

# Verification of the Session Management Protocol

Verifiering av Session Management Protocol

Karl Palmskog  
palmskog@kth.se  
2006-11-13

Master's thesis in Computer Science (20 credits)  
at the School of Computer Science and Communication,  
Royal Institute of Technology  
Supervisor: Mads Dam  
Examiner: Johan Håstad  
Project commissioned by Yuri Ismailov at Ericsson AB

# Verification of the Session Management Protocol

## Abstract

The Session Management Protocol (SMP) is an important part of a session layer for mobile, disconnection- and delay-tolerant communication developed at Ericsson Research. This layer lets applications use *sessions* instead of sockets to communicate with network endpoints. Such sessions are very persistent, surviving even network outages and changes in network attachment. SMP may run on top of TCP/IP or other network protocols, and handles data integrity and resumption after disconnection and suspension for sessions. It also deals with sending and processing messages related to the communication context.

In this thesis, we formally describe SMP, specify its correctness, and verify several protocol models in the validation language PROMELA by using the model checker SPIN. As a preparation for verification, the protocol is characterized informally in a structured manner. After outlining the modelling and verification approach, we give reports on model checking. In particular, we highlight the detection of an error in data integrity management, and describe how this discovery led to changes in the protocol.

By applying abstraction and separating the concerns of safety and liveness, protocol model complexity was reduced to permit exhaustive verification in SPIN. Model checking results prompt the conclusion that SMP reliability has increased through the verification effort.

**Keywords:** verification, formal methods, session, network protocols, mobility, model checking, formal modelling, state management

# Verifiering av Session Management Protocol

## Sammanfattning

Session Management Protocol (SMP) utgör en viktig del av ett sessionslager för mobil, avbrotts- och fördröjningstolerant kommunikation utvecklat vid Ericsson Research. Lagret låter applikationer använda *sessioner* istället för sockets för att kommunicera med nätverksändpunkter. Sådana sessioner är anmärkningsvärt persistenta – de överlever nätverksfel och ändringar av nätverksanslutningspunkt. SMP kan användas ovanpå TCP/IP eller andra nätverksprotokoll, och hanterar dataintegritet och återanslutning efter avbrott för sessioner. Protokollet sköter också sändning och mottagning av meddelanden relaterade till kommunikationskontext.

I den här rapporten beskriver vi SMP formellt, specificerar korrekthet och verifierar flera modeller i valideringsspråket PROMELA med hjälp av model checkern SPIN. Som en förberedelse för verifieringen karakteriseras protokollet informellt på ett strukturerat sätt. Efter en beskrivning av angreppssättet till modellering och verifiering, rapporteras om tillämpningen av model checking. Speciellt belyser vi upptäckten av ett fel i mekanismen som ser till att dataintegriteten bibehålls, och beskriver hur denna upptäckt ledde till ändringar i protokollet.

Genom att använda abstraktion och separera safety- och liveness-krav reducerades protokollmodellkomplexiteten, vilket tillät fullständig verifiering i SPIN. Resultaten av model checking lämnade utrymme för slutsatsen att tillförlitligheten för SMP ökat efter verifieringsinsatsen.

**Nyckelord:** verifiering, formella metoder, session, nätverksprotokoll, mobilitet, model checking, formell modellering, state management

## Acknowledgements

I would like to thank my supervisor at KTH, Mads Dam, for his encouragement and criticism. I also thank my supervisor at Ericsson AB, Yuri Ismailov, for providing valuable input and support; I am particularly grateful for his excellent summary of the session layer on which section 2.1 is based. Additional thanks are due to Petter Arvidsson and Micael Widell, for readily answering questions about their work and for useful discussions. Most of all, I thank my parents, Göran and Kajsa, for their constant support.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Formal Methods and Protocol Verification . . . . .	2
1.2	Thesis Aim . . . . .	2
1.3	Thesis Structure . . . . .	3
<b>2</b>	<b>Background and Related Work</b>	<b>4</b>
2.1	Session Layer Resurgence . . . . .	4
2.1.1	Problem Situation . . . . .	4
2.1.2	Overview of the Session Layer . . . . .	5
2.1.3	Relevance for Verification . . . . .	11
2.2	Principles of Communication Protocols . . . . .	11
2.2.1	Structure of Protocols . . . . .	11
2.2.2	Design Considerations . . . . .	12
2.2.3	An Example Protocol . . . . .	13
2.3	Concurrent Process Theory . . . . .	14
2.3.1	Origins in Sequential Computation . . . . .	14
2.3.2	Transition Systems . . . . .	15
2.3.3	Process Calculi . . . . .	16
2.3.4	Case Study: CCS . . . . .	17
2.3.5	Case Study: PROMELA . . . . .	18
2.4	Process Specification . . . . .	21
2.4.1	Properties of Concurrent Programs . . . . .	21
2.4.2	Bisimulation Theory . . . . .	22
2.4.3	Modal and Temporal Logics . . . . .	23
2.5	Verification in Theory and Practice . . . . .	24
2.5.1	Computer-assisted Theorem Proving . . . . .	24
2.5.2	Model Checking . . . . .	24
2.5.3	Case Study: The Concurrency Workbench . . . . .	25
2.5.4	Case Study: SPIN and XSPIN . . . . .	25
<b>3</b>	<b>Method</b>	<b>29</b>
3.1	Verification Approach . . . . .	29
3.1.1	Choice of Formal Methods . . . . .	29

3.1.2	Methodological Issues . . . . .	30
3.1.3	Verification Trajectory . . . . .	31
3.1.4	Restrictions and Abstractions . . . . .	32
3.2	Elements of Verification . . . . .	32
3.2.1	Protocol Description . . . . .	32
3.2.2	Protocol Models and Formal Specification . . . . .	32
3.2.3	Verification Report . . . . .	33
<b>4</b>	<b>Results</b>	<b>34</b>
4.1	Overview . . . . .	34
4.2	The Checkpoint Protocol . . . . .	35
4.2.1	Protocol Specification . . . . .	35
4.2.2	Protocol Model and Formal Specification . . . . .	37
4.2.3	Development and Verification History . . . . .	42
4.2.4	Concluding Remarks . . . . .	45
4.3	The Session Management Protocol . . . . .	46
4.3.1	Protocol Specification . . . . .	46
4.3.2	Safety of the Session Resumption Functionality . . . . .	48
4.3.3	Liveness Model of Suspend-Resume Functionality . . . . .	56
4.3.4	Development and Verification History . . . . .	60
4.3.5	Concluding Remarks . . . . .	60
<b>5</b>	<b>Conclusions and Further Work</b>	<b>61</b>
5.1	Result Summary . . . . .	61
5.2	Result Reliability . . . . .	62
5.3	Observations . . . . .	62
5.4	Future Work . . . . .	63
	<b>References</b>	<b>65</b>

# List of Figures

2.1	Shift in viewing the network protocol stack . . . . .	5
2.2	Stack architecture implementation overview . . . . .	10
2.3	Transition system for the first protocol process . . . . .	16
2.4	Execution message sequence chart . . . . .	26
2.5	Message sequence chart of a counterexample . . . . .	27
4.1	Fragment of a checkpoint protocol execution . . . . .	41
4.2	Counterexample in a model of the initial protocol . . . . .	44
4.3	Deadlock in a model of the initial protocol . . . . .	45
4.4	SMP state machine . . . . .	48
4.5	SMP safety model execution . . . . .	55
4.6	Deadlock in an incomplete SMP liveness model . . . . .	58
4.7	SMP liveness model execution . . . . .	59

# Chapter 1

## Introduction

“Every protocol should be considered to be incorrect until the opposite is proven.”

—Gerard J. Holzmann

The original design requirements of the Internet—high robustness and survivability—are reflected in its initial reliance on static mappings between users, computers, network addresses and link addresses. However, the conventional Internet infrastructure for naming, name-resolution and routing is becoming insufficient for the networks of today, which are increasingly heterogeneous and dynamic. All networked entities are becoming mobile—users change devices and devices change access networks; resources and applications change devices. Network addresses may now belong to several different address spaces and be allocated dynamically.

Obstacles in accommodating for these technological advances and changes in behaviour include the inter-layer dependencies in the TCP/IP protocol stack [44, p. 2]. For example, the end-to-end transport layer assumes that the network-layer identifier does not change after establishing a connection; consequently, there is no way to rebind a BSD socket to a new Internet Protocol (IP) address. Hence, traditional sockets cannot be used as endpoints in connections between mobile nodes in which IP addresses, and even Transmission Control Protocol (TCP) ports, may change at any time [21, 4].

A recent approach [2] to providing flexible mobile, disconnection- and delay-tolerant communication involved the design of a session layer [19], including a new network protocol running on top of TCP/IP. The session layer uses extended BSD sockets [56], which preserve network endpoints when changes are made at lower levels in the protocol stack. For locating endpoints, there is a name server which keeps track of which IP addresses, ports and protocols the services in the current domain are using. The session layer enables using a *session* as a network programming abstraction. In contrast to sockets, sessions are highly reconfigurable, relocatable and persistent over time [2, p. 11]. However, using sessions requires explicit

support from the operating system and from network-enabled applications. In some cases, where the communicating endpoints use different address schemes, even infrastructure support in the form of a session gateway may be needed. A proof-of-concept implementation has provided the Linux kernel with session capabilities and is showing promise in terms of both stability and performance [2, ch. 5–6].

## 1.1 Formal Methods and Protocol Verification

While providing a protocol specification is not the only part of designing a layer in a protocol suite, it is one of the most critical. When the interpretation of protocol rules is automated, ambiguity and logical inconsistencies in a specification may have devastating consequences [15, ch. 1]. Protocol designers frequently face errors that are disproportionately hard to discover given a relatively simple informal description [15, p. 26].

The usual way of improving the quality of programs—testing—can in most cases only *detect* inconsistencies and other errors, not guarantee their absence [8]. Researchers have recognized the problem of deciding program correctness [13, 9] and developed various techniques, based on branches of mathematics such as logic, graph theory and automata theory, for unambiguously describing and rigorously analyzing software systems [36, p. 2]. In this field of formal methods, instead of providing test results or other anecdotal evidence of functionality, programs are *verified*. By verification, we mean the act of providing a mathematical proof of program correctness with respect to a formal specification. Since manual proofs are only feasible for smaller programs, formal methods make heavy use of computer-assisted proofs and algorithms for automatic verification.

Communication protocols differ from ordinary computer programs in that they are concurrently executed on distributed systems, and in that they may run, in principle, forever. Hence, protocols cannot simply be understood in terms of input and output; a framework for describing and specifying the properties of concurrent, interacting processes must be used.

## 1.2 Thesis Aim

The goal of this thesis is to formally describe and verify the Session Management Protocol (SMP), a part of the session layer by Arvidsson and Widell. In principle, SMP provides suspend-resume capabilities for any type of network session, but the initial focus is on TCP sessions, allowing reliable data transfers. The challenge in using TCP at the transport layer lies primarily in the fact that SMP and TCP are both stateful protocols, which interact in nontrivial ways during a session.

A secondary aim is to highlight the use of applied formal methods in the development of software systems. Note that the use of formal methods does not usually result in a guarantee of correctness—but nevertheless verification attempts tend to increase system reliability, as has been shown in statistical studies [36, p. 9].

### 1.3 Thesis Structure

The thesis is organized as follows. Material related to the proposed session layer and its protocol, as well as an overview of relevant formal methods in theory and practice, is provided in chapter 2. A choice of verification approach, including choices of formalisms for modelling and specification, is given and discussed in chapter 3. The actual results, which entail structured informal protocol descriptions, models in the selected formalism and reports from verification runs, may be found in chapter 4. Results are then summarized and discussed from a more general perspective, and indications about possible future work given, in chapter 5.

Reading the whole of chapter 2 is not necessary for understanding the results and conclusions; sections 2.1.2, 2.3.5, 2.4.3 and 2.5.4 provide the central background and may be studied in isolation. However, the remaining sections enable a deeper understanding of the methodological choices and verification in practice.

## Chapter 2

# Background and Related Work

“We need many *levels of explanation*: many different languages, calculi and theories for different specialisms.”

—Robin Milner

In this chapter, we describe the proposed session layer as an approach to mobile and nomadic communication. We then introduce formal methods for concurrent programs—of which communication protocols form a subset—and provide an overview of the theory of concurrent processes. Subsequently, frameworks for formal process specification are described, and we review tools for automatic and computer-assisted verification.

### 2.1 Session Layer Resurgence

This section describes the session layer in general and the Session Management Protocol (SMP) in particular. Note that some familiarity with networking and Internet mobility is assumed. A more thorough presentation may be found in Arvidsson and Widell’s thesis [2, ch. 2–3].

#### 2.1.1 Problem Situation

The demand for services is one of the major drivers for increasing the complexity of the Internet. Technological advances create possibilities for the development of new services of social, economic and behavioural significance to modern society. However, these new services often require network functionality beyond that which the current Internet architecture has to offer. Attempts to expand functionality have frequently involved the design of new network protocols within the framework of the TCP/IP suite (e.g. Mobile IP [37] and HIP [32]). Such extensions, however, may not be compatible and may not consider architectural aspects of system design. Because of this,

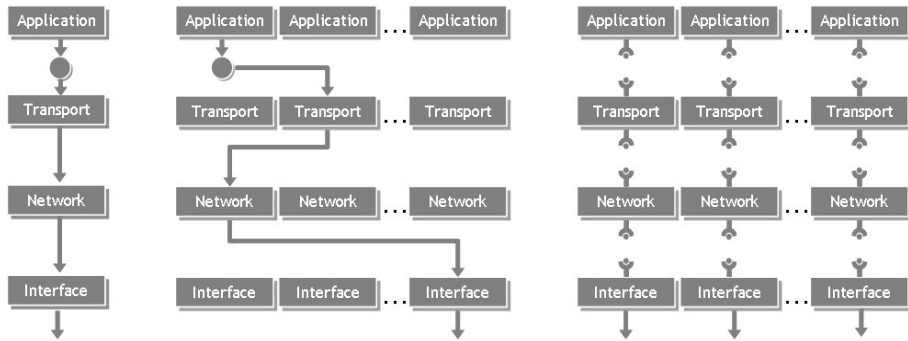
the difficulty in accommodating for new services in a communication infrastructure tends to increase with the number of capabilities that are added. There is also a gap between technology capabilities and utilization; while users demand more simple ways of managing their devices and communication, protocol enhancements may lead to more options and less intuitive controls. The issues that need to be addressed can be summarized thus [44, 21]:

- Ongoing extension of networking functionality from remote resource utilization to information access and distribution
- Increasing network heterogeneity
- Increasing network dynamics
- Increasing demand for security

The simultaneous use of different network addressing schemes, such as IPv4 and IPv6, and the coexistence of wireless and wired networks, are examples of network heterogeneity. Mobility, which will here be defined as a change in network attachment point for a device, and nomadicity, which is the change of location of a device, are examples of network dynamics.

### 2.1.2 Overview of the Session Layer

The issues raised in the preceding section can be partly solved by adopting a flexible view of the network protocol stack, as illustrated in figure 2.1. By removing the static bindings in the logical model and actual view (to the right and in the middle of the figure, respectively), the stack becomes dynamically reconfigurable and thus adjustable to changes in the underlying network infrastructure—as shown to the right in the figure. This is the approach used in the session layer by Arvidsson and Widell, and it is influenced by earlier work on TCP connection passing [1] and the Migrate framework [30, 44, 45].



**Figure 2.1:** Shift in viewing the network protocol stack

To allow for tolerance to disconnections, sessions can enter a suspended state—and in theory, they may remain there for any length of time before resumption.

Major considerations when designing a dynamically reconfigurable stack include: revision of socket design, preservation and restoration of communication states, dynamic updates between communicating endpoints about network changes, naming of communicating endpoints, name resolution, and bridging between networks with different addressing schemes.

## Revised Socket Design

Applications usually use a socket API to access the network. However, sockets also represent a cross-layer mechanism providing important multiplexing and demultiplexing functions for traffic across the stack. Currently, the setup of a path through the stack occurs only at session establishment. A dynamically reconfigurable stack requires adding new multiplexing functions between adjacent layers for handling reconfiguration. For providing such protocol stack path changes, the notion of a Communicating End-Point (CEP) [21, pp. 3ff] is central.

Through context awareness, greater flexibility and efficiency in CEP stack configuration may be achieved. Hence, the proposed architecture uses an event-driven approach to controlling dynamic bindings in the stack. The current design defines system events for each type of networked resource—users, devices, content, and applications. An important feature of the system is the ability to dynamically manage these system events. Events can be added to or withdrawn from the system. When the latter is the case, an occurrence of the selected event will be detected but result in no action. There are few limitations as to what can be the source of an event—for example, events can be generated by the user (via an application) or by the network. Events available in the proof-of-concept implementation include:

User-related events:

- User logon/logoff
- Screen lock/unlock
- No activity during defined time interval

Device-related events:

- New interface configured or disabled or removed from the system
- Interface configuration changed
- Wireless signal strength below or above a defined threshold
- Another wireless network in the vicinity
- TCP retransmits or gives up

Content-related events:

- New volume mount or unmount

Application-related events:

- An application service instance started or stopped
- An application instance changed interface, device, or port number

Every event occurrence is processed by the system; a decision on reaction is reached through the use of a set of rules and preferences defined for each event or combination of events.

