Entwicklung von geometrisch unterscheidbaren Komponenten zur Vereinheitlichung von Mulan-Protokollen

Lawrence Cabac

22. Oktober 2002

Inhaltsverzeichnis

1 Einleitung

1.1 Kontext

In der Informatik ist seit den Anfängen der Programmierung das Problem der Lesbarkeit von Programmcode bekannt. Mit verschiedenen Mitteln wird versucht, den Code verständlicher, einfacher zugänglich und übersichtlicher zu gestalten. Das allgemeine Problem, das es zu verringern gilt, heißt Komplexität. Zu den Methoden, um dies zu erreichen, gehören zum Beispiel die strukturierte[1] oder die objektorientierte Programmierung, aber auch integrierte Entwicklungsumgebungen (*Integrated Development Environment*, IDE), Hervorheben von Wörtern durch Farbe (*Syntax Highlighting*) und nicht zuletzt das Kenntlichmachen von syntaktischen Einheiten durch Einrücken des Codes (*Indentation*).

Visuelle Programmiersprachen (VPL) kennen ähnliche Probleme. Die als "Limit von Deutsch" bekannt gewordene Aussage [McI98], dass nicht mehr als fünfzig graphische Elemente gleichzeitig auf einem Bildschirm dargestellt werden können, lässt das Problem erahnen. Viele Ansätze versuchen, mit modularen Elementen die Komplexität hinter graphischen Elementen, z.B. *Icons*, zu verstecken. Dies führt zu sehr komplexen Inhalten von einzelnen graphischen Elementen. Dennoch ist dies ein guter Ansatz, denn durch die modulare Vergröberung des Codes wird die Möglichkeit geschaffen, die Übersicht zu erhöhen. Allerdings wird die Anordnung der Programmelemente von Programmiersprachen, wie z.B. LabVIEW [Lab] oder Prograph [Pic], durch die Tools erheblich eingeschränkt, um einen einheitlichen Stil zu gewährleisten.

Workflows sind zwar keine Programme, aber sie beschreiben Prozesse. Somit treten auch bei ihrer Modellierung ähnliche Komplexitätsprobleme auf. Van der Aalst [vdA00] bietet mit den *Workflow-Pattern* einen Vorschlag zur Vereinheitlichung von immer wiederkehrenden Mustern in *Workflows*. Die Vereinheitlichung von Modellierungsmethoden und die Benennung von einzelnen Teilen der Prozesse durch *Pattern* scheinen brauchbare Mittel zu sein, mit denen die Verständlichkeit von Prozessbeschreibungen verbessert werden kann.

Petrinetze können wie *Workflows* zur Modellierung von Prozessen benutzt werden. Sie können aber auch (wie bei VPL) simuliert werden, das heißt sie sind Programme. Deswegen ist es durchaus legitim, im Zusammenhang mit Petrinetzen von Code zu sprechen. Es gibt generelle Empfehlungen, wie Petrinetze aussehen sollten. Diese beziehen sich in den meisten Fällen auf die einzelnen Elemente der Netze: Transitionen, Stellen, Kanten und Beschriftungen. Die von Jensen [Jen96] genannten Empfehlungen beziehen sich entweder auf die Elemente der Netze oder es sind grundsätzliche Regeln, die sehr frei ausgelegt werden können. Übergeordnet und für spezielle Netze gibt es leider keine weiteren Empfehlungen und somit sehen Petrinetze sehr unterschiedlich aus, abhängig vom Entwickler, dem verwendeten Tool, und dem Anwendungsgebiet. Zum Teil liegt dies an dem überaus einfachen Konzept der Petrinetze, die mit drei Elementen auskommen, das trotz der Einfachheit dennoch sehr mächtig ist. Generell ist zu sagen, dass es natürlich von Vorteil sein

[1] O.-J. Dahl,E. W. Dijkstra, C. A. R. Hoare, *Structured Programming*, [DDH75]

kann, dass Netze sehr unterschiedlich aussehen können. Wenn es aber um die Darstellung von gleichartigen Netzen geht, ist eine konforme Darstellung nützlicher.

Petrinetze sind ideale Konstrukte für die Darstellung von nebenläufigen Prozessen. Das Tool der Wahl am Fachbereich Informatik der Universität Hamburg für die Modellierung und Simulation der Petrinetze ist Renew. Es bietet sowohl eine komfortable Möglichkeit, die Netze zu erstellen, als auch zu simulieren. Die Netze, die mit Renew erstellt werden können, sind Referenznetze, eine Erweiterung der gefärbten Netze (*Coloured Petri Nets*,[Jen96],CPN). Netzelemente können mit Anschriften versehen werden, die Java-Code enthalten dürfen.

Mulan (Multiagentennetze) basiert auf Renew und ist ein System, das die Vorzüge der Petrinetze (Nebenläufigkeit) ausnutzt. Es bietet zusammen mit Renew die Möglichkeit, Software nach dem agentenorientierten Paradigma auf Petrinetz-Basis zu erstellen. Das Paradigma der agentenorientierten Softwareentwicklung (AOSE) ist noch relativ jung im Vergleich zu den anderen Paradigmen wie das funktionale oder das objektorientierte. Der Grund für dieses neue Paradigma liegt in der hohen Komplexität der Softwaresysteme. Besonders bei nebenläufigen und adaptiven Prozessen sind die herkömmlichen Verfahren sehr eingeschränkt. Da Agenten per Definition unabhängige, eigenständige Komponenten sind, bieten sie gerade für stark nebenläufige Prozesse einen interessanten Ansatz.

1.2 Motivation

Diese Arbeit beschäftigt sich mit der Benutzbarkeit (*Usability*) von Petrinetzen. Speziell sollen die Protokolle von Mulan-Agenten betrachtet werden. Die Aufgabe der Protokolle ist es, das Verhalten und die Kommunikation der Agenten festzulegen. Dabei kann ein Agent auf eine bestimmte Nachricht, die er erhält, in einer bestimmten Weise reagieren. Zu einer Nachricht wird ein Protokoll ausgewählt, das die Reaktion des Agenten bestimmt. Alternativ können Agenten Protokolle auch pro-aktiv starten, d.h. das Starten eines Protokolls ist nicht zwingend eine Reaktion auf eine Nachricht.

Anhand der Protokolle, die im Laufe des Projektes AOSE[2] [BGK$^+$02] entstanden sind, sollen die Verbesserungsmöglichkeiten für das Design von solchen Netzen betrachtet werden. Aufbauend auf Mulan wurde in dem Projekt das bekannte Spiel *Die Siedler von Catan* implementiert. Das Spiel sollte vollständig aus Agenten bestehen, diese waren die Bank, die Insel, der Spieler, die Spielleitung, das Handelsamt, der Planer und der menschliche Spieler. Die Hauptaufgabe der Projektteilnehmer bei der Implementation bestand darin, die Protokolle zu zeichnen. Im Verlauf des Projektes entstanden so mehr als 50 Protokolle. Bei der Entwicklung dieser Protokolle wurde sehr schnell klar, dass die Umsetzung der Modellierung eine umständliche und fehleranfällige Aufgabe ist. Weiterhin kam hinzu, dass Teile, und damit bestimmte Aufgaben der Protokolle, immer wieder vorkamen. Um die Entwicklung wiederkehrender Netzteile effizienter zu gestalten, wurden meistens *"Copy and Paste"* Verfahren benutzt. Es liegt nahe, ein Verfahren zu entwickeln, dass dem

[2] 18.336 Projekt: Agentenorientierte Softwareentwicklung, Michael Köhler, Daniel Moldt, Heiko Rölke Wintersemester 2001, FB Informatik, Universität Hamburg

Entwickler redundante Entwickleraufgaben abnimmt. Genau dies soll die hier vorgestellte Erweiterung von Renew leisten.

Für das Zeichnen von Protokollen soll in dieser Arbeit eine einheitliche Form gefunden werden. Es wird ein Ansatz verfolgt, Kontrollfluss und Datenstrukturen strikt voneinander zu trennen. Dadurch wird die Übersicht über den Ablauf eines Protokolls verbessert. Weiterhin sollen sinnvolle syntaktische Einheiten (Netz-Komponenten) für diese Netze gefunden werden. Diese sollen möglichst sämtliche Aufgaben der Protokolle abdecken, so dass die Protokolle möglichst ausschließlich aus den Netz-Komponenten bestehen. Außerdem sollen diese Netz-Komponenten so modular aufgebaut sein, dass sie leicht miteinander verknüpft werden können. Netz-Komponenten können dann für zukünftige Entwicklungen zur Verfügung gestellt werden. Die Vorteile eines solchen Ansatzes sind ein modularer Aufbau und eine einheitliche Form der Protokolle. Daraus folgt ein besserer Überblick und eine verbesserte Lesbarkeit dieser Petrinetze. Das ist gerade in Projekten, an denen mehrere Personen arbeiten, wünschenswert. Ohne einheitliche Form des Codes (und des Programmierstil) ist sowohl ein Austausch von Code als auch die Kommunikation der Projektteilnehmer (über die Implementation) nur schlecht möglich.

1.3 Überblick

In den folgenden Kapiteln werden zunächst die Systeme und Begriffe eingeführt (Kapitel 2). Anschließend werden einige Protokolle exemplarisch vorgestellt. Die Evaluation der Protokolle beschließt den Analyseteil (Kapitel 3).

In dem Implementationsteil soll ein exemplarischer Vorschlag für die Darstellung von Protokollen mit Hilfe von vordefinierten (visuellen) Komponenten erörtert werden (Kapitel 4). Weiterhin soll ein Tool erstellt werden, um die entworfenen Netz-Komponenten in Renew komfortabel benutzen zu können (Kapitel 5).

2 Einführung

In diesem Kapitel sollen die einzelnen Komponenten des Systems erläutert und einige Begriffe, die in dieser Arbeit Verwendung finden, eingeführt werden. Es wird in dieser Arbeit vorausgesetzt, dass der Leser oder die Leserin sich mit den Konzepten der gefärbten Petrinetze auskennt.[3]

2.1 Das benutzte System

Mulan (Multiagentennetze) ist ein Multiagentensystem, dass auf Renew[4], einem Editor und Simulator für Referenznetze, aufbaut. Es ist vollständig mit Petrinetzen implementiert.

2.1.1 Petrinetze

Petrinetze sind gerichtete Graphen. Die Knoten von Petrinetzen bestehen aus zwei disjunkten Mengen, dies sind die Transitionen und die Stellen. Kanten dürfen sich nur zwischen Transitionen und Stellen oder umgekehrt befinden. Auf den Stellen können sich Marken (*Token*) befinden, die durch Transitionen über die gerichteten Kanten auf die folgenden Stellen übertragen werden können. Dieser Vorgang wird schalten (*to fire*) genannt. Petrinetze haben sowohl eine graphische als auch eine formale Repräsentation.

Einfache Petrinetze werden auch Stellen-/Transitionsnetze genannt (S/T-Netze). Bei ihnen gibt es nur anonyme Marken, die somit nicht voneinander zu unterscheiden sind. Als Erweiterung dazu gibt es die gefärbten Netze (*Coloured Petri Nets*), bei denen die Marken benannt sind, sich also voneinander unterscheiden.

Referenznetze sind Petrinetze, bei denen die Marken nicht nur individuell unterscheidbar sind, sondern bei denen Marken beliebige Objekte darstellen können. Speziell können Marken auch wieder Petrinetze sein.

2.1.2 Renew

Renew ist ein Petrinetzeditor und -simulator. Mit Renew können Referenznetze dargestellt und simuliert werden, was ein entscheidender Faktor für die Modularität und Erweiterbarkeit der entwickelten Netze ist. Mit dem Editor ist es möglich, Petrinetze zu erstellen (zeichnen). Ein Netz, das in dem Editor geöffnet ist, kann durch den Simulator ausgeführt werden. Dazu wird von dem Simulator ein Exemplar (*Instance*) des Netzes erzeugt. Das Verhältnis von Netz und Exemplar ist vergleichbar mit dem Verhältnis zwischen Objektklasse und Objektvorkommnis. Es können beliebig viele Exemplare der Netze erzeugt werden.[5] Abbildung 1 zeigt die Bedieneroberfläche von Renew, ein Petrinetz (im Hintergrund) und ein Netzexemplar. Die Bedieneroberfläche besteht aus einer Menüleiste,

[3]Nähere Informationen z.B. bei Kurt Jensen, *Coloured Petri Nets* [Jen96] und Wolfgang Reisig, *Petrinetze* [Rei82].

[4]Mehr Informationen zu Renew können im *Renew User Guide* [KWD01] gefunden werden.

[5]Dies kann für Protokolle insbesondere dann zutreffen, wenn die Agenten gleiche oder ähnliche Nachrichten von verschiedenen Agenten erhalten.

zwei Werkzeugpaletten und einer Informationszeile. Die erste Palette besteht aus Knöpfen für die Werkzeuge zum Erstellen, Einfügen und Manipulieren von einfachen graphischen Elementen, die zweite bietet Werkzeuge (*Tools*) für Petrinetzelemente.

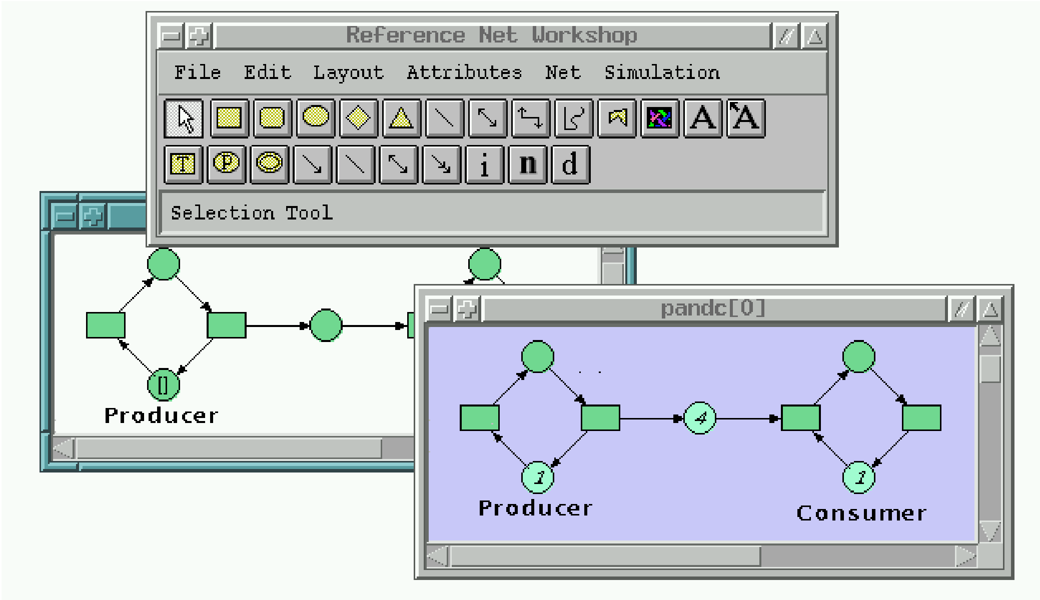

Abbildung 1: Renew GUI, Petrinetz und Netzexemplar (Producer/Consumer)

Eine Besonderheit von Renew ist, dass es Referenznetze simulieren kann. Referenznetze sind Petrinetze, bei denen Marken selbst wieder Petrinetze sein dürfen. Dabei wird eine Referenzsemantik angenommen, die Marke bildet also eine Referenz auf ein Netz. In Renew können Marken beliebige Java-Objekte referenzieren und somit Renew-Netze, die auch Java-Objekte sind. Eine Marke kann somit eine Referenz auf ein anderes Netz bilden. Die Verbindungen zwischen den beiden Netzen, referierendes und referenziertes, werden über synchrone Kanäle gewährleistet. Ein synchroner Kanal besteht aus zwei Transitionen: *Downlink* und *Uplink*. Diese beiden Transitionen können nur gemeinsam schalten. Dazu müssen beide Transitionen aktiviert sein. Sie dürfen dem selben oder zwei unterschiedlichen Netzen angehören. In beiden Fällen können durch den synchronen Kanal beliebige Objekte zwischen den Netzen ausgetauscht werden. Wenn zwei unterschiedliche Netze involviert sind, ist es also möglich, diese zu synchronisieren und auch Objekte zwischen ihnen auszutauschen.[6] Auch Protokolle werden durch dieses Verfahren angesprochen. Durch synchrone Kanäle werden sie gestartet und beendet und auch Nachrichten werden über die synchronen Kanäle vom Agenten an Protokolle übergeben.[7]

[6]Weiterführende Informationen zu Renew, Referenznetzen und synchronen Kanälen können im *Renew User Guide* [KWD01] gefunden werden.

[7]Hierauf wird noch in Abschnitt 3.2 Mulan-Protokolle näher eingegangen.

2.1.3 Mulan

Das verwendete Agentensystem ist Mulan (Multiagentennetze). Es baut auf Renew auf, ist also als ein System von Netzen implementiert. Es hält sich an die offenen Spezifikationen der Foundation of Intelligent Physical Agents (FIPA) [FIP] für Multiagentensysteme.

Das System - so wie es in dem Projekt AOSE [MKR01] verwendet wurde - besteht aus diversen Netzen. Diese sind die Agentenplattform, der Agent und einige Hilfsnetze. Dabei befinden sich die Agenten auf der Plattform, die die Kommunikation zwischen den Agenten gewährleistet. Die Agenten wiederum bestehen auch aus verschiedenen Netzen: dem “eigentlichen” Agenten (Abbildung 2), der Protokollfabrik, der Wissensbasis und den Protokollen. Alle Netze können in beliebig vielen Exemplaren (*Instances*) vorkommen, allerdings hat jeder Agent nur ein Wissensbasisexemplar und nur ein Protokollfabrikexemplar.

