INSim
Java-basierte prototypische Implementierung eines IN-Switches mit Endgeräten

Autor: Vitalian A. Danciu
Betreuer: Rainer Hauck
Aufgabensteller Prof. Dr. H.-G. Hegering

16. Juli 2001

Zusammenfassung

Das vorliegende Papier beschreibt einen im Rahmen einer Projektarbeit erstellten Prototypen eines IN-Switch. Der Prototyp soll eine Grundlage bereitstellen, die für die Entwicklung von Komponenten höherer Schichten eines IN notwendig ist.

Nach der Hervorhebung einiger relevanten Hintergründe im ersten Abschnitt wird im zweiten Abschnitt die Implementierung des Switch-Prototypen beschrieben. Der dritte Teil behandelt die Schnittstellen und bietet Vorschläge zur Erweiterung und Weiterentwicklung des Prototypen.

Inhaltsverzeichnis

1 Einführung

1.1 Einleitung

Die Bereitstellung von Mehrwertdiensten in der Telekommunikations-Branche erfordert eine geeignete Infrastruktur. Diese steht durch ein Intelligentes Netzwerk (IN) zur Verfügung.

Das Intelligente Netzwerk ist ein ständig weiter entwickeltes Konzept, das aus spezialisierten Diensten[1] entstand. Es existieren zwei Entwicklungslinien, deren Ansätze sich weitgehend decken: das AIN[2] von BellCore und das von der ITU-T spezifizierte IN.

1.1.1 Die Teile eines IN / Begriffsbestimmung

Einem IN liegt ein herkömmliches TK-Netzwerk zugrunde, bestehend unter anderem aus Vermittlungsstellen und Endgeräten. Durch Erweiterung der Fähigkeiten der bestehenden Vermittlungsknoten (CCF), sowie durch das Hinzufügen IN-spezifischer Komponenten, kann die für ein IN vorgesehene Funktionalität erreicht werden.

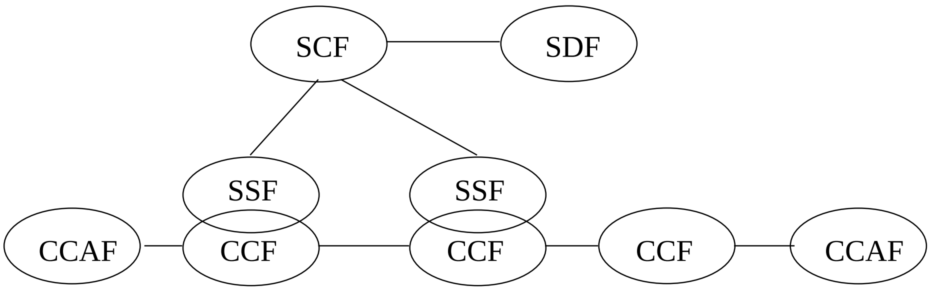

Abbildung 1: Distributed Functional Plane (DFP) des IN

Abbildung 1 zeigt folgende Teile eines IN:

- CCAF - Call Control Agent Function
- CCF - Call Control Function
- SSF - Service Switching Function
- SCF - Service Control Function
- SDF - Service Data Function

Die CCAF dient als Schnittstelle zwischen CCF und Benutzer. Die SSF dient als Schnittstelle zwischen CCF und SCF; sie verwaltet die Anforderungen der IN-Dienste seitens des Benutzers und leitet diese an die SCF weiter. Die SCF selbst stellt IN-Dienste zur Verfügung. Zu ihrer Unterstützung steht die SDF, von der Daten bezüglich der Benutzer und der Infrastruktur abgerufen werden können (vgl. Abbildung 1).

[1] 0800-Dienst und Operator Services Systems (OSS) der Bell Operating Company, vgl. [Black], S. 5

[2] Advanced Intelligent Network

1.1.2 CCF

Für einen Prototypen eines IN-Switches spielt die Call Control Function eine zentrale Rolle. Sie übernimmt in einer realen Implementierung die Aufgaben einer einfachen Vermittlungsstelle und liegt somit der Telekommunikations-Infrastruktur zugrunde, auf die ein Intelligentes Netz aufbaut. Ihre wichtigste Aufgabe ist die Herstellung von Verbindungen zwischen Endgeräten.

Um eine Call Control Function in ein IN einzubinden, ist es erforderlich, gewisse Erweiterungen in den herkömmlichen Vermittlungsverfahren zu realisieren, um eine Steuerung von aussen zu ermöglichen. Die Funktion der Verwaltung der IN-Funktionalität einer oder mehreren Call Control Function wird durch eine Service Switch Function (SSF) realisiert. Dabei werden die Aufgaben der Call Control Function, d. h. der eigentlichen Vermittlungsstelle, weiterhin einfach gehalten; dieses Vorgehen ermöglicht eine Erweiterung der Funktionalität des TK-Netzes ohne Performance-Verluste bei Vorgängen "ohne" IN.

1.2 Das BCSM

Das *Basic Call State Model* (BCSM) beschreibt die Vorgänge in einem CCF/SSF-Paar. Es gliedert sich in zwei Teile, die den originierenden bzw. den terminierenden Teil des Modells beschreiben.

Jeder der beiden Teile besitzt Zustände (PIC[3]), die von dem Switch auszuführende Aufgaben bestimmen. Das BCSM beschreibt auch die Ereignisse, die zu Zustandsänderungen führen. Zusätzlich dazu sind bei den Transitionen Kontollpunkte (*Detection Points*) vorgesehen, mittels derer eine Kontrolle des Switch durch eine SCF möglich ist.

1.2.1 O_BCSM

Das *Originating BCSM* beschreibt die Art und Weise, wie ein Verbindungsaufbau im Switch (genauer: in der CCF) angestoßen wird. Es beinhaltet sechs Zustände (PICs), die während eines Verbindungsaufbaus von dem Switch eingenommen werden können (vgl. Abbildung 2). Diese entsprechen den Aufgaben, die der Switch jeweils zu erfüllen hat.

1. `O_Null & Authorize Origination_Attempt` - dies ist der Wartezustand des Switch. Wird ein Verbindungsaufbau angefordert, kann außerdem die Berechtigung des Benutzers geprüft werden.

2. `Collect_Info` - in diesem Zustand wartet der Switch auf das Wählen einer Nummer durch den Benutzer. Das Ende der Eingabe muss seitens des Switch erkannt werden; die Art und Weise, in der dies geschieht, wird von dem BCSM absichtlich nicht spezifiziert, um Herstellern die Art der Implementierung freizustellen.

3. `Analyze_Info` - die Daten, die in Collect_Info gesammelt wurden, können in diesem Zustand überprüft werden. In diesem PIC wird zwischen lokalen, regionalen und internationalen Verbindungsanforderungen unterschieden. Gegebenenfalls (z. B. bei Auftreten einer "IN-Nummer"[4]) kann eine entsprechende Sonderbehandlung des Verbindungsaufbaus angestossen werden.

[3] Point In Call, siehe [DFPIN, S. 6]

[4] d. h. einer Nummer, durch die ein IN-basierter Dienst in Anspruch genommen wird.

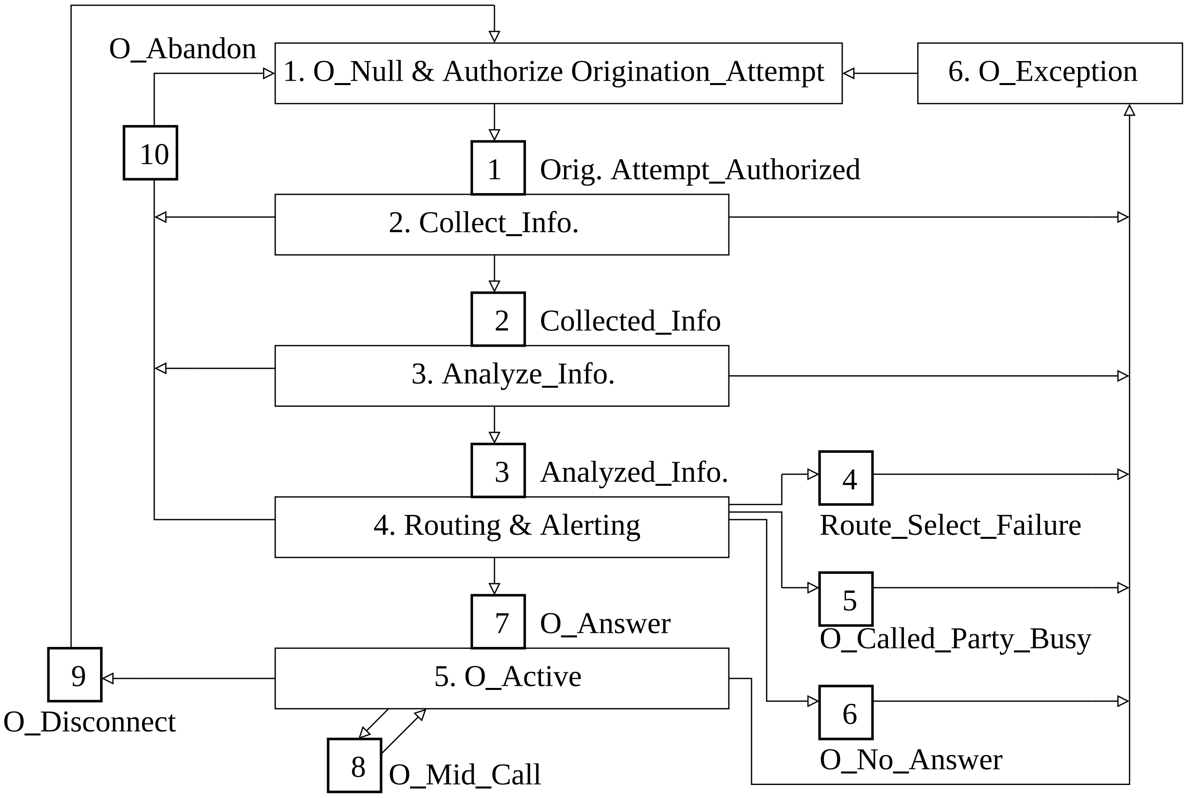

Abbildung 2: Das Originating Basic Call State Model (O_BCSM)

4. `Routing & Alerting` - der Switch versucht, eine Leitung zu vermitteln. Bei Erfolg wird die terminierende Komponente benachrichtigt. `Routing & Alerting` dient außerdem als "Wartezustand" für den Fall, dass eine Verzögerung bei der terminierenden Komponente entsteht (z. B. manuelles Annehmen der Verbindung, "Hörer abheben"). Unter Umständen ist ein Timeout möglich, falls die Verbindung nach einer gewissen Wartezeit nicht etabliert ist; in diesem Fall geht der Switch über den DP `O_No_Answer` in die Fehlerbehandlung über (`O_Exception`).