## **Naming, Routing and Security**

Names assigned to a CEP consist of three components:

- A globally unique name of a mobile resource on behalf of which the CEP was created. This name is long-lived and belongs to a user, a device, content (e.g. a collection of files), or an application service. In the current design, a Network Address Identifier (NAI) format for this name is used, i.e. `user@realm`.
- A globally unique identifier generated at CEP creation time. This identifier is statistically unique and has the same life-time as a CEP. In the current design, a Universally Unique Identifier (UUID) [20] is utilized.
- A service ID in the form of a text string of a well known service, such as “http”, “cvs”, or “smtp”. This ID is used for connecting incoming session requests to the correct CEP as maintained by the session layer.

The naming components are entirely independent of the underlying networking infrastructure and can thus be dynamically mapped to any network-layer address and transport-layer identifier.

The name resolution system consists of a set of databases holding information relevant for a networked resource. The minimal number of databases is two: a dynamically updated database of current IP addresses mapped to the resource’s name, and a database of resource names mapped to public keys. This supports basic functionality. The set of databases can be extended with, for example, a capabilities base and a context base.

Bridging between heterogeneous networks inside the stack is facilitated through the dynamic reconfiguration mechanism. More specifically, the mechanism allows changing the set of underlying protocols used by a CEP, and thereby the network addressing scheme. Obviously, it is not enough to implement such features only in the stack—infrastructure support is needed as well, since there is no requirement for end-hosts to implement all protocols

in the stack. A CEP that uses a stack which is missing one of the network protocols should still be able to communicate with other CEPs which use this protocol. Another situation that must be handled arises when communicating peers are attached to networks with different addressing schemes. To solve these problems, the architecture specifies a gateway operating at the CEP level.

In the architecture, security is based on an administrative domain local Public Key Infrastructure (PKI). While it does not provide global authentication across administrative domains, it is sufficient to secure dynamic reconfigurations during ongoing sessions. Consider when communication takes place between two peer CEPs. A session setup starts with a name resolution phase where the initiator (client) retrieves an IP address and a public key from the service CEP. The public key is assumed to be mapped to the CEP name assigned to the CEP by the resource (user, device, content, or application) which created it. After successful authentication, the client and the service CEP establish a session. The session can be established either end-to-end or through a CEP (session) layer gateway. In the latter case, the gateway may even act as a proxy for applications using ordinary sockets. After session establishment, the session is managed by SMP.

### **State Management and Reconfiguration**

Communication state management is one of the important characteristics of the architecture. A state can be preserved during arbitrarily long suspend periods and restored when the session resumes. This approach eliminates the dependency of session management on, for example, TCP—specifically TCP timeout mechanisms, which assume an “always connected” scenario. It falls upon SMP to process and transmit information about local changes for an endpoint to the name server and the peer CEP—see section 2.1.2.

One of the central decisions in the architecture concerned where to place the CEP and the corresponding protocol. Note that this is not about assigning a specific layer to support, e.g., mobility. The architecture provides such support across the whole communication stack. Since the proposal requires functionality for performing dynamic reconfiguration at any layer, it is logical to place CEP control and configuration right below the application layer. Thus, the choice fell upon the session layer. However, there is yet another reason to exploit session layer capabilities: the original specification of the session layer [19] introduced a similar suspend-resume functionality along with protocol elements to support it.

### **Stack Extensions**

There are several extensions which the architecture relies on:

- Event collector and dispatcher

- Preferences database
- Processing rules database
- Socket rebind extension
- API for controlling session establishment, management, and termination
- TCP state controller
- Session Management Protocol (SMP)

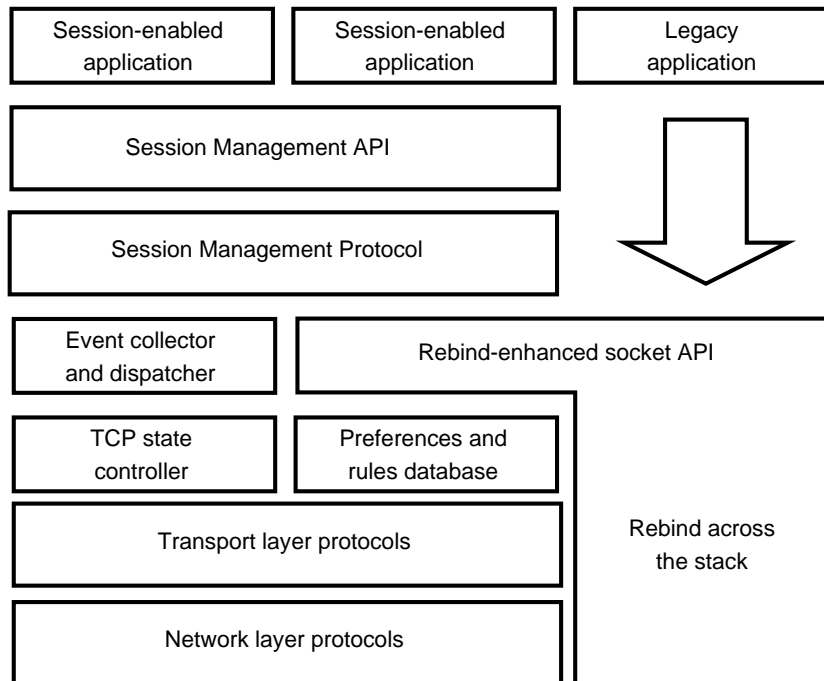
The first three extensions are generic and may be used directly by applications or any other communication software. They support the event-driven nature of the architecture and provide triggering of SMP elements and other system functions to implement dynamic reconfiguration of the stack.

The socket rebind extension [56] performs dynamic stack reconfiguration. This functionality is a cornerstone of the stack design. Leaving existing functionality inside the stack unchanged was a major consideration during the development of the rebind extension. Rebind functionality can be triggered by any of the system events described above. One of the more interesting problems concerns the impact of TCP retransmission algorithms and retransmission timeout values. There has been a lively debate and a lot of work in the area of TCP performance tuning, and one of the identified problems is that when TCP starts retransmission, it does not know in advance if the reason is packet loss or congestion. In the architecture, there is yet another distinct reason for TCP retransmissions—that a peer CEP has been remapped to a different interface, port number, network address, etc. One may argue that this is the same situation as for packet loss. In fact, the TCP retransmission trigger is the packet loss. However, the reaction to packet loss due to the errors on the link and to the remapping of the peer CEP may, and should be, different since the goal is to preserve existing TCP functionality. When TCP starts to retransmit, a device-related event is generated. Relevant TCP state information, such as retransmission timeouts, will be event attributes.

The event collector stores an event until one of two things happen. First, a message of type *resume* may be received on a SMP control channel (see section 2.1.2), meaning that there was a packet loss due to CEP remapping at the other end. In this case, the event dispatcher triggers a reset of TCP timers to their original values (before retransmission) and necessary actions, including remapping of the local CEP according to the SMP state machine, will be performed to resume the session. Second, TCP may give up and attempt to close the socket. In this case, the TCP state controller takes over and keeps the socket until either a resume attempt is triggered or a CEP timeout occurs (which, in theory, may never happen), leading to the removal of the CEP and the end of the session.

How the actual rebind operation is performed in the architecture is influenced by the Migrate approach to mobility [44, pp. 3ff]. The present

design differs mainly in that socket remapping has been extended to allow for port number changes as well as network address changes, and in that the socket is kept in the `ESTABLISHED` TCP state when the session is suspended. Another addition is the mechanism for triggering reactions to system events, and the coupling of that mechanism to preferences and context information, allowing intelligent event processing. Figure 2.2 depicts the proposed stack architecture when partially implemented in the test environment. Another important feature in the architecture is that events causing session suspension can be generated by many sources, such as applications, users, the network and local changes in network configuration.



**Figure 2.2:** Stack architecture implementation overview

### The Session Management Protocol

SMP [2, pp. 50ff] performs several major functions for a CEP. This includes sending notifications about local changes, such as mobility and session suspension, to the peer CEP and to the name server. SMP also keeps track of session data transmission and maintains checkpoints for enabling session resumption. A checkpoint essentially consists of two input-stream references, one for each CEP, and is associated with an integer identifier. Finally, the protocol handles session establishment and termination.

Endpoints exchange two types of messages: control messages and data

messages. The former are used for notifications and sent synchronously on a separate channel that is established anew every time, and the latter are used for application data and checkpoint transmission in the context of an active session. Important control message types include *resume* and *resume\_ok*, which are used for session resumption.

SMP is a stateful protocol: an endpoint may be in a state **ACTIVE**, where application data transfer is possible, or in a state **READY\_RESUME**, in which session resumption is attempted at the earliest opportunity. When a *resume* control message has been sent, a CEP awaits a reply in the state **SENT\_RESUME**. Finally, a CEP may, due to user interaction, be in a state **SUSPENDED**, which indicates that session resumption is not wanted. A diagram of the complete SMP state machine, along with a more formal and complete definition of the protocol, is given in section 4.3.

### 2.1.3 Relevance for Verification

We may now ask which parts of the architecture are relevant for verification. Obviously, SMP depends on infrastructural components, such as the name server and session gateway; while we are not concerned with their verification here, the services they provide must be specified and taken into account. The rebind functionality and certain specifics about the TCP state machine may be treated similarly—what is interesting is not the internal structure, but behaviour during interaction. Verification of security also falls outside the scope of the present investigation.

Since the multitude of events (listed in section 2.1.2) that may occur in the system cannot be treated individually, a uniform approach must be found. One option is that of introducing nondeterminism, as described in section 3.1.4. Of particular interest is also the processing of messages, context management, and data integrity handling, since such issues are directly related to the correctness of the protocol.

## 2.2 Principles of Communication Protocols

The problem of protocol design is as old as communication itself [15, pp. 1ff]. However, protocol implementation on high-speed computers have led to higher demands on procedure consistency and transmission reliability. Consequently, communication protocols have gradually become more sophisticated and grown in size [15, p. 15], as have software systems in general [36, p. 2].

### 2.2.1 Structure of Protocols

Following Holzmann [15, p. 21], a protocol may be specified by providing five elements: a service specification, assumptions about the environment,

message vocabulary, message format, and procedure rules. Usually, both the service specification and the procedure rules are given informally, and thus run the risk of being misinterpreted [15, p. 22].

A protocol's service specification defines what the protocol does in general terms, what its purpose is, and usually provides hints about the context of its usage. Environmental assumptions include the conditions of message transmission: whether and what type of errors can occur, which lower-layer protocols are to be used and which demands are made on the communicating parties. The message vocabulary lists the message types used, while the message format defines the content and layout of message types.

The procedure rules describe how interaction between the communicating parties is carried out. They must guarantee that the service is delivered as specified. Finding the proper set of rules for a specification is usually the hardest part of protocol design [15, p. 21]. There are standardized formal languages for protocol procedure rules, such as LOTOS [52], but no individual language is widely used in practice.

Protocols may themselves be viewed as languages [15, p. 22]: the vocabulary and rules define the syntax, and the service specification defines the semantics. Considering the many deep interrelations of computer programs, automata and formal languages [36, pp. 15ff], it comes as no surprise that many concurrent programs may be viewed as communication protocols, and vice-versa.

## 2.2.2 Design Considerations

Computer protocol design is very involved in terms of the number and range of techniques that must be applied [15, p. 36]. While, e.g., encoding and decoding of data can be handled as an isolated problem, the impact of decisions made at a higher level of abstraction can be hard to grasp fully. Frequently, hidden assumptions about the protocol environment can make themselves known and prompt design reconsiderations. There is no single technique that solves all problems, but a number of heuristics have materialized.

First of all, the problem that the protocol attempts to solve must be well defined. There should be no doubts as to what a solution entails—and this applies also to any subproblems resulting from applying abstraction. A design should use as few and as simple parts as possible, to make analysis easier and maintain efficiency. Different parts should be kept as much apart as is possible, while still being extensible. Ideally, a design should be prototyped and verified before it is implemented.

Techniques from formal methods may be useful during the whole design process, but excel for use in analysis at later stages—refer to section 2.5 in particular.

### 2.2.3 An Example Protocol

We consider a mutual exclusion algorithm by Peterson [38] as a communication protocol—although it is most commonly used as a concurrent program using shared variables. The rationale of this example will become clear later, particularly in sections 2.3 and 2.4.

#### Service Specification

The purpose of the protocol is to let two processes (identified by the numbers 0 and 1, respectively) share a resource in a way such that at most one process accesses the resource at any time. A process that uses the resource is considered to be in its *critical section*.

#### Environmental Assumptions

The processes are unable to communicate directly, but must use a third process as a proxy—the shared memory. Messages to and from memory are sent synchronously and without errors. Access to and release of the shared resource is instantaneous.

#### Message Vocabulary and Format

There are three types of messages, one for each variable used in the algorithm:

$$V = \{b_0, b_1, k\}.$$

A message consists of a message type field and a variable value field. Messages of type  $b_0$  and  $b_1$  only contain boolean values, i.e. *true* or *false*, while messages of type  $k$  only contain integer values in the set  $\{0, 1\}$ .

#### Procedure Rules

Suppose both processes continually want to access the resource. Let both the boolean variables in the memory process be initially *false* and let the integer variable be 0.

Process 0 proceeds by sending a  $b_0$  message with the value *true* to the memory process. After that, it sends a  $k$  message with the value 1 and blocks waiting for messages of type  $b_1$  or  $k$ . If the value field of a received  $b_1$  message is *false*, or the value field of a received  $k$  message is 0, the process accesses the resource; otherwise it continues to wait. When the process has finished using the resource, it sends a  $b_0$  message with the value *false*, and then repeats from the beginning.

Process 1 proceeds in the same way as process 0, but with the boolean variables and the  $k$  values reversed. The memory process nondeterministically receives and sends messages of any type, such that the value in a

message of a type is the same as the value in the message of that type last received. Refer to section 2.5.4 for example executions of a model of the protocol.

## 2.3 Concurrent Process Theory

### 2.3.1 Origins in Sequential Computation

Investigations into the theoretical basis of computing produced two canonical frameworks for expressing sequential programs: Church’s  $\lambda$ -calculus [5] and the Turing machine [51]. The  $\lambda$ -calculus purports to be a formalism for unambiguously describing mathematical functions—but there was initially no agreement on how to deal with its terms in the conventional theory of functions [35, p. 57]. Scott later developed a mathematical theory of partially ordered sets called *domains* [50, pp. 443ff], which was capable of providing satisfactory models for the  $\lambda$ -calculus. Scott showed that sequential programs may be understood as partial functions over memory states—given an initial memory state containing specific values for relevant variables, a program gives instructions on how to produce a finishing state [25, p. 2]. Hence, the domain-theoretic representation of sequential computation formalizes the view of programs as relations between input data and output data, and offers opportunities for extensive mathematical analysis. In Scott’s framework, two syntactically differing programs may be viewed as equivalent if they both denote the same function [57, pp. 55ff].

One attractive property of the Scott-Strachey approach to programming language semantics—the denotational approach—is its *compositionality*; the meaning of a program may be constructed from the meaning of its parts, according to concise definitions by structural induction [57, p. 60]. The compositionality and mathematical rigour of denotational semantics motivated attempts in extending it to deal also with concurrent programs. Plotkin gave a powerdomain construction [39], analogous to the powerset construction of set theory, enabling formal treatment of nondeterministic programs—of which concurrent programs are a special case. Through the use of powerdomains, a nondeterministic program may be viewed as a function accepting initial memory states, producing a *set* of finishing memory states. However, as Milner notes [27, p. 3], parallel composition in a shared-memory environment implies a risk of interference—two programs  $P_1$  and  $P_2$ , denoting the same state-function, may behave differently when composed with a third program that writes to a memory location which only  $P_1$  reads. In a similar way, Owicki and Gries laid great emphasis on the notion of interference when they gave an axiomatic framework for proving the correctness of parallel programs [33].

Interference may be viewed as a form of interaction or communication. This was one of the insights which led Milner to develop a theory of concur-

rent programs in which synchronous, atomic interaction is the basic notion [25, p. 2]. Milner’s work resulted in a Calculus of Communicating Systems (CCS) [25], providing the means of expressing, combining, and reasoning about processes in an algebraic way. Around the same time, Hoare independently showed that atomic interaction can be used to express higher-level concepts in concurrent programming such as semaphores and monitors; the theory he presented was named Communicating Sequential Processes (CSP) [14].

The protocol of section 2.2.3 illustrates Milner’s point that the memory of a concurrent program is not a passive entity, but may be viewed as an independent interactive process [27, p. 3].

### 2.3.2 Transition Systems

Transition systems are a common formal representation of parallel and distributed systems which change states in discrete steps [49, p. 44]. Transition systems may be either labelled or unlabelled, a property which definitions 2.1 and 2.2 make precise.

