Modellierung von Kommunikationsprotokollen

Studienarbeit

Mike Herzog

10. März 2009

Betreuer:

Dipl.-Inf. Daniela Weinberg

Humboldt Universität zu Berlin
Institut für Informatik
Unter den Linden 6
10099 Berlin

Zusammenfassung

Aufgabe der vorliegenden Arbeit ist es, die Protokolle HTTP, SMTP und SSL als offene Netze, einer speziellen Petrinetz-Klasse, zu modellieren. Die Aufmerksamkeit gilt dabei der Kommunikation, wobei interne Zustände konkreter Instanzen nur abstrakt dargestellt werden.

Beim HTTP spielen die Inhalte der Nachrichten eine erhebliche Rolle. Diese können in offenen Netzen nicht modelliert werden. Als Alternative können die Inhalte der Nachrichten einerseits so weit verallgemeinert werden, dass ein einfaches, wenig anschauliches Modell entsteht. Andererseits können diese Inhalte — wenn sie erschöpfend abgebildet werden — aufgrund eines exponentiellen Wachstums der Anzahl der Knoten im resultierenden Netz, nicht dargestellt werden.

Das SMTP lässt sich anschaulich abbilden. Im Gegensatz zum HTTP lassen sich Nachrichten in 13 Klassen zusammen fassen. Es wurde ein Modell erarbeitet, das sowohl der Intuition dient, als auch eine gute, skalierbare Grundlage für Benchmarks des Analyse-Programms Fiona ist.

Unser SSL-Modell veranschaulicht das SSL-Protokoll intuitiv. Das synchrone SSL lässt sich jedoch wegen des Paradigmas der asynchronen Kommunikation der offenen Netze nur bedingt modellieren. Für Benchmarks ist es nur bedingt geeignet, da es nicht skalierbar ist.

Inhaltsverzeichnis

1 Einleitung und Definitionen **1**
1.1 Aufbau der Arbeit 1
1.2 Begriffe und Definitionen 1
1.3 Fiona 3
1.4 OSI Referenzmodell 3

2 Hypertext Transfer Protokoll **5**
2.1 Beschreibung des HTTPs 5
2.2 Ein simples Modell 7
2.3 Analyse des simplen Modells 7
2.4 Ein komplexes HTTP-Modell 8
2.5 Analyse der Modelle des HTTPs 12
2.6 Fazit zur Modellierung des HTTPs 12

3 Simple Mail Transfer Protocol **17**
3.1 Beschreibung des SMTPs 17
3.2 Modellierung des SMTPs 19
3.3 Analyse des SMTP-Modells 25
3.4 Fazit zur Modellierung des SMTPs 26

4 Secure Socket Layer **29**
4.1 Der SSL-Handshake 29
4.2 Modell des SSL-Handshakes 31
4.3 Analyse des Modells des SSL-Handshakes 34
4.4 Fazit zur Modellierung des SSL 34

5 Verwandte Arbeiten und Fazit **37**
5.1 Verwandte Arbeiten 37
5.2 Fazit 37

Literaturverzeichnis **39**

Abbildungsverzeichnis

2.1 Genereller Ablauf des HTTPs 6
2.2 HTTP-Anfrage 6
2.3 HTTP-Antwort 7
2.4 HTTP-Server als offenes Netz 8
2.5 HTTP-Server als High-Level-Petrinetz 9
2.6 Teilnetz der Entfaltung des High-Level-Petrinetz' des HTTP-Servers . 15
2.7 HTTP-Server für Benchmarks 16

3.1 SMTP Design 17
3.2 Beispielhafte Mailtransaktion 18
3.3 Der Kern des SMTPs als offenes Netz 20
3.4 Das Noop-Kommando des SMTP 21
3.5 Das Help-Kommando des SMTP 21
3.6 Das Verify-Kommando des SMTP 22
3.7 Das Reset-Kommando des SMTP 23
3.8 Das Quit-Kommando des SMTP 24

4.1 Zustände einer SSL-Verbindung 30
4.2 SSL Handshake 1.Teil 32
4.3 SSL Handshake 2.Teil 33
4.4 Zustände einer SSL-Verbindung als offenes Netz 35

Tabellenverzeichnis

2.1 IG Statistiken des HTTP-Modells 13
2.2 OG Statistiken des HTTP-Modells 13

3.1 Statuscodes des SMTPs 25
3.2 Statistiken des SMTP-Modells 25
3.3 IG Statistiken des SMTP-Modells 26
3.4 OG Statistiken des SMTP-Modells 26

4.1 Statistiken des Modells es SSL-Handshakes 34
4.2 Statistiken des SSL-Modells aus Abb. 4.1 34

1 Einleitung und Definitionen

Das heutige Internet wird in sehr vielfältiger Art und Weise genutzt. Um einen reibungslosen Ablauf zwischen den unterschiedlichen Akteuren zu ermöglichen, müssen die verwendeten Kommunikationsprotokolle verlässlich sein. Die Prüfung der Verlässlichkeit eines Protokolls erfordert systematische formale Methoden. Analyse-Werkzeuge, die solche Methoden anwenden, erfordern ein formales Modell als Eingabe während die Spezifikationen meist umgangssprachlich formuliert sind.

1.1 Aufbau der Arbeit

In dieser Arbeit werden ausgewählte Kommunikationsprotokolle (HTTP, SMTP und SSL) als Services aufgefasst. Als *Service* versteht man im Allgemeinen ein selbstständiges Modul, das über eine wohl-definierte Schnittstelle mit seiner Umgebung über asynchrone Nachrichten kommuniziert. Services können als offene Netze, einer speziellen Petrinetz-Klasse, modelliert werden. Wir stellen die Kommunikation aus Sicht der Server als offene Netze dar. Das Hauptaugenmerk legen wir auf die Kommunikation, da das jeweilige Protokoll – nicht eine konkrete Implementation – dargestellt werden soll.

Wir werden zunächst in der Einleitung die verwendeten Begriffe einführen und das Analyse-Programm Fiona kurz vorstellen. Es werden dann nacheinander alle Protokolle abgehandelt. In Kapitel 2 wird das HTTP behandelt, in Kapitel 3 das SMTP und schliesslich in Kapitel 4 das SSL. Jedes Protokoll wird dem Einsatzgebiet nach eingeordnet und kurz beschrieben. Im Hauptteil wird jeweils die Modellierung beschrieben und anschließend diskutiert. Im letzten Teil jedes Kapitels werden die erarbeiteten Modelle mit Fiona analysiert und die Ergebnisse dargestellt. Am Ende der Arbeit werden wir verwandte Arbeiten vorstellen, in denen ebenfalls Kommunikationsprotokolle als Petrinetze modelliert werden, und zeigen, inwiefern sich die vorliegende Arbeit unterscheidet.

1.2 Begriffe und Definitionen

Wir benutzen in dieser Arbeit die folgende Definition eines *Petrinetzes* (vgl. [Rei98]):

Definition 1.1 (Petri Netz)
Ein Tripel $N = (P, T, F)$ heißt *Petrinetz*, falls gilt:

- P und T sind disjunkte Mengen
- $F \subseteq (P \times T) \cup (T \times P)$ ist eine zweistellige Relation, die Flussrelation von N.$_\star$

Netze, die dieser Definition entsprechen werden auch als Low-Level-Pertinetze bezeichnet.

Wir benutzen außerdem die Standardnotation für den Vor- und Nachbereich von Plätzen und Transitionen: $\bullet x = \{y \mid (y, x) \in F\}$ und $x\bullet = \{y \mid (x, y) \in F\}$.

Ein *offenes Netz* ist – intuitiv – ein um speziell ausgewiesene Interface-Plätze erweitertes Low-Level-Petrinetz. Jeder der *Interface-Plätze* stellt einen Kommunikationskanal des Services dar. Das Senden und Empfangen von Nachrichten wird als *Ereignis* bezeichnet. Wir abstrahieren vom Inhalt der Nachrichten und bilden lediglich deren vorkommen ab. Jeder Interface-Platz ist entweder Sende- oder Empfangsplatz. Auf diese Weise wird sicher gestellt, dass eine gesendete Nachricht nicht vom Service „zurück genommen" und auch das Empfangen einer Nachricht nicht ungeschehen gemacht werden kann. Wir verwenden die Definition von [MRS05, LMW07]:

Definition 1.2 (Offenes Netz)
Ein *offenes Netz* $N = (P, T, F, I, O, m_0, \Omega)$ setzt sich aus einem Petrinetz (P, T, F) und

- einem Interface, das aus einer Menge $I \subseteq P$ von Inputplätzen mit $\bullet p = \emptyset$ für alle $p \in I$ und einer Menge $O \subseteq P$ von Outputplätzen mit $p\bullet = \emptyset$ für alle $p \in O$ und $I \cap O = \emptyset$, und
- einer ausgezeichneten Anfangsmarkierung m_0 und einer Menge Ω von Endmarkierungen, so dass keine Transition von N in einem $m \in \Omega$ aktiviert ist, $_\star$

zusammen.

Eine Korrektheitseigenschaft eines Service ist die *Bedienbarkeit*. Unter Bedienbarkeit verstehen wir, dass die Komposition zweier Services keine Deadlocks enthält. Als einzige Ausnahme seien Deadlocks, die Endzustände der Komposition sind, zugelassen. Bedienbarkeit gibt somit an, ob es einen Partner für einen Service gibt, mit der Service in einen Endzustand gelangt. Einen solche Partner bezeichnen wir als *Controller*. Bedienbarkeit lässt sich beispielsweise mit Hilfe des Interaktionsgraphen (IG) entscheiden (vgl. [Wei08]). Vollständige Informationen wie Partner mit einem Service interagieren können, liefert die *Bedienungsanleitung* (Operating Guideline, OG) [MRS05, LMW07]. Die OG charakterisiert alle Controller des Services. Wenn der Service nicht bedienbar ist, es also keine Controller gibt, ist die OG leer. Bedienbarkeit lässt sich somit auch mit Hilfe der OG entscheiden, der IG ist jedoch effizienter.

Für einige unserer Modelle müssen Marken voneinander unterscheidbar sein. Dies ist per Definition mit Low-Level-Petrinetzen nicht möglich. Diese Modelle werden als algebraische Netze nach Reisig[Rei98] dargestellt. Algebraische Netze sind eine Klasse der High-Level-Petrinetze, da ihre Marken unterscheidbar sein können[Jen95].

1.3 Fiona

Ein weiteres Ziel, dem diese Arbeit Rechnung trägt, ist das Erstellen von aussagekräftigen Beispielen für *Benchmarks* des Programms FIONA[1]. Mit FIONA kann u. a. die Korrektheit des Verhaltens eines Services überprüft werden. Dazu werden Controller synthetisiert. Im Rahmen dieser Arbeit werden wir für verschiedene Netze den IG und die OG berechnen lassen. Die Ausgabe von FIONA sind Graphen. Während deren Berechnung werden mehr Knoten untersucht als im Ergebnis ausgegeben werden. Wir werden die Gesamtzahl der untersuchten Knoten und die Anzahl der Knoten der Controller, die bei FIONA intern als „blaue" Knoten bezeichnet werden, gegenüber stellen.

Die hier vorgestellten Modelle werden im Rahmen dieser Arbeit mit FIONA 3.0 analysiert. Der Aufruf für die Berechnung des IGs ist:

```
fiona -t ig -s allnodes -p no-dot <NETZ>.
```

Der Aufruf für die Berechnung der OGs ist entsprechend:

```
fiona -t og -s allnodes -p no-dot <NETZ>.
```

1.4 OSI Referenzmodell

Das *ISO-OSI-Referenzmodell* (OSI-Modell) wurde von der International Standards Organization (ISO) entwickelt. Es behandelt die Vernetzung offener Systeme (Open Systems Interconnection, OSI). Im OSI-Modell werden eine Reihe voneinander unabhängiger Protokolle beschrieben, die aufeinander aufbauen. Jedes Protokoll kommuniziert dabei nur mit der jeweils niedrigeren Schicht und stellt der jeweils Höheren Dienste zur Verfügung. Es wird heute „kaum noch verwendet"[Tan03], ist jedoch so offen formuliert, dass Netzwerkprotokolle einer der sieben Schichten zugeordnet werden. Die Protokolle, die wir in der vorliegenden Arbeit modellieren gehören der höchsten, der Anwendungsschicht, an.