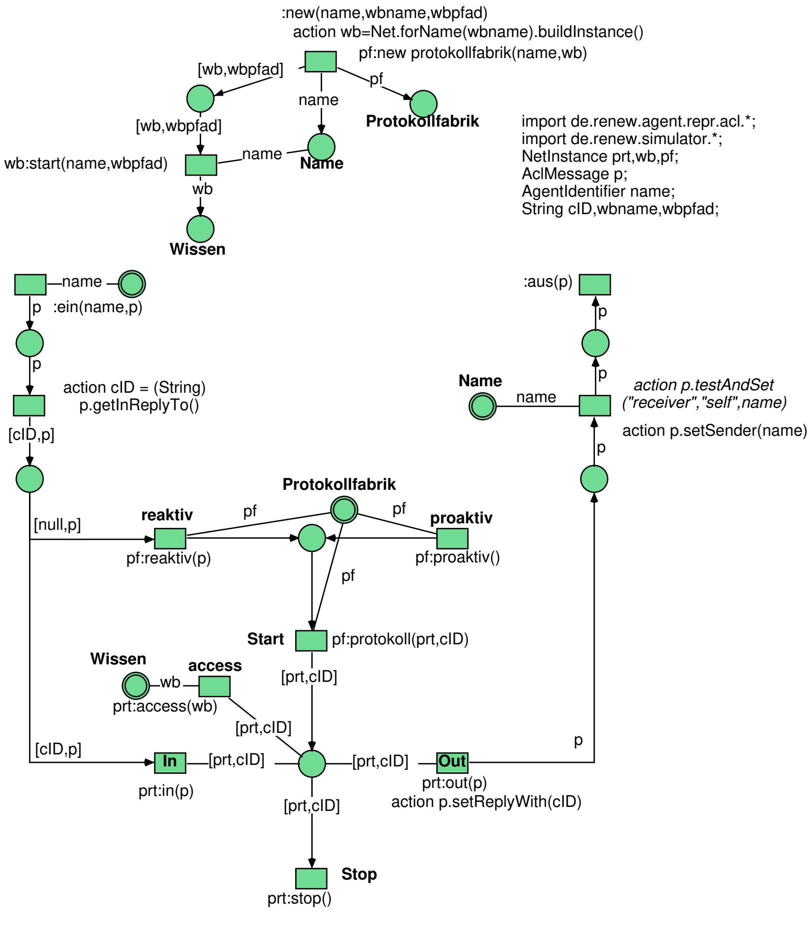

Abbildung 2: Das Netz Agent.

Agenten kommunizieren in Konversationen (*Conversations*) miteinander, sie schicken sich Nachrichten. Diese Nachrichten - auch Performativ (*Performative*) genannt - sind von der FIPA in verschiedene Performativtypen gegliedert. Typische Performativtypen sind *"request"*, *"reject"*, *"inform"*, *"failure"* (Deutsche Entsprechung: Anforderung, Ablehnung, Bestätigung, Fehlschlag). Weiterhin beinhalten die Nachrichten den Sender, den Empfänger, eine Konversationsnummer (*Conversation-ID*) und den Inhalt (*Content*) der Nachricht.

Auf eine Nachricht reagiert ein Agent mittels eines Protokolls. Mit Hilfe der Wissensbasis, in der bestimmte Verhaltensweisen festgelegt sind, wählt die Protokollfabrik das entsprechende Protokoll aus, das den weiteren Verlauf der Konversation bestimmt. Dabei kann das Protokoll die gesamte Konversation bestimmen oder sich die Kontrolle mit anderen Protokollen teilen.[8]

2.2 Begriffe

In diesem Abschnitt werden die Begriffe Agent, Protokoll und Netz-Komponente eingeführt, wie sie in dieser Arbeit verwendet werden.

2.2.1 Agent

Ein Agent ist eine eigenständige, unabhängige Softwarekomponente, die ein eigenes Ziel - alleine oder in einer Gruppe mit anderen Agenten - verfolgt. Agenten kommunizieren über Nachrichten miteinander und können eigenständig handeln. Außerdem können sie ihr Verhalten ändern, sind also adaptionsfähig, und haben ein gewisses Sozialverhalten, deswegen kann man Agenten als intelligent bezeichnen.

Informell können Agenten als Softwareroboter bezeichnet werden.

2.2.2 Protokoll

Ein Protokoll legt ein bestimmtes Verhalten fest. In der agentenorientierten Sichtweise bestimmt ein Protokoll das Kommunikationsverhalten von Agenten. Mulan-Protokolle sind, ebenso wie Mulan-Agenten, Petrinetze. Ein Agent kann beliebig viele Protokolle haben. Der Begriff des Protokolls wird in Kapitel 3 noch ausführlicher behandelt.

2.2.3 Netz-Komponente

Eine Netz-Komponente ist eine Menge von Netzelementen. Die Netz-Komponente sollte aus syntaktisch zusammengehörenden Netzelementen bestehen. Außerdem hat die Netz-Komponente eine visuelle Bedeutung. Sowohl die geometrische Anordnung als auch die Ausrichtung der Netzelemente einer Netz-Komponente sind festgelegt. Die Netz-Komponente schafft, genauso wie sie eine syntaktische Einheit erzeugt, auch eine geometrische. Dadurch erhält die Netz-Komponente einen visuellen Charakter, der sie leicht und eindeutig erkennbar macht. Kapitel 4 geht weiter auf die Netz-Komponenten ein.

---

[8]Mulan, Mulan-Agenten und Mulan-Protokolle werden in der Projektarbeit *Agentenorientierte Implementierung des Spiels "Die Siedler von Catan"* [BGK$^+$02] ausführlich behandelt.

3 Protokolle

In diesem Kapitel wird näher auf die Protokolle eingegangen. Zunächst wird die Funktion der Protokolle beschrieben. In dem darauf folgenden Abschnitt werden die Designprobleme und die hier realisierten Lösungen anhand der Beispiele aus dem Projekt vorgestellt und diskutiert. Die endgültige Designempfehlung wird allerdings noch auf das folgende Kapitel verschoben, da dazu zunächst die Netz-Komponenten benötigt werden. Nichtsdestotrotz soll in diesem Kapitel die Entwicklung der Lösungen nachvollziehbar werden.

3.1 Definition

Die Protokolle bestimmen das Verhalten der Agenten. Zunächst bezieht sich dieses auf das extern sichtbare Verhalten des Agenten. Allerdings können Protokolle auch internes Verhalten beeinflussen. Mit seiner Umwelt, die im allgemeinen aus anderen Agenten besteht, interagiert bzw. kommuniziert ein Agent mit Hilfe von Nachrichten. Damit die Kommunikation geregelt ablaufen kann, wird der Ablauf eines kommunikativen Aktes festgelegt. Dieser geregelte Ablauf wird Konversation genannt und wird mit Protokollen beschrieben. Die Nachrichten, die an andere Agenten geschickt werden, werden von den Protokollen generiert. Eine zusammengehörende Folge von Nachrichten zwischen zwei Agenten bildet also eine Konversation. Protokolle können neue Konversationen starten oder auf Nachrichten reagieren, im letzteren Fall antworten sie innerhalb der Konversation, die schon besteht. Die Nachrichten sind in bestimmten Typen kategorisiert. Exemplarische Typen sind *request* oder *inform*.

Ein Beispiel für eine Konversation ist die Interaktion zweier Agenten, von denen einer einen Service anbietet, der von dem anderen angefordert wird. Letzterer schickt eine Anforderung (*request*) an den Serviceanbieter und erhält bei Erfolg eine Antwort (*inform*) zurück, die als Inhalt die gewünschte Information oder eine Bestätigung des Auftrages enthalten kann. Falls der Serviceanbieter nicht in der Lage ist, den gewünschten Service zu erbringen, kann er den Auftrag ablehnen (*reject*); eventuell mit einer Begründung.

Interaktionen zwischen Agenten werden von der FIPA in Protokolldiagrammen (PD) dargestellt. Dies ist ein Diagramm, das in der für Agenten angepassten AUML (*Agent Unified Modeling Language*) [FIP] festgelegt ist. Die AUML ist eine erweiterte Form der UML [BRJ96]. Protokolldiagramme erweitern die Sequenzdiagramme um einige Elemente. Abbildung 3 zeigt das *FIPA-ContractNet* Protokoll. Das gezeigte Diagramm entspricht im Wesentlichen einem Sequenzdiagramm, wobei die gerichteten horizontalen Kanten den Nachrichten zwischen den Agenten, die gepunkteten vertikalen Linien den Lebenslinien der Agenten und die Kästen deren Aktionen entsprechen. Hinzugekommen sind einige Alternativen beim Senden von Nachrichten. So können in der AUML verschiedene Möglichkeiten des Sendens berücksichtigt werden. Diese entsprechen den Junktoren AND, OR und XOR. Verschiedene Nachrichten bewirken beim Empfänger verschiedene Reaktionen. Abbildung 4 zeigt noch mal die neuen Elemente von Protokolldiagrammen.

Das *FIPA-ContractNet* Protokoll beschreibt den Ablauf des Zustandekommens bzw. Nicht-Zustandekommens einer vertraglichen Verpflichtung. Der Initiator des Vertrages sendet eine Anfrage für einen Vorschlag (*Call For Proposal*) an den Partizipierenden, der diesen ablehnen, mit einem Vorschlag annehmen oder eine "Nicht-Verstanden" Nachricht zurücksenden kann. Auf einen Vorschlag kann der Initiator dann seinerseits mit einer Ablehnung oder einem Annehmen des Vorschlag reagieren, woraufhin der Partizipierende wiederum als Abschluss eine Bestätigung zurückschickt oder einen Fehlschlag meldet.

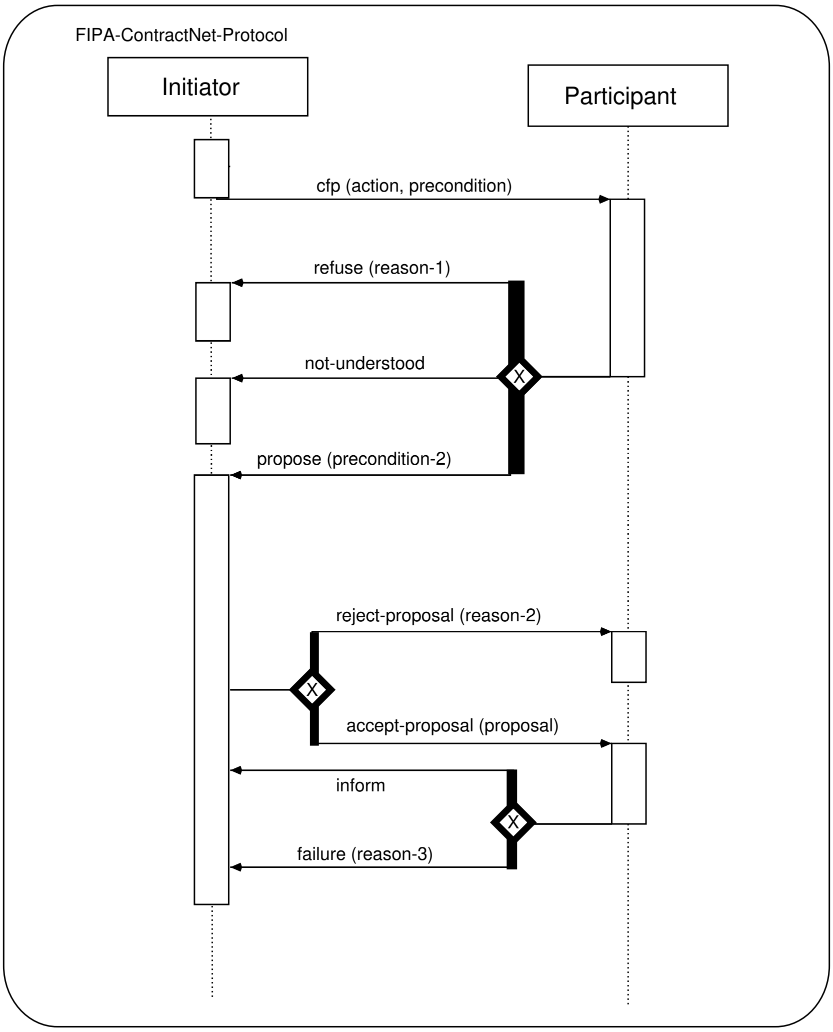

Abbildung 3: *Das FIPA-ContractNet* Protokoll

Interaktionen zwischen Agenten werden von der FIPA [FIP] in Interaktionsprotokollen (IP) festgelegt. Die FIPA definiert mehrere standardisierte Interaktionsprotokolle, die Konversationen beschreiben. Dieser allgemeine Sinn des Begriffes Protokoll bezeichnet also den geregelten Ablauf einer Konversation von den beteiligten Agenten. Diese Auffassung des Begriffs Protokoll kommt dem umgangssprachlichen weitaus näher als die engere Auffassung. Letztere bezeichnet nur das Verhalten eines Agenten zu einem Abschnitt der

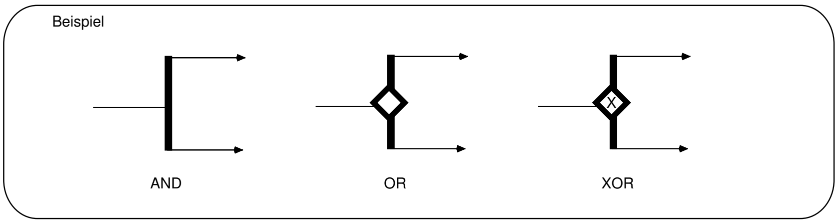

Abbildung 4: Die neuen Elemente in Protokolldiagrammen

Konversation aus seiner Sicht. Das Interaktionsprotokoll soll deswegen zur besseren Unterscheidung weiterhin Konversation genannt werden. Mit Protokoll wird im weiteren Verlauf dieser Arbeit die Variante im engeren Sinne bezeichnet. Ein Protokoll legt demnach fest, wie sich **ein** Agent in (einem Teil) einer Konversation verhält. Für ein Protokolldiagramm heißt das, dass ein Protokoll nur einen Teil der Aktionen auf einer Lebenslinie bezeichnet. Eventuell kann somit ein Protokoll auch nur einer Aktion entsprechen.

Innerhalb einer Konversation müssen die Protokolle verschiedener Agenten, die an dieser Konversation beteiligt sind, miteinander kooperieren, damit eine vernünftige Kommunikation zustande kommen kann. Sie müssen also dem in der Konversation festgelegten Schema folgen.

Gestartete Protokolle behalten die Kontrolle über die von ihnen geführte Konversation, bis sie diese an andere Protokolle abgeben oder bis die Konversation beendet ist. Somit haben die gestarteten Protokolle, ob pro-aktiv oder reaktiv, die Entscheidungsgewalt. Wenn der Agent allerdings eine neue eingehende Nachricht erhält, die nicht zu einer bereits bestehenden Konversation gehört und für die noch kein Protokoll gestartet worden ist, so muss zunächst ermittelt werden, wie der Agent auf diese Nachricht reagiert. Für diese Nachricht muss ein passendes Protokoll ausgesucht und dann gestartet werden. Diese Aufgabe übernehmen die Wissensbasis und die Protokollfabrik. In dieser Phase hat kein Protokoll die Möglichkeit, direkt auf die Auswahl Einfluss zu nehmen. Ist aber ein Protokoll erst einmal gestartet, so übernimmt es die Kontrolle über diese Konversation.

3.2 Mulan-Protokolle

Mulan-Protokolle sind Petrinetze. Diese Netze haben einen klaren, kontrollflussorientierten (*Workflow* ähnlichen) Charakter. Sie haben eine Anfangstransition (Anschrift: *:start()*) und eine oder mehrere Endtransitionen (*:stop()*). Diese Transitionen bestimmen die Lebensdauer der Protokollexemplare. Um Nachrichten von den Agenten entgegenzunehmen, gibt es Eingangstransitionen (*:in(p)*), um Nachrichten zurückzugeben Ausgangstransitionen (*:out(p)*). Diese Transitionen (*:start(), :stop(), :in(p), :out(p)*) sind *Uplinks* und bilden zusammen mit den entsprechenden *Downlinks* in den Agenten-Netzen die synchronen Kanäle zwischen diesen Netzen. Protokolle werden durch sie gestartet und beendet. Außerdem werden durch die mit *:in(p)* und *:out(p)* beschrifteten *Uplinks* die Nachrichten p zwischen den Agenten und den Protokollen ausgetauscht. Der Agent erhält die von einem anderen Agenten an ihn gesendete eingehende Nachricht und übergibt sie zur weiteren Bearbeitung an das entsprechende Protokoll. Eine ausgehende Nachricht wird von den Protokollen generiert und an die Agenten übergeben, damit sie von diesen an den entsprechenden Empfänger gesendet werden können.

Das Protokoll bestimmt durch seinen Kontrollfluss das Verhalten des Agenten in der Konversation, für das es "verantwortlich" ist. Es entscheidet also, wie eine Nachricht verarbeitet wird, an welchen Agenten welche Nachricht verschickt wird und welche internen Aktionen stattfinden. Dennoch sind es die Agenten, die die Nachrichten von anderen Agenten empfangen oder Nachrichten an andere Agenten verschicken. Sie entscheiden auch, mit Hilfe ihrer Wissensbasis und der Protokollfabrik, welches Protokoll gestartet wird.

Abbildung 5 beschreibt beispielhaft das Ablaufschema eines einfachen Mulan-Protokolls. In der Darstellung werden die in Renew benutzten Anschriften für die *Uplinks* benutzt. Die Flussrichtung geht von links nach rechts. Das Protokoll wird zunächst gestartet, dann erhält es eine Nachricht p. Als nächstes könnten noch einige Aktionen stattfinden, die in diesem Schema nur durch eine Transition angedeutet werden. Dann wird eine Nachricht verschickt und das Protokoll beendet.

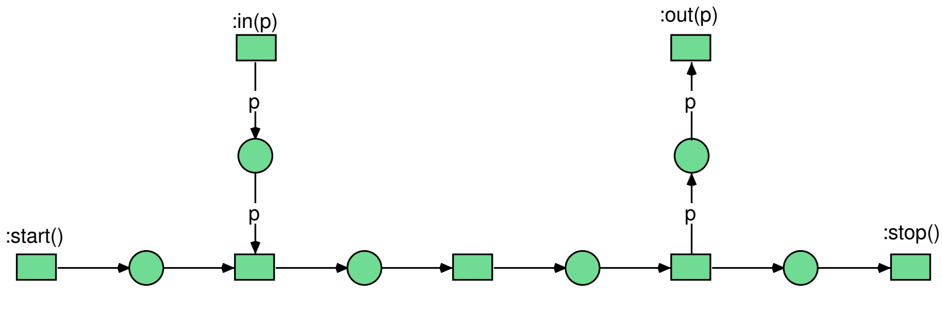

Abbildung 5: Ein Protokollschema.

3.3 Probleme der Implementation

Auch für Mulan-Protokolle kann es sinnvoll sein, die Spezifikation der Konversation mit Hilfe von Protokolldiagrammen zu erstellen. Ganz besonders deswegen, weil beide Repräsentationen, Protokolldiagramm und Petrinetz, graphisch sind. Ist der Ablauf einer Konversation erst einmal festgelegt, so können jeweils für einen Agenten ein oder mehrere Protokolle erstellt werden, die die Aufgaben des Agenten bezüglich dieser Konversation regeln. Für Mulan-Protokolle heißt das, dass ein Petrinetz zu zeichnen ist, das den Anforderungen des Systems und der Spezifikation entspricht.

Ein Protokoll wird üblicherweise sequentiell gezeichnet. Da es sich bei Protokollen um Abläufe handelt, ist diese Vorgehensweise, dem Prozess zu folgen, sicherlich angebracht. Dennoch werden oft gleichartige Teile von Netzen benötigt, da immer wieder gleichartige Aufgaben zu erfüllen sind. Es liegt also nahe, diese redundante Arbeit dadurch zu verkürzen, dass Komponenten entwickelt werden, die diese Aufgaben erfüllen. Darüber hinaus bringt das sequentielle Vorgehen noch weitere Probleme mit sich.

Gleichartige Aufgaben von Protokollen sehen nicht immer gleich aus. Dafür gibt es verschiedene Gründe. Einerseits werden Aufgaben vermischt, wodurch zusammengesetzte Funktionen entstehen, die zwar sehr kompakt, aber auch hochkomplex sein können, wodurch die Verständlichkeit der Netze erschwert wird. Andererseits passiert es immer wieder, dass Teile der Netze einfach anders gezeichnet werden. Gründe dafür gibt es viele: Platzbeschränkungen, keine Richtlinien, etc. Als Beispiel[9] soll hier die unterschiedliche Anordnung von zwei Konflikten in einem Netz dienen. Abbildung 6 zeigt das Protokoll des Agenten *Insel* zum Bauen eines Bauwerks. Eindeutiger wäre eine gleichartige Anordnung der Konflikte, dann würde die Gleichartigkeit der Aufgabe sofort erkannt werden. Es ist also wünschenswert, eine konsistente Darstellung von ähnlichen Aufgaben zu erreichen. Die beiden Konflikte wurden in dieser Abbildung zur Verdeutlichung zusätzlich farbig unterlegt.

Projekt-Übergreifend war die Art der Darstellung bezüglich ähnlicher Aufgaben noch vielfältiger, so dass noch weniger Konsistenz erreicht wurde. Abbildung 7 zeigt zwei Varianten für das Verschicken einer Nachricht mit anschließendem Warten auf Antwort (bereinigt von den Anschriften). Die beiden Varianten sind von unterschiedlichen Gruppen erstellt worden. Nur bei näherer Untersuchung ist die Ähnlichkeit der Aktion zu erkennen.

Sinnvolle Beschriftungen von Kanten und Knoten sind sehr nützlich, dennoch gibt es viele Probleme mit diesen. Transitionsanschriften neigen dazu, sehr groß und komplex zu sein. Bei mehreren langen Anschriften an Transitionen oder Kanten ist es schwierig diese eindeutig zu positionieren oder zuzuordnen, da Anschriften länger als Kanten werden können oder über diese hinweg laufen. Abbildung 8 zeigt einen Ausschnitt aus dem schon erwähnten Protokoll *Insel_Bau* (Abbildung 6). Da der Platz auf einem Bildschirm für ein Netz sehr begrenzt ist, kann es nicht beliebig groß gemacht werden, so dass letzten Endes ein Kompromiss zwischen zwei Kriterien gefunden werden muss. Zum einen muss der Platzmangel berücksichtigt werden, zum anderen dürfen Netze auch nicht zu eng gezeichnet werden. In beiden Fällen wird die Übersicht verschlechtert; einmal durch die zu große, das

[9] Alle Beispielprotokolle und Ausschnitte sind in dem Projekt AOSE [MKR01] entstanden.

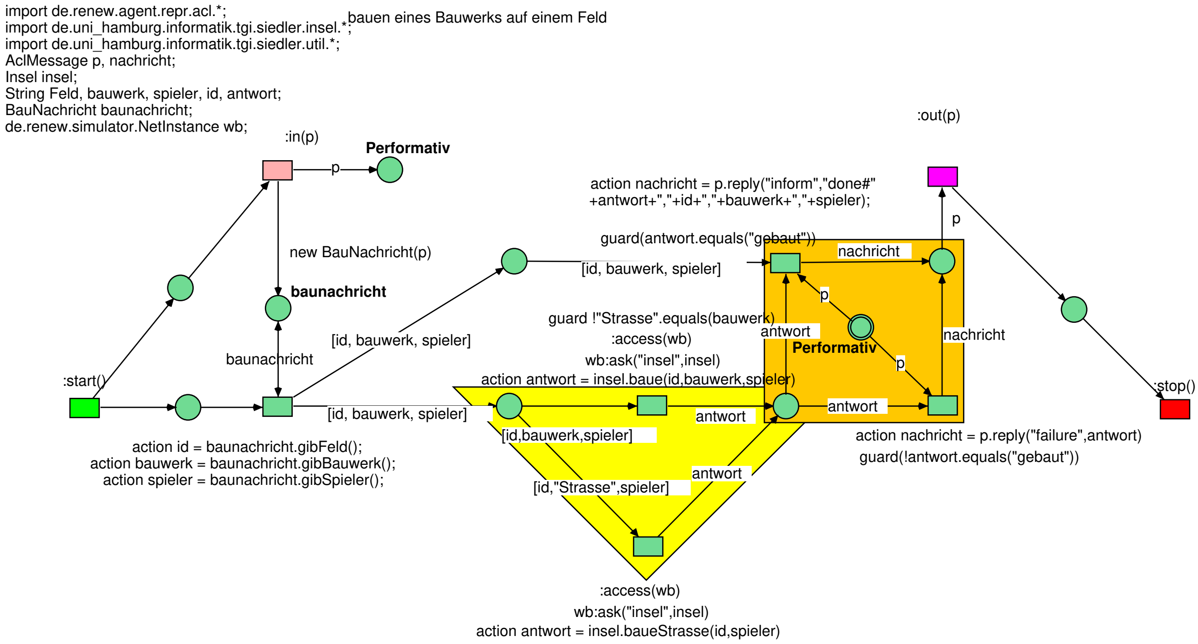

Abbildung 6: Das Protokoll *Insel_Bau* mit farbig gekennzeichneten unterschiedlichen Anordnungen von Konflikten.

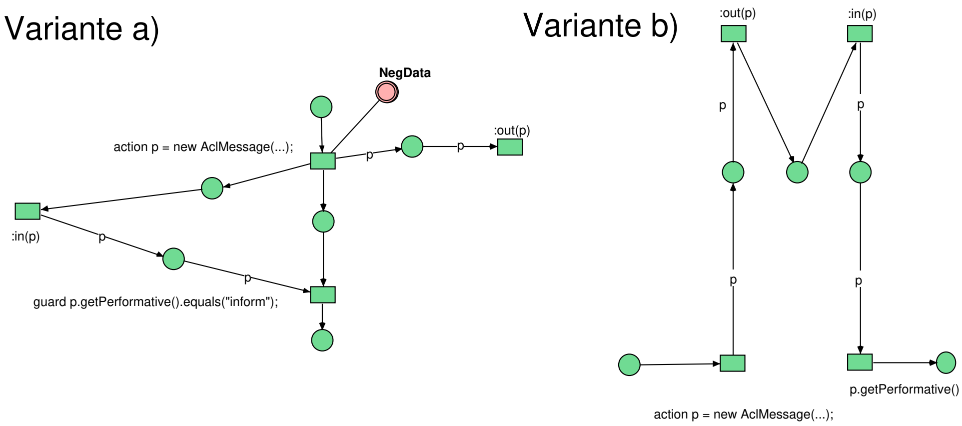

Abbildung 7: Zwei unterschiedliche Darstellungen der gleichen Aktion.

andere Mal durch die zu kleine Informationsdichte.

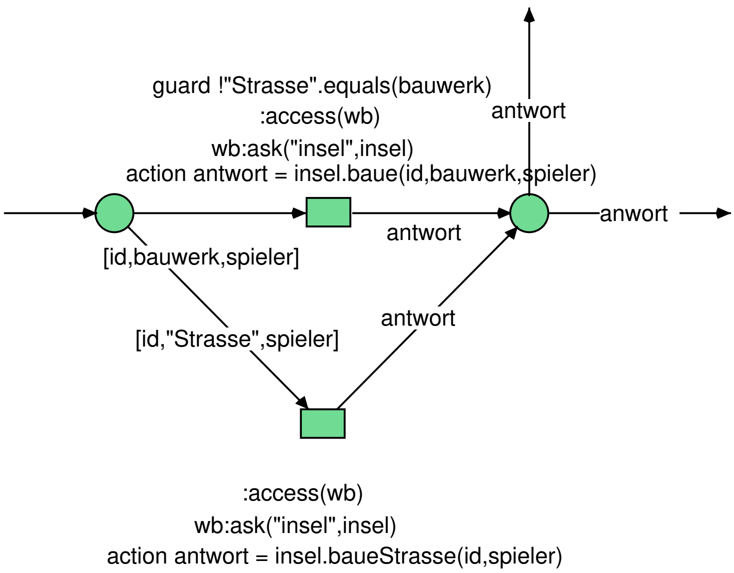

Abbildung 8: Ausschnitt aus dem Protokoll Insel_Bau.

Ein weiteres Problem ist die Vermischung von Aufgaben. In Abbildung 8 ist zu sehen, dass an der oberen Transition drei verschiedene Aktionen durchgeführt werden. 1. Die Bedingung wird geprüft, damit der Konflikt eindeutig aufgelöst werden kann. 2. Aus der Wissensbasis wird das Objekt *insel* ermittelt. 3. Die Methode *insel.bau()* wird aufgerufen. Diese Zusammenfassung von verschiedenen Aufgaben führt dazu, dass eine Modularisierung der Netze nur schlecht möglich ist. Der entstandene Code ist zwar sehr kompakt, allerdings auch sehr dicht, was die Lesbarkeit nicht unbedingt fördert.

Es stellt sich also die Frage, wie das Design von Protokollen benutzerfreundlich gestaltet werden kann. Um dies zu klären, werden zunächst die Designregeln für Petrinetze betrachtet.

3.4 Ergonomie-Regeln für Petrinetze

Es gibt viele Möglichkeiten Petrinetze zu zeichnen, und in vielen Fällen ist es eine gute Idee, die Form an das zugrunde liegende Problem anzupassen. So ist es z.B. sinnvoll, das Philosophenproblem kreisförmig darzustellen, weil die Philosophen um einen runden Tisch herumsitzen. Dies fördert die Anschauung des Netzes, allerdings ist dieses Problem eher ein abstraktes Modell, als ein ablauffähiges Programm. In [Jen96] beschreibt Kurt Jensen, dass für das Design von "vernünftigen" Petrinetzen viel Erfahrung notwendig ist. Er schlägt 10 grundsätzliche Regeln für das Zeichnen von Petrinetzen vor.

1. Sämtliche Eingangskanten sollten auf einer Seite des Knotens gesammelt werden. Sämtliche Ausgangskanten auf der gegenüberliegenden Seite.

2. Elemente (Knoten und Kanten), die zusammengehören, sollten in einer ansprechenden geometrischen Figur angeordnet werden - zum Beispiel Kreise, Ellipsen oder Rechtecke. Diese Figuren sollten möglichst hervorgehoben werden.

3. Es sollte eine einheitliche Flussrichtung eingehalten werden. Von oben nach unten oder von links nach rechts.

4. Kanten sollten sich nicht überkreuzen. Falls dies doch notwendig sein sollte, dann sollte der Winkel der Kreuzung möglichst rechtwinklig sein.

5. Kanten sollten möglichst einfach und gleichmäßig sein.

6. Parallele Kanten sollten nicht zu nah nebeneinander gelegt werden.

7. Gestrichelte Kanten sollten korrekt gemalt werden.

8. Formen von Objekten sollten sich eindeutig unterscheiden.

9. Beschriftungen sollten sich so nah wie möglich an dem dazugehörigen Element befinden.

10. Sämtliche Stilmittel sollten konsistent angewendet werden.

Mit Hilfe der Regeln Figuren (2) und Konsistenz (10) ließen sich die im vorangehenden Abschnitt beschriebenen Probleme teilweise lösen. Wie allerdings die Figuren aus Regel (2) zu realisieren sind, wird von Jensen nicht festgelegt. Weiterhin gibt es auch keine Hinweise, wie es möglich ist, die Stilmittel konsistent anzuwenden (Regel 10).

Ein Ziel für die Gestaltung der Protokolle ist es ihren Fluss (Kontrollfluss) so sichtbar wie möglich zu gestalten und störende Elemente weitgehend zu eliminieren. Um dieses zu erreichen, wird in dem folgenden Abschnitt vorgeschlagen, den Kontrollfluss vom Datenfluss strikt zu trennen.

3.5 Trennung von Daten- und Kontrollfluss

Daten und Kontrollfluss werden in Renew[10] nicht getrennt. Dies liegt an dem Konzept der Petrinetze und der Lokalität der Variablen. Zum einen wird bei gefärbten Netzen oder Referenznetzen das Datum zur Marke, somit verschmelzen die beiden Konzepte miteinander. Zum anderen ist es aber auch wünschenswert, alle Daten nur lokal zu erfassen, da somit die Nebenläufigkeit der Petrinetze nicht beeinträchtigt wird. Ein rein praktisches Problem entsteht, wenn viele Daten gehalten bzw. weitergegeben werden müssen, denn

[10] Bei Petrinetzen wird im allgemeinen keine Unterscheidung zwischen Markierungen aus anonymen oder benannten Marken gemacht.

dann können die Anschriften an den Kanten sehr lang werden. Somit geht viel Übersichtlichkeit verloren, weil die Netze entweder zu groß oder zu komplex werden. Außerdem kann der Kontrollfluss nicht mehr leicht erkannt werden, wenn viele Kanten parallel zum Kontrollfluss Daten weiter reichen.

Die Projektteilnehmer des Projektes AOSE entwickelten eine sehr nützliche Methode die Übersichtlichkeit zu erhöhen. Da viele Informationen an mehreren Stellen der Netze zur Verfügung stehen mussten, wurden die Daten schließlich nicht mehr über Kanten zu den Transitionen "geleitet". Stattdessen wurden oft virtuelle Stellen eingesetzt, um die Daten an den entsprechenden Positionen, an denen sie benötigt wurden, zur Verfügung zu haben. Es wurde schnell klar, dass dieses Verfahren sehr effizient ist, und somit schließlich von allen Gruppen eingesetzt. Allerdings ist es meistens nur für einen Teil der Daten verwendet worden und die Darstellung bzw. Realisierung dieses Verfahrens war sehr unterschiedlich. Abbildung 9 zeigt einen Ausschnitt aus dem Protokoll *anmelden* der Spielleitung. Die Daten *spieler* und *wb* werden in den (virtuellen) Stellen mit den Bezeichnungen Spieler und Wissen gehalten.

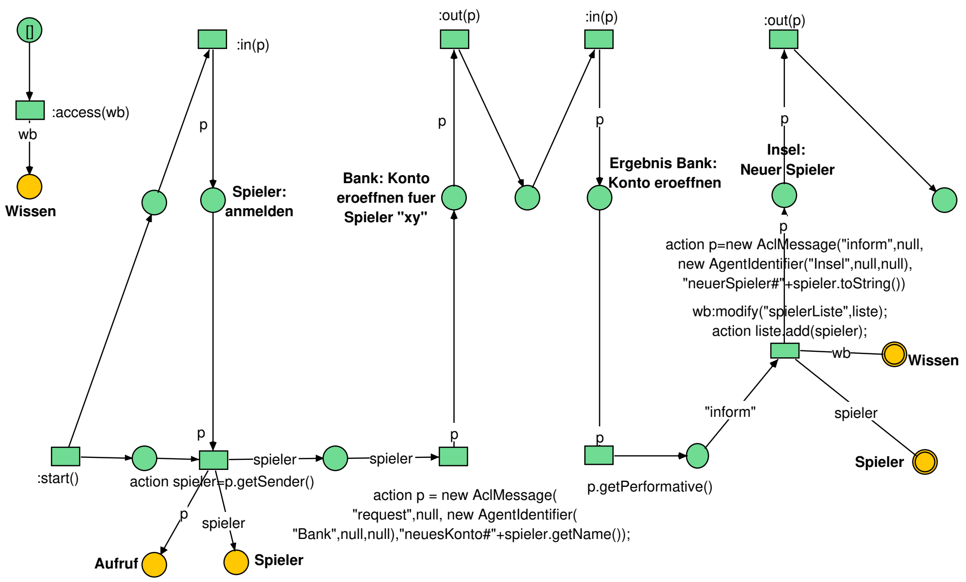

Abbildung 9: Ausschnitt aus dem Protokoll *anmelden* der Spielleitung.

Ein weiteres Problem ergibt sich aus der Lokalität der Variablen. Zwei Marken des gleichen Typs mit unterschiedlicher Bedeutung können nicht in eine Stelle gelegt werden, da es nicht möglich ist, diese voneinander zu unterscheiden. Um also zwei Marken gleichen Typs weiterzugeben, ist es nötig, entweder diese in verschiedene Stellen unterzubringen oder nicht die Marken selbst, sondern ein Tupel aus den Marken zu übergeben (vergl. Abbildung 8). Beide Lösungen führen zu mehr Unübersichtlichkeit, da entweder die Kantenbeschriftungen sehr lang werden oder die Anzahl der Kanten vermehrt wird.

Der Lösungsansatz für diese Probleme ist die Trennung von Daten und Kontrollfluss.

Dazu werden die Daten gesondert behandelt.[11] Dies erzeugt zwar einen zusätzlichen Aufwand (*Overhead*), der aber durch bessere Übersichtlichkeit ausgeglichen wird. Hierzu werden die virtuellen Stellen von Renew eingesetzt. Ähnlich einem Deklarationsteil werden die Stellen oberhalb des Netzes gesammelt. Weiterhin sind die datenhaltenden Stellen farblich von den anderen Stellen zu unterscheiden.[12] Ein Vorteil ist, dass die Daten nun nicht mehr über ggf. lange Kanten "transportiert" werden müssen, die einen nicht geringen Teil zur Unübersichtlichkeit beitragen. Die datenhaltenden Stellen erhalten alle einen Namen. Dadurch können die Variablennamen sehr kurz gehalten werden. Da die Variable immer lokal aus den benannten datenhaltenden (virtuellen) Stellen genommen wird, ist keine Verwechslung möglich. So werden nicht nur die Kantenanschriften kürzer, sondern auch die Anschriften an den Transitionen. Da viele Java-Konstrukte dazu tendieren, sehr lang zu werden, ist das Verkürzen dieser wünschenswert und wird durch kurze Bezeichner erreicht.

Um den Kontrollfluss sichtbar zu machen, sollen zwischen den Netz-Komponenten keine Daten übertragen werden. Dies scheint diskussionswürdig und es stellt sich die Frage, wie strikt diese Regel eingehalten werden kann oder muss. Gerade bei der Übergabe von Daten zwischen aufeinander folgenden Netz-Komponenten könnte eine Ausnahme auch nützlich sein. Dennoch wird hier versucht das Verfahren strikt anzuwenden, um eine möglichst lose Koppelung zwischen den Netz-Komponenten zu erhalten. Gleichzeitig wird der generische Charakter der Netz-Komponenten gefördert.

3.6 Lösungsansatz

Um eine einheitliche Form von Mulan-Protokollen zu erhalten, lassen sich einige der bisher aufgeführten Punkte zusammenfassen. Netze sollen eine einheitliche Flussrichtung haben (Jensens Regel 3). Es wäre sinnvoll, wenn diese Regel nicht nur für einzelne Netze sondern für ein ganzes Projekt gelten würde. In dieser Arbeit werden, um eine Vereinheitlichung anzustreben, die Mulan-Protokolle von links nach rechts laufen.[13]

Um die Konsistenz der Elemente (Regel 10) zu erfüllen, werden drei Punkte vorgeschlagen, die beim Zeichnen von Teilen von Netzen beachtet werden sollen:

- Gleiche Funktionalität soll auch in einer gleichen geometrischen Form dargestellt werden (bzw. ähnliche und zusammengehörende Aufgaben sollen in einer ähnlichen Form dargestellt werden).