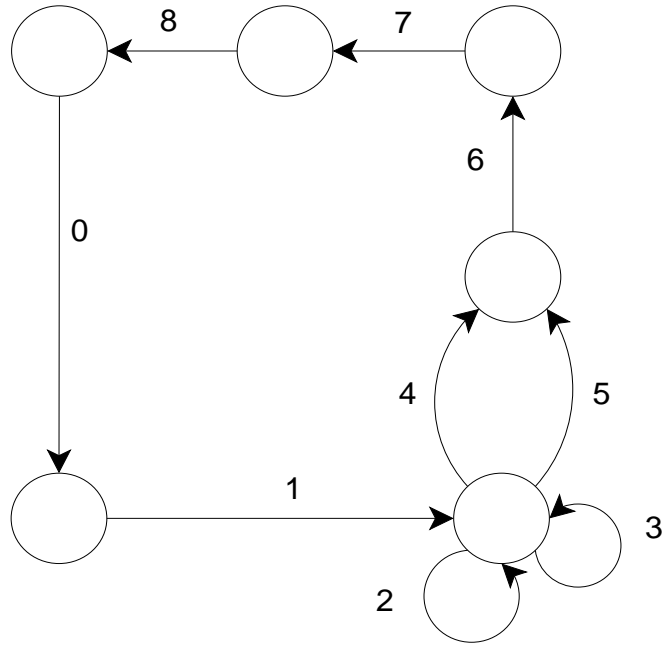
**Definition 2.1** *An unlabelled transition system is a tuple  $T = (S, \rightarrow)$ , where  $S$  is a set of states or configurations and  $\rightarrow \subseteq S \times S$  is the transition relation. We write  $s \rightarrow s'$  when  $(s, s') \in \rightarrow$ .*

**Definition 2.2** *A labelled transition system is a tuple  $T = (S, \rightarrow)$ , where  $S$  is a set of states or configurations, and  $\rightarrow \subseteq S \times A \times S$  is the transition relation, with  $A$  a set of labels. We write  $s \xrightarrow{a} s'$  when  $(s, a, s') \in \rightarrow$ .*

A state in a transition system captures some information about a program at a certain moment in execution [36, p. 67], and the transitions represent actions such as variable assignments and message operations. We do not introduce any explicit notion of time, and assume that a system can block indefinitely before a transition is triggered.

Transition systems are flexible program models, in that concurrency can be accounted for in a reasonably straightforward way: when two systems are run in parallel, the state space of the combined system becomes the cartesian product of the individual state spaces, and whenever a transition is enabled in one of the systems, there is a transition in the combined system.

As an example, we may attempt to construct the first process of the protocol in section 2.2.3 as a labelled transition system. Using natural numbers as labels, we obtain the system in figure 2.3. Here, transitions 0 and 1 represent the transmission of the first two messages, while transitions 2, 3, 4 and 5 represent the different possible responses (two for each variable). The transmission labelled 6 then brings the process to a state where it can access the shared resource. The final transitions, 7 and 8, represent exiting the critical section and the final transmission of a message of type  $b_0$ , respectively.



**Figure 2.3:** Transition system for the first protocol process

### 2.3.3 Process Calculi

Process calculi, or process algebras, are a family of formalisms and theories for modelling and reasoning about concurrent systems. Common features include the use of algebraic laws and transition system semantics and an emphasis on communication and interaction. The calculi attempt, in different ways, to capture relevant aspects of processes and process behaviour, such as equivalences. CSP and CCS, first mentioned in section 2.3.1, are typical examples of process calculi, and define equivalences for many different levels of granularity—see section 2.4.2 for more on this. Another development is that of the Algebra of Communicating Processes [3], which emphasizes the use of an axiomatic methodology for allowing many different process semantics. Note, however, that the focus on foundational issues found in many process calculi means that suitability for use in verification is a secondary concern.

A more recent addition to the ranks of process calculi is the  $\pi$ -calculus [29]. It allows for processes to transmit channels for communication over other channels, making it possible to model mobile systems in a straightforward way. The  $\pi$ -calculus is canonical in so far as the functions of the  $\lambda$ -calculus may be adequately expressed in it [26], and in that higher-order processes can be represented [43]. Milner argues that while  $\pi$ -calculus is too low-level to be commonly used for designing interactive systems, it may

be viewed as a calculus that tie such formalisms together—in the way that differential calculus is used in electrical engineering [28, p. 153].

### 2.3.4 Case Study: CCS

We will briefly describe the syntax and semantics of CCS processes, and explore selected parts of the underlying theory [25]. In CCS, processes can perform *actions* and may, by the use of *combinators*, such as parallel composition ( $|$ ), form new processes. In short, combinators determine how a process may evolve during an execution.

Let  $\mathcal{A}$  be an infinite set of names; use  $a, b, c, \dots$  for denoting elements in  $\mathcal{A}$ . Let  $\overline{\mathcal{A}}$  be the set of co-names to names in  $\mathcal{A}$ ; use  $\overline{a}, \overline{b}, \overline{c}, \dots$  to range over this set. Form now a set of labels  $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$  and identify  $\overline{\overline{a}} = a$  for labels in that set. Define the set of process actions  $Act$  as the union of  $\mathcal{L}$  and the set containing only the silent action  $\tau$ . A simplified syntax of CCS processes may then be given in BNF form thus:

$$P ::= \alpha.P \mid P + P \mid P|P \mid P \setminus L \mid \mathbf{0}$$

Here,  $\alpha \in Act$  and  $L \in \mathcal{L}$ ;  $\alpha.P$  means sequential composition,  $P + P$  means nondeterministic choice,  $P|P$  means parallel composition,  $P \setminus L$  determines a restriction of executable actions, and  $\mathbf{0}$  denotes the inactive process which cannot perform any actions. We use constants starting with an upper-case letter to denote process expressions, as designated in equational definitions—which may be recursive. Let there be a defining process equation  $A \triangleq (P|Q) \setminus L$ , and suppose  $P$  and  $Q$  are capable of performing the actions  $a$  and  $\overline{a}$ , respectively, after which they evolve to  $P'$  and  $Q'$ . If  $a \in L$ ,  $P$  and  $Q$  may *interact* in an atomic silent action  $\tau$ , putting the system in a state  $(P'|Q') \setminus L$ . If  $a$  is not restricted, the system can instead evolve to either  $(P'|Q) \setminus L$  or  $(P|Q') \setminus L$ , which would not be possible otherwise. Note that there is no transmission of data from process to process during interaction, but that such transmission can be simulated by defining appropriate process states and using nondeterminism.

The behaviour of CCS processes may be captured in labelled transition systems, where states are pieces of syntax and actions are transition labels. As an initial example, consider a CCS representation of process 0 of the protocol in section 2.2.3:

$$\begin{aligned} P0 &\triangleq \alpha_0.\alpha_1.P00 \\ P00 &\triangleq \alpha_2.P00 + \alpha_3.P00 + \alpha_4.P01 + \alpha_5.P01 \\ P01 &\triangleq \alpha_6.\alpha_7.\alpha_8.P00 \end{aligned}$$

Now, P0 defines exactly the transitions system shown in figure 2.3, with action subscripts as labels.

As a more advanced example, we provide a CCS model of Peterson’s mutual exclusion algorithm (originally by Walker [55], slightly modified). Let actions containing ‘r’ and ‘w’ represent the reading and writing of a variable, respectively. The memory process, with states for every possible value of the variables, may then be defined as follows:

$$\begin{aligned}
B0f &\triangleq \overline{b0rf}.B0f + b0wf.B0f + b0wt.B0t \\
B0t &\triangleq \overline{b0rt}.B0t + b0wt.B0t + b0wf.B0f \\
B1f &\triangleq \overline{b1rf}.B1f + b1wf.B1f + b1wt.B1t \\
B1t &\triangleq \overline{b1rt}.B1t + b1wt.B1t + b1wf.B1f \\
K0 &\triangleq \overline{kr0}.K0 + kw0.K0 + kw1.K1 \\
K1 &\triangleq \overline{kr1}.K1 + kw1.K1 + kw0.K0 \\
Mem &\triangleq (B0f \mid B1f \mid K0)
\end{aligned}$$

We turn to the communicating processes:

$$\begin{aligned}
P0 &\triangleq \overline{b0wt}. \overline{kw1}. P00 \\
P00 &\triangleq b1rt.P00 + b1rf.P01 + kr1.P00 + kr0.P01 \\
P01 &\triangleq enter.exit. \overline{b0wf}. P0 \\
P1 &\triangleq \overline{b1wt}. \overline{kw0}. P10 \\
P10 &\triangleq b0rt.P10 + b0rf.P11 + kr0.P10 + kr1.P11 \\
P11 &\triangleq enter.exit. \overline{b1wf}. P1
\end{aligned}$$

Next, a set  $L$  of restricted actions are also needed:

$$L = \{b0rf, b0rt, b0wf, b0wt, b1rf, b1rt, b1wf, b1wt, kr0, kr1, kw0, kw1\}$$

We may now construct the final process using parallel composition:

$$Peterson \triangleq (P0 \mid P1 \mid Mem) \setminus L$$

### 2.3.5 Case Study: Promela

PROMELA (Process meta language) [15, ch. 5–6] is a formalism for concurrent processes developed with verification in mind. Compared to CCS, it is more of a high-level language, allowing variable declarations and explicit communication channels. PROMELA syntax resembles that of the C programming language, while other parts are influenced by Hoare’s CSP (mentioned in section 2.3.1) and Dijkstra’s and Hoare’s languages of guarded commands.

A PROMELA model consists of a number of **proctype** process definitions. Variables may be declared to be of global or local scope, and types include

`int`, `byte` and `bit`—as well as `chan` for communication channels. There is no difference between conditions and statements; whether a statement in a process is executed depends on its *executability*, which is determined by the current system state and the statement type. If a process has no executable statements, it blocks. Hence, executability may be exploited to achieve synchronization. Typically, a boolean condition is used to make sure a process proceeds execution only when variables have assumed certain values. Note, however, that assignment statements are always executable.

Communication channels can be either buffered or unbuffered, which means both synchronous and asynchronous message passing can be modelled. PROMELA channels may also transmit channels over channels, providing a flexible way of modelling mobile systems. Channels are declared to permit only transmission of values of certain types. For example, we may declare an unbuffered channel `mem` for messages that contain a message type and a `bit` value as follows:

```
chan mem = [0] of {mtype,bit};
```

Now, the statement

```
mem!b0,0
```

sends a message of type  $b_0$  containing the value 0 using `mem`. The message may be received by another process using the statement

```
mem?b0,b0v
```

where `b0v` is the variable in which the received value is stored. Sending and receiving from buffered channels is non-blocking if the channel is non-full and non-empty, respectively. Executability of operations on unbuffered channels depends on whether there is another process that is able to perform a matching operation, similar to interaction in CCS.

PROMELA processes may be interpreted as (unlabelled) transition systems, and process definitions are in fact converted to finite automata during verification in the model checker SPIN [17, p. 2]. The language also has features for specification of process behaviour, but this will be returned to later in sections 2.5.2 and 2.5.4.

We now attempt to construct a PROMELA model of Peterson’s mutual exclusion algorithm, as was done in CCS in the preceding section. We first let there be message types for the variables, and declare global variables to keep track of when a process is in its critical section.

```
mtype = {b0,b1,k};
bool proc0InCrit = false;
bool proc1InCrit = false;
```

Next comes the definition of the `init` process, which creates the message channels and starts the communicating processes.

```

init {
  chan mem0 = [0] of {mtype,bit};
  chan mem1 = [0] of {mtype,bit};
  run Memory(mem0, mem1, false, false, 0);
  run Process0(mem0);
  run Process1(mem1);
}

```

We require of the memory process that it is able to receive and send variable values when appropriate, and that it proceeds in doing so indefinitely. The **do** construct provides one way to achieve this. (The **if** construct works in a similar manner, but does not cause repeated selection.) Current  $b_0$ ,  $b_1$  and  $k$  values are stored in the corresponding parameter variables.

```

proctype Memory(chan mem0, mem1; bit b0v, b1v, kv) {
  do
    :: mem0?b0,b0v;
    :: mem0?k,kv;
    :: mem1?b1,b1v;
    :: mem1?k,kv;
    :: mem0!b1,b1v;
    :: mem0!k,kv;
    :: mem1!b0,b0v;
    :: mem1!k,kv;
  od;
}

```

The processes may now be expressed according to the procedure rules.

```

proctype Process0(chan mem) {
  BEGIN:
  mem!b0,true;
  mem!k,1;
  do
    :: mem?b1,false;
    break;
    :: mem?b1,true;
    :: mem?k,0;
    break;
    :: mem?k,1;
  od;
  proc0InCrit = true;
  proc0InCrit = false;
  mem!b0,false;
  goto BEGIN;
}

```

```

proctype Process1(chan mem) {
BEGIN:
  mem!b1,true;
  mem!k,0;
  do
    :: mem?b0,false;
      break;
    :: mem?b0,true;
    :: mem?k,0;
    :: mem?k,1;
      break;
  od;
  proc1InCrit = true;
  proc1InCrit = false;
  mem!b1,false;
  goto BEGIN;
}

```

## 2.4 Process Specification

While the presented fragments of concurrency theory are worthy of study in their own right, they are presented here for the purpose of protocol verification; they provide a means for expressing and reasoning about concurrent programs in an unambiguous way. Here, we review approaches to formal specification of the behaviour of processes described in some formalism.

### 2.4.1 Properties of Concurrent Programs

The correctness of a sequential program may be decomposed into partial correctness with respect to specification, and termination [36, pp. 179ff]. In brief, partial correctness requires that if an output is produced, it is a satisfactory solution to the problem, while termination requires that the program eventually halts for all valid input. It is a theorem of computability theory that there is no algorithm that decides for arbitrary programs whether they are correct [36, p. 26].

A formalism limited to expressing partial correctness and termination is easily seen as insufficient for specifying the correctness of communication protocols and distributed systems in general. The most obvious reasons are those of interaction and nontermination—a concurrent program cannot be understood simply in terms of pre- and postconditions. The generalizations of partial correctness and termination to concurrent programs are usually called safety and liveness, respectively [22]. A common formalization of

safety for transitions systems is as a property which holds in each initial configuration and is preserved by each transition [49, p. 51]. Liveness, informally deadlock-freedom and a property that is eventually or sometimes true in all configurations, may be defined in terms of well-founded partial orders [49, p. 53].

Two basic approaches to program specification may be identified [23, p. 2]: the *constructive* approach, where an abstract model of behaviour is provided, and the *axiomatic* approach, where program properties are stated in a logic. We explore these approaches in order below.

## 2.4.2 Bisimulation Theory

A bisimulation is an equivalence relation between labelled transition systems. Informally, two systems are bisimilar if they behave in externally indistinguishable ways to an observer. Bisimulations are a refinement of a notion of behavioural equivalence for CCS processes [48, p. 1].

A natural way to understand bisimulations is through the use of games [48, p. 2]. Suppose there are two players,  $A$  and  $B$ , and two labelled transition systems,  $T_0$  and  $T_1$ . A round in the game consists in  $A$  choosing a transition in either  $T_0$  or  $T_1$ , and  $B$  choosing a transition with the same label in the other transition system, with full knowledge of  $A$ 's choice. If, after a round, the states  $s \in T_0$  and  $s' \in T_1$  reached are distinguishable, i.e. there is at least one transition label available in one state that is not available in the other state,  $A$  wins.  $B$  wins if the game is infinite or if a round ends in no transitions being available in any current state. Now,  $T_0$  and  $T_1$  are bisimilar, denoted  $T_0 \sim T_1$ , if there exist a set of rules for  $B$  to follow which guarantee that, irrespective of what moves  $A$  make,  $B$  wins.

The notion of two states being distinguishable is open to modification. For CCS processes, we may allow process states to differ in their silent ( $\tau$ ) transitions, and still be considered indistinguishable. The resulting equivalence relation is called a weak bisimulation, and is used extensively for specifying the process behaviour in process calculi such as CCS. We denote weak bisimilarity between two systems  $T_0$  and  $T_1$  by  $T_0 \approx T_1$ .

Suppose we have chosen the constructive approach to specifying the behaviour of the CCS process *Peterson* of section 2.3.4. An abstract process model of system behaviour may then be expressed thus [55]:

$$Spec \triangleq enter.exit.Spec$$

We may then reduce the problem of determining the correctness of Peterson's algorithm to the question whether the *Peterson* and *Spec* processes are weakly bisimilar, i.e. whether

$$Peterson \approx Spec$$

To see that this suffices for verification, assume that the statement is true. Then, there are no observable sequences in the *Peterson* process where the `enter` action is followed by another `enter` action, only interleavings of `enter` and `exit` in the proper order. Hence, only one process is in the critical section at any stage.

Another interesting example of the constructive approach to specification can be found as a part of the verification of a CSMA/CD protocol with CCS [34].