5. `O_Active` - die Verbindung ist etabliert. Optional ist diesem Zustand eine Überwachung der Verbindung möglich[5].

6. `O_Exception` - mißlingt eine Operation, geht der Switch in diesen Zustand über. Hier wird die Fehlerbehandlung entsprechend der aufgetretenen Ausnahme ausgeführt. Es wird sichergestellt, dass die in Anspruch genommenen Ressourcen dealloziert wurden; besteht eine Bezugnahme auf eine SCF, so wird diese über die aufgetretene Ausnahme informiert.

1.2.2 T_BCSM

Das *Terminating BCSM* beschreibt die Vorgänge, die bezüglich eines terminierenden Endgerätes von dem Switch durchgeführt werden.

[5] Erkennung von z. B. DTMF

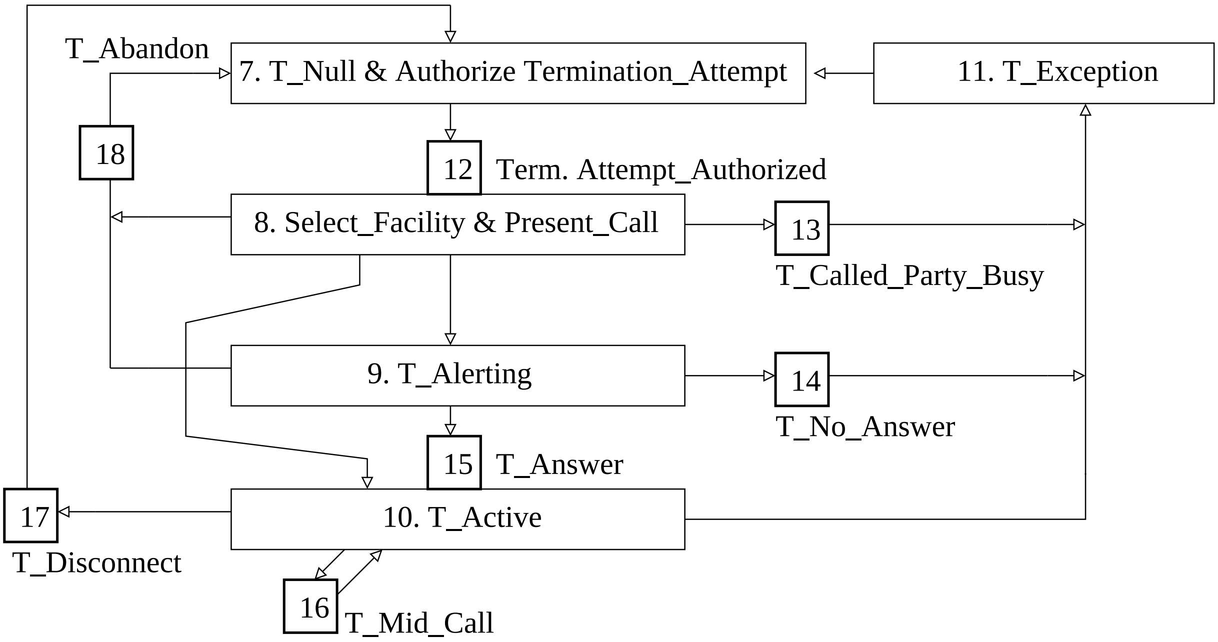

Abbildung 3: Das Terminating Basic Call State Model (T_BCSM)

Die Zustände des T_BCSM entsprechen hauptsächlich denen seines originierenden Pendants. Die Transitionen im T_BCSM werden des öfteren von der originierenden Hälfte angestoßen. Hat etwa der Switch im PIC `Routing & Alerting` (O_BCSM) einen Weg zum terminierenden Endgerät gefunden, so erfährt er im T_BCSM die Transition zu `Select_Facility & Present_Call` (vgl. Abbildung 3). In diesem Zustand wird das Endgerät über den Verbindungswunsch informiert. Kann das Endgerät ohne das Zutun des Benutzers die Verbindung nicht akzeptieren, so wartet der Switch im Zustand `T_Alerting` bis der Benutzer die Verbindung akzeptiert oder aber ein Timeout erfolgt. Der Zustand `T_Active` ist analog zu `O_Active` im O_BCSM; ebenso entsprechen sich die Startzustände `O_Null` und `T_Null`.

1.3 IN-spezifische Aspekte des BCSM

Um dem Benutzer IN-basierte Dienste zur Verfügung stellen zu können, ist es erforderlich, die Kontrolle des Switch durch die Service Control Function zu ermöglichen. Die im BCSM beschriebenen *Detection Points* stellen ein Mittel zur Verfügung, um diese Kontrolle zu realisieren.

1.3.1 Die Service Control Function

Die Service Control Function stellt Dienste bereit, die von dem Benutzer implizit oder ausdrücklich angefordert werden können[6]. Im Laufe der Herstellung einer Verbindung kann die Vermittlungsstelle (CCF) auf eine Situation stossen, die eine Anfrage an die SCF

[6] *Beispiel*: die Weiterleitung eines Anrufs. Der Angerufene hat den Dienst ausdrücklich bestellt und bezahlt; aus Sicht des IN ist ein Anrufer jedoch auch ein Benutzer, der den Dienst implizit benutzt.

erfordert. Alternativ können Trigger-Bedingungen formuliert werden, deren Erfüllung zu einer Anfrage an die SCF führt.

1.3.2 Detection Points und Trigger

Die im BCSM beschriebenen *Detection Points* (DP) verfügen über vier grundlegende Eigenschaften:

1. sie können aktiviert bzw. deaktiviert werden
2. ihnen können Trigger-Bedingungen zugeordnet werden
3. sie können eine Beziehung zu einem von der SCF kontrollierten Verfahren haben
4. sie sind in der Lage, den Verbindungsaufbau zu unterbrechen

Beim Überqueren eines DP werden die Trigger-Bedingungen ausgewertet und gegebenfalls Aktionen ausgeführt (z. B. können Nachrichten generiert und an entsprechende IN-Komponenten verschickt werden).

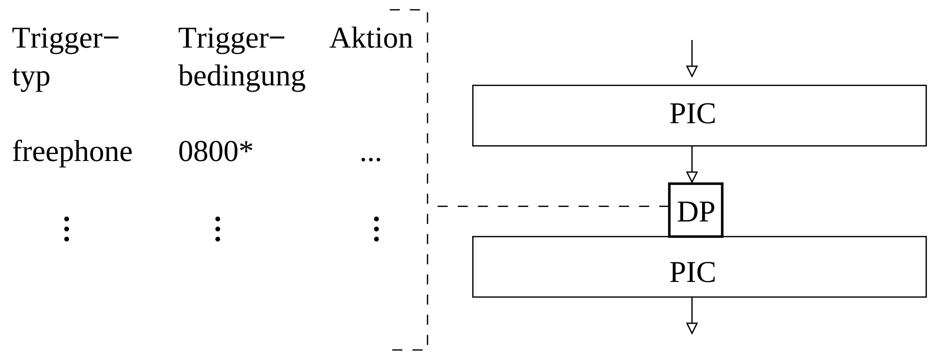

Abbildung 4: Trigger eines Detection Points

1.3.3 SIB-basierte Dienste

Eins der Ziele eines IN ist es, einem Betreiber die Möglichkeit zu bieten, Dienste schnell entwickeln und kostengünstig zur Verfügung stellen zu können. SIBs (Service Independent Building Blocks) bieten die Möglichkeit, Dienste schnell zu entwickeln, indem sie als Module grundlegende Funktionen anbieten. Aus den passenden SIBs kann mit wenig zusätzlichem Aufwand ein Dienst erstellt werden.

Beispiele für SIBs sind (vgl. [Black], S. 39f):

- *Algorithm SIB* - wendet einfache Operationen auf die Daten an
- *Authenticate SIB* - ermöglicht Authentifizierung eines Benutzers mittels eines Kennung-Passwort-Paares und bietet gegebenfalls auch Unterstütung für Verschlüsselung
- *Charge SIB* - bestimmt die Art und Weise, in der ein Anruf abgerechnet wird.
- *Compare SIB* - vergleicht einen gegebenen Wert mit einem Referenzwert.

- *Queue SIB* - dieser SIB stellt eingehende Aufrufe in eine Warteschlange
- *Service Data Management SIB* - verwaltet Benutzerdaten, evtl. mittels einer Datenbank
- *Screen SIB* - vergleicht einen gegebenen Wert (z. B. einen PIN) mit Werten aus einer Liste (screen list); unter Umständen entscheidet der SIB anhand dieses Vergleichs, ob eine Verbindung hergestellt wird oder nicht.
- *Status Notification SIB* - überwacht Netzwerk-Ressourcen
- *User-interaction SIB* - übernimmt die Kommunikation mit (originierenden oder terminierenden) Ressourcen, die einem Benutzer zugeordnet sind.

Aus solchen SIB können Dienste erstellt werden, wie zum Beispiel (vgl. [Black], S. 161f):

- Weiterleitung eines Anrufs (*follow me*): Weiterleitung von Anrufen (evtl. zeitabhängig) zu unterschiedlichen Anschlüssen (Büro, zu Hause, etc). Möglich ist so auch die Filterung von Anrufern, so dass nur "erwünschte" Anrufe weiter geleitet werden.
- Zeitabhängiges Routing (*time-dependent routing*): beispielsweise für 0800-Nummern
- Notfall-Routing (*emergency routing*): Umleitung der Anrufe, falls z. B. eine Hotline technische Probleme erfährt. Für den Anrufer geschieht dies transparent.
- OCM (*outgoing call management*): mittels dieses Dienstes kann der Kunde selbst entscheiden, welche Typen von Anrufen von seinem Anschluss aus getätigt werden können. So können Mehrwertdienste wie 0900/0190 oder Fern- und Auslandsgespräche ausgeschlossen werden. "Teure" Anrufe können mittels einer PIN getätigt werden. Mittels Benutzung von mehreren, unterschiedlichen Benutzern zugeordneten PIN können so freigeschaltete Anrufe einzelnen Benutzern zugeordnet werden.