[1] Verfügbar bei `http://www.service-technology.org/fiona`

2 Hypertext Transfer Protokoll

Das *Hypertext Transfer Protokoll* (HTTP) ist ein Protokoll der Anwendungsschicht des OSI-Referenzmodells und ist „das Standardübertragungsprotokoll im Web” [Tan03]. Es wird in RFC 2616[Fie99] der IETF beschrieben. HTTP ist zustandslos und definiert neben dem Laden von WWW-Seiten auch persistente Verbindungen, das Aushandeln von Optionen und das Verhalten von Proxys und Caches. Dazu werden acht Methoden und 47 Header definiert. Des Weiteren kann es durch eigene Methoden und Header erweitert werden um individuellen Ansprüchen zu genügen [Fie99, S.7].

Wir werden in diesem Kapitel zunächst das Einsatzgebiet des HTTPs und die Daten einer beispielhaften Sitzung skizzieren. Wir werden anschliessend ein simples Modell entwickeln, das dann weiter entwickelt wird. Das entsprechend komplexere Modell ist ein High-Level-Petrinetz und dessen Entfaltung. Für die Benchmarks von Fiona wird noch ein weiteres Modell vorgestellt und analysiert.

2.1 Beschreibung des HTTPs

Schematisch stellt sich der Ablauf einer Übertragung im einfachsten Fall folgendermaßen dar (siehe Abbildung 2.1):

- Der Client (user agent, UA) stellt eine TCP-Verbindung (v) zum Server, auf dem sich die gewünschte Ressource befindet, (origin of the ressource, O) her.
- Anschliessend schickt der Client eine Anfrage. Diese wird vom Server empfangen. Dieser Teil wird als „request chain” bezeichnet.
- Der Server interpretiert seinerseits die Anfrage und beantwortet sie mit einer Antwort. Das ist die „response chain”.
- Für weitere Anfragen sollte der Client die TCP-Verbindung geöffnet halten bis sie nicht mehr benötigt wird. Weitere Anfragen treten beispielsweise auf, wenn Grafiken in eine WWW-Seite eingebettet sind.

Falls ein oder mehrere Caches involviert sind, so besteht die Verbindung zwischen dem Client und dem Cache. Der Cache kann seinerseits wieder mit einem Cache oder mit dem Server verbunden sein. Es werden entsprechend mehrere HTTP-Verbindungen

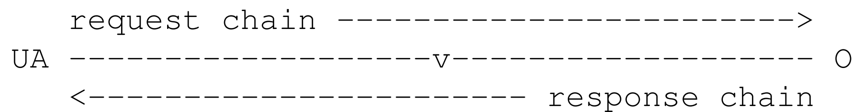


Abbildung 2.1: Genereller Ablauf des HTTPs

hergestellt. Das Modell wird somit komplexer. Für den Zweck dieser Arbeit genügt es, die Interaktion zweier Partner zu modellieren.

Ein Beispiel für die HTTP-Anfrage für eine WWW-Seite ist in Abbildung 2.2 gezeigt. Eine Anfrage besteht aus vier Teilen:

1. Dem Schlüsselwort der Anfrage. Es wird auch als Methode bezeichnet und kann CONECT, DELETE, GET, HEAD, OPTIONS, POST, PUT oder TRACE sein. In unserem Beispiel ist dies GET.
2. Der Ressource, auf die sich die Anfrage bezieht. Dies ist beispielsweise die WWW-Seite, die angezeigt werden soll, hier `/pub/WWW/TheProject.html`.
3. Der Protkoll-Version. In dieser Arbeit wird dies immer `HTTP/1.1` sein.
4. Verschiedene Header, in denen Optionen festgelegt werden, die die Anfrage oder die erwartete Antwort genauer spezifizieren. RFC 2616 zählt 47 verschiedene Header auf. Es gibt spezielle *Request*- und *Response*-Header sowie allgemeine Nachrichten Header, die für alle HTTP-Nachrichten verwendet werden können. Der einzige HTTP-Header, der in jeder Anfrage angegeben werden muss, ist der `Host`-Header[Fie99, S.80]. Damit wird die Domain angegeben, in der sich die Ressource befindet. Die übrigen Header und Optionen sind optional und werden in Abbildung 2.2 nicht gezeigt.

```
GET /pub/WWW/TheProject.html HTTP/1.1
Host: www.w3.org
```

Abbildung 2.2: HTTP-Anfrage

Nach dem Empfangen und Interpretieren der Anfrage schickt der Server seinerseits eine HTTP-Antwort an den Client. Eine beispielhafte HTTP-Antwort zeigen wir in Abbildung 2.3, wo ein Teil eines Bildes übertragen wird. Die Antwort setzt sich aus einer Statuszeile sowie verschiedenen Headern zusammen.

1. Die Statuszeile enthält die Protokollversion, einen Statuscode, in unserem Beispiel `206`, und eine Beschreibung des Status' als Klartext, `Partial content`.
2. Anschliessend sind jeweils keine, ein oder mehrere generelle Header, Antwort-Header und Datensatz-Header („entity-header"), die alle den Nachrichten-Text

(„message-body”) näher bestimmen, möglich. Im Beispiel werden das Datum der Antwort (Date-Header), der Zeitpunkt der letzten Änderung der Ressource (Last-Modified-Header), welcher Teil des Bildes (Content-Range-Header), Größe des Nachrichtentexts (Content-Length-Header) und der MIME-Typ der Ressource (Content-Type-Header) übertragen.

3. Abschliessend enthält die HTTP-Antwort gegebenenfalls den Nachrichten-Text, in dem beispielsweise die angeforderten Daten enthalten sind.

```
HTTP/1.1 206 Partial content
Date: Wed, 15 Nov 1995 06:25:24 GMT
Last-Modified: Wed, 15 Nov 1995 04:58:08 GMT
Content-Range: bytes 21010-47021/47022
Content-Length: 26012
Content-Type: image/gif

08 15 11 00 101 ...
```

Abbildung 2.3: HTTP-Antwort

2.2 Ein simples Modell

Wir erstellen nun ein Offenes-Netz-Modell der Serverseite des HTTPs. Wir modellieren die HTTP-Nachrichten zunächst durch Marken, die keinerlei Information tragen. Das einfache Modell ist gerade das Netz, das der prinzipielle Ablauf impliziert. Es wird in Abbildung 2.4 dargestellt.

Unser Netz beginnt bei der Anfangsmarkierung p0. In diesem Anfangszustand kann Transition ?request schalten, sobald sich eine Marke auf dem Platz m_{request} befindet. Durch das Schalten geht der Server in seinen Busy-Zustand über, der durch eine Marke auf p1 abgebildet wird. Die Nachricht wurde empfangen. Im nächsten Schritt schaltet !response wodurch je eine Marke auf m_{response} und p2 produziert wird. Somit ist das Interpretieren und Beantworten ebenfalls modelliert und das Netz erreicht den Endzustand p2.

2.3 Analyse des simplen Modells

Das einfache Modell (siehe Abbildung 2.4) ist nicht geeignet das HTTP zu veranschaulichen. Dieses Modell zeigt nur den Nachrichtenfluss, jedoch keine Ressourcen oder Ereignisse. Aber eben diese sind essenziell für die Kommunikation via HTTP.

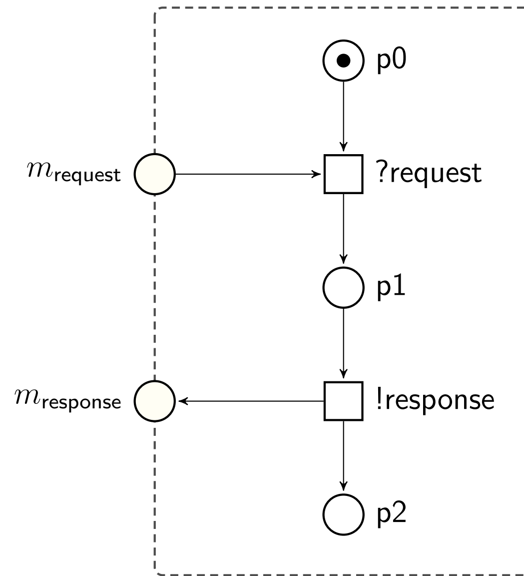


Abbildung 2.4: HTTP-Server als offenes Netz. $m_0 = [\mathsf{p0}], \Omega = \{[\mathsf{p2}]\}$

Für Benchmarks ist das im vorherigen Abschnitt vorgestellte Netz ebenfalls zu klein da sein Interaktionsgraph gerade eine Sequenz mit einem Sende- und einem Empfangsereigniss enthält.

2.4 Ein komplexes HTTP-Modell

Im vorherigen Abschnitt wurden die HTTP-Nachrichten abstrakt dargestellt. Entsprechend konnten wir mit dem Modell nicht weiter arbeiten. In diesem Abschnitt werden wir die Nachrichten genauer berücksichtigen. Wir werden die HTTP-Methoden, -Header und -Optionen und deren Verarbeitung im Einzelnen abbilden.

2.4.1 HTTP als High-Level-Petrinetz

Die Mengen der Plätze und Transitionen des Modells aus Abbildung 2.4 werden wir beibehalten. Wir ändern jedoch die Art der Marken auf den Interfaceplätzen. Im Folgenden stellen wir HTTP-Anfragen m_{request} als 4-Tupel $[M, R, V, O]$ dar. Dabei sei

- $M \in \{$CONNECT, DELETE, GET, HEAD, OPTIONS, POST, PUT, TRACE$\}$,
- R die Ressource, auf die sich die Anfrage bezieht,
- V die HTTP-Versionsnummer (in dieser Arbeit immer `HTTP/1.1`) und

- O die Liste der Header und Optionen.

Eine HTTP-Antwort m_{response} stellen wir analog als 4-Tupel $[V, S, O, D]$ dar. Dabei sei

- V die HTTP-Versionsnummer,
- S die HTTP-Statuszeile bestehend aus einem numerischen Statuscode und der entsprechenden sprachlichen Phrase,
- O eine Liste der Header und Optionen und
- D die Daten, sofern diese erforderlich sind.

Mit dieser Erweiterung können wir den Inhalt der Nachrichten charakterisieren.

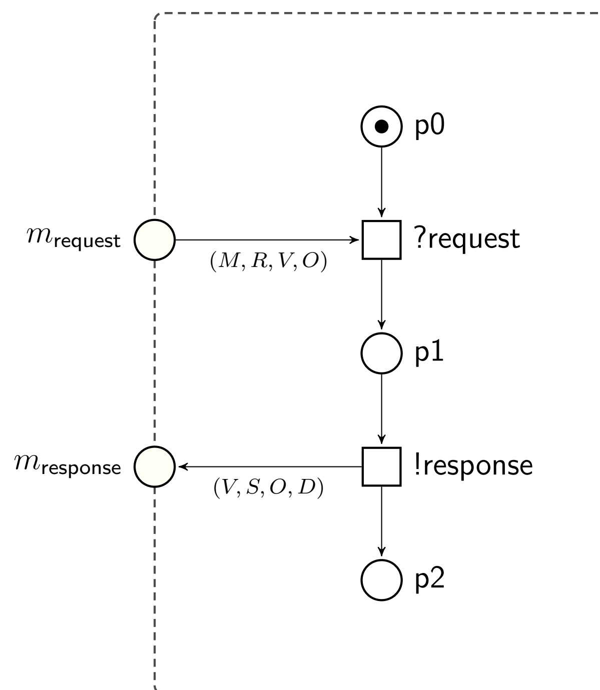


Abbildung 2.5: HTTP-Server als High-Level-Petrinetz. $m_0 = [\mathsf{p0}]$

2.4.2 Das komplexe Modell als offenes Netz

Offene Netze sind eine Klasse der Low-Level-Petrinetzte. Um das HTTP-Modell aus Abbildung 2.5 als offenes Netz darstellen zu können, müssen wir es entfalten. Da alle Domänen endlich sind, ist das entfaltete Netz ebenfalls endlich. Wenn es uns auch

nicht möglich ist das gesamte entfaltete Netz darzustellen (siehe Abschnitt 2.4.3), so konstruieren wir dennoch ein Teilnetz.