### 2.4.3 Modal and Temporal Logics

Modal and temporal logics are a family of formal systems for expressing and reasoning about statements qualified in terms of modalities and time. Modal logics were originally invented for reasoning about necessity and possibility of predicates over logically possible worlds. Two of the basic operators in the logics are  $\Box$  and  $\Diamond$ ; the former may be interpreted as “necessarily”, while the latter may be interpreted as “possibly”. In a temporal logic,  $\Box$  and  $\Diamond$  are usually interpreted as “always” and “eventually”, respectively.

It was discovered that temporal and modal logics were appropriate for specifying the behaviour of programs—in particular, nonterminating programs such as protocols and operating systems. The idea of a specification logic is, given a system model  $M$  and a formula  $\psi$  in the logic, to ask whether  $M$  satisfies  $\psi$ . For most modal and temporal logics,  $M$  must be a labelled or unlabelled transition system.

Linear Temporal Logic (LTL) [40] is a temporal logic suited for specifying properties of indefinitely executing systems. Formulae in this logic are interpreted over infinite sequences of states  $s_0, s_1, \dots$  produced by, e.g., finite-state transition systems—such as those defined by the CCS and PROMELA models above of the mutual exclusion protocol of section 2.2.3.

We briefly give the semantics of LTL formulae containing the operators  $\Box$  and  $\Diamond$ . The full logic has more operators and is more expressive. Suppose there is a suitable static logic for expressing properties of individual states. Denote by  $\xi^k$  the  $k$ th prefix of an execution  $\xi = s_0, s_1, \dots$ . Say that a formula  $\phi$  in the static logic holds of  $\xi^k$  if and only if  $\phi$  holds of the first state in  $\xi^k$ . Let  $\Box\psi$  and  $\Diamond\psi$  hold of  $\xi^k$  if and only if  $\psi$  holds in every prefix of  $\xi^k$ , and if and only if there is a prefix in  $\xi^k$  which  $\psi$  holds, respectively. We claim that  $\psi$  holds of  $M$  exactly when  $\psi$  holds of all executions of  $M$ . An LTL specification of the correctness of the PROMELA model of Peterson’s mutual exclusion algorithm in section 2.3.5 may now be formulated thus:

$$\Box(\neg(p0c \wedge p1c))$$

where we let

```
#define p0c proc0InCrit == true
#define p1c proc1InCrit == true
```

Essentially, the specification claims that, for all executions, it is always the case that not both `proc0InCrit` and `proc1InCrit` are true.

## 2.5 Verification in Theory and Practice

This section describes different approaches to verification, given the theory fragments presented in the above sections.

### 2.5.1 Computer-assisted Theorem Proving

Theorem proving [35] is a *proof-based* approach to verification of software systems. It builds upon interaction between a human verifier, which provides the mathematical theory to be proved, and a theorem prover program, which provides facilities for formal reasoning and simplification, and guarantees soundness. Using a built-in framework for program code or transition systems, the verifier can attempt to express the system specification in a logic, and proceed in steps towards a full proof of correctness.

Certain parts of a proof can typically be carried out by the program, but the verifier carries a heavy burden, even for straightforward proofs. Several thousands of manual steps may be required. Recent developments indicate that theorem proving may be successfully integrated with model checking [6, p. 295], which is described in the next section.

### 2.5.2 Model Checking

Model checking is a *model-based* approach to verification. An ideal model checker accepts as input any system model  $M$  and a specification  $\psi$  in the verifier's formalisms of choice, and quickly and automatically produces an answer to whether  $M$  satisfies  $\psi$ . However, results in computability and complexity theory [36, p. 139] rule out the existence of such a program. When the expressiveness of modelling and specification languages have been restricted to allow decidability, tractability becomes a major concern. As a result of the composition of many parallel components, concurrent systems tend to have exceptionally large state spaces.

Two strategies to model checking temporal specifications are common in the literature [6]: the logic-symbolic approach, in which transition systems are given as binary decision diagrams (BDDs)—minimal representations of boolean functions as graphs—and algorithms which use boolean formula transformations, and the automata-theoretic (enumerative) approach, in which both transition systems and modal logic specifications are represented as automata over infinite words. Both approaches have been successfully implemented in tools. It has been suggested [36, p. 175] that BDD-based model checkers are more suitable for hardware verification, since hardware tends to be synchronous and have a more regular structure than software.

Using the proper framework and algorithms when automatically verifying a system may not be enough. Applying abstraction to the model to remove details not relevant for correctness, and customizing the model to the chosen formalism and model checker are also important.

### 2.5.3 Case Study: The Concurrency Workbench

The Concurrency Workbench (CWB) [7] is a command-line based tool for modelling and verifying the behaviour of processes in variants of CCS. Particularly useful features include the ability to check for bisimulations and deadlocks. Model checking of process properties in a very powerful logic, the modal  $\mu$ -calculus, is also possible.

CWB allows verification of the CCS model of Peterson’s algorithm from section 2.3.4. In CWB syntax, the model and specification is defined as follows:

```

agent B0f = 'b0rf.B0f + b0wf.B0f + b0wt.B0t;
agent B0t = 'b0rt.B0t + b0wt.B0t + b0wf.B0f;
agent B1f = 'b1rf.B1f + b1wf.B1f + b1wt.B1t;
agent B1t = 'b1rt.B1t + b1wt.B1t + b1wf.B1f;
agent K0 = 'kr0.K0 + kw0.K0 + kw1.K1;
agent K1 = 'kr1.K1 + kw1.K1 + kw0.K0;
agent P0 = 'b0wt.'kw1.P00;
agent P00 = b1rt.P00 + b1rf.P01 + kr1.P00 + kr0.P01;
agent P01 = enter.exit.'b0wf.P0;
agent P1 = 'b1wt.'kw0.P10;
agent P10 = b0rf.P11 + b0rt.P10 + kr0.P10 + kr1.P11;
agent P11 = enter.exit.'b1wf.P1;
set L = {b0rf,b0rt,b0wf,b0wt,b1rf,b1rt,b1wf,b1wt,kr0,kr1,kw0,kw1};
agent Peterson = (P0 | P1 | K0 | B0f | B1f)\L;
agent Spec = enter.exit.Spec;

```

We may now use the `eq` command to check for weak bisimilarity:

```

eq(Peterson,Spec);

```

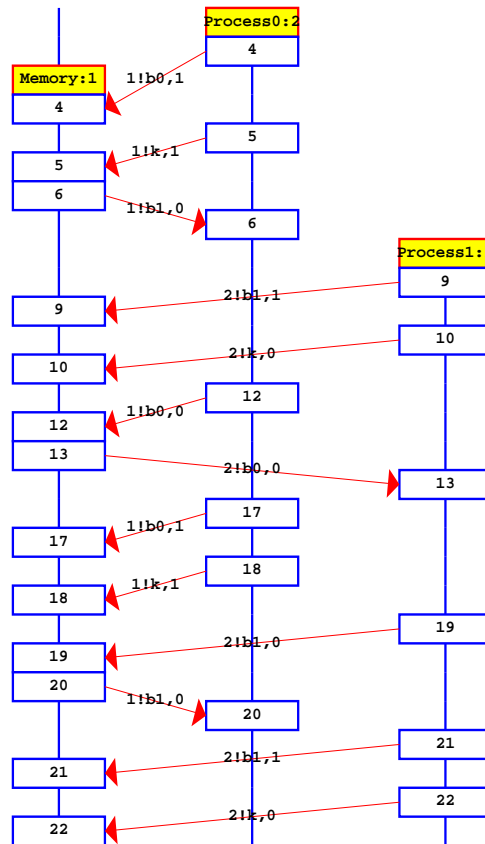
The output `true` confirms that there is indeed a bisimulation relation.

### 2.5.4 Case Study: Spin and Xspin

SPIN [18] is the most mature and widely-used enumerative (on-the-fly) model checker. It is especially suited for verifying computer protocols and other concurrent systems involving asynchronous message transmission, but allows synchronous channels as well.

SPIN accepts PROMELA input and converts a model and its specification into the source code of a customized verifier program, `pan.c`. The program may then be compiled using various options affecting verification. To make

model checking decidable, state spaces cannot not be infinite. SPIN can also simulate executions of PROMELA models, either with the user guiding the simulation or using a random seed. Using the XSPIN graphical frontend, executions may be visualized in message sequence charts. Figure 2.4 shows the chart of an execution of the PROMELA model given in section 2.3.5.



**Figure 2.4:** Execution message sequence chart

Process behaviour may be given in the form of LTL formulae or as **never** claims, with the former being converted to the latter before a verification run. In this way, a large set of properties—called the  $\omega$ -regular properties [36, p. 127]—of PROMELA models are expressible. The customized verifier program goes through the state space and reports any specification violations. Nondeterminism in PROMELA models become an important factor during such traversals, causing executions to divide up into several branches which must all be accounted for. If XSPIN is used, counterexample executions may be shown in message sequence diagrams. Consider the do

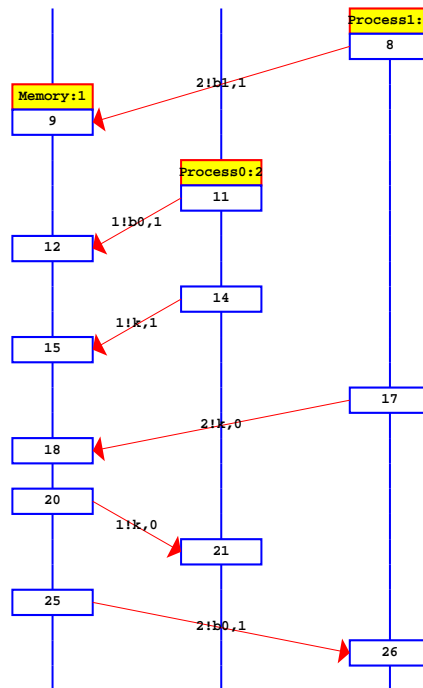
construct in `Process1` in the PROMELA model of section 2.3.5, and suppose the first `break` statement was moved down one sequence to produce the following:

```

do
  :: mem?b0,false;
  :: mem?b0,true;
  break;
  :: mem?k,0;
  :: mem?k,1;
  break;
od;

```

During attempted verification, SPIN now detects the counterexample execution shown in figure 2.5; `Process1` receives a  $b_0$  message with the value *false* and enters the critical section, while `Process0` has not yet left it.



**Figure 2.5:** Message sequence chart of a counterexample

Options for optimization available during compilation of the customized verifier program include compression (`COLLAPSE`), DMA encoding (`DMA=x`) and partial-order reduction (enabled by default, `NOREDUCE` to turn off). The program itself, `pan`, offers one of the most important options `-m`, the argument of which specifies how many transitions deep in the state space a

depth-first search may descend. According to standard practice, the discovery of a counterexample at a certain depth prompts another verification run at a lower depth to find the shortest possible counterexample.

To exhaustively verify our PROMELA model of Peterson's algorithm against the LTL specification given in section 2.4.3, we execute the following commands:

```
$ spin -a -X -N peterson.promela.ltl peterson.promela
$ gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=128 -DXUSAFE \
  -DNOFAIR pan.c
$ ./pan -v -X -m10000 -w19 -a -c1
```

As expected, no counterexample executions were found in the model. The model was also successfully checked for invalid endstates (deadlocks):

```
$ spin -a -X peterson.promela
$ gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=128 -DSAFETY \
  -DNOCLAIM -DXUSAFE -DNOFAIR pan.c
$ ./pan -v -X -m10000 -w19 -A -c1
```

# Chapter 3

## Method

“Profound truths are not be expected of methodology.”  
—Karl Popper

In this chapter, the choice of formalisms and tools for verification, along with a verification trajectory, is presented and discussed. Subsequently, an attempt is made to outline what results are to be expected in practice, and how these results relate to the trajectory.

### 3.1 Verification Approach

#### 3.1.1 Choice of Formal Methods

The choice of verification approach can be broken down into three distinct choices: formalism for modelling, formalism for specification and method of checking whether the model satisfies the specification. The choices may be, but are not necessarily, interconnected; some tools are flexible with respect to formalisms accepted as input while others are not. It may also be possible to convert a model in a specific formal language to another while preserving the important properties.

We required a modelling formalism which allowed straightforward representation of mobile, concurrent processes, and in which high-level concepts such as channels, messages and variables were easily expressed—since their use was prevalent in the informal design and the proof-of-concept implementation. We also required considerable flexibility and expressiveness of the specification formalism, and an effective tool carrying out the actual verification.

CCS and  $\pi$ -calculus were considered for modelling the protocol, but the former was lacking in expressiveness for capturing mobility of processes, and the latter was lacking in tool support. More precisely, the Mobility Workbench (MWB) [54], a tool similar to CWB for reasoning about  $\pi$  processes, was extensively evaluated—no faults were found in modelling capa-

bilities, but performance, especially for bisimulation checking of relatively large models, was deemed unsatisfactory. In fact, the maturity of SPIN compared to other model checkers prompted an attempt at the development of a translation procedure of  $\pi$  processes to PROMELA processes [47]. In the subsequent comparison of model checking in MWB versus model checking in SPIN, doubts about MWB performance were confirmed.

The BDD-based model checker SMV [24] was considered, but the fact that there was no explicit notion of communication channels made the choice untenable, since literate protocol models were a priority. As may be seen in industrial applications of SMV, such as the verification of the cache coherence protocol Futurebus+ [6, pp. 112ff], SMV models are not easily understood without experience in the formalism.

The proof assistant Isabelle offered an interesting framework for verification of distributed systems in the form of I/O Automata [31]. However, the demands the proof assistant placed on the deductive capabilities of the verifier were deemed too high given the time constraints.

Finally, SPIN was chosen as the platform for protocol verification. This choice was mainly due to the special features for communication protocol verification provided by PROMELA, and the attested efficiency of SPIN's automata-theoretic approach to model checking [53]. SPIN's versatility in formal specification and the possibility of modelling mobile processes in a straightforward manner by transmitting channels over channels in PROMELA were also factors in the choice. In addition, the use in PROMELA of familiar programming constructs and explicit message passing boded well for model literacy.

### 3.1.2 Methodological Issues

There are some fundamental problems that need to be dealt with when using a model-based approach to verification. Of particular concern is the relation between the model and the actual system being modelled. Translations to and from different formalisms nearly always results in the loss of some level of detail. When it comes to finding errors by testing, no model is as good as the implementation itself [10, p. 10]. Also, while we may establish once and for all whether a model satisfies a formal specification, we cannot conclusively answer the question of whether the model captures all relevant aspects of a real-world system. To quote Popper [41, p. 285]:

“In so far as a calculus is applied to reality, it loses the character of a *logical* calculus and becomes a descriptive theory *which may be empirically refutable*; and in so far as it is treated as irrefutable, i.e. as a system of *logically true* formulae, rather than a descriptive scientific theory, it is not applied to reality.”

Since the theory that a model, or a set of models, captures all relevant properties of the protocol is refutable, it is important that any results obtained are reproducible. The reasoning which prompted the formulation of a model should also be elaborated—in particular, environmental assumptions must be described. Even with an awareness of the problems involved, the risk of the protocol models becoming too far removed from the design and implementation was considered significant enough to warrant attention throughout the whole verification process.

### 3.1.3 Verification Trajectory

In the adopted approach to verification using model checking, the trajectory consists of a modelling phase and a verification phase; the modelling phase takes a system description and requirements as input and produces a model  $M$  and a specification  $S$ , optionally by repeated use of simulations [42, p. 47].

During the verification phase, specification properties, along with suitable abstractions of the model, are given as input to the model checker in attempts at falsification. In more detail: given a nonverified specification property  $\phi_i$  in  $S$ , an abstraction  $M_i$  of  $M$  is constructed, and falsification of  $M_i$  with respect to  $\phi_i$  is attempted. Error detection may prompt changes in the system, the model, or the specification, reverting the process to earlier stages. Verification is not considered complete until falsification of all properties has been attempted and failed. Errors encountered during falsification fall into the following categories [42, p. 57]:

- Property error—the safety or liveness property is incorrectly stated
- Abstraction error—an abstraction introduced during modeling is erroneous
- Modelling error—the model is an incorrect representation of the protocol
- Design error—the protocol design is somehow flawed

Ideally, any errors found should fall into the last category, but it may be hard to discern error types. In fact, as per the previous section on refutability, the statement that a detected counterexample is the result of a design flaw is not provable within the framework of the model checker.

The steps in the trajectory mapped well to practical steps for verification in SPIN. Construction of models and specifications corresponds to devising and refining PROMELA models and LTL formulae; visual simulation of models is facilitated through XSPIN. Model-property falsification means doing an exhaustive verification run, while error detection is discovery and interpretation of counterexample executions (both demonstrated in section 2.5.4).

### 3.1.4 Restrictions and Abstractions

Because of model checking complexity, the content of a model of a protocol—in PROMELA and in other formalisms—must be restricted to capture those properties relevant for intended usage. Based on Arvidsson and Widell’s thesis, two aspects of SMP seemed particularly important: deadlock issues in the protocol’s state machine, and integrity of application data transfers.