Solche Dienste können mit Hilfe von Frameworks erstellt werden, die kommerziell verfügbar sind.

1.4 Problemstellung

Das Ziel dieser Projektarbeit ist die prototypische Implementierung eines IN-fähigen Switch, der die grundlegenden Funktionen eines CCF/SSF-Paares zur Verfügung stellen soll, sowie die Erstellung von Endgeräten, mittels derer der Switch angesprochen werden kann. Zur Beobachtung und Bedienung von Switch und Endgeräten soll eine graphische Benutzeroberfläche zur Verfügung stehen. Als Plattform/Programmiersprache für die Implementierung ist Java vorgegeben.

1.4.1 In der Projektarbeit behandelte Teile

Um die grundlegende Funktionalität eines Switch in Software umzusetzen genügt es, einige wenige Teile eines IN zu betrachten. Da Effizienz und Skalierbarkeit für diesen Simulator eine untergeordnete Rolle spielen, kann zusätzlich die Funktionalität mancher Komponenten zusammengefasst werden. Abbildung 5 zeigt den Ansatz, der für die Implementierung des Prototypen benutzt wurde.

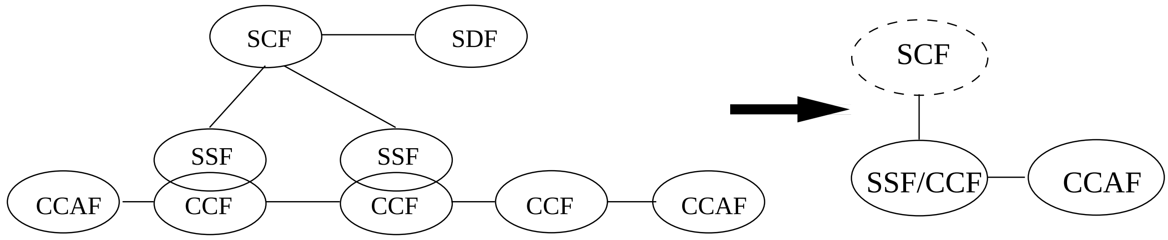

Abbildung 5: Vereinfachung des DSP

Die vorliegende Projektarbeit implementiert einige in einem IN zentralen Komponenten nur schematisch. Deshalb muss gewährleistet sein, dass für spätere Erweiterungen Schnittstellen geboten werden, um realistischere Implementierungen dieser Komponenten zu unterstützen.

1.4.2 Monitoring und Management eines SIB-basierten Systems

Auch wenn mittels SIBs Mechanismen bereitgestellt werden können, um Netzwerkressourcen zu überwachen, fehlt es an Mechanismen, um die Abläufe in den SIBs selbst (und in Aggregationen mehrerer SIBs) auf Effizienz, Fehlertoleranz und Zweckmässigkeit zu prüfen. Auf diese Weise könnten redundante und fehlerhafte Abläufe ausgeschlossen werden und die in der SCF verfügbaren Ressourcen verwaltet werden. Als weitere Hilfestellung beim Aufbau von auf SIBs basierenden Diensten wäre es also sinnvoll, in den SIBs selbst Vorkehrungen zu treffen, um Überwachung und Fehlerbeseitigung zu ermöglichen.

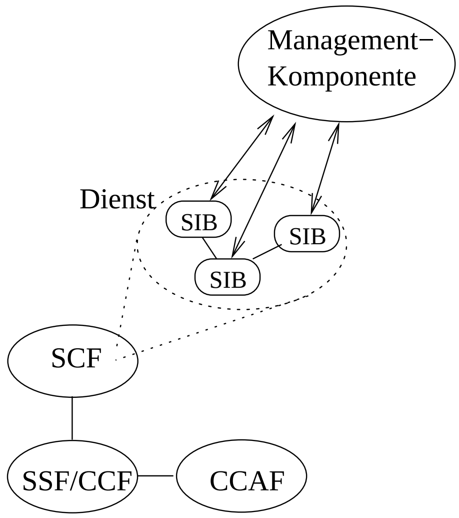

Abbildung 6: Mögliche Architektur eines IN-Simulators mit Management-Komponente

Die von den einzelnen Komponenten generierten Überwachungsdaten könnten dann einem Managementsystem zugeführt werden (vgl. Abbildung 6), das automatisch oder auch durch Eingriffe eines Administrators Abläufe verfolgen könnte.

2 Implementierung

Der Prototyp wurde als Java *application* implementiert. Dabei wurde auf Erweiterbarkeit und Schnittstellen zu weiteren Komponenten Wert gelegt. Die Visualisierung der Vorgänge im Switch und in den Endgeräten wurde mittels Java interfaces vom eigentlichen Switch-Kern getrennt. Durch den Einsatz geeigneter Design Patterns (z. B. *Proxy*, siehe [Gamma]) wurde die Option offen gehalten, die Komponenten des Systems mittels einer Verteilungsplattform (z. B. CORBA oder Java RMI) zu verteilen.

2.1 Übersicht

Die Implementierung des Switch besteht einerseits aus zentralen Komponenten, für die jeweils nur eine Instanz in jedem Switch erzeugt wird. Erwähnenswert sind hier Klassen wie `CallControlFunction`, die aufgrund ihrer Verwaltungsfunktion gegenüber den Endpoints eine Rolle spielt. Auch die zu Demonstrationszwecken erstellte Implementierung des Interfaces `ServiceControlFunction`, `SimplisticSCF`, gehört zu dieser Gruppe von Klassen.

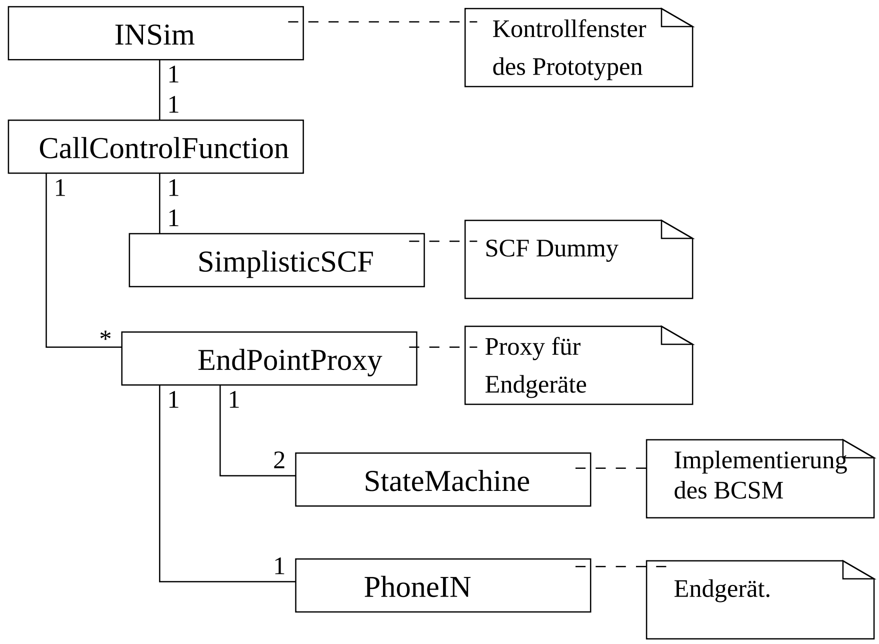

Abbildung 7: Wichtige Komponenten des Switch-Prototypen

Andererseits existieren (meist `Endpoint`-bezogene) Komponenten, die je nach Bedarf instantiiert werden. Für jeden `Endpoint` (d. h. für jeden "Teilnehmer") werden eine Reihe von Objekten erstellt, die diesen `Endpoint` repräsentieren oder seine Funktion realisieren. Hier seien die Klassen `O_StateMachine` und `T_StateMachine` genannt, die das Verhalten des Switch mit Bezug auf einen bestimmten Endpoint bestimmen, sowie `EndPointProxy`, der Switch-seitig einen `Endpoint` repräsentiert und Nachrichten von und an diesen weiter leitet (vgl. Abbildung 7).

2.2 Implementierung des Switch-Kerns

2.2.1 Die Implementierung von O_BCSM und T_BCSM

Das BCSM ist mittels einer Logik ähnlich der für Zustandsautomaten realisiert. Dabei faßt die abstrakte Klasse `StateMachine` die vom originierenden/terminierenden Charakter unabhängige Funktionalität zusammen. Die davon abgeleiteten Klassen implementieren die jeweils spezifischen Aspekte von O_BCSM bzw. T_BCSM (vgl. Abbildung 8).