Als Teilmenge wählen wir die HTTP-Anfragen m'_{request} als 4-Tupel $[M', R, V, O']$ und die HTTP-Antworten m'_{response} als 4-Tupel $[V, S', O', D]$. Dabei sei

- $M' \in \{\mathsf{GET}\}$,
- R die Ressource,
- V die HTTP-Versionsnummer und
- $O' \in \{$ HOST, IF_MODIFIED, RANGE, CONTENT_LENGTH, ENCONDING $\}$, sowie
- $S' \in \{$ m200, m400 $\}$ und
- D Daten.

Das Entfalten des Netzes aus Abbildung 2.5 geschieht auf die folgende Art und Weise: Der Platz m_{request} wird gestrichen und durch Plätze für die Methode (GET), die Header (HOST, IF MODIFIED und RANGE) sowie deren Komplementplätze ersetzt. Die Transition ?request wird gestrichen und für jedes Element des 4-Tupels m'_{request} wird eine neue Transition hinzugefügt. Für M fügen wir die Transition t1', die GET empfängt, ein. Für jedes Element der Potenzmenge $\mathcal{P}(O')$ wird je eine Transition t2, ..., t9 eingefügt. Jede dieser Transitionen produziert eine Marke auf dem jeweils korrespondierenden Platz p2, ..., p9 (siehe Abbildung 2.6), die äquivalent zum ursprünglichen Platz p1 sind.

Das Nichtvorhandensein von Nachrichten, z.B. $\neg$HOST, modellieren wir explizit, um für jede Kombination von gesendeten Headern die passende Antwort modellieren zu können. Auf diese Weise stellen wir folglich sicher, dass eine Anfrage ohne Host-Header mit einer entsprechenden Fehlermeldung beantwortet wird. Das Nichtkonsumieren eines gesendeten Headers ist auf diese Weise nicht möglich.

Das bis hier beschriebene Teilnetz bildet den Empfang der HTTP-Anfrage ab. Im Folgenden wird nun das Senden der Antwort modelliert. Ist das Netz im Zustand p2, p4, p6 oder p8, wurde der HOST-Header nicht gesendet und es wird eine Fehlermeldung m400 (Client-Fehler: Bad Request [Fie99, S.42]) gesendet. In den Zuständen p3, p5, p7 oder p9 kann entweder m200 (OK) oder m400 (Fehler) gesendet werden. Außerdem können die Header CONTENT_LENGTH und ENCONDING optional gesendet werden. Das Netz geht jeweils nach dem Senden in den Endzustand Fin über.

2.4.3 Entfaltung des High-Level-Petrinetz'

Das in Abschnitt 2.4.1 erstellte Modell entspricht nicht der Definition eines offenen Netzes, da ihm ein High-Level-Petrinetz zugrunde liegt. Im vorherigen Abschnitt haben wir den Ablauf der Entfaltung beschrieben. In diesem Abschnitt werden wir zeigen, dass das vollständige entfaltete Netz zu groß ist um sinnvoll dargestellt zu werden.

Wir diskutieren nun die Größe des vollständigen Netzes. Diese ergibt sich aus der Summe der Plätze und Transitionen, zu denen ?request, p1 und !response entfaltet werden. Weil der Platz p1 den Zustand des Servers nach dem Empfangen der Anfrage modelliert, wird er zu genau so vielen Plätzen entfaltet, wie neue Transitionen für ?request entstehen. Wir werden uns daher auf die Betrachtung der Transitionen beschränken.

Die Transition ?request aus Abbildung 2.4 schaltet im High-Level-Netz für jede HTTP-Methode in einem anderen Modus. Damit tragen wir der Anforderung, dass unterschiedliche Methoden unterschiedliche Statuscodes generieren können, aus RFC 2616 Rechnung. Entsprechend der acht HTTP-Methoden entfalten wir ?request zu acht Transitionen. Die Ressource, auf die sich die Anfrage bezieht wird in unserem Modell nicht berücksichtigt, da sie in jeder Anfrage vorhanden ist und ihr konkreter Inhalt für das Protokoll unerheblich ist. Ebenso wird auch die obligatorische Protokollversion nicht berücksichtigt. Für die Header erhalten wir folgende obere Schranke: Es gibt 47 Header, von denen jeder entweder in der Anfrage vorkommt oder nicht. Diese unterteilen sich in 21 Request-Header, 7 Response-Header, 12 Entity-Header und 7 generelle Header. In der Anfrage können also $21 + 7$ Header vorkommen oder nicht. Wir erhalten weitere 2^{28} Schaltmodi für ?request.

Die Transition !response wird für jeden der Plätze, zu denen p1 entfaltet wurde, zu folgenden Schaltmodi entfaltet: Prinzipiell ist zu jeder Anfrage jeder Statuscode möglich. Es werden jedoch je nach Methode und konkreter Situation bestimmte Responsecodes *empfohlen*. Wir erhalten somit 39 Schaltmodi. Die 26 für eine Antwort in Frage kommenden Header ergeben weitere 2^{26} Modi. Für die obligatorische Versionsnummer und die optionalen Daten erhalten wir jeweils einen Schaltmodus. Insgesamt hat unser vollständig entfaltetes HTTP-Netz demnach

$$\begin{aligned} & |M| \times |R| \times |V| \times 2^{|O_{req}|} \times |V| \times |S| \times 2^{|O_{res}|} \times |D| \\ = \; & 8 \times 1 \times 1 \times 2^{28} \times 1 \times 39 \times 2^{26} \times 1 \\ = \; & 5{,}62 \times 10^{18} \end{aligned}$$

Transitionen.

2.5 Analyse der Modelle des HTTPs

Das High-Level-Petrinetz ermöglicht eine gute Intuition für die Funktionsweise des HTTPs. Jedoch sind offene Netze per Definition keine High-Level-Petrinetze. Die Größe des entfalteten Netzes ist exponentiell abhängig von der Anzahl der Abgebildeten HTTP-Elemente (Methoden, Header etc.), weshalb eine vollständige Modellierung nicht möglich ist.

2.5.1 Analyse des komplexen Modells

Das komplexe Modell kann als High-Level-Petrinetz *nicht* mit FIONA analysiert werden, da FIONA ein offenes Netz als Eingabe erwartet. Eine vollständige Entfaltung ist nicht darstellbar.

2.5.2 Ein Modell für Benchmarks

Für die Tests des Programms FIONA haben wir ein weiteres Modell in Anlehnung an Abbildung 2.4 erstellt. In diesem Modell (siehe Abbildung 2.7) wird zu jeder der acht Methoden des HTTPs ein ausgewählter Statuscode gesendet. Ein Platz counter beschränkt die Anzahl der möglichen Runden. $m_0 = [\mathsf{counter} : 10], \Omega = \{[\mathsf{p10}]\}$.

Tabelle 2.1 fasst die Ausgaben der Statistiken zur Erstellung der Interaktionsgraphen (IG) in Abhängigkeit der Anzahl der Marken auf counter zusammen.

Tabelle 2.2 fasst die Ausgaben der Statistiken zur Erstellung der Bedienungsanleitung (OG) in Abhängigkeit der Anzahl der Marken auf counter zusammen.

Die Anzahl der Knoten, Kanten und Zustände wächst bei IG und OG jeweils exponentiell mit der Anzahl der Marken auf counter.

2.6 Fazit zur Modellierung des HTTPs

Wir haben verschiedene Zustände des Ablaufs eines HTTP-Servers in einem Petrinetz dargestellt. Unterschiede ergaben sich dadurch, dass entweder von den Daten abstrahiert wurde oder nicht. Das Abstrahieren der Daten führte zu einem Netz mit nur zwei Transitionen, das sowohl für die Intuition als auch für Benchmarks zu klein ist. Beim HTTP spielen die gesendeten Daten eine zentrale Rolle. Das ausführliche Abbilden aller Daten andererseits würde zu einem (Low-Level-) Netz mit $5{,}62 \times 10^{18}$ Transitionen führen, was nicht darstellbar ist. Wir haben auch ein High-Level-Petrinetz

counter	Knoten		Kanten		Zustände	Zustände in Knoten	
1	10	(10)	16	(16)	26	26	(26)
2	47	(47)	144	(144)	293	275	(275)
3	140	(140)	608	(608)	1.823	1.644	(1.644)
4	303	(303)	1.632	(1.632)	7.028	6.373	(6.373)
5	522	(522)	3.216	(3.216)	19.449	17.374	(17.374)
10	1.792	(1.792)	13.312	(13.312)	148.420	130.491	(130.491)
20	4.352	(4.352)	33.792	(33.792)	424.905	371.061	(371.061)
50	12.032	(12.032)	95.232	(95.232)	1.252.921	1.092.771	(1.092.771)
100	24.832	(24.832)	197.632	(197.632)	2.632.933	2.295.621	(2.295.621)

Tabelle 2.1: IG Statistiken des HTTP-Modells aus Abbildung 2.7. Netz mit 26 Plätzen (davon 8 Input-Plätze und 8 Output-Plätze) und 16 Transitionen. In Klammern stehen jeweils die „blauen" Knoten, Kanten und Zustände in Knoten.

counter	Knoten		Kanten		Zustände	Zustände in Knoten	
1	513	(513)	6.160	(6.160)	2.750	2.560	(2.560)
2	769	(769)	9.232	(9.232)	12.093	10.240	(10.240)
3	1.025	(1.025)	12.304	(12.304)	29.346	25.088	(25.088)
4	1.281	(1.281)	15.376	(15.376)	52.022	45.536	(45.536)
5	1.537	(1.537)	18.448	(18.448)	75.898	68.672	(68.672)
10	2.817	(2.817)	33.808	(33.808)	205.221	188.811	(188.811)
20	5.377	(5.377)	64.528	(64.528)	463.706	429.381	(429.381)
50	13.057	(13.057)	156.688	(156.688)	1.236.670	1.151.091	(1.151.091)
100	25.857	(25.857)	310.288	(310.288)	2.534.539	2.353.941	(2.353.941)

Tabelle 2.2: OG Statistiken des HTTP-Modells aus Abbildung 2.7. Netz mit 26 Plätzen (davon 8 Input-Plätze und 8 Output-Plätze) und 16 Transitionen. In Klammern stehen jeweils die „blauen" Knoten, Kanten und Zustände in Knoten.

gezeigt, das jedoch nicht mit Fiona analysiert werden kann, da offenen Netzen per Definition Low-Level-Petrinetze zu Grunde liegen.

Um dem Ziel der Arbeit Rechnung zu tragen, aussagekräftige und skalierbare Benchmarks zu erstellen, wurde ein Netz vorgestellt, das nicht den HTTP-Standard widerspiegelt. Vielmehr modelliert es acht ausgewählte Spezialfälle.