The focus on deadlock-freedom and data integrity allows messages to be stripped of any details not related to checkpoints or state machine transitions. Also, while disconnection and mobility must be present, such behaviour can be captured by using nondeterminism in the transition system model. For example, wherever an event that results in mobility can occur, there may be an enabled transition which, if triggered, leads to an arbitrary attachment change. The point is that only the effect an event has on the system must be represented—very few details on how it came to occur are necessary.

## 3.2 Elements of Verification

### 3.2.1 Protocol Description

To verify a protocol, it must be first be described in a sufficiently complete and unambiguous way—otherwise, accurate formalization is not possible. Therefore, an important part of verification consists in surveying and augmenting the available descriptions of the protocol.

In the case of SMP, the protocol was essentially defined by an implementation in the Linux kernel, although an overview of the central concepts was available [2, ch. 3]. Hence, the aim was, as the modelling phase progressed, to produce a less implementation-oriented and more precise informal specification. Such a specification may then be used as the basis for other protocol implementations.

The resulting protocol description may be found in sections 4.2.1 and 4.3.1, in preparation of the presentation of several protocol models.

### 3.2.2 Protocol Models and Formal Specification

Since a model-based approach was chosen, a verification effort should provide a set of models capturing relevant parts of the protocol. Here, models were constructed not only for use by the model checker, but also for the purpose of providing an accessible, unambiguous description of the protocol’s services and procedure rules, in the spirit of literate programming [42, pp. 30ff], as hinted in section 3.1.1.

Modelling in a formalism such as PROMELA forces structure and explicitness. Thus, the discovery of minor omissions and errors should not

come unexpected in the modelling phase. The use of high-level constructs in PROMELA also suggested that it would be possible to achieve a reasonable degree of correspondence between selected parts of the source code of the implementation and the models.

An obvious way of evaluating the verification effort is to attempt to assess the quality of the PROMELA models. To aid in this process, details on development history and justification of modelling choices is provided along with PROMELA code fragments in sections 4.2.2, 4.3.2 and 4.3.3. The complete models are available electronically at <http://www.palmskog.net/exjobb/>.

### 3.2.3 Verification Report

One of the distinct disadvantages of model checking is that no explicit mathematical proof of the correctness with respect to specification is produced. Not only would such a proof provide a good way to judge the outcome of a verification effort, but it may also further understanding of the protocol. As it stands with model checking, only counterexamples can be presented in full detail; for unhindered verification runs, only input parameters and tool output can be given. Still, by attempting verification of variations of a model and comparing results, it may be possible to gain an insight into protocol details pertinent to correctness.

For each verification run of a final model, SPIN command-line parameters are given, along with a summary of output. To achieve a realistic scope of the effort, we did not expect that verification of all parts of SMP would be considered conclusive.

# Chapter 4

## Results

“The process of formalisation is essentially one of organisation.”  
—Robin Milner

We present the results of the verification process, in the form of informal protocol descriptions, details of PROMELA models and formal specifications.

### 4.1 Overview

To lessen model checking complexity, and to make clear the services provided, SMP was decomposed into separate parts; notably, the protocol elements related to checkpoints, which are not directly related to the state machine transitions which allow a session to be resumed, were handled as a separate protocol. The idea was to be able to assume the correctness of the checkpoint mechanism when investigating the SMP state machine. As it turned out, this simplified informal analysis and limited model sizes considerably. Verification of the state machine, in turn, was divided into verification of safety and liveness, with the different properties verified for different models. Safety was defined as data integrity preservation during session resumption, and liveness was essentially defined as the absence of deadlocks during control message exchange; more details are given in section 4.3.2 and 4.3.3.

The informal protocol descriptions which precede the presentations of the PROMELA models contain qualifications that are oriented towards verification. Hence, they should not be seen as a complete resource for carrying out a protocol implementation. Note also that the protocol presented here differs in some places from that used the first proof-of-concept implementation [2, ch. 4], since verification-related changes have been incorporated. Perhaps the most drastic change is that endpoints now must be able to agree on session data buffer sizes before establishing a data connection. This change is due to the discovery of an error related to checkpoint handling, as elaborated on in section 4.2.3.

## 4.2 The Checkpoint Protocol

### 4.2.1 Protocol Specification

We describe the service the protocol provides, the protocol's assumptions about the environment in which it is executed, the vocabulary of the messages, the encoding of messages and the protocol procedure rules which ensure communication consistency. This informal description paves the way for the formal protocol model and specification presented in section 4.2.2.

#### Service Provisions

Let  $A$  and  $B$  be two communicating endpoints and let  $S_A$  and  $S_B$  be sequences of words of data. Let  $S_A^i$  and  $S_B^j$  denote the  $i$ th and  $j$ th word of data in each sequence, respectively, where  $i$  and  $j$  are nonnegative integers. We refer to  $S_A$  as the *send stream* of  $A$ , and suppose that endpoint  $A$  wishes to transfer the data in  $S_A$  to  $B$  in the proper order; symmetrically for  $B$ .

The purpose of the protocol is to let  $A$  and  $B$  continually agree on a checkpoint tuple  $\langle i, j \rangle$ , such that endpoint  $A$  is guaranteed to have properly received the words  $S_B^0, S_B^1, \dots, S_B^{j-1}$  and, symmetrically, endpoint  $B$  is guaranteed to have properly received the words  $S_A^0, S_A^1, \dots, S_A^{i-1}$ . This provision allows a disconnected communication session to be resumed without re-sending all data, and lets an endpoint discard safely transferred data from its send stream.

#### Environmental Assumptions

The protocol is executed in the context of an established connection allowing simultaneous communication in both directions, in which lower layer protocols ensure that messages, if they reach their destination, are error-free, received in the order they were sent and received only once. If a message is lost, no further messages are sent over the connection. Of the two communicating endpoints, exactly one of them must be able to be identified as the *connection initiator*. Finally, the connection initiator must know the session data buffer size of its peer.

#### Message Vocabulary

There are two types of messages: *checkpoint* for messages with a checkpoint identifier and a send stream position, and *data* for messages containing words of data. Consequently, the vocabulary may be expressed as a set

$$V = \{\text{checkpoint}, \text{data}\}.$$

## Message Format

A protocol message consists of two control fields identifying the *message type* and *payload size*, respectively, and a data field containing the *payload*. If the message is of type *checkpoint*, the payload consists of a *checkpoint identifier* field and a *send stream position* field; otherwise it contains application-specific data.

## Procedure Rules

An endpoint has an acknowledged and a pending checkpoint, identified by numbers in the set  $\{0, 1, 2\}$  contained in the variables `ackedId` and `pendId`, where

$$\text{pendId} \equiv \text{ackedId} + 1 \pmod{3}.$$

The current acknowledged checkpoint is defined by the integers contained in the variables `ackedSent` and `ackedRecd`, which are absolute references to sequence positions, and the current pending checkpoint is defined by the integers contained in the variables `pendSent` and `pendRecd`, which refer to positions in the sequence relative to the acknowledged checkpoint. The idea is that, across both endpoints, checkpoints having the same identifier should refer to the same places in the sequences. To ensure this is the case immediately after the connection has been established, all checkpoints are set to refer to the beginning of the sequences. After a proper initialization, an endpoint continually attempts to send data and may receive data or checkpoint messages at any time. Both endpoints keep the variables `sent` and `recd`, which contain the number of words sent and received relative to the latest acknowledged checkpoint, updated as the connection progresses.

If the endpoint is the connection initiator and the number of words sent or received reaches a certain fraction of the buffer size, it lets its pending checkpoint become the acknowledged checkpoint and creates a new pending checkpoint from the current values of `sent` and `recd`, relative to the position of the new acknowledged checkpoint. Then, a *checkpoint* message to the other endpoint containing the updated values of `pendId` and `pendSent` is sent. After this is done, the initiator sets a boolean variable `recvCpAck` to true, indicating that it is expecting a checkpoint reply; while `recvCpAck` is true, no more checkpoints may be created. When a checkpoint reply is received, the value of `pendRecd` is updated to the value given in the message, and `recvCpAck` is set to false.

When the noninitiator receives a checkpoint message, it lets its pending checkpoint become the acknowledged checkpoint and creates a new pending checkpoint from the current value of `sent` relative to the position of the new acknowledged checkpoint, and the position given in the message. Then, a reply message containing the updated values of `pendId` and `pendSent` is sent. If the noninitiator has sent or received as many words of data as

the buffer size, it refrains from sending more data and awaits a checkpoint message. Note that after a new checkpoint is created, data words in the send stream up to, but not including, `ackedSent`, may be purged to leave room for buffering more sent data.

## 4.2.2 Protocol Model and Formal Specification

Using the implementation and informal protocol description as a guide, the checkpoint protocol was modelled in PROMELA. Models were gradually refined through the use of simulations. Correctness was formulated in LTL, and the models exhaustively checked and corrected iteratively.

### Model Details

In the PROMELA model, we first define the message types and a message data structure:

```

mtype = {data,cp};

typedef data_msg {
    mtype type;
    byte cpId;
    byte cpPos;
}

```

Note that a message of type *data* is assumed to contain precisely one word of data. Next, we represent the two endpoints by two PROMELA processes, `Initiator` and `Noninitiator`, both having buffered channels `send` and `recv`, such that the `send` channel of the one is the `recv` channel of the other. To control the buffer size and the buffer size fraction which may be filled up before a checkpoint is created, we introduce two constants `bufTrigger` and `bufFactor`; the buffer size is defined as the product of the constants, and the buffer fraction is defined as the quotient of `bufTrigger` and the product. Finally, the size of the buffer of communication channels is controlled through the constant `queueSize`. We define reasonable constant values for verification as follows:

```

#define bufTrigger 3
#define bufFactor 2
#define queueSize 2

```

Before the processes start to communicate, necessary variables are initialized according to the procedure rules. An initial obstacle concerns which PROMELA type should be used for the variables. The checkpoint identifiers and the relative stream positions offer no problem, but the absolute stream positions `ackedSent` and `ackedRecd` are another story. Clearly, if the underlying transition system is to be finite, variables cannot grow arbitrarily.

In implementation, the variables are pointers rather than sequence element indices, so there is no problem there—but for verification, we need to have some kind of absolute measure of the progress in the sequence to which we can compare our relative references. The solution is to always compute our absolute stream indices modulo a constant larger than the buffer size, allowing all variables to be defined to be of type `byte`:

```
#define wrap 13
```

For each endpoint, we put the label `ACTIVE` immediately before the point where communication can take place. After the `Initiator` process reaches `ACTIVE`, the actions available to it depends on the current values of `send`, `recd` and `recvCpAck`. If either `send` or `recd` is equal to the buffer size and `recvCpAck` is true, it executes a blocking receive in order to get a checkpoint reply. If `recvCpAck` is false, a new checkpoint is created and a checkpoint message is sent. Supposing `send` and `recd` are both less than the buffer size, the `Initiator` process may both receive messages and send data; the latter action may trigger checkpoint creation and the sending of a checkpoint message.

The PROMELA code for checkpoint creation in the `Initiator` process is perhaps the most interesting part of the model. Of particular importance is the use of the constant `wrap` for modulo arithmetic on the variables `ackedSent` and `ackedRecd` and the use of the current value of `send` in the checkpoint message. For atomicity, the `d_step` construct [18, p. 56] is used:

```
d_step {

    /* move the reference points */
    send = send - pendSent[0];
    recd = recd - pendRecd[0];

    /* overwrite old acked and update absolute counters */
    ackedId[0] = pendId[0];
    ackedSent[0] = ((ackedSent[0] + pendSent[0]) % wrap);
    ackedRecd[0] = ((ackedRecd[0] + pendRecd[0]) % wrap);

    /* new pend */
    pendId[0] = (pendId[0] + 1) % 3;
    pendSent[0] = send;
    pendRecd[0] = recd;

    /* prepare message */
    sendCpMsg.cpId = pendId[0];
    sendCpMsg.cpPos = pendSent[0];

};
```

```

/* send checkpoint request */
send!sendCpMsg;
recvCpAck = true;
goto ACTIVE;

```

Note that the checkpoint-related variables of both processes are elements in arrays of global scope—this is because SPIN’s partial-order reduction strategy does not work properly during verification if there are remote variable references in correctness claims. However, no interprocess communication is facilitated through these variables.

Before the **Initiator** process receives a reply to a checkpoint message, it may receive additional data messages. Hence, the **pendRecd** variable of the process is updated to the send stream position of a checkpoint reply message:

```

d_step {

    pendRecd[0] = recvMsg.cpPos;
    recvCpAck = false;

};
goto ACTIVE;

```

The **Noninitiator** process, in contrast, cannot create any checkpoints of its own accord, but only update checkpoints on receiving a checkpoint message. If **send** or **recd** is equal to the buffer size, it will only attempt to receive messages—otherwise, it may send data as well. We have seen that the **Initiator** process appends its current value of **pendSent** to a message; one important question concerns what use the **Noninitiator** process can make of this value. Using the environmental assumption that if a message was received, all formerly received messages were error-free and in the proper order (section 4.2.1), we conclude that the variable **pendRecd** may be assigned this value and that **pendSent** may be assigned the value of **send** relative to the new acknowledged checkpoint. The new value of **pendSent** is then used in the checkpoint reply message:

```

d_step {

    sent = sent - pendSent[1];
    recd = recd - pendRecd[1];

    /* overwrite old acked and update absolute counters */
    ackedId[1] = pendId[1];
    ackedSent[1] = ((ackedSent[1] + pendSent[1]) % wrap);
    ackedRecd[1] = ((ackedRecd[1] + pendRecd[1]) % wrap);

```

```

    /* new pend */
    pendId[1] = recvMsg.cpId;
    pendSent[1] = sent;
    pendRecd[1] = recvMsg.cpPos;

    /* prepare message */
    sendCpMsg.cpId = pendId[1];
    sendCpMsg.cpPos = pendSent[1];

};
/* send checkpoint ack */
send!sendCpMsg;
goto ACTIVE;

```

### Example Model Execution

To show how the procedure rules and the model works, we offer a message sequence chart, shown in figure 4.1. For a more compact illustration, `bufTrigger` and `bufFactor` were set to 2, and `queueSize` was 1. The execution exemplifies how the protocol depends on a reliable message channel; when the `Initiator` process properly receives a *checkpoint* message in response, it cannot have failed to receive the *data* message sent earlier.

### Formal Correctness Claims

For safety [49, pp. 50f], we claim that the two endpoints always have a checkpoint identifier in common, and that the checkpoint denoted by this identifier is the same on both endpoints. In LTL, this may be expressed as

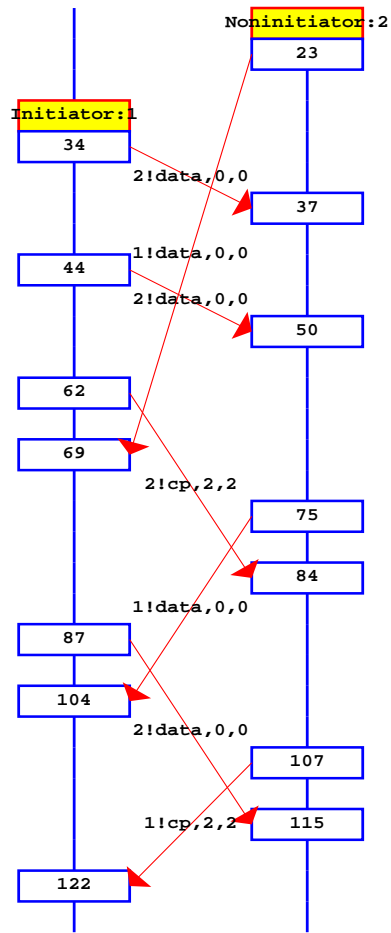
$$\begin{aligned}
& \square((ak \rightarrow (akSn \wedge akRc)) \wedge (akPn \rightarrow (akPnSn \wedge akPnRc)) \wedge \\
& \quad (pnAk \rightarrow (pnAkSn \wedge pnAkRc)) \wedge ((ak \wedge \neg akPn \wedge \neg pnAk) \vee \\
& \quad \quad (\neg ak \wedge akPn \wedge \neg pnAk) \vee (\neg ak \wedge \neg akPn \wedge pnAk)))
\end{aligned}$$

where we let

```

#define ak (ackedId[0] == ackedId[1])
#define akSn (ackedSent[0] == ackedRecd[1])
#define akRc (ackedRecd[0] == ackedSent[1])
#define akPn (ackedId[0] == pendId[1])
#define akPnSn (ackedSent[0] == ((ackedRecd[1] + pendRecd[1]) % wrap))
#define akPnRc (ackedRecd[0] == ((ackedSent[1] + pendSent[1]) % wrap))
#define pnAk (pendId[0] == ackedId[1])
#define pnAkSn (((ackedSent[0] + pendSent[0]) % wrap) == ackedRecd[1])
#define pnAkRc (((ackedRecd[0] + pendRecd[0]) % wrap) == ackedSent[1])

```



**Figure 4.1:** Fragment of a checkpoint protocol execution

For liveness [49, pp. 52f], we claim that both endpoints must always eventually reach a state in which they can communicate. In LTL, this becomes

$$(\Box \Diamond inAct) \wedge (\Box \Diamond ninAct)$$

where we let

```
#define inAct Initiator@ACTIVE
#define ninAct Noninitiator@ACTIVE
```

## Verification Results

The model was exhaustively verified against the LTL claims in SPIN version 4.2.6 on a 3 GHz Pentium 4 computer with 2 GB of memory running Linux, using the commands

```
$ spin -a -X -N claim.ltl checkpoint-final-global.promela
$ gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=1900 -DCOLLAPSE \
-DXUSAFE -DNOFAIR -DMA=18 pan.c
$ ./pan -v -X -m30000000 -w19 -a -c1
```

where we substituted `checkpoint-final-global.promela.liveness.ltl` and `checkpoint-final-global.promela.safety.ltl` for `claim.ltl`, in turn. No counterexamples were found for either of the claims. The liveness claim was verified using the value 2 for the `wrap` constant, to lessen model checking complexity.