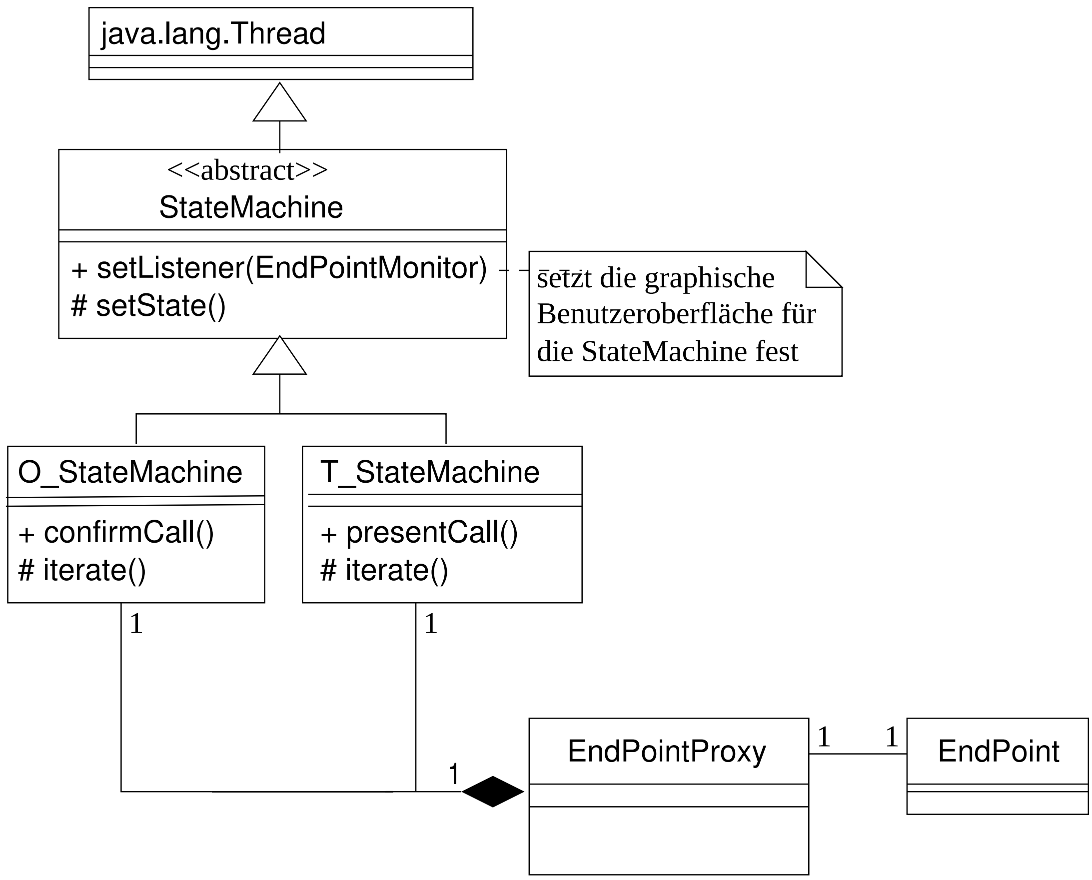

Abbildung 8: Vererbung zur Erstellung der spezialisierten Klassen `O_StateMachine` und `T_StateMachine`.

Jedem Endgerät entsprechen jeweils eine Instanz von `O_StateMachine` beziehungsweise `T_StateMachine`. Ein Verbindungsaufbau zwischen zwei Endgeräten wird von der `O_StateMachine`-Instanz des originierenden Endgerätes und der `T_StateMachine` des terminierenden Endgerätes durchgeführt. Die Zustände einer `StateMachine` entsprechen den im BCSM beschriebenen PICs.

Um die Endgeräte zeitlich unabhängig verwalten zu können, ist es erforderlich, die Instanzen von O_ bzw. T_StateMachine in verschiedenen Threads zu implementieren. Die zentrale Methode dieser Klassen ist `boolean iterate()`. Diese Methode entscheidet anhand der Werte von Instanzvariablen welcher PIC gegenwärtig der aktuelle ist; `iterate()` wird erstmals nach der Initialisierung der Objekte von der `run()`-Methode des Threads aufgerufen und löst die Aktionen des Switch in einem gegebenen PIC aus. Nach Rückkehr eines Aufrufs von `iterate()` wird anhand des Rückgabewertes entschieden, ob die Ausführung des Threads unterbrochen wird[7] oder ob `iterate()` sofort neu aufgerufen wird. Letzteres ist insbesondere zutreffend, wenn mehrere Transitionen in Folge durchgeführt werden sollen.[8] Wird eine Eingabe seitens des Benutzers oder eine Nachricht von einem anderen Objekt erwartet, dann bleibt der Thread in schlafendem Zustand bis ein Ereignis eintritt. Die Wahl dieser Vorgehensweise (im Gegensatz zum Methodenpaar `wait() / notify()`, das Java-Threads zur Verfügung steht) begründet sich darin, dass auf diese Weise eine schlafende `StateMachine`-Instanz von beliebigen Threads geweckt werden kann.

Eine weitere wichtige Methode der BCSM-Implementierung ist `void setState()`. Sie wird bei jedem Zustandsübergang aufgerufen und führt den Zustandsübergang durch (aktualisieren der betroffenen Instanzvariablen). Zusätzlich findet hier die Erkennung von aktiven statischen Triggern statt, die einen unwillkürlichen Aufruf an die SCF zur Folge haben[9]. Diese Trigger können durch den Benutzer jederzeit aktiviert oder deaktiviert werden. Gegebenenfalls wird die SCF-Anfrage ausgeführt und die Antwort ausgewertet. Die SCF-Anfrage geschieht als Methodenaufruf von dem betroffenen `StateMachine`-Thread aus.

Primärszenario: Ein Verbindungsaufbau und -abbau (ohne IN-Intervention) könnte im einfachsten Fall folgendermassen ablaufen (vgl. Abbildung 9):

1. Endgerät **a** signalisiert Anforderung zum Verbindungsaufbau ("Hörer wird abgehoben"). Die zum Endgerät gehörende `O_StateMachine` geht aus dem PIC `O_Null` in den PIC `O_Collect` über und wartet auf Eingabe seitens des Endgerätes.

2. Endgerät **a** schickt eine Zeichenkette von ein oder mehreren Ziffern ("Benutzer wählt"). Die `O_StateMachine` überprüft, ob die Zeichenkette eine gültige Nummer darstellt. Ist dies nicht der Fall, so wird überprüft, ob durch Anhängen von Ziffern eine gültige Nummer gebildet werden kann. Ist dies möglich, wird auf eine weitere Eingabe gewartet.

3. Ist eine gültige Nummer gewählt worden, geht die `O_StateMachine` in den PIC `O_Analyze`, wo eine Erkennung von "besonderen" Nummern geschehen könnte. Danach wird im PIC `Routing & Alerting` eine Referenz auf die `T_StateMachine` des designierten terminierenden Endgerätes **b** erfragt.

4. Ist die Referenz gültig, so wird die `presentCall()`-Methode der `T_StateMachine` von **b** aufgerufen. Als Parameter wird eine Referenz auf die aufrufende `O_StateMachine` übergeben. Der Rückgabewert von `presentCall()` bestimmt, ob die Anforderung zur Terminierung der Verbindung akzeptiert oder abgelehnt wurde (in der Praxis: ob sich die `T_StateMachine` zum Zeitpunkt des Aufrufes in ihrem `T_Null` befand oder

[7] java.lang.Thread.suspend()

[8] Beispiel: Bei Auftreten einer Ausnahme geschieht ein Übergang von dem PIC, in dem die Ausnahme aufgetreten ist zum Exception-PIC und dann (nach Ausführen der Fehlerbehandlung) nach Null. Dies geschieht ohne Unterbrechung in zwei aufeinanderfolgenden Aufrufen von `iterate()`. Das erste Mal liefert `iterate()` *false* zurück, das zweite mal *true*.

[9] Soll der Switch mit einer "echten" SCF benutzt werden, könnte es angemessen sein, diese Funktionalität in eine eigene Klasse auszulagern, um eine Verwaltung statischer und dynamischer Trigger für die einzelnen Endpoints zu ermöglichen.

nicht). Die `T_StateMachine` geht falls möglich in den Zustand `Select_Facility & Present_Call`.

5. Wurde die Anforderung von der `T_StateMachine` akzeptiert, wird dem Endgerät dies mitgeteilt ("Rufton"). Gleichzeitig geht die `T_StateMachine` in den PIC `T_Alerting` und weist Endgerät **b** an, seinen Benutzer zu benachrichtigen ("Klingelton").

6. Endgerät **b** terminiert den Anruf ("Hörer wird abgehoben"); seine `T_StateMachine` ruft die `confirmCall()`-Methode der `O_StateMachine` von Endgerät **a** auf. Beide StateMachines betreten ihren `Active`-PIC. Beiden Endgeräten wird der erfolgreiche Verbindungsaufbau mitgeteilt.

7. Endgerät **a** beendet die Verbindung ("Hörer wird aufgelegt"). Seine `O_StateMachine` ruft die `disconnect()`-Methode der korrespondierenden `T_StateMachine` von **b** auf; diese teilt ihrem Endgerät den Abbau der Verbindung mit. Beide StateMachines gehen in ihren `Null`-PIC zurück.

