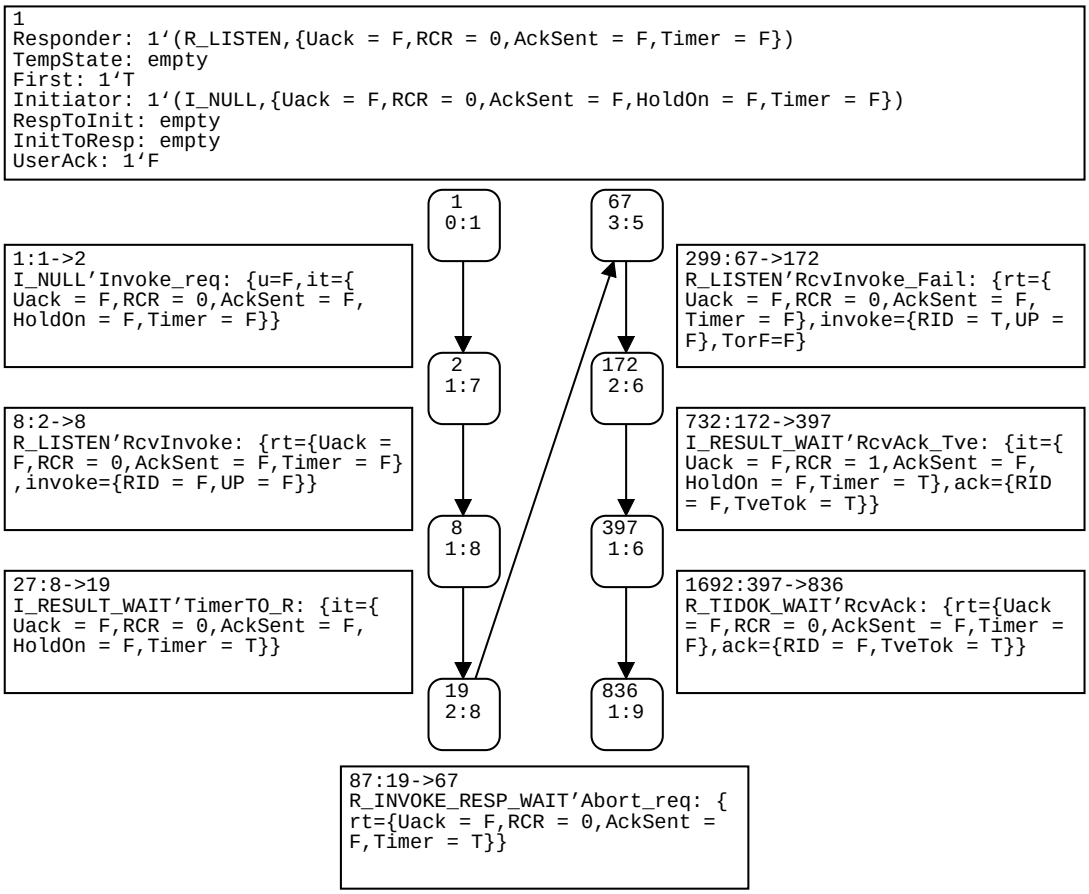

Figure 8.8: Path in the TR-Protocol (Configuration 1) state space showing two TR-Invoke.ind primitives

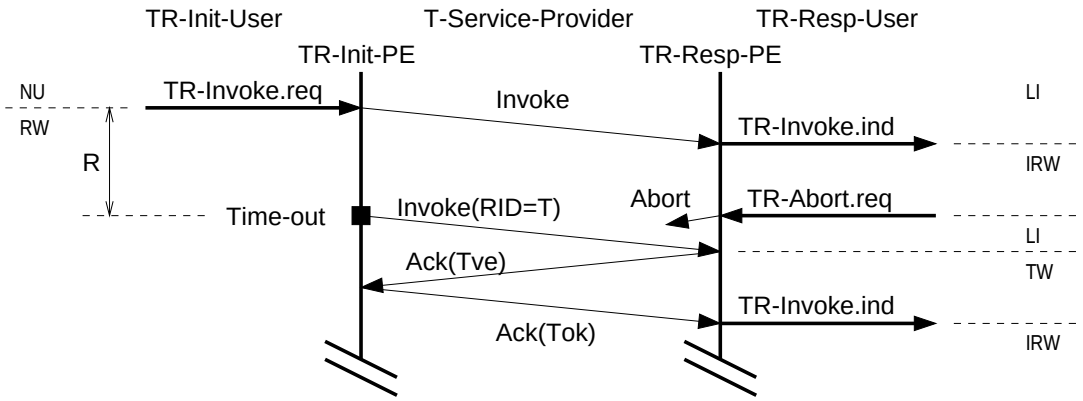

Figure 8.9: Time sequence diagram of the TR-Protocol in Configuration 1 showing two TR-Invoke.ind primitives

with interval `W`. If an Abort PDU has also been sent, then the flag `AbortSent` is set.

- Three new entries are added to the TR-Resp-PE LISTEN state table (Table 8.15):

  1. Upon receipt of an unexpected Invoke PDU, if a TID is not allowed and an Abort PDU has not been sent, then simply ignore the Invoke PDU (Entry 3a).
  2. Upon receipt of an unexpected Invoke PDU (i.e. incorrect TID), if a TID verification is not allowed (timer with interval `W` is on) and an Abort PDU has been sent, then re-send the Abort PDU (Entry 3b).
  3. When the timer with interval `W` expires, a TID verification is now possible (Entry 3c).

	Event	**Condition**	**Action**	**Next State**
3	RcvInvoke	Class == 2 | 1 Invalid TID *Timer Off*	Send Ack(TIDve)	TIDOK WAIT
3a		*Class == 2 | 1* *Invalid TID* *Timer On*	*Ignore*	*LISTEN*
3b		*Class == 2 | 1* *Invalid TID* *Timer On* *AbortSent==True*	*Send Abort PDU (USER)*	*LISTEN*
3c	*TimerTO_W*		*Stop timer*	*LISTEN*
4	RcvErrorPDU		Send Abort PDU (PROTOERR) *Set AbortSent* *Start timer, W*	LISTEN

Table 8.15: Entries 3, 3a, 3b, 3c and 4 of the TR-Resp-PE LISTEN state table (Table B.5) modified to fix the TID verification error

	Event	**Condition**	**Action**	**Next State**
2	RcvErrorPDU		Send Abort PDU (PROTOERR) Abort transaction *Set AbortSent* *Start timer, W*	LISTEN

Table 8.16: Entry 2 of the TR-Resp-PE TIDOK WAIT state table (Table B.6) modified to fix the TID verification error

When the TR-Resp-PE begins, the timer is off. Therefore, the new state table entries in Table 8.15 do not apply. A transaction can proceed as in the initial TR-Protocol configuration (e.g. Entry 1/2, Table B.5). When a transaction is aborted by the TR-Resp-User or as a result of a time-out by the TR-Resp-PE, the TR-Resp-PE enters the LISTEN state. The modified state table entries in Tables 8.17, 8.18 and 8.19 indicate that the timer is started, with interval `W`. The interval `W` will be discussed shortly. Also,

	Event	Condition	Action	Next State
3	TR-Abort.req		Abort transaction Send Abort PDU (USER) *Set AbortSent* *Start timer, W*	LISTEN
6	RcvErrorPDU		Abort transaction Send Abort PDU (PROTOERR) *Set AbortSent* *Start timer, W* Generate TR-Abort.ind	LISTEN
8	TimerTO_A	AEC == AEC_MAX	Abort transaction Send Abort PDU (NORESPONSE) *Set AbortSent* *Start timer, W*	LISTEN

Table 8.17: Entries 3, 6 and 8 of the TR-Resp-PE INVOKE RESP WAIT state table (Table B.7) modified to fix the TID verification error

	Event	Condition	Action	Next State
5	RcvErrorPDU		Abort transaction Send Abort PDU (PROTOERR) *Set AbortSent* *Start timer, W* Generate TR-Abort.ind	LISTEN
6	TR-Abort.req		Abort transaction Send Abort PDU (USER) *Set AbortSent* *Start timer, W*	LISTEN

Table 8.18: Entries 5 and 6 of the TR-Resp-PE RESULT WAIT state table (Table B.8) modified to fix the TID verification error

	Event	Condition	Action	Next State
1	TR-Abort.req		Abort transaction Send Abort PDU (USER) *Set AbortSent* *Start timer, W*	LISTEN
4	RcvErrorPDU		Abort transaction Send Abort PDU (PROTOERR) *Set AbortSent* *Start timer, W* Generate TR-Abort.ind	LISTEN
6	TimerTO_R	RCR == RCR_MAX	Generate TR-Abort.ind Abort transaction *Start timer, W*	LISTEN

Table 8.19: Entries 1, 4 and 6 of the TR-Resp-PE RESULT RESP WAIT state table (Table B.9) modified to fix the TID verification error

for all cases except the time-out with `RCR` at the maximum in the RESULT RESP WAIT state (Entry 6, Table 8.19), the flag `AbortSent` is set since an Abort PDU has been sent to the TR-Init-PE.

The timer is used in the LISTEN state to determine if TID verification can follow an abort initiated by the TR-Resp-PE or TR-Resp-User. If we assume the transaction aborted had a TID of x, then when the timer is on, the receipt of an Invoke PDU with TID x cannot initiate TID verification. Instead, if an Abort PDU has been sent, then it is re-transmitted and the TR-Resp-PE remains in the LISTEN state (Entry 3b, Table 8.15). If an Abort PDU was not sent, then the Invoke PDU is ignored (Entry 3a, Table 8.15).

When the timer with interval `W` expires, the timer is stopped, and the TR-Resp-PE remains in the LISTEN state. With the timer off, the receipt of an Invoke PDU with TID x (which is an unexpected TID), initiates TID verification. This allows TID verifications to be initiated, if necessary, when transactions with new incarnations of the TID x are created. Figure 8.10 shows an example.

The TR-Init-User initiates a transaction with TID=0. After being delivered a TR-Invoke.ind primitive, the TR-Resp-User submits a TR-Abort.req primitive. The TR-Resp-PE enters the LISTEN state and starts the timer with interval W. The TR-Resp-PE is now expecting an Invoke PDU with TID=1. As the Abort PDU has not been received by the TR-Init-PE, it re-transmits the Invoke PDU. Upon receipt of the Invoke PDU with TID=0, the TR-Resp-PE does not initiate TID verification because the transaction has just been aborted. Instead, the TR-Resp-PE re-sends the Abort PDU.

The TR-Init-User may initiate a new transaction with TID=1. The TR-Resp-PE accepts the Invoke PDU and proceeds, as normal, with the transaction (for clarity, only the Invoke and Result PDUs are shown—the Ack PDU is omitted).

If the TR-Init-User initiates transactions with a different TR-Resp-User (B), the TID used by the TR-Init-PE will be increased, but the TID expected by the original TR-Resp-User (A) will remain the same. Therefore, after executing 32765 transactions with a different TR-Resp-User (B), the TR-Init-User can then initiate a transaction with TID=32767 with the original TR-Resp-User (A). As a TID of 2 was expected, TID verification is initiated and eventually the transaction is successfully completed.

The next transaction initiated by the TR-Init-User has TID = 0. This is a new incarnation of this TID value, because the TID space has wrapped. The TR-Resp-PE can treat the Invoke PDU as a new transaction, not a re-transmission or duplicate. This is because the TR-Init-PE can only initiate 2^{14} transactions within two maximum packet lifetimes (2MPL) (see Section 5.3.1 and Section 7.2.1). Assuming MPL is significantly larger than the processing time for the transaction plus the time to send a PDU, the PDUs of an old incarnation (e.g. the first transaction with TID=0) will not be present in the network, and the transaction completed. Therefore, rather than immediately rejecting

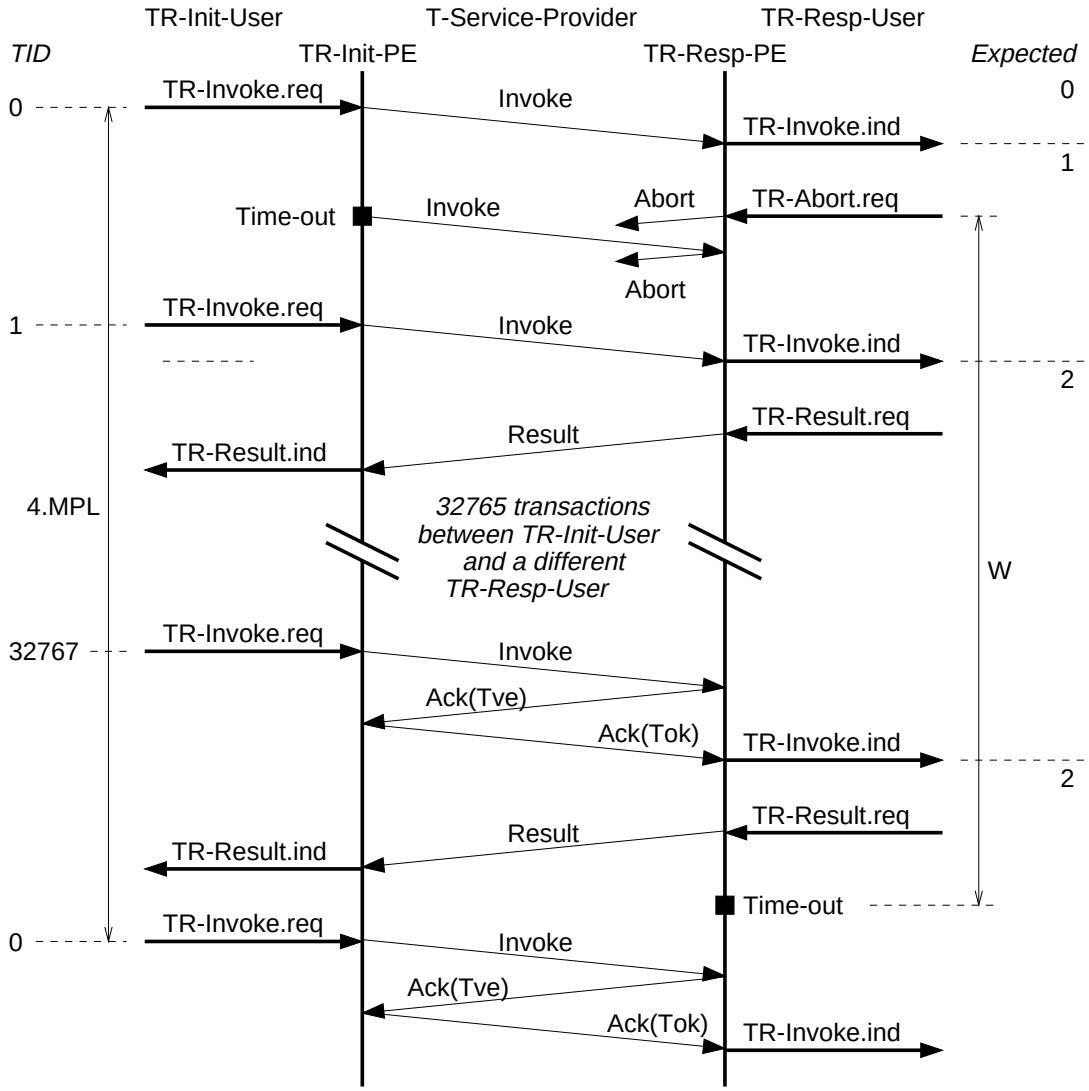

Figure 8.10: A TID verification can only occur W seconds after a previous incarnation has been aborted for the TR-Protocol.

the Invoke PDU, TID verification should be performed. As the timer with interval W has expired, TID verification can be performed on receipt of the Invoke PDU. Then the transaction can proceed, as normal.

The changes to the TR-Protocol specified in Tables 8.15 to 8.19 do not allow TID verification to be performed before the time-out (for a new incarnation of a TID value). If the value of the interval W is less than 4MPL, then TID verification will be possible for the new incarnation of TID x. If W is greater than 4MPL, then the Invoke PDU with the new incarnation of the TID x may be unnecessarily rejected. This could reduce efficiency of the TR-Protocol. The minimum value of W should be based on the time it takes for a transaction to be completed, and all of the associated PDUs to be cleared from the network. 2MPL may be a reasonable value for this [19]. Further investigation into the optimal values for W is needed.

8.5.3 Changes to the TR-Protocol CPN

The TR-Protocol CPN can be updated to reflect the changes to the TR-Protocol in the following way:

- A new transition, called RcvInvoke_Abo, is added to the page R_LISTEN. This models the receipt of an unexpected Invoke PDU while TID verification is not allowed and an Abort PDU has been sent (Entry 3b, Table 8.15). Entry 3c of Table 8.15 is not modelled because this only ignores the Invoke PDU.

- The guards for transitions RcvInvoke and RcvInvoke_Fail on page R_LISTEN are updated to include the condition #Timer(rt)=F.

- The inscriptions on the output arcs for the following transitions to the place Responder are updated to set the AbortSent flag and start the timer with interval W: TimerTO_A_Max and Abort_req on page R_INVOKE_RESP_WAIT, Abort_req on page R_RESULT_WAIT and Abort_req on page R_RESULT_RESP_WAIT.

- The inscription on the output arc from the following transitions to the place Responder are updated to start the timer with interval W: TimerTO_R_Max on page R_RESULT_RESP_WAIT and ProviderAbort on page R_ABORT.

Note that the RcvErrorPDU state table entries are not modelled in the TR-Protocol CPN, and therefore there are no changes required to reflect modifications to these entries. Also, the time-out on timer with interval W (Entry 3c, Table 8.15) is not modelled because after the time-out the transaction must have been completed and no PDUs can be received by the TR-Resp-PE (this is ensured because all PDUs are discarded after

2MPL). Therefore, nothing can occur after the time-out (effectively, the interval W is infinite).

Again, the desired properties must be updated to reflect these changes to the TR-Protocol. When $RCRImax$=0, the transition RcvInvoke_Abo on the R_LISTEN page will be dead because it models the receipt of a re-transmitted Invoke PDU. Table 8.3 has an additional entry, as shown in Table 8.20, to include the new dead transition.

No.	*Condition*	*Dead Transition*	*Page*
16	RCRImax=0	RcvInvoke_Abo	R_LISTEN

Table 8.20: Condition 16 for an expected dead transition in the TR-Protocol CPN (see Tables 8.3 and 8.14)

The desired terminal markings (Property 8.2) must now take into account the new possible states of the TR-Resp-PE upon termination. That is, in the general case (Table 8.2) the marking of the place Responder may not return to its default values. The Timer should be T, and AbortSent may be either T or F in a successful terminal marking.

8.6 Misinterpreted Ack(Tok) PDU

In Section 8.5 we saw that a second TR-Invoke.ind primitive can be erroneously delivered to the TR-Resp-User as a result of a re-transmitted Invoke PDU. A timer was used to restrict the TID verification (and hence, TR-Invoke.ind) after an aborted transaction at the TR-Resp-PE. However, a re-transmitted Invoke PDU can still result in an error, if the Ack(Tok) PDU is re-transmitted.

8.6.1 Description and Example of the Error

It is possible for the TR-Init-PE to re-transmit an Invoke PDU and subsequently re-transmit an Ack(Tok) PDU. If these PDUs are delayed, then the TR-Resp-PE may misinterpret the Ack(Tok) PDU as a positive acknowledgment of a TID verification, when the TR-Init-PE has no outstanding transaction. Figure 8.11 shows one occurrence sequence that leads to this error. A TSD of this sequence is shown in Figure 8.12.

Figure 8.12 shows a transaction beginning which immediately requires TID verification. The TR-Resp-PE sends an Ack(Tve) PDU, and on receipt of the Ack(Tok) PDU it delivers the TR-Invoke.ind primitive to the TR-Resp-User. In the meantime, two time-outs have occurred at the TR-Init-PE, one before the receipt of the Ack(Tve) PDU (triggering the re-transmission of the Invoke PDU), and one after its receipt (triggering the re-transmission of the Ack(Tok) PDU). These two re-transmitted PDUs are delayed in the network. The TR-Init-User submits a TR-Abort.req primitive, and upon receipt

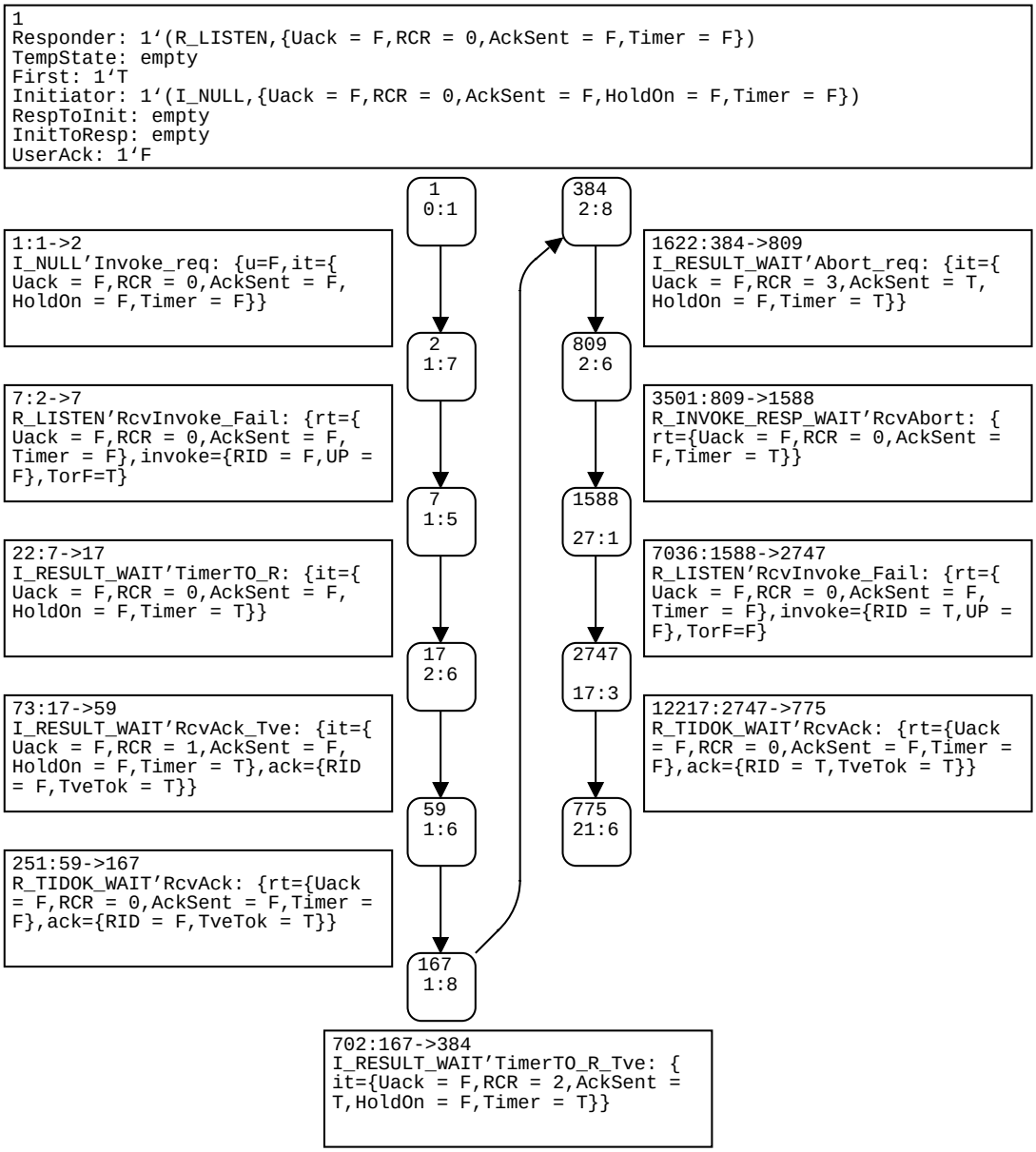

Figure 8.11: Path in the TR-Protocol state space (Configuration 1) showing the TR-Resp-PE misinterpret the Ack(Tok) PDU as acknowledging the Result PDU

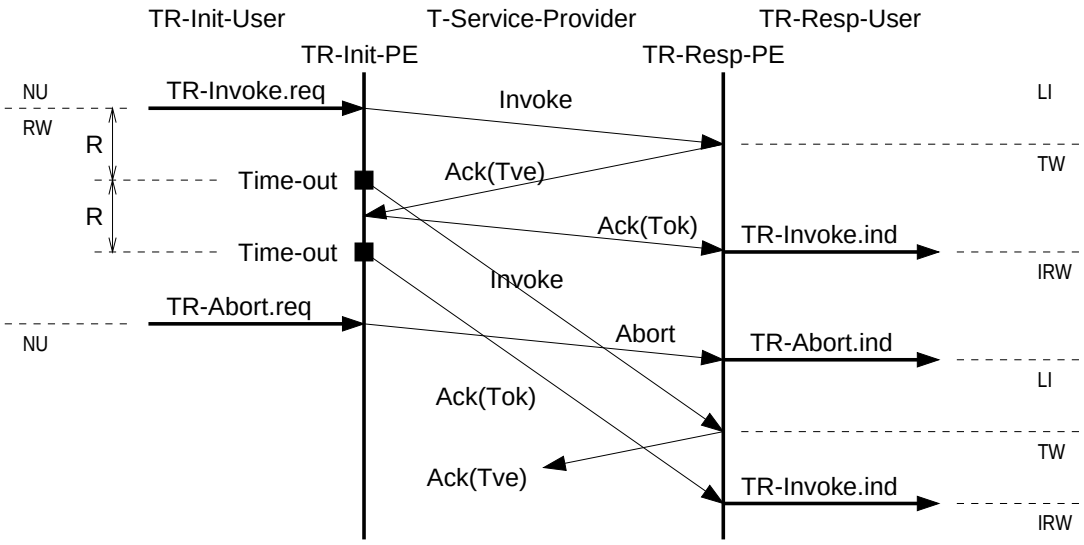

Figure 8.12: Time sequence diagram of the TR-Protocol (Configuration 1) showing the TR-Resp-PE misinterpret the Ack(Tok) PDU as acknowledging the Result PDU

of the Abort PDU, the TR-Resp-PE delivers a TR-Abort.ind primitive to the TR-Resp-User. The transaction is complete at both TR-Init-PE and TR-Resp-PE. However, upon receipt of the re-transmitted Invoke PDU the TR-Resp-PE initiates TID verification (note that the TID verification is not restricted by the changes suggested in Section 8.5 in this case because the transaction was not aborted by the TR-Resp-PE or TR-Resp-User). The re-transmitted Ack(Tok) PDU can be received by the TR-Resp-PE, which interprets it as a positive acknowledgment (i.e. proceed with this transaction). However, the TR-Init-PE does not have an outstanding transaction—the TID verification should have resulted in the transaction being aborted.

This error is similar to that identified in Section 8.5 where a second transaction is started at the TR-Resp-PE as a result of a re-transmitted Invoke PDU. In that case however, the success of the TID verification is correct (the TR-Init-PE did have an outstanding transaction). For this error the TID verification is incorrect.

8.6.2 Suggested Changes to the TR-Protocol

The proposed fix to this problem is to further restrict when TID verification can be initiated. In Section 8.5.2 the interval W was introduced to restrict the initiation of TID verification after a transaction has been aborted by the TR-Resp-PE or TR-Resp-User. We extend this restriction to apply after a transaction is completed by any means. This includes a successful transaction and a transaction aborted by the TR-Init-PE or TR-Init-User. In summary, after a transaction with TID x has been completed (i.e. the TR-Resp-PE returns to the LISTEN state), TID verification cannot be initiated for TID x until after the interval W.

Whenever the TR-Resp-PE enters the LISTEN state, the timer must be started with interval W. The changes to the state tables comprise the addition of actions to those entries entering the LISTEN state. They are shown in Tables 8.21 to 8.24.

	Event	Condition	Action	Next State
3	RcvAbort		Abort transaction *Start timer, W*	LISTEN

Table 8.21: Entry 3 of the TR-Resp-PE LISTEN state table (Table B.5) modified to restrict a TID verification so an Ack(Tok) PDU is not misinterpreted

	Event	Condition	Action	Next State
4	RcvAbort		Generate TR-Abort.ind *Start timer, W* Abort transaction	LISTEN

Table 8.22: Entry 4 of the TR-Resp-PE INVOKE RESP WAIT state table (Table B.7) modified to restrict a TID verification so an Ack(Tok) PDU is not misinterpreted

	Event	Condition	Action	Next State
7	RcvAbort		Generate TR-Abort.ind *Start timer, W* Abort transaction	LISTEN

Table 8.23: Entry 7 of the TR-Resp-PE RESULT WAIT state table (Table B.8) modified to restrict a TID verification so an Ack(Tok) PDU is not misinterpreted

8.6.3 Changes to the TR-Protocol CPN

The TR-Protocol CPN can be updated to reflect the changes to the TR-Protocol by:

- Modifying the inscriptions on the output arcs for the following transitions to the place Responder to start the timer with interval W: RcvAbort on the page R_TIDOK_WAIT; RcvAbort on the page R_INVOKE_RESP_WAIT; RcvAbort on the page R_RESULT_WAIT; and RcvAck, RcvAbort and RcvAck_Cnf on the page R_RESULT_RESP_WAIT.

8.7 Summary

The TR-Protocol CPN in Chapter 7 is our initial model of the TR-Protocol, with several small modifications. In this chapter we have defined the desired properties of the TR-Protocol, and then analysed the CPN. The analysis has revealed errors in the TR-Protocol, and we have suggested changes to remove the errors. The set of modifications to the initial TR-Protocol CPN lead to a Revised TR-Protocol CPN, which is analysed in Chapter 9.

	Event	Condition	Action	Next State
2	RcvAbort		Generate TR-Abort.ind *Start timer, W* Abort transaction	LISTEN
3	RcvAck	CNF==1	Generate TR-Result.cnf *Start timer, W*	LISTEN
3a	RcvAck	CNF==0	*Start timer, W*	LISTEN

Table 8.24: Entries 2, 3 and 3a of the TR-Resp-PE RESULT RESP WAIT state table (Table B.9) modified to restrict a TID verification so an Ack(Tok) PDU is not misinterpreted

The most important desired property of the TR-Protocol is that it provides the intended service described in Chapter 6. This chapter has shown several cases where comparing the TR-Protocol language with the TR-Service language has identified errors in the TR-Protocol.

The three errors identified and fixed in this chapter (Sections 8.4, 8.5 and 8.6, respectively) are:

1. The semantics of the Ack PDU and Result PDU are ambiguous, in that they can be interpreted by the receiving TR-PE as either an acknowledgment from the peer TR-User (i.e. a response primitive has been submitted) or only an acknowledgment from the peer TR-PE (no response primitive submitted). Adding a new field to each of the PDUs to indicate if the response primitive has been submitted can solve this problem.

2. The TR-Resp-User can continue a transaction (i.e. receive a second TR-Invoke.ind primitive) after aborting the transaction. Once aborted, a transaction should be complete. This error can be fixed by disallowing the acceptance of unexpected Invoke PDUs until a given time after the transaction has been aborted.

3. A re-transmitted Ack(Tok) PDU can erroneously acknowledge TID verification when the TR-Init-PE has no outstanding transaction. The solution to this problem involves extending the previous solution so that an unexpected Invoke PDU cannot be accepted until a given time after a transaction is ended by any means.

The intent of the suggested solutions is to improve the existing TR-Protocol so unexpected behaviour is eliminated, but at the same time, minimize the changes since implementations are already in use. Chapter 9 shows that the solutions do fix the problems, and do not introduce any other errors.

Chapter 9

Verification of the Revised Transaction Protocol

A Coloured Petri net model of the Transaction Protocol, with a set of changes to solve the errors described in Chapter 8, is presented in Appendix E. The analysis of the Revised TR-Protocol is the topic of this chapter. The intent is to show the changes suggested do indeed fix the problems found in the TR-Protocol, and that the Revised TR-Protocol is functionally correct, for a given set of parameter values.

The analysis of the Revised TR-Protocol involves using state space and language analysis to prove the desired properties described in Chapter 8. The properties are: successful termination; absence of deadlocks, livelocks and unexpected dead transitions; and, the most important property, the faithful refinement of the TR-Service of Chapter 6. These properties must be proved for different sets of parameter values, or configurations, of the Revised TR-Protocol. Section 9.1 discusses the selection of the values. Section 9.2 discusses the analysis results obtained, and proves the desired properties. Section 9.3 presents results from using the sweep-line analysis method [31], an approach to tackle the state explosion problem. Properties are proved for configurations "out of reach" of ordinary state space analysis. Section 9.4 uses the analysis results of the preceding sections to conjecture about proving properties of the Revised TR-Protocol, independently of particular parameter values. Section 9.5 closes this chapter with a discussion of results obtained from analysing the Revised TR-Protocol.

Professor Jonathan Billington provided useful feedback on the results of this chapter. The expression of Equation 9.4 as a polynomial instead of a summation is due to Professor Billington. This also applies for Equation 9.5, although the final form is given in terms of factorials.