- Unterschiedliche Aufgaben sollen sich auch in ihrer Darstellung klar voneinander unterscheiden und voneinander getrennt werden.

[11] Abschnitt 4.4.1 wird das Protokoll *anmelden* noch einmal aufgreifen und die Realisierung der Trennung verdeutlichen.

[12] In dieser Arbeit werden die datenhaltenden Stellen orange dargestellt. Außerdem werden diese Stellen grundsätzlich benannt, so dass neben ihnen der Name der Variablen steht. Somit kann auch in einem Schwarz-Weiß Druck erkannt werden, welche Stellen gemeint sind.

[13] Es gibt mehrere Argumente für beide, die horizontale und die vertikale Ausrichtung. Die Festlegung ist willkürlich und begründet sich damit, dass die horizontale Ausrichtung auch die primäre Lesrichtung darstellt und dass ein Bildschirm breiter als hoch ist.

- Wiederkehrende Aufgaben sollen identifiziert und generalisiert werden, so dass die Effizienz beim Erstellen von Protokollen gesteigert werden kann.

Gleichzeitig sollte bei dem Erstellen von Teilen der Netze oder Netz-Komponenten die Regel der geometrischen Anordnung der Netzelemente (2) beachtet werden. Falls die geometrische Form der Netz-Komponenten eindeutig ist, können sie auch durch ihre Form identifiziert werden.

4 Netz-Komponenten

Ziel ist es, ein brauchbares Verfahren zu erstellen, mit dem ein Entwickler oder ein Entwicklerteam in der Lage ist ein Softwaresystem auf der Basis von Mulan schneller als bisher zu implementieren. Hier soll der Ansatz verfolgt werden, das Entwickeln und Zeichnen der Mulan-Protokolle zu erleichtern und letztere sowohl zu vereinheitlichen als auch übersichtlicher zu gestalten. Um dieses zu erreichen, werden Netz-Komponenten eingesetzt. In diesem Kapitel soll zunächst auf die Anforderungen an die Netz-Komponenten eingegangen werden, dann werden die Netz-Komponenten vorgestellt und als Abschluss soll ein Anwendungsbeispiel folgen.

4.1 Anforderungen

In diesem Abschnitt werden die Anforderungskriterien für Netz-Komponenten beschrieben. Die Netz-Komponenten sollen einen generischen Charakter haben, leicht miteinander verknüpfbar sein, eine syntaktische Einheit bilden und eine ansprechende und intuitiv erfassbare Form einnehmen.

4.1.1 Generischer Charakter

Wie in Kapitel 3 beschrieben müssen Mulan-Protokolle verschiedene Teilaufgaben immer wieder erfüllen, dies bedeutet, dass in der Implementation sehr viel redundante Arbeit geleistet wird. Nun ist es gängige Praxis in der Softwareentwicklung - und auch nahe liegend, dass diese Redundanz durch ein automatisiertes und standardisiertes Verfahren abgelöst oder der Aufwand für die wiederkehrende Arbeit reduziert wird. Komponenten können diese Anforderung erfüllen, sie müssen allerdings einen generischen Charakter haben, um flexibel in verschiedenen Netzen eingebunden werden zu können.

4.1.2 Verknüpfungen

Die Komponenten der Netze sollen in einer einfachen und flexiblen Weise miteinander und mit anderen Teilen des Netzes verknüpfbar sein. Dazu müssen von den Komponenten die jeweiligen Schnittstellen bekannt sein. Einheitliche Schnittstellen können den Aufwand beim Benutzen von neuen Komponenten erleichtern. Hier werden die Schnittstellen zwischen den Netz-Komponenten fast ausschließlich durch asynchrone Kanäle realisiert, d.h. zwischen zwei aufeinander folgenden Netz-Komponenten befindet sich nur eine Stelle. Weiterhin wird der Ansatz verfolgt, dass diese Verbindungsstelle keine Daten übergeben soll, also nur anonyme Marken enthalten darf.

4.1.3 Syntaktische Einheitlichkeit

Jede Netz-Komponente soll eine wohldefinierte Aufgabe erfüllen, damit sie in verschiedenen Kontexten eingesetzt werden kann und ihre Aufgabe trotzdem klar bleibt. Die Aktivitäten der Komponenten sollten eine syntaktische Einheit bilden, also in sich und nach außen abgeschlossen sein. Somit könnte auch erreicht werden, dass möglichst wenig an den Komponenten verändert werden muss um sie einzusetzen. Es wäre ideal, wenn beim Einbinden der Komponenten, zusätzlich zu den Schnittstellen, nur eine Anschrift geändert werden muss.

4.1.4 Darstellung

Visuell sollen die Netz-Komponenten leicht erkennbar sein und eine einprägsame und intuitiv erfassbare Form besitzen. Um das zu erreichen, können die Netz-Komponenten entweder, nach Jensens Regel (2), eine klare geometrische und einprägsame Form haben, oder sie kann ein Symbol bzw. eine Metapher bilden, wodurch sich die Aufgabe der Netz-Komponente schon erschließen lassen kann. Sie sollen eine geometrische Einheit bilden und Zusätzlich auch einen knappen "sprechenden" Namen haben.

4.2 Netz-Komponenten konkret

Netz-Komponenten sollen sich einerseits in ihrer Darstellung voneinander unterscheiden, andererseits einem einheitlichen Schema folgen. Im Petrinetz-Formalismus gibt es bereits die Möglichkeit, bestimmte Netzelementmengen zusammenzufassen, diese nennt man Vergröberung. Es gibt Stellen- und Transitionsvergröberungen. Zwei entscheidende Argumente sprechen hier für die Benutzung der letzteren. 1. Netz-Komponenten sollen Aktivitäten beschreiben, sind also Aktionen. In Petrinetzen sind die Transitionen die aktiven Elemente. 2. Transitionsberandete Netz-Komponenten lassen sich leicht auf zwei Weisen miteinander verbinden: Durch einen asynchronen oder durch einen synchronen Kanal. Beide Möglichkeiten der Verbindung von transitionsberandeten Netz-Komponenten lassen sich leicht realisieren. Der asynchrone Kanal wird mit einer einfachen Stelle zwischen den Komponenten realisiert, die mit zwei Kanten die beiden verbindet. Für den synchronen Kanal müssen nur die Transitionsanschriften an die beiden zu verbindenden Transitionen geschrieben werden.

Bei der Realisierung der Netz-Komponenten wurden die Schnittstellen zwischen diesen schon zu den Netz-Komponenten hinzugefügt. Die hinzugefügten Stellen erleichtern das Verbinden der Netz-Komponenten miteinander, dennoch gehören sie nicht zur eigentlichen Netz-Komponente.[14]

4.3 Eine Basis für Netz-Komponenten

In diesem Abschnitt werden nun die einzelnen Netz-Komponenten vorgestellt. Die Sammlung soll möglichst umfassend sein, d.h. es soll angestrebt werden, dass mit diesen Netz-Komponenten alle erforderlichen Aufgaben von Mulan-Protokollen erfüllt werden können.[15]

[14] Netzelemente, die zu den Netz-Komponenten gehören, werden grau dargestellt.

[15] Die Bezeichnungen der Netz-Komponenten lehnen sich, soweit dies möglich ist, an die von van der Aalst benutzten Bezeichnungen für *Workflow-Pattern* [vdA00] an.

Es wird zunächst anhand des Konditionals der Entscheidungsprozess, der Schließlich zur hier gezeigten Darstellung der Netz-Komponenten geführt hat, beschrieben. Dieser exemplarische Entscheidungsprozess wird anschließend für die anderen Netz-Komponenten nicht ausführlich dargelegt, da sich dieser ständig wiederholt. Dennoch ist zu sagen, dass für viele Netz-Komponenten diverse Formen in betracht gezogen wurden und durch Ausschluss schließlich dieses Ergebnis entstand. Kriterien für den Ausschluss der verworfenen Formen waren Modularität, Konformität und Anschaulichkeit.

Es werden drei verschiedene Gruppen unterschieden:

- Kontrollstrukturen: Konditional, Zusammenführung, Parallelverarbeitung, Schleifen und Semaphore
- Datenaustauschfunktionen: Eingang und Ausgang von Nachrichten
- Elementare Strukturen: Beginn und Ende der Protokolle

4.3.1 Kontrollstrukturen

Konditional *(NC cond)*

Das Konditional - das Schlüsselwort in den Programmiersprachen ist für gewöhnlich *if* - ist die erste hier vorgestellte Netz-Komponente. Sie wird hier voran gestellt, da ihre Funktion allgemein bekannt ist. Zunächst wird bei dieser Netz-Komponente exemplarisch der Entscheidungsprozess anhand einer Diskussion zweier alternativer Lösungen illustriert.

Bei einem Konditional soll eine Entscheidung über den weiteren Verlauf des Programms getroffen werden. Bei Petrinetzen wird dieses auch Konflikt genannt. Dieser Konflikt wird bei einem Konditional durch den Entscheidungsträger aufgelöst. Dies ist im Allgemeinen ein Bool'scher Wert oder ein (oder mehrere) Wächter (*Guard*).

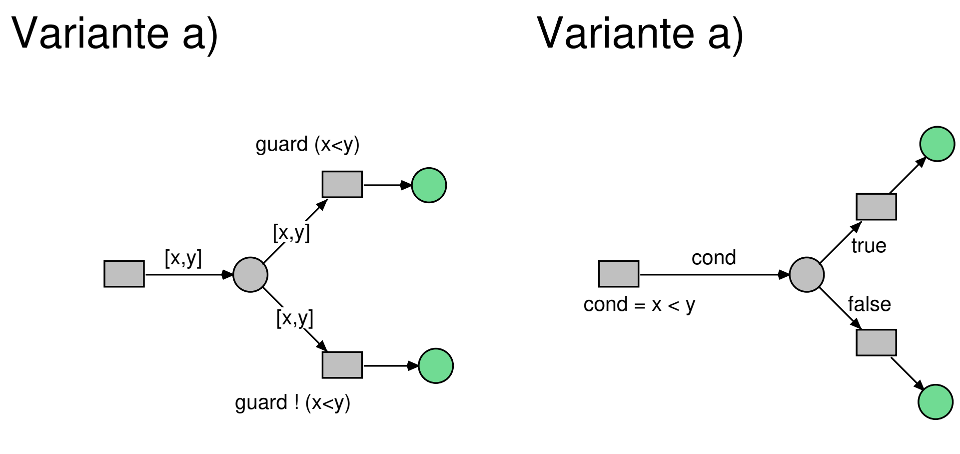

Abbildung 10: Zwei Varianten der Netz-Komponente Konditional *(NC cond)*.

Abbildung 10 zeigt zwei alternative Ansätze. Version a) ist ein wenig kompakter als Version b), hat allerdings den Nachteil, dass zwei Wächter benötigt werden, wobei die beiden Bedingungen der Wächter zueinander invers sind. Dies bedeutet aber, dass der Benutzer dieser Netz-Komponente zwei Beschriftungen anpassen muss, wohingegen es bei Variante b) ausreicht, die Bedingung einmal an die Variable *cond* zu binden.

Die Entscheidung für Variante b) liegt also nahe.

Das klassische Symbol für das Konditional in Flussdiagrammen ist die Raute. In Anlehnung an dieses bekannte Symbol bildet der hintere Teil der Netz-Komponente ein offenes Dreieck. Zusammen mit der Zusammenführung *(NC ajoin)* bilden beide Symbole die Raute. Außerdem ist diese Form der klassische Konflikt in Petrinetzen.

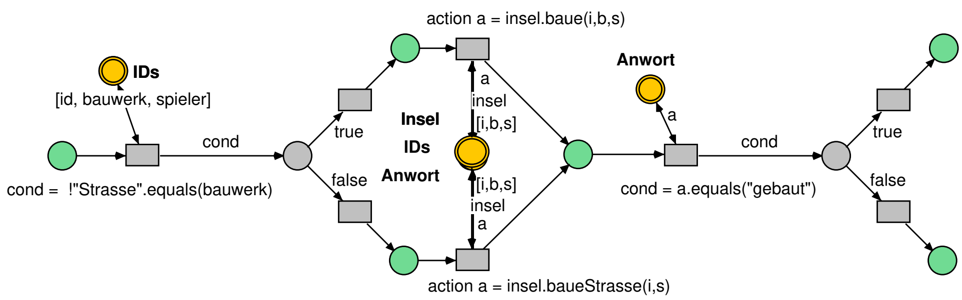

Abbildung 11: Ein Anwendungsbeispiel für *(NC cond)*.

Als Beispiel, wie das Konditional verwendet werden kann, wird hier gezeigt wie das entsprechende Teil des Protokolls aus dem Abschnitt 3.3 mit der Netz-Komponente aussieht. Abbildung 11 zeigt einen Ausschnitt aus dem modifizierten Protokoll *Insel_Bau*.[16] Um diese Netz-Komponente in ein bestehendes Netz einzubinden wird sie an die entsprechende Stelle gezeichnet, dann mit einer eingehenden Kante versehen und die Anschrift an der Transition geändert. Zusätzlich ist es noch erforderlich benutzte Variablen zur Verfügung zu stellen. Dazu werden virtuelle Stellen erzeugt, die mit Reservierungskanten an die betreffenden Transitionen angebunden werden.

Zusammenführung bei Konflikten *(NC ajoin)* (Asynchronous Join)

(NC ajoin) ist sehr einfach und wird als Ergänzung zum Konditional (*NC cond*) benötigt.

Die Form ist an das Konditional angelehnt. Als Komplement zum Konflikt liegt es nahe dies spiegelbildlich auszudrücken. Zusammen bilden Konflikt und Zusammenführung eine Raute.

Parallelverarbeitung/ Nebenläufigkeit *(NC psplit)* *(Parallel Split)*

NC psplit benötigt keine Anschriften und besteht nur aus einer Transition.

Die Form ist so gewählt, damit sie sich klar von der des Konditionals *(NC cond)* unterscheidet.

[16] Vergleiche Abbildung 6.

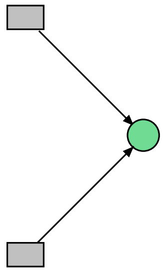

Abbildung 12: Die Netz-Komponente Zusammenführung *(NC ajoin)*.

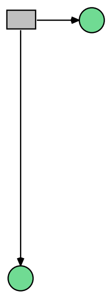

Abbildung 13: Netz-Komponente Nebenläufigkeit *(NC psplit)*.

Zusammenführung bei Nebenläufigkeit/ Synchronisation *(NC pjoin)* *(Parallel Join)*

Die Netz-Komponente *NC pjoin* ist sehr einfach und wird als Ergänzung zu *NC psplit* benötigt.

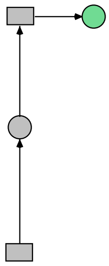

Abbildung 14: Netz-Komponente Synchronisation *(NC pjoin)*.

Die Form ist an die von *NC psplit* angelehnt.

Iterator *(NC iterator)*

NC iterator ist eine klassische Schleife. Das was eine *for*-Schleife für ein Feld leistet, erfüllt der *Iterator* sehr komfortabel für eine *Collection*[17]. Eine Sequenz wird für alle iterierten Elemente der *Collection* durchlaufen. Dies kann synchron oder asynchron geschehen. Die einzige Anschrift, die verändert werden muss, ist die Generierung des *Iterators it*. Zusätzlich muss allerdings darauf geachtet werden, dass ein *Iterator* immer Java-Objekte des Typs *Object* zurückgibt. Es muss dann ggf. auf den entsprechenden Typus ein *cast*[18] stattfinden.

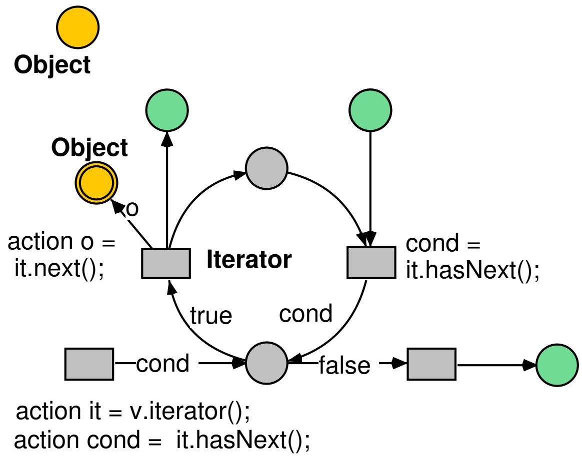

Abbildung 15: Die Netz-Komponente iterator *(NC iterator)*

Der Iterator beschreibt eine Schleife oder einen Kreislauf, deswegen liegt ein rundes Gebilde nah. Die oberen beiden Stellen bilden die Schnittstellen zu dem Schleifenkern. Wobei die linke und die rechte den Anfang und das Ende des Schleifenkerns entsprechen.

Für alle Elemente eines Feldes *(NC forall)*

Bei der Netz-Komponente *NC forall* wird für alle Elemente eines Feldes (*Array*) eine Sequenz gestartet. Allerdings werden im Gegensatz zur *NC iterator* diese Sequenzen nicht der Reihe nach, sondern nebenläufig, abgearbeitet. Dennoch kann auch hier sichergestellt werden, dass alle Sequenzen beendet worden sind.

Diese Form ist dem Symbol des Allquantors nachempfunden. Sie ist nicht so anschaulich wie die Form von *NC iterator*, dennoch kann ihre Bedeutung - einmal verstanden - sofort wiedererkannt werden.

Semaphore *(NC semaphore)*

NC semaphore entspricht einem Zähler und gewährleistet einen Ausschluss. Dies ist

[17] *Iterator*, *Collection* und *Object* sind Java Objekte.

[18] Ein *casting* ist eine Spezialisierung von Objekten.

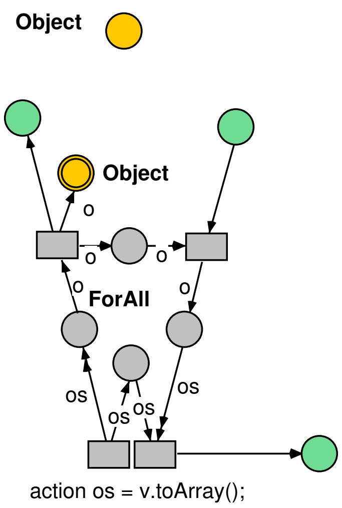

Abbildung 16: Die Netz-Komponente für alle Elemente eines Feldes *(NC forall)*.

insbesondere notwendig, weil es nicht möglich ist, im Normalbetrieb von Renew Inhibitorkanten einzusetzen. Somit ist dies nur ein *Work-Around* für die letzteren.