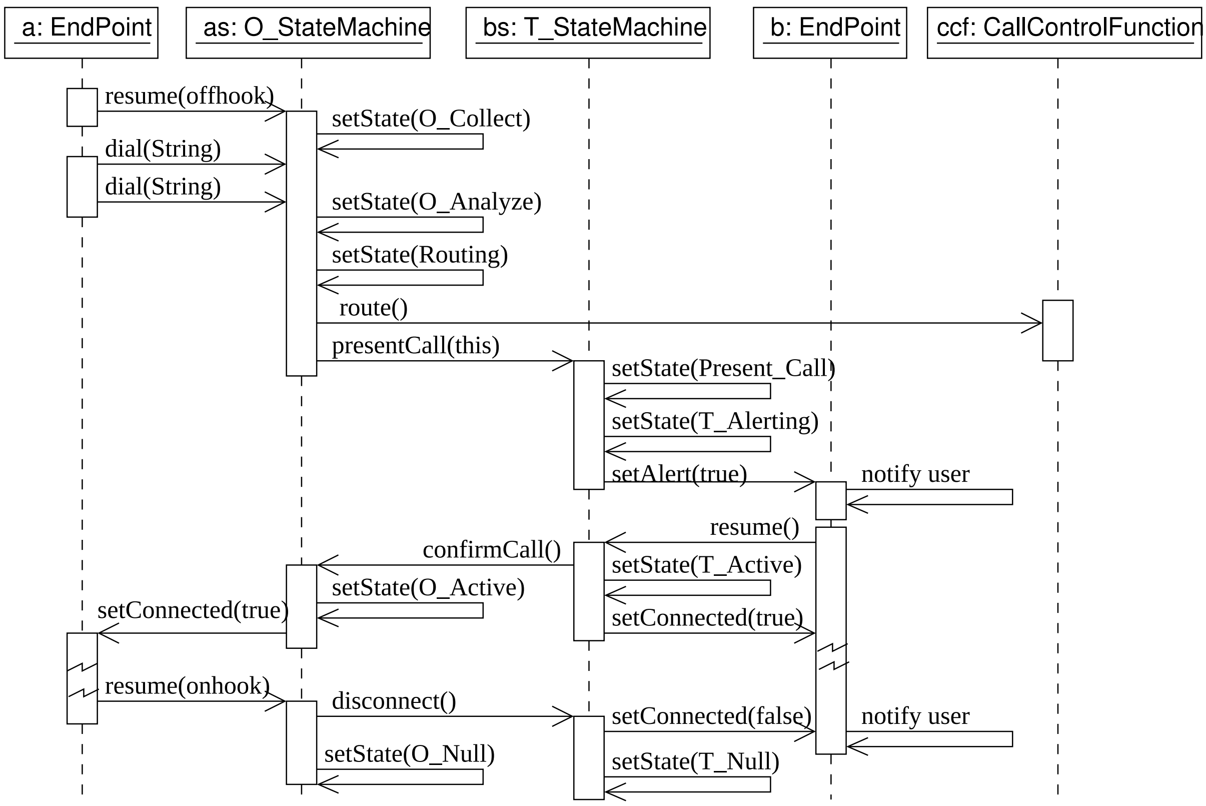

Abbildung 9: Verbindungsaufbau: Primärszenario

2.3 Die Endgeräte

2.3.1 Die Implementierung von EndPoint

Ausgehend von den grundlegenden Funktionen eines einfachen Telefons wurde ein Java interface mit dem Namen `EndPoint` erstellt, das Methoden vorschreibt, mittels denen ein Switch ein Endgerät ansteuern kann. Eine Implementierung dieses Interfaces stellt die Klasse `PhoneIN` dar. Hierbei handelt es sich um ein *appletcation*[10] mit graphischer Oberfläche, das ein Telefon veranschaulichen soll (vgl. Abbildung 10). Gegenwärtig werden Instanzen von `PhoneIN` direkt von der Switch-Oberfläche aus erstellt und initialisiert. Durch die Kapselung der Funktionalität und der Ausführung von `PhoneIN` als Applet ist es jedoch mit geringem Aufwand (z. B. unter Benutzung einer Verteilungsplattform) möglich, die Endgeräte direkt im einem Web-Browser zu benutzen.

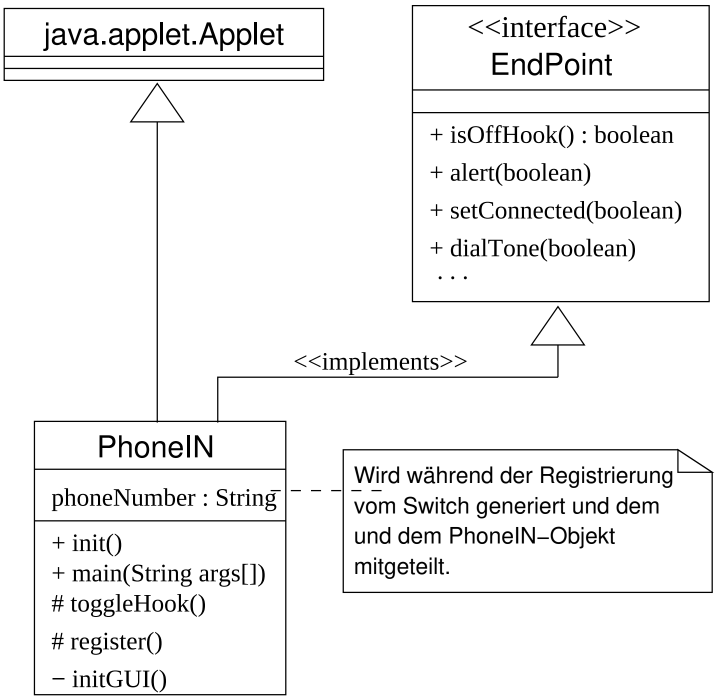

Abbildung 10: Vererbungsbeziehungen des `PhoneIN`-Endgerätes

Wurde ein `EndPoint` erstellt, versucht er sich zunächst bei der `CallControlFunction` zu registrieren. Dies geschieht durch einen Aufruf[11] an die `CallControlFunction`-Instanz, zu der seinem Konstruktor eine Referenz übergeben wurde. Obwohl eine Lösung mittels eines `CallControlFunction`-Singletons[12] eleganter gewesen wäre, bietet diese Vorgehensweise den Vorteil, bei Bedarf mehrere `CallControlFunction`-Instanzen in der selben JVM[13] zu benutzen. Für einen verteilten Switch ergibt sich jedoch das Problem, dem

[10] eine Klasse, die von java.applet.Applet erbt, die für ein Applet notwendigen Methoden implementiert, aber auch eine main()-Methode besitzt.

[11] public synchronized EndPointProxy registerEndPoint(EndPoint ep)

[12] vergleiche [Gamma] Singleton pattern

[13] Java Virtual Machine, entwickelt wurde mit Version 1.1.8

Endgerät die Referenz auf die ihm zugeordnete `CallControlFunction`-Instanz zu übergeben. Dies würde unter Umständen ein weiteres Objekt erfordern, das die Verwaltung der unterschiedlichen Switch-Instanzen übernimmt. Die Registrierung beim Switch entspricht in der realen Welt der Einrichtung eines Anschlusses durch den Netzbetreiber.

Während der Registrierung erstellt die `CallControlFunction` ein neues Proxy-Objekt (`EndPointProxy`) für das Endgerät und teilt diesem eine Rufnummer zu. Weiterhin werden jeweils eine Instanz von `O_StateMachine` und `T_StateMachine` erzeugt, initialisiert und dem `EndPointProxy` zugeordnet (vgl. Abbildung 11). Nach erfolgter Registrierung verläuft die Kommunikation zwischen dem Endgerät und dem Switch ausschließlich über das `EndPointProxy`-Objekt.

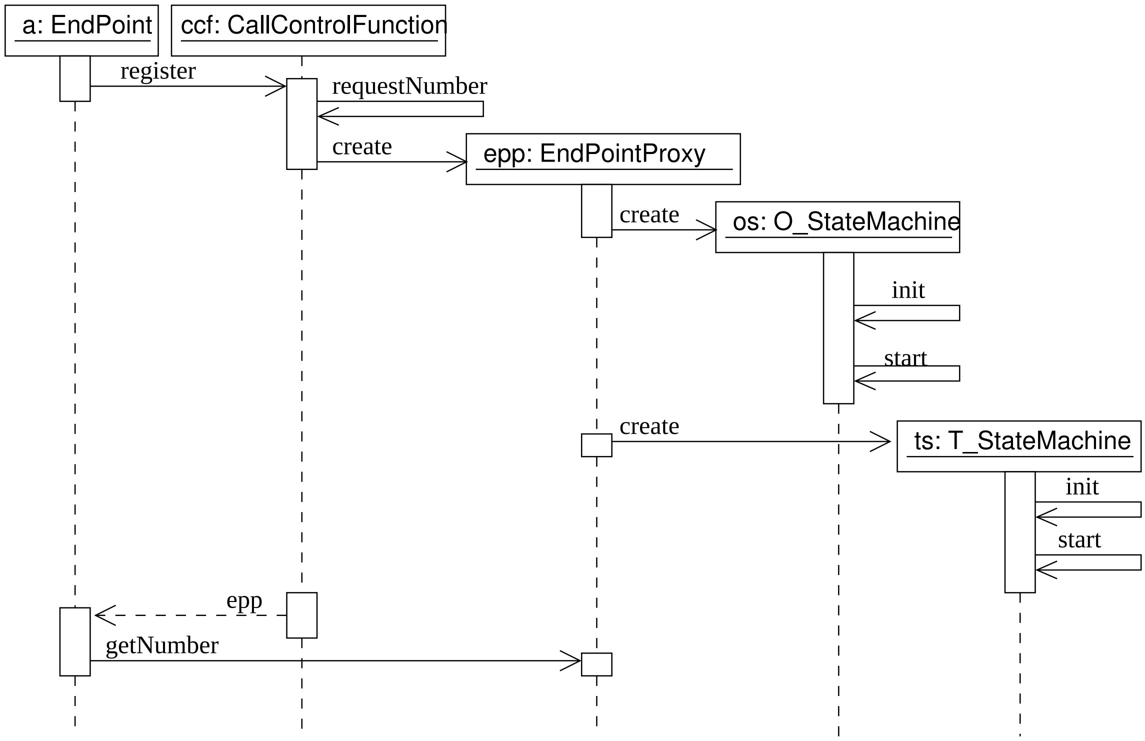

Abbildung 11: Registrierung eines Endgerätes

Die Endgeräte- bzw. Teilnehmernummern werden mit Hilfe eines Objekts der Klasse `SimpleNumberingPlan` erzeugt und verwaltet; dieses ist der `CallControlFunction`-Instanz zugeordnet. In der gegenwärtigen Implementierung werden zweistellige, aufeinanderfolgende Nummern erzeugt, beginnend mit einer willkürlich eingestellten Nummer. Die Schnittstelle, die `SimpleNumberingPlan` definiert ist jedoch für die Verwaltung beliebig erzeugter Nummern anwendbar. Gegenwärtig kann zusätzlich zum Erzeugen und Löschen von Nummern die Gültigkeit einer Nummer oder einer Teilnummer überprüft werden. Es werden allerdings nur die Nummern selbst verwaltet, ihre Zuordnung zu Endgeräten wird von `CallControlFunction` selbst vorgenommen.

2.3.2 EndPointProxy

Die Klasse `EndPointProxy` stellt die Schnittstelle zwischen der BCSM-Implementierung (`O_StateMachine` und `T_StateMachine`), den Endgeräten und der `CallControlFunction` dar. Instanzen dieser Klasse repräsentieren Switch-seitig ein Endgerät; `EndPointProxy` implementiert selbst das `EndPoint`-Interface. Alle Nachrichten von `StateMachine`-Instanzen an andere Komponenten[14] werden über das zugehörige `EndPointProxy`-Objekt verschickt.

Dies erlaubt einerseits die Bündelung der zu einem Engerät gehörigen Objekte um ein `EndPointProxy`-Objekt, andererseits könnten die Proxy-Objekte als Adapter[15] eingesetzt werden, falls z. B. `PhoneIN` als Endgerät-Implementierung ersetzt wird. Insbesondere Aufrufe von Hilfsmitteln, die von der `CallControlFunction` bereitgestellt und den `StateMachine`-Instanzen in Anspruch genommen werden, werden über `EndPointProxy`-Objekte abgewickelt. Beispiele dafür sind die `route()`-Methode (in diesem Fall der Versuch, eine Referenz einer terminierenden `StateMachine` anhand einer Nummer zu ermitteln) sowie das Überprüfen der vom Benutzer eingegebenen (Teil-) Nummern auf Gültigkeit.