9.1 Selection of Parameter Values

The Revised TR-Protocol has three parameters: the maximum values of the two counters (*RCRImax* and *RCRRmax*), and the value of User Acknowledgment (*UserAck*) (see Section 8.2.1). In theory, the two counter parameters are unbounded. The *UserAck* parameter can only take the values of true (`T`) or false (`F`). Therefore, we must select reasonable values for the counter parameters, so that the state space can be calculated within memory limits, and a reasonable number of configurations are required to make observations about the general behaviour. From the configurations analysed, the goal is to inductively prove that the properties hold, so that the results are independent of particular parameter values. As this is a complex task, we are also interested in proving the properties for configurations likely to be used in practice. The WTP Specification gives several suggested values for the counter parameters: *RCRImax* and *RCRRmax* both 8 for bearers supporting IP; and 4 in GSM SMS networks [183].

Our first goal is to verify the Revised TR-Protocol for all configurations up to and including that suggested for GSM SMS networks. That is, all permutations of the counter parameters within the range of 0 to 4, for both UserAck On and Off, are considered. This gives $2.5^2 = 50$ configurations. Proving properties for these simple configurations increases our confidence that the Revised TR-Protocol operates correctly. However, this approach of simply increasing the counter parameters is not sufficient. The number of configurations quickly becomes unmanageable, and as we see in Section 9.2, higher values of the counter parameters exhaust the memory available when calculating the state space.

To investigate the TR-Protocol in more complex scenarios, properties must be able to be verified with some independence of the parameter values. This independence may be achieved once the exact impact the parameters have on the Revised TR-Protocol is known. From the 50 configurations chosen, a trend in the size of the state space with respect to the parameter *RCRRmax* is observed. For the parameter *RCRImax*, our intuition tells us that a relationship may be found if *RCRImax* is increased to higher values (up to 10). However, the state space size becomes too large to calculate with these higher values. Therefore, we employ the sweep-line method [31] to alleviate the state explosion problem. Section 9.3 describes the analysis results using the sweep-line method for these configurations with larger state space sizes. Section 9.4 then discusses how the state space sizes relate to the parameter values.

9.2 State Space and Language Analysis

Design/CPN [109] and FSM [4] are used to calculate the state space and language analysis results for the Revised TR-Protocol CPN. The configurations analysed are those with all

permutations of the counter parameters within the range of 0 to 4, for both UserAck On and Off, except Configuration 4-4-F (recall from Chapter 8 that the Configuration is given in the order of *RCRImax*, *RCRRmax* then *UserAck*). Results for this configuration are unobtainable on the computer used (further comments on the limits of the tool are given in Appendix G). Also analysed are Configurations 5-0-T, 6-0-T and 7-0-T, the relevance of which is discussed in Section 9.4.

Standard ML and shell scripts were written to automate the process of calculating the state spaces of the set of configurations, minimizing the FSA and recording the relevant statistics. Appendix F describes how this is achieved. In summary, a place and transition (given in Figure F.1) are added to the CPN which are used to instantiate the parameter values. This has no impact on the analysis results except to create one more node and arc in the state space.

This section gives the results of the analysis, and discusses how they are used to prove the desired properties of the Revised TR-Protocol. Section 9.2.5 also discusses the bounds on the communication places.

9.2.1 Language Equivalence

Once the state space of a configuration was calculated (see Section 9.2.2 for the statistics), it was treated as a FSA, on which language analysis was performed. Listings E.3 and E.4 show the functions for mapping the state space to a FSA. As stated in Chapter 8, showing that the TR-Protocol faithfully refines the sequences of events in the TR-Service is the main purpose of the analysis.

Table 9.1 summarizes the statistics for the FSA and language calculated for all configurations (the complete set of results is given in Table E.1). Recall from Section 6.3.2 the number of sequences in the TR-Service language when UserAck is Off is 182 and when UserAck is On is 130. Table 9.1 shows every configuration with *UserAck*=T has identical minimized FSAs and languages. The FSA has 19 nodes, 61 arcs and 2 halt states, and the Revised TR-Protocol language contains 130 possible sequences of primitives, the longest being 8 and the shortest 2 primitives. Comparison with the TR-Service language (Chapter 6) reveals the two languages are identical (i.e. *NIS*=0 and *NIP*=0). When *UserAck*=F, all configurations have identical FSAs (19 nodes, 63 arcs and 4 halt states) and languages (182 sequences, longest is 8 and shortest is 2). Again, the Revised TR-Protocol language is identical to the TR-Service language.

UserAck	*Nodes*	*Arcs*	*Halts*	*Sequences*	*Longest*	*Shortest*	*NIS*	*NIP*
On	19	61	2	130	8	2	0	0
Off	19	63	4	182	8	2	0	0

Table 9.1: Statistics for both Revised TR-Protocol FSAs and languages (cf. Table 6.6)

The language analysis shows that for all configurations analysed, the Revised TR-Protocol faithfully refines the TR-Service, in terms of global sequences of service primitives, proving Property 8.1.

9.2.2 Terminal Markings and Deadlocks

Table 9.2 gives the statistics of the state space size, in terms of number of nodes (N), number of arcs (A) and calculation time in seconds (T), for each configuration. Note that a new place and transition is added to the CPN model to facilitate the batch analysis (see Appendix F). Therefore, the number of nodes and number of arcs are one more than if each were analysed individually. This does not affect the applicability of the results. Further discussion on the relationship between the state space size and parameter values is given in Section 9.4. Appendix G categorizes the performance of the state space analysis, as a guide for other users of Design/CPN.

RCRmax		*UserAck=T*					*UserAck=F*				
I	*R*	*N*	*A*	*T*	*TM*	*DL*	*N*	*A*	*T*	*TM*	*DL*
0	0	105	304	0	28	0	198	606	1	55	0
0	1	234	700	1	68	0	542	1702	2	167	0
0	2	403	1232	1	124	0	1046	3342	3	343	0
0	3	612	1900	2	196	0	1710	5526	5	583	0
0	4	861	2704	3	284	0	2534	8254	8	887	0
1	0	513	1677	2	78	0	1009	3430	3	163	0
1	1	1180	3992	4	188	0	2817	9898	10	477	0
1	2	2061	7133	7	342	0	5481	19670	22	967	0
1	3	3156	11100	11	540	0	9001	32746	39	1633	0
1	4	4465	15893	16	782	0	13377	49126	62	2475	0
2	0	1872	6528	6	219	0	3796	13664	15	465	0
2	1	4407	15824	17	541	0	10762	39888	54	1389	0
2	2	7778	28502	33	995	0	21072	79640	130	2841	0
2	3	11985	44562	55	1581	0	34726	132920	259	4821	0
2	4	17028	64004	84	2299	0	51724	199728	446	7329	0
3	0	6032	21865	25	541	0	12545	46591	67	1178	0
3	1	14487	53614	76	1363	0	36023	136708	307	3566	0
3	2	25798	97025	162	2533	0	70925	273473	887	7346	0
3	3	39965	152098	297	4051	0	117251	456886	1961	12518	0
3	4	56988	218833	488	5917	0	175001	686947	3682	19082	0
4	0	17024	63597	99	1181	0	36165	137474	317	2630	0
4	1	41507	157145	348	3021	0	104777	404209	1799	8036	0
4	2	74450	285317	859	5661	0	207229	809440	6004	16642	0
4	3	115853	448113	1723	9101	0	343521	1353167	16242	28448	0
4	4	165716	645533	3022	13341	0	-	-	-	-	-
5	0	42561	163366	378	2315	0	-	-	-	-	-
6	0	96070	378206	1388	4178	0	-	-	-	-	-
7	0	199338	803527	5080	7056	0	-	-	-	-	-

Table 9.2: Statistics on the state space size, terminal markings and deadlocks for the TR-Protocol configurations

Also included in Table 9.2 is the number of terminal markings (TM) and the number of deadlocks (DL). A Standard ML query was applied to each dead marking calculated by

Design/CPN to determine if the marking was desired (terminal marking) or not (deadlock). The query, given in Section E.2, returns true if all dead markings are of the form defined in Property 8.2. As Table 9.2 shows, all configurations of the Revised TR-Protocol analysed, terminated successfully, and contained no deadlocks, therefore proving Property 8.2.

9.2.3 Livelocks

Table 9.3 lists the statistics for the SCC Graph (number of nodes (N) and arcs (A)) of each configuration. (Also included are the bounds on the communication places, discussed in Section 9.2.5.) A comparison with Table 9.2 shows that, for every configuration the size of the state space is equivalent to the size of the SCC Graph. Also, from inspection of the CPN, the Revised TR-Protocol state space contains no self loops. This means there are no cycles in the state space and, therefore, no livelocks (column LL in Table 9.3 lists the number of livelocks). Property 8.3 has been proved for all analysed configurations of the Revised TR-Protocol.

RCRmax		*UserAck=T*					*UserAck=F*				
I	*R*	*N*	*A*	*LL*	*I2R*	*R2I*	*N*	*A*	*LL*	*I2R*	*R2I*
0	0	105	304	0	2	3	198	606	0	2	3
0	1	234	700	0	2	4	542	1702	0	2	4
0	2	403	1232	0	3	5	1046	3342	0	3	5
0	3	612	1900	0	4	6	1710	5526	0	4	6
0	4	861	2704	0	5	7	2534	8254	0	5	7
1	0	513	1677	0	3	4	1009	3430	0	3	4
1	1	1180	3992	0	3	5	2817	9898	0	3	5
1	2	2061	7133	0	4	6	5481	19670	0	4	6
1	3	3156	11100	0	5	7	9001	32746	0	5	7
1	4	4465	15893	0	6	8	13377	49126	0	6	8
2	0	1872	6528	0	4	5	3796	13664	0	4	5
2	1	4407	15824	0	4	6	10762	39888	0	4	6
2	2	7778	28502	0	5	7	21072	79640	0	5	7
2	3	11985	44562	0	6	8	34726	132920	0	6	8
2	4	17028	64004	0	7	9	51724	199728	0	7	9
3	0	6032	21865	0	5	6	12545	46591	0	5	6
3	1	14487	53614	0	5	7	36023	136708	0	5	7
3	2	25798	97025	0	6	8	70925	273473	0	6	8
3	3	39965	152098	0	7	9	117251	456886	0	7	9
3	4	56988	218833	0	8	10	175001	686947	0	8	10
4	0	17024	63597	0	6	7	36165	137474	0	6	7
4	1	41507	157145	0	6	8	104777	404209	0	6	8
4	2	74450	285317	0	7	9	207229	809440	0	7	9
4	3	115853	448113	0	8	10	343521	1353167	0	8	10
4	4	165716	645533	0	9	11	-	-	-	-	-
5	0	42561	163366	0	7	8	-	-	-	-	-
6	0	96070	378206	0	8	9	-	-	-	-	-
7	0	199338	803527	0	9	10	-	-	-	-	-

Table 9.3: Statistics on the size of the SCC Graph, livelocks and bounds on the communication places for the Revised TR-Protocol configurations

9.2.4 Dead Transitions

Table 9.4 lists the dead transitions for each configuration. The numbers in the header row correspond to the transition numbers in Tables 8.3, 8.14 and 8.20. The total number of dead transitions is also listed for each configuration. Closer inspection reveals the dead transitions for each configuration are those expected (see Tables 8.3, 8.14 and 8.20). Therefore, there are no unexpected dead transitions, proving Property 8.4 for all configurations analysed.

9.2.5 Upper Bounds on Communication Places

The upper integer bounds on the two communication places (i.e. the maximum number of PDUs in the communication medium at any one time), InitToResp and RespToInit, is an important statistic for the Revised TR-Protocol. It can be used, for example, to dimension buffers in the design and implementation of the TR-PEs so that received PDUs need not be discarded. During the state space analysis we recorded these bounds for each configuration (Table 9.3). The bounds on the InitToResp and RespToInit places are referred to as *I2R* and *R2I*, respectively. Closer inspection of *I2R* and *R2I* reveals they are related to the configuration. Equations 9.1 and 9.2 show the relationship between the bounds and parameters that hold for all configurations analysed.

$$I2R = \begin{cases} RCRImax + 2 & \text{if } RCRRmax = 0 \\ RCRImax + RCRRmax + 1 & \text{otherwise} \end{cases} \tag{9.1}$$

$$R2I = RCRImax + RCRRmax + 3 \tag{9.2}$$

As one would expect, the bounds depend on the maximum values of both of the re-transmission counters, *RCRImax* and *RCRRmax*. The more times a TR-PE re-transmits a PDU, the more number of PDUs can be in the communication medium.

For *I2R* (Equation 9.1), the TR-Init-PE can send an Invoke PDU (contributes 1 to *R2I*) and then re-transmit the Invoke PDU if it hasn't received any response from the TR-Resp-PE (contributes *RCRImax* to *I2R*). Then the TR-Init-PE may send an Abort PDU (contributes 1 to *I2R*). This is the maximum number of PDUs that can be in the communication channel if *RCRRmax*=0. If *RCRRmax*>0 then *before* the TR-Init-PE sends the Abort PDU, it may receive the Result PDU from the TR-Resp-PE. The TR-Init-PE can then send an Ack PDU (contributes 1 to *I2R*). For every re-transmitted Result PDU received, the TR-Init-PE re-transmits Ack PDUs (contributes *RCRRmax* to *I2R*). Note that after the TR-Init-PE has acknowledged the receipt of a Result PDU, it cannot send an Abort PDU. Thus there are two parts to Equation 9.1: one when the Abort PDU can be sent (*RCRRmax*=0), and the other when it cannot be sent.

Config	*DT*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
0-0-T	14	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0-1-T	12	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1
0-2-T	12	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1
0-3-T	12	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1
0-4-T	12	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1
1-0-T	7	0	0	1	1	1	1	1	0	0	0	0	0	1	0	1	0
1-1-T	5	0	0	1	1	1	0	0	0	0	0	0	0	1	0	1	0
1-2-T	5	0	0	1	1	1	0	0	0	0	0	0	0	1	0	1	0
1-3-T	5	0	0	1	1	1	0	0	0	0	0	0	0	1	0	1	0
1-4-T	5	0	0	1	1	1	0	0	0	0	0	0	0	1	0	1	0
2-0-T	6	0	0	1	1	1	1	1	0	0	0	0	0	0	0	1	0
2-1-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
2-2-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
2-3-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
2-4-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
3-0-T	6	0	0	1	1	1	1	1	0	0	0	0	0	0	0	1	0
3-1-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
3-2-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
3-3-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
3-4-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
4-0-T	6	0	0	1	1	1	1	1	0	0	0	0	0	0	0	1	0
4-1-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
4-2-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
4-3-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
4-4-T	4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0
5-0-T	6	0	0	1	1	1	1	1	0	0	0	0	0	0	0	1	0
6-0-T	6	0	0	1	1	1	1	1	0	0	0	0	0	0	0	1	0
7-0-T	6	0	0	1	1	1	1	1	0	0	0	0	0	0	0	1	0
0-0-F	11	1	1	0	0	0	1	1	1	1	1	1	1	1	0	0	1
0-1-F	9	1	1	0	0	0	0	0	1	1	1	1	1	1	0	0	1
0-2-F	9	1	1	0	0	0	0	0	1	1	1	1	1	1	0	0	1
0-3-F	9	1	1	0	0	0	0	0	1	1	1	1	1	1	0	0	1
0-4-F	9	1	1	0	0	0	0	0	1	1	1	1	1	1	0	0	1
1-0-F	5	1	1	0	0	0	1	1	0	0	0	0	0	1	0	0	0
1-1-F	3	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1-2-F	3	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1-3-F	3	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1-4-F	3	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
2-0-F	4	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0
2-1-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2-2-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2-3-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2-4-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3-0-F	4	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0
3-1-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3-2-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3-3-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3-4-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4-0-F	4	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0
4-1-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4-2-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4-3-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 9.4: Dead transitions for the Revised TR-Protocol configurations

For *R2I* (Equation 9.2), the TR-Resp-PE can send an Ack PDU (contributes 1 to *R2I*) after receiving the Invoke PDU, and re-transmit an Ack PDU every time it receives a re-transmitted Invoke PDU from the TR-Init-PE (contributes *RCRImax* to *R2I*). The TR-Resp-PE may then send the Result PDU (contributes 1 to *R2I*), which may be re-transmitted *RCRRmax* times (contributes *RCRRmax* to *R2I*). Finally, the TR-Resp-PE may send an Abort PDU (contributes 1 to *R2I*).

Further work is needed to determine the effects of other features, such as duplicates and SAR, on these bounds (see Chapter 10).

9.3 Applying the Sweep-Line Method

Ordinary state space analysis has been sufficient for proving the properties of simple configurations of the Revised TR-Protocol CPN. Section 9.4 shows how the results of Section 9.2 can be used as a first step towards verifying the properties independently of the parameter values. But to do this, we require some additional results of configurations that were "out of reach" of ordinary state space analysis, because of the state explosion problem. The sweep-line method [31] is used to obtain the required results.

The sweep-line analysis is performed using Design/CPN [109] and a prototype implementation of the Sweep-Line library [30]. The analysis process involves defining a progress measure (Section 9.3.1), setting up the sweep-line algorithm to calculate properties and results on-the-fly (Section 9.3.2), and then calculating the state space (the results are given in Section 9.3.3). We have again made some artificial changes to the TR-Protocol CPN to accommodate calculating a batch of state spaces (see Appendix F).

9.3.1 Progress Measure

For the sweep-line analysis to be applicable, we require a progress measure that never decreases as we calculate the state space. That is, the progress measures of the successor nodes of a node in the state space, node A, must be greater than or equal to the progress measure of node A. It is also advantageous, in terms of the number of nodes that need to be stored in memory at any one time, for the progress measure to partition the state space into many, small groups (as opposed to, for example, having half the nodes of the full state space with a progress measure of 1 and the other half with 2). For the Revised TR-Protocol we use the counter values used by the TR-PEs in the progress measure (we will denote them as *RCRI* and *RCRR*). The progress measure, given in Listing E.5, never decreases during the execution of the Revised TR-Protocol CPN (Appendix E). In general, the progress measure (PM) of a node is given by Equation 9.3. This has been chosen so that a unique value of the PM is obtained for each permutation of the counters.

In the following we discuss how this is achieved, and the benefits of the progress measure.

$$PM = RCRI * (RCRRmax + 2) + RCRR \tag{9.3}$$

Equation 9.3 calculates the progress measure from the two counter parameters, giving *RCRI* the most significance. That is, *RCRI* is given a weighting that makes it more than the maximum value of *RCRR*. As only one transaction can be initiated by the TR-Init-User, and the counters are not reset or decremented *during* a transaction, the progress measure in Equation 9.3 will always be increasing as the state space is calculated. Shortly we discuss how the progress measure is calculated *after* a transaction has been completed (either successfully or aborted). This also explains the decision to add 2 to the maximum counter value in the weighting, whereas adding 1 would normally be sufficient.

Table 9.5 shows PM for different values of the counters, when the Configuration is 1-2-T. As the two counters increase, so does PM. For a comparison, the fourth column (*Sum*) gives the value of the progress measure if a summation without weightings is used (i.e. *RCRI*+*RCRR*).

RCRI	*RCRR*	*PM*	*Sum*
0	0	0	0
0	1	1	1
0	2	2	2
1	0	4	1
1	1	5	2
1	2	6	3

Table 9.5: Example progress measure for Configuration 1-2-T

The effectiveness of the sweep-line method depends on the number of nodes with unique progress measures, and the rate at which garbage collection is performed. The goal is to delete as many nodes as possible when garbage collection occurs. Equation 9.3 partitions the state space into more (and smaller) groups of nodes with the same progress measure than, say a progress measure calculated from the sum of the counter values (cf. the *Sum* column in Table 9.5). However, the ordering of the counters significance is arbitrary. Further investigation into optimal progress measures and garbage collection rates is required (see Chapter 10 and Appendix G).

While a transaction is in progress, the transaction data in the Initiator and Responder places store the counter values. During state space generation, the marking of these places can be inspected to calculate the progress measure. However, when the place TempState on page I_RW_RcvResult_Cnf (Figure E.7) contains a token, Initiator does not. In these markings, the counter values are taken from TempState instead of Initiator. The calculation of PM is unchanged.

When a transaction is aborted or successfully completed, the TR-PEs re-enter their initial states (NULL and LISTEN), and the counters are re-set to their default values

(0). This violates the requirement for increasing counters. For example, if the PM is 2 from Table 9.5 and the TR-Resp-PE aborts the transaction, *RCRR* is set to 0, giving a PM of 0. To overcome this, on completion of a transaction at a TR-PE, the value of the re-transmission counter used in the calculation of the progress measure is set to one more than its maximum value. Returning to the example, after the TR-Resp-PE aborts, the value *RCRR* used in Equation 9.3 is 3 (*RCRRmax*+1) (although its actual value is 0). This gives PM=3, which is greater than the maximum value of any PM when *RCRI* is 0. As the counter values can be set to one more than their maximum value, the weighting in Equation 9.3 is calculated from the maximum value plus 2 (*RCRRmax*+2). Table 9.6 updates the progress measures from Table 9.5 with the values used in Equation 9.3 when either or both TR-PEs have completed the transaction.

Status	*RCRI*	*RCRR*	*PM*
In progress	0	0	0
In progress	0	1	1
In progress	0	2	2
TR-Resp-PE complete	0	3	3
In progress	1	0	4
In progress	1	1	5
In progress	1	2	6
TR-Resp-PE complete	1	3	7
TR-Init-PE complete	2	0	8
TR-Init-PE complete	2	1	9
TR-Init-PE complete	2	2	10
Both TR-PEs complete	2	3	11

Table 9.6: Example progress measure for Configuration 1-2-T including values for completed transactions

The markings of UserAck and First are used to determine whether the TR-Init-PE or TR-Resp-PE have completed the transaction. When UserAck is empty, and the value of *RCRI* is 0, the TR-Init-PE has completed the transaction. When the marking of First is 1‘F, and the value of *RCRR* is 0, the TR-Resp-PE has completed the transaction.

9.3.2 Properties Investigated

The advantage of the sweep-line method over ordinary state space analysis is that less nodes need to be stored at any one time. In ordinary state space analysis, the full state space is stored in memory, and then queries can be made to verify properties (e.g. dead markings, bounds). Using the sweep-line method, properties must be proved on the fly (i.e. during the calculation of the state space). For the Revised TR-Protocol CPN, collecting data to prove successful termination (Property 8.2) and absence of unwanted dead transitions (Property 8.4) is straightforward. To obtain the Revised TR-Protocol language, the arcs of the state space must be written to a file on-the-fly. This creates significant processing overhead, but memory consumption is still reduced in comparison

to ordinary state space analysis. Our approach of using the SCC graph to prove the absence of livelocks (Property 8.3) is not suited to sweep-line analysis, because the SCC graph is calculated from the full state space. This is a limitation of the approach.

Design/CPN's Sweep-Line Library [30] was only in a prototype form when first used. There was no support for proving properties on-the-fly. Lars Kristensen implemented the functionality to check the desired properties of the Revised TR-Protocol. The Standard ML code and a description is given in Appendix E.

9.3.3 Results of Sweep-Line Analysis

The configurations of the Revised TR-Protocol analysed using the sweep-line method are: 4-4-F, 0-8-T, 0-9-T and 0-10-T. The first configuration is part of the set of configurations up to, and including the suggested maximum counter values for GSM SMS networks [183]. The choice of the latter three configurations will become apparent in Section 9.4. Garbage collection is performed every 10000 nodes. This was an estimate to minimize processing overhead and memory usage based on results from other applications of the sweep-line method [31].

For the configurations analysed using the sweep-line method, the language of the Revised TR-Protocol was equivalent to the corresponding language of the TR-Service (see Table 9.7), proving Property 8.1.

Config	*N*	*A*	*H*	*Seq*	*L*	*S*	*NIS*	*NIP*
4-4-F	19	63	4	182	8	2	0	0
8-0-T	19	61	2	130	8	2	0	0
9-0-T	19	61	2	130	8	2	0	0
10-0-T	19	61	2	130	8	2	0	0

Table 9.7: Statistics on the size of the language for the Revised TR-Protocol configurations using sweep-line analysis

Table 9.8 lists the state space statistics, number of terminal markings and deadlocks and bounds on the communication places measured during the sweep-line analysis. As well as the nodes (N), arcs (A) and calculation time (T), the statistics include the maximum number of nodes stored at any one time (*Stored*) and the reduction when compared to ordinary state space analysis. This may be used as a guide to the amount of memory saved using the sweep-line method, although there are other factors that influence this (e.g. the amount of memory per node may increase when getting deeper into the state space).

As well as proving Property 8.2 (Successful Termination), Table 9.8 shows the sweep-line method can result in savings of the order of 50% of memory. This is particularly useful for the Revised TR-Protocol. Appendix G evaluates the performance enhancements of the sweep-line method (in terms of memory and time saved) in more detail.

Config	*N*	*Stored*	*Reduction*	*A*	*T*	*TM*	*DL*	*I2R*	*R2I*
4-4-F	513653	322632	37%	2035390	12289	43454	0	9	11
8-0-T	385848	196143	49%	1589883	5607	11306	0	10	11
9-0-T	704881	334327	53%	2963984	14056	17348	0	11	12
10-0-T	1226460	554059	55%	5254414	34369	25685	0	12	13

Table 9.8: State space statistics, dead markings and bounds for the Revised TR-Protocol configurations when sweep-line analysis used

Table 9.9 shows the dead transitions for the configurations analysed using the sweep-line method. These are all expected transitions, proving Property 8.4 (Absence of Unexpected Dead Transitions). Property 8.3 (Absence of Livelocks) has *not* been proved for these configurations.

Config	*DT*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
4-4-F	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8-0-T	6	0	0	1	1	1	1	1	0	0	0	0	0	0	0	1	0
9-0-T	6	0	0	1	1	1	1	1	0	0	0	0	0	0	0	1	0
10-0-T	6	0	0	1	1	1	1	1	0	0	0	0	0	0	0	1	0

Table 9.9: Dead transitions for the Revised TR-Protocol configurations using sweep-line analysis

9.4 Impact of Parameters on State Space Size

State spaces have been calculated for a total of 56 configurations using both ordinary state space (Section 9.2) and sweep-line analysis (Section 9.3). These configurations have covered the following range of parameter values:

- all permutations of the counter parameters (*RCRImax* and *RCRRmax*) with the values 0, 1, 2, 3 and 4, for both *UserAck* set to T and F; and
- *RCRImax* increased from 5 to 10 when *RCRRmax*=0 and *UserAck*=T.

For all the analysed configurations we have proved that the TR-Protocol: a) refines the TR-Service (Property 8.1); b) terminates successfully (Property 8.2); and c) does not contain unwanted dead transitions (Property 8.4). For all configurations except those analysed using the sweep-line method (see Section 9.3), absence of livelocks has also been proved (Property 8.3).

The increase in the size of the state space that can be calculated using the sweep-line method is promising, but is not enough for more complex configurations of the Revised TR-Protocol. For example, the WTP Specification suggests 8 as a default value for *RCRImax* and *RCRRmax* in an IP network [183]. A rough estimate puts the number of

nodes for Configuration 8-8-T at 10,000,000[1]. Furthermore, including the possibility of errors in the CPN model (e.g. duplicates, losses) is likely to lead to further explosion of the state space. Clearly, the brute force approach of calculating the state space for more complex configurations is not adequate.

In this section we use the state space and sweep-line analysis results to make observations regarding the dependence of the state space size on the configuration parameters. This is a step towards the desirable situation of having a CPN that models the Revised TR-Protocol independently of the parameter values (or at least the counters), but still captures the effects of the parameters on behaviour. This was achieved for the acknowledgment expiration counter (`AEC`), however it is not clear what the best approach is for `RCR`. Our parameterized Revised TR-Protocol CPN has served not only as a platform for proving properties of specific configurations, but also for experimenting with the effects of the parameters on the TR-Protocol. The observations in the remainder of this section are based on these experiments. Due to time limitations, some of the observations (especially for *RCRImax*) remain just that, in that no attempt has been made to prove them. This is an area for future research.

9.4.1 UserAck

There are no clear relationships between the size of the state space and the *UserAck* parameter. However, we do know that there is some commonality of the Revised TR-Protocol when *UserAck* is T and F and, therefore, expect much of the state space to be common. It may be that modelling *UserAck* as a non-deterministic choice (as opposed to setting it to either T or F) will not dramatically increase the state space size. Further investigation of this is required.

9.4.2 RCRRmax

For *RCRRmax* there is a clear relationship observed with it and the size of the state space. As an example, Table 9.10 shows the number of nodes for different values of *RCRRmax* when *RCRImax* is 0 and *UserAck*=T. The trend observed is that the difference between the number of nodes when *RCRRmax*$= r$ and *RCRRmax*$= r+1$ always increases by 40 as r increases. Similar trends are also observed for the number of arcs, and for configurations when *RCRImax* is set to 1, 2, 3 and 4 and *UserAck*=F. Equation 9.4 characterizes these trends. C is either the number of nodes (N) or number of arcs (A). The constants C_0, X_C and Y_C are given in Table 9.11.

[1] The rate of increase in the number of nodes when *RCRRmax* is 0 and *RCRImax*=0...8 is used to extrapolate the rate of increase when *RCRRmax*=8 for *RCRImax*=5...8 (the number of nodes when *RCRImax*=0...4 and *RCRRmax*=8 are calculated from Equation 9.4 in Section 9.4.2).

RCRRmax (r)	*Nodes (N)*	$X = N_{r+1} - N_r$	$Y = X_{r+1} - X_r$
0	105	129	40
1	234	169	40
2	403	209	40
3	612	249	-
4	861	-	-

Table 9.10: Change in number of state space nodes when varying *RCRRmax*

$$C_r = \frac{Y_C}{2} r^2 + (X_C - \frac{Y_C}{2}) r + C_0 \tag{9.4}$$

UserAck	*RCRImax*	*Nodes*			*Arcs*		
		N_0	X_N	Y_N	A_0	X_A	Y_A
T	0	105	129	40	304	396	136
	1	513	667	214	1677	2315	826
	2	1872	2535	836	6528	9296	3382
	3	6032	8455	2856	21865	31749	11662
	4	17024	24483	8460	63597	93548	34624
F	0	198	344	160	606	1096	544
	1	1009	1808	856	3430	6468	3304
	2	3796	6966	3344	13664	26224	13528
	3	12545	23478	11424	46591	90117	46648
	4	36165	68612	33840	137474	266735	138496

Table 9.11: Constants for Eq. 9.4 when varying *UserAck* and *RCRImax*

Equation 9.4 fits the empirical data obtained from analysing the configurations of the Revised TR-Protocol. If it can be proved that the equation holds for all configurations (e.g. when *RCRImax* and *RCRRmax* are greater than 4), then the state space size can be obtained from the equation. For example, for Configuration 2-8-F, the number of nodes and arcs in the state space can be calculated from Equation 9.4 as follows:

$$\begin{aligned}
N_r &= \frac{Y_N}{2} r^2 + (X_N - \frac{Y_N}{2}) r + N_0 \\
N_8 &= \frac{3344}{2} 8^2 + (6966 - \frac{3344}{2}) 8 + 3796 \\
&= 153156 \\
A_r &= \frac{Y_A}{2} r^2 + (X_A - \frac{Y_A}{2}) r + A_0 \\
A_8 &= \frac{13528}{2} 8^2 + (26224 - \frac{13528}{2}) 8 + 13664 \\
&= 602240
\end{aligned}$$

To increase our confidence in the applicability of Equation 9.4, Configuration 2-8-F has been analysed, and the state space does indeed have 153156 nodes and 602240 arcs. Listing 9.1 shows the state space report for this configuration.

Listing 9.1: State space report generated by Design/CPN for Configuration 2-8-F of the Revised TR-Protocol CPN

```
1  Statistics
2 ------------------------------
3  Occurrence Graph
4    Nodes:  153156
5    Arcs:   602240
6    Secs:   2042
7    Status: Full
8
9  Scc Graph
10   Nodes:  153156
11   Arcs:   602240
12   Secs:   223
13
14 Liveness Properties
15 ------------------------------
16 Dead Markings:  22641 [996,99330,99329,9900,99,...]
17 Dead Transitions Instances:
18
19 I_RESULT_RESP_WAIT'TimerTO_A_Max 1
20 R_INVOKE_RESP_WAIT'TimerTO_A_Max 1
```

Proving Equation 9.4 holds for all configurations requires an understanding of how *RCRRmax* impacts the state space. The value of *RCRRmax* determines the number of times transition TimerTO_R occurs (R_RESULT_RESP_WAIT page, Figure E.14). This transition re-transmits a Result PDU. The Result PDU can be received by the TR-Init-PE while its in the RESULT WAIT state (see Figure E.6) or the WAIT TIMEOUT state (Figure E.9). Further investigation of the influence of the re-transmitted Result PDU on the Revised TR-Protocol is required to determine if a more abstract model of *RCRRmax* can be used (e.g. if the properties are proved when a single re-transmission occurs, then can we inductively prove the properties for all other re-transmissions?).

9.4.3 RCRImax

For the parameter *RCRRmax*, trends were observed from calculating the state space with the values ranging from 0 to 4. The size of the state spaces were all within the capabilities of Design/CPN on the computer used. For *RCRImax*, no clear trend was observed for this range of values, except that the state space size increased more rapidly than for *RCRRmax*. However, analysing configurations for greater values of *RCRImax* (when *RCRRmax*=0 and *UserAck*=T) has given strong evidence of a trend similar to that observed for *RCRRmax*. Ordinary state space analysis was used for *RCRImax* with values 5, 6 and 7, and sweep-line analysis for values 8, 9 and 10. Table 9.12 shows the number of nodes for these configurations.