### 4.2.3 Development and Verification History

We survey the problem situation which prompted protocol development, and describe the protocol prior to verification-related changes.

#### Protocol Origins and Initial Version

To solve the problem of properly resuming stateful sessions in an implementation of a disconnection-tolerant session-layer, Arvidsson and Widell created a protocol which resembles a simplified version of TCP, handling large chunks of data rather than individual bytes [2, p. 98]. Sent data is buffered in a segment, awaiting acknowledgment from the other endpoint before it is purged; a session is resumed from the last segment that has not been acknowledged. It turned out that optimal segment size and the optimal number of available segments depended greatly on the network and context. Also, the segment protocol did not conform to OSI session layer specification of a similar mechanism. In view of these problems, a new data consistency protocol, using the notion of checkpoints in the data streams, was developed.

The initial version of the checkpoint protocol used identical procedure rules for the endpoints: the connection initiator and the other endpoint both attempt to create checkpoints when the number of words of data sent reaches a certain limit [2, pp. 42f]. The message format differed from the current protocol in that both the current value of `pendSent` and `pendRecd` are appended to checkpoint messages. One interesting property of this protocol is that a *checkpoint* message meant by the sender as a request for a reply may be interpreted by the receiver as a reply. This happens when the receiver has previously sent an as-yet unanswered *checkpoint* message of its own. To reconcile differences in checkpoints established concurrently

at each endpoint, the send stream position field of any *checkpoint* message interpreted as a reply is compared with the current value of `pendRecd`, after which `pendRecd` is assured to contain the maximum of the two values.

### Modelling and Verification Attempts

The initial version of the checkpoint protocol was modelled in much the same way as described in section 4.2.2. Early in the modelling phase, the underlying assumptions that the checkpoint messages are sent in the same channel as the data messages, and that the channels respect the order of messages sent through them, were made explicit. More specifically, certain executions of a model in which the *data* messages and *checkpoint* messages were sent through different channels violated the LTL safety claim given above. In retrospect, it is not surprising that when *checkpoint* messages asserting that a certain number of words of data have been sent arrive before the *data* messages it pertains to, checkpoint definitions may become ambiguous across endpoints. However, since there was no formal protocol description available at the time, the discovery was of some importance.

During the course of model checking, a rather obscure error in the protocol was found [2, p. 49] and formally characterized as a violation of the safety claim. Consider the execution of the protocol model shown in figure 4.2 below, where the first endpoint interprets a checkpoint request as a reply to an earlier *checkpoint* message it sent, but which has not yet been received by the second endpoint. Subsequent to receiving the reply, the first endpoint creates another checkpoint, after which the checkpoint with identifier 2 is ambiguous: for the first endpoint, it is defined by  $\langle 3, 3 \rangle$ , but for the second endpoint, is defined by  $\langle 0, 3 \rangle$ . Should the connection be lost before the *checkpoint* message from the first endpoint arrives, the session cannot be properly resumed. In the execution in the figure, `bufTrigger` was 3 and the message queue size was 2.

There were also problems with the liveness claim in the model. Both endpoints may try to send checkpoint messages at the same time when both message queues are full, resulting in deadlocked executions, as is illustrated in figure 4.3. In the example execution, `bufTrigger` was 3 and the message queue size was 1. The deadlock problems may be amended by always letting a process choose nondeterministically between sending and receiving a message. Checkpoint messages may in this way be scheduled to be sent when the `send` channel is no longer full. In implementation, such a control structure can be interpreted to mean that a message which cannot be sent is kept in memory, and that re-send attempts are made periodically.

These shortcomings of the initial checkpoint protocol led to the development of the current protocol, as is outlined briefly below. Work on a model of the initial protocol version and on the correctness claims suggested the present formulation of the checkpoint protocol service provisions in section

4.2.1, which is less ambiguous and less implementation-oriented than the one originally given by Arvidsson and Widell [2, p. 42].

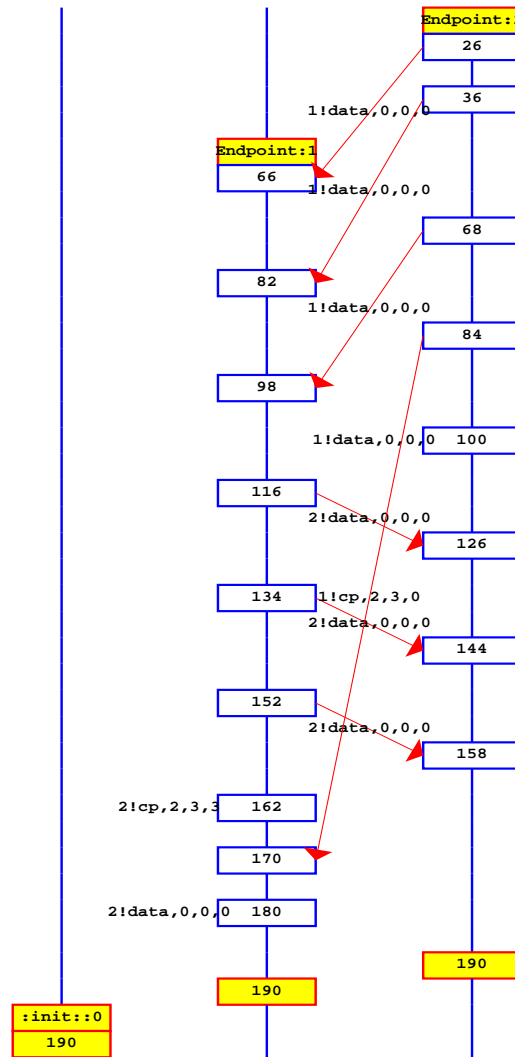


Figure 4.2: Counterexample in a model of the initial protocol

### Current Protocol Version

For protocol simplification and correction, Arvidsson suggested that the connection initiator should be the only endpoint allowed to create checkpoints. This idea was formalized in the model described in section 4.2.2; one notable difference from the initial checkpoint protocol is that a *checkpoint* message

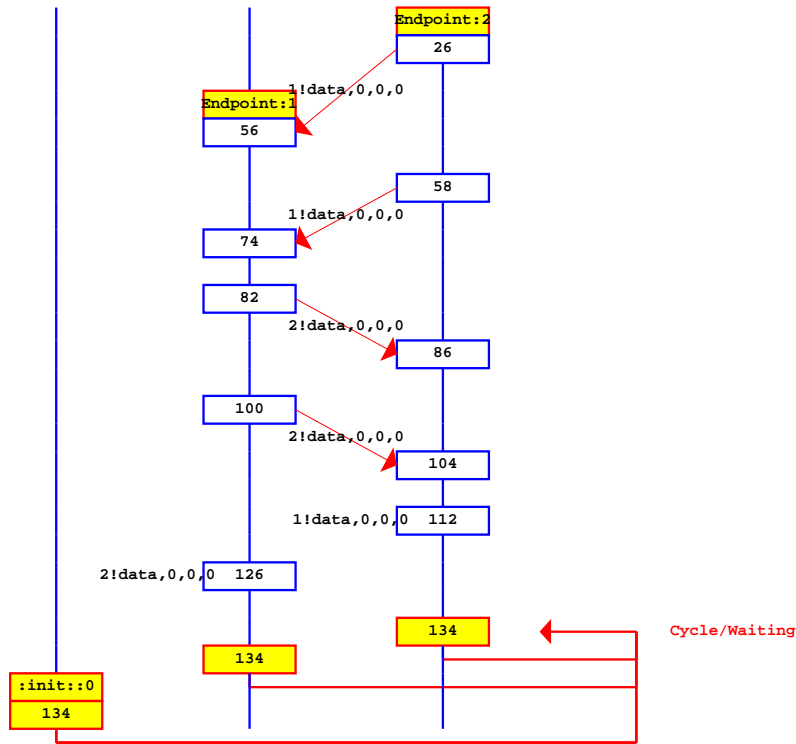


Figure 4.3: Deadlock in a model of the initial protocol

only contains one stream position field as opposed to two. Hence, the current protocol has lower resource consumption (specifically, lower bit complexity [49, p. 70]) than the original version, in addition to the obvious advantage of correctness in all executions.

#### 4.2.4 Concluding Remarks

While correctness is arguably the most important property of a protocol, resource consumption and efficiency are other factors significant in implementation. The initial checkpoint protocol described in section 4.2.3, allowing simultaneous checkpoint creation, intuitively has a greater degree of concurrency than the current protocol and may thus perform better in some situations. The effect of different values of the parameters `bufTrigger` and `bufFactor` on performance also remain unexplored in a systematic way. To sum up, a probabilistic analysis of protocol throughput may be needed.

## 4.3 The Session Management Protocol

### 4.3.1 Protocol Specification

Similarly to the presentation of the checkpoint protocol, we describe the protocol elements of SMP, in preparation of a formal treatment.

#### Service Provisions

The protocol aims to provide reliable and effective transmission of data between two network endpoints which may experience intermittent connectivity in both space and time. Suppose once again (section 4.2.1) that each process has a sequence of input data that it wishes to transfer without error to the other process. If transmission of sequence data is somehow aborted, the protocol must see to that, when opportunity arises, the session resumes from an agreed-on state such that retransmission is minimal.

#### Environmental Assumptions

Endpoints must be able to continually establish reliable transport-layer connections (using e.g. TCP/IP) over the network. Messages must be received error-free, in the order they were sent and only once, or not at all. The processes are assumed not to terminate, but may, at any time, become unable to communicate over the network or change network attachment. However, disconnection always admits the possibility of reconnection. Of the two processes, exactly one of them must be able to be identified as the *connection initiator*. Although the underlying network is considered as asynchronous, we assume fully synchronous communication can be simulated.

There is a name server process on the network, which never changes network attachment and cannot be disconnected. Hence, a communicating process may, if it is connected, register with the name server and receive the last known endpoint address of the other process.

In addition to the name server, we assume each process has a mechanism for detecting network disconnection and changes in network attachment. Finally, processes are assumed to have session data buffers separate from transport-layer buffers for storing sent data and, as before, the connection initiator must know the session data buffer size of its peer.

#### Message Vocabulary

The data messages types may be divided into those which are used only during data channel establishment (*connect*, *connect\_ok* and *connect\_denied*) and those which are used in an active channel (*data*, *checkpoint* and *close*). The former group of types are not relevant for verification; additionally, since we assume that communication goes on indefinitely, the *close* message need

not be used. Control message types used by the protocol include *suspend*, *resume*, *resume\_ok* and *resume\_denied*. Hence, the messages pertinent to verification may be expressed as a set

$$V = \{checkpoint, data, suspend, resume, resume\_ok, resume\_denied\}.$$

### Message Format

An idealized data channel message consists of a control field identifying the message type, and a data field containing the payload. If the message is of type *checkpoint*, the payload consists of a *checkpoint identifier* field and a *send stream position* field; otherwise it contains application-specific data.

The control channel message format is, without details irrelevant in this context, essentially the same as the data channel format: it consists of a message type field and a payload field. For *suspend* messages, the payload field is unused; for *resume* and *resume\_ok* messages, the payload consists of an *endpoint address* field and a *checkpoint identifier* field. In contrast, *resume\_denied* messages only use the *endpoint address* field.

### Procedure Rules

For data channel transfers, which happen asynchronously, the checkpoint protocol procedure rules given in section 4.2 are used. For handling the suspend-resume functionality, each endpoint uses a state machine, depicted in figure 4.4, which determines which control messages are to be sent and which are to be received in certain contexts. We denote the left and right junction by `USER_RESUME` and `SEND_RESUME_OK`, respectively. Only when both endpoints are in the state `ACTIVE` is data transfer possible. All exchanges of control messages take place synchronously.

Should a process suspend, it first attempts to send a *suspend* message and then enters the `SUSPENDED` state. A session in which at least one process is suspended cannot become active until all processes have elected to leave the `SUSPENDED` state. However, a suspended process is still able to receive control messages and promptly sends a *resume\_denied* message in response to any *resume* message it gets.

Should a change in network attachment take place, the process registers with the name server and attempts to send a *resume* message to the other process. Resume messages include the identifier of the present acknowledged checkpoint. If sending the message fails, the process enters a state `READY_RESUME`, in which it is ready to resume at the request of the other process; if the message is sent successfully, the process enters a state `SENT_RESUME`, where it expects to eventually receive a reply. Should a process in `SENT_RESUME` receive a proper *resume\_ok* message, confirming the checkpoint from which to send and receive, the session is resumed—with both processes performing a rollback to the common checkpoint.

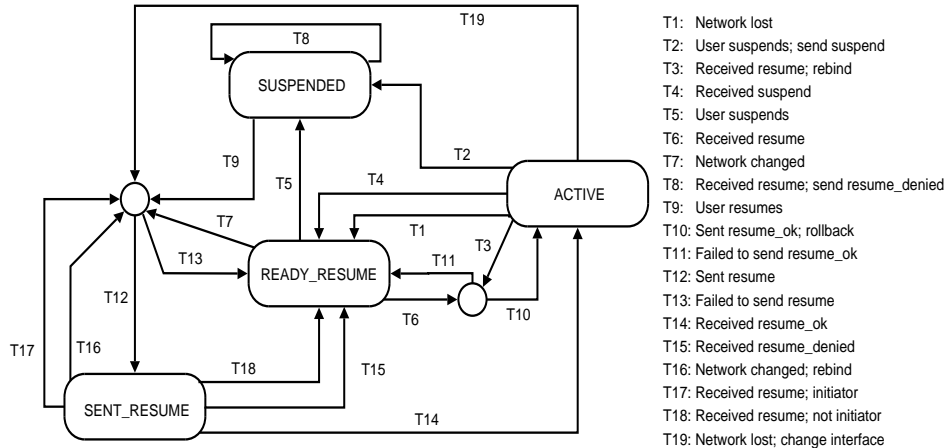


Figure 4.4: SMP state machine

A process always appends the identifier of its acknowledged checkpoint to a *resume* message, but it may happen that the other process does not have a checkpoint with that identifier. Hence, the checkpoint identifier appended to a *resume\_ok* message is important when the other process performs a rollback before resuming data transfer.

To avoid deadlocked executions where one process is in **SENT\_RESUME** expecting a reply to a *resume* message, and the other process is in the state **READY\_RESUME**, a flag is set after any failed attempt to respond to a *resume* message. Later, when that process tries to send a *resume* message, it first sends a *resume\_denied* message and, if successful, then clears the flag. Exactly how these deadlocks manifest, and whether they are removed by this measure, is investigated in section 4.3.3.

### 4.3.2 Safety of the Session Resumption Functionality

Informally, we may consider the safety of the protocol as the property that data transfer consistency is always preserved. For this to be the case, the checkpoint protocol is central—but there must also be correct procedure rules for state negotiations when data transfers are resumed. We build upon the results in section 4.2.2 and assume endpoints always have a checkpoint in common. Again, because of the changes made to the checkpoint protocol, the procedure rules for rollback to an established checkpoint differs from the proof-of-concept implementation.

We argue that the **SUSPENDED** state is not relevant for the safety of the protocol, since no resumption-related activity can take place there. Therefore, the state was not included in any safety model.

## Model Details

To enable an initial formalization of the procedure rules for session resumption in PROMELA, the protocol environment was restricted; it was assumed that endpoints could not be disconnected from the network—only move instantaneously from one attachment point to another. The idea was that an endpoint attempting to send a control message would be able to decide whether the attempt was doomed to fail (because the other endpoint had changed attachment) or not.

**Parameters, Messages and Channels** Message types and formats were defined using the same approach as for the checkpoint protocol. Since there is no *SUSPENDED* state, and no possibility of deadlock after failure of sending a *resume* message, there are no message types *suspend* or *resume\_denied*:

```
mtype = {cp,data,resume,resume_ok};
```

```
typedef dataMsg {  
    mtype type;  
    byte cpId;  
    byte cpPos;  
}
```

```
typedef ctrlMsg {  
    mtype type;  
    chan addr;  
    byte cpId;  
}
```

Obviously, there must be three processes in the model in addition to the mandatory `init` process: the two endpoint processes (hereafter referred to as `Initiator` and `Noninitiator`, respectively) and a name server process (`NameServer`). An initial problem concerns the distribution of channels for message passing. In addition to unbuffered (rendezvous) channels for control messages and buffered channels for data messages, processes need channels for changing attachment (`move` channels) and exchanging messages with the name server (`name` or `ns` channels). Finally, there is a shared synchronous channel used in the final steps of session resumption (`openData`).

Since the checkpoint protocol is integrated into the model, the same parameterization occurs, albeit with somewhat lower given values to restrict model state space size:

```
#define bufTrigger 2  
#define bufFactor 2  
#define wrap 9  
#define queueSize 1
```

Let `recvCtrl` be the channel a communicating process uses to receive control messages. Suppose each process is associated with an array, where each element is a control message channel. Define a change of attachment as the replacement of `recvCtrl` with a value from the array. Let there be an array index variable `pointIdx` that is initially 0. By continually replacing `recvCtrl` with the channel in the array at `pointIdx` and subsequently incrementing the index modulo the array size, random changes in attachment may be simulated. The underlying idea is to let control message channels act as both network addresses and sockets. For example, in a control message, the `addr` value is the `recvCtrl` channel of the sending process.

To enable processes to detect whether an attempt to send control or data messages will fail, the current `recvCtrl` channel of each process is held in an array of global scope. Hereafter, `recvCtrl[0]` and `recvCtrl[1]` will refer to the current channel of the `Initiator` and `Noninitiator` process, respectively.

**Processes** The `init` process starts the other processes, and then triggers changes in network attachment of processes by sending them new control channels over the `move` channels:

```

/* start processes */
run Nameserver(point1Name, point2Name, point1CtrlRecv[0],
               point2CtrlRecv[0]);
run Initiator(point1Move, point1Name, point2CtrlRecv[0],
               point1DataRecv, point2DataRecv, openData);
run Noninitiator(point2Move, point2Name, point1CtrlRecv[0],
                  point2DataRecv, point1DataRecv, openData);

/* trigger movement */
do
  :: point1Move!point1CtrlRecv[point1Idx];
     point1Idx = (point1Idx + 1) % 2;
     point2Move!point2CtrlRecv[point2Idx];
     point2Idx = (point2Idx + 1) % 2;

od;

```

For reasons of avoiding deadlocks unrelated to the actual protocol design, the order in which the two processes change attachment is deterministic. Then, if a process attempts to send a control message over a channel that the other process does not use, there is no possibility of the sending process losing execution privileges and the receiving process changing attachment back to the unused channel.

Consider next the name server, which we model as a process with private synchronous channels to each endpoint. The server stores the last reported

`recvCtrl` channel for each process. When a change in network attachment takes place, an endpoint process sends its new address to the name server and receives the stored address of the other endpoint process, which it from then on uses for sending all control messages. Without the name server, processes would be unable to communicate after changing attachments simultaneously. PROMELA code for the server is as follows:

```

do
  :: atomic { ns1?point1Recv;
             ns1!point2Recv; };

  :: atomic { ns2?point2Recv;
             ns2!point1Recv; };

od;

```

We may now define precisely how a process goes about for changing network attachment. Consider the following code fragment in the `Initiator` process, which is used as a sequence in an `if` construct where attachment changes are supposed to be possible:

```

atomic { move?recvCtrl[0];
          /* moved to new network, register with name server */
          ns!recvCtrl[0];
          ns?sendCtrl; };
goto USER_RESUME;

```

The variables in the endpoint processes essentially form a superset of the set of variables used in the PROMELA model of the checkpoint protocol. The only notable additions were auxiliary variables of the type `ctrlMsg` for use in sending and receiving control messages.

**Protocol States and Sending Messages** The SMP states were captured through the use of labels in the code, with transitions as `goto` statements. The central problem was to decide where transitions should be placed. Consider the `ACTIVE` state; a change in network attachment may happen when able to receive control messages and data messages, but not when attempting to send them, because of atomicity. Similarly, no mobility can take place when a process is in the junction `USER_RESUME`.

When sending a message, the case that the peer process may have changed network attachment must be accounted for, since some executions may otherwise block forever. The only reliable way of detecting that the other process has moved is to do a comparison of the channel used for sending control messages (`sendCtrl`) and the channel the peer uses for receiving control messages (`recvCtrl`). A fragment of the code for the junction `USER_RESUME` which illustrates this:

```

if
  :: sendCtrl!sendCtrlMsg;
  goto SENT_RESUME;

  :: sendCtrl != recvCtrl[1];
  /* failed to send */
  goto READY_RESUME;

fi;

```

When a process ends up in the latter branch of the `if` statement, this may be interpreted to mean that the endpoint has detected that sending the message is impossible.

**Session Resumption** The most interesting aspect of the model is the interaction of processes during session resumption. In a typical case, both processes are initially in `ACTIVE` and exchange *data* and *checkpoint* messages. Suppose the `Initiator` process changes attachment when its `ackedId` contains the same value as the `pendId` of the `Noninitiator` process. After successfully sending a *resume* message containing the currently used control channel and the current `ackedId` value, the `Initiator` process enters the state `SENT_RESUME`. Meanwhile, the `Noninitiator` process receives a *resume* message in the state `ACTIVE`. In response, the process enters the junction `SEND_RESUME_OK` where it prepares and attempts to send a *resume\_ok* message. In this case, the checkpoint identifier is equivalent to `pendId`, so the message variables are set atomically as follows, using `d_step`:

```

  sendCtrlMsg.type = resume_ok;
  sendCtrlMsg.addr = recvCtrl[1];
  sendCtrlMsg.cpId = pendId;

```

Supposing the *resume\_ok* message is sent successfully, the `Noninitiator` process performs an atomic rollback where it replaces the old acknowledged checkpoint to match the other process:

```

/* reset */
sent = 0;
recd = 0;

/* overwrite old acked; update absolute counter */
ackedId[1] = pendId;
ackedSent[1] = ((ackedSent[1] + pendSent) % wrap);
ackedRecd[1] = ((ackedRecd[1] + pendRecd) % wrap);

/* new pend */
pendId = (pendId + 1) % 3;

```

```
pendSent = sent;
pendRecd = recd;
```

If the acknowledged checkpoint identifiers of the processes been equivalent, the `Noninitiator` process would only have reset the `sent`, `recd`, `pendSent` and `pendRecd` variables, which is what the `Initiator` process does:

```
/* reset to acked */
sent = 0;
recd = 0;
pendSent = sent;
pendRecd = recd;
recvCpAck = false;
```

Next, the processes attempt to synchronize using the `openData` channels. This fails if the `Noninitiator` process changes attachment; otherwise, the data channel buffers are emptied and data communication is resumed.

Consider the case where both processes are again initially in `ACTIVE`, but where the `Noninitiator` process changes attachment. The `Noninitiator` then sends a `resume` message containing its `ackedId` value, after which it makes a transition to `SENT_RESUME`. The `Initiator` process, after receiving `resume` and updating its `sendCtrl` value using the `addr` field in the message, proceeds by attempting to send a `resume_ok` message using its current `ackedId` value—which in this case differs from the `cpId` in the original message. Suppose the `Noninitiator` receives the `resume_ok` message. It then detects that the identifier in the message differs from its `ackedId` identifier, and replaces the acknowledged checkpoint with its pending checkpoint, as described above.

The two remaining cases, where the `ackedId` value of both processes match, does not involve overwriting any checkpoints—only resetting the data word counters.

## An Alternative Model

To restrict the environmental assumptions concerning possibility of disconnections, as was done for the model presented in the previous section, is not without consequences—there may be protocol errors in the executions not captured in the chosen scenario. Another approach, in which the central idea was that sending data and control messages fail nondeterministically, was attempted. One of the disadvantages of the use of nondeterminism in this way is that it results in a coarse-grained model: no distinction is made between transmission failure related to disconnection, mobility, or the network.

In `PROMELA`, nondeterministic failure may be implemented through the use of a sequence in an `if` construct containing only `true`; this creates a

transition that is always enabled. Since there is then no longer any possibility of deadlock stemming from the order of attachment changes, we may now allow nondeterministic changes:

```

do
  :: point1Move!point1CtrlRecv[point1Idx];
     point1Idx = (point1Idx + 1) % 2;

  :: point2Move!point2CtrlRecv[point2Idx];
     point2Idx = (point2Idx + 1) % 2;

od;

```

The code fragment in the junction `USER_RESUME` from was consequently changed to

```

if
  :: sendCtrl!sendCtrlMsg;
     goto SENT_RESUME;

  :: true;
     /* failed to send */
     goto READY_RESUME;

fi;

```

with similar updates made to other parts of the primary model.

### Example Model Execution

To illustrate how the procedure rules and the models work, we offer a message sequence chart of a fragment of an execution, shown in figure 4.5 below. Note that the messages coming from the `init` process result in changes in network attachment.

### Specifying Correctness

The safety of the session resumption functionality depends on the checkpoint protocol—the processes agree to resume from a checkpoint with a specific identifier, and assume that there exists such a checkpoint and that it is unambiguously defined. Precisely when both processes enter the `ACTIVE` state after having successfully negotiated on resuming the session, we include a PROMELA assertion stating the following, namely, that the agreed-on acknowledged checkpoint is unambiguous:

```

assert(ackedId[0] == ackedId[1] &&
        ackedSent[0] == ackedRecd[1] &&
        ackedRecd[0] == ackedSent[1]);

```

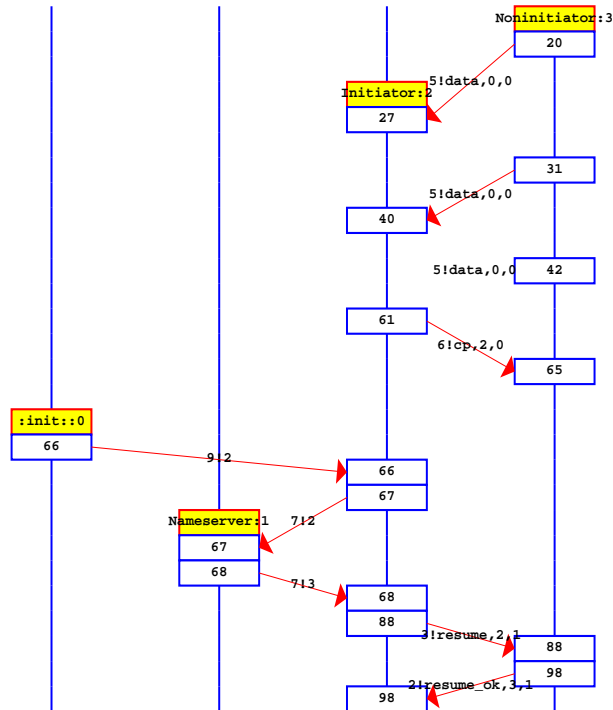


Figure 4.5: SMP safety model execution

## Verification Results

Due to the large state space size, the primary safety model could only be exhaustively verified in SPIN with the value 2 for the `wrap` constant, potentially excluding some counterexample executions. However, several partial state space searches using SPIN’s bitstate hashing algorithm [16] were carried out for larger parameter values. Specifically, the model was exhaustively verified in SPIN version 4.2.6 on a 3 GHz Pentium 4 computer with 3 GB of memory running Linux, using the following commands:

```
$ spin -a -X smp-safety-synchronous.promela
$ gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=3100 \
-DSAFETY -DCOLLAPSE -DNOCLAIM -DXUSAFE -DNOFAIR pan.c
$ ./pan -v -X -m1500000 -w19 -c1
```

No assertion violations were found.

The alternative model (`smp-safety-synchronous-fail.promela`), having an even larger state space, could not be exhaustively verified at all because of memory consumption issues. Partial searches for a range of parameter values did not detect any counterexamples.

### 4.3.3 Liveness Model of Suspend-Resume Functionality

#### Model Details

Development of a separate liveness model was motivated by the fact that particular conditions need to be present for deadlocks of the form described by Arvidsson and Widell [2, p. 61] to materialize. Essentially, one of the processes must fail to send a message over a control channel without the possibility of the other process detecting that failure. This is obviously not possible in the primary safety model, where failure to send a message always depends on mobility. For protocol liveness to be reflected in a model, spontaneous disconnection from the network needed to be available.

There seemed to be no relation between the checkpoint protocol and the deadlock issues, so the liveness model did not include checkpoint-related fields and sequences. To locate any deadlocked executions, the number of changes of network attachment and transitions to the state `SUSPENDED` for endpoints were limited by the constants

```
#define netChanges 2
#define userSuspend 2
```

To enforce the `netChanges` value, the `init` process loop for sending new control channels to processes was changed into the following:

```
int changesLeft = netChanges;
do
  :: changesLeft > 0;
    point1Move!point1CtrlRecv[point1Idx];
    point1Idx = (point1Idx + 1) % 2;
    point2Move!point2CtrlRecv[point2Idx];
    point2Idx = (point2Idx + 1) % 2;
    changesLeft--;
  :: else;
    break;
od;
```

To enforce the `userSuspend` value for each endpoint process, restrictions were placed before the `goto` statement in the main `if` construct of the `ACTIVE` state:

```
:: suspendLeft > 0;
  suspendLeft--;
  /* user suspend */
  goto SUSPEND;
```

where `suspendLeft` is set to `userSuspend` initially. Disconnection was modelled as the replacement of the `sendCtrl` and `recvCtrl` channels in a process with the channel `disconnect`, which no other process has access to.

The variables `storedRecv` and `storedSend` were used for keeping track of the replaced channels:

```
d_step {  
    storedRecv = recvCtrl[0];  
    storedSend = sendCtrl;  
    recvCtrl[0] = disconnected;  
    sendCtrl = disconnected;  
};
```

The use of disconnection called for changes in PROMELA code for many SMP states and junctions compared to the safety models. For example, the `USER_RESUME` junction had to be adjusted to allow for scenarios where a process, or its peer, is already disconnected or becomes disconnected.

Finally, the deadlocks described by Arvidsson and Widell were formally characterized. In the execution in figure 4.6, the rightmost noninitiator process sends a *resume* message, but the initiator process becomes disconnected and fails to send a *resume\_ok* message. After reconnecting, the initiator sends a *resume* message of its own, after which it stays perpetually in the state `SENT_RESUME`, while the noninitiator stays in `READY_RESUME`. This means that, although both processes share the same channels, no attempts at control message transmission take place. The recommended countermeasure was then inserted into the model, i.e. a boolean variable `gotResume` was declared for each process and set to `true` at every occurrence of a process failing to send a *resume\_ok* or *resume\_denied* message in response to a *resume* message. The idea is that, when `gotResume` is `true`, every attempt to send a *resume* message must be preceded by an attempt to send a *resume\_denied* message. If the other process properly receives the *resume\_denied* message, there is no risk of it ending up in `READY_RESUME` after receiving a *resume* message. Hence, the `gotResume` variable may be set to `false` after the successful transmission of a *resume\_denied* message. A typical execution of the final liveness model may be seen in figure 4.7.

## Specifying Correctness

No explicit specification of deadlock-freedom in the model was needed since SPIN is able to check automatically, given the proper parameters, whether a model has any improper process terminations (invalid endstates).

Despite several attempts, no satisfactory property that eventually becomes true could be found for the SMP state machine. Essentially, the protocol makes no guarantees about a suspended session eventually being resumed, or of a process eventually leaving a certain state, such as `SENT_RESUME`. For example, it may be the case that a process continually fails to send *resume\_denied* in response to a *resume* message, effectively creating a livelocked execution.

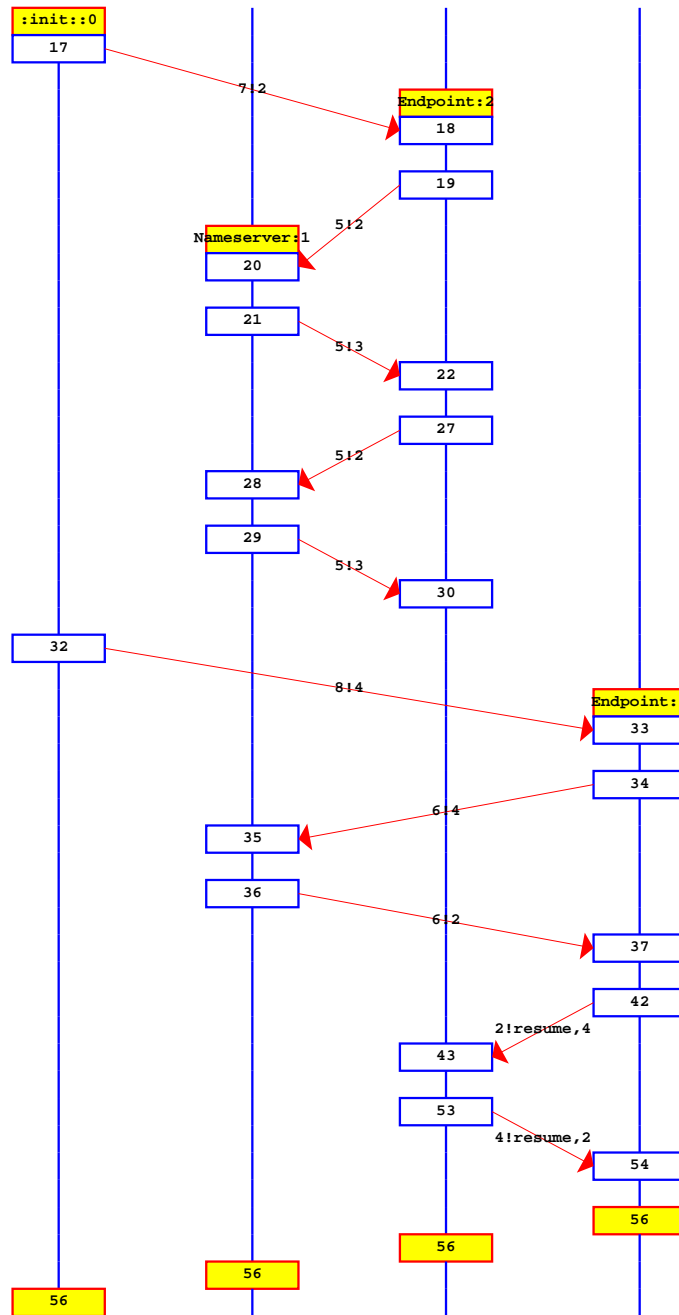


Figure 4.6: Deadlock in an incomplete SMP liveness model

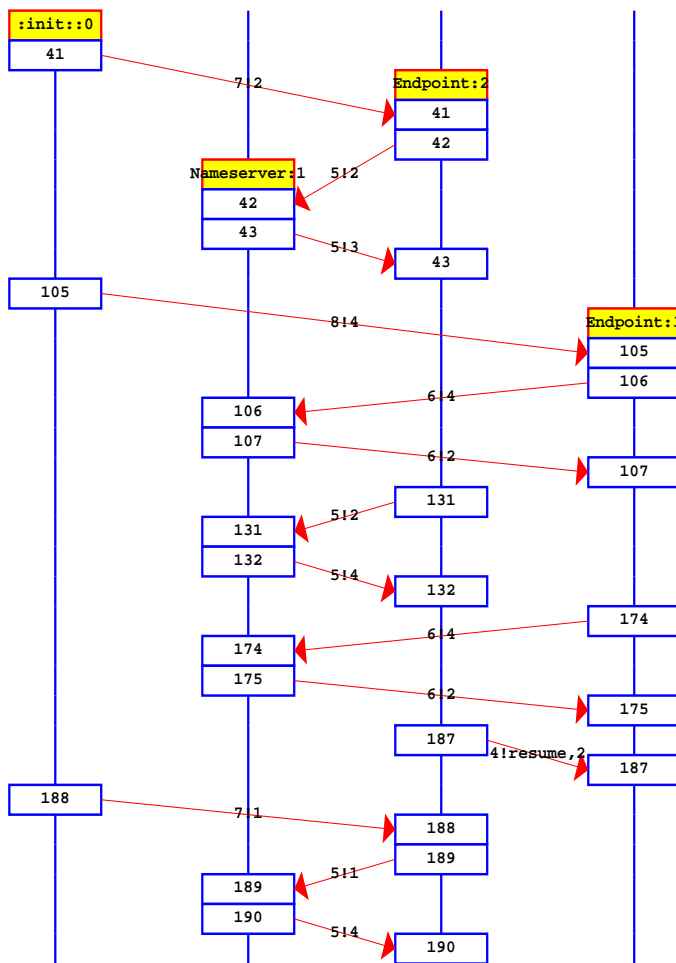


Figure 4.7: SMP liveness model execution

## Verification Results

Deadlock-freedom for the liveness model was verified in SPIN version 4.2.6 on a 3 GHz Pentium 4 computer running Linux, using the following commands:

```
$ spin -a -X smp-liveness-synchronous-suspend-fix.promela
$ gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=3100 -DSAFETY \
-DXUSAFE -DNOFAIR pan.c
$ ./pan -v -X -m10000 -w19 -c1
```

No invalid endstates were found. Concerning messages we discovered, using SPIN's reachability analysis functionality, that:

- processes should not receive *resume\_ok* or *resume\_denied* messages while in state `ACTIVE`
- processes should not receive *suspend* messages while in `SENT_RESUME`
- processes should not receive *resume\_ok* or *suspend* messages while in state `READY_RESUME`

It remains to be seen whether there are implementation-specific issues that violate these claims. Another interesting reachability result is that processes may receive *resume\_ok* messages while in state `SUSPENDED`, and that ignoring such messages does not cause any deadlocks.

#### 4.3.4 Development and Verification History

The SMP state machine is largely the result of test-driven, implementation-oriented development rather than careful high-level design, which may be the chief reason why the formal correctness of the protocol turned out to be difficult to specify, especially liveness. Even after modelling and model checking, some parts of the protocol—for example, event processing and the TCP state controller—remain implementation-defined.

One of the central problems during modelling of the state machine concerned how failed attempts to send control messages should be represented. In practice, such failures can have many causes, and may or may not affect both communicating endpoints simultaneously. Considerable time was spent investigating different control structures, with the aim of accounting for as many scenarios as possible in one model. The checkpoint protocol model, in which there can be no failed attempts to send messages, went through significantly fewer revisions than the primary safety model.

Some of the changes made to the initial checkpoint protocol (detailed in section 4.2.3) had to be propagated to the state machine procedure rules. Specifically, in the implementation, any endpoint, not just the connection initiator, can be one checkpoint ahead of the other during session resumption. Hence, both processes must be able to update a checkpoint after receiving *resume\_ok* when in the state `SENT_RESUME`, which is not the case in the current protocol version.

#### 4.3.5 Concluding Remarks

There are well-known problems with implementing synchronous communication in an inherently asynchronous network. The probability of a process considering an erroneous message exchange as error-free goes down as more confirmation messages are added to the protocol, but is never zero. It may be possible to make SMP tolerant to such errors by changing the procedure rules.

## Chapter 5

# Conclusions and Further Work

“The job of formal methods is to elucidate the assumptions upon which formal correctness depends.”

—C. A. R. Hoare

We conclude the investigation and try to evaluate the reliability of our results. In addition, some observations made during verification, along with pointers to possible further work, are given.

### 5.1 Result Summary

We presented a network protocol, the Session Management Protocol (SMP), and its use for handling connection resumption and data integrity in a session-layer based architectural approach to the problem of providing mobile, disconnection- and delay-tolerant communication. The choice of formal methods for the verification of SMP fell on model checking with SPIN, following a tool evaluation focusing on the expressiveness of the modelling language and model checker efficiency in theory and practice.

We described in section 4.2.3 the discovery by formal methods of a counterexample in the original version of the checkpoint protocol, which handles communication state information for session resumption, and noted that this error prompted revisions in other parts of the protocol.

A central idea in the models designed for verification of protocol safety and liveness was to use PROMELA channels as both network addresses and sockets. By making restrictions concerning possible events for endpoints, models capturing session resumption could be formulated. For each model describing a particular part of SMP, we gave rationale for modelling choices and how they related to verification.

The main contribution of the verification effort lies in the protocol specifications and models detailed in chapter 4. Again, the complete models are available electronically at <http://www.palmskog.net/exjobb/>. To sum up, we have seen that a protocol verification effort may be beneficial in that the protocol procedure rules are formulated unambiguously and without redundancy, the environmental assumptions are made explicit, and in that it may characterize errors and suggest ways to lessen complexity.

## 5.2 Result Reliability

First of all, the presented protocol models are parameterized on message queue sizes and other values. This means that each model is a sample from an infinite family of models. A necessary condition for verifying the whole protocol would be to verify every member of each model family, which is clearly not possible using only model checking. Instead, some other more advanced framework relying on induction may have to be used [11]. However, such an undertaking was considered outside the scope of the present investigation. Instead, attempts were made to verify models for a reasonable number of parameter values, using both exhaustive and partial state space searches.

Even having highlighted these issues, we believe the checkpoint protocol has been verified rigorously, on account of the service specification being straightforward and capable of formalization with a limited number of assumptions. In contrast, the verification of SMP session resumption and liveness may be considered more problematic and experimental, due to the many interactions with the environment that take place (e.g. through network events). The restriction of protocol models to certain scenarios, while enabling swifter verification, may have resulted in certain environmental assumptions and executions being omitted in the models. Still, we claim some of the fundamental components of SMP have been accurately captured and proven correct.

No attempt was made to implement and test the suggested changes in the protocol. Such an undertaking could possibly have resulted in further confirmation of the suggested protocol changes being feasible.

## 5.3 Observations

The fact that verification began relatively late in the protocol development process meant that significant time was spent formalizing the procedure rules and the environmental assumptions from a protocol implementation. As Dijkstra and Gries have emphasized [12], there are considerable benefits to letting verification and development of software systems proceed in parallel. Experience from working with SMP supports the sentiment that verification

should be an integrated part of all stages of development of a system where correctness is essential [42].

During the verification process, the following observations on SPIN usage were made:

- Counterexamples in models, if any, could generally be found when inspecting the first few hundred transitions; deep errors were much less common and were invariably discovered to be modelling errors.
- The bitstate hashing algorithm, which offers very low memory consumptions in exchange for skipping portions of model state space, did not fail to find any counterexamples that an exhaustive verification run found.
- Verification in SPIN is far from automatic, even given the models and specifications; verification runs needed to be constantly supervised to detect improper parameter values and estimate state space sizes. There is long way to go before verification is as easy as pushing a button.
- A considerable problem when using SPIN concerns how to optimize the size of the memory allocated to the depth-first search stack. Because of this issue, a lot of time needs to be spent exploring the state space of models through repeated attempts at exhaustive verification with different search depth limitations.

## 5.4 Future Work

Committing and testing the proposed changes to the proof-of-concept implementation is of particular interest for further development. The statement made at the end of section 4.2.3—that a probabilistic analysis of checkpoint protocol performance may be needed—holds true about SMP as a whole. Although a simple session-aware file transfer application has been implemented and is showing promise in terms of throughput and time spent on attachment changes, there is still no theoretical basis for speaking about SMP performance.

There were few TCP-specific elements in the protocol models, indicating that SMP can be made to work with, in principle, any reliable transport-layer protocol. Using an unreliable protocol such as the User Datagram Protocol (UDP) instead of TCP is another matter, however. Handling checkpoints would become redundant for any session using UDP. However, the deadlock-freedom of the state machine would still be relevant. Hence, incorporating support for UDP in the session layer would be an interesting task.

The original intent was to formally describe and verify both SMP itself and its interaction with TCP during rebind, but as SMP verification progressed, it became apparent that, due to time constraints, verifying interaction was not possible. As it stands, such interaction remains undefined—there seems to be no straightforward way of specifying how it should take place. Reasonably, timers and timeout values are important factors, suggesting that verification using a real-time model checker would be appropriate.

# References

- [1] W. Almesberger. TCP connection passing. In *Linux Symposium*, July 2002.
- [2] P. Arvidsson and M. Widell. Design of a session layer based system for endpoint mobility. Master's thesis, KTH, 2006.
- [3] J. C. M. Baeten and W. P. Weijland. *Process algebra*. Cambridge University Press, New York, NY, USA, 1990.
- [4] V. G. Cerf and E. Cain. The DoD internet architecture model. *Computer Networks*, 7:307–318, Oct. 1983.
- [5] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.
- [6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
- [7] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: a semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, 1993.
- [8] E. W. Dijkstra. Notes on Structured Programming.  
<http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>, Apr. 1970.
- [9] E. W. Dijkstra. On the cruelty of really teaching computing science.  
<http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF>, Dec. 1988.
- [10] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *5th Intl. Conference on Verification, Model Checking and Abstract Interpretation*, 2004.
- [11] G. E. Gallasch and J. Billington. A parametric state space for the analysis of the infinite class of stop-and-wait protocols. In *Proc. 13th Spin Workshop on Model Checking Software*, volume 3925 of *Lecture Notes in Computer Science*. Springer, 2006.

- [12] D. Gries. *The Science of Programming*. Springer-Verlag, New York, USA, 1981.
- [13] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1):53–56, 1983.
- [14] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.
- [15] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Upper Saddle River, NJ, USA, 1991.
- [16] G. J. Holzmann. An analysis of bitstate hashing. In *Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314, Warsaw, Poland, 1995. Chapman & Hall.
- [17] G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
- [18] G. J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.
- [19] International Organization for Standardization. ISO standard 7498-1, "Information Processing Systems - OSI Reference Model. The Basic Model". [http://www.acm.org/sigcomm/standards/iso\\_stds/OSI\\_MODEL/ISO\\_IEC\\_7498-1.TXT](http://www.acm.org/sigcomm/standards/iso_stds/OSI_MODEL/ISO_IEC_7498-1.TXT).
- [20] International Organization for Standardization. ISO standard 9834-8, "Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components". <http://www.itu.int/ITU-T/studygroups/com17/oid/X.667-E.pdf>.
- [21] Y. Ismailov, J. Höller, S. Herborn, and A. Seneviratne. Internet mobility: An approach to mobile end-system design. In *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICNICONSMCL)*, volume 0, pages 124–131, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
- [22] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering SE-3*, 2:125–143, 1977.
- [23] L. Lamport. *Information Processing 83*, chapter What Good Is Temporal Logic?, pages 657–668. Elsevier, 1983.
- [24] K. L. McMillan. *Symbolic Model Checking: an Approach to the State Explosion Problem*. Kluwer Academic, 1993.
- [25] R. Milner. *Communication and concurrency*. Prentice Hall, Hertfordshire, UK, 1989.

- [26] R. Milner. Functions as processes. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 167–180, New York, NY, USA, 1990. Springer-Verlag New York.
- [27] R. Milner. Elements of interaction: Turing award lecture. *Commun. ACM*, 36(1):78–89, 1993.
- [28] R. Milner. *Communicating and Mobile Systems: the  $\pi$ -Calculus*. Cambridge University Press, May 1999.
- [29] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, Sept. 1992.
- [30] J. Mogul et al. Unveiling the transport. In *2nd Workshop on Hot Topics in Networks*, Nov. 2003.
- [31] O. Müller. I/O automata and beyond: Temporal logic and abstraction in Isabelle. In *Proc. 11th Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 331–348. Springer, 1998.
- [32] P. Nikander, J. Ylitalo, and J. Wall. Integrating security, mobility, and multi-homing in a HIP way. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, pages 87–99. Ericsson Research NomadicLab, feb 2003.
- [33] S. Owicki and D. Gries. An axiomatic proof technique of parallel programs i. *Acta Informatica*, 6:319–340, 1976.
- [34] J. Parrow. Verifying a CSMA/CD-protocol with CCS. In *Proceedings of the IFIP WG6.1 Eighth International Symposium on Protocol Specification, Testing, and Verification*, pages 373–384, June 1988.
- [35] L. C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.
- [36] D. A. Peled. *Software Reliability Methods*. Springer-Verlag New York, Secaucus, NJ, USA, 2001.
- [37] C. Perkins. RFC 2002: IP mobility support, Oct. 1996. Updated by RFC2290 [46]. Status: PROPOSED STANDARD.
- [38] J. L. Peterson and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, second edition, 1985.
- [39] G. D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5(3):452–487, 1976.

- [40] A. Pnueli. A temporal logic for concurrent programs. *Theoretical Computer Science*, 1(13):45–60, 1980.
- [41] K. Popper. *Conjectures and Refutations*. Routledge, London, 1990.
- [42] T. C. Ruys. *Towards Effective Model Checking*. PhD thesis, University of Twente, 2001.
- [43] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1993. CST-99-93 (also published as ECS-LFCS-93-266).
- [44] A. Snoeren, H. Balakrishnan, and F. Kaashoek. Reconsidering Internet Mobility. In *8th Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, May 2001.
- [45] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *6th ACM/IEEE Intl. Conference on Mobile Computing and Networking*, Aug. 2000.
- [46] J. Solomon and S. Glass. RFC 2290: Mobile-IPv4 configuration option for PPP IPCP, Feb. 1998. Updates RFC2002 [37]. Status: PROPOSED STANDARD.
- [47] H. Song and K. J. Compton. Verifying  $\pi$ -calculus processes by Promela translation. *Technical Report 472*, University of Michigan, 2003.
- [48] C. Stirling. The joys of bisimulation. *Lecture Notes in Computer Science*, 1450:142–152, 1998.
- [49] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.
- [50] R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, 1976.
- [51] A. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42 of 2, 1936.
- [52] P. H. J. van Eijk, C. A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS: Results of the Esprit SEDOS Project*. Elsevier Science, Amsterdam, the Netherlands, 1989.
- [53] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. Logic in Computer Science*, pages 322–331, 1986.

- [54] B. Victor. *A Verification Tool for the Polyadic  $\pi$ -Calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.
- [55] D. Walker. Analysing mutual exclusion algorithms using CCS. *Formal Aspects of Computing*, 1:273–292, 1989.
- [56] Y. Wang. Mobility support for networked applications built in the TCP/IP stack. Master’s thesis, KTH, 2006.
- [57] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.