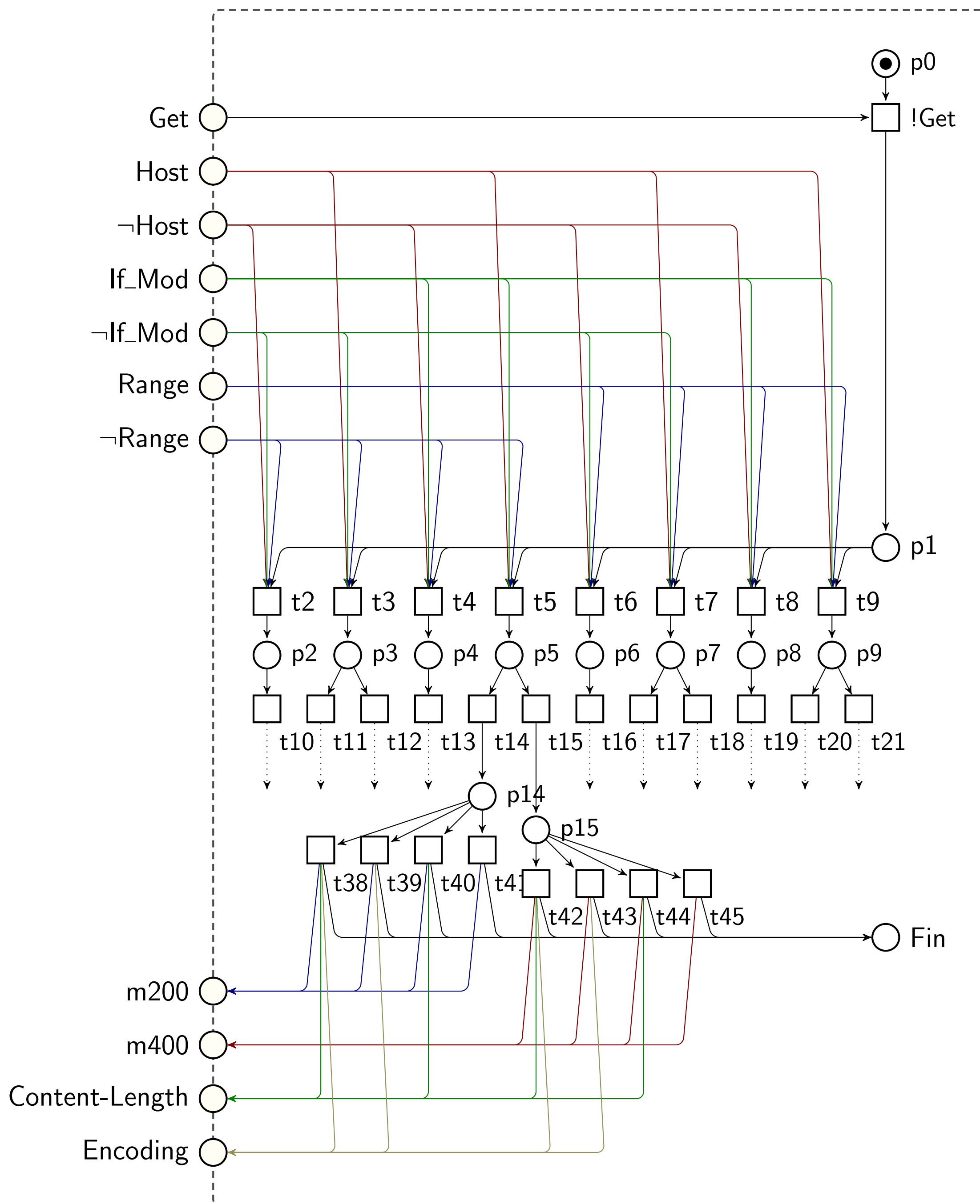


Abbildung 2.6: Teilnetz der Entfaltung des High-Level-Petrinetz' des HTTP-Servers. m_0 = [p0], Ω = {[Fin]}. Einige Kanten sind teilweise zusammengefasst, falls sie in gleicher Richtung *und* von oder zu denselben Knoten verlaufen.

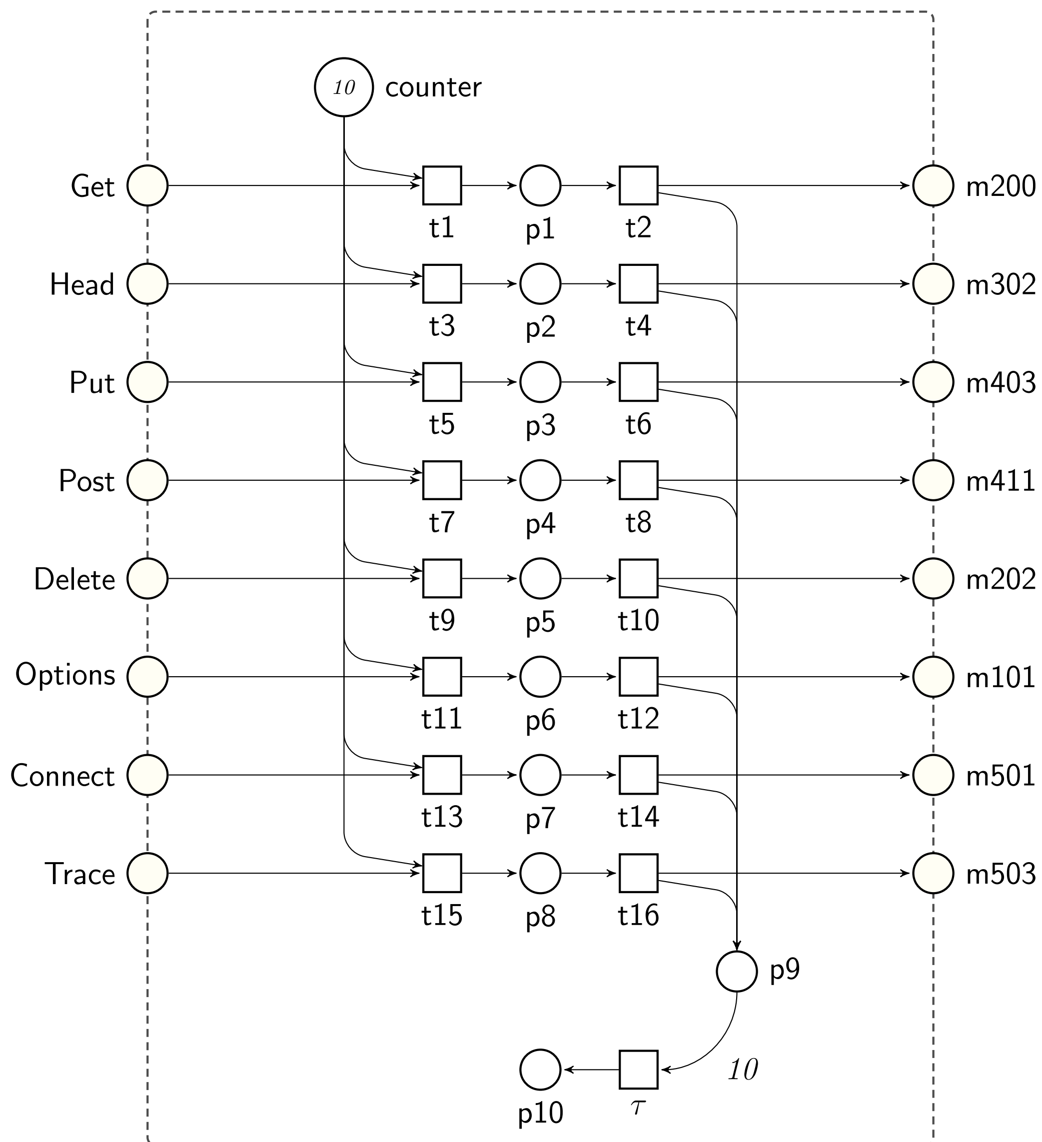


Abbildung 2.7: HTTP-Server für Benchmarks. $m_0 = [\mathsf{counter} : 10], \Omega = \{[\mathsf{p10}]\}$. Einige Kanten sind teilweise zusammengefasst, falls sie in gleicher Richtung *und* von oder zu denselben Knoten verlaufen.

3 Simple Mail Transfer Protocol

Das *Simple Mail Transfer Protocol* (SMTP) ist ein Protokoll der Anwendungsschicht des OSI-Modells, mit dem textbasierte Nachrichten (E-Mails) übertragen werden können. Es wird in RFC 2821[Kle01] der IETF spezifiziert. Das SMTP wird zur Übertragung von E-Mails vom Nutzer zum Mail-Server des Empfängers eingesetzt. Beim SMTP wird die Kommunikation vom Client initiiert, weshalb der Server ständig auf eine Verbindung warten muss. Abbildung 3.1 zeigt diesen Zweck des SMTPs schematisch. Als Client- oder Server-SMTP verstehen wir hierbei jeweils die Programme, die E-Mails und eventuelle Dateianhänge verarbeiten, versenden und empfangen. Da die Desktoprechner der Nutzer üblicherweise nicht rund um die Uhr mit dem Internet verbunden sind, können diese nicht ständig auf eingehende Nachrichten reagieren. Daher werden die E-Mails meist auf E-Mail-Servern gespeichert, wo sie vom Nutzer mit anderen Protokollen als SMTP herunter geladen werden können.

Wir werden in diesem Kapitel wird zunächst das Einsatzgebiet des SMTPs skizziert und anhand einer beispielhaften Sitzung werden einige Aspekte des Protokolls erläutert. Anschliessend wird ein Modell entwickelt und beschrieben. Im letzten Teil dieses Kapitels folgt die Analyse unseres Modells.

3.1 Beschreibung des SMTPs

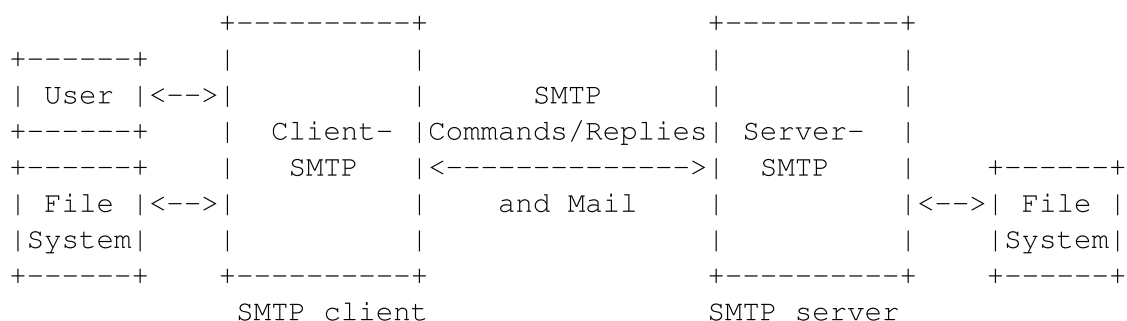


Abbildung 3.1: SMTP Design. Grafik: RFC 2821

Die Übertragung einer E-Mail wird als Mail-Transaktion bezeichnet. Abbildung 3.2 zeigt die Daten, die für eine beispielhafte E-Mail-Transaktion über das TCP[Inf81]

```
1:  > 220 mail.provider.com SMTP Foo Mailserver
2:  < EHLO mail.example.org
3:  > 250 Ok
4:  < MAIL FROM: alice@example.org
5:  > 250 Ok
6:  < RCPT TO: bob@provider.com
7:  > 250 Ok
8:  < DATA
9:  > 354 End data with .
10: < From: alice@example.org
11: < To: bob@provider.com
12: < Subject: Testmail
13: <
14: < Testmail
15: < .
16: > 250 Ok
17: < QUIT
18: > 221 Bye
```

Abbildung 3.2: Beispielhafte Mailtransaktion. Vollständige SMTP-Sitzung, in der Alice eine E-Mail an Bob schickt. Nachrichten von Alice' Client sind mit < und Antworten des SMTP-Servers mit > markiert.

übertragen werden. Die für eine E-Mailübertragung nötigen SMTP-Kommandos sind Hello (HELO), Mail (MAIL), Recipient (RCPT) und Data (DATA). Der Datenteil besteht üblicherweise aus einer MIME-Nachricht gemäß RFC 2045 [Fre96]. Das Ende der E-Mail-Daten wird durch <CRLF>.<CRLF> gekennzeichnet. Zu diesem Zeitpunkt könnte der Client weitere E-Mails übertragen. In Abbildung 3.2 beendet der Client die Sitzung durch Senden des Quit-Kommandos (QUIT). Der Server quittiert jedes der Kommandos mit einer Statusmeldung. Diese Statusmeldung muss einen der festgelegten numerischen Codes enthalten. Optional kann dem Code noch ein Klartext als Beschreibung folgen.

Das Kommando „Hello" dient dem Client sich am Server zu identifizieren. Es lautet je nach vom Client verwendeter Protokollversion HELO oder EHLO („extended Hello"). In der aktuellen Version hat der Client EHLO zu verwenden. Der Server muss jedoch auch mit älteren Clients kommunizieren können. Er muss entsprechend bei Clients, die sich mit HELO identifizieren auf Erweiterungen der RFC 2821 gegenüber ihrem Vorläufer RFC 821 verzichten können.

Wir werden im Folgenden ein Modell des SMTPs entwickeln, das alle für eine Mailtransaktion erforderlichen Kommandos und Statusmeldungen enthält. Die fehlenden Kommandos werden anschliessend nacheinander ergänzt.