Für die Realisierung dieser Netz-Komponente wurden synchrone Kanäle aus zwei Gründen eingesetzt. Zum einen ist es somit möglich eine Aktion auszuführen - also eine Transition schalten zu lassen - und gleichzeitig den Zähler zu inkrementieren bzw. dekermentieren, ohne in die eigentliche Aktion direkt einzugreifen. Zum anderen kann ein Vorgang der an einer anderen Stelle des Netzes stattfindet ohne lästige Kanten, die quer durch das Netz laufen, auf die Semaphore Einfluss nehmen. Durch die Verwendung der synchronen Kanäle ist also die Modularität dieser Komponente gewährleistet.

Die Form ist eine Metapher und lehnt sich an das Bild einer Ampel[19] an. Solange eine Marke auf der grünen[20] Stelle liegt, kann die untere Transition schalten, sonst sperrt sie. Auf der grünen Transition liegt genau dann eine Marke, wenn die Marke auf der roten eine 0 ist.

[19](semáforo - spanisch für Verkehrsampel)

[20]Anmerkung für Schwarz-Weiß Drucke: Die untere Stelle ist grün, die obere rot.

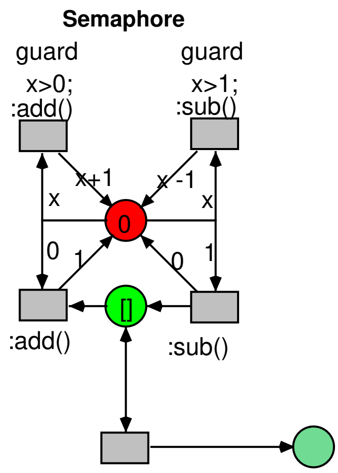

Abbildung 17: Die Netz-Komponente Semaphore *(NC semaphore)*.

4.3.2 Datenaustauschfunktionen

Nachricht empfangen *(NC in)*

Der Empfang einer Nachricht ist eine oft wiederkehrende Aufgabe bei Mulan-Protokollen. Schließlich ist es ihre Hauptaufgabe diese zu verarbeiten. Es ist sinnvoll, den Sender einer Nachricht als Kommentar an die Komponente zu schreiben. Die entgegengenommene Nachricht wird über die virtuelle Stelle *P* an den oberen Block übergeben, der frei positionierbar ist. Über die Transition zwischen *P* und *Perf* können weitere Daten, die aus der Nachricht ermittelt werden müssen, an eigenständige datenhaltende Stellen übergeben werden. Dazu sind ggf. weitere Stellen zu erzeugen, die ebenfalls orange sein sollen. Als Beispiel dafür dient die nicht benannte orange Stelle unter der Stelle *Perf*.

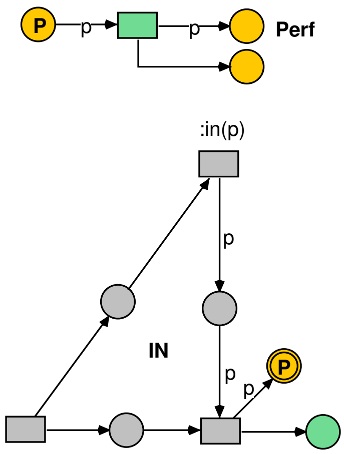

Abbildung 18: Die Netz-Komponente Nachricht empfangen *(NC in)*.

Die Form hat keine symbolische Bedeutung, vielmehr lehnt sie sich an die Grundmetapher an, dass Protokolle von links nach rechts ablaufen und Nachrichten von oben (Norden) in das Protokoll fließen. Weiterhin muss bei mehreren *:in(p)-Uplinks* sichergestellt sein, dass die richtige Transition schaltet, weswegen eine Stelle als Vorbedingung benötigt wird. Dies führt fast automatisch zur dreieckigen Form.

Nachricht senden *(NC out)*

Entsprechend der eingehenden Nachricht kommt die ausgehende Nachricht auch sehr häufig vor. Hier ist es ebenso sinnvoll, den Empfänger der Nachricht als Kommentar an die Komponente zu schreiben.

Die Form hat - genau wie bei *(NC in)* - keine symbolische Bedeutung, vielmehr lehnt sie sich an die Grundmetapher, dass Protokolle von links nach rechts ablaufen und Nachrichten nach oben (Norden) aus dem Protokoll fließen, an. Außerdem bilden die Formen der beiden Netz-Komponenten zueinander das jeweilige Gegenstück.

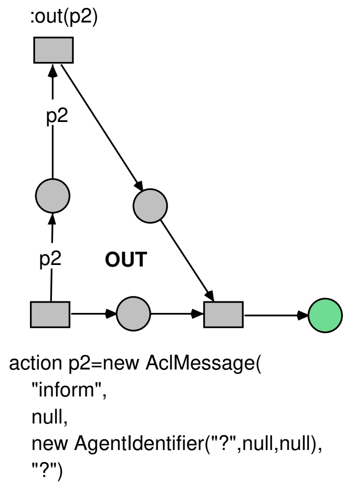

Abbildung 19: Die Netz-Komponente Nachricht senden *(NC out)*.

Nachricht senden und auf eine Antwort warten *(NC out-in)*

Diese Netz-Komponente ist die Kombination von *NC in* und *NC out*. Sie hat den Vorteil, dass sie sehr kompakt ist.

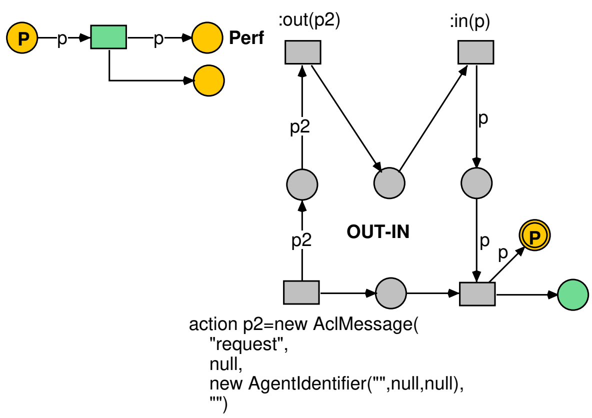

Abbildung 20: Die Netz-Komponente Nachricht senden und auf eine Antwort warten *(NC out-in)*.

Die Form dieser Netz-Komponente ergibt sich aus dem Übereinanderlegen der Figuren von *NC out* und *NC in*. Sie ist dadurch intuitiv erfassbar.

4.3.3 Elementare Strukturen

Starten des Protokolls *(NC start)*

Diese Komponente enthält zusätzlich zum eigentlichen Starten eines Protokolls, das eigentlich nur aus einem *Uplink* besteht, einen Deklarationsblock für alle *Imports* und für alle Variablen. Es enthält zusätzlich den Zugang zu der Wissensbasis und eine komplette Netz-Komponente "Nachricht empfangen". Letztere ist hinzugefügt, da zuerst, bevor irgend etwas anderes passieren kann, die eingehende Nachricht übergeben wird. Der Deklarationsblock ist eingefügt, um Schreibarbeit zu sparen. Es werden schon alle von den Netz-Komponenten benutzten Variablennamen deklariert, womit außerdem vermieden wird, dass es zu Konflikten in der Namensgebung kommen kann. Da alle Netz-Komponenten leicht anpassbar sind, ist es möglich, z.B. den Deklarationsblock für das jeweilige Projekt anzupassen. Durch das Einfügen der Wissensbasis wird ein Vorschlag für die Anordnung vorgegeben, der sich dadurch in einer Gruppe als Quasistandard durchsetzen kann.

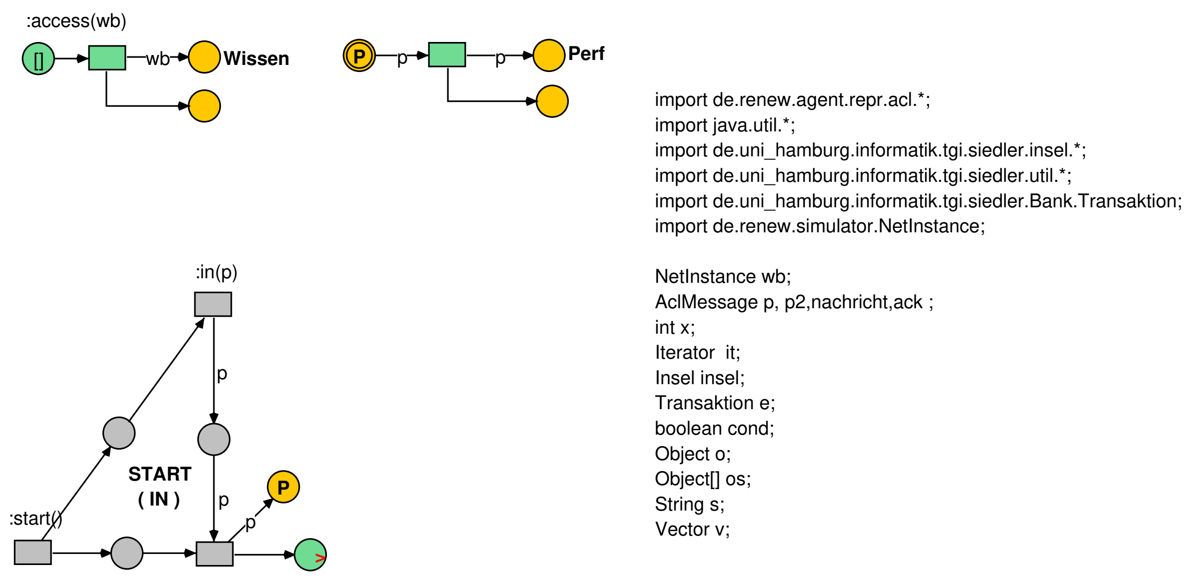

Abbildung 21: Die Netz-Komponente Starten des Protokolls *(NC start)*.

Stoppen des Protokolls *(NC stop)*

Das Stoppen des Protokolls wird durch einen *Uplink* gewährleistet. Es können im Gegensatz zum Starten des Protokolls mehrere Endtransitionen in einem Protokoll benutzt werden.[21] Diese Transition hat eine eigene Farbe, um das Ende eines Protokolls hervorzuheben.

[21] Mehrere Endtransitionen dürfen sich nur an Enden von alternativen Kontrollflüssen befinden.

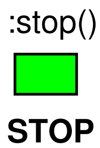

Abbildung 22: Die Netz-Komponente Stoppen des Protokolls *(NC stop)*.

4.4 Ein Anwendungsbeispiel

Zum Schluss diese Kapitels wird nun noch ein etwas komplizierteres Protokoll aus dem Projekt AOSE [MKR01] mit Hilfe der Netz-Komponenten neu erstellt. Dies soll die Vorgehensweise exemplarisch verdeutlichen und gleichzeitig die Möglichkeit eines Vergleichs zwischen den ursprünglichen und den neu gestalteten Protokollen bieten. Das benutzte Netz ist eines der besser gestalteten Netze des Projektes. Einige Konzepte, die hier vorgestellt worden sind, finden sich auch schon in diesem Netz wieder, sind allerdings nicht konsistent ausgeführt worden.

4.4.1 Das Protokoll "*anmelden*" der Spielleitung.

Das Protokoll "*anmelden*" (Abbildung 23) bewerkstelligt die Anmeldung eines Spielers bei der Spielleitung. Diese erhält eine Nachricht, dass sich ein Spieler anmelden möchte. Daraufhin sendet die Spielleitung an die Bank eine Nachricht, um ein neues Konto zu eröffnen. Wenn diese eine Bestätigung zurück schickt, wird die Insel über den neuen Spieler benachrichtigt, und die Bank erhält eine Nachricht über das Startkapital. Nachdem die Bank das bestätigt hat, wird - falls sich alle Spieler angemeldet haben - eine Nachricht an die Spielleitung selbst geschickt, die besagt, dass an alle Spieler eine Benachrichtigung über den Beginn der Bauphase gesendet werden soll. Schließlich wird eine Bestätigung an den Spieler zurückgeschickt. Falls kein neues Konto erstellt werden konnte, wird nur ein Fehlschlagen an den Spieler gesendet.

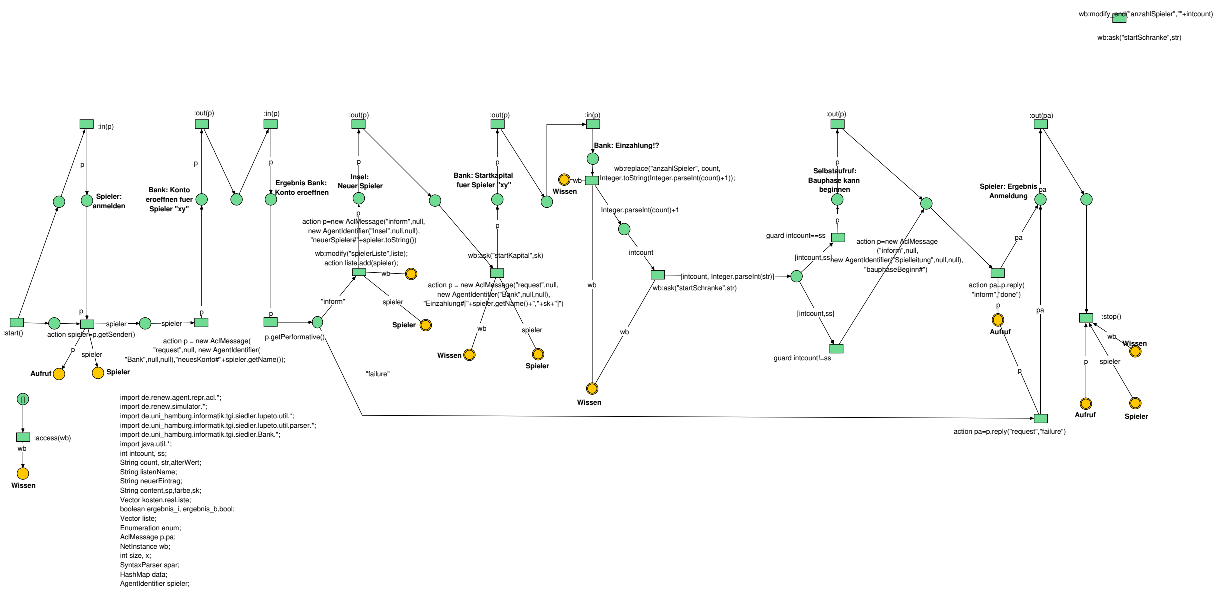

Abbildung 23: Das ursprüngliche Protokoll "*anmelden*" der Spielleitung.

4.4.2 Das Protokoll "*anmelden*" mit Netz-Komponenten

Das neue Protokoll "*anmelden*" (Abbildung 24) hat nicht nur die gleiche Funktionalität wie das ursprüngliche, sondern bewerkstelligt dies auch auf die gleiche Weise.

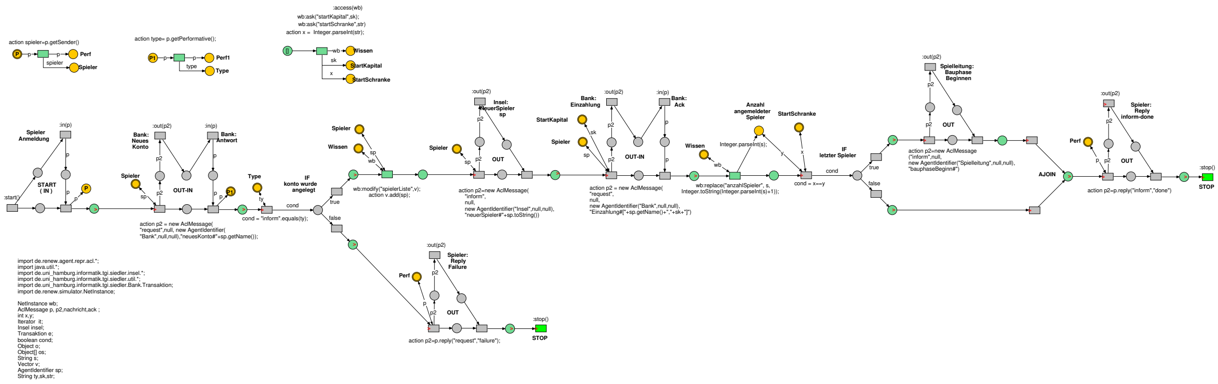

Abbildung 24: Das neue Protokoll "*anmelden*" mit Netz-Komponenten.

Es wird hier nicht erwartet, dass das Netz in der vorliegenden Größe[22] gelesen werden kann. Aber gerade weil die Details nicht mehr zu erkennen sind, werden die übergeordnete Struktur und die Netz-Komponenten besser sichtbar. Letztere werden in der Abbildung 25 zusätzlich mit geometrischen Figuren unterlegt.

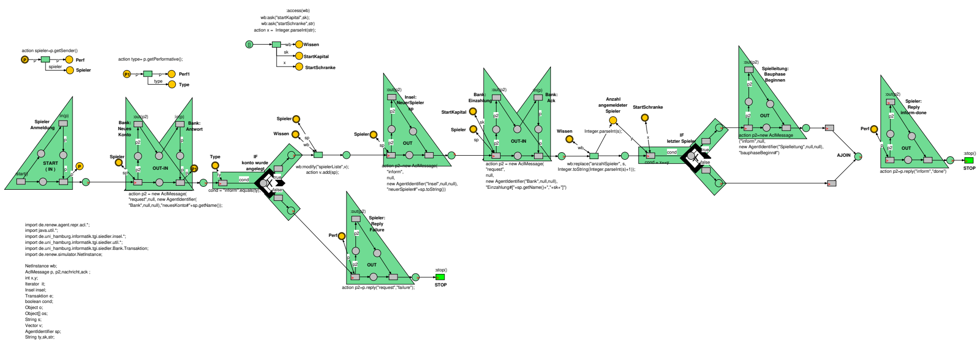

Abbildung 25: Das Protokoll "*anmelden*" mit farbig gekennzeichneten Netz-Komponenten.

Der hier vorgestellte Weg, ein Netz zu erstellen, dass aus zusammengefügten Netz-Komponenten besteht, ist nicht der übliche, und deswegen soll hier auch die Entwicklung des Protokolls in der üblichen Weise erfolgen.

[22]Im Anhang sind größere Abbildungen der Netze vorhanden.

4.4.3 Entwicklung eines Protokolls

Wie in Kapitel 3 beschrieben, ist die Modellierung des Ablaufes von Konversationen mit Hilfe von Protokolldiagrammen überaus sinnvoll. In diesem Abschnitt wird der gleiche Ablauf (Anmelden eines Spielers), der am Anfang dieses Abschnittes beschrieben wurde, mit einem Protokolldiagramm ausgedrückt. Abbildung 4.4.3 zeigt das passende Protokolldiagramm in der ausführlichen Variante.

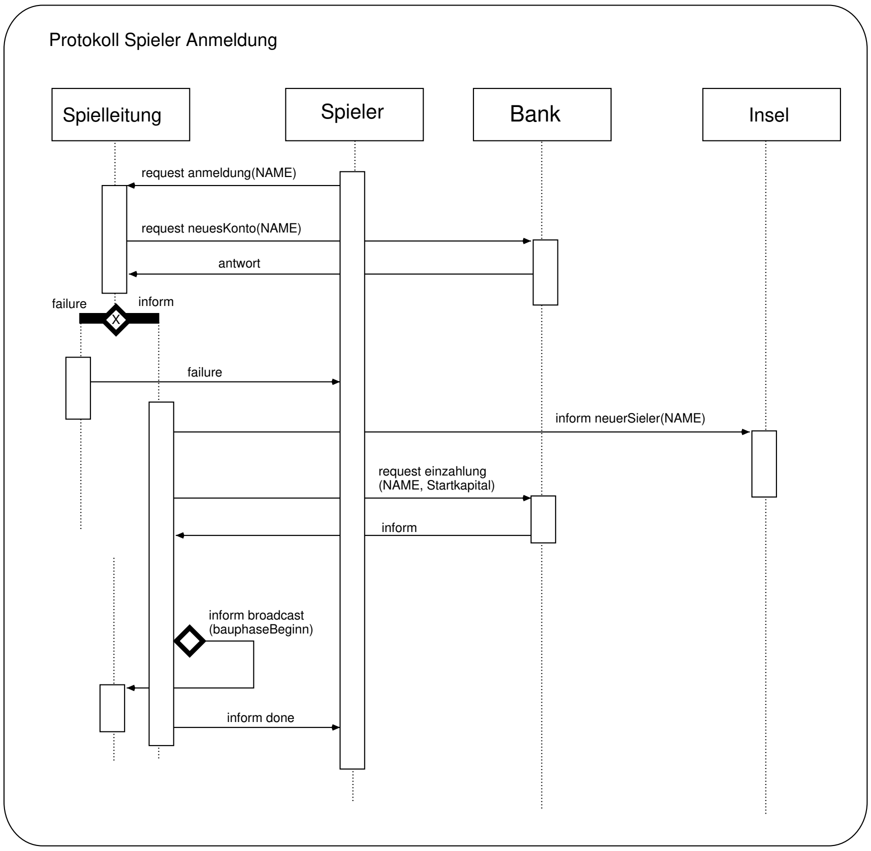

Abbildung 26: Das Protokolldiagramm zur Konversation "Spieler anmelden".

Als nächstes wird zur Veranschaulichung der betreffende Teil aus dem Diagramm herausgegriffen und, um eine zu den Protokollen einheitliche Leserichtung zu erhalten, um neunzig Grad gedreht (Abbildung 27).

Wenn jetzt die einzelnen Aufgaben des Protokolls für die Spielleitung durch Symbole, die schon aus den Netz-Komponenten bekannt sind, hervorgehoben werden (Abbildung 28), kann man leicht sehen, das der Übergang von der Spezifikation zur Implementation sehr leicht wird.[23]

[23]Die Netze sind noch mal im Anhang aufgeführt.

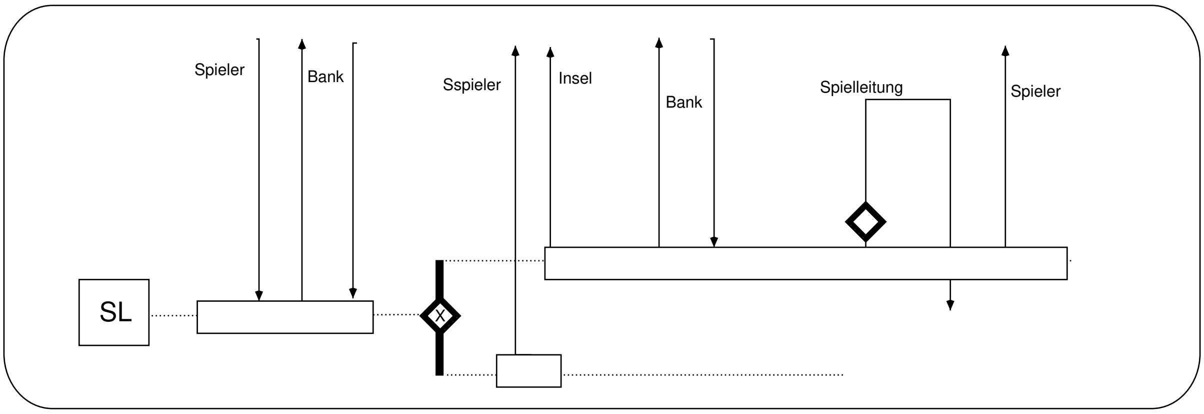

Abbildung 27: Ausschnitt aus dem Protokolldiagramm "Spieler anmelden" (hier horizontal).

Es ist deutlich erkennbar, dass die Formen der Protokolldiagramme sehr nahe an denen der Mulan-Protokolle liegen können. Weiterhin ist durch die Verwendung von den Netz-Komponenten eine uniforme und übersichtlichere Struktur erreicht worden. Das neue Protokoll ist zwar größer, dennoch ist die Form weitaus leichter zu erfassen und - wenn die Formen der Netz-Komponenten bekannt sind - auch leichter zu lesen. Das Beispiel hat weiterhin gezeigt, dass die Form der Protokolle durch die Netz-Komponenten sehr eng an die Struktur der Protokolldiagramme angepasst werden kann. Dies kann dazu beitragen, dass die Entwicklung von Mulan-Protokollen vereinfacht wird.

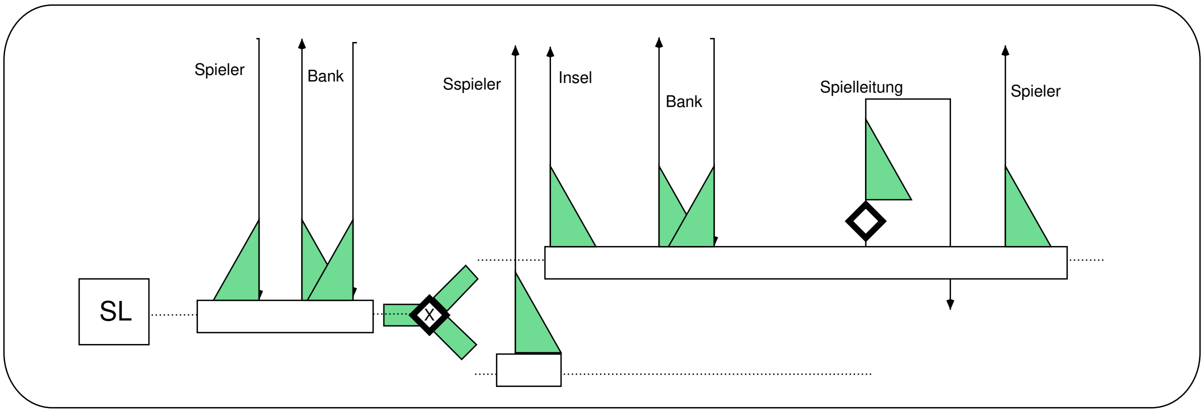

Abbildung 28: Ausschnitt aus dem Protokolldiagramm“Spieler anmelden” ergänzt durch geometrische Figuren, die an Netz-Komponenten angelehnten sind.

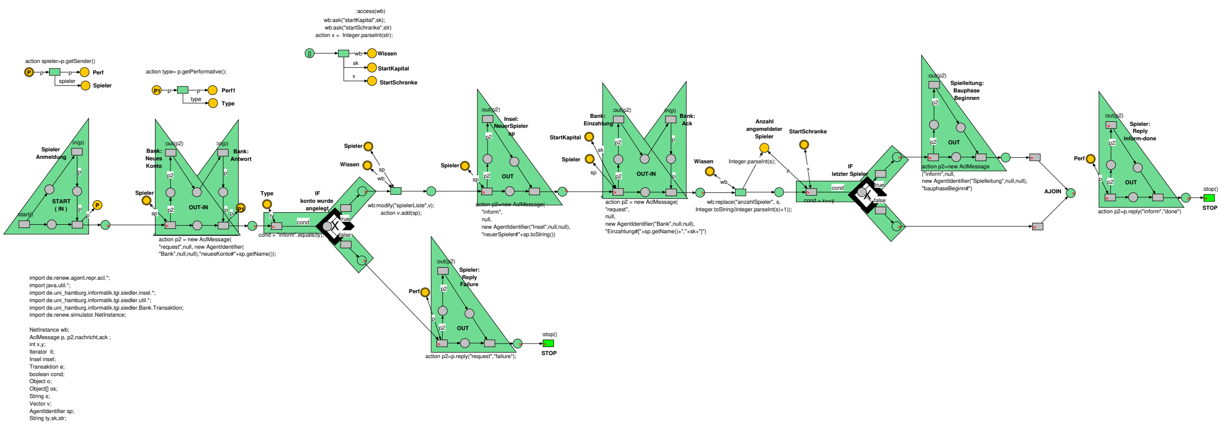

Abbildung 29: Das Protokoll “*anmelden*” von 4.4.2 zum Vergleich.

5 Das Tool

Petrinetze können mit Hilfe von Renew komfortabel erstellt werden. Um auch Netz-Komponenten einfach in den Programmierprozess mit einzubinden, ist es wünschenswert, dafür ein Werkzeug zur Verfügung zu haben. Im diesem Kapitel wird als Erweiterung von Renew eine Werkzeugpalette vorgestellt, mit der Netz-Komponenten leicht erstellt werden können.

5.1 Anforderungen

Das Tool sollte leicht zu bedienen sein, deswegen scheidet ein Menüeintrag für jede Netz-Komponente, wegen der umständlichen Bedienung aus. Es sollte sich an das bestehende System anpassen und es ergänzen, also nicht wie ein Fremdkörper wirken. Alle Werkzeuge zum Zeichnen von Netzelementen sind über Paletten zu erreichen. Es liegt also nahe, das erforderliche Netz-Komponenten-Tool auch als Palette zu realisieren.

5.2 Realisierung

Als Palette sind alle Netz-Komponenten immer und leicht zugänglich. Da das benutzen der Netz-Komponente gefördert und nicht behindert werden soll, ist diese einfache Lösung ideal geeignet. Abbildung 30 zeigt die Bedieneroberfläche von Renew mit der Erweiterung für die Netz-Komponenten.

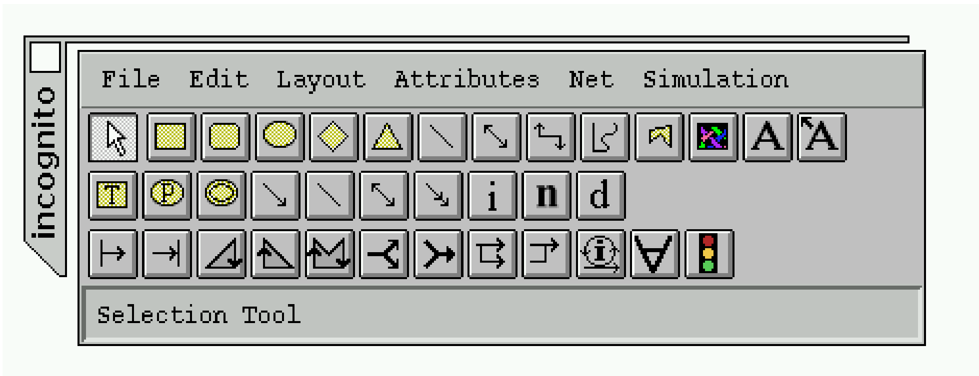

Abbildung 30: Die Bedieneroberfläche von Renew um eine Palette aus Knöpfen für die Netz-Komponenten erweitert.

Alle Komponenten liegen als Renew-Dateien vor, so dass sie einfach verändert werden können. Die Lage des Repository ist durch Angabe einer Startoption veränderbar, dennoch scheint es wünschenswert, dass in einem Projekt alle Teilnehmer mit den gleichen Netz-Komponenten arbeiten und somit diese aus demselben Repository erzeugen. Ein Vorteil dabei ist, dass Änderungen an den Dateien alle Projektteilnehmer gleichermaßen betreffen. Die Erweiterung ist eigenständig und beeinflusst Renew nicht. So kann Renew ohne die

Erweiterung betrieben werden, falls dies erwünscht ist. Es kann sogar eine bestehende "*renew.jar*" Datei erweitern[24]. Es ist auch leicht möglich, die Erweiterung direkt in Renew zu integrieren, da es bei nicht vorhandenen Netz-Komponenten Repository die Funktionalität von Renew nicht beeinträchtigt. Als letzte Möglichkeit wäre auch eine dynamische Anbindung über einen Menüeintrag oder als *Plug-in* realisierbar.

5.3 Benutzung

Die Netz-Komponenten sind genauso wie Netzelemente (Transitionen oder Stellen) in die Zeichnung einzufügen. Es wird mit dem Mausklick eine Netz-Komponente ausgewählt und dann die Position im Netz angeklickt, an der die Netz-Komponente erscheinen soll. Ein Unterschied zu der bisherigen Bedienungsweise ist, dass die hinzugefügten Elemente nach dem Erstellen selektiert sind. Dies gewährleistet eine komfortable Positionierung der neuen Netz-Komponente. Da die meisten Netz-Komponenten nach oben hin ausgerichtet sind, markiert der Mausklick bei fast allen Netz-Komponenten die untere linke Transition.[25] Dies hängt allerdings von dem Ersteller der Komponenten ab, da dieser, durch die Positionierung des Netzes in dem Original, das Positionieren der Netz-Komponente beeinflussen kann.

[24]Mit einer kleinen Einschränkung: Eine Methode muss noch den Status *public* erhalten.
[25]Ausnahmen: *NC pjoin*, *NC psplit*

6 Diskussion

Es wurde auf die Notwendigkeit aufmerksam gemacht beim Kodieren von Applikationen mit Petrinetzen sowohl einer strukturierten Vorgehensweise zu folgen als auch den Code zu strukturieren. Netz-Komponenten bieten im Allgemeinen die Möglichkeit Strukturen vorzugeben. Im Speziellen bieten die hier vorgestellten Netz-Komponenten zusätzliche Elemente und Konventionen, die das Erstellen und das Lesen von Mulan-Protokollen weiterhin vereinfachen. Das vorgestellte Tool fügt sich nahtlos an Renew an und ermöglicht die praktische Umsetzung des Konzeptes.

6.1 Fazit

Richtlinien oder Konventionen festzulegen, ist ein adäquates Mittel, um die Übersichtlichkeit zu erhöhen. Gerade bei größeren Projekten ist es unerlässlich, den Überblick nicht zu verlieren. Gleichzeitig ist es wünschenswert, schnell Resultate erzielen oder zumindest Teilziele erreichen zu können. Um auf Teilergebnissen aufbauen und Erweiterungen bzw. Ergänzungen zu dem bestehenden System hinzufügen zu können, ist es wichtig, dass das Pflegen und Warten des Systems nicht übermäßig viel Zeit in Anspruch nimmt. Dies kann nur erreicht werden, wenn der bis dato erstellte Code leicht anpassbar ist. Dazu muss er für den Betrachter leicht zugänglich, also lesbar sein.

Bei jedem Projekt spielt die Zeit eine Rolle, meistens eine entscheidende. Der Aufwand, der bei dem Durchforsten von "unschön" geschriebenen Code oder Auffinden und Beseitigen von Fehlern entsteht, steigt natürlich mit der Menge des produzierten Codes. Der Vorteil, den eine Konvention für den Programmcode und ein Beschränken auf bestimmte Methoden bringt, liegt auf der Hand.

Das Erstellen von Mulan-Protokollen mit Hilfe von Netz-Komponenten ist eine effiziente Implementierungsmethode. Die Modellierung kann schnell und direkt in Netze umgesetzt werden, denn die Netz-Komponenten sind an die im Modell spezifizierten Aufgaben angepasst. Der erzeugte Code wird übersichtlicher und entspricht einem konformen Stil. Durch die wieder verwendbaren Netz-Komponenten kann ein Entwickler die erstellten Netze schneller begreifen, weil syntaktische Einheiten erkennbar werden. Falls sich die Bedürfnisse der Entwickler ändern, so ist eine Modifikation der Netz-Komponenten einfach zu bewerkstelligen. Insofern bieten sie eine ausreichende Flexibilität.

Netz-Komponenten sind keine neuen Elemente (z.B. Piktogramme), sondern Teilnetze. Die Information über ihre Bedeutung befindet sich somit nicht in einem Netzelement, sondern in der Anordnung der Netzelemente selbst. Dies unterscheidet sie von Modularisierungen, die von den visuellen Programmiersprachen benutzt werden. Vergleichbar sind sie eher mit den *Workflow-Pattern*, von denen sie sich durch die ausgeprägte geometrische Form unterscheiden, die zwar auch funktional bedingt ist, aber vorwiegend eine anschauliche Motivation hat.

Eine Ergänzung der Paletten von Renew scheint eine sowohl einfache als auch sinnvolle Lösung zu sein. Zum einen ist so eine Erweiterung relativ leicht hinzuzufügen, zum anderen wird der Funktionsumfang von Renew durch die Erweiterung nicht eingeschränkt. Ein

Zwei-Klick Mechanismus - bestehend aus Auswahl der Netz-Komponente und Auswahl des Ortes der Komponente - scheint eine erhebliche Vereinfachung im Vergleich mit *"Copy and Paste"* Verfahren, zu sein. Eine Einschränkung des Funktionsumfangs liegt insofern nicht vor, da in den Teilen der Netze, die nicht durch die Konvention abgedeckt werden, die volle Mächtigkeit der Petrinetze weiterhin verwendbar ist.

Durch die strikte Trennung von Daten und Kontrollfluss wird die Übersichtlichkeit stark gesteigert. Erst durch sie wird die Modularität der Netz-Komponenten ermöglicht. Der Kontrollfluss wird klar erkennbar, d.h. das Netz hat einen klaren Ablauf der auch nachvollziehbar ist. Dies ist insbesondere von Vorteil, wenn die Netze einen kontrollflussorientierten Charakter haben, wie es bei Mulan Protokollen der Fall ist. Ein weiterer entscheidender Vorteil ist, dass die Funktion der einzelnen Netz-Komponenten klar erkennbar und eindeutig ist. Da eine Netz-Komponente eine syntaktische Einheit bildet, werden für klar getrennte Aufgaben auch klar getrennte Netz-Komponenten benutzt. Dies erhöht zwar den Aufwand für die Darstellung der Netze, weil mehr Netzelemente benötigt werden, aber es verbessert gleichzeitig die Lesbarkeit.

Durch die in Kapitel (4) vorgestellten Netz-Komponenten wird die Form von Mulan-Protokollen stark bestimmt. Sie geben die allgemeine Ausrichtung (Kontrollfluss) des Protokolls vor. Weiterhin ist durch die Verwendung der Netz-Komponenten eine starke Modularisierung der Mulan-Protokolle erreicht worden. Jede Aufgabe, die durch eine Netz-Komponente realisiert wird hat eine festgelegte geometrische Form. Außerdem werden durch die Realisierung der Netz-Komponenten die Trennung der Daten schon vorgegeben, sa dass es leicht ist diesen Vorgaben zu folgen. Durch diese 4 Punkte (Ausrichtung, Modularisierung, geometrische Anordnung und Trennung von Daten und Kontrollfluss) wird eine vereinheitlichte Form der Mulan-Protokolle erreicht. Diese ist nicht nur klar und leicht lesbar, sondern lehnt sich auch eng an die Form der Modellierung mit Protokolldiagrammen an, so dass letztere leicht in Mulan-Protokolle überführt werden können.

Netz-Komponenten und die Trennung von Daten und Kontrollfluss, zusammen mit einigen Richtlinien, wie Netz-Komponenten zusammen gefügt werden können, bieten eine einfache und schnelle Möglichkeit, Mulan-Protokolle in einer einheitlichen Form zu erstellen. Somit kann mit Hilfe der hier vorgestellten Methoden die Effizienz bei der Entwicklung von Mulan-Protokollen erheblich gesteigert werden.

6.2 Ausblick

Obwohl diese Arbeit gezeigt hat, dass die Verwendung von Netz-Komponenten und die konsequente Trennung von Kontrollfluss und Daten beim Entwickeln von Mulan-Protokollen sinnvoll ist, bleibt die Frage offen, inwieweit die hier vorgestellten Prinzipien auf allgemeine Petrinetze zu übertragen sind. Bei der Entwicklung bestimmter Typen von Netzen ist der Gebrauch von wieder verwendbaren Komponenten sicherlich möglich und zweckmäßig. Um die Palette für andere Arten von Petrinetzen zu verwenden, müssen nur neue Bibliotheken von Netz-Komponenten und Piktogramme für die Paletten erzeugt werden. Es wäre auch möglich, mehrere Paletten gleichzeitig oder abwechselnd zu benutzen. Allerdings wird das Erstellen von Paletten bisher noch nicht genügend unterstützt. Dies zu verbessern ist einfach möglich, da die Komponenten leicht mit dem Editor von Renew erstellt werden können. Nur die Entwicklung der Darstellung der Paletten müsste verbessert werden.

Die Modularisierung der Netze könnte mit dem Ersetzen von Netz-Komponenten durch andere Symbole weiter fortgeführt werden. Dabei könnten Netz-Komponenten durch einfache geometrische Gebilde, die den Figuren ähneln, oder durch Piktogramme (*Icons*) repräsentiert werden. Allerdings geht dann ein wichtiger Aspekt der Netz-Komponenten verloren. Mulan-Protokolle würden dann nicht mehr als Petrinetze bezeichnet werden können. Die Aufgabe der Netz-Komponenten ist es nicht die Darstellungsart, sondern die Darstellungsweise der Mulan-Protokolle zu verbessern.

Ein Problem, dass bisher in dieser Arbeit nur teilweise gelöst werden konnte, ist die Verwendung der Daten. Da alle Variablen nur lokal verwendet werden können, müssen diese immer explizit gesichert und weitergegeben werden. Der Ansatz, die Daten von dem Kontrollfluss zu trennen, ist eine Möglichkeit das Problem auch in größeren Netzen zu bewältigen. Allerdings täuscht er nicht über die Tatsache hinweg, dass die Behandlung von Daten in Petrinetzen eher umständlich ist. Wenn es die Möglichkeit gäbe, Variablen zu definieren, die eine netzweite Gültigkeit (*Scope*) besitzen, so könnte der Aufwand, die Daten zu verwalten, erheblich reduziert werden. Offen ist allerdings inwieweit ein solches neues Konzept die Nebenläufigkeit der Netze und ihren Formalismus beeinträchtigt. Dies zu betrachten würde allerdings den Rahmen dieser Arbeit sprengen.

Die Unterteilung in lose verknüpfte syntaktische Einheiten zusammen mit der Tatsache, dass die Beschreibung der Netz-Komponenten an den Spezifikationen der Protokolle angelehnt ist, legt nahe, dass die Netze direkt aus der Spezifikation generiert werden könnten. Gerade wenn die Spezifikationen in Form von Protokolldiagrammen vorliegen, ist diese Möglichkeit nahe liegend (vergl. Abbildungen 28 und 29). Dies könnte entweder halb- oder vollautomatisch geschehen. Bei der halbautomatischen Umwandlung werden die Gerüste der Protokolle vorgeneriert, die nachträglich (manuell) angepasst werden können. Inwieweit es gelingen könnte, vollständige Netze aus den Spezifikationen zu generieren, bleibt offen. Auch dies geht über den Rahmen dieser Arbeit weit hinaus. Allerdings wäre dazu wohl eine Erweiterung der Spezifikationssprache nötig, um die genauen Funktionen der Protokollteile zu definieren. Gerade die Semantik der neuen Elemente der Protokolldiagramme (vergl. Abbildung 4) ist nicht eindeutig festgelegt, so dass die Umsetzung dieser Möglichkeit start einschränkt ist. Eine Erweiterung der Protokolldiagramm-Semantik die, es erlauben würde die Mulan-Protokolle automatisch zu generieren, könnte dann allerdings auch als eigene Protokollsprache angesehen werden, und es stellt sich die Frage ob diese nicht schon von sich aus - ohne den Umweg über die Petrinetze - ablauffähig wären.

7 Glossar

Agent Eine eigenständige, unabhängige, anpassungsfähige und intelligente Softwarekomponente. Siehe auch 2.2.1

AOSE Die Agentenorientierte Softwareentwicklung ist ein Gebiet der verteilten künstlichen Intelligenz.

Code/ Programmcode Dies bezeichnet den Text eines Programms. Allerdings wird auch ein Petrinetz als Code bezeichnet, denn das Netz ist der Text der das Programm beschreibt.

Downlink Eine Transition die zusammen mit dem *Uplink* einen synchronen Kanal bildet.

FIPA Foundation of Intelligent Physical Agents. www.fipa.org

GUI Graphical User Interface, die Bedieneroberfläche eines Programms.

Icon/ Piktogramm Bilder, die bez. ihrer Bedeutung selbsterklärend sind; oft Metaphern.

Konversation Beschreibt die Interaktion/ Kommunikation zwischen mehreren Agenten und die Nachrichten, die dabei ausgetauscht werden.

Indentation Einrücken des Programmcodes, um die Struktur zu verdeutlichen.

Lesbarkeit Mit Lesbarkeit wird in dieser Arbeit die Eigenschaft beschrieben, dass ein Programm leicht zu ergründen ist.

Mulan Multiagentennetze. Ein Multiagentensystem, das in Petrinetzen implementiert ist und auf Renew aufsetzt.

***Instance*/ Exemplar** Das simulierte Petrinetz. Zu einem Netzmuster können mehrere Exemplare instanziert werden. Vergleichbar zum Objektvorkommnis im Gegensatz zur Objektklasse.

Palette Eine abnehmbare Leiste von graphischen Knöpfen die zur GUI eines Programms gehören.

***Protocoldiagram*/ Portokolldiagramm, PD** Erweiterung der UML-Sequenzdiagramme, um Konversationen zwischen Agenten zu beschreiben. Die Erweiterungen entsprechen den Junktoren AND, OR und XOR.

Protokoll Beschreibt das Verhalten eines Agenten in einer Konversation.

Renew ist ein Petrinetz Editor und Simulator. www.renew.de

Stelle Ein Petrinetzelement, dass eine passiven Charakter hat. Stellen haben eine runde Form.

Synchroner Kanal Eine Verknüpfung von zwei Transitionen, die ein synchrones Schalten dieser beiden gewährleistet. Über einen synchroner Kanal können auch Daten ausgetauscht werden.

Syntax Highlighting Beim Syntax Highlighting werden Elemente des Programm-Codes durch verschiedene Farben der Schrift voneinander abgehoben.

Transition Ein Petrinetzelement, dass eine aktiven Charakter hat. Transitionen haben eine viereckige Form.

VPL Visual Programming Language, eine Programmiersprache, deren Programme graphisch repräsentiert werden.

UML Unified Modeling Language.

Uplink Eine Transition die zusammen mit dem *Downlink* einen synchronen Kanal bildet.

Usability/ Benutzbarkeit Eine Software heißt Benutzerfreundlich, wenn ihre Bedienung einfach ist und sich an die Denk- und Arbeitsweise des Benutzers anpasst.

Workflow Prozessbeschreibung. Zum Teil automatisiert ablaufende Gesamtheit von Aktivitäten. (Jablonski, Böhm, Schulze, [JBS97])

Workflow-Pattern Ein wiederkehrendes Muster, das von der Form abstrahiert.

Petrinetz/ Netz Systemmodell für Vorgänge, Organisationen und Geräte (Reisig, [Rei82]). In dieser Arbeit werden Petrinetze auch als Programmiersprache.

Literatur

[BGK+02] Bosch, Tobias, Oliver Gries, Heiko Kausch, Maxim Klenski, Kolja Lehmann, Michael Morales, Valentin Seegert and Anatolij Vilner: *Agentenorientierte Implementierung des Spiels "Die Siedler von Catan"*, 2002.

[BRJ96] Booch, Grady, James Rumbaugh and Ivar Jacobson: *The Unified Modeling Language User Guide.* Addison-Wesley, Reading, Massachusetts, 1996.

[DDH75] Dahl, O.-J., E. W. Dijkstra and C. A. R. Hoare: *Structured Programming.* Acad. Press, London, 7 edition, 1975.

[FIP] *Foundation of Intelligent Physical Agents.* http://www.fipa.org.

[FIP01] FIPA: *FIPA Interaction Protocol Library Specification.* pdf, August 2001. http://www.fipa.org/specs/fipa00025/XC00025E.pdf.

[JBS97] Jablonski, Stefan, Markus Böhm and Wolfgang Schulze: *Workflow-Management Entwicklung von Anwendungen und Systemen.* dpunkt-Verlag, Heidelberg, 1997.

[Jen96] Jensen, Kurt: *Colored Petri Nets*, volume 1. Springer, Berlin, Second edition, 1996.

[KWD01] Kummer, Olaf, Frank Wienberg and Michael Duvigneau: *Renew - User Guide.* Dokumentation, Theoretische Grundlagen der Informatik, FB Informatik, Universität Hamburg, 2001. http://www.renew.de.

[Lab] LabVIEW: *LabVIEW Home.* http://www.labview.com.

[McI98] McIntyre, David: *Comp.Lang.Visual - Frequently-Asked Questions List*, 1998. ftp://rtfm.mit.edu/pub/usenet/comp.lang.visual/comp.lang.visual_Frequently-Asked_Questions_(FAQ).

[MKR01] Moldt, Daniel, Michael Köhler and Heiko Röhlke: *Projekt: Agentenorientierte Softwareentwicklung*, WS 2001.

[Obe87] Oberquelle, Horst: *Sprachkonzepte für benutzergerechte Systeme.* Springer-Verlag, Berlin, 1987.

[Pic] Pictorius, Inc.: *The Home of Visual Object-Oriented Development Environments.* http://www.pictorius.com/home.html.

[Rei82] Reisig, W.: *Petrinetze - Eine Einführung.* Springer-Verlag, Berlin, 1982.

[Val99] Valk, Rüdiger: *Modelle für Rechensysteme.* -, WS 1998/99. Skript.

[vdA00] AALST, WILL VAN DER: *Workflow Patterns*, 2000.
http://tmitwww.tm.tue.nl/research/patterns/wfs-pat-2000.pdf.

A Abbildungen

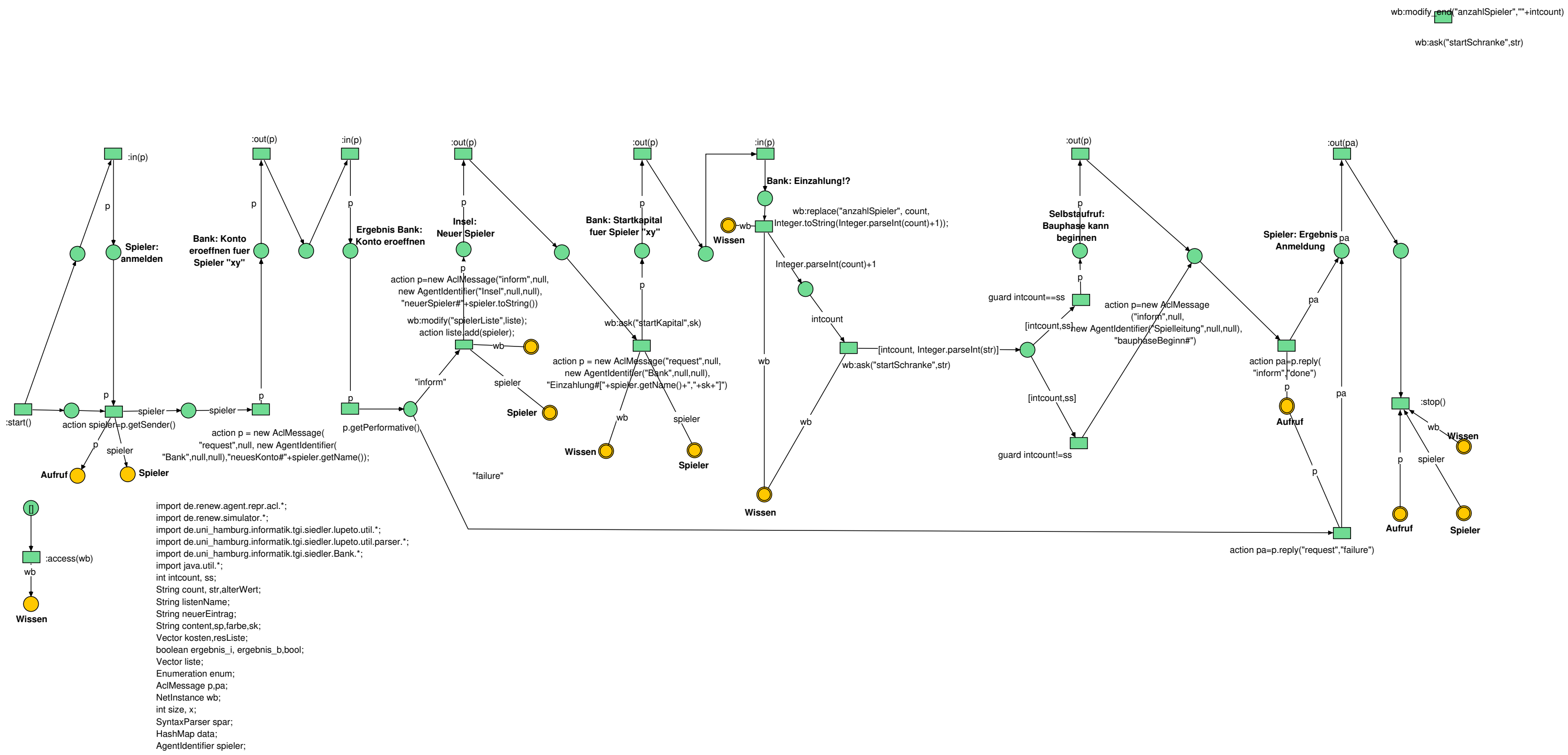

Abbildung 31: Das ursprüngliche Protokoll *anmelden* der Spielleitung.

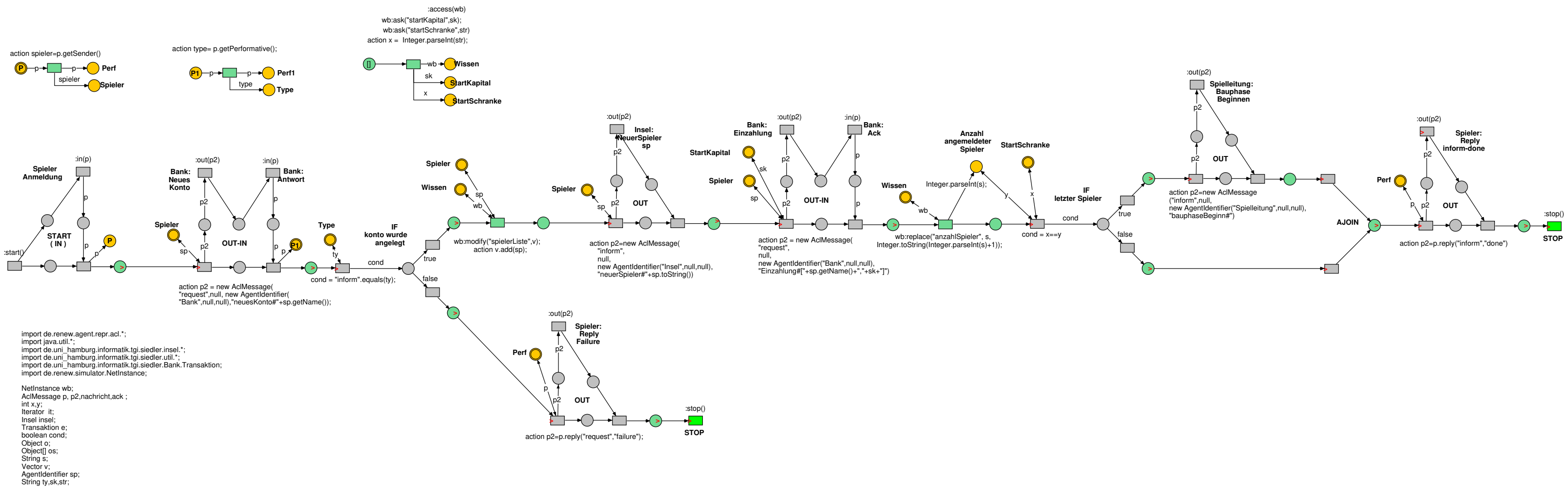

Abbildung 32: Das neue Protokoll *anmelden* mit Netz-Komponenten.

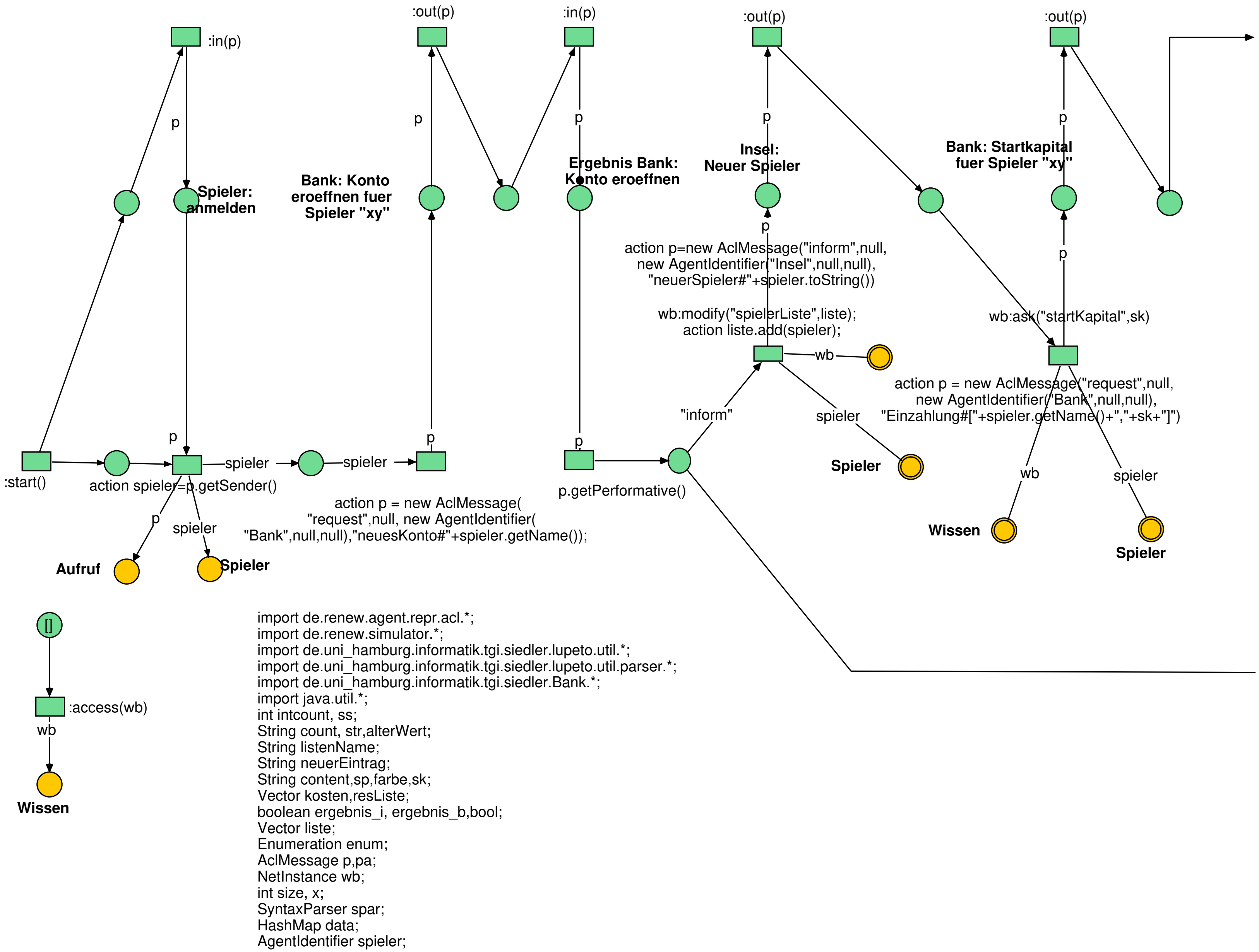

Abbildung 33: Linker Ausschnitt aus dem ursprünglichen Protokoll *anmelden* der Spielleitung.

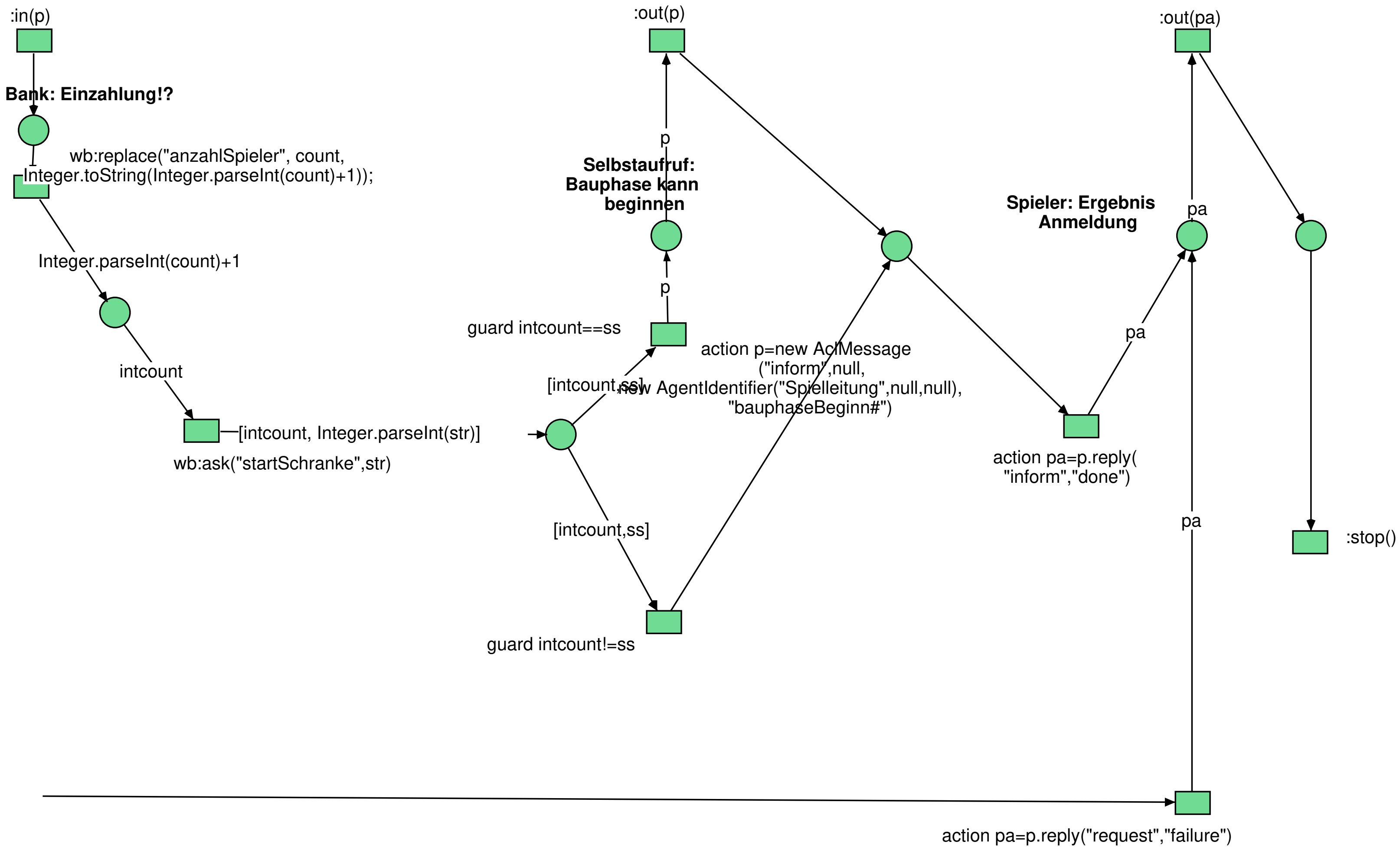

Abbildung 34: Rechter Ausschnitt aus dem ursprünglichen Protokoll *anmelden* der Spielleitung.

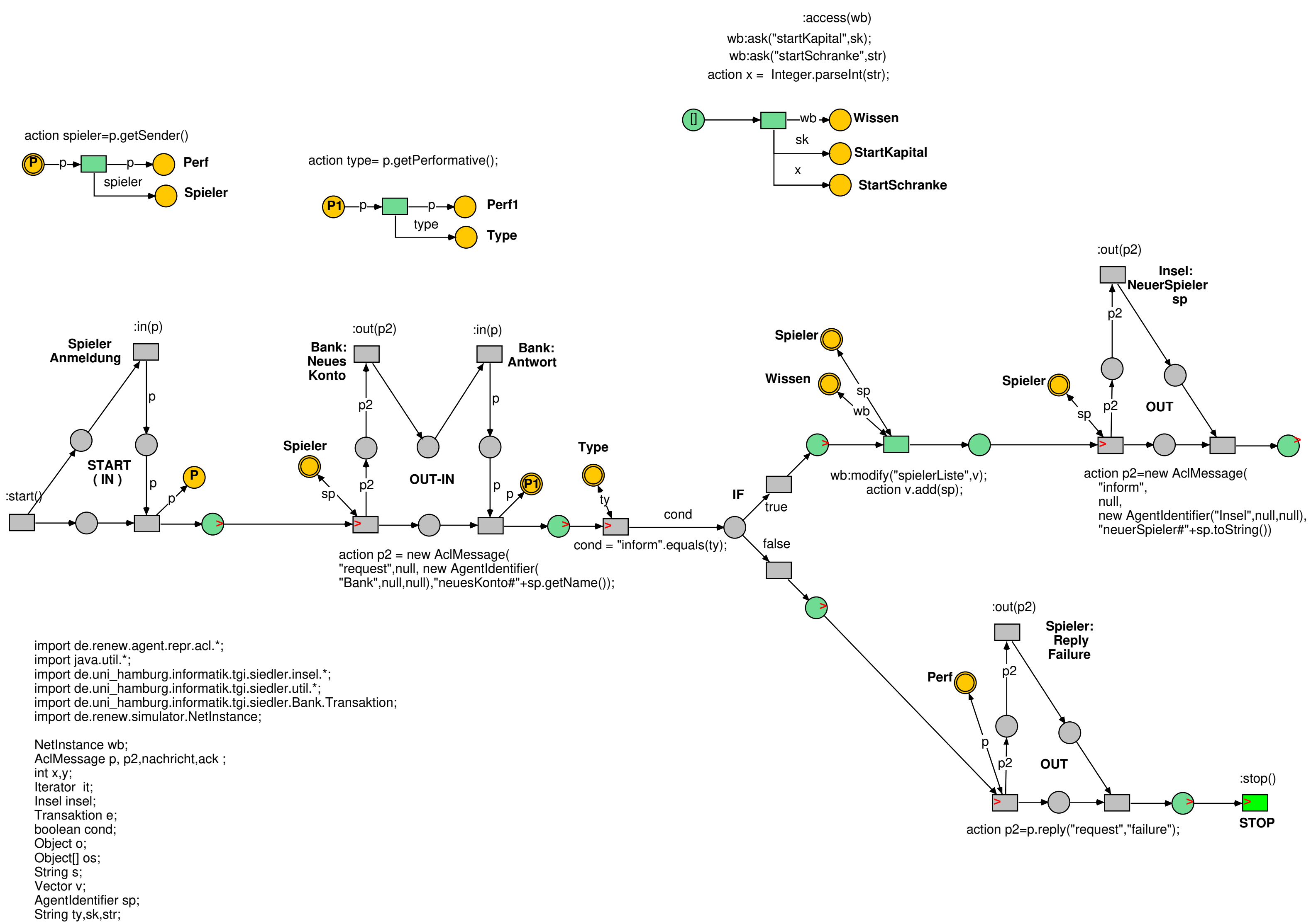

Abbildung 35: Linker Ausschnitt aus dem neuen Protokoll *anmelden* mit Netz-Komponenten.

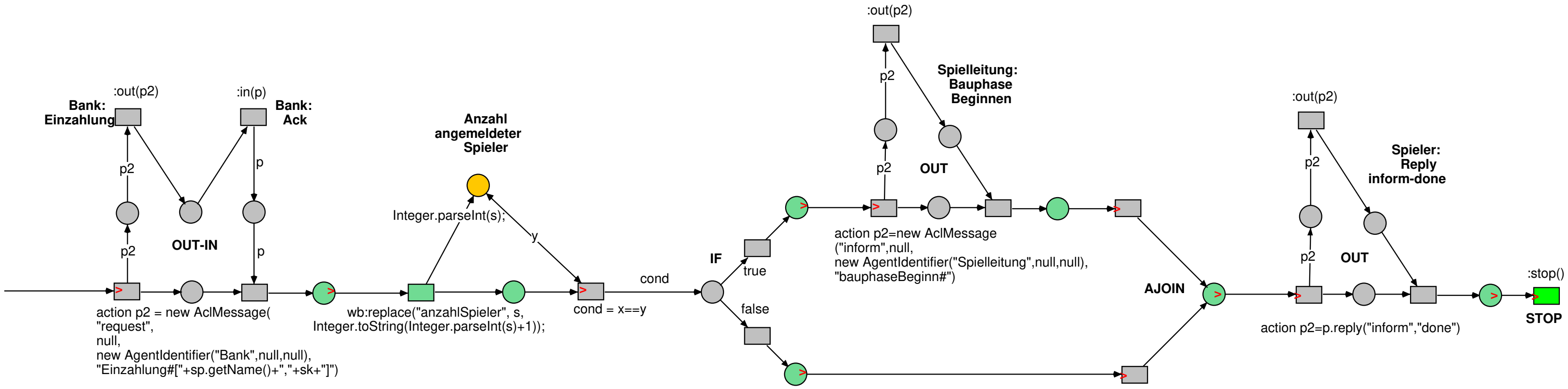

Abbildung 36: Rechter Ausschnitt aus dem neuen Protokoll *anmelden* mit Netz-Komponenten.

B Programmcode

```
package de.renew.gui;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.application.*;

import de.renew.util.StringUtil;


/**
 * MApp.java
 *
 * Mulan Application (MApp) extends the usual functionality of the CPNApplication
 * by adding a new palette to the gui.
 * It reads the drawings - which hold the Net-Components - from the
 * ".../tools" directory without opening them. Instead the figures are stored
 * and will be inserted into the current drawing when useing the tools.
 * To create the palette there have to be, in addition to each tool drawing in the
 * ".../tools" directory - also three gif images located in ".../tools/images"
 * directory.
 *
 * 0.3 Now sequence ist determined by the ".../tools/.sequence" file
 *
 * 0.4 The tools directory is now determined by the user proterty "user.toolsdir"
 *     which defines the parent directory  of tools. So the tools are now in
 *     user.toolsdir+"tools/". If no user.toolsdir is specified the current
 *     directory is parent directory to tools.
 *
 *
 *
 * @author jo
 * @version 0.4,  June 2002
 *
 */
```

```
public class MApp extends CPNApplication {

    private Vector _additionalPalettes;

    protected MApp()
    {
super();
    }
    protected MApp(String[] args)
    {

super(args );
    }

    /**
     * Creates in addition to to usual tool palettes also the palette for the
     * Net-Komponents
     */
    protected void createTools(Panel toolPanel) {

// create all usual tools
super.createTools(toolPanel);

// create tools for Net-Components
Tool tool;
  Palette palette1 = new Palette(menuFrame,"Komponents");

// *** have to decide still where to put those
String dirName     = System.getProperty("user.toolsdir");
if(dirName == null)
    dirName =".";
dirName += "/tools/";
File dir           = new File(dirName);
File imageDir      = new File(dirName + "/images");

// read the sequence
Vector v_seq = new Vector();
try{
    BufferedReader seq = new BufferedReader( new FileReader(dirName + ".sequence"));
while (seq.ready()){
v_seq.add(seq.readLine());
    }
    seq.close();
```

```
}catch (Exception e) {System.out.println(e);}

// check if "tools" directory exists
if (dir.isDirectory()){
    if (imageDir.isDirectory()){

// add all renew drawings that are not in sequence list
String[] fileNames = dir.list();
for(int i=0; i < fileNames.length; i++){
    String s = StringUtil.getFilename(fileNames[i]);
    String ext = StringUtil.getExtension(fileNames[i]);
    // check if file is a renew drawing
    if ((new File(dirName + fileNames[i])).isFile() && "rnw".equals(ext))
if (! v_seq.contains(s))
    v_seq.add(s);
}
Iterator it = v_seq.iterator();
while (it.hasNext()){
    String toolName =(String)it.next();
    String fileName = toolName + ".rnw";

    Drawing drawing = loadDrawing(dirName + fileName);

    // have to put all the figures each by each into the Vector
    // any other way? ****
    FigureEnumeration fe = drawing.figures();
    Vector v = new Vector();
    while (fe.hasMoreElements()){
v.addElement((Figure)fe.nextElement());
    }

    tool = new InsertionTool(this, v);
    //String toolName = StringUtil.getFilename(fileName);
    String imageName = "";
    // check Images
    // in case of missing gifs use default, plain button
    // (1.gif 2.gif 3.gif)
    if ((new File(dirName + "images/" + toolName + "1.gif")).exists()){
imageName = toolName;
    }


    palette1.add(createToolButton(dirName+"images/"+ imageName,
```

```
  "NC "+ toolName,
  tool));


// }  // if
} // while it has next
//  tool = new CreationTool(this, new NewRectangleFigure());

//  palette1.add(createToolButton(dirName+"images/"+ "in", "NewRectangle Tool", tool)

 toolPanel.add(palette1);

    } // images dir exists
    else {System.err.println("The directory "+ dirName +  "/images does not exist!");
}  // tools dir exists
else {System.err.println("The directory "+ dirName + " does not exist!");}
    }  // createTools




    //-- main -----------------------------------------------------------

    public static void main(String[] args) {

//  Vector additionalPalettes = new Vector();
//  additionalPalettes.add("tools");
//  Vector v_args = new Vector();
//  for(int i=0; i < args.length; i++){
//      if (args[i].startsWith("-X")){

//  additionalPalettes.add( args[i].substring(2));
//  System.out.println( args[i].substring(2));
//      }else {
//  v_args.add(args[i]);
//      }
//  }

//  String[] a = objects2Strings(v_args);

      new MApp(args);
    }
```

```
//     static private String[] objects2Strings(Vector v){
//  Object[] os = v.toArray();
//  String[] ss = new String[os.length];
//  for (int i=0; i < os.length; i++){
//      ss[i] = (String) os[i];
//  }
//  return ss;
//     }



}

/*
 * @(#)InsertionTool.java
 *
 */

package de.renew.gui;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.util.Vector;
import java.util.Iterator;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;


/**
 * A tool for inserting a vector of figures into a drawing .
 * All figures will be inserted individually
 * into the current drawing.
 * After insertion the inserted figures will automatically be selected
 * for convenient location adjustment.
 *
 * @author jo
 * @version 0.2,  June 2002
 */

public  class InsertionTool extends UndoableTool {
```

```
    /**
     * the vector of figures to be drawn
     */

    private Vector _fv;
    /**
     * the vector of figures that was drawn
     */

    private Vector _selectedf;

    /**
     * Constructs a Tool for inserting a vector of figures
     */

    public InsertionTool(DrawingEditor editor, Vector fv) {
super(editor);
_fv = fv;
    }

    /**
     * inserts the vector of figures into the drawing at the given position
     * @param e - the mouse event
     * @param x,y - the position
     */

    public void mouseDown(MouseEvent e, int x, int y) {

try{
    Vector v = _fv;

    FigureSelection fs = new FigureSelection(v);
    Vector figures = (Vector)fs.getData(FigureSelection.TYPE);
    _selectedf = figures;
    view().addAll(figures);
    view().addToSelectionAll(figures);
    StandardDrawingView.moveFigures(figures,x,y);
    // view().clearSelection();
    view().checkDamage();
}
catch(Exception ex){System.err.println(ex);}
changesMade();
```

```
    }



   /**
     * Nothing to do
    */
    public void mouseDrag(MouseEvent e, int x, int y) {
//view().moveSelection(x,y);

    }

   /**
     * Reset tool choice to default.
    */
  public void mouseUp(MouseEvent e, int x, int y) {
      editor().toolDone();
      view().addToSelectionAll(_selectedf);
  }

}
```