2.3.3 Die Trigger

Um den Switch-Prototypen mit einer `ServiceControlFunction`-Implementierung zu betreiben, die "echte" SCF-Dienste zur Verfügung stellt, muss ein Trigger-Konzept bereitgestellt werden, das in der Lage ist, sowohl statische als auch dynamische Trigger an die Detection Points zu binden.

Gegenwärtig werden lediglich statische Trigger unterstützt, die mit Hilfe der GUI aktiviert und deaktiviert werden können. Befindet sich beim Betreten eines Detection Points der zugehörige Trigger im aktiven Zustand, so erfolgt ein Aufruf der zur Zeit verfügbaren "SCF-Dummy" (`SimplisticSCF`). Dies ist durch einen Methodenaufruf realisiert, der als Parameter relevante Werte (Teilnehmernummer, PIC) annimmt. Die Antworten der SCF erfolgen in Form eines Antwortobjektes der Klasse `SCFReply` oder einer davon abgeleiteten Klasse. Gegenwärtig existieren zwei abgeleitete Klassen, nämlich `DataChangeReply` und `StateChangeReply`. `DataChangeReply` ist für Antworten vorgesehen, die eine Änderung der aktuellen Werte der SCF bewirken soll (z. B. eine Änderung der Teilnehmernummer, etwa bei Weiterleitung eines Anrufes). Ein `StateChangeReply`-Objekt sollte für Antworten benutzt werden, die den gegenwärtigen PIC der aufrufenden `StateMachine` willkürlich ändern. Beispielsweise könnte im Rahmen einer Zugriffseinschränkung eine `O_StateMachine` nach Analyse der Nummer in den PIC O_Collect versetzt werden, um dem Benutzer die Eingabe eines Zugriffs-PIN zu ermöglichen (bei Sperrung von Prefixes etc.) `SCFReply` selbst kann für Antworten benutzt werden, die keinen Einfluß auf den weiteren Verlauf des Verbindungsaufbaus haben ("PROCEED").

Die Erkennung von aktiven Triggern geschieht direkt in der `setState()`-Methode von `O_StateMachine` bzw. `T_StateMachine`. Hier wird vor dem Betreten eines PIC ermittelt, welcher Detection Point (falls vorhanden) passiert wird. Für die in Frage kommenden Detection Points wird dann geprüft, ob der Benutzer in der GUI einen Trigger gesetzt hat. Trifft dies zu, so erfolgt der SCF-Aufruf mit dem aktuellen (alten) PIC und (falls verfügbar) den bisher ermittelten Daten als Parameter. Die Antwort der SCF wird ausgewertet und

[14] mit Ausnahme der SCF

[15] vergleiche [Gamma, Adapter pattern]

eine eventuelle Anweisung ausgeführt, bevor der "neue" PIC betreten wird. Gegenwärtig liefert die SCF nur "PROCEED"-Antworten; die Auswertung der Antworten ist also nur als Beispiel anzusehen.

2.4 Die Benutzeroberfläche

Um die Vorgänge in dem Simulator beobachten zu können, steht eine graphische Benutzeroberfläche zur Verfügung. Diese beinhaltet eine graphische Darstellung der Endpoints sowie der Vorgänge im Switch. Mittels eines Kontrollfensters wird der Switch gestartet/initialisiert und das Netzwerk aufgebaut (d. h. Endgeräte werden erstellt).

Die GUI ist mit *IBM Visual Age for Java 3.0* unter Benutzung des AWT (Abstract Window Toolkit) erstellt worden. Auch wenn die AWT-Komponenten gegenüber JFC/Swing optisch weniger ansprechend sind, bietet das AWT den Vorteil thread-sicher zu sein, eine Synchronisierung der Aufrufe von GUI-Methoden kann also entfallen.

2.4.1 Das Kontrollfenster

Der Simulator-Prototyp wird von einem Kontrollfenster aus gestartet.[16] Das Kontrollfenster verfügt über ein Menü, mit Hilfe dessen grundlegende Aktionen ausgeführt werden können. In einer Liste werden die registrierten Endgeräte angezeigt. Im unteren Bereich des Fensters kann eine Verzögerung der Aktionen des Switch eingestellt werden (vgl. Abbildung 12).

Mit dem Menüpunkt "Simulator->Init" kann der Benutzer ein Switch-Objekt[17] erzeugen und initialisieren [18].

Ist der Switch initialisiert, kann mit dem Menüpunkt "Simulator->Neuer Endpoint" ein `PhoneIN`-Endgerät erzeugt; dieses versucht sich anschließend beim Switch zu registrieren. Beim Erstellen eines neuen Endgerätes wird ebenfalls ein `EndpointMonitor` für das neue Endgerät erzeugt. Das Hinzufügen eines neuen Endgerätes kann generell jederzeit nach der Initialisierung geschehen. Der Menüpunkt "Beenden" beendet die JVM-Sitzung[19].

Hat sich ein Endgerät erfolgreich beim Switch registriert (vgl. Seite 15), wird es in der Liste des Kontrollfensters angezeigt. Wenn solch ein Eintrag markiert und anschliessend der Knopf "Anzeigen" gedrückt wird, so versucht der Simulator, das zugehörige `PhoneIN`-Fenster sowie das zugehörige `EndpointMonitor`-Fenster an die Oberfläche des Desktops zu bringen[20].

Bei jedem Übergang zwischen PICs wird eine Verzögerung benutzt, um die Verfolgung der Abläufe im Switch zu ermöglichen. Im unteren Bereich des Fensters kann entweder die fest eingestellte Verzögerung (250 ms) oder eine benutzerdefinierte Zeit (in Millisekunden) gewählt werden. Die neue Verzögerungsdauer wird nach Drücken des Knopfs "Festlegen"

[16] dazu reicht es, die Klasse lmu.mnm.insim.INSim zu starten. Die Klassen aus dem package lmu.mnm.insim müssen über die CLASSPATH-Variable erreichbar sein.

[17] genauer: eine Instanz von CallControlFunction und SimplisticSCF

[18] Gegenwärtig sollte der Initialisierungsbefehl genau einmal gegeben werden, da eine Unterstützung mehrerer Switches nicht vorhanden ist.

[19] System.exit()

[20] Hierzu werden nur die der AWT zur Verfügung stehenden Methoden (java.awt.Window) benutzt. Je nach benutztem Betriebsystem/Graphiksystem/Fenstermanager können die Ergebnisse variieren.

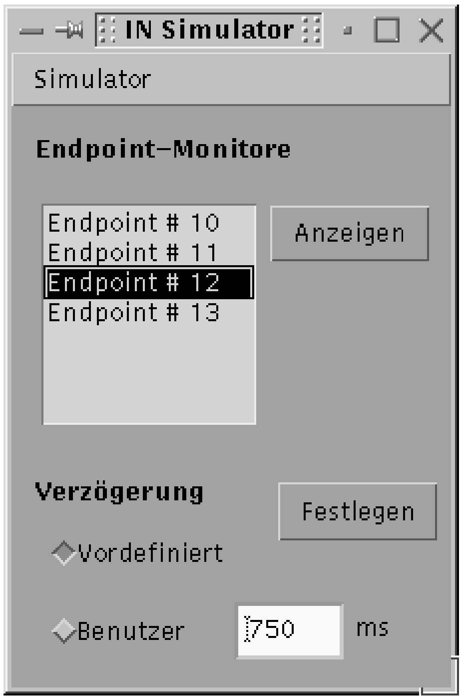

Abbildung 12: Screenshot des Kontrollfensters (mit vier registrierten Endgeräten)

eingestellt[21]. Will man die BCSM-Abläufe genau verfolgen, können hier Werte von einigen Sekunden durchaus sinnvoll sein.

2.4.2 Das PhoneIN Applet

Dieses Applet entspricht in seinem Funktionsumfang einem einfachen Endpoint, d. h. es stellt die Funktionen eines normalen Telefons zur Verfügung (vgl. Abbildung 13). Mit dem Knopf "Hörer" kann der Benutzer den Hörer abheben oder auflegen, mit den Tasten des Ziffernblocks kann eine Nummer gewählt werden, die oberhalb des Ziffernblocks zur Kontrolle angezeigt wird.

In dem auf der rechten Seite befindlichen Statusfeld werden die Zustände bzw. "Meldungen" eines Telefons dargestellt. Der Anzeigepunkt "Besetzt" etwa entspricht dabei dem Besetztton etc. Der Anzeigepunkt "Amtsleitung verfügbar" gibt an, dass der `Endpoint` bei einem Switch registriert ist . Er entspricht somit in etwa der Freischaltung des Anschlusses durch einen TK-Anbieter.

Der untere Bereich des Fensters wird nach erfolgreichem Verbindungsaufbau aktiviert und stellt eine Chat-Funktion zur verfügung. Eingegebener Text wird nach Drücken des Knopfs "Senden" im Textfenster des verbundenen `PhoneIN`-Fensters angezeigt.

[21] Es handelt sich hierbei um eine Klassenvariable in der Klasse StateMachine. Der Thread einer jeden O_StateMachine oder T_StateMachine wird bei Aufruf der setState()-Methode um die angegebene Anzahl Millisekunden angehalten.

Abbildung 13: Originierendes und terminierendes Endgerät (originierend links).

2.4.3 Der Endpoint-Monitor

Jedem `PhoneIN`-Endgerät entspricht Switch-seitig ein `EndPointMonitor`.

Das Monitor-Fenster ist in drei Bereiche aufgeteilt (vgl. Abbildung 14). Der oberste Bereich repräsentiert den originierenden, der mittlere den terminierenden Teil des BCSM. Im unteren Teil werden in einer Textbox die Meldungen der dem Endgerät zugehörigen `O_StateMachine` und `T_StateMachine` ausgegeben. Der aktuelle Zustand (PIC, DP) wird durch das Aufleuchten von beschrifteten “Status-Lämpchen”[22] im originierenden bzw. terminierenden Teil des Fensters gezeigt. Es wurde versucht, diese entsprechend der Diagramme für O_BCSM bzw. T_BCSM anzuordnen (vgl. Abbildungen 2 und 3).

Die graphischen Komponenten des Monitors befinden sich in der Klasse `EndPointDisplay`, die von `java.awt.Panel` abgeleitet ist und das Interface `EndPointMonitor` implementiert.