3.2 Modellierung des SMTPs

Ausgehend von einer E-Mail-Transaktion modellieren wir den SMTP-Server als offenes Netz. Wir werden vom Inhalt der Nachrichten abstrahieren, so dass nur die Kommandos und numerischen Codes abgebildet werden. Analog zum Modell des HTTPs wird ein Statuscode der Form „`220 mail.provider.com SMTP Foo Mailserver`” als m220 abgebildet. Ein vereinfachtes Modell des SMTPs, das alle für eine E-Mail-Transktion erforderlichen Kommandos abbildet, zeigt Abbildung 3.3. Weiterhin enthält Abbildung 3.3 als Vereinfachung nur positive Rückmeldungen.

Mit diesem Modell kann man den Ablauf der E-Mail-Transktion aus Abbildung 3.2 nachvollziehen. Zusätzlich haben wir bereits Merkmale modelliert, die später benötigt werden. 1. Der Server kann im Zustand p1 entweder EHLO oder HELO empfangen, wie verlangt. 2. Wir haben die Möglichkeit mehrere Empfänger auszugeben modelliert. 3. Der Platz v ist genau dann markiert, wenn eine TCP-Verbindung hergestellt ist.

Bereits an diesem einfachen Modell des SMTPs stellt sich ein Grundsatz der offenen Netze als ungünstig für die Modellierung des SMTPs heraus: In offenen Netzen geschieht der Nachrichtenaustausch asynchron. Das *Simple Mail Transfer Protocoll* arbeitet dagegen synchron. Der SMTP-Server erwartet Nachrichten laut Spezifikation in einer definierten Reihenfolge. Er muss Nachrichten, die nicht in der erwarteten Reihenfolge eintreffen, nicht speichern und *kann* sie als Fehler zurückweisen.

3.2.1 Noop und Help

In diesem Modell sind nicht alle SMTP-Kommandos als Input-Plätze modelliert. Es fehlen noch sechs Kommandos, die wir im Folgenden ergänzen werden. Laut RFC 2821 muss der Server das Noop-Kommando (NOOP) immer mit dem Statuscode m250 beantworten. Am Zustand des Servers soll sich dabei nichts ändern. Dieses Kommando dient vor allem dazu die TCP-Verbindung, über die die SMTP-Sitzung abgewickelt wird, offen zu halten und einen Timeout zu verhindern. Da der Server nur dann ein NOOP empfangen kann, wenn auch ein Client verbunden ist, fragen wir den Platz v ab, der markiert ist, sobald ein Connect vom Server empfangen wird und eine Verbindung besteht. Abbildung 3.4 zeigt die Erweiterung des Netzes aus Abildung 3.3.

Mit dem Help-Kommando (HELP) kann der Client eine Liste unterstützter Kommandos abfragen. Der Standard schreibt hier keinen Code vor, mit dem der Server zu antworten hat. Es werden lediglich m211, m214, m502 und m504 vorgeschlagen. Verschiedene Server, darunter der Mail-Server der Humboldt Universität zu Berlin (`ir1.cms.hu-berlin.de`), verwenden m214. Daher werden wir in dieser Arbeit ebenfalls m214 verwenden. Analog zum Noop-Kommando kann HELP nur empfangen

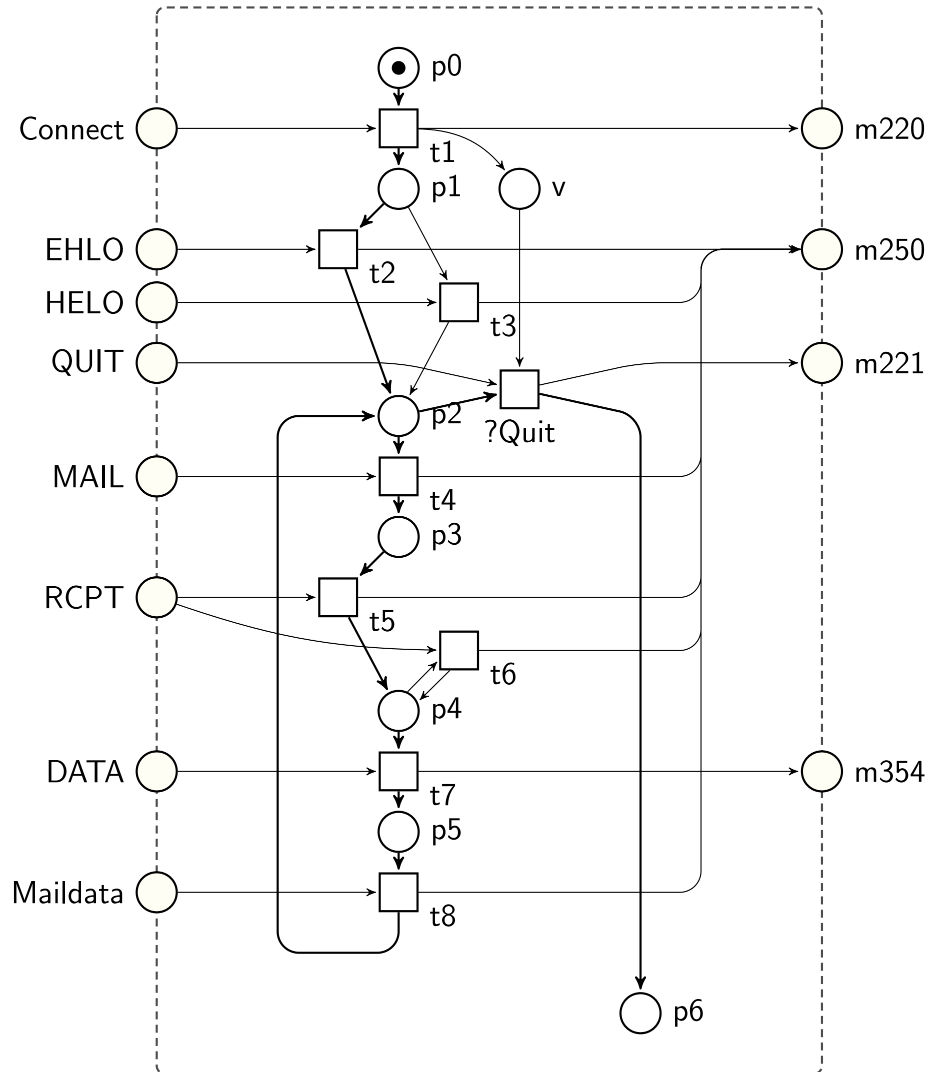


Abbildung 3.3: Der Kern des SMTPs als offenes Netz. $m_0 = [p_0]$, $\Omega = \{[p_6]\}$. Einige Kanten sind teilweise zusammengefasst, falls sie in gleicher Richtung *und* von oder zu denselben Knoten verlaufen.

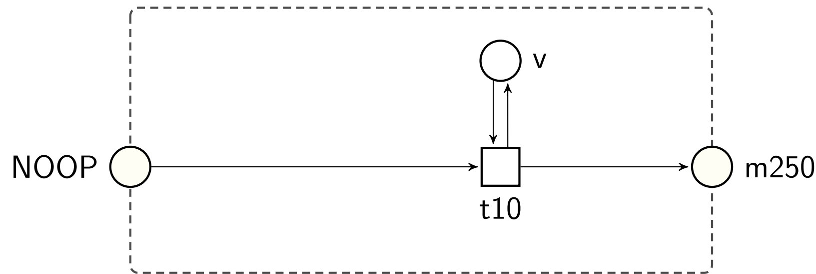


Abbildung 3.4: Das Noop-Kommando des SMTP zur Ergänzung des Modells aus Abbildung 3.3

werden, wenn eine Verbindung zum Server besteht, d.h. der Platz v markiert ist. Die Erweiterung des SMTP-Modells ist in Abbildung 3.5 dargestellt.

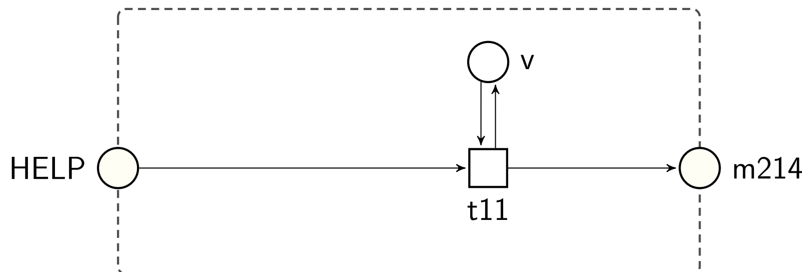


Abbildung 3.5: Das Help-Kommando des SMTP zur Ergänzung des Modells aus Abbildung 3.3

3.2.2 Verify und Expand

Das dritte Kommando, das dem Modell hinzugefügt werden soll, ist „Verify" (VRFY). Es dient der Überprüfung eines Empfängers. Wenn der Server für den gewünschten Empfänger Nachrichten zustellen kann, so antwortet er mit m250. Falls er dies nicht sicherstellen kann, aber die Empfängeradresse wohlgeformt ist, so kann er etwa mit m252 antworten. Des Weiteren sind noch m4yz- und m5yz-Antworten möglich, wenn eine Überprüfung nicht möglich ist oder fehl geschlagen ist. In dieser Arbeit werden exemplarisch nur zwei mögliche Antworten modelliert: m250 für den Erfolgsfall und m550 für unbekannte Adressaten. In unserem Modell tritt eine Transition t12 ein, sobald VRFY gesendet wurde. Diese Transition bildet das Prüfen des Empfängers ab und speichert das Ergebnis auf den Platz p12. Je nach Ergebnis tritt t13 oder t14 ein, so dass der Server entweder mit m250 oder mit m550 antwortet. Die Erweiterung ist in Abbildung 3.6 dargestellt.

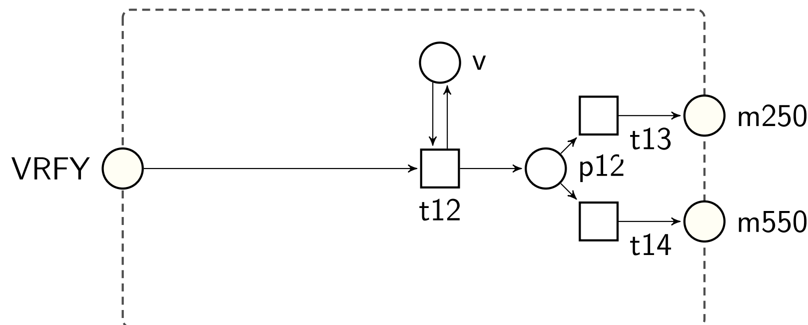


Abbildung 3.6: Das Verify-Kommando des SMTP zur Ergänzung des Modells aus Abbildung 3.3

Das Expand-Kommando (EXPN) dient dem Auflösen von E-Maillisten. Der Client kann auf diese Weise die Mitglieder einer E-Mailverteilerliste erfragen, falls diese Liste existiert und der Client berechtigt ist, diese Informationen zu erhalten. Dieses Kommando ist als optional deklariert, da möglicherweise generelle Sicherheitsaspekte gegen seine Implementation sprechen. Da es in unserem Modell ganz analog zum Verify-Kommando dargestellt werden würde, wird es nicht berücksichtigt.

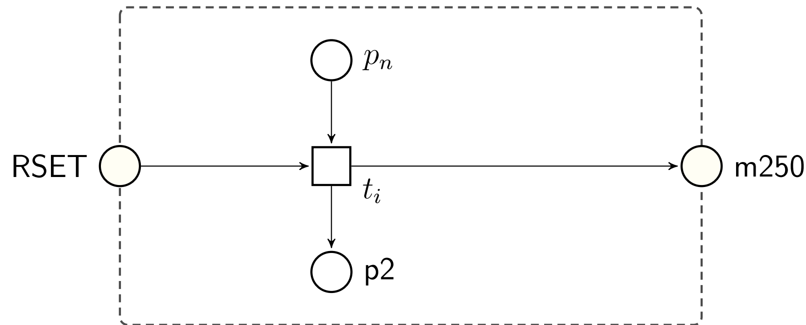
3.2.3 Reset

Empfängt der Server ein Reset-Kommando (RSET, aber auch EHLO oder HELO), so muss er die Mail- und Adress-Puffer der aktuellen E-Mailtransaktion löschen. Reset kann zu jeder Zeit gesendet werden und muss immer mit einer positiven Antwort (m250) beantwortet werden. Ausserhalb einer Mailtransaktion, also direkt nach EHLO oder nach der positiven Bestätigung nach Maildata, sind Reset-Kommandos ohne Effekt und funktionieren genau wie NOOP.