Table 9.12 shows the number of nodes increases much faster than for *RCRRmax*. The

RCRImax (i)	*Nodes (N)*	*T*	*U*	*V*	*W*	*X*	*Y*	*Z*
0	105	408	951	1850	2181	1501	531	83
1	513	1359	2801	4031	3682	2032	614	76
2	1872	4160	6832	7713	5714	2646	690	76
3	6032	10992	14545	13427	8360	3336	766	76
4	17024	25537	27972	21787	11696	4102	842	-
5	42561	53509	49759	33483	15798	4944	-	-
6	96070	103268	83242	49281	20742	-	-	-
7	199338	186510	132523	70023	-	-	-	-
8	385848	319033	202546	-	-	-	-	-
9	704881	521579	-	-	-	-	-	-
10	1226460	-	-	-	-	-	-	-

Table 9.12: Change in number of state space nodes for *RCRRmax*=0 and *UserAck*=T

differences between the successive configurations' number of nodes are shown in column T, the differences between successive values in column T are shown in column U, and so on. The final column, Z, indicates a possible relationship between number of nodes and *RCRImax*, although *RCRImax*=0 is an exception. Equation 9.5 gives the number of nodes in the state space for $i = 1 \ldots 10$, where i is *RCRImax*. The constants are given in the row where *RCRImax*=1 in Table 9.12. Note that when the subject of the factorial evaluates to a negative number in the denominator, the fraction becomes 0.

$$N_{i+1} = N_1 + T_1 i + i! \left(\frac{U_1}{2!(i-2)!} + \frac{V_1}{3!(i-3)!} + \frac{W_1}{4!(i-4)!} + \frac{X_1}{5!(i-5)!} + \frac{Y_1}{6!(i-6)!} + \frac{Z_1}{7!(i-7)!} \right) \quad (9.5)$$

As an example, to calculate the number of nodes for Configuration 4-0-T, we have:

$$\begin{aligned} N_4 &= N_1 + 3T_1 + 3!\frac{U_1}{2!\,1!} + 3!\frac{V_1}{3!\,0!} \\ &= 513 + 3.1359 + 3.2801 + 4031 \\ &= 17024 \end{aligned}$$

Note that Equation 9.5 only applies for the nodes. The arcs do not fit this equation. The four corresponding values of Z for the arcs are 498, 459, 460 and 462. For $i = 1 \ldots 10$, where the difference of the Z values for the number of nodes is 0, for arcs the difference is 1 (i.e. $460 - 459$), then 2 (i.e. $462 - 460$). With *RCRImax* set to 11 or 12, we may be able to better observe the relationship for arcs.

Equation 9.5 may also be expressed as a 7th order polynomial in i ($i \geq 7$), clearly showing the relationship between *RCRImax* and the number of nodes and arcs is more complex than for the parameter *RCRRmax* (which is quadratic). (Equation 9.5 is expressed in terms of factorials for readability, as the coefficients of the polynomial are

non-trivial.) This is because *RCRImax* determines the number of times an Invoke PDU (transition TimerTO_R in Figure E.6) or Ack(Tok) PDU (transitions TimerTO_R_Tve and RcvAck_Tve in Figure E.6) may be re-transmitted. These re-transmitted PDUs have an impact on the remainder of the CPN, which has yet to be determined. This is an area for future work (Chapter 10).

9.5 Summary

Three errors were discovered, and fixes suggested, in the TR-Protocol in Chapter 8. In this chapter the Revised TR-Protocol, which incorporates the fixes, was analysed. The analysis in Section 9.2 has shown that for a set of configurations leading up to the suggested configuration for a GSM SMS network [183] (Section 9.1), the Revised TR-Protocol provides the intended TR-Service, successfully terminates, and does not contain livelocks nor unwanted state table entries. These properties give high confidence that the design of the Revised TR-Protocol is functionally correct.

In a first step towards extending the analysis results to other configurations, sweep-line analysis was used to calculate state spaces that have sizes out of reach of ordinary state space analysis (Section 9.3). The results were then used to determine relationships between the state space size and the parameter values (Section 9.4). Further work is necessary to see if the trends observed can be used to inductively prove properties of the Revised TR-Protocol, giving results that are independent of the values of the two parameters *RCRImax* and *RCRRmax*.

We have now reached our aim of verifying the Wireless Transaction Protocol, albeit only once several changes were made to the WTP Specification [183] and for a small set of configurations. Steps have been made towards a more general verification which is independent of specific parameter values. It is envisaged the results presented will be used to improve the design of the WAP architecture, and subsequent implementations, resulting in a stable and reliable system for providing mobile data services.

Chapter 10

Conclusions

10.1 Contributions of the Dissertation

The Wireless Transaction Protocol has recently been defined and updated [183, 187], but as far as we are aware there have been no other published attempts at verifying the correctness of the design. This thesis presents several key results from the verification procedure applied to WTP Class 2.

10.1.1 Service Specification

In the WTP Specification [183], the Transaction Service is described using narrative descriptions and a table specifying when a service primitive can immediately follow another. A critique of the existing TR-Service has been given, identifying the following problems:

- the TR-Invoke and TR-Result confirmation primitives do not exhibit end-to-end behaviour, in that, for example, a confirmation of the TR-Invoke can be delivered to the TR-Init-User without the TR-Resp-User acknowledging the TR-Invoke (with a response);
- there is no complete definition of when a transaction ends, either successfully or as a result of an abort; and
- the purpose of the TR-Abort primitive is not clear (e.g. the TR-Init-User should not be able to submit a TR-Abort primitive after acknowledging the result).

These problems were clarified by introducing five assumptions (Assumptions 6.4 to 6.8) that, along with the table in the WTP Specification, define the legal sequences representing successful transactions (when user acknowledgment is On or Off) and aborted transactions, and when TR-Abort primitives can be submitted by, or delivered to, the TR-Init-User or TR-Resp-User. (The latter two ambiguities were presented, along with a CPN model and language of the service, in [54].)

A Coloured Petri net of the Transaction Service was developed for the purpose of automatically generating all possible sequences of service primitives, and to give confidence that the sequences accurately represent the behaviour in practice. The state space of the CPN was used to obtain two Transaction Service languages: one when the User Acknowledgment (UserAck) feature is On, and the other when UserAck is Off. The languages are used as a basis for the verification of the Transaction Protocol (Section 10.1.3).

10.1.2 Protocol Specification

The Transaction Protocol is described by a set of state tables in the WTP Specification [183]. A CPN model of the Transaction Protocol, based on these state tables for validation purposes, was developed. The model was tailored towards the verification task, by introducing a set of simplifications and restrictions. The simplifications do not limit the applicability of the analysis results, whereas the restrictions do. The restrictions are:

- only Class 2 transactions of WTP version 1.2.1 (given in [183]) are modelled, Class 0 and 1 transactions are omitted and so is the version handling protocol feature;
- the segmentation and re-assembly protocol feature is not modelled; and
- the Transport Service provider is assumed error free (i.e. no corruption, losses or duplicates, although overtaking is allowed), and has infinite capacity.

A set of 10 assumptions were made due to errors being present in the WTP Specification. These include:

- conditions in state table entries being incomplete, in that the conditions for two different entries with the same event are not mutually exclusive when the actions suggest they must be;
- no specification of actions upon the receipt of PDUs in some states (see Table 7.2), when the PDUs can be received in these states;
- the counter RCR going above its maximum value when the TR-Init-PE receives an Ack(Tve) PDU after re-transmitting the Invoke PDU or Ack(Tok) PDU the maximum possible number of times;
- a TR-User not being notified of the end of a transaction (via the delivery of a TR-Abort.ind primitive) when the TR-PE aborts due to no response (within a given time limit) from the TR-User; and
- a TR-Result.req primitive being submitted by the TR-Resp-User before a TR-Invoke.res primitive, when UserAck is On.

Several of these errors, along with an initial CPN model of the protocol, were presented in [56]. They were also submitted to the WAP Forum [55], who followed the suggestions for improving the protocol, as can be seen in Version 2.0 of WTP [187].

10.1.3 Analysis of the Protocol

Four desired properties of the Transaction Protocol, which give a high level of confidence in its design, were defined. They are: faithful refinement of the service (language equivalence between protocol and service); successful termination (including absence of deadlocks); absence of livelocks; and absence of un-used state table entries (dead transitions in the CPN).

Generation of the state space and language from the CPN, and comparison with the corresponding TR-Service language, revealed further errors in the TR-Protocol. The errors, and proposed solutions that led to a Revised TR-Protocol (Section 10.1.4), are:

- The Ack and Result PDUs are ambiguous, in that they do not correctly convey whether or not the peer TR-User has acknowledged an invocation or a result. For example, the TR-Init-User can be delivered the TR-Invoke.cnf primitive, without the TR-Resp-User submitting the TR-Invoke.res primitive.

  Solution: Both the Ack PDU and Result PDU require a new 1-bit header field, called `CNF`. When set to 1 in a PDU sent by a TR-PE, the `CNF` field indicates that the TR-User has submitted an acknowledgment (i.e. a response primitive). When set to 0, `CNF` indicates the acknowledgment is only from the TR-PE, i.e. the TR-User has not submitted an acknowledgment. The state tables are modified so that there are entries that receive these PDUs with `CNF` set to 0 or 1, and the relevant primitive is delivered to the TR-User, if necessary.

- When the TR-Resp-PE receives a re-transmitted Invoke PDU with TID x after it has aborted transaction with TID x, the TR-Resp-PE initiates TID verification. If the TR-Init-PE still has transaction x outstanding (e.g. it has received no notification of the abort by the TR-Resp-PE), then the TID verification will be successful, and the TR-Resp-PE will re-start transaction x, delivering a second TR-Invoke.ind primitive to the TR-Resp-User. This is an error because only one TR-Invoke.req primitive has been submitted by the TR-Init-User.

  Solution: We introduce a period, W, which the TR-Resp-PE must wait before it can re-start transaction x (which was previously aborted). A timer begins with the interval W after the TR-Resp-PE or TR-Resp-User aborts transaction x. The TR-Resp-PE cannot initiate TID verification upon receipt of an Invoke PDU with TID x while the timer is running. Hence, the second TR-Invoke.ind cannot be delivered

to the TR-Resp-User. The value of W must be greater than the maximum packet lifetime (MPL) for the network (but should be less than 4MPL) so that all re-transmitted or duplicated Invoke PDUs with TID x have been discarded when the timer expires.

- After a successful transaction (with TID x), or a transaction aborted by the TR-Init-PE, a re-transmitted Invoke PDU with TID x may be received by the TR-Resp-PE initiating TID verification. If a re-transmitted Ack(Tok) PDU with TID x is also received by the TR-Resp-PE, then it is treated as a successful acknowledgment of TID verification. This is an error because the TR-Init-PE has no outstanding transaction with TID x.

  Solution: We now start the waiting period suggested in the previous proposal after a transaction is completed by any means (including the TR-Init-PE aborting the transaction, or the transaction being successful). This ensures any re-transmitted Invoke PDUs cannot initiate TID verification, therefore preventing the misinterpretation of a re-transmitted Ack(Tok) PDU.

10.1.4 Revised Protocol and its Verification

The changes suggested to correct errors in the Transaction Protocol are implemented, giving rise to the Revised Transaction Protocol (and corresponding CPN). To demonstrate the proposed solutions are valid, and increase confidence that the Revised TR-Protocol is functionally correct, state space and language analysis of the Revised TR-Protocol was performed. The analysis proved that the Revised TR-Protocol satisfied the four desired properties (see Section 10.1.3)[1] for the following ranges of initial parameter values:

- *UserAck* is T or F, and all permutations of *RCRImax* and *RCRRmax* taking the values 0, 1, 2, 3 and 4. The suggested values of the counter parameters in a GSM SMS network are both 4 [183], so this provides a practical result of commercial significance.

- *UserAck* is T, *RCRRmax*=0 and *RCRImax* is increased from 5 to 10.

10.1.5 Closed Form Solutions for the Size of the State Space

Trends in the size of the state space were observed from the range of initial configurations of the Revised TR-Protocol that were analysed. Closed form solutions for the number of nodes and arcs of the state spaces, in terms of the maximum counter value *RCRRmax*,

[1] Absence of livelocks was *not* proved for Configuration 4-4-F or when *RCRImax* is greater than 7.

have been given, for when $RCRImax$=0...4 and $UserAck$ is T or F (see Equation 9.4). A similar solution (Equation 9.5) has been given for the number of nodes when $RCRImax$ increases from 1 to 10 (for $RCRRmax$=0 and $UserAck$=T).

10.1.6 Application of the Sweep-Line Method

This is one of the first applications of the sweep-line method to a case study of significant size, and indeed the first for the domain of distributed transaction protocols. (The only other such case studies we are aware of are presented in [31].) In particular, for the Revised TR-Protocol, sweep-line analysis has allowed the proof of properties for configurations with state spaces with more than 1 million nodes. This is approximately double the number of nodes that can be calculated using ordinary state space analysis. Therefore, for this application, the sweep-line method has been useful for obtaining new results. However, it is not sufficient on its own to seriously tackle the state explosion problem.

10.2 Future Work

10.2.1 Obtaining Results for Arbitrary Parameter Values

The trends in the state space size observed in Chapter 9 indicate that it is likely that the number of markings can be computed from a closed form expression involving the input parameters. Further investigation should be made to determine the validity of these expressions for arbitrary values of $RCRImax$ and $RCRRmax$. Once known, abstractions of the parameters may be able to be used in the Revised TR-Protocol CPN, giving analysis results with a wider applicability. This may facilitate proofs by induction to allow the results to be generalised for any values of the input parameters.

10.2.2 Relaxing Restrictions on the Revised TR-Protocol CPN

Three restrictions were imposed on the TR-Protocol CPN to simplify the analysis (Chapter 7). The Revised TR-Protocol should be investigated with these restrictions relaxed, and eventually removed. The model of the Transport Service provider (i.e. the communication places) can be modified to include loss and duplication of PDUs (e.g. using transitions that arbitrarily remove and/or duplicate PDU tokens in the communication places). A more detailed model of the buffers and state table pages is required if segmentation and re-assembly of PDUs is allowed. Finally, it should be possible to obtain results for Class 0 and Class 1 transactions from the current work, as, in most cases, the features are a subset of those already present in Class 2. After including transitions

that model the state table entries used by the Class 0 and 1 transactions, a parameter or non-determinism could be used in the CPN to select a class of transaction.

10.2.3 The Sweep-Line Method and Other Analysis Techniques

To alleviate the problems of state space analysis, other analysis techniques should be investigated. The sweep-line method has been useful for this application, achieving reductions in the amount of memory used of approximately 50%. However, little effort has been put into examining alternative progress measures or garbage collection mechanisms. Currently, the progress measure only uses the values of the two re-transmission counters. A progress measure with finer granularity (i.e. less nodes have the same progress value) may be obtained if the state names of the protocol entities are also used. A greater reduction may then result. Other state space reduction techniques, such as stubborn sets [162, 96], may also prove useful.

10.2.4 Other Protocol Engineering Activities

This thesis has focused on verifying the TR-Protocol (and the subsequently Revised TR-Protocol) against the TR-Service. Other activities in the protocol engineering methodology may also be applied. The current CPN model could be easily modified to represent a formal specification of the Revised TR-Protocol by including all PDU header fields and primitive parameters. Performance evaluation of the Wireless Transaction Protocol is a vital step (especially for evaluating the impact of the changes to the TR-Protocol suggested in Chapter 8), which may be performed by augmenting the CPN with time [87]. Automatic code generation may be used to produce a verified implementation from the CPN. Experiments may be undertaken with this implementation as another method for investigating the performance, or it may be used as a baseline implementation when testing other implementations for interoperability. Finally, test cases may be generated from the verified formal specification for conformance testing.

10.2.5 Maintenance of the Revised TR-Protocol CPN

The modelling and analysis reported in this thesis has been of the June 2000 Conformance Release of the Wireless Transaction Protocol [183]. Version 2.0 is the latest version of WTP [187]. The Revised TR-Protocol CPN will need to incorporate any changes in Version 2.0. (Preliminary investigations into Version 2.0 suggests the changes are minor.) More importantly, the CPN model should be designed with maintainability in mind, to support further updates. From our experience with updating the CPN from Version 1.0 [172], changes such as new state table entries are well supported in the model. The

possibility of incrementally analysing the CPN [100], so the existing results can be re-used, should also be investigated. This may prove fruitful for analysing the Revised TR-Protocol with the Segmentation and Re-assembly (SAR) protocol feature, as SAR may be able to be treated with some independence of other features.

10.2.6 Generalisation to Other Transaction Protocols

Considerable effort has been spent on investigating the Wireless Transaction Protocol. It may be that some of the ideas (especially modelling patterns and conventions), models and results can be applied to other transaction protocols (e.g. T/TCP [20], IOTP [23], SIP [65]). The TR-Service is a reasonable starting point for defining a general transaction service. However, the requirements of users of other transaction protocols may also need to be incorporated. Although the details of a protocol may be specific to a domain, there will be some features of transaction protocols in common. Identifying, modelling and analysing such features, both individually and as a whole (for which the Revised TR-Protocol CPN may be of use), would be beneficial.

References

[1] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP over satellite channels using standard mechanisms. IETF RFC 2488 URL: `ftp://ftp.isi.edu/in-notes/rfc2488.txt`, Jan. 1999.

[2] T. Arts. Private communication, 24 Nov. 1999.

[3] Assoication of Radio Industries and Businesses (ARIB). Web site: `http://www.arib.or.jp/` [Last accessed: 23 Nov. 2001], 9 Aug. 2001.

[4] AT&T. FSM Library. Web site: `http://www.research.att.com/sw/tools/fsm` [Last accessed: 23 Nov. 2001].

[5] AT&T. GraphViz. Web site: `http://www.research.att.com/sw/tools/graphviz` [Last accessed: 23 Nov. 2001].

[6] AT&T. Lextools. Web site: `http://www.research.att.com/sw/tools/lextools` [Last accessed: 23 Nov. 2001].

[7] H. Balakrishnan. *Challenges to Reliable Data Transport over Heterogeneous Wireless Networks*. PhD thesis, Electrical Engineering and Computer Sciences Department, University of California, Berkeley, CA, Aug. 1998.

[8] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, Dec. 1997.

[9] W. A. Barret, R. Bates, D. Gustafson, and J. D. Couch. *Compiler Construction: Theory and Practice*. Science Research Associates, second edition, 1986.

[10] M. Y. Bearman, M. C. Wilbur-Ham, and J. Billington. Specification and analysis of the OSI Class 0 Transport Protocol. In *Proceedings of the 7th International Conference on Computer Communications*, pages 602–607, Sydney, Australia, 30 Oct.–2 Nov. 1984. North-Holland.

[11] H. Beker and F. Piper. *Cipher Systems*. Northwood Books, London, 1982.

[12] T. Berners-Lee and M. Fischetti. *Weaving the Web: The Past, Present and Future of the World Wide Web by its Inventor.* Orion Business, London, 1999.

[13] J. Billington. Abstract specification of the ISO Transport service definition using labelled Numerical Petri nets. In *Protocol Specification, Testing, and Verification, III*, pages 173–185. Elsevier Science Publishers, Amsterdam, 1983.

[14] J. Billington. Formal specification of protocols: Protocol engineering. In *Encyclopedia of Microcomputers*, pages 299–314. Marcel Dekker, New York, NY, 1991.

[15] J. Billington. Protocol specification using P-Graphs, a technique based on Coloured Petri nets. In *Advances in Petri Nets, Advance Course in Petri Nets*, pages 293–330. Springer-Verlag, 1999.

[16] J. Billington, M. Diaz, and G. Rozenberg, editors. *Application of Petri Nets to Communication Networks: Advances in Petri Nets*, volume 1605 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1999.

[17] J. Billington, M. Farrington, and B. Du. Modelling and analysis of multi-agent communication protocols using CP-nets. In *Proceedings of the 3rd Biennial Engineering Mathematics and Applications Conference*, pages 119–122, Adelaide, Australia, 13-16 July 1998.

[18] J. Billington, M. C. Wilbur-Ham, and M. Y. Bearman. Automated protocol verification. In *Protocol Specification, Testing and Verification*, V, pages 59–70. North Holland, 1986.

[19] R. Braden. Extending TCP for transactions—concepts. IETF RFC 1379 URL: `ftp://ftp.isi.edu/in-notes/rfc1379.txt`, Nov. 1992.

[20] R. Braden. T/TCP—TCP extensions for transactions functional specification. IETF RFC 1644 URL: `ftp://ftp.isi.edu/in-notes/rfc1644.txt`, July 1994.

[21] E. A. Brewer, R. H. Katz, Y. Chawathe, S. D. Gribble, T. Hodes, G. Nguyen, M. Stemm, T. R. Henderson, E. Amir, H. Balakrishnan, A. Fox, V. N. Padmanabhan, and S. Seshan. A network architecture for heterogeneous mobile computing. *IEEE Personal Communications*, 5(5):8–24, Oct. 1998.

[22] S. Budkowski, B. Alkechi, M. L. Benalycherif, P. Dembinski, M. Gardie, E. Lallet, J. P. M. L. Fusse, and Y. Soussi. Formal specification, validation and performance evaluation of the Xpress Transfer Protocol. In *Protocol Specification, Testing and Verification, XIII*, Liege, Belgium, 25-28 May 1993. North-Holland.

[23] D. Burdett. Internet Open Trading Protocol—IOTP version 1.0. IETF RFC 2801 URL: `ftp://ftp.isi.edu/in-notes/rfc2801.txt`, Apr. 2000.

[24] J. Cai and D. J. Goodman. General Packet Radio Service in GSM. *IEEE Communications Magazine*, 35(10):122–131, Oct. 1997.

[25] A. T. Campbell, J. Gomez, S. Kim, Z. Turanyi, C.-Y. Wan, and A. Valko. Design, implementation and evaluation of Cellular IP. *IEEE Personal Communications*, 7(4):42–49, Aug. 2000.

[26] A. T. Campbell and J. Gomez-Castellanos. IP micro-mobility protocols. *Mobile Computing and Communications Review*, 4(4):45–53, Oct. 2000.

[27] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Events Systems.* Kluwer Academic Publishers, Boston Dordrecht London, 1999.

[28] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Transactions on Networking*, 5(4):514–524, Aug. 1997.

[29] O. Catrina. Protocol analysis and verification methods, application to the Xpress Transport Protocol (XTP) 4.0. In *Protocol Specification, Testing and Verification, XV*, pages 385–400. Chapman & Hall, London, UK, 1995.

[30] S. Christensen, L. M. Kristensen, and T. Mailund. *Design/CPN Sweep-Line Method Library.* Department of Computer Science, Aarhus University, Aarhus, Denmark, 2001.

[31] S. Christensen, L. M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *Proceedings of the Seventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 450–464, Genova, Italy, 2-6 Apr. 2001. Volume 2031 of Lecture Notes in Computer Science, Springer-Verlag.

[32] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 117–127, Austin, Texas, Jan. 1983.

[33] B. Collas, S. Gordon, and H. Widjaja. *IWS Design Specification V1.1.* University of South Australia, Adelaide, Australia, 1997.

[34] D. E. Comer. *Computer Networks and Internets.* Prentice Hall, Upper Saddle River, NJ, second edition, 1999.

[35] D. E. Comer. *Internetworking with TCP/IP.* Prentice Hall, Upper Saddle River, NJ, fourth edition, 2000.

[36] Computer Systems Engineering Centre. University of South Australia, Web site: `http://www.unisa.edu.au/eie/csec/` [Last accessed: 23 Nov. 2001], 1 June 2001.

[37] Consultative Committee for Space Data Systems. Space Communications Protocol Standards (SCPS): Rationale, requirements and application notes. CCSDS Recommendation 710.0-G-0.4, Aug. 1998.

[38] DataTAC Open Protocol Forum. Web site: `http://www.dopforum.com/` [Last accessed: 23 Nov. 2001], 19 Mar. 2001.

[39] J. C. A. de Figueiredo and L. M. Kristensen. Using Coloured Petri nets to investigate behavioural and performance issues of TCP protocols. In *Proceedings of the 2nd Workshop on the Practical Use of Coloured Petri Nets and Design/CPN*, pages 21–40, Aarhus, Denmark, 13–15 Oct. 1999. Department of Computer Science, Aarhus University.

[40] M. Degermark, M. Engan, B. Nordgren, and S. Pink. Low-loss TCP/IP header compression for wireless networks. *Wireless Networks*, 3(5):375–387, Oct. 1997.

[41] J. Desel and J. Esparza. *Free choice Petri nets.* Cambridge University Press, Cambridge, New York, 1995.

[42] A. A. Desrochers and R. Y. Al-Jaar. *Applications of Petri Nets in Manufacturing Systems: Modeling, Control and Performance Analysis.* IEEE Press, Piscataway, NJ, 1995.

[43] D. Dougherty and A. Robbins. *sed and awk.* O'Reilly and Associates, Sebastopol, CA, 2nd edition, Mar. 1997.

[44] K. Enoki. i-mode: the mobile Internet service of the 21st century. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 12–15, San Francisco, CA, 5–7 Feb. 2001.

[45] ETSI. Radio Equipment and Systems (RES); Digitial Enhanced Cordless Telecommunications (DECT) Common Interface Part 1: Overview. ETS 300 175-1, Oct. 1992.

[46] ETSI. Radio Equipment and Systems (RES); Trans-European Trunked Radio (TETRA); Voice plus Data (V+D) Part 1: General Network Design. ETS 300 392-1, Nov. 1994.

[47] M. Farrington and J. Billington. Analysing a Coloured Petri net model of an assembly line. In *Proceedings of the 3rd Biennial Engineering Mathematics and Applications Conference*, pages 193–196, Adelaide, Australia, 13-16 July 1998.

[48] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. IETF RFC 2616 URL: `ftp://ftp.isi.edu/in-notes/rfc2616.txt`, June 1999.

[49] D. Flanagan. *JavaScript: the definitive guide*. O'Reilly & Associates, Cambridge, Sebastopol, CA, 2nd edition, 1997.

[50] V. K. Garg. *IS-95 CDMA and cdma2000: Cellular/PCS Systems Implementation*. Prentice-Hall, Upper Saddle River, NJ, 2000.

[51] S. Gordon and J. Billington. Analysing a missile simulator with Coloured Petri nets. *International Journal on Software Tools for Technology Transfer*, 2(2):144–159, Dec. 1998.

[52] S. Gordon and J. Billington. Applying Coloured Petri nets and Design/CPN to an air-to-air missile simulator. In *Proceedings of the Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, pages 1–14, Aarhus, Denmark, 10-12 June 1998. Department of Computer Science, Aarhus University, PB-512.

[53] S. Gordon and J. Billington. Modelling and analysis of an air-to-air missile engagement simulator using Coloured Petri nets. In *Proceedings of the 3rd Biennial Engineering Mathematics and Applications Conference*, pages 225–228, Adelaide, Australia, 13-16 July 1998.

[54] S. Gordon and J. Billington. Modelling the WAP Transaction Service using Coloured Petri nets. In *Proceedings of the First International Conference on Mobile Data Access*, pages 105–114, Hong Kong, 16-17 Dec. 1999. Volume 1748 of Lecture Notes in Computer Science, Springer-Verlag.

[55] S. Gordon and J. Billington. WAP Forum Input Document: Inconsistencies in the Wireless Transaction Protocol. Submitted to the WAP Forum, 19 Mar. 1999.

[56] S. Gordon and J. Billington. Analysing the WAP class 2 Wireless Transaction Protocol using Coloured Petri nets. In *Proceedings of the 21st International Conference on Application and Theory of Petri Nets*, pages 207–226, Aarhus, Denmark, 26–30 June 2000. Volume 1825 of Lecture Notes in Computer Science, Springer-Verlag.

[57] S. Gordon and J. Billington. Coloured Petri net specification of the WAP Transaction protocol. Technical report, Computer Systems Engineering Centre, University of South Australia, Adelaide, Australia, Jan. 2001.

[58] S. Gordon, L. Kristensen, and J. Billington. An approach to generalising the state space of a distributed missile simulator. In *Proceedings of the Eleventh Annual International Symposium of The International Council on Systems Engineering*, Melbourne, Australia, 1-5 July 2001.

[59] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison Wesley, Reading, MA, 1996.

[60] R. Gotzhein. Specifying communication services with temporal logic. In *Protocol Specification, Testing and Verifiction, X*. North-Holland, Amsterdam, Oxford, 1990.

[61] R. Gotzhein, J. Bredereke, W. Effelsberg, S. Fischer, T. Held, and H. Koenig. Improving the efficiency of automated protocol implementation using Estelle. *Computer Communications*, 19(14):1226–1235, Dec. 1996.

[62] J. C. Gregoire. Engineering families of protocols using a formal language: Successes and failures. In *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pages 61–70, Barcelona, Spain, 1-3 Oct. 1997.

[63] GSM Association. GSM World. Web site: `http://www.gsmworld.com/` [Last accessed: 23 Nov. 2001], 3 Aug. 2001.

[64] B. Han and J. Billington. An analysis of TCP connection management using Colured Petri nets. In *Proceedings of the 5th World Multi-conference on Systematics, Cybernetics and Informatics*, pages 590–595, Orlando, FL, 22–25 July 2001.

[65] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session Initiation Protocol. IETF RFC 2543 URL: `ftp://ftp.isi.edu/in-notes/rfc2543.txt`, Mar. 1999.

[66] A. A. Hanish and T. S. Dillon. Communication protocol design to facilitate reuse based on the object-oriented paradigm. *Mobile Networks and Applications*, 2(3):285–301, Dec. 1997.

[67] P. G. Harrison and N. M. Patel. *Performance Modelling of Communication Networks and Computer Architecrures*. Addison-Wesley, Reading, MA, 1993.

[68] P. Herrmann and H. Krumm. Compositional specification and verification of high-speed transfer protocols. In *Protocol Specification, Testing and Verification, XIV*, pages 339–346. Chapman & Hall, London, UK, 1995.

[69] C. A. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[70] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[71] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[72] J. E. Hopcroft and J. D. Ullman. *Formal languages and their relation to automata.* Addison-Wesley, Reading, MA, 1969.

[73] P.-A. Hsiung and F. Wang. User-friendly verification. In *Formal Methods for Protocol Engineering and Distributed Systems.*, pages 279–294. Kluwer Academic Publishers, Boston Dordrecht London, 1999.

[74] P.-A. Hsiung, F. Wang, and R.-C. Chen. On the verification of Wireless Transaction Protocol using SGM and RED. In *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, pages 379–383, Cheju Island, South Korea, 12–14 Dec. 2000. IEEE Computer Society.

[75] InformationWeek.Com. I-mode wireless internet could be in US by 2002. Web site: `http://www.informationweek.com/story/IWK20010314S0004` [Last accessed: 23 Nov. 2001], 14 Mar. 2001.

[76] ISO. Information Processing Systems—Open Systems Interconnection—ESTELLE—a formal description technique based on an extended state transition model. ISO 9074 (Withdrawn 1999-05-06), 1989.

[77] ISO. Information Processing Systems—Open Systems Interconnection—LOTOS—a formal description technique based on the temporal ordering of observational behaviour. ISO 8807:1989, 1989.

[78] ISO/IEC. High-level Petri nets – concepts, definitions and graphical notation. ISO/IEC Final Draft International Standard 15909, May 2001.

[79] ITU. Information Technology—Open Systems Interconnection—Basic reference model: Conventions for the definition of OSI services. ITU-T Recommendation X.210, Nov. 1993. Also ISO/IEC 10731:1994.

[80] ITU. Information Technology—Open Systems Interconnection—Basic reference model: The basic model. ITU-T Recommendation X.200, July 1994. Also ISO/IEC 7498–1:1994.

[81] ITU. Information Technology—Open Systems Interconnection—Protocol for providing the connection-mode transport service. ITU-T Recommendation X.224, Nov. 1995. Also ISO/IEC 8073:1997.

[82] ITU. Information Technology—Open Systems Interconnection—Transport service definition. ITU-T Recommendation X.214, Nov. 1995. Also ISO/IEC 8072:1996.

[83] ITU. Information Technology—Open Systems Interconnection—Protocol for providing the connectionless-mode network service: Protocol specification. ITU-T Recommendation X.233, Aug. 1997. Also ISO/IEC 8473–1:1998.

[84] ITU. Specification and Description Language (SDL). ITU-T Recommendation Z.100, Nov. 1999.

[85] C. W. Janczura. *Modelling and Analysis of Railway Network Control Logic Using Coloured Petri Nets.* PhD thesis, School of Mathematics, University of South Australia, Adelaide, Australia, Aug. 1998.

[86] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1997.

[87] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1997.

[88] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1997.

[89] K. Jensen, S. Christensen, and L. M. Kristensen. *Design/CPN Occurrence Graph Manual, Version 3.0.* Department of Computer Science, Aarhus University, Aarhus, Denmark, 1996.

[90] J. B. Jørgensen and L. M. Kristensen. *Design/CPN OE/OS Graph Manual, Version 1.0.* Department of Computer Science, Aarhus University, Aarhus, Denmark, 1996.

[91] H. Kahlouche and J.-J. Girardot. Design of the ISO Class 0 Transport Protocol: A stepwise refinement based approach. In *Proceedings of the IEEE International Performance, Computing and Communications Conference*, pages 363–370, Phoenix, AZ, 5-7 Feb. 1997.

[92] Y. Kakuda, Y. Wakahara, and H. Saito. Component-based protocol synthesis. *Transactions of the Institute of Electronics and Communication Engineers of Japan, Part D*, J74D-I(6):369–378, June 1991.

[93] T. Kamada. Compact HTML for small information appliances. W3C Note `http://www.w3c.org/TR/1998/NOTE-compactHTML-19980209` [Last accessed: 23 Nov. 2001], 9 Feb. 1998.

[94] M. M. Khan. The development of personal communication services under the auspices of existing network technologies. *IEEE Communications Magazine*, 35(3):78–82, Mar. 1997.

[95] P. King and T. Hyland. Handheld Device Markup Language Specification. W3C Note `http://www.w3c.org/TR/NOTE-Submission-HDML-Spec.html` [Last accessed: 23 Nov. 2001], 9 May 1997.

[96] L. M. Kristensen and A. Valmari. Finding stubborn sets of coloured petri nets without unfolding. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, pages 104–123, Lisbon, Portugal, June 1998. Volume 1420 of Lecture Notes in Computer Science, Springer-Verlag.

[97] S. S. Lam and A. U. Shankar. A relational notation for state transition systems. *IEEE Transactions on Software Engineering*, 16(7):755–775, July 1990.

[98] P. Langendoerfer and H. Koenig. COCOS—a configurable SDL compiler for generating efficient protocol implementations. In *SDL'99. The Next Millenium*, pages 259–274. Elsevier Science Publishers, Amsterdam, Netherlands, 1999.

[99] J.-K. Lee and K.-H. Lee. Modelling of the multicast transport protocols using Petri nets. In *Proceedings of the IEEE Singapore International Conference on Networks*, pages 106–110, Singapore, 3–7 July 1995.

[100] G. A. Lewis and C. A. Lakos. Incremental state space construction for Coloured Petri nets. In *Proceedings of the 22nd International Conference on Application and Theory of Petri Nets*, pages 263–282, Newcastle upon Tyne, UK, 25–29 June 2001. Volume 2075 of Lecture Notes in Computer Science, Springer-Verlag.

[101] B. Lindstrøm and L. Wells. *Design/CPN Performance Tool Manual, Version 1.0.* Department of Computer Science, Aarhus University, Aarhus, Denmark, Sept. 1999.

[102] R. J. Linn Jr. Conformance evaluation methodology and protocol testing. *IEEE Journal on Selected Areas on Communications*, 7(7):1143–1158, Sept. 1989.

[103] M. T. Liu. Protocol engineering. In *Advances in Computers, Vol. 27*, pages 79–195. Academic, New York, NY, 1989.

[104] M. T. Liu. Special issue on protocol engineering. *IEEE Transactions on Computers*, 40(4), Apr. 1991.

[105] G. M. Lundy and R. C. McArthur. Formal model of a high speed transport protocol. In *Protocol Specification, Testing and Verification, XII*, pages 97–111, Lake Buena Vista, FL, 22-25 June 1992. North-Holland.

[106] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceshinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley & Sons, New York, 1995.

[107] H. Mehrpour and A. E. Karbowiak. Modelling and analysis of DOD TCP/IP protocol using numerical Petri nets. In *Proceedings of IEEE TENCON'90: IEEE Region 10 Conference on Computer Communication Systems*, pages 617–622, Hong Kong,, 24-27 Sept. 1990.

[108] A. Menezes. *Elliptic Curve Public Key Cryptosystems.* Kluwer Academic Publishers, Boston, 1993.

[109] Meta Software Corporation. *Design/CPN Reference Manual for X-Windows, Version 2.0.* Meta Software Corporation, Cambridge, MA, 1993.

[110] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1980.

[111] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML.* The MIT Press, revised edition, 1997.

[112] Mobile Media Japan. i-mode Java specifications. Web site: `http://www.mobilemediajapan.com/newsdesk/imode-java.html` [Last accessed: 23 Nov. 2001], 2001.

[113] K. H. Mortensen. Automatic code generation method based on Coloured Petri net models applied on an access control systems. In *Proceedings of the 21st International Conference on Application and Theory of Petri Nets*, pages 367–386, Aarhus, Denmark, 26–30 June 2000. Volume 1825 of Lecture Notes in Computer Science, Springer-Verlag.

[114] Motorola. Flex Technologies. Web site: `http://www.motorola.com/FLEX/` [Last accessed: 23 Nov. 2001].

[115] Motorola. iDEN. Web site: `http://www.motorola.com/iden/` [Last accessed: 23 Nov. 2001].

[116] M. Mouly and M.-B. Pautet. *The GSM System for Mobile Communications.* M. Mouly & Marie-B Pautet, Palaiseau, France, 1992.

[117] A. Murase, A. Maebara, I. Okajima, and S. Hirata. Mobile radio packet data communications in a TDMA digital cellular system. In *Proceedings of the 47th IEEE Vehicular Technology Conference*, pages 1034–1038 Vol. 2, 4–7 May 1997.

[118] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[119] M. Nakamura, Y. Kakuda, and T. Kikuno. On constructing communication protocols from component-based service specifications. *Computer Communications*, 19(14):1200–1215, Dec. 1996.

[120] S. C. Nash. Format and protocol language (FAPL). *Computer Networks and ISDN Systems*, 14(1):61–77, 1987.

[121] T. Natsuno. DoCoMo's i-mode: Toward mobile multimedia in 3G. In *Proceedings of the 47th Internet Engineering Task Force Meeting*, Plenary Session, Adelaide, Australia, 26–31 Mar. 2000. IETF.

[122] B. Neelakantan and S. V. Raghavan. Protocol conformance testing—a survey. In *Computer Networks, Architecture and Applications*, pages 175–191. Chapman & Hall, London, UK, 1995.

[123] NTT DoCoMo. DoCoMoNet. Web site: `http://www.nttdocomo.com/` [Last accessed: 23 Nov. 2001].

[124] G. Peersman, P. Griffiths, H. Spear, S. Cvetkovic, and C. Smythe. A tutorial overview of the Short Message Service within GSM. *Computing & Control Engineering Journal*, 11(2):79–89, Apr. 2000.

[125] C. E. Perkins. IP mobility support. IETF RFC 2002 URL: `ftp://ftp.isi.edu/in-notes/rfc2002.txt`, Oct. 1996.

[126] C. E. Perkins. *Mobile IP: Design Principles and Practices.* Addison-Wesley, Reading, MA, 1998.

[127] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[128] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut fur Intrumentelle Mathematik, Schriften des IIM Nr. 2, Second Edition, Bonn, 1962.

[129] PHS MoU Group. General description of public Personal Handyphone System. Technical Specification A-GN0.00-01-TS, Apr. 1997.

[130] PHS MoU Group. Personal Handyphone System Memorandum of Understanding Group. Web site: `http://www.phsmou.or.jp/` [Last accessed: 23 Nov. 2001], 23 July 2001.

[131] J. Postel. User Datagram Protocol. IETF RFC 768 URL: `ftp://ftp.isi.edu/in-notes/rfc768.txt`, 28 Aug. 1980.

[132] J. Postel. Internet Protocol. DARPA Internet Program Protocol Specification. Information Sciences Institute, University of Southern California, Marina del Ray, CA, Sept. 1981. IETF RFC 791 URL: `ftp://ftp.isi.edu/in-notes/rfc791.txt`.

[133] J. Postel. Transmission Control Protocol. DARPA Internet Program Protocol Specification. Information Sciences Institute, University of Southern California, Marina del Ray, CA, Sept. 1981. IETF RFC 793 URL: `ftp://ftp.isi.edu/in-notes/rfc793.txt`.

[134] Proceedings of the International Conferences on Application and Theory of Petri Nets. Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1993–2000.

[135] Proceedings of the IFIP WG6.1 International Workshops/Symposiums on Protocol Specification, Testing and Verification (PSTV). Vol. II (1983) to XIII (1993): North-Holland, Amsterdam Oxford. Vol. XIV (1994) to XVII (1997): Chapman & Hall, London, UK. Vol. XVIII (1998) to XX (2000): Kluwer Academic Publishers, Boston Dordrecht London. Vol. XVI to XX are joint proceedings with Formal Description Techniques (FORTE).

[136] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1985.

[137] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets: Advances in Petri Nets. Volume I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

[138] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets: Advances in Petri Nets. Volume II: Applications*, volume 1492 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

[139] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 26(1):96–99, Jan. 1983.

[140] T. G. Robertazzi. *Computer Networks and Systems: Queuing Theory and Performance Evaluation*. Springer-Verlag, New York, 2nd edition, 1994.

[141] K. Saleh. Special issue on protocol engineering. *Computer Communications*, 19(14), Dec. 1996.

[142] K. Saleh. Synthesis of communications protocols: An annotated bibliography. *Computer Communication Review*, 26(5):40–59, Oct. 1996.

[143] A. K. Salkantzis and C. Chamzas. Mobile packet data technology: An insight into MOBITEX architecture. *IEEE Personal Communications Magazine*, 4(1):10–18, Jan. 1997.

[144] A. K. Salkintzis. Packet data over cellular networks: the CDPD approach. *IEEE Communications Magazine*, 37(6):152–159, June 1999.

[145] A. K. Salkintzis. A survey of mobile data networks. *IEEE Communications Surveys*, 2(3):2–18, Third Quarter 1999.

[146] A. K. Salkintzis, C. Chamzas, and P. T. Mathiopoulos. Special issue on mobile data networks: Advanced technologies and services. *Mobile Networks and Applications*, 5(1), 2000.

[147] M. J. Sanders and K. R. Parker. Modelling a low earth orbit satellite protocol with a new object-oriented Petri net language. Technical report, Department of Digital Systems, Monash University, Melbourne, Australia, 1997.

[148] D. Sheppard. *An Introduction to Formal Specification with Z and VDM*. The McGraw-Hill International Series in Software Engineering. McGraw-Hill, Boston, MA, 1995.

[149] M. A. Smith. Formal verification of communication protocols. In *Formal Description Techniques IX: Theory, Applications, and Tools*, pages 129–144. Chapman & Hall, London, UK, Oct. 1996.

[150] M. A. Smith. Reliable message delivery and conditionally-fast transactions are not possible without accurate clocks. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 163–171, Puerto Vallarto, Mexico, June 1998.

[151] I. Sommerville. *Software Engineering*. Addison Wesley, Reading, MA, fourth edition, 1992.

[152] Special issue on the rise of protocol engineering. *IEEE Software*, 9(1), Jan. 1992.

[153] ST Mobile Data Pte Ltd. Web site: `http://www.stmd.st.com.sg/` [Last accessed: 23 Nov. 2001].

[154] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, NJ, third edition, 1996.

[155] TechWeb. i-mode: Coming soon to phones worldwide? Web site: `http://content.techweb.com/wire/story/TWB20010118S0010` [Last accessed: 23 Nov. 2001], 18 Jan. 2001.

[156] Telecommunications Industry Assoication. TDMA Cellular PCS. TIA/EIA-136, Mar. 1999.

[157] The Open Group. WAP Certification. Web site: `http://www.opengroup.org/wap/cert/` [Last accessed: 23 Nov. 2001], 18 July 2001.

[158] A. A. Tokmakoff. *Modelling, Analysis and Prototyping of the ODP Trader Using Coloured Petri Nets and Java*. PhD thesis, School of Physics and Electronic Systems Engineering, University of South Australia, Adelaide, Australia, Mar. 1998.

[159] J. D. Ullman. *Elements of ML Programming*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[160] United Wireless. MOBITEX—the intelligent network. Web site: `http://www.uw.com.au/` [Last accessed: 23 Nov. 2001].

[161] University of Aarhus. Design/CPN Online. Web site: `http://www.daimi.au.dk/designCPN/` [Last accessed: 23 Nov. 2001], 17 June 2000.

[162] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 9th European Workshop on Application and Theory of Petri Nets*, Venice, Italy, 1990. Volume 424 of Lecture Notes in Computer Science, Springer-Verlag.

[163] A. Valmari. The state explosion problem. In *Lectures on Petri Nets: Advances in Petri Nets. Volume II: Applications*, pages 429–528. Springer-Verlag, Berlin, 1998.

[164] A. Valmari and M. Tienari. Compositional failure-based semantic models for basic LOTOS. *Formal Aspects of Computing*, 7(4):440–468, 1995.

[165] M. Villapol and J. Billington. Modelling and initial analysis of the Resource Reservation Protocol using Coloured Petri nets. In *Proceedings of Workshop on Practical Use of High-level Petri Nets*, pages 91–110, Aarhus, Denmark, 26–30 June 2000. Department of Computer Science, Aarhus University.

[166] C. Vissers and L. Logrippo. The importance of the service concept in the design of data communication protocols. In *Protocol Specification, Testing and Verification*, V, pages 3–17. North Holland, 1986.

[167] G. von Bochmann and C. A. Sunshine. Formal methods for protocol specification and validation. In *Computer Network Architectures and Protocols*, pages 513–531. Plenum Press, NY, 2nd edition, 1989.

[168] B. Wang and D. Hutchison. Protocol testing techniques. *Computer Communications*, 10(2):79–87, Apr. 1987.

[169] J. Wang. *Timed Petri Nets, Theory and Application*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.

[170] WAP Forum. Wireless Application Protocol. Web site: `http://www.wapforum.org/` [Last accessed: 23 Nov. 2001].

[171] WAP Forum. WAP Architecture Specification. June 2000 Conformance Release. Available via: `http://www.wapforum.org/`, 30 Apr. 1998.

[172] WAP Forum. WAP Wireless Transaction Protocol Specification. Version 1.0. Available via: `http://www.wapforum.org/`, 30 Apr. 1998.

[173] WAP Forum. WAP Push Architectural Overview. June 2000 Conformance Release. Available via: `http://www.wapforum.org/`, 8 Nov. 1999.

[174] WAP Forum. WAP Wireless Transaction Protocol Specification. Available via: `http://www.wapforum.org/`, 11 June 1999.

[175] WAP Forum. Wireless Application Group User Agent Profile Specification. June 2000 Conformance Release. Available via: `http://www.wapforum.org/`, 10 Nov. 1999.

[176] WAP Forum. WAP Conformance Process and Certification Policy. WAP-216-CertPolicy. Available via: `http://www.wapforum.org/`, 14 Nov. 2000.

[177] WAP Forum. WAP Specification Information Note for Wireless Transaction Protocol Specification. June 2000 Conformance Release. Available via: `http://www.wapforum.org/`, 12 Dec. 2000.

[178] WAP Forum. WAP Wireless Application Environment Specification Version 1.3. June 2000 Conformance Release. Available via: `http://www.wapforum.org/`, 29 Mar. 2000.

[179] WAP Forum. WAP Wireless Datagram Protocol Specification. June 2000 Conformance Release. Available via: http://www.wapforum.org/, 19 Feb. 2000.

[180] WAP Forum. WAP Wireless Markup Language Specification Version 1.3. June 2000 Conformance Release. Available via: http://www.wapforum.org/, 19 Feb. 2000.

[181] WAP Forum. WAP Wireless Session Protocol Specification. June 2000 Conformance Release. Available via: http://www.wapforum.org/, 4 May 2000.

[182] WAP Forum. WAP Wireless Telephony Application Specification. June 2000 Conformance Release. Available via: http://www.wapforum.org/, 7 July 2000.

[183] WAP Forum. WAP Wireless Transaction Protocol Specification. June 2000 Conformance Release. Available via: http://www.wapforum.org/, 19 Feb. 2000.

[184] WAP Forum. WAP Wireless Transport Layer Security Specification. June 2000 Conformance Release. Available via: http://www.wapforum.org/, 18 Feb. 2000.

[185] WAP Forum. WAP WMLScript Language Specification Version 1.2. June 2000 Conformance Release. Available via: http://www.wapforum.org/, 24 Mar. 2000.

[186] WAP Forum. WAP Architecture Specification. Version 2.0. Available via: http://www.wapforum.org/, 12 July 2001.

[187] WAP Forum. WAP Wireless Transaction Protocol Specification. Version 2.0. Available via: http://www.wapforum.org/, 10 July 2001.

[188] G. R. Wheeler. Numerical Petri Nets: A definition. Technical report, Telecom Australia, Research Laboratory Report 7780, May 1985.

[189] World Wide Web Consortium. HTML 4.0 specification. W3C Recommendation. Available via: http://www.w3.org/ [Last accessed: 23 Nov. 2001], 18 Dec. 1997.

[190] World Wide Web Consortium. XHTML 1.0: The Extensible HyperText Markup Language. W3C Recommendation: http://www.w3c.org/tr/xhtml1/ [Last accessed: 23 Nov. 2001], 26 Jan. 2000.

[191] XTP Forum. Xpress Transport Protocol Specification. XTP Revision 4.0, XTP Forum, Santa Barbara, CA, Mar. 1995.

Appendix A

Finite State Automata

Verifying the Transaction Protocol involves checking if it contains the same set of global primitive sequences as the Transaction Service. Chapter 4 described the relationship between CPNs (and their state space) and languages. This appendix describes the practical steps of obtaining the FSA from the CPN models (Section A.1), minimizing the FSA and comparing two FSAs (Section A.2).

A.1 Finite State Automata and State Spaces

A state space is a directed graph with nodes representing the state of the system, and arcs representing the changes in state. For Coloured Petri nets, the nodes also correspond to markings of the net and the arcs correspond to binding elements. A state space can be treated as a deterministic FSA, where the set of binding elements is the input alphabet. The purpose of doing so is to take advantage of well-known techniques [72] for comparing FSAs (or languages). These techniques will become apparent in Section A.2.

In general, it is not necessary for a one-to-one mapping of binding elements to symbols of the alphabet. For example, the TR-Protocol maps binding elements that correspond to service primitives to symbols representing those primitives (allowing many binding elements to map to one symbol), and maps all other (non-primitive) binding elements to the empty string, ϵ. As a result, the FSA is non-deterministic.

Design/CPN [109] is used for creating the state spaces, whereas FSM [4] is used for the FSA and language analysis. Therefore, Standard ML functions are used in Design/CPN to write the state space to a text file in a format suitable for use by FSM. This process includes the mapping of binding elements to symbols. The format used by FSM is of the form:

```
src_node_1 dest_node_1 transition_1
src_node_2 dest_node_2 transition_2
```

```
...
src_node_n dest_node_n transition_n
halt_node_1
...
halt_node_m
```

where `src_node_1` is the initial or starting node in the FSA, and nodes and transitions are given as integers. A transition value of `0` represents an empty string or transition (ϵ-transition).

Listing A.1 gives the two functions used for writing the Design/CPN state space to text files. og2fsmtrans() writes binding elements, line by line, to a text file. This function can be applied to any state space. A function given as an argument performs the mapping from arcs in a specific state space to symbols or transitions in the FSA. For example, Listing 6.2 gives the mapping specification for the TR-Service and Listing D.4 gives the mapping specification for the TR-Protocol. The output file name is also given as an argument.

Listing A.1: Standard ML code to convert a Design/CPN state space to FSM text format

```
1  (* write OG to text file in format: srcnodeno destnodeno primitiveno *)
2  (* (Arc -> string) -> string -> unit *)
3  fun og2fsmtrans arc2fsm filename =
4  let
5          val outfile = open_out(filename)
6          fun getfsminput (a) = output( outfile ,makestring(SourceNode(a))^" "^
7                  makestring(DestNode(a))^" "^arc2fsm(a) ^ "\n")
8  in
9          EvalAllArcs(getfsminput);
10         close_out ( outfile )
11 end;
12
13 (* calculate halt states for FSM and write to text file , one per line *)
14 (* (Node -> bool) -> string -> unit *)
15 fun og2fsmhalts findhalts filename =
16 let
17         val outfile = open_out(filename)
18         fun WriteNodes(n)=output(outfile,st_Node(n) ^ "\n")
19 in
20         SearchAllNodes( findhalts ,fn n => WriteNodes(n),[],op ::);
21         close_out ( outfile )
22 end;
```

The second function in Listing A.1, og2fsmhalts(), writes all halt nodes, one per line, to a text file. Again, a predicate for a specific state space is used to determine which nodes are halt states. For example, FindHalts() in Listing 6.2 returns true when a node in the TR-Service state space is classified as a halt state.

Listing A.2 shows a script used in minimising the TR-Protocol FSA and comparing it to the TR-Service FSA. Many details of this script are not discussed—it is included to assist other Design/CPN and FSM users.

The text files containing the transitions and halts of the FSA can be concatenated (Line 25 of Listing A.2), and used as an input to FSM. The text file is compiled by FSM into a binary format (Line 41).

Listing A.2: Shell script to minimize a Revised TR-Protocol configuration using FSM

```
1  #!/bin/sh
2  if test $3
3  then echo 'Processing OG Results ...';
4  else echo 'Usage: minog transfile halts file on|off '; exit;
5  fi
6
7  trans=$1
8  halts=$2
9  userack=$3
10
11 # Setup the directories
12  thesisdir =/home/sgordon/thesis/
13  servicedir =$thesisdir 'cpn/service /'
14  protocoldir =$thesisdir 'cpn/wtp/protocol/'
15  labels =$thesisdir 'cpn/service /symbols.txt'
16 case $userack in
17 on) servicefsm =$servicedir 'uack/ogmin.fsm' ;;
18  off ) servicefsm =$servicedir 'nouack/ogmin.fsm' ;;
19 esac
20
21 # Setup OG from Design/CPN output for FSM input
22 # trans file doenst exist , then try and use its gzipped counterpart
23 if test -e $trans
24 then
25    cat $trans $halts > og.txt
26 else
27    tmpgzip=`basename $trans .txt`
28    if test -e $tmpgzip'.txt.gz'
29    then
30       gunzip $tmpgzip'.txt.gz'
31       cat $trans $halts > og.txt
32    else
33       echo 'Error : ' $trans 'does not exist .';
34       exit;
35    fi
36 fi
37 gzip $trans
38 echo '    Minimizing ...'
39
40 echo '       Compiling ...'
41 fsmcompile og.txt > og.fsm
42 if test -f og.fsm
```

```
43 then rm og.txt;
44 else echo 'Error: fsm compilation failed .'; exit;
45 fi
46
47 echo '       Removing empties...'
48 fsmrmepsilon og.fsm > ognoeps.fsm
49 if test -f ognoeps.fsm
50 then rm og.fsm;
51 else echo 'Error: removal of epsilons failed .'; exit;
52 fi
53
54 echo '       Determinizing ...'
55 fsmdeterminize ognoeps.fsm > ogdeterm.fsm
56 if test -f ogdeterm.fsm
57 then rm ognoeps.fsm;
58 else echo 'Error: fsm determinization failed .'; exit;
59 fi
60
61 echo '       Minimizing ...'
62 fsmminimize ogdeterm.fsm > ogmin.fsm
63 if test -f ogmin.fsm
64 then rm ogdeterm.fsm;
65 else echo 'Error: fsm minimization failed .'; exit;
66 fi
67
68 fsminfo -n ogmin.fsm > info.txt
69
70 echo '   Processing language ...'
71 # Obtain language (if possible)
72 lexfsmstrings -l $labels ogmin.fsm > lang.txt
73
74 # Graphical form
75 fsmdraw -i $labels ogmin.fsm > ogdraw.txt
76 dot -Tps -o ogdraw.ps ogdraw.txt
77
78 # Comparisons
79 fsmdifference ogmin.fsm $servicefsm > diff-service.fsm
80 fsmdifference $servicefsm ogmin.fsm > diff-protocol.fsm
81 lexfsmstrings -l $labels diff-service.fsm > difflang-service.txt
82 lexfsmstrings -l $labels diff-protocol.fsm > difflang-protocol.txt
83
84 echo 'Complete.'
```

A.2 FSA Minimization and Comparison

The purpose of treating the state space as a FSA (e.g. the TR-Protocol FSA) is that it can be compared to another FSA (e.g. TR-Service FSA) and determined if the two contain the same language (e.g. the sequences of primitives defined by the TR-Service are preserved by the TR-Protocol). This is based on the fact that any non-deterministic

FSA with ϵ-transitions can be converted into a canonical form, a minimized deterministic FSA [9]. If the canonical forms of two FSAs are isomorphic, then they have identical languages. The steps for minimizing a FSA are [9]:

1. Remove all ϵ-transitions from the FSA using the rule: a transition from node 1 to node 2 on input a, followed by an ϵ-transition from node 2 to node 3, is equivalent to a transition from node 1 to node 3 on input a. The command `fsmrmepsilon` is used in FSM to perform this step (Line 48 of Listing A.2).

2. Convert the non-deterministic FSA into a deterministic FSA. This can be achieved by: identifying nodes that are the destination of a single source node on the input of one symbol; creating a new node based on these destination nodes; and then removing the old destination nodes. Eventually, the non-determinism will be removed. Any inaccessible states as a result of this process can also be removed. `fsmdeterminize` performs these functions in FSM (Line 55 of Listing A.2).

3. Minimize the deterministic FSA to its canonical form by identifying and merging equivalent states. For example, this can be done by finding states that will be indistinguishable if they were merged, and then merge them, deleting the component states. The command `fsmminimize` performs the minimization in FSM (Line 62 of Listing A.2).

Once two FSA have been minimized they can be either directly compared (for example, `fsmdifference`, Line 79 of Listing A.2, returns an FSA of the differences), or their languages compared. The complete language of a FSA produced in FSM can be written to a text file using `lexfsmstrings` (Line 72 of Listing A.2), which is part of LexTools [6]. LexTools and GraphViz [5] include a set of commands for manipulating FSAs into languages and graphical representations (e.g. PostScript). The reader is referred to the corresponding web sites of FSM [4], LexTools [6] and GraphViz [5] for further information.

Appendix B

Transaction Protocol State Tables

The operation of the Transaction Protocol (TR-Protocol) is described mainly using state tables. These are given in Section 10.5 (WTP Initiator) and Section 10.6 (WTP Responder) of the WTP Specification [183]. In Chapter 5 we summarize the structure of the state tables. This appendix gives the state tables in full. Although only Transaction Class 2 is modelled and analysed in this thesis, the state table entries for all classes (0, 1, and 2) are shown. The only changes made are the correcting of typographical errors. We have added numbers to the left of each Class 2 entry so they can be referred to easily.

	Event	Condition	Action	Next State
1	TR-Invoke.req	Class == 2 | 1	SendTID = GenTID Send Invoke PDU Reset RCR Start timer, R[RCR] Uack = False	RESULT WAIT
2		Class == 2 | 1 UserAck	SendTID = GenTID Send Invoke PDU Reset RCR Start timer, R[RCR] Uack = True	
		Class == 0	SendTID = GenTID Send Invoke PDU	NULL

Table B.1: TR-Protocol state table: TR-Init-PE NULL

	Event	Condition	Action	Next State
1	TR-Abort.req		Abort transaction Send Abort PDU (USER)	NULL
2	RcvAck	Class == 2	Stop timer Generate TR-Invoke.cnf HoldOn = True	RESULT WAIT
		Class == 1	Stop timer Generate TR-Invoke.cnf	NULL
3		TIDve Class == 2 | 1	SendAck(TIDok) Increment RCR Start timer, R[RCR]	RESULT WAIT
4	RcvAbort		Abort transaction Generate TR-Abort.ind	NULL
5	RcvErrorPDU		Abort transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	NULL
6	TimerTO_R	RCR< MAX_RCR	Increment RCR Start timer, R[RCR] Send Invoke PDU	RESULT WAIT
7		RCR< MAX_RCR, Ack(TIDok) already sent	Increment RCR Start timer, R[RCR] Send Ack(TIDok)	RESULT WAIT
8		RCR == MAX_RCR	Abort transaction Generate TR-Abort.ind	NULL
9	RcvResult	Class == 2 HoldOn == True	Stop timer Generate TR-Result.ind Start timer, A	RESULT RESP WAIT
10		Class == 2 HoldOn == False	Stop timer Generate TR-Invoke.cnf Generate TR-Result.ind Start timer, A	

Table B.2: TR-Protocol state table: TR-Init-PE RESULT WAIT

	Event	Condition	Action	Next State
1	TR-Result.res		Queue(A) Ack PDU Start timer, W	WAIT TIMEOUT
2		ExitInfo	Queue(A) Ack PDU with Info TPI Start timer, W	
3	RcvAbort		Abort transaction Generate TR-Abort.ind	NULL
4	TR-Abort.req		Abort transaction Send Abort PDU (USER)	
5	RcvErrorPDU		Abort transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	NULL
6	RcvResult		Ignore	RESULT RESP WAIT
7	TimerTO_A	AEC< AEC_MAX	Increment AEC Start timer, A	RESULT RESP WAIT
8		AEC == AEC_MAX	Abort transaction Send Abort PDU (NORESPONSE)	NULL
9		Uack == False	Queue(A) Ack PDU Start timer, W	WAIT TIMEOUT

Table B.3: TR-Protocol state table: TR-Init-PE RESULT RESP WAIT

	Event	Condition	Action	Next State
1	RcvResult	RID=0	Ignore	WAIT TIMEOUT
2	RcvResult	RID=1	Send Ack PDU	WAIT TIMEOUT
3	RcvResult	RID=1,ExitInfo	Send Ack PDU with info TPI	WAIT TIMEOUT
4	RcvAbort		Abort transaction Generate TR-Abort.ind	NULL
5	RcvErrorPDU		Abort transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	NULL
6	TimerTO_W		Clear transaction	NULL
7	TR-Abort.req		Abort transaction Send Abort PDU (USER)	NULL

Table B.4: TR-Protocol state table: TR-Init-PE WAIT TIMEOUT

	Event	Condition	Action	Next State
1	RcvInvoke	Class == 2 | 1 Valid TID U/P flag	Generate TR-Invoke.ind Start timer, A Uack = True	INVOKE RESP WAIT
2		Class = 2 | 1 Valid TID	Generate TR-Invoke.ind Start timer, A Uack = False	
		Class == 0	Generate TR-Invoke.ind	LISTEN
3		Class == 2 | 1 Invalid TID	Send Ack(TIDve)	TIDOK WAIT
4	RcvErrorPDU		Send Abort PDU (PROTOERR)	LISTEN

Table B.5: TR-Protocol state table: TR-Resp-PE LISTEN

	Event	Condition	Action	Next State
1	RcvAck	Class == 2 | 1 TIDok	Generate TR-Invoke.ind Start timer, A	INVOKE RESP WAIT
2	RcvErrorPDU		Send Abort PDU (PROTOERR) Abort transaction	LISTEN
3	RcvAbort		Abort transaction	LISTEN
4	RcvInvoke	RID=0	Ignore	TIDOK WAIT
5		RID=1	Send Ack(TIDve)	TIDOK WAIT

Table B.6: TR-Protocol state table: TR-Resp-PE TIDOK WAIT

	Event	Condition	Action	Next State
	TR-Invoke.res	Class == 1 ExitInfo	Queue(A) Ack PDU with InfoTPI Start timer, W	WAIT TIMEOUT
		Class == 1	Queue(A) Ack PDU Start timer, W	
1		Class == 2	Start timer, A	RESULT WAIT
2	TR-Result.req		Reset RCR Start timer, R[RCR] Send Result PDU	RESULT RESP WAIT
3	TR-Abort.req		Abort transaction Send Abort PDU (USER)	LISTEN
4	RcvAbort		Generate TR-Abort.ind Abort transaction	LISTEN
5	RcvInvoke		Ignore	INVOKE RESP WAIT
6	RcvErrorPDU		Abort transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	LISTEN
7	TimerTO_A	AEC< AEC_MAX	Increment AEC Start timer, A	INVOKE RESP WAIT
8		AEC == AEC_MAX	Abort transaction Send Abort PDU (NORESPONSE)	LISTEN
		Class == 1 Uack == False	Queue(A) Ack PDU Start timer, W	WAIT TIMEOUT
9		Class == 2 Uack == False	Send Ack PDU	RESULT WAIT

Table B.7: TR-Protocol state table: TR-Resp-PE INVOKE RESP WAIT

	Event	Condition	Action	Next State
1	TR-Result.req		Reset RCR Start timer, R[RCR] Send Result PDU	RESULT RESP WAIT
2	RcvInvoke	RID=0	Ignore	RESULT WAIT
3		RID=1	Ignore	RESULT WAIT
4		RID=1, Ack PDU already sent	Resend Ack PDU	RESULT WAIT
5	RcvErrorPDU		Abort transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	LISTEN
6	TR-Abort.req		Abort transaction Send Abort PDU (USER)	LISTEN
7	RcvAbort		Generate TR-Abort.ind Abort transaction	LISTEN
8	TimerTO_A		Send Ack PDU	RESULT WAIT

Table B.8: TR-Protocol state table: TR-Resp-PE RESULT WAIT

	Event	Condition	Action	Next State
1	TR-Abort.req		Abort transaction Send Abort PDU (USER)	LISTEN
2	RcvAbort		Generate TR-Abort.ind Abort transaction	LISTEN
3	RcvAck		Generate TR-Result.cnf	LISTEN
4	RcvErrorPDU		Abort transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	LISTEN
5	TimerTO_R	RCR< RCR_MAX	Increment RCR Send Result PDU Start timer, R[RCR]	RESULT RESP WAIT
6		RCR == RCR_MAX	Generate TR-Abort.ind Abort transaction	LISTEN

Table B.9: TR-Protocol state table: TR-Resp-PE RESULT RESP WAIT

Event	Condition	Action	Next State
RcvInvoke	RID=0	Ignore	WAIT TIMEOUT
RcvInvoke	RID=1	Send Ack PDU	WAIT TIMEOUT
RcvInvoke	RID=1, ExitInfo	Send Ack PDU with Info TPI	WAIT TIMEOUT
RcvErrorPDU		Abort transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	LISTEN
RcvAbort		Abort transaction Generate TR-Abort.ind	LISTEN
TimerTO_W		Clear transaction	LISTEN
TR-Abort.req		Abort transaction Send Abort PDU (USER)	LISTEN

Table B.10: TR-Protocol state table: TR-Resp-PE WAIT TIMEOUT (Class 1 Only)

Appendix C

Transaction Service State Space Reports

Chapter 6 describes the TR-Service CPN and discusses the analysis results. The analysis includes generation of the TR-Service state space when UserAck is On and Off. Design/CPN produces a report on the state space analysis, summarizing the statistics and properties. The reports for both TR-Service state spaces are given in Listings C.1 (UserAck On) and C.2 (UserAck Off).

Listing C.1: Report on the TR-Service state space produced by Design/CPN when UserAck is On

```
1  Statistics
2 ------------------------------------
3  Occurrence Graph
4    Nodes:  57
5    Arcs:   114
6    Secs:   0
7    Status: Full
8
9  Scc Graph
10   Nodes:  57
11   Arcs:   114
12   Secs:   0
13
14 Boundedness Properties
15 ------------------------------------
16  Best Integers Bounds          Upper      Lower
17  InvokeResult'AckType 1          1          0
18  InvokeResult'InitToResp 1       2          0
19  InvokeResult' Initiator 1       1          1
20  InvokeResult'RespToInit 1       3          0
21  InvokeResult'Responder 1        1          1
22
23  Best Upper Multi-set Bounds
24  InvokeResult'AckType 1
25        1`On
```

```
 InvokeResult'InitToResp 1
        1`Invoke++ 1`Ack++ 1`Abort
 InvokeResult' Initiator 1
        1`Uinvoke++ 1`Pinvokeack++ 1`Presult++ 1`Uresultack++ 1`Icomplete
 InvokeResult'RespToInit 1
        1`Result++ 1`Ack++ 1`NoAck++ 1`Abort
 InvokeResult'Responder 1
        1`Pinvoke++ 1`Uinvokeack++ 1`Uresult++ 1`Presultack++ 1`Rcomplete

 Best Lower Multi-set Bounds
 InvokeResult'AckType 1          empty
 InvokeResult'InitToResp 1       empty
 InvokeResult' Initiator 1       empty
 InvokeResult'RespToInit 1       empty
 InvokeResult'Responder 1        empty

Home Properties
--------------------------------
 Home Markings: None

Liveness Properties
--------------------------------
 Dead Markings:   22 [57,56,55,54,53,...]
 Dead Transitions Instances: None
 Live Transitions Instances: None

Fairness Properties
--------------------------------
 No infinite occurrence sequences.
```

Listing C.2: Report on the TR-Service state space produced by Design/CPN when User-Ack is Off

```
Statistics
--------------------------------
 Occurrence Graph
   Nodes: 60
   Arcs:  129
   Secs:  0
   Status: Full

 Scc Graph
   Nodes: 60
   Arcs:  129
   Secs:  0

Boundedness Properties
--------------------------------
 Best Integers Bounds          Upper      Lower
 InvokeResult'AckType 1          1          0
 InvokeResult'InitToResp 1       2          0
 InvokeResult' Initiator 1       1          1
 InvokeResult'RespToInit 1       3          0
```

```
21  InvokeResult'Responder 1          1              1
22
23  Best Upper Multi-set Bounds
24  InvokeResult'AckType 1
25        1`Off
26  InvokeResult'InitToResp 1
27        1`Invoke++ 1`Ack++ 1`Abort
28  InvokeResult' Initiator 1
29        1`Uinvoke++ 1`Pinvokeack++ 1`Presult++ 1`Uresultack++ 1`Icomplete
30  InvokeResult'RespToInit 1
31        1`Result++ 1`Ack++ 1`NoAck++ 1`Abort
32  InvokeResult'Responder 1
33        1`Pinvoke++ 1`Uinvokeack++ 1`Uresult++ 1`Presultack++ 1`Rcomplete
34
35  Best Lower Multi-set Bounds
36  InvokeResult'AckType 1          empty
37  InvokeResult'InitToResp 1       empty
38  InvokeResult' Initiator 1       empty
39  InvokeResult'RespToInit 1       empty
40  InvokeResult'Responder 1        empty
41
42  Home Properties
43 ------------------------------------
44  Home Markings: None
45
46  Liveness Properties
47 ------------------------------------
48  Dead Markings:   22 [60,59,58,57,54,...]
49  Dead Transitions Instances: None
50  Live Transitions Instances: None
51
52  Fairness Properties
53 ------------------------------------
54  No infinite occurrence sequences.
```

Appendix D

Transaction Protocol CPN and Results

Chapter 7 gives a detailed description of the TR-Protocol CPN. Analysis of the CPN model is presented in Chapter 8. This appendix contains supporting material for the model and analysis results. For completeness, all declarations for the TR-Protocol CPN are given in Section D.1. Section D.2 presents the state space analysis results recorded in Design/CPN [109]. Section D.3 lists the code for obtaining the FSA from Design/CPN, and gives several of the language analysis results.

D.1 TR-Protocol CPN Declarations

The complete set of declarations for the TR-Protocol CPN are given in Listing D.1.

Listing D.1: Declarations for the TR-Protocol CPN

```
1  (* Design/CPN Options *)
2  NewOGGeneration := true;
3
4  (* Base Types & Variables *)
5  color Flag = bool with (F,T);
6  color RCR_c = int;
7  var u, TorF:Flag;
8
9  (* Maximum Counter Values *)
10 val RCRImax = 3;
11 val RCRRmax = 1;
12
13 (* PDUs *)
14 color InvokePDU_c = record
15    RID:Flag      *   (* Retransmission Indicator *)
16    UP:Flag;      (* User or PE acknowledgment *)
17 color ResultPDU_c = Flag;   (* RID *)
18 color AckPDU_c = record
19    RID:Flag *    (* Retransmission Indicator *)
```

```
20    TveTok:Flag; (* TID Verification /TID Ok *)
21 color AbortPDU_c = with abort; (* No fields *)
22 color PDU = union InvokePDU:InvokePDU_c +
23                   ResultPDU:ResultPDU_c +
24                   AckPDU:AckPDU_c    +
25                   AbortPDU:AbortPDU_c;
26 var invoke:InvokePDU_c;
27 var result :ResultPDU_c;
28 var ack:AckPDU_c;
29
30 (* Initiator & Responder State Names *)
31 color IStateName = with
32    I_NULL              |
33    I_RESULT_WAIT       |
34    I_RESULT_RESP_WAIT  |
35    I_WAIT_TIMEOUT  ;
36
37 color RStateName = with
38    R_LISTEN            |
39    R_TIDOK_WAIT        |
40    R_INVOKE_RESP_WAIT  |
41    R_RESULT_WAIT       |
42    R_RESULT_RESP_WAIT ;
43
44 (* Transaction Data - Initiator *)
45 color ITransData = record
46    Uack:Flag    *    (* True if UserAck On *)
47    RCR:RCR_c *       (* Retransmission Counter *)
48    AckSent:Flag*     (* True if Ack(TIDok) PDU sent *)
49    HoldOn:Flag *     (* True if Ack has been received *)
50    Timer:Flag   ;    (* True if Timer on *)
51
52 (* Transaction Data - Responder *)
53 color RTransData = record
54    Uack:Flag    *    (* True if UserAck On *)
55    RCR:RCR_c *       (* Retransmission Counter *)
56    AckSent:Flag*     (* True if Ack PDU sent *)
57    Timer:Flag   ;    (* True if Timer on *)
58
59 color InitState = product IStateName * ITransData;
60 color RespState = product RStateName * RTransData;
61
62 var isn :IStateName;
63 var rsn :RStateName;
64 var it :ITransData;
65 var rt :RTransData;
66
67 (* Functions for modifying Initiator Transaction data *)
68 fun AssignUackI (t:ITransData, u:Flag):ITransData=
69    {Uack=u,
70     RCR=#RCR(t),
71     AckSent=#AckSent(t),
```

```
72      HoldOn=#HoldOn(t),
73      Timer=#Timer(t)};
74  fun SetAckSentI (t:ITransData):ITransData=
75      {Uack=#Uack(t),
76       RCR=#RCR(t),
77       AckSent=T,
78       HoldOn=#HoldOn(t),
79       Timer=#Timer(t)};
80  fun IncRCRI (t:ITransData):ITransData=
81      {Uack=#Uack(t),
82       RCR=#RCR(t)+1,
83       AckSent=#AckSent(t),
84       HoldOn=#HoldOn(t),
85       Timer=#Timer(t)};
86  fun StartTimerI (t:ITransData):ITransData=
87      {Uack=#Uack(t),
88       RCR=#RCR(t),
89       AckSent=#AckSent(t),
90       HoldOn=#HoldOn(t),
91       Timer=T};
92  fun StopTimerI (t:ITransData):ITransData=
93      {Uack=#Uack(t),
94       RCR=#RCR(t),
95       AckSent=#AckSent(t),
96       HoldOn=#HoldOn(t),
97       Timer=F};
98  fun SetHoldOnI (t:ITransData):ITransData=
99      {Uack=#Uack(t),
100      RCR=#RCR(t),
101      AckSent=#AckSent(t),
102      HoldOn=T,
103      Timer=#Timer(t)};
104 fun  ClearInitI  (t:ITransData):ITransData=
105     {Uack=F,
106      RCR=0,
107      AckSent=F,
108      HoldOn=F,
109      Timer=F};
110
111 (* Functions for modifying Responder Transaction data *)
112 fun AssignUackR (t:RTransData, u:Flag):RTransData=
113     {Uack=u,
114      RCR=#RCR(t),
115      AckSent=#AckSent(t),
116      Timer=#Timer(t)};
117 fun SetAckSentR (t:RTransData):RTransData=
118     {Uack=#Uack(t),
119      RCR=#RCR(t),
120      AckSent=T,
121      Timer=#Timer(t)};
122 fun IncRCRR (t:RTransData):RTransData=
123     {Uack=#Uack(t),
```

```
124      RCR=#RCR(t)+1,
125      AckSent=#AckSent(t),
126      Timer=#Timer(t)};
127 fun ResetRCRR (t:RTransData):RTransData=
128    {Uack=#Uack(t),
129      RCR=0,
130      AckSent=#AckSent(t),
131      Timer=#Timer(t)};
132 fun StartTimerR (t:RTransData):RTransData=
133    {Uack=#Uack(t),
134      RCR=#RCR(t),
135      AckSent=#AckSent(t),
136      Timer=T};
137 fun StopTimerR (t:RTransData):RTransData=
138    {Uack=#Uack(t),
139      RCR=#RCR(t),
140      AckSent=#AckSent(t),
141      Timer=F};
142 fun ClearRespR (t:RTransData):RTransData=
143    {Uack=F,
144      RCR=0,
145      AckSent=F,
146      Timer=F};
```

D.2 TR-Protocol State Space Results

The state space analysis of the TR-Protocol discussed in Chapter 8 was performed using Design/CPN [109]. Design/CPN produces a report summarizing important statistics and properties of the state space. This report for Configuration 1 of the TR-Protocol is given in Listing D.2.

Listing D.2: State space report generated by Design/CPN for Configuration 1 of the TR-Protocol CPN

```
1  Statistics
2 ------------------------------------
3   Occurrence Graph
4     Nodes:  40386
5     Arcs:   182395
6     Secs:   377
7     Status: Full
8
9   Scc Graph
10    Nodes:  40386
11    Arcs:   182395
12    Secs:   58
13
14  Boundedness Properties
15 ------------------------------------
16   Best Integers Bounds                Upper       Lower
```

```
17  I_NULL'UserAck 1                    1          0
18  I_RW_RcvResult_Cnf'TempState 1      1          0
19  R_LISTEN'First 1                    1          1
20  TR_Init_PE' Initiator  1            1          0
21  TR_Protocol'InitToResp 1            5          0
22  TR_Protocol'RespToInit 1            9          0
23  TR_Resp_PE'Responder 1              1          1
24
25 Liveness Properties
26 ------------------------------------
27  Dead Markings:  1884 [979,9780,978,9779,9617,...]
28  Dead Transitions Instances:
29
30  I_RESULT_RESP_WAIT'TimerTO_A_Max 1
31  R_INVOKE_RESP_WAIT'TimerTO_A_Max 1
```

A desired property of the TR-Protocol is successful termination (Property 8.2). To prove this property, a query is applied on all dead markings in the state space to determine if they are desirable. Listing D.3 gives the query and Figure D.1 shows the results of this query. The predicate IsValidTerminal() takes a node as input and returns true if it is of the form of the desirable markings specified in Tables 8.1 and 8.2. This predicate is applied to all nodes in the list of dead markings, and if any elements are false (i.e. a dead marking is unexpected), then invalidterminal will evaluate to true. Figure D.1 shows all dead markings are desirable (i.e. invalidterminal is false).

Listing D.3: Standard ML code for checking validity of dead markings in the TR-Protocol

```
1  fun IsValidTerminal (n)=
2          ((Mark.I_NULL'UserAck 1 n) == empty)
3  andalso
4          ((Mark.I_RW_RcvResult_Cnf'TempState 1 n) == empty)
5  andalso
6          ((Mark.I_NULL' Initiator 1 n) == (1,(I_NULL,{RCR=0,Uack=F,
7                  AckSent=F,HoldOn=F,Timer=F}))!!empty)
8  andalso
9  ((         ((Mark.R_LISTEN'First 1 n) == (1,T)!!empty) andalso
10         ((Mark.R_LISTEN'Responder 1 n) == (1,(R_LISTEN,{RCR=0,Uack=F,
11                 AckSent=F,Timer=F}))!!empty) andalso
12         ((Mark.I_NULL'InitToResp 1 n) == empty) andalso
13         ((Mark.R_LISTEN'RespToInit 1 n) == empty))
14     orelse
15         (((Mark.R_LISTEN'First 1 n) == (1,F)!!empty) andalso
16         ((Mark.R_LISTEN'Responder 1 n) == (1,(R_LISTEN,{RCR=0,Uack=F,
17                 AckSent=F,Timer=F}))!!empty)));
```

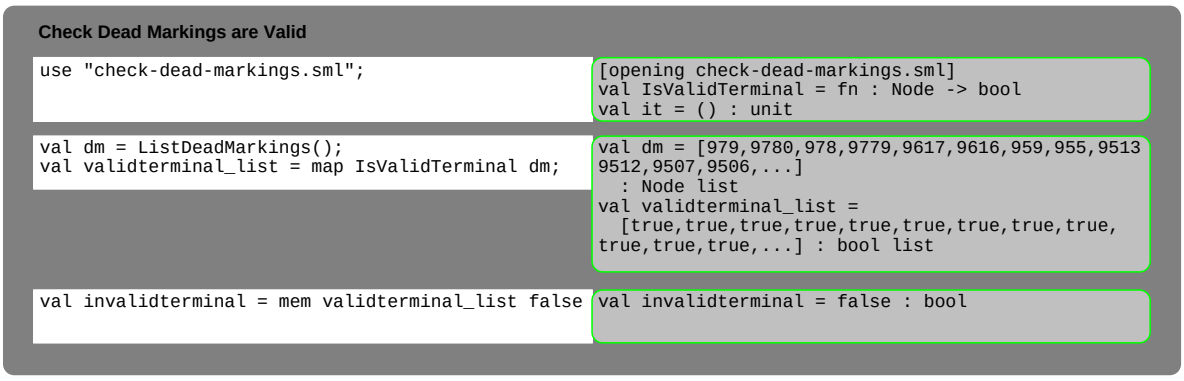

Figure D.1: Standard ML code that matches all desired dead markings in the state space of Configuration 1 of the TR-Protocol

D.3 TR-Protocol Language Results

D.3.1 Binding Element Map Specification

Language analysis involves calculating the TR-Protocol language and comparing it with the TR-Service language. The TR-Protocol language is obtained from the TR-Protocol state space using the process introduced in Chapter 4 and described in detail in Appendix A. Listing D.4 shows the function used for mapping the binding elements in the TR-Protocol CPN to integers representing service primitives (as defined in Table 6.5). Nearly all of the mappings can be obtained directly from the names and labels of the transitions in the TR-Protocol CPN. The exceptions are the ProviderAbort transitions on the pages I_ABORT (Figure 7.17) and R_ABORT (Figure 7.18). The labels on these transitions indicate that the delivery of the TR-Abort.ind primitive to the TR-User is conditional. For transition ProviderAbort on the I_ABORT page, a TR-Abort.ind primitive is not delivered to the TR-Init-User if the abort occurs while the TR-Init-PE is in the I_WAIT_TIMEOUT state (isn=I_WAIT_TIMEOUT). For transition ProviderAbort on the R_ABORT page, a TR-Abort.ind primitive is not delivered to the TR-Resp-User if the abort occurs while the TR-Resp-PE is in the R_TIDOK_WAIT state (rsn=R_TIDOK_WAIT).

Listing D.4: Standard ML code for mapping state space arcs to primitive numbers for the TR-Protocol CPN

```
1  fun be2str (Bind.I_NULL'Invoke_req (1,_))              = "1"
2    | be2str (Bind.I_NULL'RcvAck_Tve (1,_))              = "0"
3    | be2str (Bind.I_RESULT_WAIT'Abort_req (1,_))        = "9"
4    | be2str (Bind.I_RESULT_WAIT'TimerTO_R_Max (1,_))  = "11"
5    | be2str (Bind.I_RESULT_WAIT'TimerTO_R (1,_))       = "0"
6    | be2str (Bind.I_RESULT_WAIT'TimerTO_R_Tve (1,_))  = "0"
7    | be2str (Bind.I_RESULT_WAIT'RcvAck_Tve (1,_))      = "0"
8    | be2str (Bind.I_RESULT_WAIT'RcvAck_Cnf (1,_))      = "4"
9    | be2str (Bind.I_RESULT_WAIT'RcvResult (1,_))       = "6"
10   | be2str (Bind.I_RESULT_WAIT'RcvAbort (1,_))        = "11"
```

```
11    | be2str (Bind.I_RESULT_WAIT'UserAbort (1,_))          = "9"
12    | be2str (Bind.I_RESULT_WAIT'ProviderAbort (1,_))     = "11"
13    | be2str (Bind.I_RW_RcvResult_Cnf'Invoke_cnf (1,_))   = "4"
14    | be2str (Bind.I_RW_RcvResult_Cnf'Result_ind (1,_))   = "6"
15    | be2str (Bind.I_RESULT_RESP_WAIT'Abort_req (1,_))  = "9"
16    | be2str (Bind.I_RESULT_RESP_WAIT'TimerTO_A_Max (1,_))= "11" (* fixed 0 *)
17    | be2str (Bind.I_RESULT_RESP_WAIT'TimerTO_A_Off (1,_))= "0"
18    | be2str (Bind.I_RESULT_RESP_WAIT'Result_res (1,_))  = "7"
19    | be2str (Bind.I_RESULT_RESP_WAIT'RcvAbort (1,_))  = "11"
20    | be2str (Bind.I_WAIT_TIMEOUT'Clear (1,_))           = "0"  (* fixed 9 *)
21    | be2str (Bind.I_WAIT_TIMEOUT'RcvResult (1,_))       = "0"
22    | be2str (Bind.I_WAIT_TIMEOUT'RcvAbort (1,_))        = "0"  (* fixed 11 *)
23    | be2str (Bind.I_WAIT_TIMEOUT'TimerTO_W (1,_))     = "0"
24    | be2str (Bind.I_ABORT'ProviderAbort (1,{isn=I_WAIT_TIMEOUT,it=_}))= "0"
25    | be2str (Bind.I_ABORT'ProviderAbort (1,_))          = "11"
26  (* Responder Protocol Entity *)
27    | be2str (Bind.R_LISTEN'RcvInvoke (1,_))             = "2"
28    | be2str (Bind.R_LISTEN'RcvInvoke_Fail (1,_))        = "0"
29    | be2str (Bind.R_TIDOK_WAIT'RcvAck (1,_))            = "2"
30    | be2str (Bind.R_TIDOK_WAIT'RcvAbort (1,_))          = "0"
31    | be2str (Bind.R_TIDOK_WAIT'RcvInvoke (1,_))         = "0"
32    | be2str (Bind.R_INVOKE_RESP_WAIT'RcvAbort (1,_))  = "12"
33    | be2str (Bind.R_INVOKE_RESP_WAIT'Invoke_res (1,_))  = "3"
34    | be2str (Bind.R_INVOKE_RESP_WAIT'TimerTO_A_Max (1,_))= "12" (* fixed 0 *)
35    | be2str (Bind.R_INVOKE_RESP_WAIT'TimerTO_A_Off (1,_))= "0"
36    | be2str (Bind.R_INVOKE_RESP_WAIT'Abort_req (1,_))  = "10"
37    | be2str (Bind.R_INVOKE_RESP_WAIT'Result_req (1,_))  = "5"
38    | be2str (Bind.R_RESULT_WAIT'RcvAbort (1,_))         = "12"
39    | be2str (Bind.R_RESULT_WAIT'RcvInvoke (1,_))        = "0"
40    | be2str (Bind.R_RESULT_WAIT'TimerTO_A (1,_))        = "0"
41    | be2str (Bind.R_RESULT_WAIT'Abort_req (1,_))        = "10"
42    | be2str (Bind.R_RESULT_WAIT'Result_req (1,_))       = "5"
43    | be2str (Bind.R_RESULT_RESP_WAIT'RcvAbort (1,_))  = "12"
44    | be2str (Bind.R_RESULT_RESP_WAIT'RcvAck_Cnf (1,_)) = "8"
45    | be2str (Bind.R_RESULT_RESP_WAIT'TimerTO_R_Max (1,_))= "12"
46    | be2str (Bind.R_RESULT_RESP_WAIT'TimerTO_R (1,_))  = "0"
47    | be2str (Bind.R_RESULT_RESP_WAIT'Abort_req (1,_))  = "10"
48    | be2str (Bind.R_ABORT'ProviderAbort (1,{rsn=R_TIDOK_WAIT,rt=_}))= "0"
49    | be2str (Bind.R_ABORT'ProviderAbort (1,_))          = "12"
50    | be2str (_) = "ERROR";
51
52  fun ArcToFSM a = be2str(ArcToBE(a));
```

In Listing D.4, the changes described in Section 7.2.2 that impact on the mapping to service primitives are identified by the comment fixed, where the value before the change is also given. For example, the occurrence of TimerTO_A_Max on page R_INVOKE_RESP_WAIT would not correspond to a primitive being delivered from the WTP Specification, but the introduction of Assumption 7.8 now means a TR-Abort.ind (12) is delivered to the TR-Resp-User.

As well as mapping binding elements to service primitives, nodes in the state space

that correspond to halt states in the FSA must also be determined. The function FindHalts() in Listing D.5 specifies that only dead markings are mapped to halt states.

Listing D.5: Standard ML code for mapping state space nodes to halt states for the TR-Protocol CPN

```
1 (* Find nodes that correspond to halt states in TR-Protocol CPN *)
2 (* Node -> bool *)
3 fun FindHalts n = DeadMarking(n);
```

Figure D.2 shows the Standard ML queries used in Design/CPN, and the results, that define the mapping specification (the functions in Listings D.4 and D.5 are included in the file `map-spec.sml`) and perform the mapping.

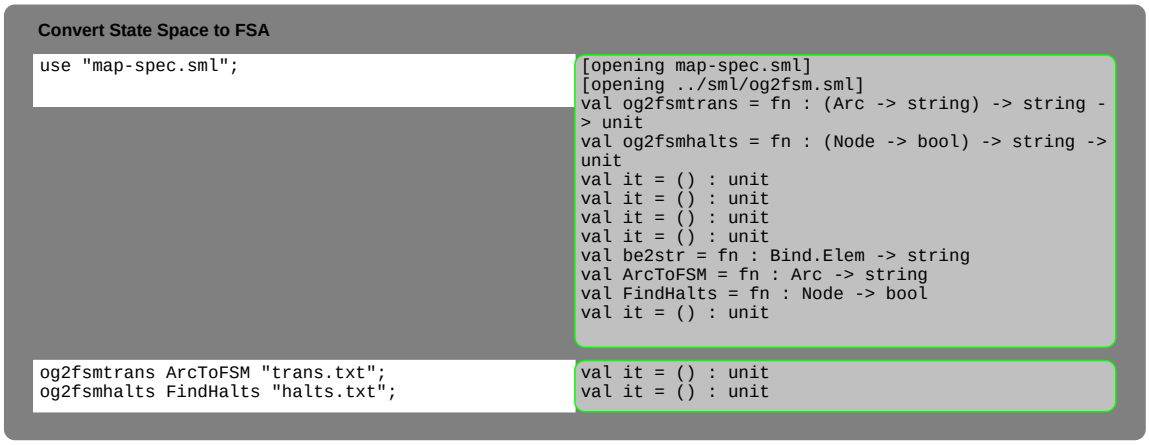

Figure D.2: Standard ML code that uses the mapping specification to convert the state space of Configuration 1 of the TR-Protocol to an FSA

D.3.2 Language Statistics

The language analysis of the TR-Protocol was performed using FSM [4]. Listing D.6 is the output of a terminal session where the statistics of the FSA (stored in the file `ogmin.fsm`) and language (stored in the file `lang.txt`) for Configuration 1 of the TR-Protocol are shown. Also shown is the number of sequences for which specific errors occur. These are calculated by searching the language (with an *awk* script [43]) for all sequences leading to the error. For example, the file `twoiind.awk` (Line 23, Listing D.6) is used to find all sequences that contain two TR-Invoke.ind primitives (designated as `[iind]` in the file `lang.txt`). Applying the script in `twoiind.awk` and counting the sequences returns the result in Line 35 of Listing D.6. `twoICNF.awk` and `noires.awk` are similar scripts to `twoiind.awk` (see Listing D.6). Each script also has a counterpart script (denoted with `-not` in the file name) which returns the sequences that do *not* match the pattern. Further interpretation of the results shown in Listing D.6 is given in Section 8.3.3.

Listing D.6: Terminal session output showing statistics for the TR-Protocol language of Configuration 1

```
> date
Thu Aug 23 09:59:08 ACST 2001
> pwd
/home/sgordon/thesis/cpn/protocol/ initial /p31F
> fsminfo -n ogmin.fsm
class                          basic
transducer                     n
# of states                    61
# of arcs                      278
 initial  state                0
# of final  states             6
# of eps                       0
# of accessible  states        61
# of coaccessible  states      61
# of connected states          61
# strongly conn components     61
> cat langstats.awk
BEGIN          {min=100}
               {if (length($0)<min) min=length($0)}
               {if (length($0)>max) max=length($0)}
END            {print ("LANG: ", NR, max/6, min/6)}
> cat twoiind.awk
$1 ~ /\[iind \](\[....\])*\[ iind \]/ {print $0}
> cat twoICNF.awk
$1 ~ /\[ICNF\](\[[ra ]...\])*\[ ICNF\]/ {print $0}
> cat noires.awk
$1 ~ /\[iind \](\[[^ i ]..[^ s]\])*\[ICNF\]/ {print $0}
> gawk -f langstats.awk lang.txt
LANG: 59562 14 2
> gawk -f langstats.awk difflang-protocol.txt
LANG: 26 6 4
> gawk -f langstats.awk difflang-service.txt
LANG: 59406 14 5
> gawk -f twoiind.awk difflang-service.txt | gawk -f langstats.awk
LANG: 57944 14 6
> gawk -f twoiind-not.awk difflang-service.txt | gawk -f twoICNF.awk |\
? gawk -f langstats.awk
LANG: 1368 11 6
> gawk -f twoiind-not.awk difflang-service.txt | gawk -f twoICNF-not.awk |\
? gawk -f noires.awk | gawk -f langstats.awk
LANG: 92 7 5
> gawk -f twoiind-not.awk difflang-service.txt | gawk -f twoICNF-not.awk |\
? gawk -f noires-not.awk
[IREQ][iind ][ ires ][ICNF][rreq ][RIND][rcnf]
[IREQ][iind ][ ires ][ rreq ][ICNF][RIND][rcnf]
>
```

Appendix E

Revised Transaction Protocol CPN and Results

Chapter 8 presents a set of suggested changes to fix the errors discovered in the TR-Protocol. These changes lead to the Revised TR-Protocol, which is analysed in Chapter 9. This appendix presents: the Revised TR-Protocol CPN in full (Section E.1), the state space analysis queries (Section E.2); and the results obtained from the model (Sections E.3 and E.4).

E.1 Revised TR-Protocol CPN

This section presents the complete Revised TR-Protocol CPN (Figure E.1 shows the hierarchy page). This model implements the suggested changes to the TR-Protocol (Chapter 7) that are given in Chapter 8. The declarations are listed in Listing E.1, followed by the 15 CPN pages. The arc inscriptions and guards that have been changed are given in bold. New transitions are outlined by a thick line. Appendix F describes the changes to the CPN so that multiple configurations can be analysed in Design/CPN.

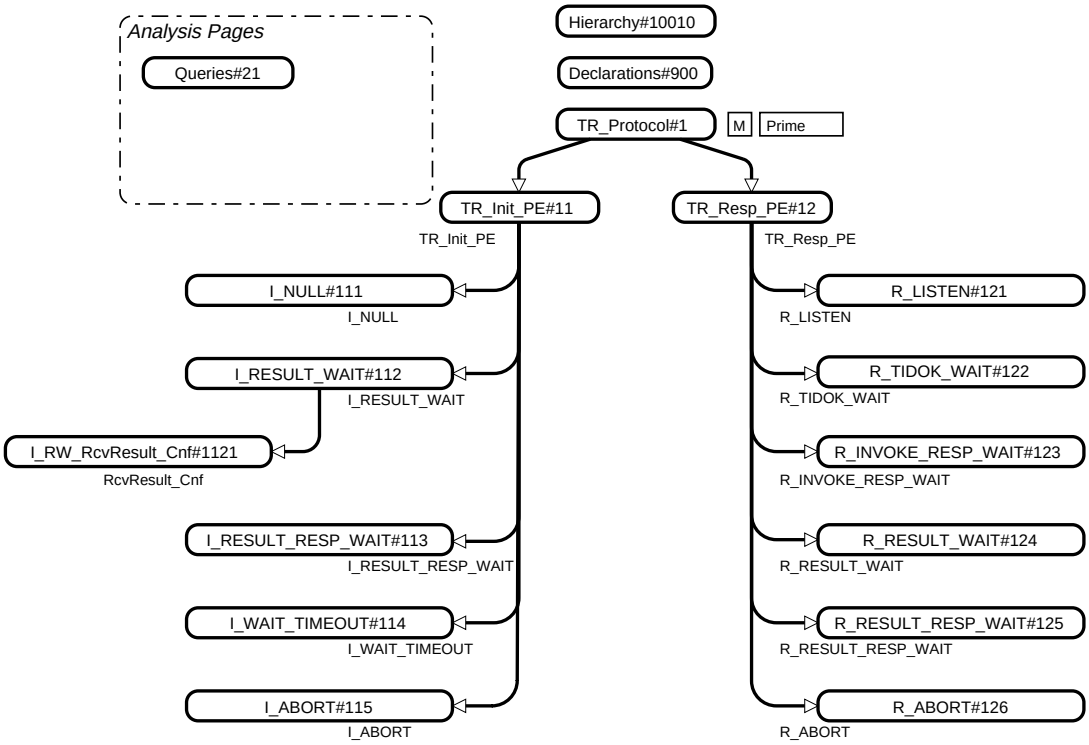

Figure E.1: Hierarchy page in the Revised TR-Protocol CPN

Listing E.1: Declarations for the Revised TR-Protocol CPN

```
(* Design/CPN Options *)
NewOGGeneration := true;

(* Base Types & Variables *)
color Integer = int;
color Flag = bool with (F,T);
color RCR_c = int;
var u, TorF:Flag;

(* Maximum Counter Values *)
val RCRImax = 1;
val RCRRmax = 1;

(* PDUs *)
color InvokePDU_c = record
    RID:Flag     *        (* Retransmission Indicator *)
    UP:Flag;              (* User Acknowledgement *)
color ResultPDU_c = record
    RID:Flag     *        (* RID *)
    CNF:Flag;             (* Confirmed from user *)
color AckPDU_c = record
    RID:Flag     *        (* Retransmission Indicator *)
    TveTok:Flag *         (* TID Verification/TID Ok *)
    CNF:Flag;             (* Confirmed from user *)
color AbortPDU_c = with abort; (* No fields *)
color PDU = union     InvokePDU:InvokePDU_c +
                      ResultPDU:ResultPDU_c +
```

```
28                          AckPDU:AckPDU_c    +
29                          AbortPDU:AbortPDU_c;
30 var invoke:InvokePDU_c;
31 var result:ResultPDU_c;
32 var ack:AckPDU_c;
33
34 (* Initiator & Responder State Names *)
35 color IStateName = with
36     I_NULL              |
37     I_RESULT_WAIT       |
38     I_RESULT_RESP_WAIT  |
39     I_WAIT_TIMEOUT      ;
40
41 color RStateName = with
42     R_LISTEN            |
43     R_TIDOK_WAIT        |
44     R_INVOKE_RESP_WAIT  |
45     R_RESULT_WAIT       |
46     R_RESULT_RESP_WAIT ;
47
48 var isn:IStateName;
49 var rsn:RStateName;
50
51 (* Transaction State Information - Initiator *)
52 color ITransData = record
53     Uack:Flag    *      (* True if User ack requested *)
54     RCR:RCR_c    *      (* Retransmission Counter *)
55     AckSent:Flag *      (* True if Ack(TIDok/TIDve) sent *)
56     Ucnf:Flag    *      (* True if user has confirmed *)
57     Timer:Flag;         (* True if Timer on *)
58
59 (* Transaction State Information - Responder *)
60 color RTransData = record
61     Uack:Flag    *      (* True if User ack requested *)
62     RCR:RCR_c    *      (* Retransmission Counter *)
63     AckSent:Flag *      (* True if Ack(TIDok/TIDve) sent *)
64     AbortSent:Flag *    (* True if Abort sent *)
65     Ucnf:Flag    *      (* True if user has confirmed *)
66     Timer:Flag;         (* True if Timer on *)
67
68 var it:ITransData;
69 var rt:RTransData;
70
71 color InitState = product IStateName * ITransData;
72 color RespState = product RStateName * RTransData;
73
74 (* Functions for modifying Initiator Transaction data *)
75 fun AssignUackI (t:ITransData, u:Flag):ITransData=
76    {Uack=u,
77     RCR=#RCR(t),
78     AckSent=#AckSent(t),
79     Ucnf=#Ucnf(t),
```

```
80      Timer=#Timer(t)};
81  fun SetAckSentI (t:ITransData):ITransData=
82      {Uack=#Uack(t),
83       RCR=#RCR(t),
84       AckSent=T,
85       Ucnf=#Ucnf(t),
86       Timer=#Timer(t)};
87  fun IncRCRI (t:ITransData):ITransData=
88      {Uack=#Uack(t),
89       RCR=#RCR(t)+1,
90       AckSent=#AckSent(t),
91       Ucnf=#Ucnf(t),
92       Timer=#Timer(t)};
93  fun StartTimerI (t:ITransData):ITransData=
94      {Uack=#Uack(t),
95       RCR=#RCR(t),
96       AckSent=#AckSent(t),
97       Ucnf=#Ucnf(t),
98       Timer=T};
99  fun StopTimerI (t:ITransData):ITransData=
100     {Uack=#Uack(t),
101      RCR=#RCR(t),
102      AckSent=#AckSent(t),
103      Ucnf=#Ucnf(t),
104      Timer=F};
105 fun SetUcnfI (t:ITransData):ITransData=
106     {Uack=#Uack(t),
107      RCR=#RCR(t),
108      AckSent=#AckSent(t),
109      Ucnf=T,
110      Timer=#Timer(t)};
111 fun ResetUcnfI (t:ITransData):ITransData=
112     {Uack=#Uack(t),
113      RCR=#RCR(t),
114      AckSent=#AckSent(t),
115      Ucnf=F,
116      Timer=#Timer(t)};
117 fun ClearInitI (t:ITransData):ITransData=
118     {Uack=F,
119      RCR=0,
120      AckSent=F,
121      Ucnf=F,
122      Timer=F};
123
124 (* Functions for modifying Responder Transaction data *)
125 fun AssignUackR (t:RTransData, u:Flag):RTransData=
126     {Uack=u,
127      RCR=#RCR(t),
128      AckSent=#AckSent(t),
129      AbortSent=#AbortSent(t),
130      Ucnf=#Ucnf(t),
131      Timer=#Timer(t)};
```

```
132 fun SetAckSentR (t:RTransData):RTransData=
133     {Uack=#Uack(t),
134      RCR=#RCR(t),
135      AckSent=T,
136      AbortSent=#AbortSent(t),
137      Ucnf=#Ucnf(t),
138      Timer=#Timer(t)};
139 fun SetAbortSentR (t:RTransData):RTransData=
140     {Uack=#Uack(t),
141      RCR=#RCR(t),
142      AckSent=#AckSent(t),
143      AbortSent=T,
144      Ucnf=#Ucnf(t),
145      Timer=#Timer(t)};
146 fun IncRCRR (t:RTransData):RTransData=
147     {Uack=#Uack(t),
148      RCR=#RCR(t)+1,
149      AckSent=#AckSent(t),
150      AbortSent=#AbortSent(t),
151      Ucnf=#Ucnf(t),
152      Timer=#Timer(t)};
153 fun ResetRCRR (t:RTransData):RTransData=
154     {Uack=#Uack(t),
155      RCR=0,
156      AckSent=#AckSent(t),
157      AbortSent=#AbortSent(t),
158      Ucnf=#Ucnf(t),
159      Timer=#Timer(t)};
160 fun StartTimerR (t:RTransData):RTransData=
161     {Uack=#Uack(t),
162      RCR=#RCR(t),
163      AckSent=#AckSent(t),
164      AbortSent=#AbortSent(t),
165      Ucnf=#Ucnf(t),
166      Timer=T};
167 fun StopTimerR (t:RTransData):RTransData=
168     {Uack=#Uack(t),
169      RCR=#RCR(t),
170      AckSent=#AckSent(t),
171      AbortSent=#AbortSent(t),
172      Ucnf=#Ucnf(t),
173      Timer=F};
174 fun ClearRespR (t:RTransData):RTransData=
175     {Uack=F,
176      RCR=0,
177      AckSent=F,
178      AbortSent=F,
179      Ucnf=F,
180      Timer=F};
181 fun SetUcnfR (t:RTransData):RTransData=
182     {Uack=#Uack(t),
183      RCR=#RCR(t),
```

```
184     AckSent=#AckSent(t),
185     AbortSent=#AbortSent(t),
186     Ucnf=T,
187     Timer=#Timer(t)};
```

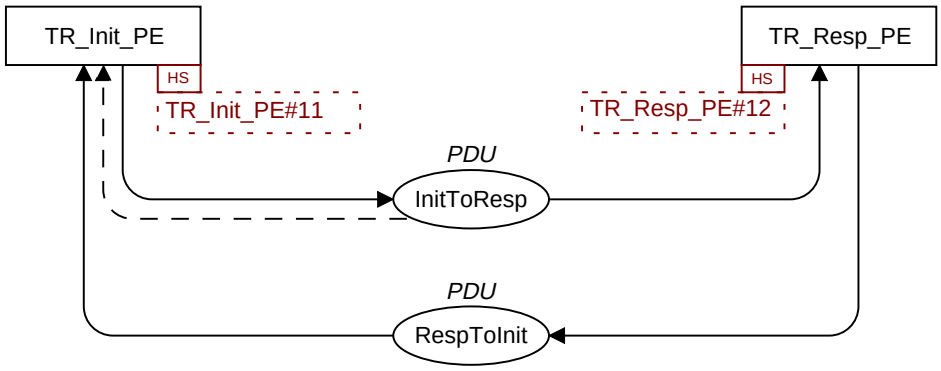

Figure E.2: TR_Protocol page in the Revised TR-Protocol CPN

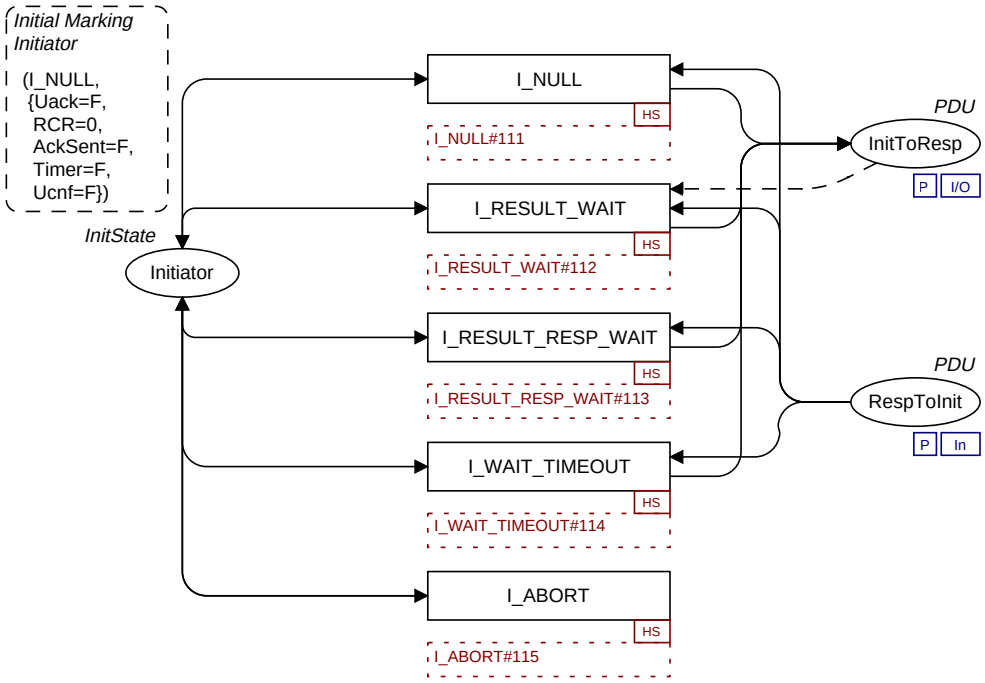

Figure E.3: TR_Init_PE page in the Revised TR-Protocol CPN

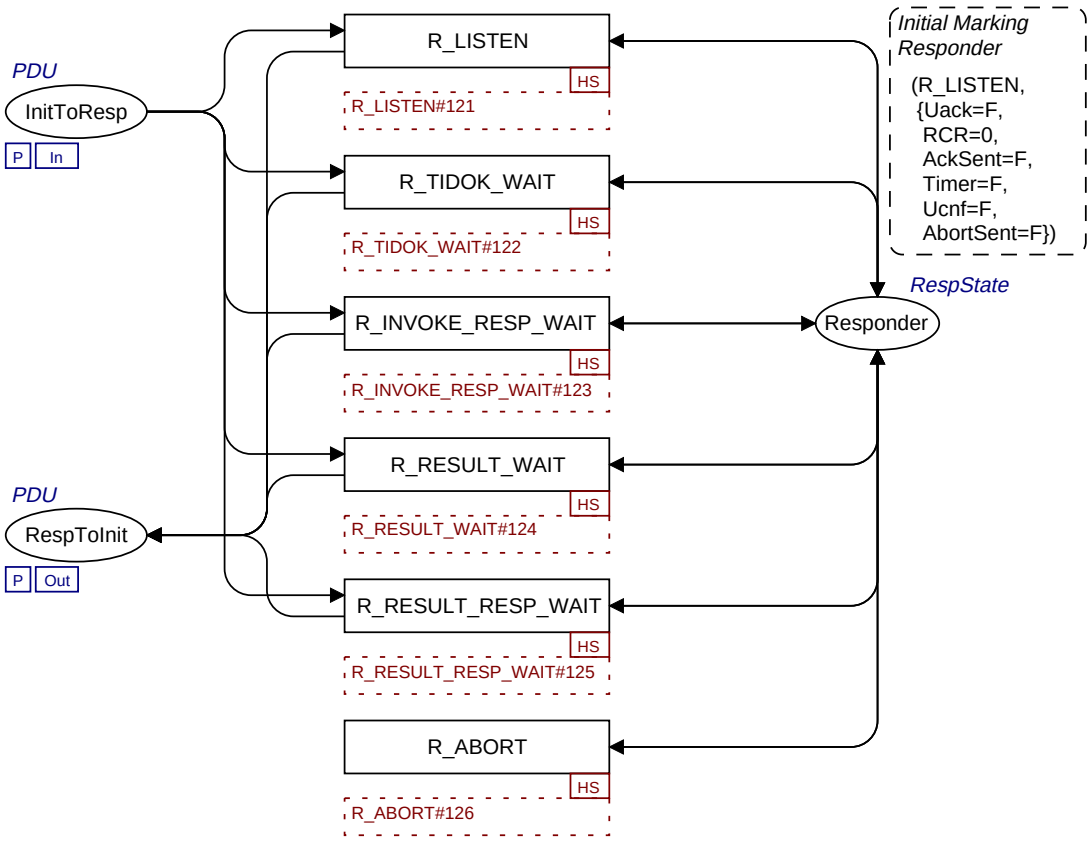

Figure E.4: TR_Resp_PE page in the Revised TR-Protocol CPN

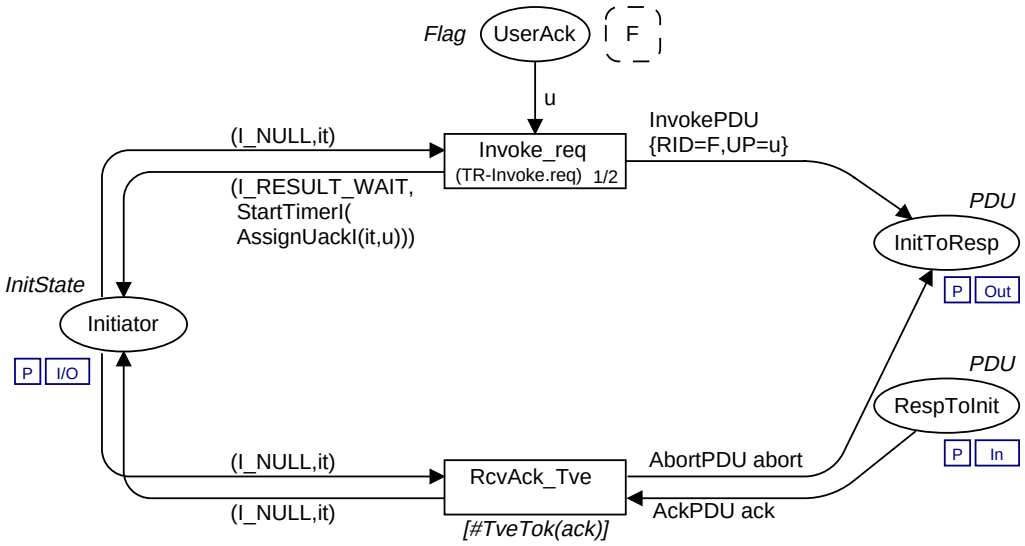

Figure E.5: I_NULL page in the Revised TR-Protocol CPN

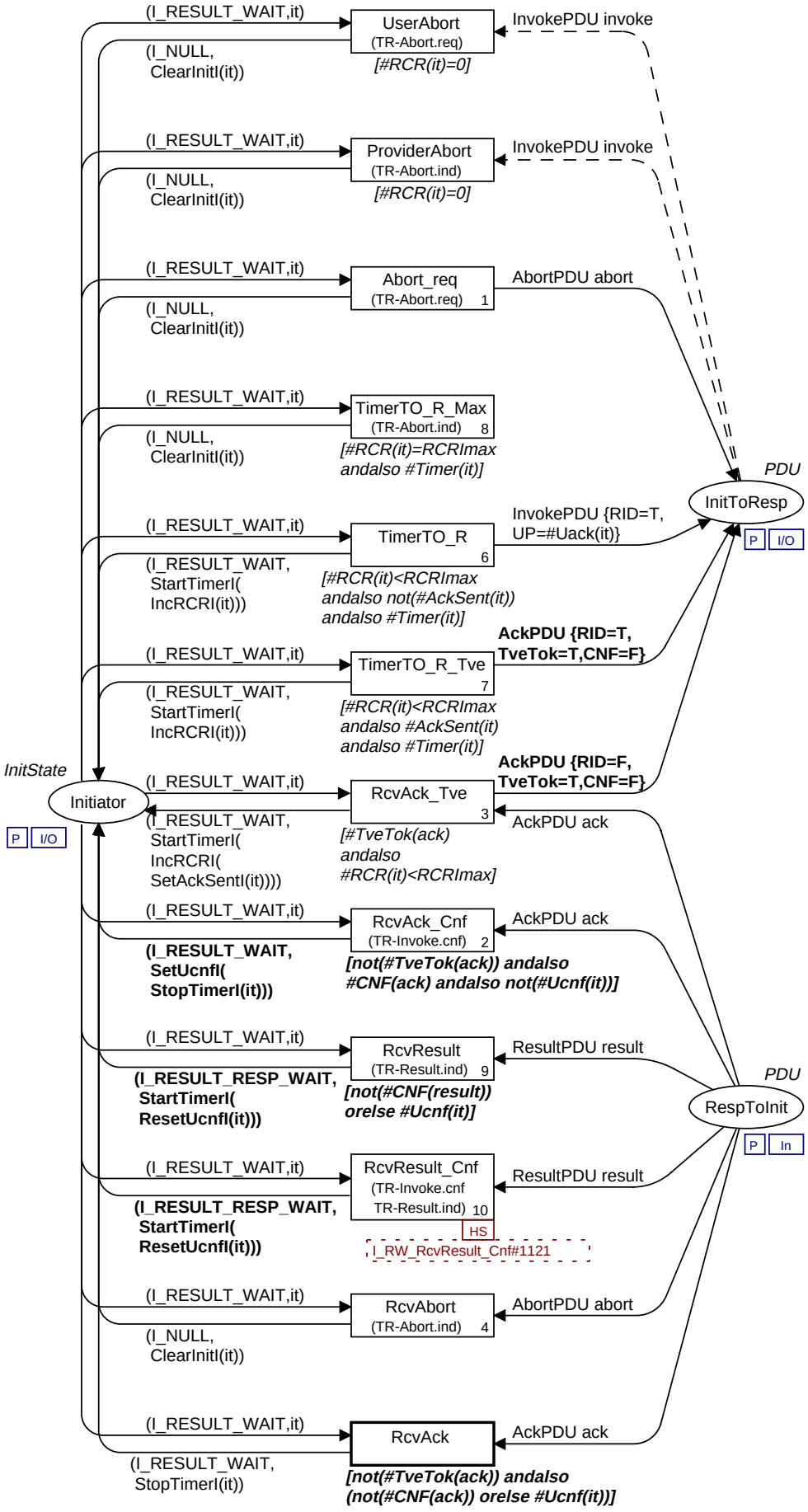

Figure E.6: I_RESULT_WAIT page in the Revised TR-Protocol CPN

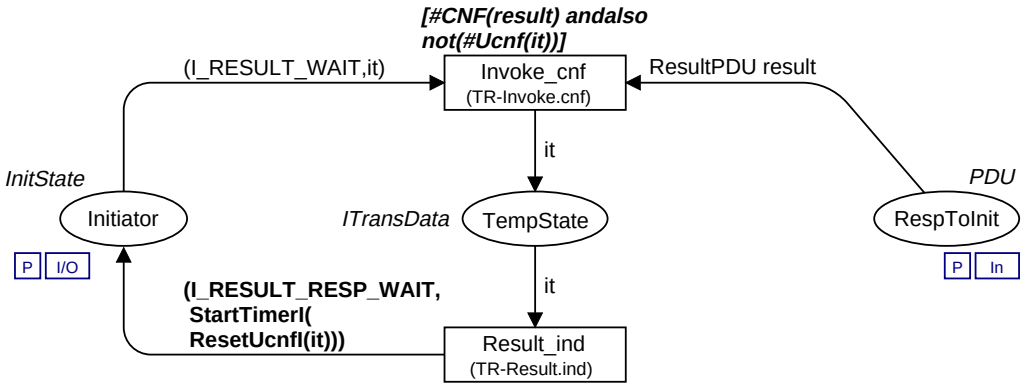

Figure E.7: I_RW_RcvResult_Cnf page in the Revised TR-Protocol CPN

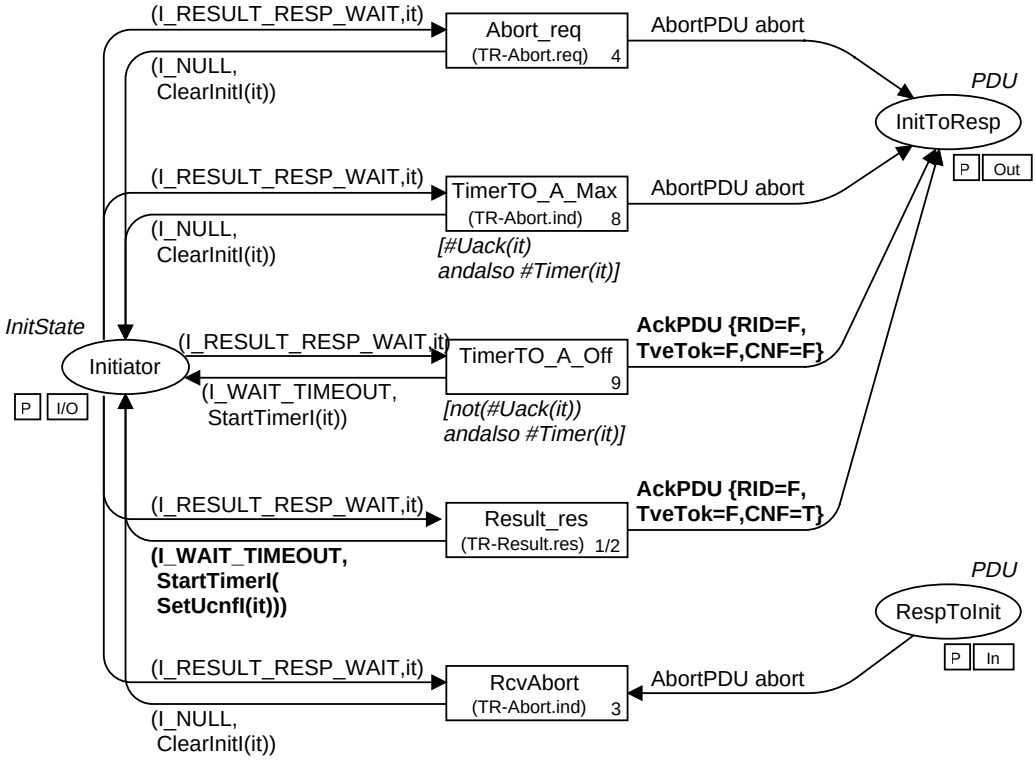

Figure E.8: I_RESULT_RESP_WAIT page in the Revised TR-Protocol CPN

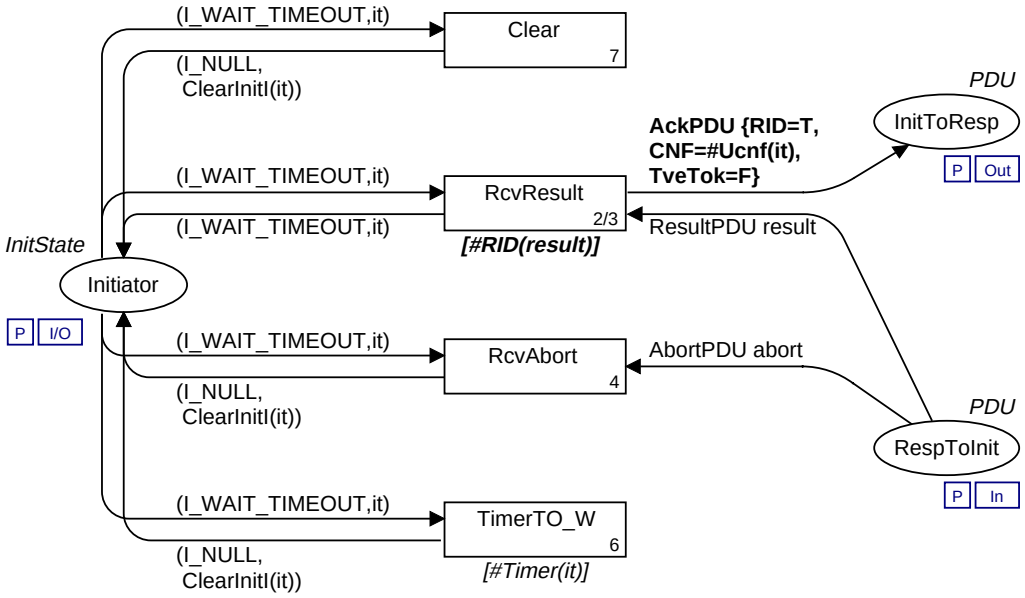

Figure E.9: I_WAIT_TIMEOUT page in the Revised TR-Protocol CPN

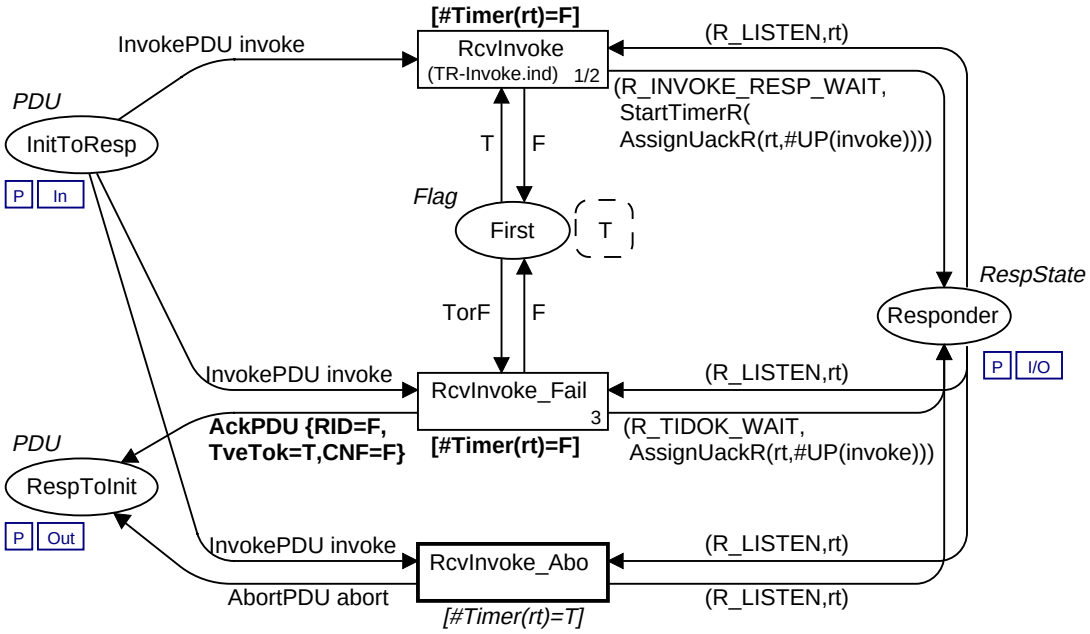

Figure E.10: R_LISTEN page in the Revised TR-Protocol CPN

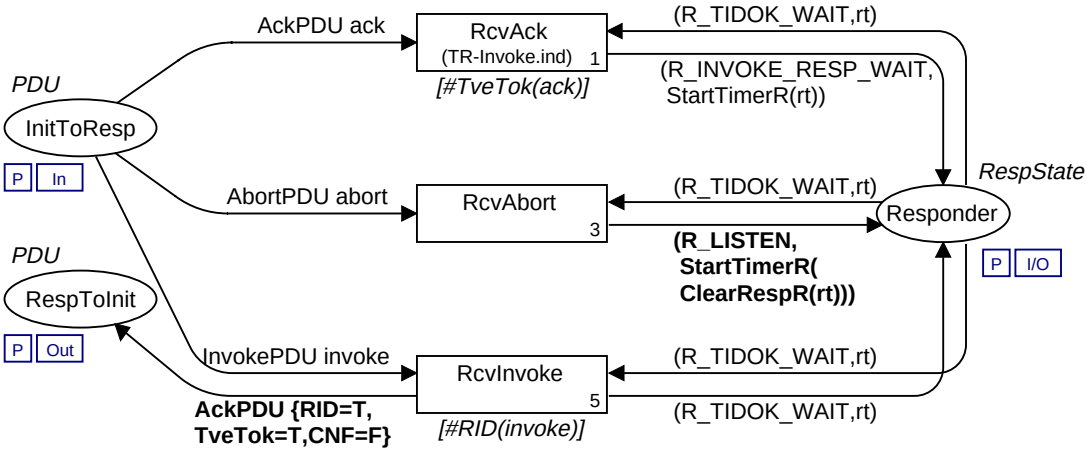

Figure E.11: R_TIDOK_WAIT page in the Revised TR-Protocol CPN

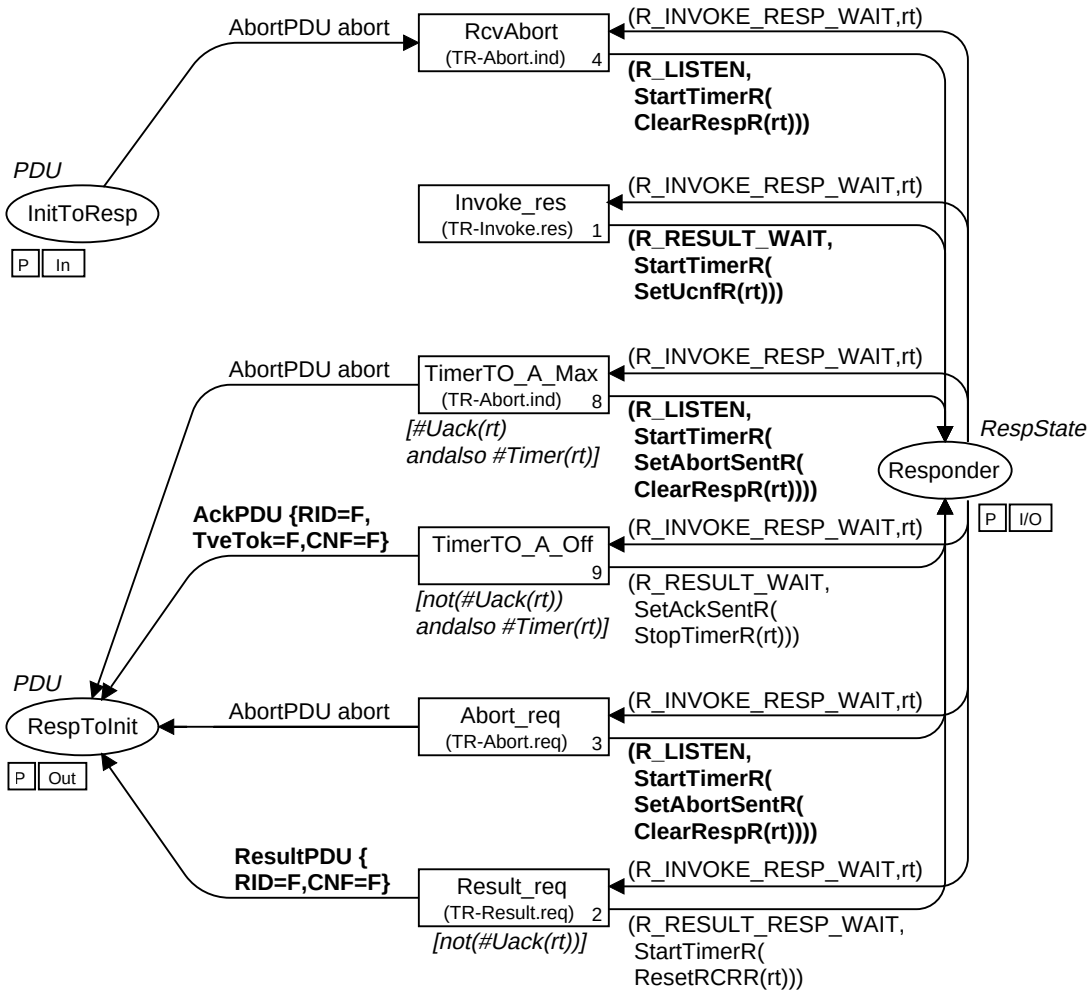

Figure E.12: R_INVOKE_RESP_WAIT page in the Revised TR-Protocol CPN

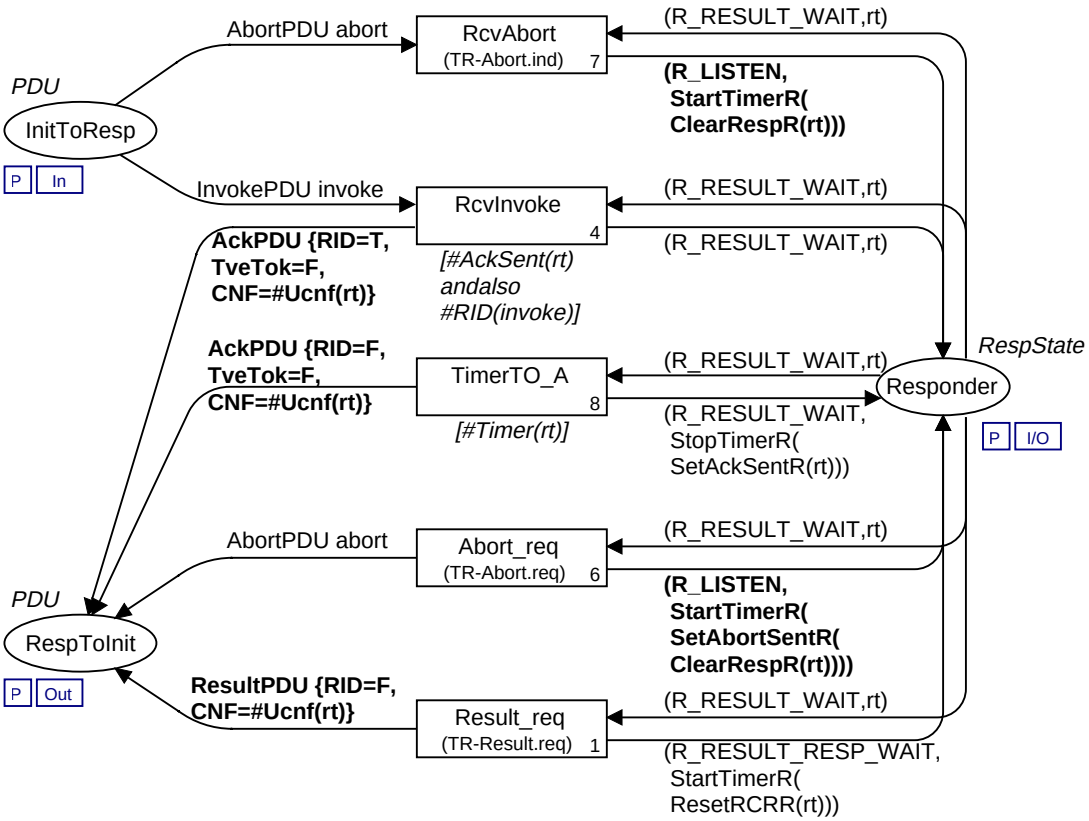

Figure E.13: R_RESULT_WAIT page in the Revised TR-Protocol CPN

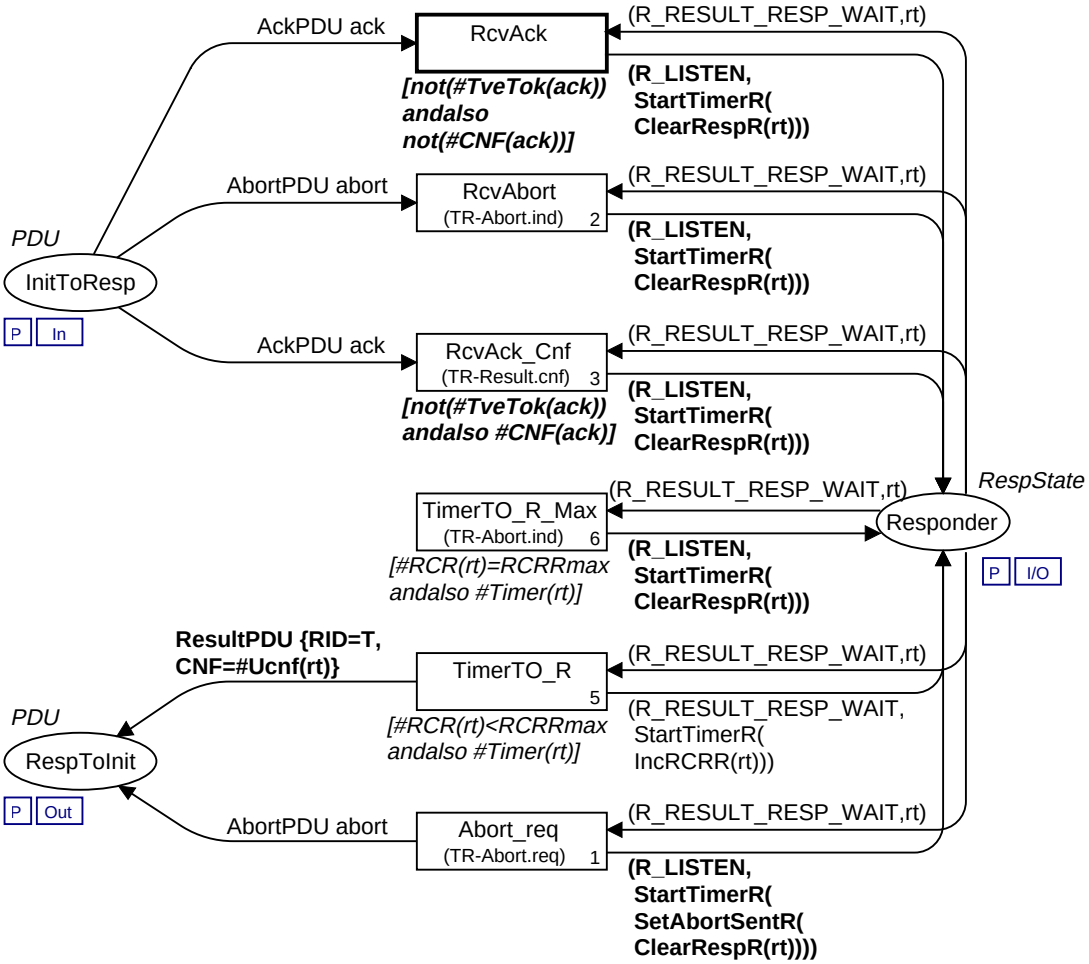

Figure E.14: R_RESULT_RESP_WAIT page in the Revised TR-Protocol CPN

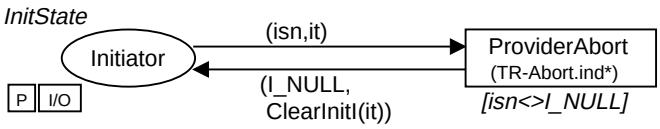

Figure E.15: I_ABORT page in the Revised TR-Protocol CPN

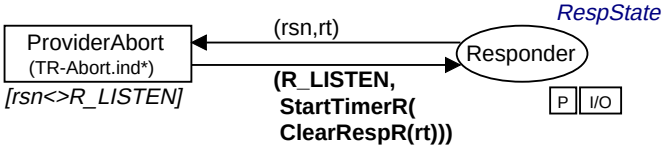

Figure E.16: R_ABORT page in the Revised TR-Protocol CPN

E.2 Revised TR-Protocol State Space Code

State spaces were calculated for more than 50 configurations of the Revised TR-Protocol in Design/CPN. This section gives the query used to determine if dead markings are expected or not.

A dead marking in the Revised TR-Protocol state space corresponds to either a terminal marking or a deadlock. The form of a terminal marking is discussed in Chapter 8. The function IsValidTerminal(), given in Listing E.2, returns true if a dead marking matches the form of a terminal marking. This is used to prove Property 8.2 in Chapter 8.

Listing E.2: Standard ML code for checking validity of dead markings in Revised TR-Protocol

```
1  fun IsValidTerminal (n)=
2          ((Mark.I_NULL'UserAck 1 n) == empty)
3  andalso
4          ((Mark.I_RW_RcvResult_Cnf'TempState 1 n) == empty)
5  andalso
6          ((Mark.I_NULL' Initiator 1 n) == (1,(I_NULL,{RCR=0,Uack=F,
7                  AckSent=F,Ucnf=F,Timer=F}))!!empty)
8  andalso
9  (   (   ((Mark.R_LISTEN'First 1 n) == (1,T)!!empty) andalso
10         ((Mark.R_LISTEN'Responder 1 n) == (1,(R_LISTEN,{RCR=0,Uack=F,
11                 AckSent=F,Timer=F,Ucnf=F,AbortSent=F}))!!empty) andalso
12         ((Mark.I_NULL'InitToResp 1 n) == empty) andalso
13         ((Mark.R_LISTEN'RespToInit 1 n) == empty))
14     orelse
15     (   ((Mark.R_LISTEN'First 1 n) == (1,F)!!empty) andalso
16         ((Mark.R_LISTEN'Responder 1 n) == (1,(R_LISTEN,{RCR=0,Uack=F,
17                 AckSent=F,Timer=T,Ucnf=F,AbortSent=F}))!!empty orelse
18         (Mark.R_LISTEN'Responder 1 n) == (1,(R_LISTEN,{RCR=0,Uack=F,
19                 AckSent=F,Timer=T,Ucnf=F,AbortSent=T}))!!empty)));
```

E.3 Revised TR-Protocol Language Results

E.3.1 Binding Element Map Specification

As with the TR-Protocol (see Appendix D), the binding elements in the Revised TR-Protocol must be mapped to symbols representing service primitives for the language analysis to be performed. Listing E.3 specifies the mapping to service primitives. Listing E.4 shows that all dead markings are mapped to halt states.

Listing E.3: Standard ML code for mapping state space arcs to primitive numbers for the Revised TR-Protocol CPN

```
1  (* Convert Arc in TR-Protocol state space into FSA string *)
2  (* Bind.Elem -> string *)
```

```
3  fun be2str (Bind.I_NULL'Invoke_req (1,_))                = "1"
4    | be2str (Bind.I_NULL'RcvAck_Tve (1,_))                = "0"
5    | be2str (Bind.I_RESULT_WAIT'Abort_req (1,_))          = "9"
6    | be2str (Bind.I_RESULT_WAIT'TimerTO_R_Max (1,_))  = "11"
7    | be2str (Bind.I_RESULT_WAIT'TimerTO_R (1,_))          = "0"
8    | be2str (Bind.I_RESULT_WAIT'TimerTO_R_Tve (1,_))  = "0"
9    | be2str (Bind.I_RESULT_WAIT'RcvAck_Tve (1,_))         = "0"
10   | be2str (Bind.I_RESULT_WAIT'RcvAck_Cnf (1,_))         = "4"
11   | be2str (Bind.I_RESULT_WAIT'RcvResult (1,_))          = "6"
12   | be2str (Bind.I_RESULT_WAIT'RcvAbort (1,_))           = "11"
13   | be2str (Bind.I_RESULT_WAIT'UserAbort (1,_))          = "9"
14   | be2str (Bind.I_RESULT_WAIT'ProviderAbort (1,_))      = "11"
15   | be2str (Bind.I_RESULT_WAIT'RcvAck (1,_))             = "0"
16   | be2str (Bind.I_RW_RcvResult_Cnf'Invoke_cnf (1,_))    = "4"
17   | be2str (Bind.I_RW_RcvResult_Cnf'Result_ind (1,_))    = "6"
18   | be2str (Bind.I_RESULT_RESP_WAIT'Abort_req (1,_))  = "9"
19   | be2str (Bind.I_RESULT_RESP_WAIT'TimerTO_A_Max (1,_))= "11"
20   | be2str (Bind.I_RESULT_RESP_WAIT'TimerTO_A_Off (1,_))= "0"
21   | be2str (Bind.I_RESULT_RESP_WAIT'Result_res (1,_))  = "7"
22   | be2str (Bind.I_RESULT_RESP_WAIT'RcvAbort (1,_))  = "11"
23   | be2str (Bind.I_WAIT_TIMEOUT'Clear (1,_))             = "0"
24   | be2str (Bind.I_WAIT_TIMEOUT'RcvResult (1,_))         = "0"
25   | be2str (Bind.I_WAIT_TIMEOUT'RcvAbort (1,_))          = "0"
26   | be2str (Bind.I_WAIT_TIMEOUT'TimerTO_W (1,_))         = "0"
27   | be2str (Bind.I_ABORT'ProviderAbort (1,{isn=I_WAIT_TIMEOUT,it=_}))="0"
28   | be2str (Bind.I_ABORT'ProviderAbort (1,_))            = "11"
29 (* Responder Protocol Entity *)
30   | be2str (Bind.R_LISTEN'RcvInvoke (1,_))               = "2"
31   | be2str (Bind.R_LISTEN'RcvInvoke_Fail (1,_))          = "0"
32   | be2str (Bind.R_LISTEN'RcvInvoke_Abo (1,_))           = "0"
33   | be2str (Bind.R_TIDOK_WAIT'RcvAck (1,_))              = "2"
34   | be2str (Bind.R_TIDOK_WAIT'RcvAbort (1,_))            = "0"
35   | be2str (Bind.R_TIDOK_WAIT'RcvInvoke (1,_))           = "0"
36   | be2str (Bind.R_INVOKE_RESP_WAIT'RcvAbort (1,_))  = "12"
37   | be2str (Bind.R_INVOKE_RESP_WAIT'Invoke_res (1,_))  = "3"
38   | be2str (Bind.R_INVOKE_RESP_WAIT'TimerTO_A_Max (1,_))= "12"
39   | be2str (Bind.R_INVOKE_RESP_WAIT'TimerTO_A_Off (1,_))= "0"
40   | be2str (Bind.R_INVOKE_RESP_WAIT'Abort_req (1,_))  = "10"
41   | be2str (Bind.R_INVOKE_RESP_WAIT'Result_req (1,_))  = "5"
42   | be2str (Bind.R_RESULT_WAIT'RcvAbort (1,_))           = "12"
43   | be2str (Bind.R_RESULT_WAIT'RcvInvoke (1,_))          = "0"
44   | be2str (Bind.R_RESULT_WAIT'TimerTO_A (1,_))          = "0"
45   | be2str (Bind.R_RESULT_WAIT'Abort_req (1,_))          = "10"
46   | be2str (Bind.R_RESULT_WAIT'Result_req (1,_))         = "5"
47   | be2str (Bind.R_RESULT_RESP_WAIT'RcvAbort (1,_))  = "12"
48   | be2str (Bind.R_RESULT_RESP_WAIT'RcvAck_Cnf (1,_)) = "8"
49   | be2str (Bind.R_RESULT_RESP_WAIT'RcvAck (1,_))    = "0"
50   | be2str (Bind.R_RESULT_RESP_WAIT'TimerTO_R_Max (1,_))= "12"
51   | be2str (Bind.R_RESULT_RESP_WAIT'TimerTO_R (1,_))  = "0"
52   | be2str (Bind.R_RESULT_RESP_WAIT'Abort_req (1,_))  = "10"
53   | be2str (Bind.R_ABORT'ProviderAbort (1,{rsn=R_TIDOK_WAIT,rt=_}))="0"
54   | be2str (Bind.R_ABORT'ProviderAbort (1,_))            = "12"
```

```
55 (* used with state space with configurations only *)
56 (*   | be2str (Bind. Initialize ' ConfigInit (1,_))          = "0" *)
57   | be2str (_) = "ERROR";
58
59 (* Convert Arc in TR-Protocol state space into FSA string *)
60 (* Arc -> string *)
61 fun ArcToFSM a = be2str(ArcToBE(a));
```

Listing E.4: Standard ML code for mapping state space nodes to halt states for the Revised TR-Protocol CPN

```
1 (* Find nodes that correspond to halt states in TR-Protocol CPN *)
2 (* Node -> bool *)
3 fun FindHalts n = DeadMarking(n);
```

E.3.2 Language Statistics

Tables E.1 gives the FSA and language statistics for the configurations analysed. These statistics include: the number of nodes (N), arcs (A) and halt states (H) in the minimized FSA; the number of sequences (Seq), including the shortest (S) and longest (L) in the language; and the number of sequences in the TR-Protocol language but not in the TR-Service language (NIS) and vice versa (NIP).

E.4 Revised TR-Protocol Sweep-Line Analysis

E.4.1 Standard ML Code

Listing E.5 gives the implementation of the progress measure described in Section 9.3.1. Listing E.6 gives the code that performs the sweep-line analysis for a single configuration. The function sweep_properties() performs on-the-fly verification of the desired properties (see Section 9.3.2), whereas the function sweepanalyse_config() does not. The code for applying these functions to multiple configurations is given in Appendix F.

Listing E.5: Progress measure used in the sweep-line analysis of the Revised TR-Protocol CPN

```
1  (* Select RCR element from ITransData *)
2  (* ITransData ms -> int *)
3  exception grab_rcri_Exn
4  fun grab_rcri td_ms =
5      let val td_l = ms_to_list td_ms;
6          fun mapout ({RCR,...}:ITransData) = RCR;
7      in  if   td_ms == empty
8          then raise grab_rcri_Exn
9          else hd(map mapout td_l)
10     end;
```

RCRmax		*UserAck=T*								*UserAck=F*							
I	*R*	*N*	*A*	*H*	*Seq*	*L*	*S*	*NIS*	*NIP*	*N*	*A*	*H*	*Seq*	*L*	*S*	*NIS*	*NIP*
0	0	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
0	1	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
0	2	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
0	3	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
0	4	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
1	0	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
1	1	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
1	2	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
1	3	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
1	4	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
2	0	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
2	1	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
2	2	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
2	3	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
2	4	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
3	0	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
3	1	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
3	2	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
3	3	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
3	4	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
4	0	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
4	1	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
4	2	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
4	3	19	61	2	130	8	2	0	0	19	63	4	182	8	2	0	0
4	4	19	61	2	130	8	2	0	0	-	-	-	-	-	-	-	-
5	0	19	61	2	130	8	2	0	0	-	-	-	-	-	-	-	-
6	0	19	61	2	130	8	2	0	0	-	-	-	-	-	-	-	-
7	0	19	61	2	130	8	2	0	0	-	-	-	-	-	-	-	-

Table E.1: Statistics on the size of the FSA for the Revised TR-Protocol configurations

```
(* Select RCR element from RTransData *)
(* RTransData ms -> int *)
exception grab_rcrr_Exn
fun grab_rcrr td_ms =
    let val td_l = ms_to_list td_ms;
        fun mapout ({RCR,...}:RTransData) = RCR;
    in  if   td_ms == empty
        then raise grab_rcrr_Exn
        else hd(map mapout td_l)
    end;
(* int * int -> int *)
fun gm(rcri, rcrr) =
    let val rcrrbase = 1;
        val rcribase = RCRRmax()+2;
    in  rcri*rcribase + rcrr*rcrrbase
    end;

(* sweep-line progress measure. *)
(* CPN'NodeRec -> IntInf *)
fun WTPProgressMeasure CPN'n =
    let val m_csinit = OEMark.Initialize'ConfigPlace 1
                       (CPN'OGUtils.NodeRecToStateRec (!CPN'n));
        val m_init_state = ext_col fst (OEMark.I_NULL'Initiator 1
                       (CPN'OGUtils.NodeRecToStateRec (!CPN'n)));
        val m_tempstate = OEMark.I_RW_RcvResult_Cnf'TempState 1
                       (CPN'OGUtils.NodeRecToStateRec (!CPN'n));
        (* Initiator Transaction data is either stored in Initiator place
           or TempState place *)
        val m_init_data = if m_tempstate == empty
                          then ext_col snd (OEMark.I_NULL'Initiator 1
                               (CPN'OGUtils.NodeRecToStateRec (!CPN'n)))
                          else m_tempstate;
        val m_userack = OEMark.I_NULL'UserAck 1
                       (CPN'OGUtils.NodeRecToStateRec (!CPN'n));
        val m_first = OEMark.R_LISTEN'First 1
                      (CPN'OGUtils.NodeRecToStateRec (!CPN'n));
        val m_resp_state = ext_col fst (OEMark.R_LISTEN'Responder 1
                       (CPN'OGUtils.NodeRecToStateRec (!CPN'n)));
        val m_resp_data = ext_col snd (OEMark.R_LISTEN'Responder 1
                       (CPN'OGUtils.NodeRecToStateRec (!CPN'n)));
    in
        if   m_csinit == empty
        then
            (* End marking - Initiator *)
            if  (m_init_state == (1,I_NULL)!!empty) andalso
                (m_userack == empty)
            then
                (* End marking - Responder *)
                if (m_resp_state == (1,R_LISTEN)!!empty) andalso
                   (m_first == (1,F)!!empty)
                then IntInf.fromInt (gm(RCRImax()+1,RCRRmax()+1))

```

```
63                  (* General marking - Responder *)
64                  else IntInf.fromInt (gm(RCRImax()+1,grab_rcrr(m_resp_data)))
65
66              (* General marking - Initiator *)
67              else
68                  (* End marking - Responder *)
69                  if (m_resp_state == (1,R_LISTEN)!!empty) andalso
70                     (m_first == (1,F)!!empty)
71                  then IntInf.fromInt (gm(grab_rcri(m_init_data),RCRRmax()+1))
72
73                  (* General marking - Responder *)
74                  else IntInf.fromInt (gm(grab_rcri(m_init_data),
75                                          grab_rcrr(m_resp_data)))
76
77          (* Initial marking - Initialize 'ConfigPlace *)
78          else IntInf.fromInt 0
79      end;
```

Listing E.6: Setup Design/CPN for sweep-line analysis of the Revised TR-Protocol CPN

```
1  (*
2   * Performs sweep-line analysis in Design/CPN of one configuration of
3   * the Transaction Protocol. setup-configs.sml must have
4   * been included previously.
5   * ACKNOWLEDGEMENT: The majority of these functions were written by Lars
6   * Kristensen (University of South Australia), or are at
7   * least based on ones that he wrote.
8   *)
9  (* Install the progress measure (must be previously defined) *)
10 OGSweepFun.Set WTPProgressMeasure;
11
12 (* Function to start sweep for safety properties *)
13 fun SafetyAnalyse writelog (CPN'nodefun,CPN'arcfun,CPN'deadfun) =
14     let val _ = SLSafety.EvalNodes CPN'nodefun;
15         val _ = SLSafety.EvalArcs CPN'arcfun;
16         val _ = SLSafety.DeadMarkings CPN'deadfun;
17     in OGTimeSweep.Generate ~1 writelog
18     end;
19 fun CPN'StateRecToNodeRec (CPN'OGState {owner=ref(noderef),...}) = !noderef;
20
21 (* Perform the sweep *)
22 fun sweep_analysesafety (gcnodelimit, globaltimelimit) config
23     (CPN'nodefun,CPN'arcfun,CPN'deadfun) =
24     let val _ = writelog ((printconfig config)^"\n");
25         val _ = setcurconfig config
26         val _ = OGSet.StopOptions {Arcs = 0, Nodes = gcnodelimit,
27                                    Predicate = (fn _ => false),
28                                    Secs = globaltimelimit};
29         val ((maxnodes,totalglob),CPN'time) =
30             timeprofile (fn () => SafetyAnalyse
31                 writelog (CPN'nodefun,CPN'arcfun,CPN'deadfun));
32         val _ = writelog ("Sweep " ^makestring(totalglob)^" "^
33                           makestring(maxnodes)^" "^
```

```
34                              CPN'time^"\n");
35          val savestats = (totalglob,maxnodes,CPN'time);
36          val _ = DeleteOccGraph ();
37          val _ = OGSet.StopOptions {Arcs = 0, Nodes = 0,
38                                     Predicate = (fn _ => false),
39                                     Secs = globaltimelimit };
40      in  savestats
41      end;
42
43  (* Setup properties to be calculated and perform sweep *)
44  (* (int * int) -> (int * int * bool) -> unit *)
45  fun sweep_properties (gcnodelimit, globaltimelimit ) config =
46      let (* files and communication place markings *)
47          val transfilename = (printconfig config)^"trans.txt";
48          val deadfilename = (printconfig config)^"halts.txt";
49          val markfun_i2r = (OEMark.TR_Protocol'InitToResp 1);
50          val markfun_r2i = (OEMark.TR_Protocol'RespToInit 1);
51          val transoutfile = TextIO.openOut transfilename;
52          val _ = (CPN'storearcs := true);
53          (* assume that all trans are dead *)
54          val deadtrans = ref (TI.All);
55          (* write FSM data to file and find dead transitions *)
56          fun getfsminputanddt (CPN'src,CPN'b,CPN'dest) =
57              let
58                  val srcnodeno = CPN'OGUtils.GetNodeNo
59                                  (CPN'StateRecToNodeRec CPN'src);
60                  val destnodeno = CPN'OGUtils.GetNodeNo
61                                  (CPN'StateRecToNodeRec CPN'dest);
62                  val str = (makestring srcnodeno)^" "^
63                            (makestring destnodeno)^" "^
64                            (be2str CPN'b)^"\n"
65                  val _ = TextIO.output (transoutfile, str)
66              in
67                  (deadtrans := (List.filter
68                      (fn CPN'b1 => CPN'b1 <> (BEToTI CPN'b)) (!deadtrans)))
69              end;
70          (* calculate upper integer bounds of communication places *)
71          val curmax_i2r = ref 0;
72          val curmax_r2i = ref 0;
73          fun checkmarking CPN'n =
74              (curmax_i2r := Int.max(mssize (markfun_i2r CPN'n),(!curmax_i2r));
75               curmax_r2i := Int.max(mssize (markfun_r2i CPN'n),(!curmax_r2i)))
76          (* check dead markings are desired *)
77          fun nodefilter CPN'n =
78          if  IsValidTerminal(CPN'n)
79          then st_Node(CPN'OGUtils.GetNodeNo (CPN'StateRecToNodeRec CPN'n))^"\n"
80          else "DEADLOCK!"^st_Node(CPN'OGUtils.GetNodeNo
81                                  (CPN'StateRecToNodeRec CPN'n))^"\n";
82          val deadoutfile = TextIO.openOut deadfilename;
83          val countdm = ref 0;
84          fun deadfun CPN'n = (TextIO.output (deadoutfile, nodefilter CPN'n);
85                               CPN'inc countdm);
```

```
86          (* perform a sweep, checking the desired properties *)
87          val savestats=sweep_analysesafety (gcnodelimit, globaltimelimit) config
88              (checkmarking,getfsminputanddt,deadfun)
89          (* write stats to log file *)
90          val _ = writelog((printconfig config)^" "^
91                   makestring(#1(savestats))^" "^
92                   makestring(#2(savestats))^" "^
93                   (#3(savestats))^" "^
94                   makestring(!countdm)^" "^
95                   makestring(!curmax_i2r)^" "^
96                   makestring(!curmax_r2i)^" "^
97                   makestring(length(!deadtrans))^" "^
98                    printdeadtis (TI.All,!deadtrans)^"\n");
99          val _ = TextIO.closeOut ( transoutfile );
100         val _ = TextIO.closeOut ( deadoutfile );
101     in  ()
102     end;
103
104 (* Perform sweep analysis without calculating the properties, only the nodes *)
105 (* (int * int) -> (int * int * bool) -> unit *)
106 fun sweepanalyse_config (gcnodelimit, globaltimelimit) config =
107     let val _ = setcurconfig config
108         val _ = writelog ("Sweep "^(printconfig config)^"\n");
109         val _ = OGSet.StopOptions {Arcs = 0, Nodes = gcnodelimit,
110                                    Predicate = (fn _ => false),
111                                    Secs = globaltimelimit };
112         (* do the sweep *)
113         val ((maxnodes,totalglob),CPN'time) =
114                  timeprofile (fn () => OGTimeSweep.Generate ~1 writelog);
115         (* write stats to log file *)
116         val _ = writelog ((printconfig config)^" "^
117                  makestring(totalglob)^" "^
118                  makestring(maxnodes)^" "^
119                  (CPN'time)^"\n");
120         val _ = DeleteOccGraph ();
121         val _ = OGSet.StopOptions {Arcs = 0, Nodes = 0,
122                  Predicate = (fn _ => false),Secs = globaltimelimit };
123     in  ()
124     end;
```

Appendix F

Tools for Analysing Multiple Configurations

Verification of the TR-Protocol and Revised TR-Protocol has comprised generating the state space and minimizing the FSA for a large set of different parameter values, or configurations. Design/CPN has standard support for analysing only one configuration at a time. The typical process is to set the initial values in the Design/CPN Editor, switch to the Design/CPN State Space Tool (which takes approximately 5 minutes) and calculate the state space. We have used several Standard ML functions to do a batch calculation of the state space for a set of configurations. Section F.1 describes these functions, and how the Revised TR-Protocol CPN is changed to facilitate this process. Section F.2 lists a set of shell scripts used after the state spaces are calculated for minimizing the FSA and collecting relevant statistics.

The Standard ML functions for calculating batch state spaces are due to Dr Lars Kristensen. Lars wrote most of the functions, although some have been adapted specifically to the Revised TR-Protocol CPN.

F.1 Analysing Multiple State Spaces in Design/CPN

The three parameters used in the Revised TR-Protocol CPN are: the two constants *RCRImax* and *RCRRmax*; and the initial marking of the place UserAck, *UserAck*. Typical use of Design/CPN requires the values of these parameters to be set in the Editor. Once the state space is calculated, we must return to the Editor, change the values, and re-switch to the State Space Tool. This is a time consuming process. To setup the CPN to perform state space analysis of multiple configurations automatically, we designate the two constants as reference variables and introduce a new transition that must occur before any other transition in the CPN. This transition, called ConfigInit, is shown in Figure F.1 along with a new place, called ConfigPlace, with an initial marking of 1`e. The output

place of ConfigInit is a fusion place with UserAck on the I_NULL page (Figure E.5). UserAck in Figure E.5 no longer has an initial marking.

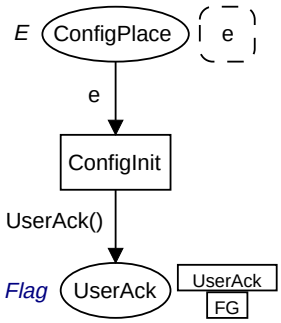

Figure F.1: Initialise page for initialising the Revised TR-Protocol CPN for analysing multiple configurations

When the State Space Tool is entered, the first and only transition that can occur is ConfigInit. The function UserAck() is executed, which reads the value of the reference variable UserAckC. This reference variable is set before the state space calculation starts using the function setcurconfig() (described shortly). The occurrence of ConfigInit effectively gives UserAck an initial marking.

All references to the two constants (RCRImax and RCRRmax) in the Revised TR-Protocol CPN are replaced with the functions RCRImax() and RCRRmax(), respectively. These functions read the value of the corresponding reference variables (RCRImaxC and RCRRmaxC). Again, the values of the reference variables are set before the state space calculation starts.

Listing F.1 gives the new declarations used for analysing multiple configurations. The declarations no longer used in Listing E.1 are commented out (i.e. the last two lines in Listing F.1). Each reference variable is given a default value (e.g. F for UserAckC). The process for calculating multiple configurations is: set the values of the three reference variables (parameters), calculate the state space, record the relevant results, delete the state space, set the parameter values for the next configuration, and so on. The Standard ML code for this process is given in Listings F.2 and F.3.

Listing F.1: Changes to declarations of the Revised TR-Protocol CPN to analyse multiple configurations

```
1  (* --- CONFIGS --- *)
2  color E = with e;
3  val UserAckC = ref F;
4  val RCRImaxC = ref 2;
5  val RCRRmaxC = ref 2;
6  fun UserAck() = (!UserAckC);
7  fun RCRImax() = (!RCRImaxC);
8  fun RCRRmax() = (!RCRRmaxC);
9  (* --- END CONFIGS --- *)
10
```

```
(* Maximum Counter Values *)
(* val RCRImax = 1; *)
(* val RCRRmax = 1; *)
```

Listing F.2 includes a general set of functions used in setting up the analysis of multiple configurations. The functions must be defined before those in Listing F.3, or if sweep-line analysis is being performed, Listing F.4. The functions, with comments, are self explanatory.

Listing F.2: Standard ML code to setup Design/CPN to analyse multiple configurations

```
(*
 * These functions setup Design/CPN so that the state spaces of the
 * Transaction Protocol can be calculated in batch. That is, the set of
 * parameters are instantiated, the state space calculated and results saved
 * and then the next set of parameters instantiated, and so on. These
 * functions must be used in conjunction with those that start the analysis
 * either for ordinary state space (e.g. ss-configs.sml) or sweep-line
 * analysis (e.g. sweep-configs.sml).
 *
 * ACKNOWLEDGEMENT: The majority of these functions were written by Lars
 * Kristensen (University of South Australia), or are at
 * least based on ones that he wrote.
 *)

(* Log file to save statistics *)
val logfilename = ref "config-stats.log";

(* Get current time *)
fun gettime () = Date.toString (Date.fromTimeLocal (Time.now ()));

(* Write message to the log file *)
fun writelog CPN'message =
    let val CPN'file = TextIO.openAppend (!logfilename);
        val _ = TextIO.output (CPN'file, CPN'message);
    in TextIO.closeOut CPN'file
    end;

(* Calculate the CPU time used *)
fun timeprofile CPN'fun =
    let val CPN'timer = Timer.startCPUTimer ();
        val CPN'start = (#usr (Timer.checkCPUTimer CPN'timer));
        val CPN'res = CPN'fun ()
        val CPN'end = (#usr (Timer.checkCPUTimer CPN'timer))
    in (CPN'res,Time.toString (Time.-(CPN'end,CPN'start)))
    end;

(* Print the configuration (i.e. parameter values) *)
(* (int * int * bool) -> string *)
fun printconfig (rcri, rcrr, ua) = "p"^(mkst_col'RCR_c rcri)^"-"^
                    (mkst_col'RCR_c rcrr)^"-"^(mkst_col'Flag ua);

```

```
42 (* Set the stop options *)
43 OGSet.StopOptions {Nodes = 1000000,Arcs= 0,Secs = 12*60*60,
44                     Predicate = (fn _ => false)};
45
46 (* Set the configuration to be analysed *)
47 (* (int * int * int * int * bool) -> unit *)
48 fun setcurconfig (rcri, rcrr, ua) =
49     (RCRImaxC := rcri;
50      RCRRmaxC := rcrr;
51      UserAckC := ua);
52
53 (* Convert a list of integers to a string with each integer sepearated by spaces *)
54 (* int list -> string *)
55 fun printintlist (nil) = "␣"
56   | printintlist (x::xs) =
57     (makestring(x)^"␣"^(printintlist xs));
58
59 (* For each transition return 1 if the transition is dead, otherwise 0 *)
60 (* (TI.TransInst list * TI.TransInst list) -> int *)
61 fun printdeadtis (all,dead) =
62         let     fun map2int a b = if mem a b then 1 else 0
63         in      printintlist ((map (map2int dead) all))
64         end
65
66 (* NOTE: the function IsValidTerminal must have been defined previously *)
67 (* Return 1 if all dead markings are desired *)
68 (* Node list -> int *)
69 fun correctterminals (dm) =
70         if      mem (map IsValidTerminal dm) false
71         then    "0"
72         else    "1";
```

Listing F.3 includes a function for performing state space analysis on one configuration (called fullanalyse_config()), and a function for applying fullanalyse_config() to a list of configurations (called fullanalyse_configs()). A configuration is given as a triple: (*RCRImax*, *RCRRmax*, *UserAck*).

fullanalyse_config() follows the steps:

1. the reference variables are set to the values for the configuration (Line 15),
2. the state space and SCC graph are calculated (Lines 17 and 20).
3. the results are written to a log file (Line 24),
4. the Design/CPN State Space Report is written to a file (Line 40),
5. the state space is mapped to a FSA and the transitions and halt states written to files (Lines 42 and 43), and
6. the state space is deleted (Line 45).

Listing F.3: Standard ML code to analyse multiple configurations of the Revised TR-Protocol

```
(*
 * Performs state space analysis in Design/CPN of multiple configurations of
 * the Transaction Protocol (ie batch analysis). setup-configs.sml must have
 * been included previously.
 *
 * ACKNOWLEDGEMENT: The majority of these functions were written by Lars
 * Kristensen (University of South Australia), or are at
 * least based on ones that he wrote.
 *)
(* Perform single pass (state space) of Transaction Protocol *)
(* (int * int * bool) -> unit *)
fun fullanalyse_config config =
    let
        (* set the current configuration *)
        val _ = setcurconfig config
        (* calculate state space *)
        val (_,CPN'time) = timeprofile CalculateOccGraph;
        val _ = writelog ((printconfig config)^" "^CPN'time^" ");
        (* calculate SCC graph *)
        val _ = CalculateSccGraph();
        (* write statistics to log file *)
        val allti = TI.All;
        val deadti = ListDeadTIs();
        val _ = writelog (makestring(NoOfNodes())^" "
                ^makestring(NoOfArcs())^" "
                ^makestring(NoOfSecs())^" "
                ^makestring(SccNoOfNodes())^" "
                ^makestring(SccNoOfArcs())^" "
                ^makestring(SccNoOfSecs())^" "
                ^makestring(length(ListDeadMarkings()))^" "
                ^ correctterminals (ListDeadMarkings())^" "
                ^makestring(length(ListDeadTIs()))^" "
                ^makestring(UpperInteger(Mark.R_LISTEN'InitToResp 1))^" "
                ^makestring(UpperInteger(Mark.R_LISTEN'RespToInit 1))^" "
                ^ printdeadtis (allti,deadti)^"\n");
        (* write Design/CPN OG Report, FSM transitions and halts to files *)
        val reportfile = (printconfig config)^"ogreport.txt";
        val transfile = (printconfig config)^"trans.txt";
        val haltsfile = (printconfig config)^"halts.txt";
        val _ = OGSaveReport.DumpReport(reportfile,
            OGSaveReport.Report(true,true,true,false,true,true,false,false));
        val _ = og2fsmtrans ArcToFSM transfile;
        val _ = og2fsmhalts FindHalts haltsfile;
        (* delete state space so that next one can be calculated *)
        val _ = DeleteOccGraph ();
    in ()
    end;

(* Calculate state spaces for set of configurations given as a list *)
(* (int * int * bool) list -> unit *)
```

```
51 fun fullanalyse_configs configs =
52     (app fullanalyse_config configs)
53     handle CPN'e => (writelog
54         ("FATAL ERROR: exception: "^(exnName CPN'e)^"\n"));
```

When multiple configurations are analysed using the sweep-line method, the functions in Listing F.4 are used to apply the sweep-line analysis of one configuration to a list of configurations.

Listing F.4: Standard ML code to analyse multiple configurations of the Revised TR-Protocol using the sweep-line method

```
1  (* Apply sweep-line analysis on set of configurations (no properties) *)
2  (* (int * int * bool) list -> bool -> int -> int -> unit *)
3  fun sweepanalyse_configs configs store_arcs gcnodelimit globaltimelimit =
4      (CPN'storearcs := store_arcs ;
5       app (sweepanalyse_config (gcnodelimit, globaltimelimit)) configs)
6      handle CPN'e => (writelog
7          ("FATAL ERROR: exception: "^(exnName CPN'e)^"\n"));
8
9  (* Apply sweep-line analysis on set of configurations (with properties) *)
10 (* (int * int * bool) list -> bool -> int -> int -> unit *)
11 fun sweepproperties_configs configs store_arcs gcnodelimit globaltimelimit =
12     (CPN'storearcs := store_arcs ;
13      app (sweep_properties (gcnodelimit, globaltimelimit)) configs)
14     handle CPN'e => (writelog
15         ("FATAL ERROR: exception: "^(exnName CPN'e)^"\n"));
```

F.2 Minimizing the FSA and Collecting Statistics

Once the state spaces have been calculated for each configuration, their FSAs must be minimized and the languages compared to the TR-Service language. Several scripts have been written so this language analysis can be performed in a batch. The process followed is:

1. Copy files for each configuration (`ogreport.txt`, `trans.txt`, `halts.txt`) into sub-directories (`cpconfigs.sh`, Listing F.5).

2. Minimize the FSA and compare the Revised TR-Protocol language with the TR-Service language for each configuration (`minconfig.sh`, Listing F.6). (Note that the command `minogosi` executes the code in Listing A.2.)

3. Collect statistics on FSAs and languages (`allstats.sh`, Listing F.7).

4. Check if there are any configurations which have a language different from the TR-Service language (`checkdiff.sh`, Listing F.8).

Listing F.9 gives the *awk* scripts that collect statistics on the languages (i.e. the files `langstats.awk`, `countnip.awk` and `countnis.awk`).

Listing F.5: Shell script to copy saved files to subdirectories

```
# Copy all trans, halts and ogreport files generated by Design/CPN from the
# current directory into a set of subdirectories (one for configuration).
#
# Usage: cpconfigs resultsdir listconfigs
# resultsdir - directory name for putting all results (must not already exist)
# listconfigs - file listing all configurations in the format of for example
#               1-1-F on every line
#
# check first parameter is desired directory
if test $1
then s=$1'/p';
else echo 'Usage: cpconfigs resultsdir listconfigs '; exit;
fi
# check second parameter is list of configurations
if test $2
then listconfigs =$2;
else echo 'Usage: cpconfigs resultsdir listconfigs '; exit;
fi
mkdir $1
# for every configuration copy files to its own directory
for i in `cat $listconfigs `
do
    mkdir $s$i;
    cp 'p'$i'ogreport.txt' $s$i'/ogreport.txt';
    cp 'p'$i'trans.txt' $s$i'/trans.txt';
    cp 'p'$i'halts.txt' $s$i'/halts.txt';
done
```

Listing F.6: Shell script to minimize all configurations of the Revised TR-Protocol

```
# Applies minogosi (the script for performing language analysis with FSM)
# to all configurations listed in listconfigs.txt
# Usage: minconfigs listconfigs
# listconfigs - file listing all configurations in the format of for example
#               1-1-F on every line
if test $1
then listconfigs =$1;
else echo 'Usage: minconfigs listconfigs '; exit;
fi
pathawk='../'
for i in `cat $listconfigs `
do
   if test -d 'p'$i
   then
      tt=`basename $i -T`
      if test -d 'p'$tt'-T' # UserAck On
      then
         cd 'p'$i;
```

```
19            if test -e trans.txt
20            then minogosi trans.txt halts.txt on
21              gawk -f $pathawk'langstats.awk' lang.txt >> fsmstats.txt
22              gawk -f $pathawk'countnis.awk' difflang-service.txt >> fsmstats.txt
23              gawk -f $pathawk'countnip.awk' difflang-protocol.txt >> fsmstats.txt
24            else echo
25            fi;
26            cd ..;
27          else # UserAck Off
28            cd 'p'$i;
29            if test -e trans.txt
30            then minogosi trans.txt halts.txt off
31              gawk -f $pathawk'langstats.awk' lang.txt >> fsmstats.txt
32              gawk -f $pathawk'countnis.awk' difflang-service.txt >> fsmstats.txt
33              gawk -f $pathawk'countnip.awk' difflang-protocol.txt >> fsmstats.txt
34            else echo
35            fi;
36            cd ..;
37          fi;
38      else
39          echo;
40      fi
41  done
```

Listing F.7: Shell script to collect FSM statistics from all configurations

```
1  # Collect all the fsm statistics for each configuration and write to a
2  # single file
3  # Usage: allstats.sh
4  total='../allstats.txt'
5  for i in `cat listconfigs.txt`
6  do
7  cd 'p'$i
8  if test -e fsmstats.txt;
9  then
10     echo 'p'$i > name.txt
11     cat name.txt >> $total
12     cat fsmstats.txt >> $total
13     rm name.txt
14     cd ..
15 else
16     cd ..
17 fi;
18 done
```

Listing F.8: Shell script to test for differences between all configurations of the Revised TR-Protocol and the TR-Service

```
1  # Test all configurations if NIP or NIS are greater than 0 ie differences
2  # in languages
3  # Usage: checkdiff.sh
4  for i in `cat listconfigs.txt`
5  do
```

```
cd 'p'$i
if test -e difflang-service.txt;
then
    if test -s difflang-service.txt
    then
        echo $i is incorrect
        cd ..;
    else
        cd ..
    fi;
else
    cd ..
fi;
done
```

Listing F.9: Awk scripts to count number of sequences and the maximum and minimum lengths for a language (`langstats.awk`) and the number of sequences not in the Revised TR-Protocol/TR-Service language (`countnip.awk/countnis.awk`)

```
# langstats.awk
BEGIN           {min=100}
                {if (length($0)<min) min=length($0)}
                {if (length($0)>max) max=length($0)}
END             {print ("LANG: ", NR, max/6, min/6)}

# countnip.awk
END             {print ("NIP: ", NR)}

# countnis.awk
END             {print ("NIS: ", NR)}
```

Appendix G

Evaluation of the Tools and Techniques Used

The major part of the modelling and analysis reported in this thesis has been undertaken using two tools: Design/CPN [109] and FSM [4]. The CPN models were created, simulated and analysed in Design/CPN. A prototype implementation of the sweep-line method was used [30], and to perform analysis of multiple configurations at once, extra Standard ML code was required (see Appendix F). FSM was used for minimizing the FSAs and comparing languages. This appendix provides some comments on the applicability of these tools to the Wireless Transaction Protocol. The purpose is to be an informative guide to other potential or existing users, and also a "wish-list" to be passed on to developers.

G.1 Coloured Petri Nets and Design/CPN

G.1.1 Limitations and Difficulties

Version Control

Many iterations of the verification process were applied to the Wireless Transaction Protocol (due to a variety of reasons). As a result, a large number of changes were made to the CPN models. Efficiently keeping track of these changes was difficult. A significant factor to this is the need for discipline in applying and recording the changes. However, support within Design/CPN for version control would also simplify the process. This may come in the form of a version control system built into Design/CPN, or for Design/CPN to make use of external version control systems. The important aspects to be considered are:

- Saving dates, authors, comments and version information with the CPN models.

- Saving incremental changes (deltas) to the CPN models.
- Including new graphical features so that the changes can be shown (e.g. highlighting the differences between versions, for both display and printing).
- Analysis techniques that take advantage of the incremental nature of the CPN model.

Standard ML

Standard ML is, in most cases, a suitable language for the inscriptions in CPNs. However, we experienced several problems with the constructs that limit the maintainability of our TR-Protocol CPN. For example, the header fields of the PDUs were chosen to be modelled as records rather than products because of the clarity records give in the model (see Section 7.5). But as several PDUs contain the same header fields, there are redundant fields in the records. A more suitable approach would be to define a record of the common fields, and then create the PDU headers from a composition of the common fields and other specific fields. As far as we are aware, this was not possible using Standard ML.

Integration of Tools

To facilitate the rapid development of models and generation of results, further integration of the tools used is desirable. Graphical support for analysis of multiple state spaces and the sweep-line method should be included in Design/CPN. Further development of the library to map the Design/CPN state space to a text file suitable for FSM is also required. With the performance evaluation [101] and code generation [113] features of Design/CPN, these enhancements will be a good step towards a suite of tools supporting the major activities in the protocol engineering methodology (and systems engineering in general).

Generation of Time Sequence Diagrams and State Tables

The time sequence diagrams and state tables in this thesis have been created manually. Automatically generating these from the model and state space is desirable. Design/CPN includes a library for automatically drawing message sequence charts (similar to TSDs) for occurrence sequences [161], but, in its current form, it is inadequate for expressing the level of detail required for the TR-Service and TR-Protocol TSDs. As the state table entries almost have a one-to-one mapping to transitions, a library to transfer the information between the two representations would also be useful. Fully or partially automating these tasks would reduce the possibility of human errors and save significant time.

G.1.2 Evaluation of State Space Analysis

State spaces of more than 50 configurations of the Revised TR-Protocol were calculated in Design/CPN. The largest state space calculated contained 343521 nodes and 1353167 arcs. This took over 4 hours of CPU time. This equates to an average calculation rate of 21 nodes per second, as opposed to rates of more than 100 nodes per second when the number of nodes is less than 50000. Figure G.1 shows how the calculation rate decreases as the number of nodes to calculate increases.

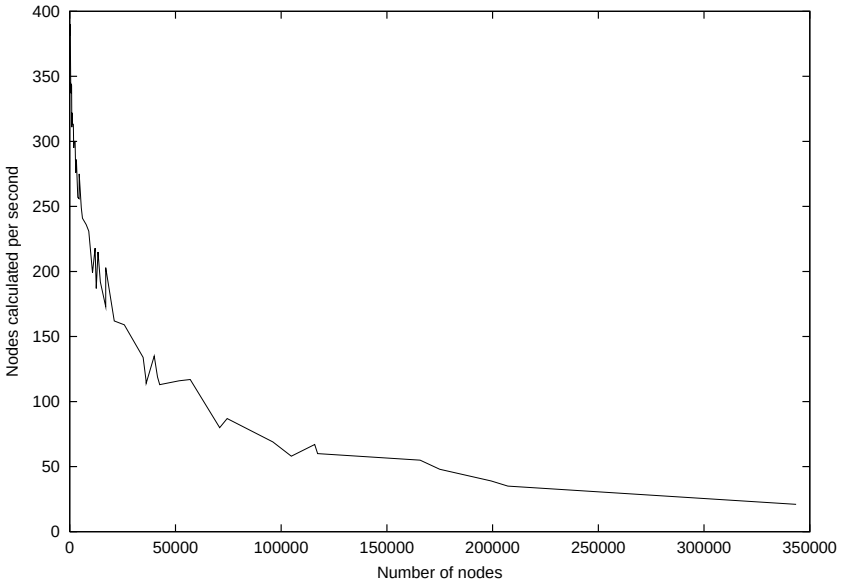

Figure G.1: Relationship between node calculation rate and total number of nodes when using ordinary state space analysis

For the computer used for analysis (see Section 8.2.3), the state space of Configuration 4-3-F could not be calculated. From the results for this configuration and Figure G.1, the limit for Design/CPN is estimated to be approximately 400000 nodes.

G.1.3 Evaluation of the Sweep-Line Method

The sweep-line method has been applied to few case studies [31]. Although the focus of this thesis is verifying the Wireless Transaction Protocol, and the sweep-line method has only been used to analyse several configurations in Chapter 9, a considerable amount of effort has been put into obtaining results using the sweep-line method. These results are used as a preliminary evaluation of the method.

The results using the sweep-line method are compared against the number of nodes and generation time for when ordinary state space analysis is used. The configurations analysed using ordinary state space analysis are grouped into five categories based on the number of nodes (N). The averages of each category are then given. The categories are: Very Small ($N < 1000$); Small ($1000 \leq N < 10000$); Medium ($10000 \leq N < 50000$);

Large ($50000 \leq N < 150000$); and Very Large ($N \geq 150000$). Four different limits for garbage collection are used: 100 nodes, 1000 nodes, 10000 nodes and 50000 nodes. The on-the-fly verification of properties is not used in the sweep-line method, only calculation of nodes is performed. This is because the time to query (or prove) the properties using ordinary state space analysis is not included in the measurements. Further investigation of the impact of the properties on the memory and calculation times should be made.

The graph in Figure G.2 shows the average of the maximum number of nodes stored in memory when the sweep-line method is applied. The memory is relative to that used when the same configuration was calculated using ordinary state space analysis (i.e. 100). There is no advantage of using the sweep-line method when the garbage collection rate is greater than the size of the full state space. For the Large and Very Large state spaces, only 65–70% of the total number of nodes need to be stored when the garbage collection rate is less than 10000 nodes. When the garbage collection rate is at 50000 nodes, the reduction is slightly lower.

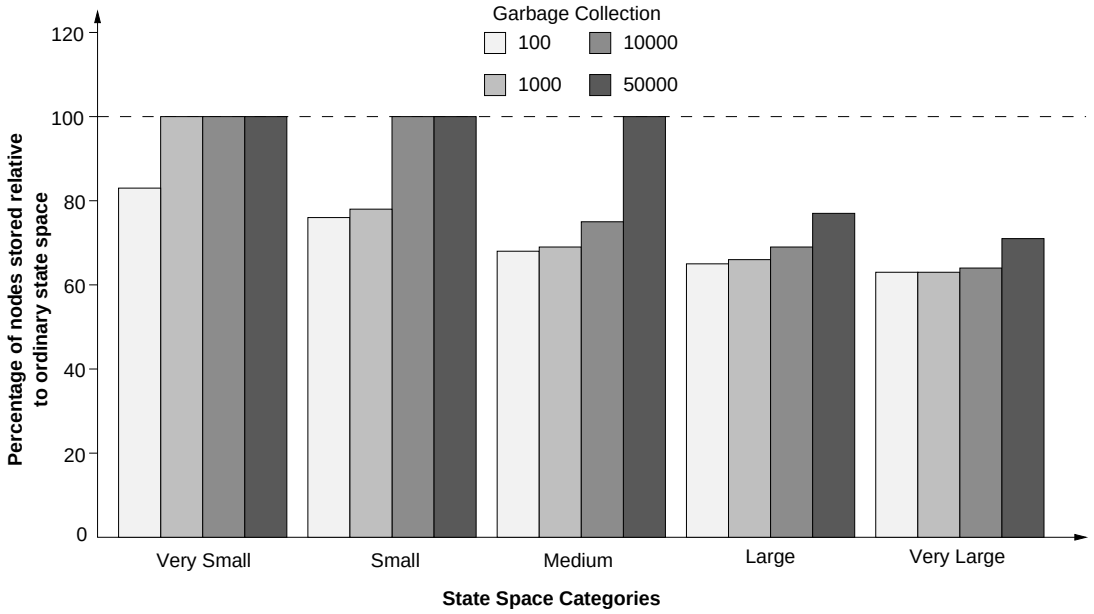

Figure G.2: Number of nodes stored in memory using sweep-line, relative to ordinary state space analysis

Figure G.3 shows the average calculation time using the sweep-line method. Again, the calculation time is relative to the calculation time when ordinary state space analysis is used. There are significant savings in time when using the sweep-line method, especially for Very Large state spaces (approximately 50%).

A trade-off must be made between the calculation time and nodes stored when choosing a garbage collection rate. From Figures G.2 and G.3, a reasonable trade-off is to perform garbage collection every 10000 nodes, where 64% of the total number of nodes need to be stored, and only 47% of the time taken to calculate the full state space is

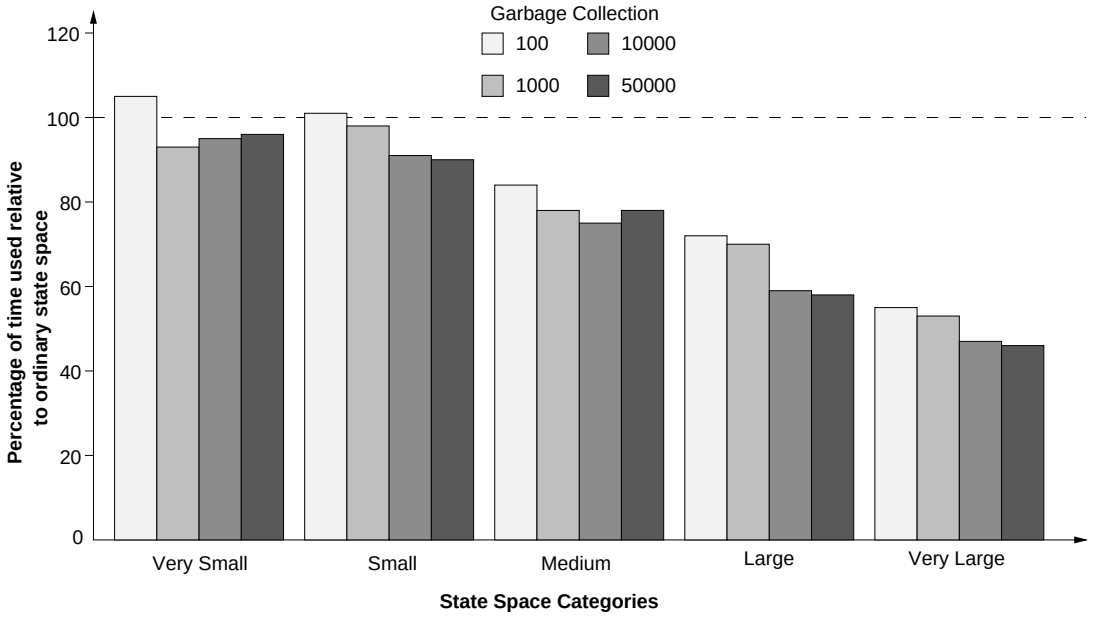

Figure G.3: Calculation time using sweep-line analysis, relative to ordinary state space analysis

required (for Very Large state spaces).

In this thesis, the sweep-line method has been used to obtain the size of state spaces when *RCRImax* is greater than 7 while *RCRRmax*=0 and *UserAck*=T. Figure G.4 plots the maximum number of nodes stored when both ordinary (solid line) and sweep-line analysis (dashed line) are used for different values of *RCRImax* (the state spaces are not calculated using ordinary analysis—the number of nodes are counted when performing the sweep-line analysis). The figure shows that, although the sweep-line method provides a significant reduction for larger state spaces, the number of nodes stored continues to increase rapidly with *RCRImax*. Therefore, the sweep-line method can only provide limited assistance in tackling state explosion for the Revised TR-Protocol CPN.

The progress measure used to perform sweep-line analysis took advantage of the monotonically increasing re-transmission counters, *RCRImax* and *RCRRmax* (see Section 9.3.1). To gain further reductions using sweep-line analysis, other progress measures may be investigated. One approach may be to also include the state names in the progress measure, as each TR-PE traverses through a set of major states.

On inspection of the full state space of Configuration 4-4-T, it was found that 87% of all nodes had a progress measure where either one or both of the TR-PEs had completed the transaction (i.e. returned to the NULL or LISTEN state). Including the state names is likely to provide further reduction in the number of nodes stored, especially for groups of nodes where only one TR-PE has completed the transaction. However, the bottleneck for the reduction occurs when both TR-PEs have completed the transaction. 33% of the total nodes have the same progress measure in this case. This is because there are a large

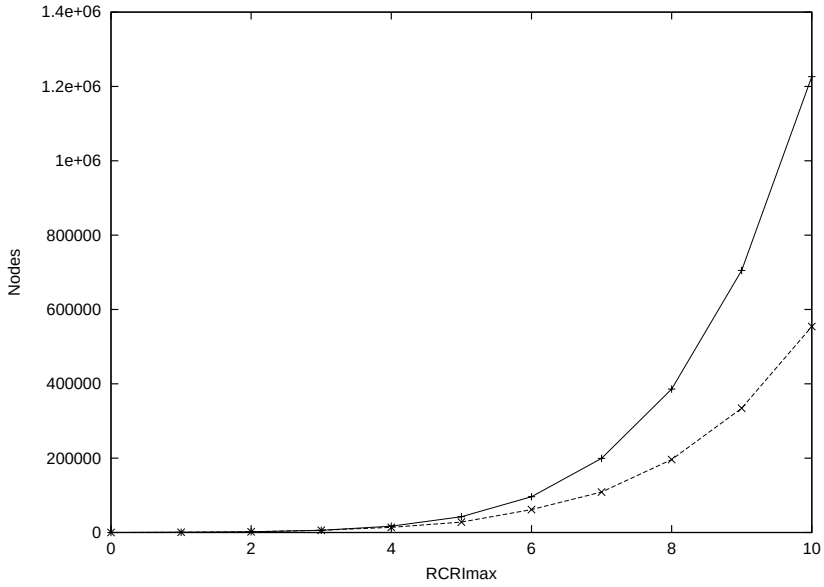

Figure G.4: Maximum number of nodes to be stored when using ordinary state space (solid line) and sweep-line analysis (dashed line) as *RCRImax* increases

number of terminal markings, and also transitions RcvAck_Tve (I_NULL page, Figure E.5) and RcvInvoke_Abo (R_LISTEN page, Figure E.10) can occur multiple times. To overcome this bottleneck, the nodes when both TR-PEs have completed must be sub-divided so they obtain different progress measures. This is an area for further investigation (see Chapter 10).

G.2 Automata Theory and FSM

G.2.1 Limitations and Difficulties

As discussed in Section G.1.1, further integration (possibly via a common format) of Design/CPN and FSM would be beneficial. Also, different (and more user-friendly) techniques for representing the languages (e.g. as regular expressions) are necessary to cope with larger languages. Apart from this, FSM has been a capable tool in the verification methodology. In the long term, it would be desirable for new algorithms to be developed and implemented to minimize much larger FSAs. Intelligent algorithms for drawing FSAs (i.e. taking into account the specific requirements of the user) would also be useful.

G.2.2 Evaluation of the Language Analysis

Although algorithms for removing empty transitions from FSAs and calculating the minimized FSA are exponential in time [9], FSM has coped with the largest FSAs with ease (relative to the time taken to generate the state space in Design/CPN). For example, the state space for Configuration 4-3-F took approximately 4.5 hours of CPU time to calcu-

late, while the language analysis took less than 90 seconds (measured in actual time, not CPU time). Further evaluation of language analysis as part of the protocol engineering methodology would require their application to other case studies that push the tools and techniques to their limit.

Appendix H

Publications

H.1 Journal Articles

S. Gordon and J. Billington. Analysing a missile simulator using Coloured Petri nets. *International Journal on Software Tools for Technology Transfer*, 2(2):144–159, December 1998.

Abstract. The operation of a distributed missile engagement simulator is modelled and analysed. The simulator is developed as a testing platform for missile guidance and control algorithms. The simulator uses concurrency and remote execution to enhance performance. Two Coloured Petri net models are created, simulated and analysed: an abstract model that specifies the service provided by the simulator to the graphical user interface and user, and a detailed model that describes the functionality of the simulator. It is shown that there are no deadlocks when communicating between components of the simulator and the simulator operates correctly. Also, for a set of input parameters, the detailed model provides the service described by the abstract model.

H.2 Conference Papers

S. Gordon, L. M. Kristensen and J. Billington. An approach to generalising the state space of a distributed missile simulator. In *Proceedings of the Eleventh Annual International Symposium of the International Council on Systems Engineering (INCOSE'2001)*, Melbourne, Australia, 1–5 July 2001. Recipient of a Best Student Paper Award.

Abstract. Formal methods can be used to verify refinements made in the design stages of systems. For example, the state space of a detailed design model can be compared with that of an abstract design model to see if it preserves sequences of events. Problems with state space analysis (state explosion, fixed initial states) make this difficult for real applications. In this paper we outline an approach for obtaining a generalised state space

of a distributed missile simulator. The original state space has a repetitive structure. Our aim is to prove that for any initial state, the system will eventually halt, after which we can define a compact graphical representation of the state space, that is independent of the initial state.

S. Gordon and J. Billington. Analysing the WAP Class 2 Wireless Transaction Protocol using Coloured Petri nets. In M. Nielson and D. Simpson, editors, *Proceedings of the 21st International Conference on Application and Theory of Petri Nets (PN2000)*, pages 207–226, Aarhus, Denmark, 26–30 June 2000. Volume 1825 of Lecture Notes in Computer Science, Berlin, Heidelberg: Springer-Verlag. ISBN: 3-540-67693-7.

Abstract. Coloured Petri nets (CPNs) are used to specify and analyse the Class 2 Wireless Transaction Protocol (WTP). The protocol provides a reliable request/response service to the Session layer in the Wireless Application Protocol (WAP) architecture. When only a single transaction is considered occurrence graph and language analysis reveals 3 inconsistencies between the protocol and service specification: (1) the initiator user can receive two TR-Invoke.cnf primitives; (2) turning User Acknowledgment on doesn't always provide the User Acknowledgment service; and (3) a transaction can be aborted without the responder user being notified. Based on the modelling and analysis, changes to WTP have been recommended to the WAP Forum^{SM}.

S. Gordon and J. Billington. Modelling the WAP Transaction Service using Coloured Petri Nets. In H. V. Leong, W. -C. Lee, B. Li, and L. Yin, editors, *Proceedings of the First International Conference on Mobile Data Access (MDA99)*, pages 109–118, Hong Kong, China, 16–17 December 1999. Volume 1748 of Lecture Notes in Computer Science, Berlin, Heidelberg: Springer-Verlag. ISBN: 3-540-66878-0.

Abstract. The Wireless Application Protocol (WAP) is an architecture designed to support the provision of wireless Internet services to mobile users with hand-held devices. The Wireless Transaction Protocol is a layer of WAP that provides a reliable request/response service suited for Web applications. In this paper Coloured Petri nets are used to model and generate the possible primitive sequences of the request/response Transaction Service. From the results we conclude that the service specification lacks an adequate description of what constitutes the end of a transaction. No other deficiencies were found in the Transaction Service.

S. Gordon and J. Billington. Middleware services over satellite networks: A survey of issues. In *Proceedings of the 8th International Aerospace Congress (IAC'99) incorporating the 12th National Space Engineering Symposium (NSES'99)*, NS5.20 on CD-ROM, Adelaide, Australia, 13–15 September 1999.

Abstract. Middleware promises seamless integration of heterogeneous networks and computing/telecommunications environments by providing distributed services to user applications. With Low Earth Orbit satellite networks being deployed as communications infrastructure, it is important that middleware can operate efficiently over these networks. For applications to have good performance, efficient and robust communications between middleware objects is mandatory. Many protocols have been proposed to provide mobility, and to improve data transport over the restrictions imposed by satellite links. This paper provides a survey of the issues and potential solutions, with the aim of stimulating further research in this developing area.

S. Gordon and J. Billington. Modelling and analysis of an air-to-air missile engagement simulator using Coloured Petri nets. In E. O. Tuck and J. A. K. Stott, editors, *Proceedings of the Third Biennial Engineering Mathematics and Applications Conference (EMAC'98)*, pages 225–228, Adelaide, Australia, 13–16 July 1998. Institution of Engineers, Australia. ISBN: 185825-686-X.

Abstract. Computer simulations of a missile engaging its target provide an environment for testing the guidance and control functions of the missile. The accuracy of the tests depends on the detail of the models used and correct communication between the models. This paper addresses the problem of analysing the communication protocols for an Integrated Weapons Simulator (IWS). IWS comprises five components which all exhibit some degree of concurrency via multi-threading and remote execution. Coloured Petri nets, a formal technique, are used to model communication between components. It is shown no deadlocks are present in the communication protocols.

S. Gordon and J. Billington. Applying Coloured Petri nets and Design/CPN to an air-to-air missile simulator. In K. Jensen, editor, *Proceedings of the Workshop on Practical Use of Coloured Petri Nets and Design/CPN (CPN'98)*, pages 1–14, Aarhus, Denmark, 10–12 June 1998. Computer Science Department, Aarhus University. DAIMI PB-532. ISSN: 0105-8517.

Abstract. In this paper the communication mechanisms of a missile engagement simulator are modelled and analysed. The simulator is developed as a testing platform for

missile guidance and control algorithms. The simulation uses concurrency and remote execution concepts to enhance performance. Coloured Petri nets are a well suited formal approach for modelling and analysis of these concepts. Design/CPN is used to create and analyse the model of the simulation. A new requirement of the graphical user interface is identified for the simulation to operate successfully. The communication mechanisms are without deadlocks and are suitable for the simulator.

H.3 Other Publications

S. Gordon and J. Billington. Inconsistencies in the Wireless Transaction Protocol. WAP Forum Input Document. Submitted to the WAP Forum, 19 November 1999.

Executive Summary. Formal analysis of the WAP Class 2 Wireless Transaction Protocol has revealed several inconsistencies in the specification. These are explained, and where possible, changes to the specification are proposed to improve the protocol. The inconsistencies are:

1. The counter RCR may be incremented to a value that is greater than RCR_MAX.
2. Two TR-Invoke.cnf primitives can be delivered to the Initiator user (within the context of one transaction).
3. The TR-Result.req primitive may immediately follow a TR-Invoke.ind primitive at the Responder user when User Acknowledgment is on.
4. A transaction may be aborted without the Responder user being notified.
5. The semantics of "Abort transaction" in the state tables is not defined.

Changes to the state tables are proposed to remedy the first 4 problems. A definition is required in the text for the final problem. Typographical errors in the state tables are also pointed out.