2.4.4 Aktivierung / Deaktivierung der Trigger

Die Statuslämpchen der Detection Points verfügen über Schaltelemente (Checkbox), mittels derer der Benutzer willkürlich einen Trigger aktivieren oder deaktivieren kann. Ist die einem Detection Point zugehörige Checkbox aktiv, wenn dieser betreten wird, so findet ein Aufruf an die SCF statt. Das “Lämpchen” des Detection Points leuchtet während der

[22] StatusLED bzw. SwitchableStatusLED

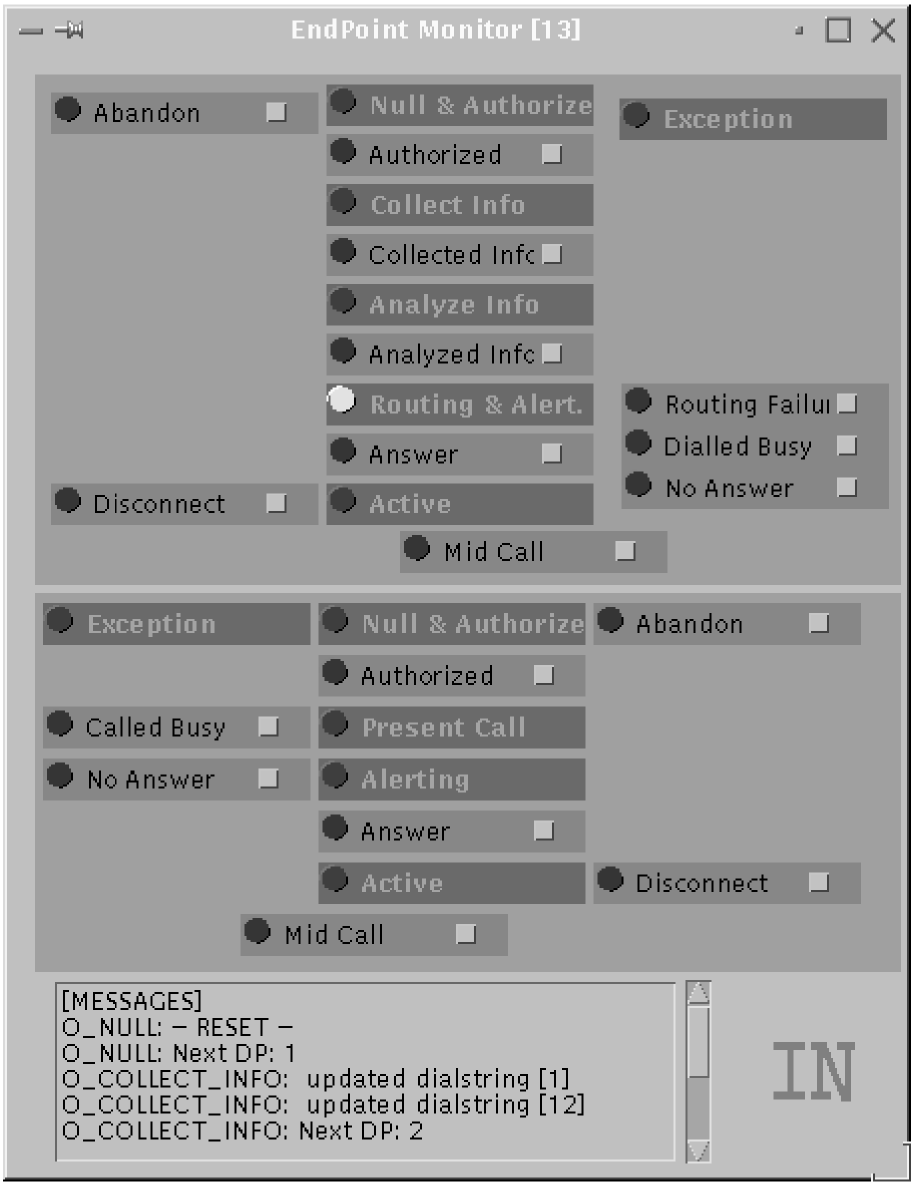

Abbildung 14: Screenshots eines `EndPointWindow`

Ausführung des Aufrufs. Rechts unten im EndPointWindow befindet sich eine Anzeige mit dem Schriftzug "IN", die während des SCF-Aufrufs rot wird (normalerweise ist sie grau bzw. nicht sichtbar. Vgl. Abbildung 14).

2.5 Installation und Ausführung des Prototypen

Der Code für den Switch-Prototypen befindet sich im Package `lmu.mnm.insim`. Eine Installation ist nicht notwendig; das Ablegen oder Entpacken der Jarfile in ein Verzeichnis ist ausreichend. Um den Prototypen auszuführen werden benötigt:

- eine Java Laufzeitumgebung (JRE) mit Version 1.1.8 oder höher.

- die zum Prototypen gehörigen `.class`-Dateien, z. B. in einer Jarfile verpackt.
- ein Verweis auf die `.class`-Dateien in der Umgebungsvariable `CLASSPATH`

Die zu startende Klasse heisst `INSim`. Nach Starten dieser Klasse erscheint das Kontrollfenster (vgl. Abbildung 12).

Beispiel zum Starten unter Un*x:

Überprüfen der Version der Java Laufzeitumgebung (optional):

```
$ java -version
java version "1.2.2"
Classic VM (build 1.2.2-L, green threads, nojit)
```

Umgebungsvariable CLASSPATH auf den Code zeigen lassen:

```
$ export CLASSPATH=insim.jar
```

Ausführen:

```
$ java lmu.mnm.insim.INSim &
```

3 Weiterentwicklung des Prototypen

Der vorliegende Prototyp versteht sich als grundlegende, zu erweiternde Arbeit. Es ist also wichtig, Aspekte möglicher Erweiterungen und Veränderungen am Switch-Prototypen zu betrachten. Im folgenden werden einige Überlegungen in bezug auf funktionelle Erweiterungen, sowie auf das Hinzufügen neuer und das Ersetzen bestehender Komponenten (z. B. GUI-Komponenten), dargelegt.

3.1 Vorschläge zum Erweitern oder Ersetzen bestehender Komponenenten

3.1.1 Erweiterungen von StateMachine und davon abgeleiteten Klassen

Für den Betrieb des befindlichen Codes mit einer SCF-Implementierung sind insbesondere einige Erweiterungen der Klasse `O_StateMachine` sinnvoll. Im PIC O_Analyze kann eine Überprüfung der gewählten Nummer implementiert werden; hier könnte z. B. eine "IN-Nummer" erkannt werden und entsprechend ein SCF-Aufruf ausgeführt werden. Dazu muss in der `iterate()`-Methode von `O_StateMachine` entsprechender Code eingefügt werden. Zum Erkennen von speziellen, nicht an ein Endgerät gebundenen Nummern bietet sich z. B. die Erweiterung der Klasse `SimpleNumberingPlan` um eine Methode `boolean isINNumber(String number)`.

Manche Dienste erfordern dynamische Trigger, die in von der SCF willkürlich aktiviert werden können. Die Erkennung von aktiven Triggern bzgl. eines Detection Point beschränkt sich gegenwärtig auf die von dem Benutzer in der GUI gesetzten (statischen) Trigger. Diese Erkennung findet gegenwärtig in der `setState()`-Methode der betreffenden `O_StateMachine`- oder `T_StateMachine`-Instanz statt, während der Zustand der Trigger von der GUI-Komponente abgefragt wird. Dynamische Trigger müssen indes für jede `StateMachine`-Instanz gesondert verwaltet werden. Für diesen Zweck bedarf es einer Verwaltungsklasse (etwa: `TriggerManager`), sowie einer Verknüpfung jeweils einer Instanz dieser Klasse mit jeder `StateMachine`-Instanz (so wäre es auch möglich, ein bedingtes Auslösen von Triggern zu implementieren). Weiterhin müsste StateMachine selbst um Methoden erweitert werden, die das Aktivieren und Deaktivieren von Triggern ermöglichen und von der SCF direkt aufgerufen werden. Alternativ könnte das Aktivieren/Deaktivieren von Triggern mittels der Nachrichten-Objekte (`SCFReply` etc.) realisiert werden.

3.1.2 Ersetzen oder Erweitern der Endgeräte (`EndPoint`-Implementierung)

Das Interface `EndPoint` beschreibt eine Schnittstelle zu einem einfachen Endgerät, das mit Hilfe einer `EndPointProxy`-Instanz mit dem Switch-Kern kommuniziert (vgl. 2.3.1). Es ist möglich, die bestehenden Endgeräte (`PhoneIN`, vgl. Abbildung 13) durch z. B. mit JFC/Swing realisierten Applets zu ersetzen, sowie diese auf mehrere Rechner verteilt (z. B. durch den Einsatz von Java RMI oder eines ORB) zu betreiben. So veränderte bzw. neu erstellte Endgeräte müssen das `EndPoint`-Interface implementieren, sowie die in 2.3.1 beschriebene Registrierungsprozedur beherrschen.

Für den Fall, dass eine Verteilungsplattform benutzt wird, muss bedacht werden, dass nicht nur das neue Endgerät und `EndPointProxy` als entferntes Objekt (*remote object*) beschrieben werden muss, sondern auch `CallControlFunction`. Das Endgerät muss zunächst eine Referenz auf eine `CallControlFunction`-Instanz erhalten (von der RMI Registry oder einem CORBA NameService) bevor es sich beim Switch registrieren kann und einen `EndPointProxy` zugewiesen bekommt.

Die Endgeräte müssen nicht unbedingt eine graphische Oberfläche besitzen. Denkbar sind auch `EndPoint`-Implementierungen, die eine Text-Schnittstelle besitzen und im Rahmen einer Simulation automatisch Verbindungen aufbauen bzw. terminieren.

3.1.3 Ersetzen der `EndPointMonitor`-Implementierung

Die in `EndPointDisplay` implementierte graphische Darstellung von Vorgängen im Switch wird seitens des Switch-Kerns (in diesem Fall: seitens der `O_StateMachine` bzw. `der T_StateMachine` eines Endgerätes) ausschließlich über das `EndPointMonitor`-Interface angesprochen. Ein Ersetzen der `EndPointMonitor`-Implementierung sollte sich also verhältnismässig einfach gestalten. Lediglich die Instantiierung der `EndPointWindow`-Objekte (mit enthaltenem `EndPointDisplay`) muss angepasst werden. Es ist allerdings auch möglich, den Switch ohne `EndPointMonitor` zu betreiben; selbstverständlich fehlt dann auch die Möglichkeit, statische Trigger zur Laufzeit zu setzen.