Um das Modell für Reset-Ereignisse zu erweitern fügen wir das in Abbildung 3.7 dargestellte Teilnetz in das SMTP-Modell ein. Das Teilnetz wird für alle Transitionen t_i der Tabelle in unser Modell eingefügt.

3.2.4 Quit

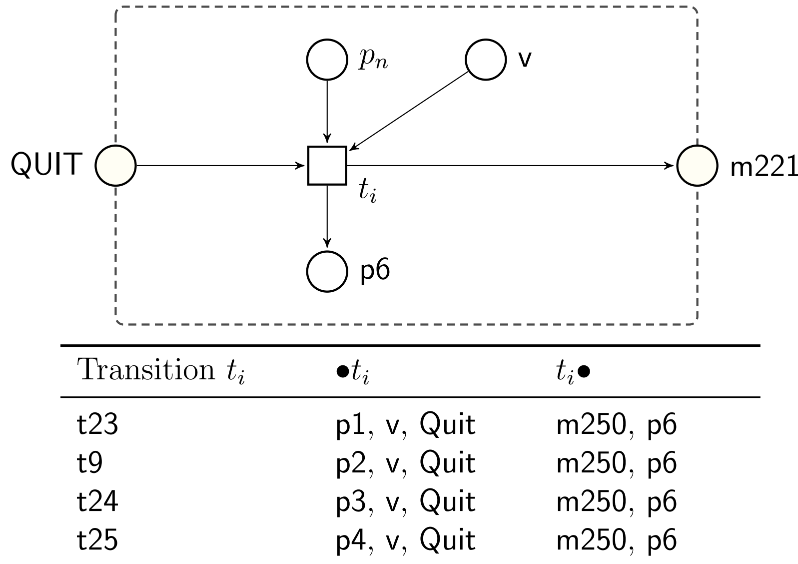
Das Quit-Kommando (QUIT) ist in Abbildung 3.3 bereits vorhanden. Es gestattet das Beenden der Verbindung, wenn das Netz im Zustand p2 ist. Laut Spezifikation muss ein QUIT jedoch „jederzeit" empfangen werden können. Abbildung 3.8 zeigt die Erweiterung unseres Modells. Wir ergänzen Transitionen, so dass auch in den Zuständen p1, p3 und p4 die Marken von QUIT und v konsumiert werden können.



Transition t_i	$\bullet t_i$	$t_i \bullet$
t14	RSET, p2	m250, p2
t15	RSET, p3	m250, p2
t16	RSET, p4	m250, p2
t3	HELO, p1	m250, p2
t17	HELO, p2	m250, p2
t18	HELO, p3	m250, p2
t19	HELO, p4	m250, p2
t2	EHLO, p1	m250, p2
t20	EHLO, p2	m250, p2
t21	EHLO, p3	m250, p2
t22	EHLO, p4	m250, p2

Abbildung 3.7: Das Reset-Kommando des SMTP. Dieses Teilnetz wird für alle Plätze $p_n, n \in \{2, 3, 4\}$ und die Transitionen $t_i, i \in \{2, 3, 14, ..., 22\}$ gemäß der Tabelle in das Modell aus Abbildung 3.3 eingefügt.

Damit geht das Netz in den Endzustand über, der durch eine Marke auf dem Platz p6 erreicht wird. Durch das Konsumieren der Marken auf v können in unserem Modell auch keine vom Zustand des Servers unabhängigen Nachrichten (NOOP, HELP, VRFY, EXPN) mehr empfangen werden.



Transition t_i	$\bullet t_i$	$t_i \bullet$
t23	p1, v, Quit	m250, p6
t9	p2, v, Quit	m250, p6
t24	p3, v, Quit	m250, p6
t25	p4, v, Quit	m250, p6

Abbildung 3.8: Das Quit-Kommando des SMTP. Dieses Teilnetz wird für alle Plätze $p_n, n \in \{1, 2, 3, 4\}$ gemäß der Tabelle in das Modell aus Abbildung 3.3 eingefügt.

3.2.5 Fehlercodes

Bisher haben wir in unserem Modell fast ausschliesslich positive Rückmeldungen abgebildet. Die einzige Ausnahme ist m550 in Abbildung 3.6. In Tabelle 3.1 sind die festgelegten und vorgeschlagenen Antwortcodes zusammengestellt. Mit jedem Eintrag eines Codes in der Tabelle korrespondiert eine Transition je möglichem Zustand in unserem Modell, die den jeweiligen Code sendet. So kann der Code m250 etwa als Antwort auf die beiden Hello-Kommandos in vier Zuständen gesendet werden. Dazu sind acht Transitionen erforderlich. Wollten wir die Fehlerfälle (m504 und m550) dieser Kommandos ebenfalls abbilden, so wären 24 Transitionen erforderlich. Für alle Kombinationen der Kommandos und Codes erhielten wir 88 Transitionen, die ein Sendeereignis auslösen. Im Modell sind 22 dieser Transitionen modelliert. Die übrigen ebenfalls abzubilden hätte keine Vorteile, da sie die Komplexität der Kommunikation nicht erhöhen würden. Die zusätzlichen Transitionen würden ohnehin nicht deterministisch ausgewählt, da sie von konkreten internen Zuständen des Servers zur Laufzeit abhängen, von denen unser Modell abstrahiert.

Kommando	Statuscode	
	Erfolg	Fehler
Connect	**220**	554
Hello, Extended Hello	**250**	504, 550
Mail	**250**	552, 451, 452, 550, 553, 503
Recipient	**250**, 251	450, 451, 452, 503, 550, 551, 552, 553
Data	**354**	451, 554, 503
Maildata	**250**	552, 554, 451, 452
Reset	**250**	
Verify	**250**, 251, 252	**550**, 551, 553, 502, 504
Expand	**250**, 252	550, 500, 502, 504
Help	211, **214**	502, 504
Noop	**250**	
Quit	**221**	

Tabelle 3.1: Statuscodes des SMTPs. **Fett** gedruckte Codes sind in unserem Modell berücksichtigt.

3.3 Analyse des SMTP-Modells

In diesem Abschnitt analysieren wir das bisher entwickelte Modell des SMTPs mit Hilfe von FIONA. Die Größe des Interaktionsgraphen (IG) und der Bedienungsanleitung (OG) sind in Tabelle 3.2 dargestellt.

Modus	Knoten		Kanten		Zustände	Zustände in Knoten	
IG	339	(181)	887	(493)	737	927	(366)
OG	5.465	(5.464)	1.515.448	(29.686)	107.738	374.760	(9.668)

Tabelle 3.2: Statistiken des SMTP-Modells. Netz mit 25 Plätzen (davon 13 Input- und 4 Output-Plätze) und 25 Transitionen. In Klammern stehen jeweils die „blauen" Knoten, Kanten und Zustände in Knoten.

3.3.1 Benchmarks

Für die Benchmarks erweitern wir das Modell — analog zum Modell des HTTPs — indem wir einen Platz c einfügen. Dieser Platz wird als Zähler (counter) dienen. Ausserdem fügen wir die Kanten ($c \to t_2, c \to t_3, c \to t_4, c \to t_6, c \to t_{10}, c \to$

$t_{11}, c \rightarrow t_{12}$), sowie entsprechende Kanten für alle Transitionen, die Reset-Kommandos ausführen, ein. Die Anzahl der Marken auf c hat direkten Einfluss auf die Anzahl und Komplexität der möglichen Interaktionen.

Für unterschiedliche Anzahl Marken auf c ergeben sich folgende Statistiken bei der Erstellung der IG und OG.

counter	Knoten		Kanten		Zustände	Zustände in Knoten	
1	149	(79)	294	(164)	279	367	(146)
2	547	(203)	1.200	(488)	1.199	1.575	(396)
3	879	(353)	2.055	(903)	1.862	2.463	(698)
4	1.211	(503)	2.910	(1.318)	2.545	3.351	(1.000)
5	1.543	(653)	3.765	(1.733)	3.143	4.239	(1.302)
10	3.203	(1.403)	8.040	(3.808)	6.457	8.679	(2.812)
20	6.523	(2.903)	16.590	(7.958)	12.936	17.559	(5.832)

Tabelle 3.3: IG Statistiken des SMTP-Modells. Netz mit 26 Plätzen (davon 13 Input- und 4 Output-Plätze) und 25 Transitionen. In Klammern stehen jeweils die „blauen" Knoten, Kanten und Zustände in Knoten.

Bei der Komplexität der Berechnung der Bedienungsanleitungen ist ein sehr starkes Wachstum der Knoten- und Kantenzahlen zu erkennen. In Tabelle 3.4 sind die Statistiken für die ersten sechs Iterationen zusammengefasst.

counter	Knoten		Kanten		Zustände	Zustände in Knoten	
1	1.460.521	(124)	8.605.522	(644)	360.971	5.486.412	(210)
2	3.984.305	(388)	24.376.008	(2.074)	607.832	16.286.308	(708)
3	4.872.817	(750)	30.220.707	(4.058)	728.377	19.779.900	(1.402)
4	5.113.665	(1.130)	31.917.241	(6.150)	819.810	20.214.788	(2.132)
5	5.354.513	(1.510)	33.613.775	(8.242)	912.017	20.649.676	(2.862)
6	5.595.361	(1.890)	35.310.309	(10.334)	1.003.848	21.084.564	(3.592)

Tabelle 3.4: OG Statistiken des SMTP-Modells. Netz mit 26 Plätzen (davon 13 Input- und 4 Output-Plätze) und 25 Transitionen. In Klammern stehen jeweils die „blauen" Knoten, Kanten und Zustände in Knoten.

3.4 Fazit zur Modellierung des SMTPs

Das Simple Mail Transfer Protocol (SMTP) besteht aus einer Reihe von Kommandos, die der Organisation und Durchführung der Mail-Transaktionen dienen. Die Kom-

mandos werden von Client der Reihe nach an den Server geschickt, der diese einzeln quittiert. Kommandos, die nicht in der vorgesehenen Reihenfolge eintreffen, muss der Server nicht speichern, sondern kann sie als Fehler zurückweisen. Das SMTP arbeitet zustandsbasiert und die Kommunikation geschieht synchron. Offene Netze sind jedoch ein Mittel, welches das Modellieren asynchroner Prozesse erlaubt. Aus diesem Grund kann das SMTP-Modell nicht der Zielstellung, ein praxisnahes Beispiel für Benchmarks zu finden, gerecht werden.

4 Secure Socket Layer

Im OSI-Modell ist *Secure Socket Layer* (SSL), oder je nach Version auch TLS (*Transport Layer Security*) genannt, oberhalb der Transportschicht (TCP) und unter Anwendungsprotokollen wie HTTP oder SMTP angesiedelt. Das SSL-Protokoll gewährleistet, dass Daten während der Übertragung nicht von Dritten gelesen oder manipuliert werden können und stellt die Identität des Kommunikationspartners sicher.

In diesem Kapitel werden wir zunächst das Einsatzgebiet und den Aufbau des SSLs skizzieren. Anschliessend wird unser Modell des Handshakes vorgestellt und analysiert werden.

Das SSL-Protokoll benutzt eine Kombination aus symmetrischer und asymmetrischer (*Public Key*) Verschlüsselung. Während der Datenübertragung wird symmetrische Verschlüsselung eingesetzt, da asymmetrische erheblich rechenaufwendiger ist. Eine Herausforderung besteht darin, den gemeinsamen Schlüssel für Client und Server über ein unsicheres Netzwerk auszuhandeln. Jede Kommunikation kann potentiell „abgehört" werden. Die Partner müssen daher einen Code aushandeln, der selbst jedoch nicht über das Netzwerk übertragen wird. Zusätzlich darf ein Angreifer den Code nicht anhand der übertragenen Daten errechnen können. Dieses Aushandeln eines Codes ist die erste Phase der Kommunikation und wird als *Handshake* bezeichnet. Nach einem erfolgreichen Handshake können sich die Partner sicher sein, dass die Verbindung den oben genannten Bedingungen genügt. In Phase 2 werden anschliessend die Daten übertragen. Client und Server können jederzeit den Handshake wiederholen, falls dies erforderlich sein sollte. Nach dem Ende der Datenübertragung wird die Verbindung geschlossen. Schlägt der Handshake dagegen fehl, wird die Verbindung sofort mit einer entsprechenden Meldung beendet.

4.1 Der SSL-Handshake

Aufgabe des Handshake-Protokolls ist das Aushandeln und Festlegen der kryptographischen Parameter der Verbindung. Es werden die zu verwendenden Verschlüsselungsmethoden und Protokollversionen festgelegt. Falls erforderlich, ermöglicht es Server und Client zu authentifizieren. Es werden außerdem alle nötigen Daten ausgetauscht, um einen gemeinsamen Schlüssel (*Master Secret*) zu errechnen. Es ist auch möglich, dass der Client eine vorherige Sitzung weiterführen möchte. Wenn der Server

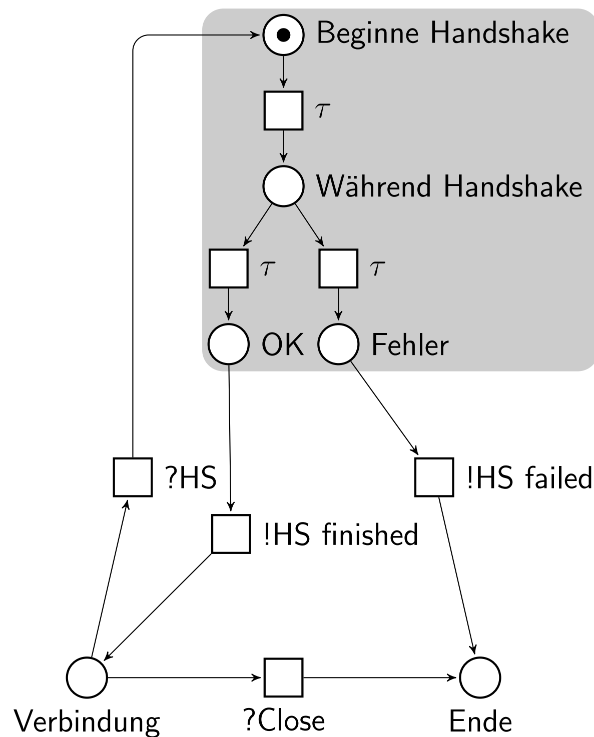


Abbildung 4.1: Zustände einer SSL-Verbindung. Die zum Handshake (HS) gehörenden Plätze und Transitionen sind grau hinterlegt.

dies – je nach Situation – erlaubt, kann der Handshake entsprechend verkürzt werden, da kein neues Master Secret errechnet werden muss.

Der Client initiiert die Verbindung. Er schickt eine Hello-Nachricht (Client Hello) an den Server. Diese Nachricht enthält die Protokoll-Version, die Session-Identifikationsnummer sowie die unterstützten Verschlüsselungs- und Kompressionsalgorithmen des Clients. Ausserdem schickt der Client eine Zufallszahl an den Server. Der Server muss diese Nachricht mit einem Server Hello beantworten oder die Verbindung wird getrennt. Der Server wählt dazu den stärksten Verschlüsselungsalgorithmus aus, den der Client unterstützt. Sollte dieser nicht sicher genug sein, wird der Handshake vom Server beendet. Des Weiteren schickt der Server seinerseits die Verschlüsselungsparameter, eine Zufallszahl sowie das eigene Zertifikat an den Client. Optional kann der Server auch ein Zertifikat vom Client verlangen (Certificate Request), falls dieser authentifiziert werden soll.

Mit den empfangenen Daten kann der Client den Server authentifizieren. Zuerst prüft der Client, ob das aktuelle Datum innerhalb der Gültigkeitsspanne des Zertifikates liegt. Ist das Zertifikat derzeit nicht gültig, wird der Handshake abgebrochen. Als nächstes muss überprüft werden, ob der ausstellenden Agentur vertraut wird. Jeder

Client führt eine Liste mit Agenturen, denen er vertraut. Ist das Zertifikat des Servers von einer dieser Agenturen oder endet die Vertrauenskette bei einer, der der Client vertraut, wird mit der Prüfung fortgefahren. Der dritte Schritt ist das Vergleichen der digitalen Signatur des Zertifikates mit der Signatur der Agentur in seiner internen Liste. So kann sichergestellt werden, dass das Zertifikat nicht verändert wurde, seit es ausgestellt wurde. Das Zertifikat ist damit akzeptabel. Im Anschluss muss der Client den Domain-Namen des Servers mit dem im Zertifikat angebenen vergleichen. So können „Man-in-the-middle"-Angriffe ausgeschlossen werden. Wenn alle Prüfungen bis hier erfolgreich sind, ist der Server authentifiziert.

Der folgende Schritt besteht darin, dass der Client aus den bisherigen Informationen das *Premaster-Secret* generiert. Dieses wird mit dem öffentlichen Schlüssel des Servers verschlüsselt und an den Server geschickt. Sollte der Server ein Zertifikat vom Client angefordert haben, schickt der Client sein Zertifikat ebenfalls an den Server.

Gegebenenfalls kann der Server den Client authentifizieren. Dies geschieht analog zur Server-Authentifizierung. Wenn keine Fehler auftreten, führen Server und Client eine Reihe von Schritten aus um (ausgehend vom gleichen Premaster Secret) unabhängig voneinander das Master Secret zu berechnen. Aus dem Master Secret wird der so genannte *Session Key* generiert. Dieser dient als Schlüssel für die symmetrische Verschlüsselung der Verbindung.

Der Client schickt dem Server eine Nachricht, dass seine zukünftige Kommunikation verschlüsselt wird (Change Cipher Spec). Anschliessend schickt er eine verschlüsselte Client Handshake finished-Nachricht. Auch der Server schickt diese beiden Nachrichten analog, womit der Handshake abgeschlossen ist.

4.2 Modell des SSL-Handshakes

Das in den Abbildungen 4.2 und 4.3 dargestellte Modell zeigt die Server-Seite des Handshake-Protokolls. Wir haben den in RFC 5246 beschriebenen Nachrichtenaustausch abgebildet. In unserem Modell abstrahieren wir vom Inhalt der Nachrichten. Wir stellen nur die Typen der Nachrichten dar. Wie bereits beim SMTP beschrieben, können auch beim SSL die Nachrichten nur in der vorgesehenen Reihenfolge empfangen werden. Ein Partner des offenen Netzes kann Nachrichten dagegen in beliebiger Reihenfolge senden. Wenn der Server in unserem Modell *kein* Zertifikat vom Client verlangt, schalten die τ-Transitionen. Entsprechend können keine Marken von Client Certificate und Client Certificate Verify konsumiert werden.

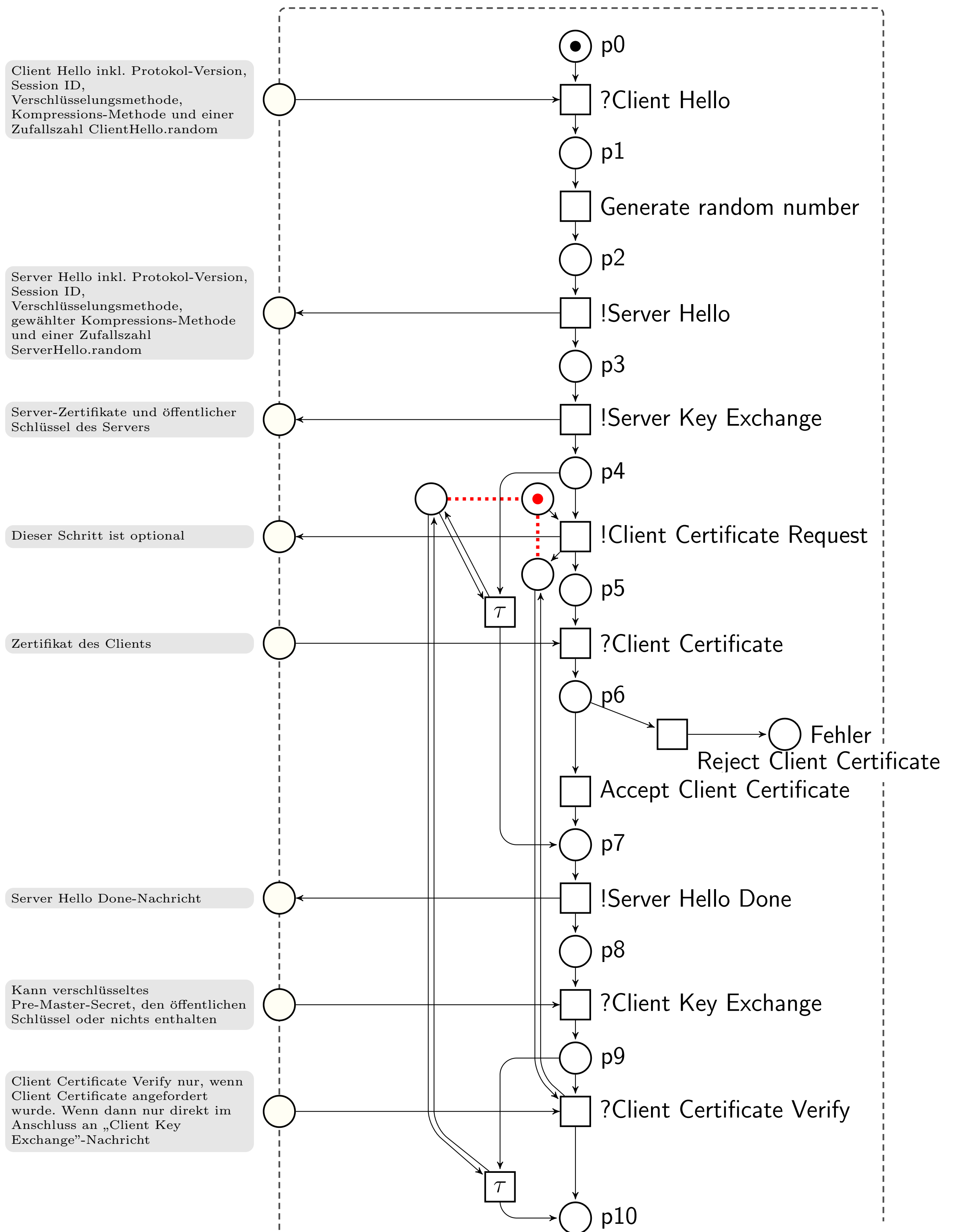


Abbildung 4.2: SSL Handshake 1.Teil. Entlang der gepunkteten Linie (····) gilt jederzeit, dass sich insgesamt genau eine Marke (•) auf den drei Plätzen befindet.

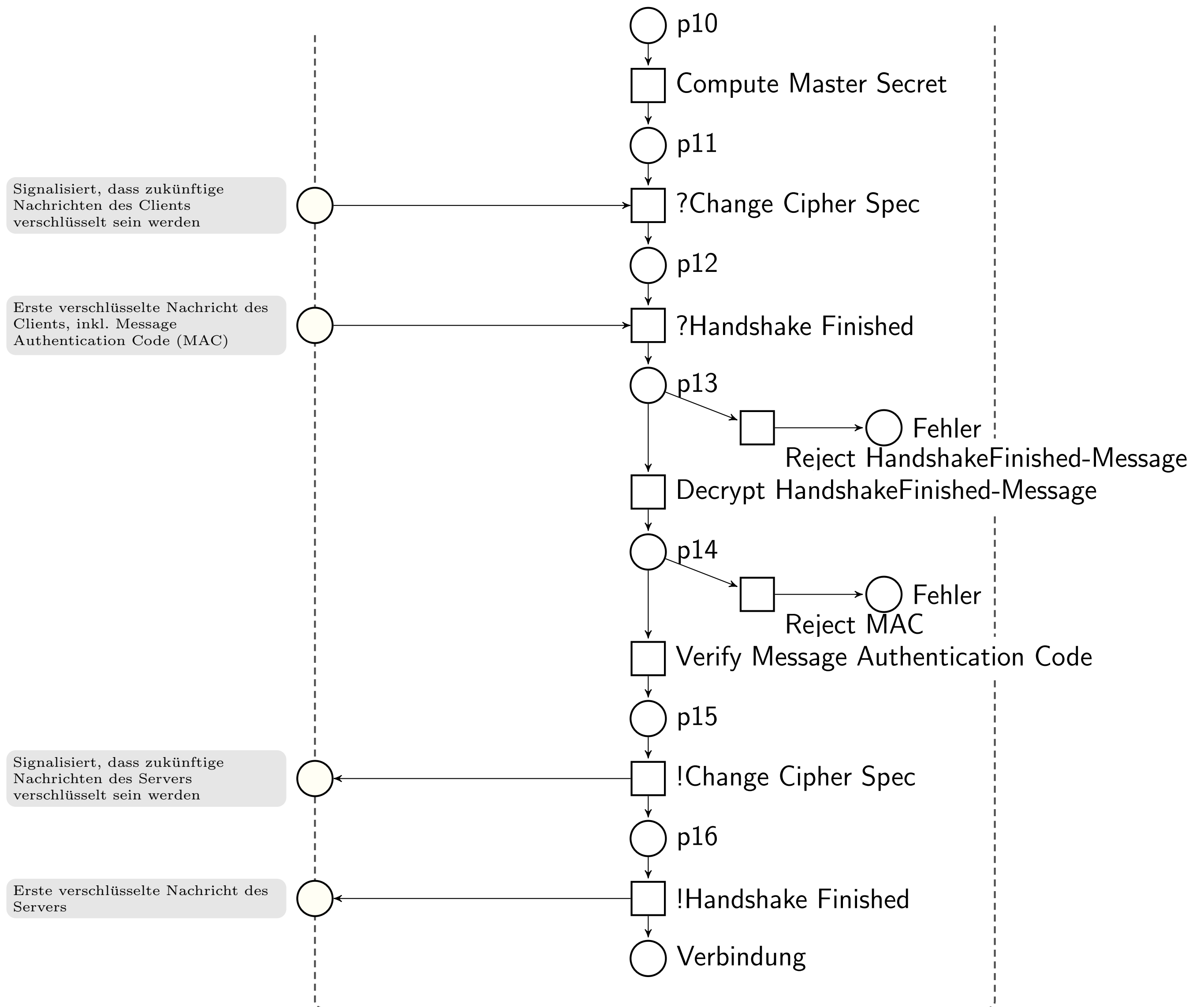


Abbildung 4.3: SSL Handshake 2.Teil. $m_0 = [\mathsf{p0}]$, $\Omega = \{[\mathsf{Verbindung}], [\mathsf{Fehler}]\}$.

4.3 Analyse des Modells des SSL-Handshakes

Im Modell des SSL-Handshakes findet — im Gegensatz zum HTTP und SMTP — viel Kommunikation statt. Dabei durchläuft das innere Netz, also das Netz ohne Interface-Plätze, linear eine Folge von Zuständen. Für die Analyse wurden die nichtdeterministischen Abläufe bei den Transitionen ?Client Certificate, ?Handshake finished und Verify MAC aufgelöst, indem je zwei Interface-Plätze (einer für gültige, einer für ungültige Daten) modelliert wurden. Ein Ablauf des Netz' endet genau dann im Zustand Fehler, wenn entsprechend ungültige Daten gesendet werden. Damit ist das Netz bedienbar. Tabelle 4.1 zeigt die Statistiken des Modells des SSL-Handshakes. Dieses Netz eignet sich ebenfalls als Grundlage für Benchmarks. Dieser ist — im Unterschied zum SMTP — nicht skalierbar.

Modus	Knoten		Kanten		Zustände	Zustände in Knoten	
IG	161	(161)	473	(473)	769	793	(793)
OG	16.385	(682)	107.448	(5.726)	36.854	31.832	(2.290)

Tabelle 4.1: Statistiken des Modells es SSL-Handshakes. Netz mit 38 Plätzen (davon 8 Input- und 6 Output-Plätze) und 24 Transitionen. In Klammern stehen jeweils die „blauen" Knoten, Kanten und Zustände in Knoten.

Auf der Suche nach weiteren Beispielen für Benchmarks wurde das Modell aus Abbildung 4.1 um die Input-Plätze ValidData, BadData, ShakeHands und Close, sowie um die Output-Plätze HsFinished und HsFailed erweitert und analysiert. Das resultierenden offene Netz ist in Abbildung 4.4 dargestellt. Die Anzahl Knoten und Kanten des Interaktionsgraphen und der Bedienungsanleitung sind in Tabelle 4.2 dargestellt.

Modus	Knoten		Kanten		Zustände	Zustände in Knoten	
IG	14	(11)	19	(14)	39	57	(38)
OG	16	(15)	145	(22)	122	162	(44)

Tabelle 4.2: Statistiken des SSL-Modells aus Abb. 4.1. Netz mit 12 Plätzen (davon 4 Input- und 2 Output-Plätze) und 7 Transitionen. In Klammern stehen jeweils die „blauen" Knoten, Kanten und Zustände in Knoten.

4.4 Fazit zur Modellierung des SSL

Es wurden zwei Netze vorgestellt und analysiert. Das Netz in Abbildung 4.4 beschreibt das SSL nur sehr abstrakt, der SSL-Handshake wird praktisch nicht betrachtet. Es ermöglicht jedoch eine gute Intuition von SSL-Protokoll und lässt sich für

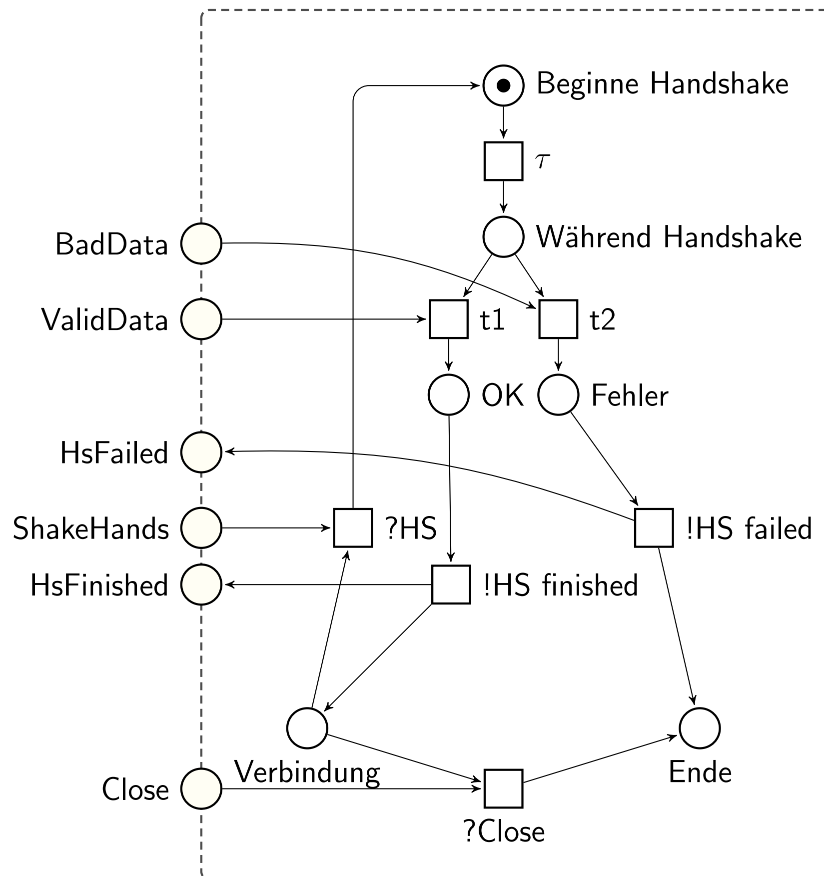


Abbildung 4.4: Zustände einer SSL-Verbindung als offenes Netz. Erweiterung des Netzes aus Abbildung 4.1 um Interface-Plätze.
$m_0 = [\mathsf{BeginneHandshake}]$, $\Omega = \{[\mathsf{Ende}]\}$.

Benchmarks benutzen. Das Netz in den Abbildungen 4.2 und 4.3 stellt den Handshake detailierter dar und lässt sich ebenfalls gut als Grundlage für Benchmarks nutzen. Die Benchmarks beider Modelle sind — entgegen der Aufgabenstellung — nicht skalierbar.

5 Verwandte Arbeiten und Fazit

5.1 Verwandte Arbeiten

In anderen Arbeiten wurden Kommunikationsprotokolle durch High-Level-Petrinetze modelliert. C. Lakos, J. Lamp, C. Keen und B. Marriott erweitern Petrinetze um einen objekt-orientierten Datentyp[LLKM95]. In ihrer Arbeit modellieren sie zur Demonstration der entstehenden *Objekt-Petrinetze* Kommunikationsprotokolle beispielhaft, analysieren diese jedoch nicht weiter. Figueiredo und Kristensen analysieren und simulieren das Transfer Control Protocol (TCP) mit Hilfe von gefärbten Petrinetzen[FK99]. Sie stellen ein hierarchisches Modell vor, mit dem der Datenfluss modelliert wird und anhand dessen spezielle Szenarien berechnet werden. Diesen Ansätzen ist gemeinsam, dass sie High-Level-Petrinetze verwenden um die Daten zu modellieren. In der vorliegenden Arbeit haben wir dagegen vom Inhalt der Nachrichten weitest gehend abstrahiert und Low-Level-Petrinetze der Modelle erstellt.

5.2 Fazit

Die Kommunikationsprotokolle konnten nicht zufrieden stellend modelliert werden. Der Hauptgrund liegt darin, dass bei den Kommunikationsprotokollen die Daten eine zentrale Rolle spielen. Stellen wir diese Daten in ihren High-Level-Notation dar, können wir das Modell nicht mit Fiona analysieren. Fiona ist auf die Analyse der Struktur von Prozessen und deren Abläufe (im Unterschied zu den Daten) spezialisiert und arbeitet mit Low-Level-Petrinetzen. Modellieren wir die Daten dagegen durch unterschiedliche Kommunikationskanäle, wachsen die Modelle exponentiell zur Größe der modellierten Kanäle. Den Widerspruch zwischen der Modellierung der Kommunikationsprotokolle — genauer gesagt: der Kommunikation — und dem Abstrahieren von konkreten Nachrichteninhalten konnten wir nicht auflösen.

Literaturverzeichnis

[Fie99] R. Fielding. RFC 2616 - Hypertext Transfer Protokoll/1.1, Juni 1999.

[FK99] Jorge C. A. De Figueiredo and Lars M. Kristensen. Using Coloured Petri nets to investigate behavioural and performance issues of TCP protocols. In *Department of Computer Science, Aarhus University*, pages 21–40, 1999.

[Fre96] N. Freed. RFC 2045 - Multipurpose Internet Mail Extensions, November 1996.

[Inf81] Information Sciences Institute USC. RFC 793 - Transmission Control Protocol, September 1981.

[Jen95] Kurt Jensen. *Coloured Petri nets*. Springer-Verlag, 1995.

[Kle01] J. Klensin. RFC 2821 - Simple Mail Transfer Protocoll, April 2001.

[LLKM95] Charles Lakos, John Lamp, Chris Keen, and Brian Marriott. Modelling Network Protocols with Object Petri Nets. In *Proceedings of Workshop on Petri Nets Applied to Protocols*, pages 31–42. Springer-Verlag, 1995.

[LMSW06] Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing Interacting BPEL Processes. In *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, volume 4102 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, September 2006.

[LMW07] Niels Lohmann, Peter Massuthe, and Karsten Wolf. Operating guidelines for finite-state services. In Jetty Kleijn and Alex Yakovlev, editors, *28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings*, volume 4546 of *Lecture Notes in Computer Science*, pages 321–341. Springer-Verlag, 2007.

[MRS05] Peter Massuthe, Wolfgang Reisig, and Karsten Schmidt. An Operating Guideline Approach to the SOA. *Annals of Mathematics, Computing & Teleinformatics*, 1(3):35–43, 2005.

[Rei98] Wolfgang Reisig. *Elements Of Distributed Algorithms: Modeling and Analysis with Petri Nets.* Springer-Verlag, September 1998.

[Tan03] Andrew S Tannembaum. *Computernetzwerke.* Pearson Studium, 4. überarbeitete Auflage edition, 2003.

[Wei08] Daniela Weinberg. Efficient controllability analysis of open nets. In Roberto Bruni and Karsten Wolf, editors, *Web Services and Formal Methods, Fifth International Workshop, WS-FM 2008, Milan, Italy, September 4–5, 2008, Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, September 2008.