Bei einer Implementierung mit JFC/Swing muss beachtet werden, dass gegebenenfalls mehrere Threads gleichzeitig auf den `EndPointMonitor` zugreifen werden. Da Swing anders als das AWT keine Synchronisierung solcher Aufrufe bietet, muss diese durch den eigenen Code vorgenommen werden.

3.2 Einbindung neuer Komponenten

3.2.1 SCF

Die Projektarbeit zielte mittelbar auf die Beobachtung von Vorgängen in einer Service Control Function (SCF) eines IN. Um dieses Ziel zu erreichen, muss der gegenwärtig vorhandene "SCF-Dummy" (`SimplisticSCF`) durch eine Implementierung ersetzt werden, die in der Lage ist, bausteinbasierte Dienste zu unterstützen.

3.2.2 IN-Dienste

Die Implementierung einer SCF zielt auf die Bereitstellung von Diensten. Diese werden aus Grundbausteinen (SIBs, vgl. 1.3.3) zusammengestellt, wobei die SCF selbst Unterstützung in Form einer Schnittstelle zu diesen Bausteinen bieten sollte. Zur Implementierung der SIBs könnten Java Beans zum Einsatz kommen, da diese den Anforderungen an SIBs am nächsten kommen und zudem leicht mit dem bestehenden Code zu integrieren sind.

Mit Java Beans oder einer ähnlichen Technologie kann so ein Framework geschaffen werden, das es erlaubt, unter Einsatz einer Entwicklungsumgebung (z. B. einer Bean Box) IN-Dienste mit minimalem Entwicklungsaufwand (z. B. mittels Drag&Drop oder "visuellen Programmierung") zu erstellen.

Die Schnittstellen zu IN-Diensten und auch zu den einzelnen SIBs sollten in der SCF definiert werden, die in einer solchen Form noch aussteht. Auch ein Managementsystem für die SIBs kann nur zusammen mit oder erst nach der Erstellung der Schnittstellen der SIBs spezifiziert werden.

3.3 Zusammenschaltung mehrerer Switches

In der Praxis existiert ein Switch nicht allein. Vielmehr bestehen TK-Netze aus vielen Switches mit angeschlossenen Endgeräten. Ein Simulator eines IN-Netzes sollte also die Möglichkeit bieten, auch solche Strukturen zu realisieren. Insbesondere sollten die Voraussetzungen gegeben sein, einen Anruf eines Endgeräts von dem "eigenen" Switch zum Switch des angewählten Endgerätes vermitteln können. Mit dem vorliegenden Prototypen ist dies auf zwei Weisen möglich: erstens über die Zusammenschaltung über das `Endpoint`-Interface und zweitens mittels eines IN-basierten Vorgehensweise.

3.3.1 Hierarchisch

Sollen zwei Switches mittels des EndPoint-Interfaces zusammen geschaltet werden, kann dem ersten der Switches eine Vorwahlnummer zugeteilt werden, die beim zweiten Switch eingetragen wird. Beginnt eine vom Benutzer gewählte Nummer mit dieser Vorwahl, so baut der erste Switch (genauer: die `O_StateMachine`, die dem originierenden Endgerät zugeordnet ist) unter Angabe der ganzen Nummer eine Verbindung zum zweiten Switch auf.

Werden die Switches baumartig in ein Netzwerk eingebunden (d. h. mit einer Hierarchie der Vorwahlen), kann auf diese Weise der Baum zunächst zur Wurzel hin durchwandert, und danach zu dem Blatt (Switch) hin, der den gesuchten Teilnehmer enthält.

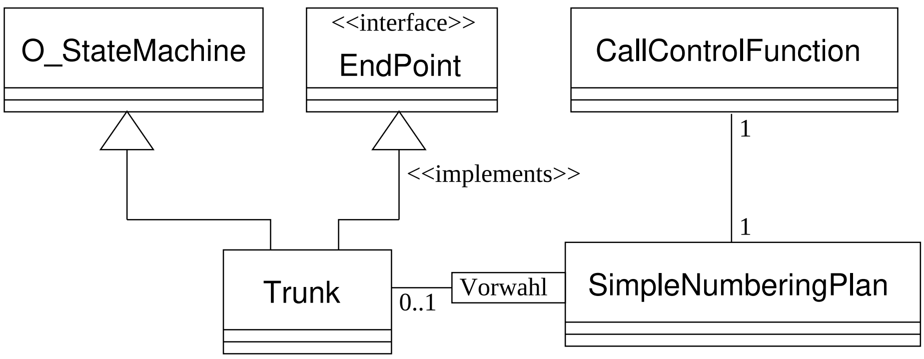

Abbildung 15: Verbindungsglied zwischen Switches

Um dies zu erreichen ist ein Verbindungsglied zwischen Switches notwendig (vgl. Abbildung 15), das einerseits das `EndPoint`-Interface implementiert, anderseits in der Lage ist, die notwendigen Operationen bezüglich der Vorwahlen durchzuführen (hier: `Trunk`). Zusätzlich muss solch ein Objekt einen Teil der Funktionalität einer `O_StateMachine` besitzen (z. B. durch Vererbung), um die Verbindung weiterleiten zu können.

3.3.2 Zusammenschaltung mittels IN-Funktionlität

Eine andere Möglichkeit, Anrufe von Switch zu Switch weiterzuleiten wäre, einen IN-Aufruf zu benutzen (vgl. Abbildung 16). Ist eine Zielnummer nicht direkt am Switch angeschlossen, wird ein IN-Dienst aufgerufen, der einen Leitweg findet und einer `O_StateMachine` eines originierenden Switch die T_StateMachine des gewünschten Verbindungspartners direkt angibt.

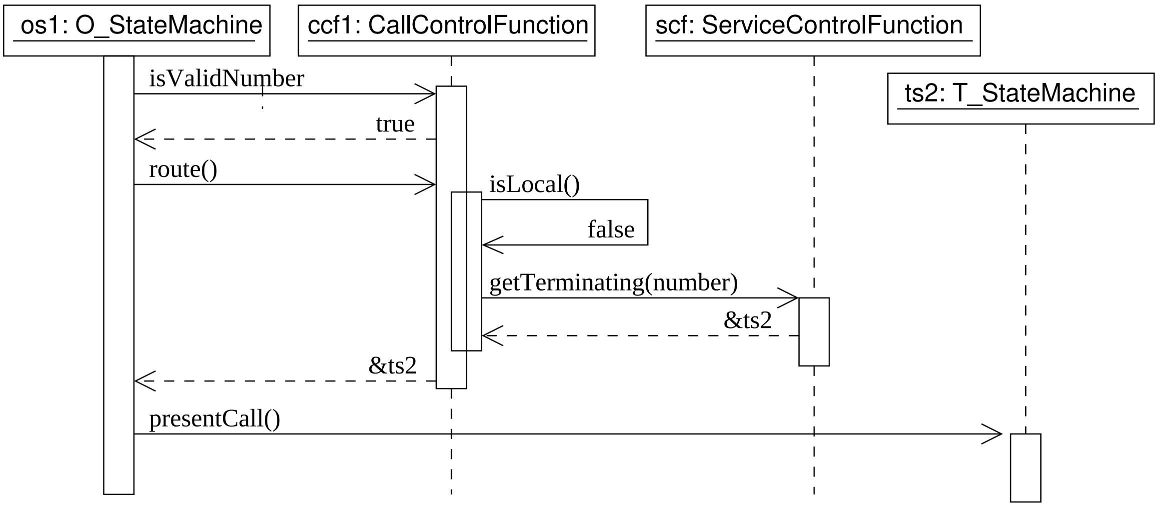

Abbildung 16: Verbindungsaufbau mit IN-Aufruf

Diese Vorgehensweise entspricht nicht der eines realen Systems (sie fordert ja, das eine Referenz eines entfernten Objekts übergeben wird, ohne dass ein Leitweg zu diesem Objekt bekannt ist). Um den Ablauf von IN-Diensten zu beobachten, insbesondere solcher, die von verschiedenen CCFs aus initiiert werden, könnte dieser Weg aufgrund einer einfacheren Implementierung hilfreich sein; zudem würden durch dieses Verfahren zusätzliche (zu beobachtende) IN-Aufrufe generiert. Er setzt allerdings eine funktionstüchtige SCF voraus, die die Weiterleitung von Anrufen unterstützt.

4 Zusammenfassung und Ausblick

Der vorliegende Prototyp kann unter Umständen als Grundlage für die Erstellung eines Simulators für IN dienen. Er würde insbesondere für die Beobachtung der Vorgänge in einer Service Control Function von Nutzen sein. Voraussetzung hierzu wäre die Implementierung entsprechender Erweiterungen. Mittels der gebotenen Strukturen und Schnittstellen sollen aufbauende Arbeiten, die dieses Ziel verfolgen, erleichtert werden.

Da ein Simulator in erster Reihe der Beochbachtung von Vorgängen dient, kommt der Visualisierung der zu beobachtenden Daten eine besondere Bedeutung zu. Je nach Anforderungen können die bestehenden Anzeigekomponenten genutzt, oder durch neue, dem Zweck angepasste ersetzt werden.

Der vorliegende Prototyp bietet lediglich eine Implementierung mit zugehörigen Visualisierung der unteren Schichten eines IN; zentral ist hierbei die Umsetzung des BCSM. Auch wenn die für die Weiterentwickung zur Verfügung gestellten Primitiva ausreichend sein können, ist es unter Umständen notwendig, Veränderungen direkt am Code des Prototypen vorzunehmen. Hierzu sei auf Abschnitt 3 verwiesen. In diesem wurde versucht, auf Details der Implementierung hinzuweisen, die für Veränderungen am Code hilfreich sein könnten.

Literatur

[DFPIN] ITU-T Recommendation Q.1214. *Distributed Functional Plane for Intelligent Network CS-1.* 1995

[Black] Black, Uyless: *The Intelligent Network.* Prenctice Hall. New Jersey 1998

[Booch] Booch et. al: *The Unified Modeling Language*

[Muller] Muller, Pierre-Alain: *Instant UML.* Wrox Press. Birmingham 1997

[Gamma] Gamma, Erich et al.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley