Spezifikation einer Ausführungssemantik für das Subjektorientierte Prozessmanagement mit CoreASM

Bachelor-Thesis von André Wolski
Tag der Einreichung:

1. Gutachten: Prof. Dr. Max Mühlhäuser
2. Gutachten: Stephan Borgert

TECHNISCHE UNIVERSITÄT DARMSTADT

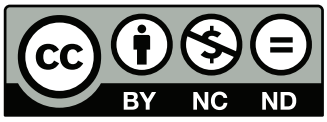
Fachbereich Informatik
Telekooperation
Prof. Dr. Max Mühlhäuser

Spezifikation einer Ausführungssemantik
für das Subjektorientierte Prozessmanagement mit CoreASM

Vorgelegte Bachelor-Thesis von André Wolski

1. Gutachten: Prof. Dr. Max Mühlhäuser
2. Gutachten: Stephan Borgert

Tag der Einreichung:



Abstract

The Subject-oriented Process Management (S-PM) enables a clearly visual modeling of concurrent systems, especially of technical processes and business processes (there: Subject-oriented Business Process Management, S-BPM) [1].

In order to reach a common understanding of the S-PM language features used during process modeling, validation and execution a precise execution semantic is needed. Such an execution semantic can be specified with the formalism of Abstract State Machines (ASM) [2] and executed with the CoreASM framework [3]. An executable specification of the S-PM language features allows an abstract execution of process models and therefore a mutual validation between the execution semantic and the process models.

Existing works related to the Subject-oriented Business Process Management are either not based on a formal specification [4, 5, 6, 7, 8], are purely theoretical works without implementation [9, 10, 11, 12, 13] or cover only a limited scope of language features [14].

This work continues existing approaches for a formal specification of the Subject-oriented Business Process Management in order to reach an executable ASM specification with a complete language scope.

Therefore a structural model was developed that combines existing S-PM language features with additionaly required features. The specifications of language features from previous works have been transferred to CoreASM, adapted to the structural model and extended with the additional language features. The specification of the execution engine has been extended to allow an abstract execution of process models. For an interactive process validation a console application, that controls the abstract execution, has been developed and connected to the CoreASM framework.

Analogical to software testing techniques the correctness of the developed specification has been ensured through unit, integration and system tests, which also yield the proof of implementation. The unit tests ensure the correct execution of single specification sections whereas the integration tests ensure a complete process execution sequence to test specific features defined by crafted process models. For the system tests practical proven process models of the S-BPM community were used along technical process models from the PolyEnergyNet project [15], which were chosen as their extend and complexity were especially useful for a comprehensive proof of implementation. The abstract execution of those processes with the console application was used for the system tests and enabled a process validation.

Zusammenfassung

Das Subjektorientierte Prozessmanagement (Subject-oriented Process Management, S-PM) ermöglicht eine übersichtliche graphische Modellierung von nebenläufigen Systemen, insbesondere von technischen Prozessen und von Geschäftsprozessen (dort: Subject-oriented Business Process Management, S-BPM) [1].

Damit bei der Prozessmodellierung, -validierung bis hin zur -ausführung ein gemeinsames Verständnis über die Bedeutung der verwendeten S-PM Sprachelemente besteht wird eine eindeutige Ausführungssemantik benötigt. Eine solche Ausführungssemantik kann mit dem Formalismus der Abstrakten Zustandsmaschinen (Abstract State Machines, ASM) [2] spezifiziert und mit dem CoreASM Framework [3] ausgeführt werden. Eine ausführbare Spezifikation der S-PM Sprachelemente ermöglicht eine abstrakte Ausführung von Prozessmodellen und damit eine wechselseitige Validierung der Ausführungssemantik und der Prozessmodelle.

Bestehende Arbeiten zum Subjektorientierten Prozessmanagement basieren entweder nicht auf einer formalen Spezifikation [4, 5, 6, 7, 8], sind rein theoretische Arbeiten ohne Implementierung [9, 10, 11, 12, 13] oder bilden nur einen begrenzten Sprachumfang ab [14].

In dieser Arbeit wurden bestehende Ansätze zur formalen Spezifikation vom Subjektorientierten Prozessmanagement mit dem Ziel weiterentwickelt, eine ausführbare ASM Spezifikation mit vollem Sprachumfang zu erhalten.

Dazu wurde ein Strukturmodell aus vorhandenen S-PM Sprachelementen entwickelt und um zusätzliche benötigte Sprachelemente erweitert. Die Spezifikationen von Sprachelementen aus bestehenden Arbeiten wurden nach CoreASM übertragen, an das Strukturmodell angepasst und um die weiteren Sprachelemente ergänzt. Dabei wurde die Spezifikation der Ausführungseinheit so erweitert, dass eine abstrakte Ausführung von Prozessmodellen ermöglicht wurde. Zur interaktiven Prozessvalidierung wurde eine Konsolenanwendung entwickelt, die mit dem CoreASM Framework verbunden ist und die abstrakte Ausführung steuern kann.

Analog zu Testverfahren der Softwareentwicklung konnte durch Komponenten-, Integrations- und Systemtests die Fehlerfreiheit der erarbeiteten Spezifikation sichergestellt und damit der Machbarkeitsnachweis erbracht werden. Durch Komponententests wurde dabei die korrekte Ausführung einzelner Spezifikationsabschnitte kontrolliert und mit Integrationstests wurde der Ablauf in Testprozessen derart vorgeben, dass gezielt die Einhaltung gewünschter Merkmale auf Erfüllung getestet wurde. Für die Systemtests wurde neben praktisch erprobten Prozessmodellen der S-BPM Community auf Prozessmodelle des PolyEnergyNet Forschungsprojekts [15] zurückgegriffen, da diese technischen Modelle wegen ihres hohen Umfangs und hoher Komplexität für einen umfassenden Machbarkeitsnachweis besonders geeignet waren. Durch die Nutzung der Konsolenanwendung konnten diese Prozesse durch eine abstrakte Ausführung validiert und dadurch gleichzeitig die Systemtests durchgeführt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Mit der zunehmenden Digitalisierung unserer Gesellschaft werden immer mehr und immer komplexere Abläufe computergestützt ausgeführt. Zur Beschreibung und Kommunikation von Abläufen und Zuständigkeiten werden Prozessmodelle verwendet, die je nach Anforderung unterschiedlich detailliert sein können. Beispielsweise sind für die Kommunikation mit Domänenexperten und Entscheidern abstrakte Modelle ausreichend, während für eine Implementierung oder direkte Prozessausführung eine detaillierte und vollständige Modellierung notwendig ist.

Das Subjektorientierte Prozessmanagement (Englisch: Subject-oriented Process Management, S-PM) ist ein Paradigma zur Prozesserfassung und -modellierung. Zur Verhaltensbeschreibung orientiert es sich an der natürlichen Sprache, vor allem an den Elementen der Subjekte, Prädikate und Objekte. Subjekte sind die agierenden Komponenten eines Prozesses, Prädikate beschreiben die Aktionen eines Subjektes, und Objekte können als Nachrichten zwischen Subjekten ausgetauscht werden. Durch den Fokus auf die Interaktionen zwischen den Subjekten ist es besonders geeignet um komplexe Zusammenhänge übersichtlich darzustellen [1].

Die graphenbasierte Modellierung basiert auf dem *Parallel Activity Specification Scheme* (PASS) [17, 18] und verwendet graphische Notationen zum Design der Prozessmodelle. Ausgehend von einer groben Erstellung der Prozessmodelle können diese schrittweise agil bis zu einer Ausführung mit einer Prozess-Engine verfeinert werden.

Vor einer tatsächlichen Ausführung oder Implementierung von Prozessmodellen müssen diese systematisch und umfassend auf mögliche Fehler untersucht werden, vor allem wenn die Prozesse automatisiert, mit nur wenigen Benutzerinteraktionen und/oder organisationsübergreifend ausgeführt werden. Ein manuelles Testen von Prozessmodellen ist durch die zunehmende Komplexität der Prozesse nicht mehr praktikabel und muss durch Computerwerkzeuge unterstützt werden.

Für die Modellierung, Verifikation, Validierung und auch für die Ausführung von S-PM Prozessmodellen existieren derzeit mehrere konkurrierende Anwendungen, die unterschiedliche Zielsetzungen verfolgen [19]. Durch die konzeptionellen Unterschiede haben sich Dialekte mit unterschiedlichen Sprachelementen gebildet. Für einen reibungslosen Austausch von Prozessmodellen zwischen unterschiedlichen Anwendungen wird eine definierte Mindestmenge von Sprachelementen sowie ein gemeinsames Verständnis von deren Ausführungsverhalten benötigt.

Nur wenn sich alle Anwendungen exakt an eine gemeinsame Semantik der Sprachelemente halten ist es garantiert, dass ein Prozessmodell sich bei der Ausführung mit einer Prozess-Engine wie erwartet verhält. Langfristig ermöglicht dies zudem eine organisationsübergreifende Ausführung von Prozessmodellen, selbst wenn die beteiligten Organisationen und Unternehmen Prozess-Engines unterschiedlicher Hersteller einsetzen.

Ziel dieser Arbeit ist eine formale Spezifikation einer Ausführungssemantik für subjektorientierte Prozessmodelle und einer zugehörigen Ausführungseinheit, mit der eine abstrakte Prozessausführung zur computergestützten interaktiven Prozessvalidierung durchgeführt werden kann.

Zur Definition der Ausführungssemantik wird die Spezifikationssprache der Abstrakten Zustandsmaschinen (Abstract State Machines, ASM) [2] verwendet. Die Ausführbarkeit der Spezifikation wird durch das CoreASM Framework [3] ermöglicht und erlaubt es, durch eine abstrakte Prozessausführung die Spezifikation auf Fehlerfreiheit zu testen. Die interaktive Prozessvalidierung wird durch eine Konsolenanwendung ermöglicht, mit der das Verhalten der Ausführungseinheit anhand von praxisnahen Prozessmodellen weiter untersucht und mit anderen Werkzeugen verglichen werden kann.

Im folgenden Kapitel 2 werden die für diese Arbeit benötigten Grundlagen vorgestellt. Zuerst werden die wichtigsten Elemente und Methodiken des Subjektorientierten Prozessmanagements erläutert, um ein grundlegendes Verständnis der Sprachelemente zu vermitteln. Im Anschluss werden die Abstrakten Zustandsmaschinen und das CoreASM Framework erklärt.

Im dritten Kapitel werden verwandte Arbeiten vorgestellt und verglichen. Dabei werden Anforderungen an eine Spezifikation für eine Ausführungssemantik aufgestellt. Im vierten Kapitel werden Sprachelemente und Konzepte bestehender Arbeiten kombiniert und durch neue ergänzt, so dass ein aktualisiertes Modell entsteht, das die vorher aufgestellten Anforderungen erfüllen kann. Im fünften Kapitel wird die Spezifikation des Subjektinterpreters vorgestellt. Im sechsten Kapitel wird die Ausführungssemantik aller Aktionen spezifiziert. Im siebten Kapitel wird die Implementierung anhand von vereinfachten Teilprozessen aus einem Forschungsprojekt und einer automatisierten Testumgebung überprüft. Zum Abschluss wird im achten Kapitel ein Fazit gezogen und ein Ausblick gegeben.

2 Grundlagen

2.1 Subjektorientiertes Prozessmanagement (S-PM)

Das *Prozessmanagement* befasst sich mit der strukturierten Erfassung und Dokumentation von Prozessen. Neben technischen Prozessen lassen sich Abläufe in Unternehmen erfassen, in dem Fall spricht man von *Geschäftsprozessverwaltung*, beziehungsweise Englisch: *Business Process Management (BPM)*.

Beispiel 1: *„Ein Anwender verfasst eine Problembeschreibung. Der Servicedesk prüft die Problembeschreibung."*

Betrachtet man Prozessbeschreibungen mit Sätzen der natürlichen Sprache, wie in Beispiel 1, so kann diese Beschreibung aus mehreren Perspektiven betrachtet werden. Ausgehend von den Satzgliedern Subjekt, Prädikat und Objekt ergeben sich aus dem Beispiel 1 die Subjekte „Anwender" und „Servicedesk", die Prädikate „verfasst" und „prüft" und das Objekt „Problembeschreibung". Viele Ansätze zur Prozessbeschreibung konzentrieren sich auf den Daten- oder Kontrollfluss, demnach stehen dort die Objekte und Prädikate im Mittelpunkt der Modellierung. Subjekte werden bei diesen Ansätzen entweder gar nicht betrachtet oder erst nachträglich oder informell ergänzt, was dazu führt, dass die Aufgaben der Subjekte, sowie die Interaktionen zwischen ihnen, nicht klar erkennbar sind.

Beim *Subjektorientierten Prozessmanagement (Englisch: Subject-oriented Process Management, S-PM)* wird der Gesamtprozess aufgeteilt in Teilabläufe einzelner Subjekte, die miteinander kommunizieren.

Die graphische S-PM Modellierung verwendet zwei Betrachtungsebenen: im ***Subject Interaction Diagram (SID)*** wird die Interaktion aller Subjekte dargestellt, das interne Verhalten der einzelnen Subjekte wird im ***Subject Behaviour Diagram (SBD)*** anhand eines gerichteten Graphen beschrieben. Dieser Modellierungsansatz ist aus dem *Parallel Activity Specification Scheme* (PASS) [17, 18] entstanden, welches sich allgemein mit der Spezifizierung von nebenläufigen Systemen befasst, und später für die Verwendung in Geschäftsprozessen vereinfacht wurde. Im Rahmen von Unternehmensabläufen wird vom *Subjektorientierten Geschäftsprozessmanagement (Englisch: Subject-oriented Business Process Management, S-BPM)* gesprochen.

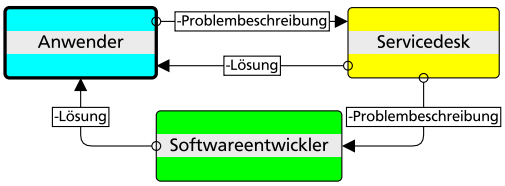


Abbildung 2.1: SID des Beispiel 2

Um das obige Beispiel 1 abzuschließen muss die Problembeschreibung nicht nur dem Servicedesk mitgeteilt werden, dieser muss sich auch darum kümmern, dass das Problem gelöst wird. Dazu wird das Beispiel 1 erweitert:

Beispiel 2: *Ein Anwender verfasst eine Problembeschreibung und erwartet eine Lösung. Der Servicedesk prüft die Problembeschreibung. Wenn das Problem durch den Servicedesk gelöst werden kann, wird dem Anwender direkt die Lösung mitgeteilt, ansonsten wird die Problembeschreibung an einen Softwareentwickler weitergegeben. Kann der Softwareentwickler das Problem lösen, teilt er dies dem Anwender mit.*

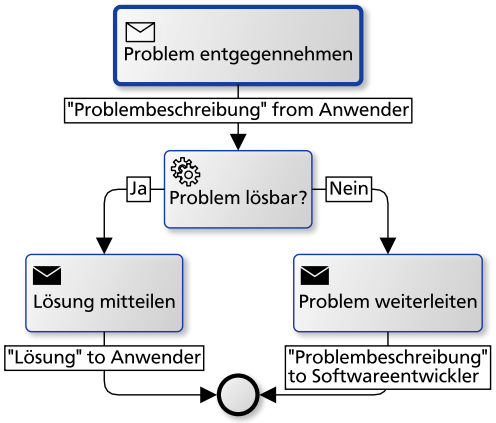


Abbildung 2.2: SBD des Subjekts Servicedesk aus Beispiel 2

Das Beispiel wurde also so ergänzt, dass dem Anwender nach Möglichkeit bereits vom Servicedesk eine Lösung mitgeteilt wird. Die zweite Ergänzung ist die Einführung eines weiteren Subjektes „Softwareentwickler", als Spezialist für den Fall, dass das Problem nicht durch den Servicedesk gelöst werden kann.

In Abbildung 2.1 wird das SID mit den drei ermittelten Subjekten dargestellt. Das Subjekt „Anwender" hat einen dicken Rahmen, um auszudrücken, dass dieses Subjekt den Prozess startet. Beim Subjektorientierten Prozessmanagement werden die Interaktionen von Subjekten dadurch abgebildet, dass sich Subjekte Nachrichten senden, um Objekte auszutauschen. Die möglichen Nachrichtentypen sind im SID als Verbindung zwischen den Subjekten zu sehen.

Das interne Verhalten von Subjekten gliedert sich in aufeinanderfolgende Aktionen; den Prädikaten aus der Prozessbeschreibung. Im Knoten der Aktion wird über ein Symbol angegeben, ob eine Nachricht gesendet oder empfangen wird,

oder ob es sich um eine interne Aktion ohne Interaktion mit einem anderen Subjekt handelt. In Abbildung 2.2 wird das SBD für das Subjekt „Servicedesk“ gezeigt. Die Startaktion ist durch einen dicken Rahmen gekennzeichnet und besteht aus dem Empfangen der Nachricht mit der Problembeschreibung. Nach der Entscheidung, ob das Problem direkt gelöst werden kann oder nicht, wird entweder eine Lösung an den Anwender gesendet oder die Problembeschreibung an den Softwareentwickler delegiert.

2.1.1 S-PM Methodik

Der Lebenszyklus eines Prozessmodells besteht aus vier aufeinanderfolgenden Phasen: Modellierung, Verifikation, Validierung und Ausführung. Aus jeder Phase kann zu der Modellierung zurückgekehrt werden, um das Prozessmodell anzupassen. Nach Änderungen am Modell muss es erneut verifiziert und validiert werden, bevor es ausgeführt werden kann. Diese Herangehensweise stellt sicher, dass ein Prozess ohne Laufzeitfehler ausgeführt werden kann und ermöglicht eine agile Modellierung, bei der das Prozessmodell schrittweise erweitert wird.

Bei der ***Verifikation*** wird ein Prozessmodell vollautomatisiert auf bestimmte Eigenschaften untersucht. Die Verifikation auf *Structural Soundness* stellt sicher, dass ein Prozessmodell keine strukturellen Fehler enthält. Ein struktureller Fehler wäre es beispielsweise, wenn ein Prozessmodell keine Startsubjekte hat. Die Verifikation auf *Interaction Soundness* stellt sicher, dass es, vor allem bei der Interaktion von Subjekten, zu keinen Verklemmungen kommt. Verklemmungen lassen sich in Deadlocks und Livelocks unterteilen. Beispiel 2 ist so formuliert, dass es zu einem Deadlock kommen kann, wenn dem Anwender keine Lösung mitgeteilt wird, da dieser dann endlos auf eine Lösung wartet. Ein Livelock ist ein Teilablauf, der endlos ausgeführt und nicht mehr verlassen werden kann. In Beispiel 2 wäre dies der Fall, wenn weder vom Servicedesk noch vom Softwareentwickler eine Lösung erstellt werden kann, und sich diese beiden Subjekte ständig die Problembeschreibung zuweisen würden.

Bei der ***Validierung*** wird untersucht, ob ein Prozessmodell praxistauglich ist und die beabsichtigten Ergebnisse erreicht werden [20, Abschnitt 5.2]. Zur Sicherstellung, dass das Prozessmodell frei von logischen Fehlern ist und dass das Ausführungsverhalten den Erwartungen der Prozessautoren entspricht, werden systematisch die möglichen Prozessabläufe betrachtet. Da die Arbeitschritte hierbei nicht tatsächlich durchgeführt werden, wird von einer *abstrakten Ausführung* gesprochen. Die Validierung kann je nach Anforderung der Organisation und Komplexität des Prozessmodells unterschiedlich detailliert erfolgen. Während in einfachen Fällen eine beispielhafte Besprechung der Prozessabläufe ausreicht können auch Rollenspiele mit mehreren zukünftigen Anwendern und neutralen Beobachtern eingesetzt werden. Insbesondere für komplexe Prozessmodelle muss die Validierung durch IT-Werkzeuge unterstützt werden, die den aktuellen Ausführungszustand und die möglichen Arbeitsschritte darstellen.

Die ***Ausführung*** von einem Prozessmodell findet dagegen in der Produktivumgebung mit echten Anwendern und realen Daten statt. Dazu kann entweder das Prozessmodell direkt durch eine S-PM Prozess-Engine ausgeführt werden oder alternativ dazu als Grundlage einer weiterfolgenden Umsetzung dienen. Beispielsweise kann das Prozessmodell zur Dokumentation von Arbeitsanweisungen verwendet werden, in andere Modellierungssprachen transformiert werden oder eine Vorgabe zur Entwicklung von Softwarekomponenten sein.

Weder die Verifikation noch die Prozessausführung in der Produktivumgebung sind Gegenstand dieser Arbeit und werden hier nur der Vollständigkeit halber aufgeführt. Für die Verifikation auf Interaction Soundness werden endliche Automaten aufgebaut, die den Zustandsraum des Prozessmodells vollständig abbilden. In dieser Arbeit wurde daher vorrausschauend bei Designentscheidungen auf die Vermeidung von Konstrukten, die zu einem unendlichen Zustandsraum führen können, geachtet.

2.1.2 Subjektinteraktionen

Um ein Prozessmodell auszuführen wird, sowohl bei der abstrakten als auch bei der konkreten Ausführung, eine ***Prozessinstanz*** angelegt. Von einem Prozessmodell können gleichzeitig mehrere unabhängige Instanzen ausgeführt werden. Beim Start einer Prozessinstanz wird das Verhalten aller Startsubjekte gestartet, die weiteren Subjekte werden erst bei Bedarf gestartet, wenn sie eine Nachricht erhalten. Das Subjektverhalten wird von ***Agenten*** ausgeführt. In Geschäftsprozessen sind dies üblicherweise Mitarbeiter eines Unternehmens, Agenten können jedoch auch technische Komponenten oder Softwarelösungen sein.

Diese Trennung ermöglicht es, dass ein Agent das Verhalten mehrerer Subjekte ausführt. In Bezug auf Beispiel 2 kann es beispielsweise in einer IT-Abteilung vorkommen, dass ein Softwareentwickler nebenbei den Servicedesk unterstützt, wenn es dort zu Engpässen kommt. Ebenso kann auch ein Mitarbeiter aus der IT-Abteilung ein Computerproblem haben und als Anwender mit einer Problembeschreibung auftreten.

Auf der anderen Seite kann ein Subjekt auch das Verhalten mehrerer unabhängiger Agenten definieren, in dem Fall wird es ***Multisubjekt*** genannt. Ein klassisches Beispiel der S-BPM Community ist die Angebotsanforderung. Dies ist

ein Prozess, bei dem der Einkauf eines Unternehmens den Vertrieb mehrerer anderer Unternehmen kontaktiert, um ein Angebot für ein Produkt oder eine Dienstleistung anzufordern, um im Anschluss das beste Angebot zu wählen und zu bestellen. Dabei wird das Verhalten für das Subjekt „Vertrieb“ für jedes kontaktierte Unternehmen unabhängig voneinander durch unterschiedliche Agenten ausgeführt.

Eine Referenz von einem Agenten zu einem Subjekt wird als ***Channel*** gespeichert. Diese Kommunikationskanäle können zwischen Subjekten ausgetauscht werden, damit bei komplexen Prozesschoreographien sichergestellt werden kann, dass alle Sender die gleichen Empfänger verwenden. Das Konzept dieser Weitergabe wird als *Mobility of Channels* bezeichnet [21].

In Beispiel 2 ist die Kommunikation in einer Dreiecksbeziehung strukturiert. Damit für den Softwareentwickler eindeutig ist, an welchen Agenten er die Lösung senden soll, muss er vorher vom Servicedesk den Channel zum Anwender erhalten haben.

2.1.3 Subjektverhalten

Das Verhalten von Subjekten wird durch einen gerichteten Graphen angegeben, dem Subject Behaviour Diagram (SBD).

Eine Aktion besteht aus einem Knoten und seinen ausgehenden Kanten, sie setzt sich aus der Knotenaktion und der Kantenaktion zusammen. Die Kanten der Aktion geben Bedingungen zum Verlassen der Aktion an und zeigen jeweils auf eine Folgeaktion. Sind die Bedingungen mehrerer Kanten erfüllt, so muss der Agent wählen welche Kante zum Verlassen der Aktion genommen werden soll. Es gibt die Möglichkeit jeweils eine Kante für einen automatischen Timeout oder einen manuellen Abbruch hinzuzufügen. Wenn die Knotenaktion beendet ist und eine Kante zum Verlassen gewählt wurde, wird die Kantenaktion ausgeführt. Im Anschluss an die Kantenaktion wird die Aktion deaktiviert und die Folgeaktion aktiviert.

Die Startaktion des SBD wird durch einen dickeren Rahmen gekennzeichnet und die Endzustände werden durch einen Kreis mit einem dicken Rahmen dargestellt.

In Abbildung 2.3 wird ein Prozess zur Beantragung, Genehmigung und Vorbereitung einer Dienstreise gezeigt. Es wird jeweils das Subject Behaviour Diagram (SBD) der beiden Subjekte „Mitarbeiter“ und „Vorgesetzter“ gezeigt. Zur übersichtlicheren Darstellung wird von nun an in dieser Arbeit eine Kombination des SID und des SBD in einer Abbildung verwendet.

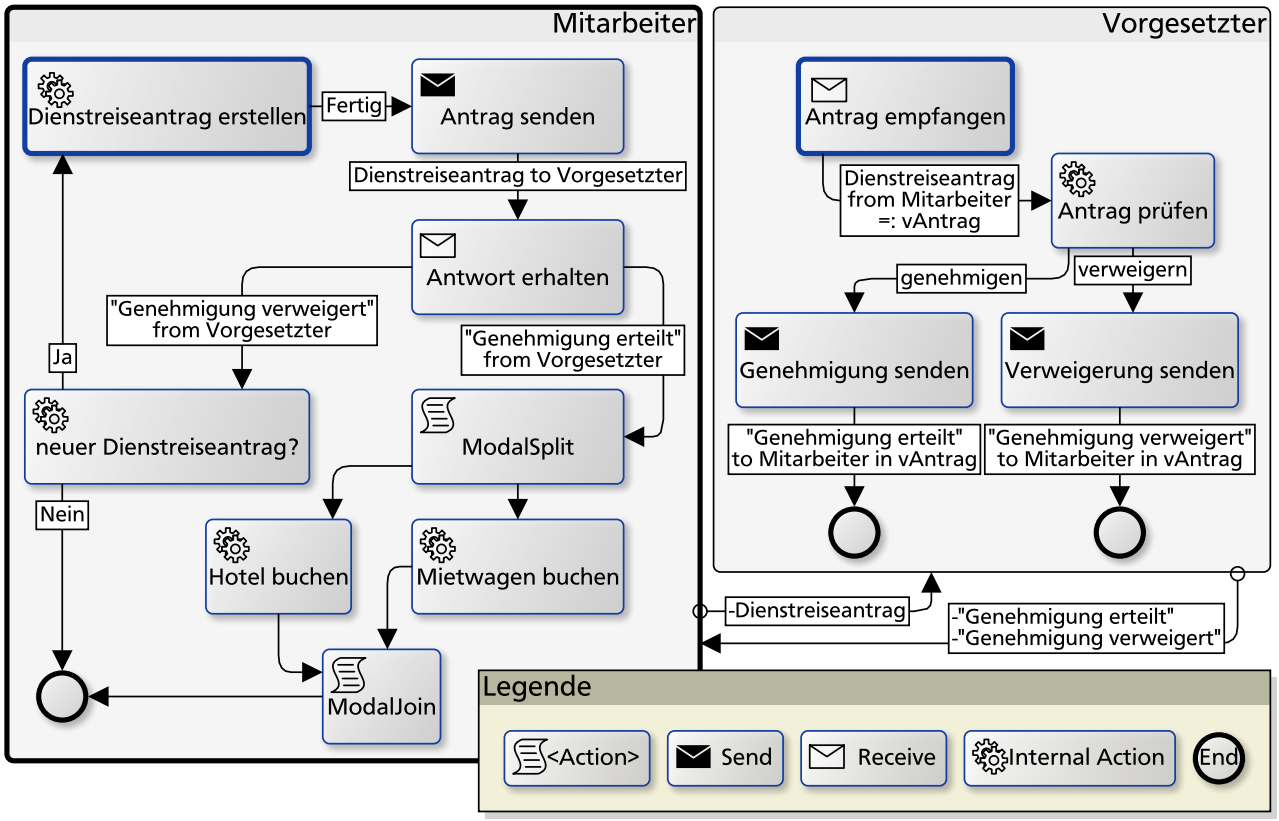


Abbildung 2.3: Dienstreiseantrag, nach [12, Abb. 2.3, S. 12], erweitert um Hotel- und Mietwagenbuchung

Das Verhalten des Mitarbeiter-Subjekts fängt mit der Erstellung seines Dienstreiseantrags an. Für diese Aktion wird eine ***Internal Action*** verwendet, mit der Aktionen des Subjekts in natürlicher Sprache formuliert werden. An der Kantenbeschriftung wird angegeben unter welchen Bedingungen die Aktion verlassen werden kann.

Wenn die Erstellung des Dienstreiseantrags abgeschlossen ist wird dieser mit einer Send Aktion versendet. Die ***Send*** Aktion versucht eine Nachricht an einen oder mehrere Empfänger zu senden. Sie ist beendet sobald die Nachricht erfolgreich jedem Empfänger zugestellt werden konnte. Dabei werden der Nachrichtentyp und das Empfängersubjekt in der Kantenbeschriftung angegeben.

Das Mitarbeiter-Subjekt wartet nun mit einer Receive Aktion auf den Eingang von Nachrichten. Der ***Receive*** Aktion werden an den ausgehenden Kanten Parameter für die zu empfangenen Nachrichten angegeben. Sobald eine passende Nachricht vorliegt wird die entsprechende Kante freigeschaltet und die Nachricht kann empfangen werden.

Der weitere Verhaltensablauf entscheidet sich nun anhand der Nachricht, die vom Vorgesetzter-Subjekt gesendet wurde. Wurde die Dienstreise verweigert, hat der Agent des Mitarbeiter-Subjekts die Auswahl, ob er einen erneuten Dienstreiseantrag senden möchte oder nicht. Entscheidet er sich gegen einen erneuten Antrag beendet die ***End*** Aktion das Subjektverhalten.

Falls die Dienstreise genehmigt wurde kann der Mitarbeiter diese vorbereiten, indem er ein Hotel und einen Mietwagen bucht. Die Reihenfolge dieser beiden Aktionen ist nicht entscheidend, und es können sogar beide Aktionen nebenläufig ausgeführt werden. Beispielsweise kann sich der Agent des Mitarbeiter-Subjekts um die Buchung des Mietwagens kümmern, während er auf Rückfragen zur Hotelbuchung wartet.

Solche parallelen Abläufe werden mit der ***Modal Split*** Aktion eingeleitet: diese aktiviert alle Folgeaktionen und beginnt damit mehrere parallelen Ausführungspfade. Zur Zusammenführung der Ausführungspfade wird die ***Modal Join*** Aktion verwendet.

Das Verhalten des Vorgesetzter-Subjekts ist in diesem Beispiel deutlich einfacher: es beginnt mit dem Empfangen eines Dienstreiseantrags. Die empfangene Nachricht, die neben dem Antrag als Inhalt auch den Absenderchannel beinhaltet, wird in der Variablen „vAntrag“ gespeichert. In einer Internal Action wird der Antrag geprüft; je nach Prüfungsergebnis wird dem Mitarbeiter eine Genehmigung oder Verweigerung gesendet. Zum Senden der Antwort wird hier der Channel des Mitarbeiters aus der Variablen vAntrag gelesen, um eine manuelle Agentenwahl zu vermeiden, bei der der Vorgesetzte fälschlicherweise auch einen anderen Mitarbeiter hätte wählen können. Nach dem Senden ist das Verhalten des Vorgesetzter-Subjekts abgeschlossen.

Entscheidet sich der Agent des Mitarbeiter-Subjekts nach einer Verweigerung einen erneuten Dienstreiseantrag zu senden wird das Verhalten des Vorgesetzter-Subjekts neu gestartet.

2.1.4 Prozessnetzwerke

Die beiden gezeigten Prozessmodelle der Abbildungen 2.1 und 2.3 sind als einzelne und eigenständige Prozesse modelliert. Üblicherweise sind Prozesse jedoch nicht eigenständig, sondern mit anderen Prozessen verbunden. Im Fall einer Dienstreise schließt sich nach der Genehmigung durch den Vorgesetzten ein Prozess zum Buchen von Unterkunft und Anfahrt an, sowie nach der Dienstreise eine Reisekostenabrechnung. Solange diese Teilabläufe innerhalb der gleichen Organisation ausgeführt werden, lassen sich alle Abläufe in einem einzelnen Prozessmodell erfassen. Mit einem über mehrere Organisationen verteilten Prozess ist dies jedoch nicht mehr möglich. Im Falle der Dienstreise ließe sich zwar die Reisekostenabrechnung noch innerhalb des Dienstreiseprozesses abbilden – übersichtlicher ist es jedoch für die Reisekostenabrechnung ein separates Prozessmodell zu entwerfen, das mit dem Prozessmodell der Dienstreise verknüpft wird.

Für die Buchung einer Unterkunft erstreckt sich der Gesamtprozess jedoch über mehrere Organisationen. Damit jede Organisation die Hoheit über ihre Prozessmodelle hat und interne Abläufe nicht veröffentlichen muss, erfolgt dazu eine Aufteilung in mehrere verbundene Prozessmodelle.

Zur Verknüpfung von Prozessmodellen werden ***Interfacesubjekte*** verwendet. Interfacesubjekte dienen in einem Prozessmodell als Platzhalter und haben kein eigenes Verhalten - stattdessen haben sie eine Referenz auf ein tatsächliches Subjekt aus einem anderen Prozessmodell, welche erst zur Laufzeit aufgelöst wird. Das verwiesene Prozessmodell wird als ***externer Prozess*** bezeichnet und das verwiesene Subjekt wird als ***externes Subjekt*** bezeichnet. In der graphischen Modellierung werden Interfacesubjekte durch einen gestrichelten Rahmen dargestellt. Die Referenz des Interfacesubjektes kann auch offen gelassen werden, um eine Verknüpfung zu beliebigen externen Subjekten zu ermöglichen. Solche Prozesse werden als ***Service Prozess*** bezeichnet.

Zur Veröffentlichung von einem Prozessmodell, beispielsweise um es auf einer Plattform anzubieten, kann von dem Modell eine anonymisierte Ansicht erstellt werden, in der nur die Interfacesubjekte und deren Kommunikationspartner vorhanden sind. Das Verhalten der Subjekte kann dabei reduziert werden auf ein äquivalentes anonymisiertes Verhalten [22, 23].

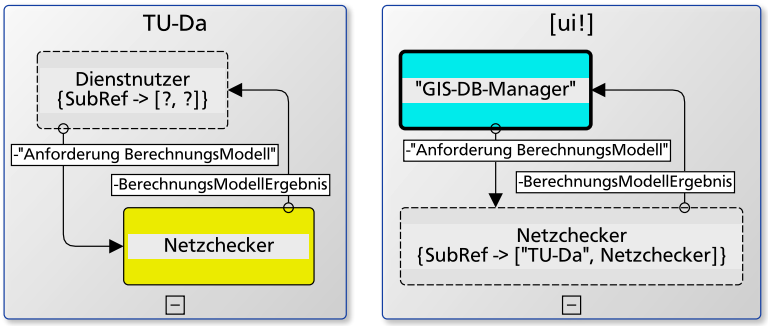


Abbildung 2.4: Extern und IP-Limit (SID), aus [16]

In Abbildung 2.4 werden zwei Prozessmodelle mit jeweils einem Interfacesubjekt gezeigt. Die Modelle stammen aus dem PolyEnergyNet Forschungsprojekt [15], bei dem technische Prozesse mit Hilfe von S-PM modelliert wurden, um widerstandsfähige Energienetze zu erforschen.

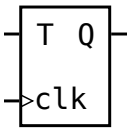
Das linke Prozessmodell ist ein Service Prozess, zu erkennen an der offengelassenen Referenz des Interfacesubjektes, die durch zwei Fragezeichen dargestellt wird. Das „Netzchecker“ Subjekt nimmt Berechnungsmodelle entgegen, prüft diese, und sendet ein Ergebnis zurück. Der Prozess wurde als Service Prozess entworfen, da es für die Berechnung unerheblich ist, von welchem Subjekt ein Berechnungsmodell stammt.

Auf der rechten Seite wird dieser Service Prozess eingebunden, um Berechungsmodelle vor der Speicherung in einer geographischen Datenbank (GIS-DB) zu prüfen.

2.2 Abstrakte Zustandsmaschinen (Abstract State Machines, ASM)

Abstrakte Zustandsmaschinen (englisch: Abstract State Machines, ASM) sind eine formale Sprache zur Systemspezifikation und -analyse. Mit ihnen wird ein Zustand und seine schrittweise Änderung beschrieben. Eine umfassende Vorstellung und Definition findet sich im Buch [2] (2003) von Egon Börger und Robert Stärk.

ASMs erlauben es ein System, beispielsweise technische Komponenten oder Algorithmen, in einer pseudocodeähnlichen Weise zu spezifizieren. Mit ihnen können neben einfachen endlichen Automaten auch Registermaschinen und virtuelle Maschinen erfasst werden; insbesondere lässt sich jede Turingmaschine durch eine ASM abbilden.




```
rule TFlipFlop(Q, T) = {
  if Q = 0 then { if T then Q := 1 else skip }
  if Q = 1 then { if T then Q := 0 else skip }
}
```

Listing 1: Beispiel für einen einfachen endlichen Automaten

2.2.1 Grundlagen

Eine Abstrakte Zustandsmaschine hat einen Zustand, der durch ***transition rules*** schrittweise verändert wird.

Dazu hat jede ASM eine ***main rule***, welche regelmäßig für einen ***run*** der ASM ausgewertet wird. Von dieser Hauptregel aus können weitere Regeln aufgerufen werden. Die Auswertungen von Regeln erzeugen jeweils ein ***UpdateSet***, welches beschreibt wie der Zustand geändert werden soll. Nach der Auswertung der Hauptregel wird diese Änderungsmenge auf den aktuellen Zustand angewendet, um den nächsten Zustand für den nächsten Berechnungsschritt zu erzeugen. Die Hauptregel wird so oft ausgewertet, bis sich in einem *run* keine Änderungen mehr ergeben können.

ASMs sind als mathematisches Modell definiert, in dem jeder Zustand eine algebraische Struktur ist [2, Kapitel 2.4]. Zum Speichern von Werten werden Interpretationen von n-stelligen Funktionen in der Algebra des Zustandes festgehalten. Der Zustand der ASM kann somit als Abbildung von Funktionsstellen zu deren Werten betrachtet werden. Diese Definition ermöglicht eine rekursive formale Definition von Regeln und weitergehende formale Untersuchungen von ASM-Spezifikationen.

Die mathematische Definition eines Zustandes als Algebra erfordert, dass der Zustand neben der Belegung von Funktionen mit Werten außerdem alle möglichen Funktionswerte beinhaltet. Dazu beinhaltet der Zustand eine Signatur, welche alle Funktionsnamen und Konstanten beinhaltet. Die Signatur kann mit ***Universen*** erweitert werden um die möglichen Werte zu strukturieren, beispielsweise beinhaltet die Signatur immer das `BOOLEAN` Universum zur Abbildung auf Wahrheitswerte und den speziellen Wert **`undef`**, der verwendet wird um auszudrücken, dass einer Funktion kein Wert zugewiesen ist.

Universen können neben klassischen Datentypen, wie den Wahrheitswerten oder Zahlen, auch zur Laufzeit definiert und von einem Zustand zum Nächsten verändert werden. Einer Funktion kann optional ihr Definitions- und Wertebereich

auf Universen eingeschränkt werden.

Eine Stelle einer n-stelligen Funktion wird mit `f(a1, ..., an)` angegeben und als ***Location*** bezeichnet. 0-stellige Funktionen werden auch als *Konstante* bezeichnet, da sie keine Parameter haben.

Im Vergleich mit klassischen Programmiersprachen entsprechen Konstanten Variablen, Funktionen n-dimensionalen Arrays, die Algebra dem Hauptspeicher und Universen entsprechen Datentypen.

2.2.2 Regeln

Im Mittelpunkt der Abstrakten Zustandsmaschinen steht die ***Update Rule***: `l := t`. Sie erzeugt ein Update mit der Zuweisung der Location `l` auf den Wert des Ausdrucks `t`.

Im UpdateSet wird auf diese Weise für jeden run festgehalten, wie die Funktionsinterpretationen in der Algebra des nächsten Zustandes sein sollen. Für jeden Zustand wird somit eine neue Algebra geschaffen - daher wurden Abstrakte Zustandsmaschinen früher als *evolving algebras* bezeichnet.

ASM Regeln werden induktiv aus einer vordefinierten Regelmenge zusammengesetzt. Im Gegensatz zu Methoden und Funktionen klassischer Programmiersprachen bestehen Regeln somit nicht aus Anweisungsfolgen.

Die einfachste Regel ist die ***Skip Rule***: `skip`. Sie macht nichts und kann als Platzhalter verwendet werden.

Mit der ***Conditional Rule*** wird ein Ausdruck auf Wahrheit überprüft: `if φ then P else Q`. Falls φ wahr ist wird die Regel `P` ausgeführt, ansonsten die Regel `Q`. Als Vereinfachung muss der zweite Teil nicht angegeben werden, in dem Fall wird für Q `skip` angenommen.

Die Zusammensetzung von Regeln erfolgt üblicherweise über die ***Block Rule***: `{P , Q}`. Das Komma kann zur einfachen Lesbarkeit auch durch einen Zeilenumbruch ersetzt werden. Eine alternative Schreibweise lautet `P par Q`. Die Regeln `P` und `Q` werden parallel ausgeführt. Nach der Auswertung werden die beiden UpdateSets zu einem einzigen UpdateSet zusammengeführt. Hierbei kann es zu einem ***inkonsistenten UpdateSet*** kommen: Die Regel `{a := a + 1 , a := a + 2}` erzeugt beispielsweise immer ein inkonsistentes UpdateSet, da durch die parallele Auswertung der Wert von `a` sowohl um eins als auch um zwei erhöht werden soll. Wenn die Auswertung der Hauptregel einer ASM ein inkonsistentes UpdateSet erzeugt hat, wird es nicht angewendet; der Zustand der ASM verändert sich also nicht.

Als sequentielle Zusammensetzung wird die ***Sequence Rule*** verwendet: `P seq Q`. Die Regeln `P` und `Q` werden nacheinander ausgeführt. Für die Ausführung der Regel `Q` wird dazu ein Zwischenzustand erzeugt, auf dem das UpdateSet von `P` bereits angewendet wurde. Nach der Ausführung von `Q` wird der Zwischenzustand entfernt und ihr UpdateSet auf das UpdateSet der Regel `P` angewendet. Wenn `P` jedoch ein inkonsistentes UpdateSet erzeugt hatte wird `Q` nicht ausgeführt.

Mit der ***Call Rule*** wird eine Regel aufgerufen: `[l ←] R(t1, ..., tn)`. Beim Aufruf von Regeln findet ein *call-by-name* statt, das heißt, dass die Vorkommnisse der Parameter innerhalb der Regel durch die übergebenen Terme und Locations substituiert werden. Eine Auswertung von Termen findet somit erst innerhalb der aufgerufenen Regel statt und nicht bereits beim Aufruf, wie es sonst bei Programmiersprachen mit dem Ansatz *call-by-value* und *call-by-reference* üblich ist.

Neben der Änderungsmenge kann mit der Call Rule auch direkt ein Ergebnis zurückgeben werden. Dazu kann optional eine Location vor dem Regelnamen mit einem Pfeil angegeben werden. In der aufgerufenen Regel wird dann der spezielle Funktionsname `result` durch die hier angegebene Location substituiert.

Mit der ***Let Rule*** wird ein Zwischenzustand angelegt, in dem der Wert des Ausdrucks t_i jeweils x_i zugewiesen wird und in dem dann die Regel `P` ausgeführt wird: `let x1 = t1 [, ..., xn = tn] in P`. Die Werte x_i können nicht verändert werden. Mit der Let Regel kann ein *call-by-value* emuliert werden, indem die Argumente von `R` nun vor dem Regelaufruf ausgewertet und zwischengespeichert werden: `let x1 = t1 , ..., xn = tn in R(x1, ..., xn)`.

Mit der ***Local Rule*** wird ein Zwischenzustand angelegt, in dem Änderungen an x_i nicht nach außen sichtbar sind: `local x1 [:= t1] [, ..., xn [:= tn]] in P`. Nach der Auswertung der Regel `P` wird dieser Zwischenzustand, und somit alle Änderungen an x_i, verworfen.

Mit der ***Import Rule*** wird ein neues Element erzeugt, das bisher nicht Teil der Signatur war, und `x` zugewiesen: `import x do P` . Dies ermöglicht es Universen zu erweitern: `extend D with x do P` kann als Vereinfachung für `import x do {D(x) := true seq P}` benutzt werden um das Universum `D` um ein Element zu erweitern.

Mit der ***Forall Rule*** wird eine Regel `P` parallel für alle Elemente `x` aufgerufen, welche φ erfüllen: `forall x with` φ `do P`

φ muss hierbei eine endliche Menge definieren, beispielsweise ein Universum, das mit einer Nebenbedingung beschränkt wird.

Mit der ***Choose Rule*** wird ein Element `x` ausgewählt das φ erfüllt: `choose x with` φ `do P` . Wenn es kein solches `x` gibt wird die Regel `P` nicht aufgerufen. Die Auswahl von `x` ist nicht deterministisch.

Mit der ***Iterate Rule*** wird eine Regel `P` so lange aufgerufen, bis sich etweder keine Änderungen mehr ergeben oder eine inkonsistente Änderungsmenge auftritt: `iterate P` . `while (cond) P` ist eine vereinfachte Schreibweise für `iterate (if cond then R)` .

```
rule SwapSort =
 choose i, j ∈ dom(a)
  with i < j and a(i) > a(j) do
   Swap(a(i), a(j))

rule Swap(x, y) =
  { x := y, y := x }
```

Listing 2: SwapSort, aus [2, Seite 40]

```
rule callByName =
  { a := 5, b := 10 }
  seq x <- inc(a+b)

rule inc(y) =
  let a = 10 in
    result := y + 1
```

Listing 3: callByName

In Listing 2 ist ein einfacher Sortieralgorithmus zu sehen, der eine Liste `a` sortiert: Bei jedem Aufruf der `SwapSort` Regel wird ein unsortiertes Paar ausgetauscht. Zum Austauschen der Elemente werden die jeweiligen Locations an die `Swap` Regel übergeben. Durch die Substitution der Parameter werden die Argumente `x` und `y` in der Regel durch diese Locations ersetzt und in der Auswertung werden somit folgende zwei Zuweisungsregeln parallel ausgeführt: `{ a(i) := a(j), a(j) := a(i) }`

Gibt es kein Paar das getauscht werden muss ist `a` sortiert und die Ausführung der `SwapSort` Regel erzeugt keine Änderungsmenge mehr. Hier ist die Mächtigkeit des *call-by-name* zu sehen und wird als Vorteil genutzt: es kann auf eine temporäre Variable verzichtet werden, was die Lesbarkeit der Regel verbessert. Zudem ist es hiermit möglich Locations als Parameter zu übergeben, die von der aufgerufenen Regel beschrieben werden können.

Dieses Prinzip kann auch weniger offensichtliche Auswirkungen haben und gerade bei größeren Regelsystemen unübersichtlich werden, wenn nicht auf den ersten Blick klar ist an welcher Stelle ein Term ausgewertet wird und wie der dortige lokale Zustand ist.

In Listing 3 werden in der `callByName` Regel zuerst die Konstanten `a` auf 5 und `b` auf 10 festgelegt. Mit der Sequence Rule wird ein Zwischenzustand angelegt, für den diese Updates bereits angewendet werden. Beim Aufruf der `inc` Regel wird als Parameter für `y` der Term `a+b` übergeben, das heißt dass in der `inc` Regel alle Vorkommen von `y` durch diesen Term ersetzt werden. Da beim Aufruf der `inc` Regel eine Zuweisung des Ergebnisses auf `x` angegeben ist wird jedes Vorkommen von `result` durch `x` ersetzt. Durch diese Ersetzungen lautet die auszuwertende Regel nun `let a = 10 in {x := (a+b)+1}` . Es wird also von der Call Rule das UpdateSet `{x -> 21}` erzeugt, da durch die Let Rule ein weiterer Zwischenzustand erzeugt wurde, bei dem `x` auf 10 festgelegt ist. Mit einem *call-by-value* wäre das Ergebnis dagegen 16 und die Let Anweisung ohne Auswirkung.

2.2.3 Funktionen

Anstatt Funktionen Werte zuzuweisen ist es auch möglich für eine Funktion eine Berechnungsvorschrift anzugeben oder den Wert aus einer externen Umgebung auszulesen.

Funktionen mit einer Berechnungsvorschrift werden als ***derived function*** bezeichnet. Die Berechnungsvorschrift wird als Term angegeben und kann neben den Argumenten auch vom Zustand abhängig sein. Einer abgeleiteten Regel kann kein Wert zugewiesen werden, der Term kann nicht verändert werden und sie ist immer frei von Seiteneffekten.

```
derived f(x) = a*x + b
```

Hier ist eine abgeleitete Funktion mit dem Parameter `x` gezeigt. Da `a` und `b` nicht als Parameter überliefert werden müssen sie im umgebenen Zustand definiert sein.

Wenn eine Funktion nicht abgeleitet ist wird sie als *basic function* bezeichnet. Diese Funktionen werden weiter unterteilt in ***static functions***, deren Werte nur im initialen Zustand bestimmt werden und nicht weiter verändert werden können, und *dynamic functions*, die ihren Wert von einem Zustand zum nächsten ändern können. Diese werden weiter unterteilt in vier Klassen:

monitored (auch *in*): werden ausschließlich durch die Umgebung verändert und können von der ASM nur gelesen werden.

controlled: können von der ASM gelesen und verändert werden, jedoch von der Umgebung weder gelesen noch verändert werden.

shared: können sowohl von der Umgebung als auch von der ASM gelesen und verändert werden.

out: können nur von der Umgebung gelesen und nur von der ASM beschrieben werden.

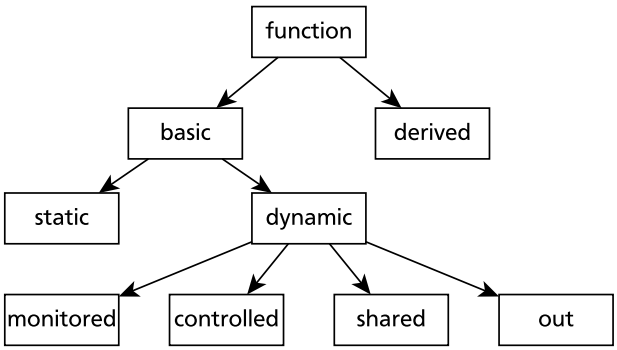


Abbildung 2.5: Funktionsklassen, aus [2, Fig. 2.4, p. 33]

2.2.4 Multi-Agent ASMs

Mit dem bisher vorgestellten ASM-Modell gibt es eine einzige Maschine mit einem Hauptprogramm. Ein verteiltes System ließe sich, wie in Listing 4 gezeigt, durch eine Regel je Komponente abbilden, wobei die Hauptregeln der Komponenten von der Mainrule aus aufgerufen werden.

```
rule Server = ...
rule Client1 = ...
rule Client2 = ...

rule main = {
  Server
  Client1
  Client2
}
```

Listing 4: ohne Multi-Agent

Mit Multi-Agent ASMs können solche verteilten Systeme einfacher abgebildet werden. Dazu wird das ASM-Modell um Agenten erweitert, die jeweils ihr eigenes Programm ausführen, sich jedoch einen gemeinsamen Zustand teilen. Die Agenten, und somit auch der globale Zustand, werden mit einer gemeinsamen logischen Uhr synchronisiert. Multi-Agent ASMs werden in die synchrone und asynchrone Variante unterteilt:

Bei ***synchronen Multi-Agent ASMs*** ist jeder Agent in jedem Schritt der ASM beteiligt. Dazu wird für jeden *run* der ASM die Hauptregel von jedem Agent ausgewertet. Die Auswertung der Hauptregel eines jeden Agenten wird als ***move*** bezeichnet. Am Ende des ASM Schrittes werden die UpdateSets aller *moves* der Agenten zu einem gemeinsamen UpdateSet des *runs* zusammengeführt. Wenn es konsistent ist wird es auf den Zustand der ASM angewendet, um den Folgezustand zu erzeugen.

Bei ***asynchronen Multi-Agent ASMs*** ist nicht festgelegt welche Agenten sich mit ihren *moves* am *run* der ASM beteiligen, daher wird daher von einem *partially ordered run* gesprochen. Die *moves* der Agenten werden weiterhin über die gemeinsame logische Uhr synchronisiert, die Programme der Agenten müssen nun jedoch so entworfen werden, dass sie auch unabhängig von den *moves* der anderen Agenten ausgeführt werden können. Die Auswahl der Agenten ist nicht vorgegeben und muss vor einer Ausführung der ASM so gewählt werden, dass es zu keinen Konflikten zwischen den Ergebnissen der Agenten kommen kann. Sie kann dabei so getroffen werden, dass es entweder eine Fairnessgarantie gibt oder nicht.

In beiden Fällen der Multi-Agent ASMs gibt es ein spezielles Universum `Agents` in dem sich alle Agenten befinden. Über die Funktion `self` kann jeder Agent eine Referenz auf sich selbst nachschlagen. Das ermöglicht es den gemeinsamen Zustandsraum in lokale Bereiche zu unterteilen, indem Funktionen einen zusätzlichen Parameter aus dem Agenten Universum erhalten.

Ein beliebtes Beispiel von einem verteilten System ist das Philosophenproblem (Englisch: dining philosophers problem). Hier sitzen fünf Philosophen um einen Tisch mit einer unbegrenzten Menge Reis und sind abwechselnd am Essen und am Denken. Zum Essen benötigt jeder Philosoph zwei Stäbchen. Da es im Haushalt jedoch nicht genügend Stäbchen für jeden Philosophen gibt wurde jeweils ein Stäbchen zwischen zwei Philosophen platziert. Es können also keine benachbarten Philosophen gleichzeitig essen, da sie sich die gemeinsame Ressource „Stäbchen“ teilen müssen. Das Problem besteht nun darin einen Ansatz zu finden in dem die Philosophen ohne Kommunikation *hinreichend* satt werden – vor allem darf kein Philosoph beim Denken gestört werden und es darf auch kein Philosoph verhungern.

Der naive Ansatz „nimm ein Stäbchen wenn es nicht in Benutzung ist und nimm dann das Stäbchen der anderen Seite wenn es nicht in Benutzung ist“ führt schnell zu einer Verklemmung in der alle Philosophen verhungern, indem benachbarte Philosophen jeweils ein Stäbchen haben und darauf warten dass das andere abgelegt wird. Wenn ein Philosoph in einer solchen Situation nach einiger Zeit sein Stäbchen ablegt kann das jedoch dazu führen, dass dieser verhungert, da immer einer seiner Nachbarn am Essen ist.

Ein fairer Ansatz basiert auf der Einführung einer weiteren Person, beispielsweise eines Kellners, bei dem sich Philosophen melden um von diesem entweder beide Stäbchen oder keines zugeteilt zu bekommen. In den meisten höheren Programmiersprachen wird dies meist über Lockingmechanismen so umgesetzt, dass das Betriebssystem garantiert dass kritische Aktionsfolgen nicht durch andere Threads unterbrochen werden können. Somit agiert das Betriebssystem als Kellner, welcher einem Philosophen das alleinige Recht zuteilt eine Aktionsfolge mit mehreren Ressourcen durchzuführen, ohne dass ein anderer Philosoph zwischenzeitlich mit diesen Ressourcen agieren kann.

In Listing 5 wird das Philosophenproblem mit Abstrakten Zustandsmaschinen beschrieben und gelöst. Dazu wird jedem Agenten die `Philosopher` Regel zugewiesen, welche nach Möglichkeit die `StartEating` Regel aufruft. Ein gleichzeitiger Aufruf dieser Regel durch zwei benachbarte Agenten im gleichen ASM-Schritt würde zu einem Konflikt führen, da dem gemeinsamen Stäbchen beide Agenten als Eigentümer zugewiesen werden würden. Aus der Definition von Multi-Agent-ASMs folgt jedoch, dass die Auswahl der an einem *run* beteiligten Agenten so gewählt werden muss, dass es zu keinem Konflikt kommen kann. Eine mögliche Agentenwahl wäre beispielsweise immer nicht-benachbarte Agenten zu wählen, um einen solchen Konflikt gar nicht erst entstehen zu lassen. In diesem Beispiel erfüllt daher die Agentenwahl die Rolle des Kellners, und es muss in der Spezifikation somit kein eigener Lockingmechanismus eingeführt werden. Bei der Umsetzung der Agentenwahl kann außerdem die Fairnessgarantie gegeben werden.

```
universe CHOPSTICK
function chopOwner: CHOPSTICK -> Agents
function eating : Agents -> BOOLEAN
function hungry : Agents -> BOOLEAN
function leftChop: Agents -> CHOPSTICK
function rightChop: Agents -> CHOPSTICK

init initRule
rule initRule = {
  // add 5 chopsticks
  CHOPSTICK(0) := true
  CHOPSTICK(1) := true
  CHOPSTICK(2) := true
  CHOPSTICK(3) := true
  CHOPSTICK(4) := true

  // add 5 philosophers
  forall x ∈ [0 .. 4] do {
    extend Agents with a do {
      program(a) := @Philosopher
      leftChop(a) := x
      rightChop(a) := ((x+1)%5)
      eating(a) := false
      hungry(a) := false
    }
  }

  program(self) := undef
}
```

```
// main program of every philosopher
rule Philosopher = {
  if hungry(self) ∧ ¬eating(self) then
    if CanPickBothChopsticks then
      StartEating
  if ¬hungry(self) ∧ eating(self) then
    StopEating
  choose b ∈ BOOLEAN do
    hungry(self) := b
}

rule StartEating = {
  chopOwner(leftChop(self)) := self
  chopOwner(rightChop(self)) := self
  eating(self) := true
}

rule StopEating = {
  chopOwner(leftChop(self)) := undef
  chopOwner(rightChop(self)) := undef
  eating(self) := false
}

derived CanPickBothChopsticks =
  (chopOwner(leftChop(self)) = undef)
  ∧ (chopOwner(rightChop(self)) = undef)
```

Listing 5: Philosophenproblem, nach [24]

2.2.5 Ausführung von Abstrakten Zustandsmaschinen

Abstrakte Zustandsmaschinen sind primär eine Spezifikationssprache und können nicht direkt durch Computer ausgeführt werden. Zur Ausführung muss die Spezifikation entweder durch einen Interpreter ausgewertet oder in Maschinencode oder eine Programmiersprache überführt werden. In allen Arten der Umsetzung muss sichergestellt werden, dass inkonsistente Änderungsmengen nicht auf den Zustand angewendet werden. Bei einer Unterstützung von Multi-Agent ASMs muss zudem ein Vorgehen zur Agentenwahl beachtet werden.

Eine Überführung in eine höhere Programmiersprache kann manuell durch Softwareentwickler oder automatisiert durch einen Compiler erfolgen. Dieser Ansatz ist für einen Einsatz in Produktivsystemen geeignet, da die Implementierung von anderen Softwarekomponenten aus genutzt werden kann [25]. Die Ausführung durch einen Interpreter bringt demgegenüber zwar Leistungseinbußen, jedoch können Änderungen an der Spezifikation direkt vorgenommen und zudem das Laufzeitverhalten mit einem Debugger untersucht werden.

Kiama ist eine in Scala eingebettete domänenspezifische Sprache [26]. Sie ermöglicht Softwareentwicklern eine intuitive Übertragung einer ASM Spezifikation nach Scala und anderen JVM-Sprachen. Der entstehende Quellcode ist eine Mischung der Sprachkonstrukte von Scala und von Abstrakten Zustandsmaschinen.

AsmL ist ein Compiler für die .NET Laufzeitumgebung, der von Microsoft Research entwickelt wurde [27]. Es wird ein sehr großer Funktionsumfang abgebildet, jedoch wurde das Projekt eingestellt.

CoreASM bietet einen Interpreter und einen Compiler nach Java an. CoreASM unterstützt durch einen Scheduler Multi-Agent ASMs und deckt alle hier vorgestellten Regeln ab, zudem werden durch Plugins viele Datentypen wie Listen oder Mengen direkt unterstützt [3].

CASM bietet einen ASM-Interpreter und einen Compiler nach C++ an, war jedoch zu Beginn dieser Arbeit noch nicht öffentlich verfügbar [28].

2.3 CoreASM

CoreASM ist ein Framework zum Entwurf und zur Ausführung von ASM Spezifikationen. Es wird unter der Academic Free License unter [29] bereitgestellt. Das Framework bietet eine Entwicklungsumgebung mit einem Interpreter und einem interaktivem Debugger an. Neben der Entwicklungsumgebung kann der Interpreter auch als Konsolenanwendung verwendet oder als Bibliothek eingebettet werden. CoreASM ist vollständig in Java entwickelt und kann somit auf verschiedenen Computerarchitekturen und Betriebssystemen verwendet werden. Mit dem TestEngineDriver wird eine JUnit Schnittstelle angeboten, die es ermöglicht Spezifikationen in einer Testumgebung zu überprüfen.

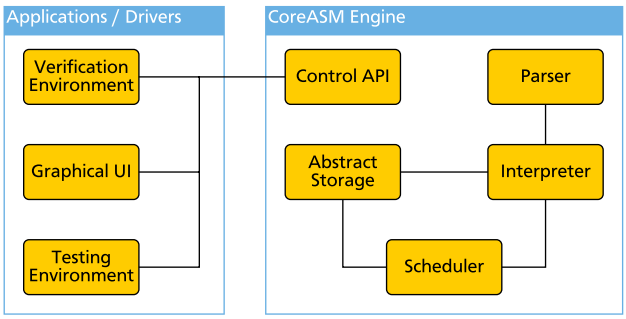


Abbildung 2.6: Architektur von CoreASM, nach [3, Fig. 4.2]

Mit dem Parser wird eine Spezifikation eingelesen. Die Ausführungszustände der ASM werden in der Abstract Storage Komponente gehalten. Der Scheduler koordiniert die Ausführung der Agentenprogramme durch den Interpreter. Die Agentenwahl besteht standardmäßig darin je *run* eine zufällige Teilmenge der Agenten zu wählen. Zur Konfliktvermeidung wird am Ende des runs das UpdateSet auf Konsistenz geprüft - sollte es inkonsistent sein wird es komplett verworfen und ein neuer *run* ausgewertet.

Die CoreASM Engine bietet eine Plugin-Schnittstelle an, mit der alle Komponenten durch eigene Javaklassen erweitert werden können. So ist es beispielsweise möglich eigene vordefinierte Universen, Funktionen und Regeln hinzuzufügen. Durch eigene Plugins werden häufig verwendete Datenstrukturen, Funktionen und Regeln angeboten, insbesondere gibt es bereits Implementierungen für die Universen `LIST`, `SET` und `MAP`. Für diese Datenstrukturen werden unter anderem die aus der funktionalen Programmierung bekannten Funktionen `filter`, `map` und `fold` implementiert.

Mit der `foreach` Regel wird eine Variante der `forall` Regel angeboten, die die Elemente sequentiell durchläuft. Die Syntax der Sequence Rule `A seq B seq C ..` lautet in CoreASM abweichend `seq A next B next C ..`. Als vereinfachte Schreibweise wird dafür außerdem `seqblock A , B , C , .. endseqblock` angeboten.

Für die Interaktion mit der Konsole und/oder dem Dateisystem werden spezielle *in* und *out functions* angeboten: mit der `input` function kann ein Text eingelesen, und mit der `print` Regel ausgegeben werden. Eine allgemeine Unterstützung zur Nutzung von *in* und *out functions* mit einer externen Umgebung ist in CoreASM jedoch nicht vorhanden.

```
function numres : -> NUMBER

derived RandomGuess(max) =
  return numres in
    choose x in [ 1 .. max ] do
      numres := x
```

Listing 6: CoreASM Beispiel derived via return

derived functions werden in CoreASM mit der `return` Regel erweitert um eine Möglichkeit den Rückgabewert nicht nur über einen Term, sondern auch über eine Zuweisung zu bestimmen. Nach der Auswertung des Rückgabewerts werden alle erzeugten UpdateSets verworfen, das heißt dass der Aufruf solcher *derived functions* weiterhin frei von Seiteneffekten ist. Diese Schreibweise erlaubt komplexere Anweisungen zur Auswertung des Ergebnisses als es mit reinen Termen möglich wäre.

In CoreASM ist es optional möglich Funktionssignaturen zu definieren und eine Typenprüfung zu aktivieren. Damit kann sichergestellt werden, dass einerseits keine undefinierten Funktionsnamen verwendet werden, als auch dass, gerade bei größeren Spezifikationen, Typisierungefehler vermieden werden.

Die Verwendung der `return` Regel und der Funktionssignaturen wird in Listing 6 gezeigt. Hier wird eine Funktion `numres` definiert, die nur numerische Werte annehmen kann. In der abgeleiteten Funktion `RandomGuess` wird sie verwendet, um den in der `choose` Regel gewählten Wert zurückzugeben.

Im ersten Teil dieses Kapitels wurde das Subjektorientierte Prozessmanagement (S-PM) vorgestellt. Mit dem Prozessmodell zum Dienstreiseantrag wurde ein subjektorientierter Geschäftsprozess gezeigt, mit dem Aktionen zum Senden und Empfangen von Nachrichten sowie zum Modellieren von parallelen Aktionen erklärt wurden. Im zweiten Teil wurde die Spezifikationssprache der Abstrakten Zustandsmaschinen (ASM) vorgestellt. Mit dem Philosophenproblem wurden dabei Multi-Agent-ASMs erklärt. Im letzten Teil wurde CoreASM vorgestellt, ein Framework zur Ausführung von ASM Spezifikationen.

Im nächsten Kapitel wird der aktuelle Forschungsstand zum Subjektorientierten Prozessmanagement in Hinblick auf eine formale Ausführungssemantik untersucht. Dabei werden Anforderungen an eine ausführbare Spezifikation aufgestellt, auf deren Basis eine interaktive Prozessvalidierung erfolgen kann.

3 State of the Art und verwandte Arbeiten

Zur Ausführung, Validierung und Verifikation von Prozessmodellen des Subjektorientierten Prozessmanagements (S-PM) existieren bereits einige quelloffene und auch kommerzielle Softwareprodukte. Obwohl sie gemeinsam aus dem Parallel Activity Specification Scheme (PASS) entstanden sind unterscheiden sich die vorhandenen Implementierungen stark im Funktionsumfang, den unterstützten Sprachelementen und deren Verhalten. Insbesondere sind Ergebnisse der Verifikation und Validierung nicht allgemein übertragbar auf eine tatsächliche Ausführung der Prozessmodelle, da für diese Arbeitsschritte unterschiedliche Implementierungen verwendet werden.

Zur Entwicklung von S-PM Werkzeugen, die aussagekräftige und übertragbare Ergebnisse liefern, wird neben einer eindeutigen Ausführungssemantik eine Referenzimplementierung zur abstrakten Ausführung von subjektorientierten Prozessmodellen benötigt. Während es in der Forschung zum Subjektorientierten Prozessmanagement bereits einige formale Arbeiten zur Spezifikation von Ausführungs- und Verifikationseinheiten gibt, existiert noch keine Implementierung, die einer formal definierten Ausführungssemantik folgt und dabei einen praxistauglichen Sprachumfang anbietet.

Zur abstrakten Ausführung von subjektorientierten Prozessmodellen anhand einer formalen Spezifikation werden folgende Anforderungen festgestellt:

1. Es muss eine Implementierung vorhanden sein, die eine interaktive Validierung der Prozessmodelle ermöglicht. Die Implementierung muss 100% konform zur formalen Ausführungssemantik sein.

2. Es müssen Prozessnetzwerke mit mehreren verteilten Kommunikationspartnern unterstützt werden. Insbesondere müssen komplexe Prozesschoreographien mit sowohl externen als auch internen Subjekten und Multisubjekten unterstützt werden.

3. Subjekte müssen sinnvoll auf externe Unterbrechungen reagieren können, indem sie Nachrichten zu Ausnahmesituationen vorrangig empfangen und behandeln können. Nach einer erfolgreichen Ausnahmebehandlung muss es möglich sein das ursprüngliche Verhalten fortzusetzen.

4. Parallele Abläufe innerhalb der Subjekte müssen unterstützt werden. Dabei muss es möglich sein, dass kritische Abschnitte unterbrechungsfrei ausgeführt werden können.

Die Erfüllung der ersten Anforderung ermöglicht es einerseits Fehler in der Spezifikation auszuschließen und andererseits die Spezifikation mit anderen Implementierungen zu vergleichen. Da der Fokus der Implementierung auf der Konformität zur formalen Ausführungssemantik liegt, muss sie nicht für einen Einsatz in Produktivsystemen geeignet sein. Sie hat daher keine besonderen Anforderungen in Hinblick auf Performance, Schnittstellen zu anderen Systemen oder der Anwenderfreundlichkeit.

Mit der S-BPM Groupware [4], entwickelt an der TU Darmstadt, existiert eine mit Scala implementierte Ausführungseinheit für S-BPM Prozessmodelle. Sie wurde in Stein [22] um eine Verifikationseinheit ergänzt, welche in Link [23] um das Mobility of Channels Konzept erweitert wurde, um auch komplexe Prozesschoreographien zu verifizieren. Die S-BPM Groupware unterstützt Multisubjekte, Unterbrechungen des Subjektverhaltens werden mit dem Observer-Konzept unterstützt. Parallele Abläufe innerhalb eines Subjektes werden mit der Modal Split Aktion umgesetzt, dabei ist es jedoch nicht möglich kritische Abschnitte unterbrechungsfrei auszuführen. Da die S-BPM Groupware nicht auf einer formalen Semantik basiert kann auch die erste Anforderung nicht erfüllt werden. Außerdem basiert die Verifikationseinheit nicht auf der Implementierung der Ausführungseinheit, was potentiell zu unterschiedlichen Ausführungsergebnissen zwischen diesen beiden Komponenten führen kann.

Mit InFlow [5], UeberFlow [6], Actorsphere [7] und Metasonic-Flow [8] gibt es weitere Implementierungen, die eine interaktive Prozessvalidierung ermöglichen. Sie basieren jedoch ebenfalls auf keiner formalen Spezifikation und ermöglichen zudem keine Prozessverifikation.

Mit ePASS-IoS [9, 10] wird PASS in Hinblick auf das „Internet of Services“ mit Multisubjekten und dem Mobility of Channels Konzept erweitert, um komplexe Prozesschoreographien zu unterstützen. Es werden sowohl parallele Abläufe innerhalb der Subjekte als auch externe Unterbrechungen unterstützt. Eine formale Ausführungssemantik wird mit dem π@-Kalkül gegeben, hierzu fehlt jedoch eine Implementierung. Das verwendete π@-Kalkül ermöglicht zwar eine sehr kompakte Notation, im Vergleich mit den pseudocode-ähnlichen Abstrakten Zustandsmaschinen (ASM) ist sie jedoch deutlich schwerer zu lesen und nicht direkt in eine Implementierung übertragbar.

Mit dem ASM-basierten Interpreter aus Börger [11] existiert eine grundlegende formale Arbeit zur Spezifikation einer Ausführungssemantik. Unterbrechungen von Subjekten werden ebenfalls mit einem Observer-Konzept ermöglicht; bei der hier verwendeten Umsetzung besteht jedoch nach einer erfolgreichen Ausnahmebehandlung keine Möglichkeit den unterbrochenen Kontrollfluss fortzuführen. Parallele Abläufe innerhalb eines Subjektes werden mit der AltAction ermöglicht, diese erlaubt jedoch ebenfalls keine unterbrechungsfreie Ausführung von kritischen Abschnitten. Während Prozessnetzwerke mit Multiprozessen unterstützt werden ist keine allgemeine Unterstützung für Multisubjekte vorhanden, eine Umsetzung des Mobility of Channels Konzeptes ist ebenfalls nicht vorhanden.

Eine ausführbare Implementierung von [11] wurde mittels CoreASM in Lerchner and Stary [14] entwickelt. Sie unterstützt jedoch nicht alle Sprachelemente aus Börger [11] - vor allem wurde weder die AltAction noch das Observer-Konzept implementiert und es fehlt auch hier eine Unterstützung von Multisubjekten oder komplexen Prozesschoreographien.

In der Arbeit von Bandmann [12] wurde eine Spezifikation mit ASM entwickelt, die ebenfalls auf [11] basiert, jedoch Elemente zur Unterstützung von Multisubjekten einbringt und zusätzlich eine Verifikationseinheit spezifiziert. Um den Zustandsraum für die Verifikation zu vereinfachen wurde auf die Unterstützung von parallelen Abläufen innerhalb der Subjekte verzichtet. Insbesondere ist es dadurch nicht möglich das Subjektverhalten für eine Ausnahmebehandlung zu unterbrechen; mit dem dort beschriebenen Ansatz ist es nur möglich Ausnahmen nach Abschluss einer Aktion zu behandeln. Konzeptionell ist das Mobility of Channels Konzept vorhanden, eine Choreographie mit externen Prozessen wird dagegen nicht betrachtet. Da es sich ebenfalls um eine rein theoretische Arbeit handelt fehlen einige Elemente, die für eine abstrakte Ausführung benötigt werden. Vor allem gibt es keine Ausarbeitung zur Unterstützung mehrerer Prozessinstanzen oder zum Starten und Beenden von Subjekten.

In Elstermann et al. [13] wird ein Konzept zur Erweiterung des Subjektverhaltens während der Laufzeit skizziert. Dazu wird der Interpreter aus [11] um ein Verfahren erweitert, mit dem zusätzliche SBDs in das Subjektverhalten geladen werden können, welche dynamisch ihre Priorität bestimmen können, um bei einer regelmäßigen Überprüfung festzustellen, welches der geladenen SBDs ausgeführt werden soll. Mit diesem Ansatz ist es denkbar eine Unterbrechung zur Ausnahmebehandlung umzusetzen, die es erlaubt anschließend mit dem ursprünglichen Verhalten fortzufahren. Da eine dynamische Änderung des Subjektverhaltens außerhalb des Rahmens der vorliegenden Arbeit ist wurde dieser Ansatz jedoch nicht weiter verfolgt.

Obwohl jede einzelne der anfangs aufgestellten Anforderungen von mindestens einer der hier vorgestellten Arbeiten abgedeckt wird gibt es keine Arbeit, die insgesamt alle Anforderungen unterstützt. Insbesondere gibt es keine formale Arbeit welche eine interaktive abstrakte Ausführung von Prozessmodellen mit externen Multisubjekten ermöglicht.

Im nächsten Kapitel wird ein Strukturmodell vorgestellt, das sich an den vorhandenen Arbeiten orientiert, und dabei, in Hinblick auf die Erfüllung aller Anforderungen, deren Ansätze zusammenführt und ergänzt.

4 S-PM Strukturmodell

Im zweiten Kapitel wurden Grundlagen zum Subjektorientierten Prozessmanagement (S-PM) behandelt. Dabei wurden einige wichtige Sprachelemente und Aktionen bereits kurz vorgestellt.

Im vorherigem Kapitel wurde festgestellt, dass es noch keine formale Spezifikation aller Sprachelemente gibt, die benötigt werden um die dort aufgestellten Anforderungen an eine Ausführungseinheit für S-PM Prozessmodelle zu erfüllen.

In diesem Kapitel wird daher ein Strukturmodell erarbeitet, mit dem es möglich wird diese Anforderungen zu erfüllen. Das Strukturmodell erweitert die graphenbasierte Modellierungsnotation des Parallel Activity Specification Scheme (PASS) [17, 18]. Es kombiniert bereits bestehende Sprachelemente aus den im letzten Kapitel vorgestellten Arbeiten und erweitert diese, wie beispielsweise die CallMacro Aktion mit der Hinzunahme von Argumenten, die der Makroinstanz übergeben werden. Zudem werden mit den Knotenprioritäten und der Cancel Aktion auch neue Konzepte eingebracht.

4.1 Begriffsdefinitionen

4.1.1 Prozessmodell

In einem *Prozessmodell* werden dessen Subjekte zusammengefasst. Zur Vereinfachung wird im Folgenden zur Vereinfachung der Begriff „Prozess“ verwendet. In der graphischen Modellierung wird ein Prozess durch ein *Subject Interaction Diagram (SID)* definiert. Im SID sind alle Subjekte des Prozesses mit deren Kommunikationsmöglichkeiten sichtbar. Die Subjekte werden über Kanten verbunden, an den Kantenbeschriftungen wird angegeben welche Nachrichtentypen zwischen den jeweiligen Subjekten ausgetauscht werden können.

Ein Prozess ist eindeutig über eine *ProzessID* identifizierbar. Ein Prozess kann mehrfach gestartet werden und es können mehrere Instanzen eines Prozesses gleichzeitig ausgeführt werden. Die Prozessinstanzen sind über eine *ProzessInstanzID* eindeutig identifizierbar.

Prozesse können durch *Interfacesubjekte* verbunden werden. Ein Interfacesubjekt besteht nur aus einer Referenz auf ein internes Subjekt eines anderen Prozesses und hat kein eigenes Verhalten. Das referenzierte Subjekt wird als *externe Subjekt* bezeichnet. Im SID werden Interfacesubjekte durch einen gestrichelten Rahmen dargestellt. *Serviceprozesse* können mit beliebigen anderen Prozessen verbunden werden, indem sie ein Interfacesubjekt mit einer offenen externen Referenz haben.

Damit ein Prozess gestartet werden kann muss es mindestens ein Interfacesubjekt oder mindestens ein *Startsubjekt* beinhalten. Startsubjekte werden durch einen dicken Rahmen dargestellt. Beim Start eines Prozesses werden alle Startsubjekte gestartet. Prozesse mit Interfacesubjekten können von externen Prozessen aus gestartet werden, indem eine Nachricht von dem externen Subjekt an das Interfacesubjekt gesendet wird.

4.1.2 Subjekt und Agent

Mit einem *Subjekt* wird ein Teilablauf eines Prozesses beschrieben, das Subjektverhalten wird durch Makros definiert. Ein Subjekt ist über eine *SubjektID* in Verbindung mit der *ProzessID* eindeutig identifizierbar.

Das Verhalten der Subjekte wird durch *Agenten* ausgeführt. Ein Agent ist mit einer *AgentenID* eindeutig identifizierbar und kann das Verhalten mehrerer Subjekte ausführen. Das Verhalten von einem Subjekt kann gleichzeitig, als unterschiedliche Instanzen mit einem äquivalentem Verhalten, von unterschiedlichen Agenten ausgeführt werden; in dem Fall nennt man es *Multisubjekt*. Wenn ein Subjekt kein Multisubjekt sein darf kann dies durch den Begriff *Single Subject* verdeutlicht werden.

Die Zuordnung eines Agenten zu einem Subjekt in einer Instanz eines Prozesses ist mit einem *Channel* eindeutig identifizierbar und wird mit AgentenID@SubjektID@ProzessInstanzID@ProzessID dargestellt. Diese Zuordnung wird auch als *Verknüpfung* oder *Kommunikationskanal* bezeichnet. Channel können in Variablen gespeichert und als Nachrichteninhalt weitergegeben werden.

Ein Agent beginnt mit der Ausführung eines Subjektverhaltens sobald er eine Nachricht zu dem Subjekt empfängt, oder falls der Agent beim Start einer Prozessinstanz für ein Startsubjekt ausgewählt wurde. Wenn einem Agenten eine Nachricht zu einem Subjekt zugestellt wird, dessen Verhalten er bereits abgeschlossen hat, wird die Ausführung des Subjektverhaltens erneut begonnen.

4.1.3 Makro

Das Verhalten von Subjekten wird durch Makros definiert; mit ihnen wird das Subjektverhalten in kleinere wiederverwendbare Teile strukturiert. Ein Makro ist durch eine *MakroID* in Verbindung mit der SubjektID und ProzessID eindeutig identifizierbar. Ein Subjekt muss mindestens ein Makro haben: das Startmakro, im folgenden auch *Mainmakro* genannt. Es kann endlich viele weitere Makros haben.

Das Verhalten eines Makros wird durch einen gerichteten Graphen, dem *Subject Behaviour Diagram (SBD)*, definiert. Dieser besteht aus einer endlichen Menge von *Aktionsknoten*, die durch ausgehende Kanten mit anderen Aktionsknoten verbunden sind. Der erste auszuführende Aktionsknoten wird als *Startknoten* bezeichnet und wird durch einen dickeren Rahmen dargestellt. Ein *Endknoten* hat keine ausgehenden Kanten und führt immer die End-Aktion aus, welche das Verhalten des Makros beendet.

Alle Aktionsknoten müssen von dem Startknoten aus erreichbar sein; es muss genau einen Startknoten und mindestens einen Endknoten geben.

Zum Ausführen eines Makros wird eine *Makroinstanz* angelegt, dabei wird automatisch der Startknoten aktiviert. In der Makroinstanz werden die dort aktiven Aktionsknoten vermerkt; die Ausführung des Verhaltens der Makroinstanz besteht aus der Ausführung des Verhaltens dieser aktiven Knoten. Beim Übergang von einem Aktionsknoten zum Nächsten wird der bisherige Aktionsknoten deaktiviert und der Folgeknoten aktiviert. Um parallele Abläufe abzubilden können auch mehrere Aktionsknoten gleichzeitig aktiviert. Von einem Makro können gleichzeitig mehrere Instanzen aktiv sein.

Zum Start des Subjektverhaltens wird eine Instanz vom Mainmakro angelegt; das Subjektverhalten ist abgeschlossen wenn diese Instanz beendet wird.

4.1.4 Aktionsknoten

Aktionsknoten sind mit einer Definition der Aktion beschriftet, die Kanten sind mit Bedingungen zum Verlassen der Aktion beschriftet und zeigen auf die Folgeaktion. Die Aktion selbst erstreckt sich über eine *Knotenaktion* und eine *Kantenaktion*. Eine Erläuterung der Aktionen befindet sich im folgenden Abschnitt 4.2.

Knotenprioritäten

Sind in einer Makroinstanz mehrere Aktionsknoten aktiv kann über *Knotenprioritäten* gesteuert werden, dass einzelne Aktionen bevorzugt ausgeführt bzw. andere pausiert werden. Könnten in einer Makroinstanz mehrere Knoten ausgeführt werden, so wird dies nun beschränkt auf Knoten der höchsten Priorität. Die Knotenpriorität ist standardmäßig für jeden Aktionsknoten „0“. Eine Änderung der Knotenpriorität zur Laufzeit ist nicht möglich.

Die jeweilige Aktion des Knotens kann zur Laufzeit erlauben, dass auch andere Knoten mit einer niedrigeren Priorität ausgeführt werden dürfen. Dies wird von Aktionen verwendet die auf ein bestimmtes Ereignis warten, um erst nach dessen Eintreffen vorrangig ausgeführt zu werden.

Kanteneigenschaften

Normale Kanten sind ausgehende Kanten eines Knotens, welche durch Kantenbeschriftungen Bedingungen zum erfolgreichen Verlassen der Aktion beschreiben. Normale Kanten werden als durchgezogene schwarze Linie dargestellt.

Abhängig von der Art der Aktion findet ein ein- oder zweistufiges Verfahren zur Kantenwahl statt. Während manche Aktionen nur eine einzige normale ausgehende Kante erlauben, und automatisch von der Knotenaktion zur Kantenaktion übergehen, gibt es auch Aktionen die zuerst die Bedingungen der Kanten auswerten, um diese *freizuschalten*, und dann erst, nach einer manuellen *Aktivierung* durch den Agenten, zu der Kantenaktion übergehen.

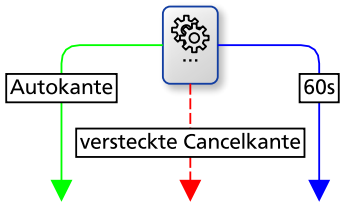


Abbildung 4.1: spezielle Kanteneigenschaften

Sind mehrere Kanten freigeschaltet muss der Agent immer entscheiden welche er aktivieren möchte. Die Auswahl der Kanten, die aktiviert werden können, kann durch *Kantenprioritäten* eingeschränkt werden. Es stehen immer nur die Kanten mit der höchsten freigeschalteten Priorität zur Verfügung. Standardmäßig ist die Kantenpriorität „0“.

Um eine Aktion, die eigentlich nur durch eine manuelle Aktivierung des Agenten abgeschlossen wird, dennoch automatisch durchzuführen, können normale Kanten zu *Autokanten* spezialisiert werden. Diese werden durch eine grüne Linie dargestellt. Eine Autokante wird automatisch aktiviert, wenn sie freigeschaltet wird und es neben ihr keine weitere freigeschaltete Kante mit gleicher oder höherer Priorität gibt.

Um die Ausführung einer Aktion abzubrechen, ohne sie erfolgreich zu beenden, gibt es die Möglichkeit jeweils eine *Cancelkante* für einen manuellen Abbruch und eine *Timeoutkante* für einen automatischen Abbruch zu ergänzen. Cancelkanten werden mit einer roten Linie dargestellt und sind immer freigeschaltet, werden aber für die Bedingungen der Autokanten und der Kantenprioritäten nicht betrachtet. Timeoutkanten werden durch eine blaue Linie dargestellt und müssen als Kantenbeschriftung eine Zeitangabe haben, nach der sie automatisch freigeschaltet und aktiviert werden.

Versteckte Kanten werden gestrichelt dargestellt. Diese sind für den Agenten, selbst wenn sie freigeschaltet sind, nicht sichtbar und können vom Agenten auch nicht aktiviert werden. Die Aktivierung einer versteckten Cancelkante kann jedoch durch die Cancel-Aktion erfolgen.

4.1.5 Nachricht

Nachrichten werden zwischen Agenten verschickt. Dabei können Nachrichten sowohl Nachrichten im klassischen Sinne, wie ein Brief oder eine E-Mail, sein als auch beliebige abstrakte Informationen, Gegenstände oder Produkte repräsentieren. Nachrichten werden vom Sender in den *Inputpool* des Agenten des Empfängersubjekts abgelegt. Eine Nachricht beinhaltet immer den Channel des Absenders, den *Nachrichtentyp* sowie einen *Nachrichteninhalt*. Optional kann einer Nachricht eine *CorrelationID* vergeben werden. Der Nachrichtentyp kann frei gewählt werden und sollte den Inhalt der Nachricht abstrakt beschreiben.

Der Nachrichteninhalt selbst hat einen Datentyp, der unabhängig vom gewähltem Nachrichtentyp ist und nur vom tatsächlichem Inhalt abgeleitet wird. Es gibt vier Datentypen:

- Text: ein einfacher Text, der durch einen Agenten eingegeben wird.
- MessageSet: eine Menge von Nachrichten. Hiermit können Nachrichten *verschachtelt* werden, um mit einer Nachricht mehrere andere empfangene Nachrichten mit einer einzigen Nachricht weiterzuleiten.
- ChannelInformation: eine Menge von Channeln.
- CorrelationID: eine einzelne CorrelationID.

Beim Empfangen können Nachrichten in einer Variablen gespeichert werden, um mit der VarMan Aktion den Absenderchannel, die CorrelationID oder den Nachrichteninhalt zu entpacken.

4.1.6 Inputpool

Der Nachrichtenaustausch ist asynchron gestaltet. Dazu hat jeder Agent zu jedem Subjekt, dessen Verhalten er ausführt, einen Inputpool, in den Nachrichten für dieses Subjekt abgelegt werden. Der Empfänger kann die Nachrichten nun zu einem späteren Zeitpunkt abrufen.

Für jedes Tupel aus (Absender-Subjekt, Nachrichtentyp, CorrelationID) wird im Inputpool eine eigene *Warteschlange* verwendet, welche als FIFO Listen umgesetzt sind. Damit wird beim Empfangen immer die älteste Nachricht abgerufen.

Um den Zustandsraum für die Verifikation zu begrenzen lässt sich die Anzahl der Nachrichten je (Absender-Subjekt, Nachrichtentyp)-Paar durch einen Inputpool-Limit begrenzen[1]. Außerdem lässt sich durch diese Limitierung ein Sendersubjekt ausbremsen, um einen Informationsstau oder Datenverlust zu vermeiden [30].

4.1.7 Variablen

Wie in Programmiersprachen üblich können mit *Variablen* Daten gehalten werden. Damit können beispielsweise Nachrichten, CorrelationIDs oder Channel für die spätere Verwendung gespeichert werden. Variablen haben einen Namen, einen Datentyp und einen Inhalt. Die möglichen Datentypen wurden im Abschnitt 4.1.5 beschrieben. Jeder Agent hat für jedes Subjekt eigenen Speicher; andere Subjekte/Agenten können darauf nicht zugreifen.

Der Sichtbarkeitsbereich einer Variablen ist standardmäßig global; sie können von jeder Aktion in jeder beliebigen Makroinstanz gelesen und geschrieben werden. Der Sichtbarkeitsbereich einer Variablen kann optional auf eine Makroinstanz beschränkt werden, in dem Fall werden sie makroinstanz-lokale Variablen genannt. Die Beschränkung der Sichtbarkeit ermöglicht eine seiteneffekt-freie Verwendung von Variablen in Makros, von denen gleichzeitig mehrere Instanzen ausgeführt werden. Sollte es eine globale Variable mit dem gleichen Namen einer makroinstanz-lokalen Variablen geben ist die globale Variable durch diese verdeckt.

[1] Da beliebig viele CorrelationIDs erzeugt bzw. verwendet werden können ist somit auch die Anzahl der Warteschlangen potentiell unendlich. Eine Limitierung der Nachrichten je Warteschlange wäre somit nicht ausreichend, da damit die Anzahl der Nachrichten insgesamt nicht begrenzt werden kann. Deshalb muss die Limitierung unabhängig von der CorrelationID sein.

4.1.8 CorrelationIDs

Um eine kausale Zuordnung von Nachrichten, insbesondere von Anfragen zu Antworten, zu ermöglichen werden CorrelationIDs verwendet. Mit diesen CorrelationIDs kann festgelegt werden auf welche vorhergehende Nachricht sich eine spätere Nachricht bezieht. Beim Senden einer Nachricht kann eine neue CorrelationID erzeugt werden, die als Teil der Nachricht mitgesendet wird. Der Empfänger verwendet beim Senden einer Antwort diese CorrelationID als Parameter für die entsprechende Warteschlange des Inputpools beim ursprünglichen Sender.

Bei der Erzeugung von CorrelationIDs ist darauf zu achten, dass diese mindestens prozessweit eindeutig sind. In der Praxis könnten dazu UUIDs verwendet werden; zur abstrakten Ausführung ist es jedoch ausreichend, wie in dieser Arbeit umgesetzt, fortlaufende Zahlen zu verwenden. Wenn eine solche Zuordnung zwischen Nachrichten nicht explizit erfasst werden muss wird standardmäßig die CorrelationID 0 verwendet.

4.2 Aktionen

Aktionsknoten werden mit einer Kombination aus einem Symbol und einer Knotenbeschriftung dargestellt.

Allgemein werden Aktionsknoten mit einem Skript-Symbol dargestellt, der Aktionstyp wird dabei durch die Knotenbeschriftung angegeben. Über anschließende runde Klammern können der Aktion Parameter übergeben werden.

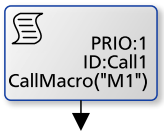


Abbildung 4.2

Für die *Internal Action* werden zwei Zahnräder als Symbol verwendet. Die Knotenbeschriftung kann nun als Freitext die von dem Agenten durchzuführende Aktion beschreiben. Die *Send* Aktion verwendet einen schwarzen Brief , für die *Receive* Aktion wird ein weißer Brief verwendet. Die Knotenbeschriftung ist bei beiden Aktionen optional, es kann ein beliebiger Freitext angegeben werden. Optional kann jeder Aktion in der Knotenbeschriftung ein eindeutiger Identifikator und eine Knotenpriorität vergeben werden. Dazu wird in der Knotenbeschriftung eine Zeile mit dem Präfix „ID:“ bzw. „PRIO:“ begonnen.

In Abbildung 4.2 ist ein Aktionsknoten der Aktion „CallMacro“ zu sehen, dem die Knotenpriorität „1“ und der Identifikator „Call1“ zugewiesen wird. Der Aktion wird der Parameter „M1“ übergeben.

Die *End* Aktion wird nur über einen geschlossenen Kreis dargestellt, die Knotenbeschriftung ist dabei optional und kann verwendet werden, um anzugeben in welchem Zustand das Makro beendet wurde. Sie ist die einzige Aktion ohne ausgehende Kanten.

4.2.1 Internal Action und Tau

Die ***Internal Action*** dient zum Erfassen von subjekt-internen Aktionen, die keine Auswirkungen auf andere Subjekte oder andere Aktionen haben. Durch mehrere ausgehende Kanten können Entscheidungen modelliert werden, dabei sind alle ausgehenden Kanten ohne weitere Bedingungen freigeschaltet und können durch den Agenten aktiviert werden. In der Knotenbeschriftung wird die Aktion, beziehungsweise die Entscheidung, als natürlicher Text formuliert. An den Kantenbeschriftungen werden die möglichen Ergebnisse der Aktion ebenfalls als natürlicher Text formuliert.

Die ***Tau*** Aktion dient als Platzhalter einer Internal Action, falls das gewünschte Verhalten während der Modellierung entweder nicht bekannt ist oder zur Vereinfachung nicht näher beschrieben werden muss. Als Hilfsmittel zur Prozessvalidierung können die ausgehenden Kanten als Autokanten definiert werden, in dem Fall wird automatisch eine zufällige Kante aktiviert.

4.2.2 Modal Split & Modal Join

Mit Hilfe der ***Modal Split*** und ***Modal Join*** Aktionen werden parallele Abläufe modelliert, die gleichzeitig und gleichberechtigt laufen. Die Modal Split Aktion aktiviert alle ihre Folgeknoten und startet damit parallele Ausführungspfade. Zu jeder Modal Split Aktion muss eine Modal Join Aktion existieren, in der die Ausführungspfade münden. Die Modal Join Aktion kann erst verlassen werden wenn die Ausführung aller Pfade diese Aktion erreicht haben, und die Pfade können sich erst in einer Modal Join Aktion vereinigen und nicht bereits vorher. Pfade können alternativ statt in einer Modal Join Aktion auch in einen Endzustand münden, um damit das Verhalten vorzeitig zu beenden. Sollten alle Pfade in einem Endzustand münden kann auf die Modal Join Aktion verzichtet werden. Es ist jedoch nicht möglich dass ein Pfad zu einer Aktion führt die ohne ein Modal Join zum vorherigem Modal Split führt, da eine solche Schleife potentiell unendlich viele Knoten aktivieren würde.

4.2.3 Cancel

Werden mehrere Pfade parallel ausgeführt so ist es durch die Verwendung von Knotenprioritäten möglich andere Pfade zu pausieren. Pausierte Knoten werden jedoch weiter ausgeführt sobald alle anderen Knoten einer höheren Priorität abgearbeitet wurden. Durch die ***Cancel*** Aktion wird es möglich den Kontrollfluss anderer Pfade direkt zu beeinflussen, indem die Cancel-Kante eines anderen Knotens aktiviert wird um damit seinen Abbruch zu erzwingen.

Die Cancel Aktion ist eine blockierende Aktion, sie kann nur ausgeführt werden wenn der andere Knoten aktiv ist. Es ist möglich der Cancel Aktion mehrere ausgehende Kanten zu geben, damit ist es möglich zu wählen welcher Knoten abgebrochen werden soll. Die Kantenbeschriftungen geben jeweils die ID des abzubrechenden Knotens an.

Wenn die Cancel-Kante eines anderen Knotens versteckt ist kann dieser ausschließlich mit einer Cancel Aktion abgebrochen werden, da der Agent keine Auswahl zur Aktivierung von versteckten Kanten hat.

Es können nur Aktionen in der gleichen Makroinstanz abgebrochen werden.

4.2.4 CallMacro

Mit der ***CallMacro*** Aktion können Makros aufgerufen werden, indem dazu von dem Makro eine neue Instanz angelegt und auf deren Terminierung gewartet wird. Beim Abbruch der CallMacro Aktion werden alle aktiven Knoten der Instanz abgebrochen und die Makroinstanz terminiert.

Makros können optional Variablennamen als Parameter haben, diese stehen innerhalb des Makros als makroinstanz-lokale Variablen zur Verfügung. Beim Aufruf eines Makros mit Parametern müssen der CallMacro Aktion die Quellvariablen als Aktionsparameter angegeben werden; deren Inhalte werden beim Start in die makroinstanz-lokalen Variablen der angelegten Makroinstanz kopiert.

Die Endzustände von Makros können beschriftet werden um damit den weiteren Kontrollfluss der aufrufenden CallMacro Aktion zu steuern. Dabei müssen die ausgehenden Kanten der CallMacro Aktion die gleichen Beschriftungen haben wie die End Aktionen.

In der Abbildung 4.3 ist ein Makro „M1“ mit dessen Aufruf zu sehen. Die Variablen „x“ und „y“ wurden als Parameter des Makros angegeben. Beim Aufruf durch die CallMacro Aktion werden die Inhalte der Variablen „r“ und „s“ in makroinstanz-lokale Variablen der erzeugten Instanz kopiert. Die Variablen „r“ und „s“ können nun, in einem zu der CallMacro Aktion parallelen Ausführungspfad, verändert werden ohne „x“ und „y“ zu beeinflussen. Eine parallele CallMacro Aktion könnte so zwischenzeitlich eine weitere Instanz des Makros M1 anlegen ohne „x“ und „y“ der ersten Instanz zu beeinflussen.

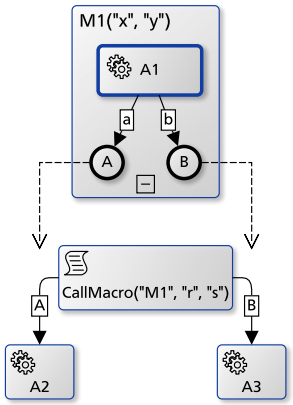


Abbildung 4.3: CallMacro Aktion

Wird in der Aktion A1 die Kante „a“ aktiviert so terminiert die Makroinstanz des Makros M1 mit dem Resultat „A“, die CallMacro Aktion wird somit über die mit „A“ beschriftete Kante verlassen und im Anschluss wird die Aktion A2. Wird in der Aktion A2 hingegen die Kante „b“ gewählt so wird im Anschluss die Aktion A3 aktiviert.

4.2.5 VarMan

Mit der ***VarMan*** Aktion wird eine *Var*iablen*Man*ipulation durchgeführt. Falls nicht anders angegeben wird die Zielvariable überschrieben und erhält den Datentyp der Quellvariablen. Bei Operatoren mit mehreren Quellvariablen müssen diese den gleichen Datentyp haben, beispielsweise ist es nicht möglich eine Vereinigungsmenge aus einer Menge von Nachrichten und einer Menge von Channeln zu bilden. Es stehen zehn Operatoren zur Verfügung:

- `assign (A, X)`: kopiert den Inhalt der Variablen A in die Variable X. Mit dieser Operation kann insbesondere der Inhalt einer makroinstanz-lokalen Variable in eine globale Variable kopiert werden.
- `storeData (A, X)`: speichert den Wert X in der Variablen A. Hiermit lässt sich beispielsweise ein Nachrichteninhalt vorbelegen.
- `clear (X)`: entfernt die Variable X und setzt damit auch ihren Datentyp zurück.
- `concatenation (A, B, X)`: bildet die Vereinigungsmenge der Inhalte von A und B und speichert sie in X. Darstellung: $X := A \cup B$
- `intersection (A, B, X)`: bildet die Schnittmenge der Inhalte von A und B und speichert sie in X. Darstellung: $X := A \cap B$

- `difference (A, B, X)`: bildet die Differenz der Inhalte von A und B und speichert sie in X. Darstellung: $X := A \setminus B$
- `selection (A, X, min, max)`: führt eine manuelle Auswahl einer beliebigen Teilmenge von A aus und speichert sie in X. Die Mächtigkeit der Teilmenge kann mit den optionalen Parametern min und max eingeschränkt werden.
- `extractContent (A, X)`: vereinigt alle Nachrichteninhalte aus A und speichert sie in X. Dazu muss A eine Variable mit dem Datentyp „MessageSet“ sein, und alle in A gespeicherten Nachrichten müssen einen Nachrichteninhalt mit dem gleichen Datentyp haben. Hiermit lassen sich verschachtelte Nachrichten entpacken.
- `extractChannel (A, X)`: vereinigt alle Absenderchannel der Nachrichten aus A und speichert sie in X. Dazu muss A eine Variable mit dem Datentyp „MessageSet“ sein.
- `extractCorrelationID (A, X)`: speichert die CorrelationID der Nachricht aus A in X. Die Variable A muss den Datentyp „MessageSet“ haben und muss genau eine Nachricht beinhalten.

4.2.6 Select Agents

Mit der ***Select Agents*** Aktion können Agenten zu einem Subjekt gewählt und in einer Variablen gespeichert werden. Diese Variable kann bei einem späterem Senden verwendet werden, um die Empfänger der Nachricht zu definieren. Somit muss die Auswahl der Empfänger nicht bei jedem Senden erneut getroffen werden.

Des Weiteren können mit der Select Agents Aktion Agenten gewählt werden, deren Channel im Sinne des Mobility of Channels Konzepts weiterverschickt werden, um in verteilten Kommunikationsabläufen eine eindeutige Zuordnung zu ermöglichen.

4.2.7 Inputpool

Ein Subjekt kann mittels der ***CloseIP*** Aktion eine Warteschlange seines Inputpools schließen, um zu verhindern, dass in diese weitere Nachrichten abgelegt werden. Mit der ***OpenIP*** Aktion kann eine Warteschlange wieder geöffnet werden. Mit den ***CloseAllIPs*** und ***OpenAllIPs*** Aktionen kann der komplette Inputpool inklusive aller Warteschlangen geschlossen bzw. wieder geöffnet werden. Standardmäßig ist der Inputpool mit allen Warteschlangen geöffnet.

Ein Subjekt kann mittels der ***IsIPEmpty*** Aktion prüfen, ob sich Nachrichten oder Reservierungen in einer Warteschlange befinden, um dadurch über den weiteren Kontrollfluss zu entscheiden. Durch das Schließen des Inputpools und einer anschließenden Prüfung, ob alle Warteschlangen leer sind, kann ein Subjekt sicherstellen, dass es alle Nachrichten abgearbeitet hat. Für die Verifikation auf Interaction Soundness wird bei der Terminierung eines Subjektes überprüft ob alle Warteschlangen leer sind, in dem Fall wird das Subjekt als *proper terminated* bezeichnet – ist dies nicht der Fall gab es einen Interaktionsfehler.

4.2.8 Senden und Empfangen

Der Nachrichtenaustausch findet asynchron mittels den ***Send*** und ***Receive*** Aktionen statt. Die Send Aktion legt Nachrichten im Inputpool der Empfänger ab, die Receive Aktion prüft ob im eigenem Inputpool Nachrichten vorhanden sind und entfernt diese beim Empfangen daraus.

Senden

Die Send Aktion kann nur eine normale ausgehende Kante haben, an der angegeben wird, zu welchem Subjekt die Nachricht gesendet werden soll und welchen Nachrichtentyp die Nachricht hat. Zum Senden an ein Multisubjekt kann optional die Anzahl der Empfänger angegeben werden, sie ist standardmäßig 1. Die Channel der Empfänger und der Nachrichteninhalt können über Variablen vorgegeben werden, ansonsten werden sie durch den Agenten eingegeben.

Beim Senden können CorrelationIDs auf zwei sich ergänzende Weisen verwendet werden:

1. Es kann optional eine neue CorrelationID erzeugt werden. Diese wird als Teil der Nachricht mitgesendet und in einer Variablen gespeichert.
2. Es kann optional eine Variable mit einer CorrelationID angegeben werden, die verwendet wird, um beim Empfänger im Inputpool die Warteschlange ebendieser CorrelationID zu verwenden.

In beiden Fällen wird die CorrelationID „0“ verwendet, wenn keine Variable angegeben wurde.

Das Senden ist in zwei Phasen aufgeteilt: Zuerst werden bei allen Empfängern Reservierungen abgelegt, die dann in der zweiten Phase durch die eigentliche Nachricht ausgetauscht werden. Somit kann sichergestellt werden, dass beim Senden an Multisubjekte die Nachricht allen Empfängern zugestellt werden kann. Beim Abbruch der Send Aktion werden die Reservierungen entfernt, um den reservierten Platz in den Inputpools der Empfänger wieder freizugeben. Eine Reservierung kann nur dann bei einem Empfänger in der Warteschlange abgelegt werden, wenn diese geöffnet ist, dort noch Platz frei ist und der Empfänger nicht *non-proper terminated* ist.

Empfangen

Der Receive Aktion können beliebig viele ausgehende Kanten angegeben werden, die jeweils, analog zum Senden, Parameter für den Nachrichtentyp, das Absendersubjekt und ebenfalls optional für Multisubjekte die Anzahl der Sender haben. Optional kann eine Variable mit einer CorrelationID gegeben werden, so dass nur Nachrichten empfangen werden können die sich auf diese CorrelationID beziehen. Gibt es keine Angabe zur CorrelationID wird standardmäßig die Warteschlange der CorrelationID 0 genommen.

Die empfangenen Nachrichten können in einer Variablen gespeichert werden, um beispielsweise die CorrelationIDs und die Absenderchannel bei einer darauffolgenden Send Aktion zu verwenden, oder um die Nachrichten an ein drittes Subjekt weiterzuleiten.

Beispiel: Konversationen

In Abbildung 4.4 wird der Ablauf zweier unabhängiger Konversationen gezeigt. Dazu wird ein Prozess „Konversation“ angenommen, in dem es die Subjekte A und B gibt. Das Subjekt A ist das Startsubjekt und hat neben dem Mainmakro ein weiteres Makro, in dem eine „Anfrage“-Nachricht gesendet und eine „Wert“-Nachricht empfangen wird. Für die Zuordnung der Werte zu den Anfragen werden CorrelationIDs verwendet, die beim Senden erzeugt und in einer makroinstanz-lokalen Variablen gespeichert werden.

Das Subjekt B hat im Mainmakro eine Schleife zum Empfangen von „Anfrage“-Nachrichten, welche beim Empfangen in einer Variablen zwischengespeichert werden und mit einer „Wert“-Nachricht beantwortet werden. Beim Senden der Antwort wird der Absenderchannel und die CorrelationID aus der gespeicherten Nachricht verwendet.

Zur Vereinfachung der Darstellung wird darauf verzichtet beim Nachrichtenaustausch Reservierungen zu verwenden, die Datentypen werden nicht dargestellt und es werden nur die Variablen von Alice, nicht jedoch von Bob, gezeigt.

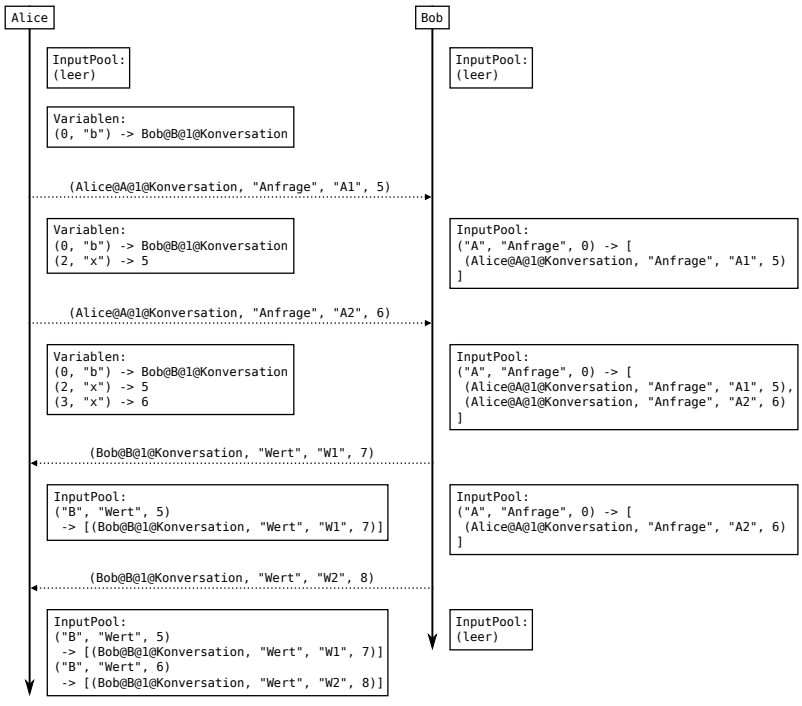


Abbildung 4.4: Konversation

Zu Beginn des Ablaufs wählt Alice den Agenten „Bob“ für das Subjekt B aus und speichert sich die Referenz in der Variablen „b“. Sie ruft nun das Makro zum ersten Mal auf, dabei wird die Makroinstanznummer 2 vergeben. Sie sendet eine Anfrage mit dem Inhalt „A1“ an Bob, dabei wird die CorrelationID 5 erzeugt, als Teil der Nachricht mitgesendet und in der makroinstanz-lokalen Variablen „x“ gespeichert.

Im nächsten Schritt ruft Alice das Makro erneut auf, während in der ersten Makroinstanz noch auf eine Antwort gewartet wird. Diese zweite Instanz bekommt die Makroinstanznummer 3. Alice versendet nun eine zweite Anfrage, diesmal mit dem Inhalt „A2“; dabei wird die CorrelationID 6 erzeugt und ebenfalls in der makroinstanz-lokalen Variablen „x“ gespeichert.

Im Inputpool von Bob befindet sich nun eine Warteschlange mit den beiden Nachrichten von Alice. Er empfängt die älteste Nachricht, mit dem Inhalt „A1“ und der CorrelationID 5, und beantwortet sie. Dabei gibt er beim Senden an, dass die Nachricht mit dem Wert „W1“ bei Alice in der Warteschlange zur CorrelationID 5 gespeichert werden soll, und vergibt eine eigene CorrelationID 7. Analog dazu empfängt Bob auch die zweite Anfrage; seine Antwort mit dem Inhalt „W2“ wird bei Alice in der Warteschlange zur CorrelationID 6 gespeichert.

Im Inputpool von Alice befinden sich nun zwei Warteschlangen mit jeweils einer „Wert“-Nachricht von Bob. Alice kann nun anhand der Variablen „x“, die in den Instanzen des Makros unterschiedliche Werte haben, die jeweils zugehörige Nachricht empfangen.

4.2.9 End

Mittels der ***End*** Aktion - die immer ein Endknoten ist - wird die Ausführung des aktuellen Makros beendet. Wird dabei das Mainmakro beendet so terminiert das Subjekt. Der End Aktion kann eine optionale Knotenbeschriftung gegeben werden, beim Aufruf eines Makros kann damit die CallMAcro Aktion feststellen mit welchem Zustand ein Makro terminiert wurde.

Zum Terminieren der Makroinstanz bricht die End Aktion alle sonstigen aktiven Knoten ab. Dies verhält sich ähnlich zur Cancel Aktion, jedoch wird der Abbruch direkt ausgeführt und nicht die Cancel Kante aktiviert. Somit können auch Knoten terminiert werden die keine Cancel Kante haben, es wird kein Folgeknoten aktiviert. Die End Aktion wartet auf den Abschluss aller Abbrüche. Somit ist die Terminierung einer Prozessinstanz erst abgeschlossen wenn keine weiteren aktiven Knoten vorhanden sind.

Durch die saubere Terminierung aller aktiven Knoten ist es möglich, dass beim Modal Split nicht alle Pfade im Modal Join enden müssen, sondern alternativ auch in einem Endzustand enden können.

Bei der Terminierung eines Subjektes wird überprüft ob sich noch Nachrichten in einem Inputpool befinden; dies deutet auf einen Modellierungsfehler hin. Nur wenn das nicht der Fall ist so ist das Subjekt ***proper terminated***. Beim Empfangen weiterer Nachrichten wird das Subjekt neu gestartet. Wenn ein Subjekt nicht proper terminated ist so kann es nicht neu gestartet werden, und ein Senden an dieses Subjekt blockiert, um diesen Fehler zu verdeutlichen.

4.3 Zusammenfassung und Vergleich

In diesem Kapitel wurde im ersten Teil eine Struktur zur subjektorientierten Modellierung von Prozessen vorgestellt. Im zweiten Teil wurden Aktionen definiert.

Die Begriffsdefinitionen von Prozessen, Subjekten und Agenten orientieren sich in dieser Arbeit an Bandmann [12], da in Fleischmann et al. [1] und Börger [11] anstelle von Multisubjekten das Konzept von Multiprozessen verfolgt wird. Daraus ergibt sich, dass sich die Definitionen von Nachrichten, dem InputPool und dem Senden und Empfangen ebenfalls an [12] orientieren. Die Definition des Channels wurde zur Unterstützung mehrerer Prozessinstanzen um die beiden Felder ProzessID und ProzessInstanzID erweitert. Insbesondere wurde das Konzept der CorrelationIDs übernommen, die Limitierung des Inputpools wurde jedoch so geändert, dass sie unabhängig von der CorrelationID ist, was eine potentielle Unendlichkeit der Nachrichtenmenge vermeidet. Die Select Agents Aktion vereinigt die *Create New Subjects* Aktion aus Stein [22] mit der *Choose Agents* Aktion aus Link [23]. *Serviceprozesse* wurden aus [1, Sec. 5.6.3] übernommen.

Die Definition von Makros und der CallMacro Aktion orientiert sich an [4, 9, 22, 23], da Makros in [12] nicht betrachtet werden. Dabei ist das hier vorgestellte Modell erweitert um ein neues Konzept der makroinstanz-lokalen Variablen, sowie um Parameter der Makroinstanz, um die gleichzeitige Ausführung mehrerer Instanzen des gleichen Makros besser zu unterstützen. Die Möglichkeit, in einem Makro mehrere unterschiedliche Endzustände zu verwenden, um die entsprechende ausgehende Kante der Call Makro Aktion zu bestimmen, wurde aus [9, 11] übernommen.

Automatische Kanten, Kantenprioritäten, Timeout- und Cancel-Kanten wurden ebenfalls aus [12] übernommen. Zur Differenzierung von der Cancel-Kante muss eine Timeoutkante jedoch über eine Zeitangabe verfügen und wird beim Überschreiten der Zeit immer automatisch aktiviert. Versteckte Kanten wurden neu hinzugefügt, um optional zu ermöglichen, dass eine automatische Kante oder eine Cancel-Kanten ausschließlich vom System aktiviert werden darf.

Die VarMan Aktion wird bereits in [4, 12, 22, 23] verwendet und in dieser Arbeit um weitere Operatoren ergänzt.

Das in dieser Arbeit verwendete Konzept der Modal Split und Modal Join Aktionen orientiert sich an [4, 22, 23], da die in [12] getroffene Sequentialisierung der Aktionen keine gleichberechtigte Ausführung von zwei Aktionen ermöglicht. Der hier verwendete Ansatz ist somit vergleichbar mit der *AltAction* aus [11], jedoch werden hier die aktiven Knoten auf Ebene der Makroinstanz verwaltet, um durch die neu eingeführten Knotenprioritäten eine unterbrechungsfreie Ausführung von kritischen Aktionen zu ermöglichen.

Auf eine Aufnahme des Observer-Konzeptes in dieses Strukturmodell wurde bewusst verzichtet, da die bisherigen Umsetzungen wie in [22] und [11] keine Fortsetzung des unterbrochenen Verhaltens ermöglichen, oder wie in [12] es gar nicht erst ermöglichen eine bereits begonnene Aktion zu unterbrechen.

Als Alternative zum priorisierten Empfangen einer Nachricht in einem separaten Observer-Verhalten ist es in dieser Arbeit möglich ein äquivalentes Verhalten zu modellieren, indem durch eine Modal Split Aktion das Verhalten zur Ausnahmebehandlung als paralleler Pfad erfasst wird, der durch Knotenprioritäten vorrangig ausgeführt wird. Mit diesem Ansatz ist es zusätzlich möglich, im Ausnahmefall die ursprüngliche Aktion standardmäßig nur zu pausieren oder alternativ diese gezielt mit der neu eingeführten Cancel Aktion abzubrechen.

In den nächsten beiden Kapiteln wird eine Ausführungssemantik für dieses Modell mit Abstrakten Zustandsmaschinen spezifiziert.

5 Spezifikation der Ausführungseinheit

Nachdem im vorherigen Kapitel ein Strukturmodell für subjektorientierte Prozessmodelle erarbeitet wurde, wird in diesem Kapitel eine zugehörige Ausführungseinheit entworfen, mit der Prozessmodelle in einer abstrakten Ausführung validiert werden können.

In Bandmann [12] wurde bereits eine Ausführungseinheit mittels ASM erarbeitet, die einen Großteil des oben beschriebenen S-PM Strukturmodells unterstützt, und sich von der ASM Spezifikation in Börger [11], und dessen CoreASM Implementierung in Lerchner and Stary [14], vor allem durch die Unterstützung von Multisubjekten, sowie dem auch hier verwendetem Konzept zum Senden und Empfangen von Nachrichten, unterscheidet.

Die Ausführungseinheit aus [12] wird daher als Grundlage der folgenden Ausführungseinheit verwendet und nach CoreASM übertragen. Um eine interaktive Validierung von Prozessmodellen zu ermöglichen wird eine Benutzeroberfläche und eine Prozessverwaltung ergänzt. Zudem wird ein Verfahren zum Initialisieren und Starten von Subjektinstanzen erarbeitet. Zur Unterstützung von Makros, von Knotenprioritäten und von den Modal Split und Modal Join Aktionen wird die `SubjectBehaviour` Regel um eine `MacroBehaviour` Regel erweitert.

5.1 Architektur der Ausführungseinheit

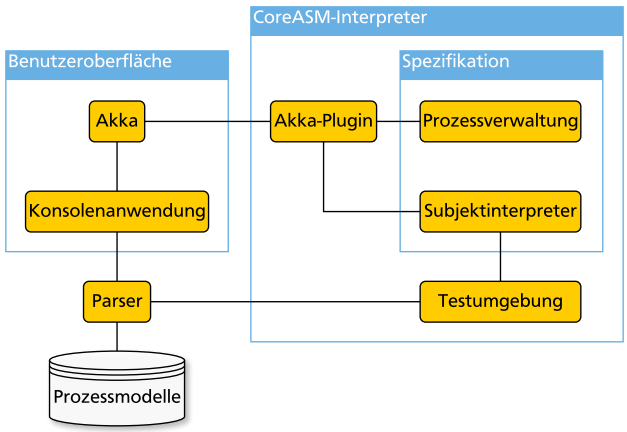


Abbildung 5.1: Architektur

Die Benutzeroberfläche ist eine in Scala entwickelte Konsolenanwendung in der die Prozessvalidierung interaktiv durchgeführt wird. Sie ist über Akka mit der Ausführungseinheit verknüpft. Sie kann Prozessmodelle einlesen, der Ausführungseinheit übergeben und diese starten und deren Ausführung kontrollieren. Dazu liest die Konsolenanwendung die Zustände der laufenden Subjekte aus, um dem Anwender Optionen zur Ablaufsteuerung zu geben.

Die Ausführungseinheit ist als asynchrone Multi-Agent-ASM modelliert und wird mit dem CoreASM Framework ausgeführt. Dabei führt ein separater ASM Agent das AkkaPlugin[1] aus, welches es ermöglicht auf den Datenspeicher von CoreASM zuzugreifen, um ASM Funktionen zu lesen und schreiben. Ein zweiter ASM Agent ist für die Prozessverwaltung zuständig. Für jede Subjekt-Agent-Verknüpfung gibt es einen weiteren ASM Agenten, der für diese den Subjektinterpreter ausführt, um deren internes Verhalten zu steuern.

Die Testumgebung ist eine Alternative zur Konsolenanwendung und Prozessverwaltung. Mit ihr können Prozessmodelle automatisiert ausgeführt werden, um Änderungen an der Semantik zu validieren. Im Folgenden wird zur Vereinfachung ausschließlich der Begriff „Konsolenanwendung“ verwendet, es ist damit jedoch auch diese Testumgebung als Alternative gemeint.

Der Parser ist eine in Scala entwickelte Bibliothek, welche Prozessmodelle in die Datenstruktur der Ausführungseinheit überführt.

5.2 Prozessverwaltung

Die Prozessverwaltung nimmt von der Konsolenanwendung Prozessmodelle entgegen und startet Prozessinstanzen.

[1] Dieses Plugin wurde im Rahmen dieser Arbeit entwickelt, da CoreASM keine *shared functions* unterstützt.

5.2.1 Kommunikation mit der Konsolenanwendung

Die Kommunikation mit der Konsolenanwendung erfolgt über zwei Funktionen in denen Aufgaben abgelegt werden:

```
function taskSetIn : -> SET
function taskSetOut : -> SET
```

Listing 7: taskSet

Die Funktion `taskSetIn` wird von der Konsolenanwendung beschrieben um Aufgaben an die Prozessverwaltung zu übergeben:

- AddProcess: hat als Parameter ein Prozessmodell das geladen werden soll.
- StartProcess: hat als Parameter die zu startende ProzessID.
- InitializeAndStartSubject: hat als Parameter eine Subjekt-Agenten-Verknüpfung des zu startenden Subjektes.

Die Funktion `taskSetOut` wird von der Konsolenanwendung gelesen um Aufgaben der Prozessverwaltung entgegenzunehmen:

- StartSubject: hat als Parameter die ProzessID, Prozessinstanznummer und die SubjektID eines Startsubjektes zu dem ein Agent ausgewählt werden muss.

Mit der `"StartProcess"` Aufgabe wird für einen bereits geladenen Prozess eine neue Instanz gestartet. Dabei hinterlegt die `StartProcess` Regel für alle Startsubjekte eine `"StartSubject"` Aufgabe an die Konsolenanwendung, da zum Erzeugen der Subjekt-Agent-Verknüpfung zuerst noch ein Agent gewählt werden muss. Die Konsolenanwendung legt nach der Agentenwahl eine `"InitializeAndStartSubject"` Aufgabe für die Prozessverwaltung ab, damit diese die Subjekt-Agent-Verknüpfungen der Startsubjekte anlegen und die Ausführung starten kann.

Mit der `WatchTaskSetIn` Regel überprüft die Prozessverwaltung regelmäßig die `taskSetIn` Funktion nach Aufgaben:

```
rule WatchTaskSetIn = {
  if (|taskSetIn| > 0) then {
    choose task in taskSetIn do {
      remove task from taskSetIn
      case (task())("task") of
        "AddProcess"   : AddProcess((task())("processID"), (task())("processMap"))
        "StartProcess" : StartProcess((task())("processID"), undef, undef)
        "InitializeAndStartSubject" : seqblock
          InitializeSubject((task())("ch"))
          StartSubject((task())("ch"))
        endseqblock
      endcase
    }
  }
}
```

Listing 8: WatchTaskSetIn

5.2.2 Starten von Prozessen

Prozesse können durch zwei Auslöser gestartet werden: durch eine `"StartProcess"` Aufgabe der Konsolenanwendung an die Prozessverwaltung, sowie bei verteilten Prozessen durch Nachrichten von externen Subjekten.

Eine neue Prozessinstanz wird mit der `StartProcess` Regel angelegt. Als ersten Parameter bekommt sie die ProzessID übergeben, über zwei optionale Parameter kann eine Subjekt-Agenten-Verknüpfung übergeben werden. Diese Verknüpfung wird übergeben wenn die Prozessinstanz durch eine Nachricht von einem externen Subjekt gestartet wird, denn dann wurde vor dem Senden der Nachricht bereits eine Zuordnung getroffen und das Subjekt kann direkt initialisiert werden.

Bevor die Startsubjekte des Prozesses gestartet werden können muss für diese eine Zuordnung der Agenten getroffen werden. Dazu wird für die Konsolenanwendung eine "StartSubject" Aufgabe in der taskSetOut Funktion abgelegt.

Um eine automatisierte Ausführung eines Prozesses, beispielsweise in einer Testumgebung, zu ermöglichen, kann über die Funktion predefinedAgents die Agentenwahl fest hinterlegt werden.

```
rule StartProcess(processID, additionalInitializationSubject, additionalInitializationAgent) = {
  // die aktuelle Prozessinstanznummer
  let PI = nextPI in {
    nextPI := PI + 1
    nextPIUsedBy(PI) := self
    result := PI

    // Schleife über alle Subjekte des Prozesses
    foreach sID in keySet(processSubject(processID)) do {
      if (sID = additionalInitializationSubject) then {
        // die Verknüpfung zum Agenten wurde als Parameter übergeben
        let ch = [processID, PI, sID, additionalInitializationAgent] in {
          InitializeSubject(ch)
        }
      }
      else {
        if (isStartSubject(processID, sID) = true) then {
          if (|predefinedAgents(processID, sID)| = 1) then {
            // die Verknüpfung zum Agenten wurde vordefiniert
            let agent = firstFromSet(predefinedAgents(processID, sID)) in
            let ch = [processID, PI, sID, agent] in {
              seq
                InitializeSubject(ch)
              next
                StartSubject(ch)
            }
          }
          else {
            // die Verknüpfung zum Agenten muss in der Konsolenanwendung getroffen werden
            add {"task" -> "StartSubject",
                 "processID" -> processID, "PI" -> PI, "subjectID" -> sID} to taskSetOut
          }
        }
      }
    }
  }
}
```

Listing 9: StartProcess

Die Vorbelegung der Agentenzuordnungen wird über die predefinedAgents Funktion abgefragt. Da Multisubjekte keine Startsubjekte sein können wird überprüft ob es auch nur genau einen vorausgewählten Agenten gibt.

Die StartProcess Regel gibt die angelegte Processinstanznummer zurück indem diese in result gespeichert wird. Falls die Regel beim Senden an ein Interfacesubjekt aufgerufen wird kann darüber der Channel des Empfängers gebildet und gespeichert werden.

```
function nextPI : -> NUMBER
function nextPIUsedBy : NUMBER -> Agents
```

Listing 10: nextPI

In der nextPI Funktion werden die Prozessinstanznummern hochgezählt.

In der `nextPIUsedBy` Funktion wird gespeichert von welchem ASM Agenten die `nextPI` Funktion mit dem entsprechenden Wert inkrementiert wurde. Dadurch wird verhindert dass in einem ASM Schritt die gleiche Prozessinstanznummer von unterschiedlichen ASM Agenten verwendet wird, da ein gleichzeitiges Inkrementieren in der ASM-Schritt-Aggregation zwar keinen Konflikt erzeugt, jedoch das Schreiben von `nextPIUsedBy`, da die gleiche Stelle mit unterschiedlichen Werten belegt werden würde. Dieser Konflikt wird von dem CoreASM Scheduler gelöst durch eine sequentielle Neuausführung der ASM Agenten.

5.3 Subjektausführung

Jede Verknüpfung eines Subjektes mit einem Agenten in einer Prozessinstanz wird mit einem eigenem ASM Agenten interpretiert. Beim (Neu-)Start eines Subjektes wird dazu ein neuer ASM Agent erzeugt und ihm der Channel der Verknüpfung zugewiesen. Im Folgenden werden die Verknüpfungen von Subjekten mit Agenten in einer Prozessinstanz vereinfacht nur als „Subjekt“ bezeichnet.

5.3.1 Initialisierung von Subjektinstanzen

Die Initialisierung eines Subjektes besteht aus der Vorbereitung von Funktionen die für den Nachrichtenaustausch und für die spätere Ausführung benötigt werden:

```
// Channel -> Boolean
function properTerminated : LIST -> BOOLEAN

rule InitializeSubject(ch) = {
  if (properTerminated(ch) = undef) then {
    // da undefiniert: es wurde noch keine Initialisierung vorgenommen
    properTerminated(ch) := true

    inputPoolDefined(ch) := {}
    inputPoolClosed(ch, undef, undef, undef) := false

    variableDefined(ch)       := {[0, "$self"], [0, "$empty"]}
    variable(ch, 0, "$self")  := ["ChannelSet", {ch}]
    variable(ch, 0, "$empty") := ["Text", ""]

    killNodes(ch) := []
  }
}
```

Listing 11: InitializeSubject.

Anhand der `properTerminated` Funktion kann festgestellt werden ob die Initialisierung bereits stattgefunden hat, ansonsten ist der Wert noch undefiniert. Nur wenn ein Subjekt wohlterminiert ist kann es gestartet werden; der Wert wird beim Beenden eines Subjektes, mit der End Aktion die in 6.12 vorgestellt wird, beschrieben.

Die Funktionen `inputPoolDefined` und `inputPoolClosed` werden zum Empfangen von Nachrichten benötigt, und werden im nächsten Kapitel im Abschnitt 6.6 vorgestellt.

Die Funktionen `variableDefined` und `variable` werden zur Verwaltung von Variablen genutzt und im nächsten Kapitel im Abschnitt 6.3 erläutert.

Die Funktion `killNodes` wird für das Abbruchverhalten benötigt im Abschnitt 5.7 vorgestellt.

5.3.2 Starten von Subjektinstanzen und deren Ausführung

Die Ausführung eines Subjektes besteht aus der Ausführung des Mainmakros, später können aus diesem heraus mit der CallMacro Aktion können weitere Makros aufgerufen werden. Der Subjektinterpreter wird nun schrittweise konstruiert, ausgehend von der Ausführung des Mainmakros werden die Regeln nach und nach in den folgenden Abschnitten erweitert.

Um die Ausführung eines Subjektes zu starten wird ein dazugehöriger ASM Agent angelegt. Dessen Programm wird zuerst auf die `StartMainMacro` Regel festgelegt, welche die Instanz des Mainmakros vorbereiten wird:

```
rule StartSubject(ch) = {
  // erzeuge einen ASM Agenten aus dem Agents universe
  extend Agents with a do {
    channelFor(a) := ch
    add a to asmAgents
    program(a) := @StartMainMacro
  }
}
```

Listing 12: StartSubject

In der `channelFor` Funktion wird gespeichert welches Subjekt der ASM Agent ausführt. Über die Funktionen `processIDFor`, `processInstanceFor`, `subjectIDFor` und `agentFor` kann der ASM Agent vereinfacht Informationen seines Subjektes nachschlagen. Beispielsweise werden `processIDFor(self)` und `subjectIDFor(self)` genutzt um Daten aus dem Prozessmodell zu laden.

Die Funktion `asmAgents` wird verwendet um eine Referenz auf alle ASM Agenten, welche Subjekte ausführen, zu halten. Die `running` Funktion dient zum Überprüfen ob ein Subjekt ausgeführt wird.

```
function asmAgents : -> SET

derived running(ch) = exists a in asmAgents with channelFor(a) = ch

// ASM Agent -> Channel
function channelFor : Agents -> LIST

derived processIDFor(a)       = nth(channelFor(a), 1)
derived processInstanceFor(a) = nth(channelFor(a), 2)
derived subjectIDFor(a)       = nth(channelFor(a), 3)
derived agentFor(a)           = nth(channelFor(a), 4)
```

Listing 13: Definitionen für StartSubject

Mit der `EnsureRunning` Regel wird ein Subjekt gestartet, falls es nicht bereits aktiv ist:

```
rule EnsureRunning(ch) = {
  if (running(ch) != true) then {
    StartSubject(ch)
  }
}
```

Listing 14: EnsureRunning

Ein Subjekt kann neben dem Mainmakro auch weitere Makroinstanzen ausführen. Um die Instanzen zu unterscheiden werden diese mit einer Makroinstanznummer identifiziert. Dem Mainmakro wird immer die Makroinstanznummer 1 zugewiesen, für jede weitere Instanz wird diese Nummer in der `nextMacroInstanceNumber` Funktion hochgezählt.

In der Funktion `macroID` wird zu jeder Makroinstanz gespeichert zu welchem Makro sie gehört. Dies wird später bei der Benutzung von Variablen genutzt um festzustellen ob deren Sichtbarkeit auf die Makroinstanz beschränkt ist .

Jeder Makroinstanz ist eine Liste aktiver Knoten `activeNodes` zugeordnet. Die Reihenfolge der Knoten in der Liste ist irrelevant; da der Knoten einer Modal Join Aktion zwischenzeitlich potentiell mehrfach vorhanden sein kann können die aktiven Knoten nicht als eine Menge modelliert werden.

```
// Channel * MacroInstanceNumber -> MacroID
function macroID : LIST * NUMBER -> NUMBER

// Channel -> Number
function nextMacroInstanceNumber : LIST -> NUMBER

// Channel * MacroInstanceNumber -> List[NodeID]
function activeNodes : LIST * NUMBER -> LIST
```

Listing 15: Definitionen für StartMainMacro

Die `StartMainMacro` Regel bereitet diese drei Funktionen nun für das Mainmakro vor:

```
rule StartMainMacro = {
  let MI          = 1 in
  let mainMacroID = subjectMainMacro(processIDFor(self), subjectIDFor(self)) in
  let startNode   = macroStartnode(processIDFor(self), mainMacroID) in {
    // Vorbereiten der Mainmakro-Instanz
    macroID(channelFor(self), MI) := mainMacroID
    activeNodes(channelFor(self), MI) := [startNode]
    nextMacroInstanceNumber(channelFor(self)) := MI  + 1
  }

  // Übergang zum normalen Verhalten
  program(self) := @SubjectBehaviour
}
```

Listing 16: StartMainMacro

Nach dem Start des Mainmakros wird das Programm des ASM Agenten festgelegt auf die `SubjectBehaviour` Regel, welche von nun an das Subjektverhalten und damit das Mainmakro ausführt:

```
rule SubjectBehaviour = {
  MacroBehaviour(1)
}
```

Listing 17: SubjectBehaviour, erster Entwurf

5.4 Ausführung von Makroinstanzen

Die `MacroBehaviour` Regel kontrolliert die Ausführung der aktiven Knoten einer Makroinstanz; als Parameter hat sie dessen Makroinstanznummer `MI`. Sie wählt einen zufälligen Knoten `n` der `activeNodes` aus und führt die `Behavior` Regel mit diesem Knoten aus.

```
rule MacroBehaviour(MI) = {
  // wähle einen zufälligen aktiven Knoten aus
  choose n in activeNodes(channelFor(self), MI) do
  seqblock
    addNodes(channelFor(self), MI, n)    := []
    removeNodes(channelFor(self), MI, n) := []

    Behaviour(MI, n)

    // entferne alte aktive Knoten
    foreach x in removeNodes(channelFor(self), MI, n) do {
      let xMI = nth(x, 1), xN = nth(x, 2) in {
        remove xN from activeNodes(channelFor(self), xMI)
      }
    }

    // füge neue aktive Knoten hinzu
    foreach x in addNodes(channelFor(self), MI, n) do {
      let xMI = nth(x, 1), xN = nth(x, 2) in {
        add xN to activeNodes(channelFor(self), xMI)
      }
    }
  endseqblock
}
```

Listing 18: MacroBehaviour, erster Entwurf

Die zufällige Auswahl des zu betrachtenden Knotens mittels **choose** stellt sicher dass Knoten nicht verhungern. Würde die Liste immer in der gleichen Reihenfolge abgearbeitet werden könnte es zu Verklemmungen kommen.

In jedem Aufruf des `MacroBhaviours` wird jeweils nur ein Knoten ausgeführt. Würden alle aktiven Knoten parallel zueinander ausgeführt dann würden sich deren Aktionen überschneiden; beispielsweise könnte die gleiche Nachricht von mehreren Receive-Knoten empfangen werden, obwohl nur eine Nachricht vorliegt. Würden die aktiven Knoten sequentiell zueinander in einem Durchgang ausgeführt werden, so wäre nicht erkennbar welche Änderungen durch welchen Knoten ausgelöst wurden. Insbesondere könnte eine Verifikationseinheit die Zwischenzustände nicht erkennen und dementsprechend keine unterschiedlichen Ausführungsreihenfolgen betrachten. [2]

Das Verhalten eines Knotens kann über die `addNodes` und `removeNodes` Funktionen weitere Knoten der Liste der aktiven Knoten einer beliebigen Makroinstanz hinzufügen und entfernen. Zur einfacheren Verwendung werden die Regeln `AddNode` und `RemoveNode` angeboten:

```
// Channel * MacroInstanceNumber * NodeID => List[(MI, NodeID)]
function addNodes : LIST * NUMBER * NUMBER -> LIST
// Channel * MacroInstanceNumber * NodeID => List[(MI, NodeID)]
function removeNodes : LIST * NUMBER * NUMBER -> LIST

rule AddNode(MI, n, MINew, nNew) = {
  add [MINew, nNew] to addNodes(channelFor(self), MI, n)
}

rule RemoveNode(MI, n, MIOld, nOld) = {
  add [MIOld, nOld] to removeNodes(channelFor(self), MI, n)
}
```

Listing 19: MacroBehaviour, Definitionen.

[2] Dass auch bei parallelen Ausführungspfaden je Durchgang nur ein Knoten ausgeführt wird ist bereits bei Börger in PERFORMSUBDGMSTEP vorhanden und wird *interleaving scheme* genannt.

Die Modal Split Aktion fügt über die `AddNode` Regel alle ihrer Folgeknoten hinzu, die Modal Join Aktion entfernt gegebenenfalls den eigenen Knoten mit der `RemoveNode` Regel, und die CallMacro Aktion hinterlegt mit der `AddNode` Regel den Startknoten ihrer erzeugten Makroinstanz.

Dieses Hinzufügen und Entfernen von aktiven Knoten wird auch von der `Proceed` Rule verwendet um von einem Knoten zum Nächsten überzugehen:

```
rule Proceed(MI, n_from, n_to) = {
  AddNode(MI, n_from, MI, n_to)
  RemoveNode(MI, n_from, MI, n_from)
}
```

Listing 20: Proceed

5.5 Ausführung von Knoten und Kanten

Die `Behaviour` Regel kontrolliert die Ausführung eines aktiven Knotens. Als Parameter bekommt sie neben der Makroinstanznummer `MI` nun auch die entsprechende KnotenID `n` übergeben. Zur Verwaltung werden folgende Funktionen verwendet:

```
// Channel * MacroInstanceNumber * NodeID -> BOOLEAN
function initializedNode : LIST * NUMBER * NUMBER -> BOOLEAN
function initializedActiveEdge : LIST * NUMBER * NUMBER -> BOOLEAN
function completed : LIST * NUMBER * NUMBER -> BOOLEAN

// Channel * MacroInstanceNumber * EdgeID -> BOOLEAN
function edgeCompleted : LIST * NUMBER * NUMBER -> BOOLEAN
```

Listing 21: Definitionen für Behaviour

```
rule Behaviour(MI, n) = {
  if (initializedNode(channelFor(self), MI, n) != true) then {
    StartNode(MI, n)
  }
  else { // der Knoten ist bereits initialisiert
    if (completed(channelFor(self), MI, n) != true) then {
      Perform(MI, n)
    }
    else { // die Knotenaktion ist bereits abgeschlossen
      if (initializedActiveEdge(channelFor(self), MI, n) != true) then {
        StartActiveEdge(MI, n)
      }
      else { // die Kantenaktion ist bereits gestartet
        let e = activeEdge(MI, n) in {
          if (edgeCompleted(channelFor(self), MI, e) != true) then {
            PerformEdge(MI, n, e)
          }
          else { // die Kantenaktion ist abgeschlossen
            Proceed(MI, n, target(processIDFor(self), e))
          }
        }
      }
    }
  }
}
```

Listing 22: Behaviour, erster Entwurf

Die StartNode Regel initialisiert den Knoten:

```
rule StartNode(MI, n) = seqblock
  InitializeCompletion(MI, n)

  DisableAllEdges(MI, n)
  initializedActiveEdge(channelFor(self), MI, n) := false

  initializedNode(channelFor(self), MI, n) := true
endseqblock
```

Listing 23: StartNode, erster Entwurf

Die Ausführung einer Aktion gliedert sich in mehrere Phasen: Zuerst wird der Knoten initialisiert und `initializedNode` auf `true` gesetzt. Die Knotenaktion wird durch die `Perform` Regel ausgeführt, bis sie selbst durch die `completed` Funktion angiebt, dass sie abgeschlossen ist. Die in der Knotenaktion gewählte ausgehende Kante wird gestartet und dies in der `initializedActiveEdge` Funktion vermerkt. Dann wird die Kantenaktion durch die `PerformEdge` Regel ausgeführt, bis sie durch das Setzen der `edgeCompleted` Funktion anzeigt, dass sie abgeschlossen ist. Nun ist das Verhalten der Aktion abgeschlossen und der Folgeknoten wird durch die `Proceed` Regel aktiviert.

Die StartActiveEdge Regel initialisiert die aktive Kante:

```
rule StartActiveEdge(MI, n) = {
  let e = activeEdge(MI, n) in {
    InitializeCompletionEdge(MI, e)
    initializedActiveEdge(channelFor(self), MI, n) := true
  }
  SetExecutionState(MI, n, REPEAT)
}
```

Listing 24: StartActiveEdge

Zur einfacheren Verwendung werden für Knoten und Kanten jeweils zwei Regeln angeboten:

```
rule InitializeCompletion(MI, n) = {
  completed(channelFor(self), MI, n) := false
}

rule InitializeCompletionEdge(MI, e) = {
  edgeCompleted(channelFor(self), MI, e) := false
}
```

Listing 25: Helfer zur Knotenausführung (1/2)

```
rule SetCompleted(MI, n) = {
  completed(channelFor(self), MI, n) := true
}

rule SetCompletedEdge(MI, n, e) = {
  edgeCompleted(channelFor(self), MI, e) := true
}
```

Listing 26: Helfer zur Knotenausführung (2/2), erster Entwurf

5.5.1 Freischaltung von Kanten

In der Knotenaktion können die ausgehenden Kanten *freigeschaltet* und *aktiviert* werden. Das Freischalten von Kanten ermöglicht es dass diese zur Auswahl stehen um aktiviert zu werden. Das Aktivieren einer Kante zeigt an dass der Knoten über ebendiese Kante verlassen werden soll. Dabei können beliebig viele Kanten freigeschaltet werden, jedoch kann nur eine Kante aktiviert werden. Bevor die Knotenaktion abgeschlossen werden kann muss genau eine der ausgehenden Kanten aktiviert worden sein.

```
// Channel * MacroInstanceNumber * EdgeID -> BOOLEAN
function exitCondition : LIST * NUMBER * NUMBER -> BOOLEAN
function edgeDecision : LIST * NUMBER * NUMBER -> BOOLEAN
```

Listing 27: Definitionen für Kanten

Ob eine Kante freigeschaltet ist oder nicht wird in der `exitCondition` Funktion hinterlegt. Ob eine Kante aktiviert wurde oder nicht wird in der `edgeDecision` Funktion hinterlegt.

In der Knotenaktion können die Freischaltungen und Aktivierungen auch zurückgesetzt werden. Dies wird beispielsweise von der Receive Aktion verwendet, um immer nur die Kanten zur Auswahl anzubieten, für die auch entsprechende Nachrichten im Inputpool vorliegen.

Zum Freischalten und Aktivieren von Kanten werden folgende Regeln angeboten:

```
rule EnableEdge(MI, e) = {
  exitCondition(channelFor(self), MI, e) := true
}

rule EnableAllEdges(MI, n) = {
  forall e in outEdgesNormal(processIDFor(self), n) do {
    EnableEdge(MI, e)
  }
}

rule ActivateEdge(MI, e) = {
  exitCondition(channelFor(self), MI, e) := true
  edgeDecision(channelFor(self), MI, e) := true
}

rule DisableEdge(MI, e) = {
  exitCondition(channelFor(self), MI, e) := false
  edgeDecision(channelFor(self), MI, e) := false
}

rule DisableAllEdges(MI, n) = {
  forall e in outEdgesNormal(processIDFor(self), n) do {
    DisableEdge(MI, e)
  }
}
```

Listing 28: Helfer zur Kantenausführung

Da nur eine Kante aktiviert werden darf wird zur Feststellung dieser Kante die Hilfsfunktion `activeEdge` angeboten. Dabei gibt die `outEdgesEnabled` Funktion - unter Berücksichtigung der Kantenprioritäten - alle ausgehenden Kanten zurück, welche freigeschaltet wurden.

```
derived isActive(MI, n, e) = (edgeDecision(channelFor(self), MI, e) = true)

derived activeEdge(MI, n) = return numres in {
  choose e in outEdgesEnabled(channelFor(self), MI, n) with isActive(MI, n, e) do {
    numres := e
  }
}
```

Listing 29: isActive und activeEdge

5.5.2 Kommunikation mit der Konsolenanwendung

Während die Freischaltung der ausgehenden Kanten automatisch durch die Knotenaktion erfolgt ist eine automatische Aktivierung einer Kante nur möglich wenn diese die „auto“-Eigenschaft hat. Im Normalfall findet daher die Auswahl der zu aktivierenden Kante in der Konsolenanwendung statt.

Die Knotenaktion muss daher der Konsolenanwendung anzeigen, dass eine Eingabe erwartet wird. Dazu wird eine Funkion `wantInput` eingeführt, in der jede Knotenaktion beliebige Aufgaben hinterlegen kann.

```
// Channel * MacroInstanceNumber * NodeID -> Set[String]
function wantInput : LIST * NUMBER * NUMBER -> SET
```

Listing 30: wantInput

Die Konsolenanwendung prüft für alle aktiven Subjekte ob deren aktive Knoten eine Eingabe benötigen und bietet dem Anwender entsprechende Optionen an. Wenn in der Konsolenanwendung eine Entscheidung getroffen wurde setzt sie die `wantInput` Funktion für diesen Knoten zurück.

Um der Konsolenanwendung anzuzeigen dass eine Kantenwahl getroffen werden soll wird die `"EdgeDecision"` Aufgabe durch die `SelectEdge` Regel hinterlegt:

```
rule SelectEdge(MI, n) = {
  if (|outEdgesEnabled(channelFor(self), MI, n)| = 0) then {
    // keine freigeschaltete Kante wählbar
  }
  else if (not(contains(wantInput(channelFor(self), MI, n), "EdgeDecision"))) then {
    // Aufgabe an Konsolenanwendung hinterlegen
    add "EdgeDecision" to wantInput(channelFor(self), MI, n)
  }
  else {
    // warten auf Konsolenanwendung bis edgeDecision gesetzt wurde
  }
}
```

Listing 31: SelectEdge, erster Entwurf

5.6 Hinzunahme von Knotenprioritäten

Die bisher vorgestellten Regeln bilden ein Grundgerüst, welches bereits mehrere aktive Knoten einer Makroinstanz unterstützt, jedoch noch keine priorisierte Ausführung eines Knotens anhand der Knotenprioritäten.

Zur Unterstützung von Knotenprioritäten muss die Auswahl des auszuführenden Knotens so angepasst werden, dass die Regeln der Knotenprioritäten berücksichtigt werden. Demnach dürfen Knoten mit einer niedrigeren Priorität nur dann ausgeführt werden, wenn Knoten mit einer höheren Priorität dies explizit zulassen. Damit die interleaving Eigenschaft beibehalten wird darf es nicht vorkommen, dass in einem Aufruf der `MacroBehaviour` Regel mehrere Knoten Änderungen

des Zustandsraums erzeugen[3]. Dennoch ist es möglich die `Behaviour` Regel für mehrere unterschiedliche Knoten in einem Durchgang auszuführen, da Aktionen auf externe Änderungen wie Benutzereingaben oder den Eingang von Nachrichten warten könnten und somit deren Ausführung keine Änderungen hervorrufen. In diesem Fall ist es also möglich die `Behaviour` Regel von weiteren Knoten aufzurufen. Um dies festzustellen wird ein Ausführungszustand eingeführt, der in der `executionState` Funktion hinterlegt wird.

Mit der Feststellung, ob durch die Ausführung der `Behaviour` Regel der Zustandsraum geändert wurde, ist es möglich in der `MacroBehaviour` Regel die aktiven Knoten in einer Schleife durchzugehen, anstatt je Durchlauf nur einen Knoten zu wählen. Zuerst wird dazu die Auswahl des auszuführenden aktiven Knotens so eingeschränkt, dass nur aus Knoten der höchsten Priorität gewählt wird. Nach dem Aufruf der `Behaviour` Regel wird überprüft, ob der Ausführungszustand DONE ist, in dem Fall wurde eine Änderung vorgenommen und es dürfen keine weiteren aktiven Knoten ausgeführt werden. Ist der Ausführungszustand NEXT wurde keine Änderung vorgenommen und es dürfen weitere Knoten mit der gleichen Priorität ausgeführt werden. Ist der Ausführungszustand LOWER wurden ebenfalls keine Änderung vorgenommen, es dürfen zusätzlich auch Knoten mit einer niedrigeren Priorität ausgeführt werden.

Mit den bisherigen Regeln werden beim Übergang von der Knoten- zur Kantenaktion mehrere globale ASM Schritte gemacht, zwischen denen sowohl andere ASM Agenten, als auch andere Knoten des eigenen Subjekts, den globalen Zustandsraum lesen und schreiben können. Das bedeutet:

- für eine Verifikationseinheit ergeben sich mehr Zwischenzustände und Kombinationen die betrachtet werden müssen
- die Zwischenschritte können zu unerwünschtem Verhalten führen. Wird beispielsweise bei der Ausführung der Receive Aktion festgestellt, dass eine auto-Kante gegangen werden kann, so würde die Kantenaktion erst in einem späterem Schritt ausgeführt werden. Zwischenzeitlich könnten jedoch andere Receive Aktionen oder andere Subjekte den Inputpool verändern.

Es muss also ermöglicht werden mehrere aufeinanderfolgende Änderungen des Zustandsraums in einem ASM Schritt zusammenzufassen. Dazu wird der Ausführungszustand REPEAT eingeführt, welcher dazu führt, dass die `Behaviour` Regel für den Knoten erneut aufgerufen wird. Wie bei dem Ausführungszustand DONE werden keine weiteren Knoten ausgeführt.

Somit ergeben sich für den Ausführungszustand folgende Rückgabewerte:

- LOWER: erlaubt die Ausführung weiterer Knoten mit niedrigerer Priorität (Beispiel: Receive Aktion ohne passende Nachrichten)
- NEXT: erlaubt die Ausführung weiterer Knoten mit gleicher Priorität (Beispiel: Receive Aktion mit passender Nachricht ohne auto-Eigenschaft)
- REPEAT: die `Behaviour` Regel soll erneut für diesen Knoten angewendet werden, andere Knoten dürfen nicht ausgeführt werden (Beispiel: Receive Aktion mit passender Nachricht für eine Auto-Kante -> Übergang zur Kantenaktion)
- DONE: erlaubt keine weitere Ausführung, die aktuellen Änderungen sollen in einem globalem ASM Schritt gespeichert werden (Beispiel: Die Kantenaktion der Receive Aktion hat eine Nachricht aus dem InputPool empfangen und entfernt)

Der Ausführungszustand wird von jeder Aktion in der `executionState` Funktion vermerkt:

```
// Channel * MacroInstanceNumber * NodeID => Int
function executionState : LIST * NUMBER * NUMBER -> NUMBER
```

Listing 32: executionState

[3] Knoten mit gleicher Priorität können nicht parallel ausgeführt werden, da es sonst möglich wäre eine Nachricht von zwei parallelen Receive Aktionen im gleichen Schritt zu empfangen. Deshalb ist es wichtig, dass die Knoten sequentiell nacheinander ausgeführt werden; wie es durch die Schleife geschieht. Wenn jedoch mehrere Knoten sequentiell ausgeführt werden und anschließend deren Änderungen in einem globalem ASM Schritt zusammengefasst werden ist es von außen nicht erkennbar, welche der Änderungen aufgrund von welchem Knoten passiert sind. Für eine Verifikationseinheit wäre damit der Zwischenzustand nicht mehr ersichtlich, obwohl er insbesondere durch den Indeterminismus der `choose` Regel relevant wäre, um andere Kombinationen der Ausführungsreihenfolge zu erkennen und zu betrachten.

Des Weiteren steigt mit jedem zusätzlichen Ausführungsergebnis die Wahrscheinlichkeit, dass die Änderungen eines ASM Agenten mit denen anderer ASM Agenten kollidieren und nicht in den globalen ASM Zustand aggregiert werden können.

```
rule MacroBehaviour(MI) = {
 let processID = processIDFor(self) in
 // merke alle aktiven Knoten zur weiteren Behandlung
 local remainingNodes := activeNodes(channelFor(self), MI) in {
   while (|remainingNodes| > 0) do
   // wähle einen zufälligen aktiven Knoten aus, der die höchste Priorität hat
   // und noch nicht behandelt wurde
   choose n in remainingNodes
     with not(exists n2 in nodes with nodePriority(processID, n2) > nodePriority(processID, n))
   do seqblock
     executionState(channelFor(self), MI, n) := undef
     addNodes(channelFor(self), MI, n)       := []
     removeNodes(channelFor(self), MI, n)    := []

     Behaviour(MI, n)

     // prüfe den Ausführungszustand: welche Knoten dürfen weiter behandelt werden?
     let state = executionState(channelFor(self), MI, n) in {
       if (state = REPEAT) then {
         remainingNodes := [n]
       }
       else if (state = DONE) then {
         remainingNodes := []
       }
       else if (state = NEXT) then {
         seq
          remove n from remainingNodes
         next
          remainingNodes := filterNodesWithSamePrio(processID, remainingNodes, nodePriority(processID, n))
       }
       else if (state = LOWER) then {
         remove n from remainingNodes
       }
     }

     // entferne alte aktive Knoten
     foreach x in removeNodes(channelFor(self), MI, n) do {
       let xMI = nth(x, 1), xN = nth(x, 2) in {
         remove xN from activeNodes(channelFor(self), xMI)
         initializedNode(channelFor(self), xMI, xN) := false
         if (xMI = MI) then {
           remove xN from remainingNodes
         }
       }
     }

     // füge neue aktive Knoten hinzu
     foreach x in addNodes(channelFor(self), MI, n) do {
       let xMI = nth(x, 1), xN = nth(x, 2) in {
         add xN to activeNodes(channelFor(self), xMI)
         if (xMI = MI) then {
           // wenn der Knoten in der gleichen Makroinstanz ist, um ihn sofort zu initialisieren
           add xN to remainingNodes
         }
       }
     }
   endseqblock
 }
}
```


Listing 33: MacroBehaviour, zweiter Entwurf

Die `MacroBehaviour` Regel wird nun um eine lokale Liste `remainingNodes` erweitert, in der zu Beginn alle aktiven Knoten vorhanden sind. Es wird in einer Schleife geprüft, ob in dieser Liste noch Knoten ausgeführt werden müssen. Ein Knoten mit der höchsten Priorität wird ausgewählt und ausgeführt. Anhand des Ausführungszustands wird diese Liste für den nächsten Schleifendurchgang gefiltert.

Da Änderungen der aktiven Knoten eine Zustandsänderung sind muss dabei der Ausführungszustand DONE zurückgegeben werden. Beim Entfernen von aktiven Knoten aus der gleichen Makroinstanz müssen diese auch aus der `remainingNodes` Liste gestrichen werden. Werden neue Knoten hinzugefügt, so werden sie in die `remainingNodes` Liste mit aufgenommen, um deren Startverhalten direkt auszuführen. Damit das Startverhalten für alle neu hinzugefügten Knoten ausgeführt werden kann muss das Startverhalten der Knoten den Ausführungszustand LOWER setzen - ansonsten würde das Startverhalten nicht für alle neuen Knoten ausgeführt, sondern möglicherweise nur für diesen einen.

Zum Setzen des Ausführungszustandes wird die `SetExecutionState` Regel angeboten:

```
rule SetExecutionState(MI, n, state) = {
  executionState(channelFor(self), MI, n) := state
}
```

Listing 34: SetExecutionState

Die in 5.5.2 eingeführte `SelectEdge` Regel muss nun angepasst werden, um einen Ausführungszustand zurückzugeben. Wenn auf eine Benutzereingabe gewartet wird oder keine Kanten zur Auswahl stehen führt die Ausführung der Regel zu keiner Änderung am Zustandsraum. Da die Auswahl einer Kante erforderlich für das Knotenverhalten ist werden keine Knoten mit einer niedrigeren Priorität erlaubt, es können jedoch problemlos Knoten mit der gleichen Priorität ausgeführt werden. Demnach wird in diesen beiden Fällen der Ausführungszustand NEXT gesetzt. Wenn mindestens eine Kante auswählbar ist, und noch nicht auf die Benutzereingabe gewartet wird, findet eine Zustandsänderung durch die Änderung der `wantInput` Funktion statt. In diesem Fall muss ein globaler ASM Schritt gemacht werden und es wird der Ausführungszustand DONE gesetzt.

```
rule SelectEdge(MI, n) = {
  if (|outEdgesEnabled(channelFor(self), MI, n)| = 0) then {
    // keine freigeschaltete Kante wählbar
    SetExecutionState(MI, n, NEXT)
  }
  else if (not(contains(wantInput(channelFor(self), MI, n), "EdgeDecision"))) then {
    // Aufgabe an Konsolenanwendung hinterlegen
    add "EdgeDecision" to wantInput(channelFor(self), MI, n)
    SetExecutionState(MI, n, DONE)
  }
  else {
    // warten auf Konsolenanwendung bis edgeDecision gesetzt wurde
    SetExecutionState(MI, n, NEXT)
  }
}
```

Listing 35: SelectEdge, finale Variante

5.6.1 macroExecutionState

Da die `MacroBehaviour` Regel später durch die CallMacro Aktion rekursiv für weitere Makroinstanzen aufgerufen wird muss auch das Makroverhalten einen Ausführungszustand zurückgeben, damit die CallMacro Aktion diesen übernehmen kann. Wenn in einem aufgerufenen Makro alle Aktionen die Ausführung von Aktionen mit einer niedrigeren Knotenpriorität erlauben setzt die CallMacro Aktion ebenfalls den Ausführungszustand NEXT und in der übergeordneten Makroinstanz können weitere Knoten ausgeführt werden.

Dazu wird nun ein Ausführungszustand der Makroausführung mit der `macroExecutionState` Funktion eingeführt. Der Ausführungszustand REPEAT wird hier jedoch nicht benötigt. Somit ergeben sich für die `macroExecutionState` Funktion folgende Werte:

- LOWER: erlaubt die Ausführung weiterer Knoten in der aufrufenden Makroinstanz mit niedrigerer Priorität (Beispiel: Makro mit einer Receive Aktion ohne passende Nachrichten)
- NEXT: erlaubt die Ausführung weiterer Knoten in der aufrufenden Makroinstanz mit gleicher Priorität (Beispiel: Makro mit einer Receive Aktion mit passender Nachricht ohne auto-Kanten)
- DONE: erlaubt keine weitere Ausführung von Knoten in der aufrufenden Makroinstanz, die aktuellen Änderungen der aufgerufenen Makroinstanz sollen in einem globalem ASM Schritt gespeichert werden (Beispiel: Makro in dem die Kantenaktion einer Receive Aktion eine Nachricht aus dem InputPool empfangen und entfernt hat)

```
// Channel * MacroInstanceNumber => Int
function macroExecutionState : LIST * NUMBER -> NUMBER
```

Listing 36: macroExecutionState

Der Wert für die `macroExecutionState` Funktion ergibt sich aus den Ausführungszuständen der Knoten dieser Makroinstanz. Dazu wird der Ausführungszustand der Makroinstanz zuerst zurückgesetzt und nach jedem Aufruf der `Behaviour` Regel eines Knotens neu bestimmt.

```
rule MacroBehaviour(MI) = {
 let processID = processIDFor(self) in
 // merke alle aktiven Knoten zur weiteren Behandlung
 local remainingNodes := activeNodes(channelFor(self), MI) in {
   seq
     // zurücksetzen des Ausführungszustands dieser Makroinstanz
     macroExecutionState(channelFor(self), MI) := undef
   next
   while (|remainingNodes| > 0) do
   // wähle einen zufälligen aktiven Knoten aus, der die höchste Priorität hat
   // und noch nicht behandelt wurde
   choose n in remainingNodes
     with not(exists n2 in nodes with nodePriority(processID, n2) > nodePriority(processID, n))
   do seqblock
     executionState(channelFor(self), MI, n) := undef
     addNodes(channelFor(self), MI, n)       := []
     removeNodes(channelFor(self), MI, n)    := []

     Behaviour(MI, n)

     // prüfe den Ausführungszustand: welche Knoten dürfen weiter behandelt werden?
     // welchen Ausführungszustand hat somit die Ausführung dieser Makroinstanz?
     let state = executionState(channelFor(self), MI, n) in {
       if (state = REPEAT) then {
         remainingNodes := [n]
         macroExecutionState(channelFor(self), MI) := DONE
       }
       else if (state = DONE) then {
         remainingNodes := []
         macroExecutionState(channelFor(self), MI) := DONE
       }
       else if (state = NEXT) then {
         seq
          remove n from remainingNodes
         next
          remainingNodes := filterNodesWithSamePrio(processID, remainingNodes, nodePriority(processID, n))
         if (macroExecutionState(channelFor(self), MI) != DONE) then {
           macroExecutionState(channelFor(self), MI) := NEXT
         }
       }
       else if (state = LOWER) then {
         remove n from remainingNodes
         if (macroExecutionState(channelFor(self), MI) != DONE and
             macroExecutionState(channelFor(self), MI) != NEXT) then {
           macroExecutionState(channelFor(self), MI) := LOWER
         }
       }
     }

     // entferne alte aktive Knoten
     foreach x in removeNodes(channelFor(self), MI, n) do ... // wie zuvor in Listing 33
     // füge neue aktive Knoten hinzu
     foreach x in addNodes(channelFor(self), MI, n) do ... // wie zuvor in Listing 33

   endseqblock
 }
}
```

Listing 37: MacroBehaviour, finale Variante

5.7 Hinzunahme von Knotenabbrüchen und Timeouts

Der Abbruch einer Knotenaktion kann durch vier Fälle ausgelöst werden: 1. es liegt ein Timeout vor, 2. in der Konsolenanwendung wurde der manuelle Abbruch gewählt, 3. der Abbruch wurde durch die Cancel-Aktion aktiviert sowie 4. durch die Beendigung der zugehörigen Makroinstanz.

In den ersten drei Fällen wird nach dem Abbruch des Knotenverhaltens der auf die Cancel- bzw. Timeout-Kante folgende Knoten aktiviert. Ob ein regulärer Timeout oder Cancel vorliegt kann über die `hasTimeout` bzw. `cancelDecision` Funktionen festgestellt werden. Die `cancelDecision` Funktion wird entweder durch die Konsolenanwendung oder die Kantenaktion der Cancel Aktion gesetzt. In diesen drei Fällen wird das Abbruchverhalten entsprechend der Regeln zur Knotenpriorität ausgeführt, das heißt erst dann, wenn die `Behaviour` Regel des Knotens von der `MacroBehaviour` Regel aus aufgerufen wird.

Wird dagegen eine Knotenaktion durch den vierten Fall abgebrochen, da die komplette Makroinstanz abgebrochen wird, so wird kein Folgeknoten aktiviert. Daher wird das Abbruchverhalten unmittelbar aufgerufen: dieser Knoten wird in der `killNodes` Funktion vermerkt und direkt im nächsten ASM Schritt von der `SubjectBehaviour` Regel aus abgebrochen.

Mit der `abortNode` Funktion kann vereinfacht überprüft werden ob ein Knoten abgebrochen werden soll. In der `abortionCompleted` Funktion wird vermerkt ob der Abbruch abgeschlossen ist oder weiter durchgeführt werden muss.

```
// Channel -> List[(MacroInstanceNumber, NodeID)]
function killNodes : LIST -> LIST

// soll die Aktion unmittelbar abgebrochen werden?
derived killNode(MI, n) = contains(killNodes(channelFor(self)), [MI, n])

// soll das Abbruchverhalten für die Aktion durchgeführt werden?
derived abortNode(MI, n) = (
  (hasTimeout(channelFor(self), MI, n) = true) or
  (cancelDecision(channelFor(self), MI, n) = true) or
  (killNode(MI, n) = true)
)

// Channel * MacroInstanceNumber * NodeID -> BOOLEAN
function abortionCompleted : LIST * NUMBER * NUMBER -> BOOLEAN
function cancelDecision    : LIST * NUMBER * NUMBER -> BOOLEAN
```

Listing 38: abortNode

Die `Behaviour` Regel wird nun erweitert um die Überprüfung ob ein Abbruch durchgeführt werden soll, welcher dann durch den Aufruf der `Abort` Regel erfolgt. Außerdem wird hier das Setzen der `executionState` Funktion mit aufgenommen, die im ersten Entwurf noch nicht betrachtet wurde:

```
rule Behaviour(MI, n) = {
  if (initializedNode(channelFor(self), MI, n) != true) then {
    StartNode(MI, n)
  }
  else { // ist bereits initialisiert
    if (abortNode(MI, n) = true) then {
      if (abortionCompleted(channelFor(self), MI, n) != true) then {
        Abort(MI, n)
      }
      else { // Abbruch ist abgeschlossen
        if (killNode(MI, n) = true) then {
          // vormerken zum direkten Entfernen
          RemoveNode(MI, n, MI, n)
        }
        else {
          // weiter mit dem Folgeknoten der Timeout- bzw. Cancel-Kante
          let e = getActiveAbortEdge(MI, n) in {
            Proceed(MI, n, target(processIDFor(self), e))
          }
        }
        SetExecutionState(MI, n, DONE)
      }
    }
    else if (completed(channelFor(self), MI, n) != true) then {
      Perform(MI, n)
    }
    else { // Knotenaktion ist bereits abgeschlossen
      if (initializedActiveEdge(channelFor(self), MI, n) != true) then {
        StartActiveEdge(MI, n)
      }
      else { // Kantenaktion ist bereits gestartet
        let e = activeEdge(MI, n) in {
          if (edgeCompleted(channelFor(self), MI, e) != true) then {
            PerformEdge(MI, n, e)
          }
          else { // Kantenaktion ist abgeschlossen
            Proceed(MI, n, target(processIDFor(self), e))
            SetExecutionState(MI, n, DONE)
          }
        }
      }
    }
  }
}
```

Listing 39: Behaviour, finale Variante

Wenn ein Knoten unmittelbar abgebrochen werden soll wird er nach Beendigung des Abbruchs aus den aktiven Knoten entfernt; bei einem Timeout oder Cancel wird der Folgeknoten der entsprechenden Kante aktiviert.

Die `SubjectBehaviour` Regel wird nun so erweitert, dass für Knoten, die unmittelbar abgebrochen werden sollen, der Abbruch direkt ausgeführt wird. Ansonsten könnte sich der Abbruch aufgrund von Knotenprioritäten anderer Knoten verzögern:

```
rule SubjectBehaviour = {
  // gibt es Knoten die direkt abgebrochen werden müssen?
  choose x in killNodes(channelFor(self)) do {
    KillBehaviour(nth(x, 1), nth(x, 2))
  }
  ifnone {
    MacroBehaviour(1)
  }
}
```

Listing 40: SubjectBehaviour, finale Variante

Die `KillBehaviour` Regel ruft die `Behaviour` Regel des abzubrechenden Knotens auf, welche aufgrund der `killNode` Funktion direkt das Abbruchverhalten ausführt.

```
rule KillBehaviour(MI, n) = {
  if (initializedNode(channelFor(self), MI, n) != true) then {
    remove [MI, n] from killNodes(channelFor(self))
    remove n from activeNodes(channelFor(self), MI)
  }
  else seqblock
    executionState(channelFor(self), MI, n) := undef
    addNodes(channelFor(self), MI, n)       := []
    removeNodes(channelFor(self), MI, n)    := []

    Behaviour(MI, n)

    // entferne alte aktive Knoten
    foreach x in removeNodes(channelFor(self), MI, n) do {
      let xMI = nth(x, 1),
        xN  = nth(x, 2) in {
        remove [xMI, xN] from killNodes(channelFor(self))
        initializedNode(channelFor(self), xMI, xN) := false
        remove xN from activeNodes(channelFor(self), xMI)
      }
    }

    // neue Knoten können nicht hinzukommen
  endseqblock
}
```

Listing 41: KillBehaviour

Wenn der zu entfernende Knoten noch nicht initialisiert wurde kann er direkt entfernt werden. Um die `Behaviour` Regel für den Knoten aufzurufen müssen, wie bei der `MacroBehaviour` Regel, die Funktionen `executionState`, `addNodes` und `removeNodes` initialisiert werden. Nach dem Aufruf der `Behaviour` Regel können die zu entfernenden Knoten entfernt werden.

Zur Verwaltung der Timeout werden zwei Funktionen benutzt:

```
// Channel * MacroInstanceNumber * NodeID -> NUMBER
function startTime : LIST * NUMBER * NUMBER -> NUMBER

// Channel * MacroInstanceNumber * NodeID -> BOOLEAN
function timeoutActive : LIST * NUMBER * NUMBER -> BOOLEAN
```

Listing 42: Definitionen Abbruch

In der `startTime` Funktion wird gespeichert zu welchem Zeitpunkt mit der Überwachung des Timeouts begonnen wurde. Die Startzeit wird je Knotentyp unterschiedlich gesetzt - beim Empfangen bereits am Anfang des Knotenverhaltens, beim Senden jedoch erst bei dem ersten Zustellversuch, und nicht bereits vor der Auswahl der Empfänger oder des Nachrichteninhaltes.

In der `timeoutActive` Funktion wird vermerkt ob die Knotenaktion einen Timeout festgestellt und aktiviert hat.

Über die Funktion `shouldTimeout` kann die Knotenaktion überprüfen ob der Timeout aktiviert werden soll, mit `hasTimeout` kann überprüft werden ob der Timeout aktiviert wurde:

```
derived shouldTimeout(ch, MI, n) = return boolres in {
  let processID = processIDOf(ch) in {
    if (hasTimeoutEdge(processID, n) = true and startTime(ch, MI, n) != undef) then {
      let e       = timeoutOutEdgeID(processID, n) in
      let timeout = edgeTimeout(processID, e) * 1000 * 1000 * 1000 in {
        boolres := ((nanoTime - startTime(ch, MI, n)) > timeout)
      }
    }
    else {
      boolres := false
    }
  }
}

derived hasTimeout(ch, MI, n) = ((shouldTimeout(ch, MI, n) = true) and (timeoutActive(ch, MI, n) = true))
```

Listing 43: shouldTimeout, hasTimeout

Zur übersichtlicheren Verwaltung der Timeouts werden drei Regeln angeboten:

```
rule StartTimeout(MI, n) = {
  startTime(channelFor(self), MI, n) := nanoTime
}
rule ResetTimeout(MI, n) = {
  startTime(channelFor(self), MI, n) := undef
  timeoutActive(channelFor(self), MI, n) := false
}
rule ActivateTimeout(MI, n) = {
  timeoutActive(channelFor(self), MI, n) := true
}
```

Listing 44: Definitionen Timeout

5.8 Knotentypen

Die bisher spezifizierten Regeln reichen bis zum Aufruf des Knoten-, Kanten-, bzw. des Abbruchverhaltens und sind unabhängig von der jeweiligen Aktion.

Die Aktionen des S-PM Strukturmodells unterschieden sich bei der Verwendung von Timeouts, bei der Freischaltung von Kanten und bei der Rückgabe der Ausführungszustände. Zudem haben nicht alle Aktionen eine Kantenaktion oder ein Abbruchverhalten. Für das Datenmodell des Interpreters wurden die Aktionen in fünf Knotentypen zusammengefasst: Action, Internal Action, Send, Receive und End.

Allgemein werden Aktionen also als Action modelliert, wobei die gewünschte Aktion in der Knotenbeschriftung gegeben wird. Diese Aktionen haben ein Standardverhalten der Timeouts und Kantenfreischaltung: Der Timeout startet beim Beginn der Aktion und anfangs sind keine ausgehenden Kanten freigeschaltet. Die Freischaltung der Kanten geschieht entweder in der Knotenaktion, oder es wird direkt beim Beenden der Knotenaktion genau eine ausgehende Kante aktiviert. Das Kantenverhalten und Abbruchverhalten ist jeweils optional.

Die anderen Knotentypen unterscheiden sich durch eigene Definitionen zum Start des Timeouts und zur Freischaltung der Kanten, weshalb diese Trennung notwendig ist.

Um bei der Action die in der Knotenbeschriftung gegebene Aktion ausführen zu können müssen die entsprechenden ASM Regeln den Beschriftungen zugeordnet sein. Diese Zuordnungen werden mit der `DefineActions` Regel beim Start des Interpreters hinterlegt, so dass zur Laufzeit die entsprechende ASM Regel nachgeschlagen werden kann.

In der Funktion `performAction` werden dabei die Regeln des Knotenverhaltens aller Aktionen hinterlegt. Eine optionale Startregel kann in der Funktion `startAction` angegeben werden. Eine optionale Regel zum Abbruchverhalten kann in der Funktion `abortAction` angegeben werden. Eine optionale Kantenaktion kann in der Funktion `performEdgeAction` angegeben werden.

```
function startAction       : STRING -> RULE
function performAction     : STRING -> RULE
function performEdgeAction : STRING -> RULE
function abortAction       : STRING -> RULE

rule DefineActions = {
  startAction("Tau")              := @StartTau
  performAction("Tau")            := @Tau

  performAction("CallMacro")      := @CallMacro
  abortAction("CallMacro")        := @AbortCallMacro

  performAction("Cancel")         := @Cancel
  performEdgeAction("Cancel")     := @PerformEdgeCancel

  performAction("VarMan")         := @VarMan
  abortAction("VarMan")           := @AbortVarMan

  performAction("ModalSplit")     := @ModalSplit
  performAction("ModalJoin")      := @ModalJoin
  performAction("CloseIP")        := @CloseIP
  performAction("OpenIP")         := @OpenIP
  performAction("CloseAllIPs")    := @CloseAllIPs
  performAction("OpenAllIPs")     := @OpenAllIPs
  performAction("IsIPEmpty")      := @IsIPEmpty
  performAction("SelectAgents")   := @SelectAgentsAction
}
```

Listing 45: performAction

Mittels der `StartNode` Regel wird jede Aktion initialisiert. Sie wurde bereits im Abschnitt 5.5 eingeführt und wird hier um den Aufruf des knotentypspezifischen Verhaltens ergänzt. Zudem wird die Initialisierung des Abbruchverhaltens ergänzt und die `wantInput` Funktion zurückgesetzt.

```
rule StartNode(MI, n) = seqblock
  InitializeCompletion(MI, n)
  abortionCompleted(channelFor(self), MI, n) := false

  ResetTimeout(MI, n)
  cancelDecision(channelFor(self), MI, n) := false

  DisableAllEdges(MI, n)
  initializedActiveEdge(channelFor(self), MI, n) := false

  wantInput(channelFor(self), MI, n) := {}

  case nodeType(processIDFor(self), n) of
    "action"         : StartAction(MI, n)
    "internalAction" : StartInternalAction(MI, n)
    "send"           : StartSend(MI, n)
    "receive"        : SetExecutionState(MI, n, LOWER)
    "end"            : StartEnd(MI, n)
  endcase

  initializedNode(channelFor(self), MI, n) := true
endseqblock
```

Listing 46: StartNode, finale Variante

Da die Receive Aktion keine Initialisierungen benötigt ist das Startverhalten direkt abgeschlossen.

Zum Initialisieren der normalen Aktionen wird der Timeout gestartet. Wenn für die Aktion eine Regel zum Start hinterlegt ist wird diese aufgerufen. Das Setzen des Ausführungszustands auf LOWER erlaubt es, dass weitere Knoten im gleichen ASM Schritt gestartet werden können.

```
rule StartAction(MI, n) = {
  let actionName = nodeAction(processIDFor(self), n) in {
    if (startAction(actionName) != undef) then {
      call startAction(actionName) (MI, n)
    }
  }

  StartTimeout(MI, n)

  SetExecutionState(MI, n, LOWER)
}
```

Listing 47: StartAction

Die `Perform` Regel ruft nur die jeweilige Knotenaktion auf:

```
rule Perform(MI, n) = {
  case nodeType(processIDFor(self), n) of
    "action"         : PerformAction(n)
    "internalAction" : PerformInternalAction(MI, n)
    "send"           : PerformSend(MI, n)
    "receive"        : PerformReceive(MI, n)
    "end"            : PerformEnd(MI, n)
  endcase
}
```

Listing 48: Perform

Für normale Aktionen wird geprüft, ob der Timeout aktiviert werden soll, ansonsten wird die Regel des Knotenverhaltens aufgerufen.

Im Prozessmodell können an Knoten von normalen Aktionen Parameter übergeben werden, beispielsweise Variablennamen für die VarMan Aktionen. Diese werden der Regel als Liste übergeben.

```
rule PerformAction(MI, n) = {
  if (shouldTimeout(channelFor(self), MI, n) = true) then {
    ActivateTimeout(MI, n)
    SetCompleted(MI, n)
  }
  else {
    let actionName = nodeAction(processIDFor(self), n),
        args       = nodeActionArguments(processIDFor(self), n) in {
      call performAction(actionName) (MI, n, args)
    }
  }
}
```

Listing 49: PerformAction

Die `PerformEdge` Regel ruft nur das Kantenverhalten auf. Die Internal Action hat kein Kantenverhalten. Da die End Aktion keine ausgehenden Kanten hat, hat sie auch kein Kantenverhalten.

```
rule PerformEdge(MI, n, e) = {
  case nodeType(processIDFor(self), n) of
    "action"         : PerformEdgeAction(MI, n, e)
    "internalAction" : SetCompletedEdge(MI, n, e)
    "send"           : PerformEdgeSend(MI, n, e)
    "receive"        : PerformEdgeReceive(MI, n, e)
  endcase
}
```

Listing 50: StartActiveEdge

Bei normalen Aktionen ist ein Kantenverhalten optional. Wurde keine Regel hinterlegt so ist es sofort abgeschlossen.

```
rule PerformEdgeAction(MI, n, e) = {
  let actionName = nodeAction(processIDFor(self), n) in {
    if (performEdgeAction(actionName) = undef) then {
      SetCompletedEdge(MI, n, e)
    }
    else {
      call performEdgeAction(actionName) (MI, n, e)
    }
  }
}
```

Listing 51: PerformEdgeAction

Die `Abort` Regel ruft das Abbruchverhalten auf.

Die Internal Action und Receive Aktionen haben kein Abbruchverhalten, von daher wird direkt die `SetAbortionCompleted` Regel aufgerufen. Die End Aktion kann nicht abgebrochen werden, daher wird für sie kein Abbruchverhalten definiert.

```
rule Abort(MI, n) = {
  case nodeType(processIDFor(self), n) of
    "action"         : AbortAction(MI, n)
    "internalaction" : SetAbortionCompleted(MI, n)
    "send"           : AbortSend(MI, n)
    "receive"        : SetAbortionCompleted(MI, n)
  endcase
}
```

Listing 52: Abort

Für normale Aktionen ist ein Abbruchverhalten optional. Wurde keine Regel für das Abbruchverhalten hinterlegt so ist es sofort abgeschlossen.

```
rule AbortAction(MI, n) = {
  let actionName = nodeAction(processIDFor(self), n) in {
    if (abortAction(actionName) = undef) then {
      SetAbortionCompleted(MI, n)
    }
    else {
      call abortAction(actionName) (MI, n)
    }
  }
}
```

Listing 53: AbortAction

Die `SetAbortionCompleted` Regel wird angeboten um den Abschluss des Abbruchverhaltens zu setzen. Die Regeln `SetCompleted` und `SetCompletedEdge` wurden bereits im Abschnitt 5.5 eingeführt, jedoch noch ohne dem Setzen eines Ausführungszustandes.

```
rule SetCompleted(MI, n) = {
  SetExecutionState(MI, n, DONE)
  completed(channelFor(self), MI, n) := true
}

rule SetAbortionCompleted(MI, n) = {
  SetExecutionState(MI, n, DONE)
  abortionCompleted(channelFor(self), MI, n) := true
}

rule SetCompletedEdge(MI, n, e) = {
  SetExecutionState(MI, n, REPEAT)
  edgeCompleted(channelFor(self), MI, e) := true
}
```

Listing 54: Helfer zur Knotenausführung (2/2), finale Variante

Die `SetCompletedAction` Regel wird von den normalen Aktionen genutzt um die Knotenaktion mit einem Resultat `res` abzuschließen, welches die entsprechende ausgehende Kante aktiviert. Wird kein Resultat übergeben so wird eine zufällige ausgehende Kante aktiviert. Dies ist üblicherweise der Fall wenn bei einer Aktion nur eine ausgehende Kante vorgesehen ist.

```
rule SetCompletedAction(MI, n, res) = {
  if (res = undef) then {
    choose e in outEdgesNormal(processID, n) do {
      ActivateEdge(MI, e)
    }
  }
  else {
    let e = getEdgeByName(processID, n, res) in {
      ActivateEdge(MI, e)
    }
  }
  SetCompleted(MI, n)
}
```

Listing 55: SetCompletedAction

In diesem Kapitel wurde zu Beginn die Architektur der Ausführungseinheit vorgestellt. Die Schnittstelle zur Konsolenanwendung erfolgt über eine eigene CoreASM Erweiterung zum Lesen und Schreiben von ASM Funktionen. Die Funktionen `taskSetIn` und `taskSetOut` werden für allgemeine Ein- und Ausgaben zur Prozessverwaltung verwendet. Über die Funktion `wantInput` können von der Konsolenanwendung Eingaben zur weiteren Ausführung der Knotenaktionen angefordert werden.

Jede Instanz eines Subjektes wird von einem ASM Agenten gesteuert. Der `SubjectBehaviour` Regel ist dabei die Mainrule des Agenten. Sie prüft zuerst ob es aktive Knoten gibt die abgebrochen werden müssen; wenn nicht wird das Verhalten des Mainmakros mit der `MacroBehaviour` Regel ausgeführt. Diese ruft das Verhalten der aktiven Knoten, je nach deren Priorität und Ausführungszustand, auf und verwaltet die Übergänge von einem aktiven Knoten zum nächsten.

Das Verhalten eines aktiven Knotens wird mit der `Behaviour` Regel gesteuert. Sie prüft ob ein manueller Abbruch oder ein automatischer Timeout vorliegt. Anschließend wird die Knoten- und Kantenaktion der jeweiligen Aktion aufgerufen, welche im nun folgenden Kapitel spezifiziert werden.

6 Spezifikation der Sprachelemente

Im vorherigen Kapitel wurde die Ausführungseinheit vorgestellt und bis zum Aufruf der jeweiligen Knoten- und Kantenaktion spezifiziert. Um die Ausführung mehrerer aktive Knoten in einer Makroinstanz, anhand des interleaving Verfahrens, zu ermöglichen wurden vier Ausführungszustände eingeführt: DONE gibt an, dass in der Regel Werte verändert wurden, REPEAT gibt an, dass das Knotenverhalten erneut aufgerufen werden soll, NEXT erlaubt die Ausführung von Knoten mit gleicher Priorität und LOWER erlaubt auch die Ausführung von Knoten mit niedrigerer Priorität. Bei jeder Auswertung der Knoten- und Kantenaktion muss einer dieser Ausführungszustände gesetzt werden müssen. Als Vereinfachung werden Hilfsregeln, wie die `SetCompleted` Regel, angeboten, die automatisch einen Ausführungszustand setzen.

In diesem Kapitel folgt die Spezifikation der Ausführungssemantik jeder Aktion des S-PM Strukturmodells. Zuerst wird das Verhalten der Internal Action spezifiziert, da sie der einfachste Knotentyp ist. Darauf folgt die Spezifizierung der Tau Aktion, als einfachste Aktion mit dem Knotentyp „Action“. Nach der Spezifizierung der Variablenmanipulation- und Inputpool-Aktionen wird das Senden und Empfangen spezifiziert, das teilweise auf die vorherigen Definitionen zurückgreift. Zum Abschluss folgen mit den Aktionen Modal Split, Modal Join, Cancel, CallMacro und End Aktionen zum Kontrollfluss.

6.1 Internal Action

Die *Internal Action* wird verwendet, um subjekt-interne Aktionen und Entscheidungen abzubilden. Sie ist beschriftet mit einer textuellen Beschreibung der Aktion, die durch den Agenten ausgeführt werden soll. Die ausgehenden Kanten sind mit den möglichen Ergebnissen der Aktion beschriftet. Da die Durchführung der modellierten Aktion durch den Agenten außerhalb der Ausführungseinheit stattfindet gibt es hier keine Kantenaktion und die Knotenaktion besteht nur aus der Auswahl der zu aktivierenden Kante.

```
rule StartInternalAction(MI, n) = {
  StartTimeout(MI, n)
  EnableAllEdges(MI, n)
  SetExecutionState(MI, n, LOWER)
}
```

Listing 56: StartInternalAction

Beim Start der Internal Action werden alle ausgehenden Kanten freigeschaltet und der Timeout gestartet. Durch das Setzen des Ausführungszustandes auf LOWER können weitere Knoten im gleichen ASM Schritt gestartet werden.

```
rule PerformInternalAction(MI, n) = {
  if (shouldTimeout(channelFor(self), MI, n) = true) then {
    ActivateTimeout(MI, n)
    SetCompleted(MI, n)
  }
  else {
    if (exists e in outEdgesEnabled(channelFor(self), MI, n)
        with isActive(MI, n, e)) then {
      SetCompleted(MI, n)
    }
    else {
      // keine ausgehende Kante aktiviert, Benutzerwahl notwendig
      SelectEdge(MI, n)
    }
  }
}
```

Listing 57: PerformInternalAction

Die Knotenaktion überprüft zuerst ob ein Timeout eingeleitet werden soll. Ist das nicht der Fall wird überprüft ob es eine ausgehende Kante gibt, die aktiviert wurde; in dem Fall ist die Knotenaktion abgeschlossen. Ansonsten wird die `SelectEdge` Regel aufgerufen, welche für die Konsolenanwendung die Aufgabe hinterlegt, dass eine Kantenwahl getroffen werden muss.

6.2 Tau (τ)

Die Tau Aktion wird als Platzhalter einer Internal Action verwendet, falls während der Modellierung das gewünschte Verhalten unbekannt ist oder als Vereinfachung nicht beschrieben wurde. Zusätzlich zur Internal Action ist es möglich anzugeben, dass automatisch eine zufällige ausgehende Kante aktiviert werden soll. Somit ist in der abstrakten Ausführung keine Benutzerinteraktion notwendig, was eine schnellere Prozessvalidierung ermöglicht.

```
rule StartTau(MI, n) = {
  // alle ausgehenden Kanten freischalten
  EnableAllEdges(MI, n)
}
```

Listing 58: StartTau

Beim Starten der Tau Aktion werden alle ausgehenden Kanten freigeschaltet, da beim Start von Aktionen vorerst alle ausgehenden Kanten deaktiviert wurden.

```
rule Tau(MI, n, args) = {
  choose e in outEdgesEnabled(processIDFor(self), n)
    with (edgeIsAuto(processIDFor(self), e) = true) do {
    ActivateEdge(MI, e)
    SetCompleted(MI, n)
  }
  ifnone {
    if (exists e in outEdgesEnabled(channelFor(self), MI, n) with isActive(MI, n, e)) then {
      SetCompleted(MI, n)
    }
    else {
      // keine ausgehende Kante aktiviert, Benutzerwahl notwendig
      SelectEdge(MI, n)
    }
  }
}
```

Listing 59: Tau

Die Knotenaktion prüft zuerst ob es automatische ausgehende Kanten gibt und aktiviert eine zufällige davon. Gibt es keine automatische Kante so wird, wie bereits bei der Internal Action, auf die `SelectEdge` Regel zurückgegriffen, welche für die Konsolenanwendung eine Aufgabe zur Kantenwahl hinterlegt.

6.3 Variablen

Zur Wiederholung aus dem S-PM Strukturmodell: Variablen werden über einen Namen identifiziert, sie haben einen Datentyp und einen Inhalt. Variablen sind auf Subjektebene definiert und können nicht von anderen Subjekten gelesen oder manipuliert werden. Variablen sind standardmäßig für alle Aktionen des Subjekts sichtbar, die Sichtbarkeit kann jedoch auf eine Makroinstanz beschränkt werden.

Zwei Variablen haben eine besondere Bedeutung und können nur gelesen werden:

`$empty` ist eine Variable ohne Inhalt. Sie kann beispielsweise beim Versenden von Nachrichten verwendet werden, falls kein Nachrichteninhalt benötigt wird.

$self ist eine Variable die den eigenen Channel beinhaltet. Diese Variable kann anstelle der Select Agents Aktion verwendet werden, um in Prozessen mit einer Dreieck- oder Sternkommunikation die Referenz auf sich selbst als Vorauswahl an andere Subjekte weiterzusenden. Zudem ermöglicht diese Variable Nachrichten an sich selbst zu senden.

In der `variableDefined` Funktion wird für jedes Subjekt hinterlegt welche Variablen in Verwendung sind. Darüber lassen sich bei der Terminierung des Subjektes alle Variablen zurücksetzen.

Die Variablen selbst werden in der `variable` Funktion gespeichert. Standardmäßig wird eine Variable unter der Makroinstanznummer 0 gespeichert, um anzugeben, dass sie in jeder Makroinstanz sichtbar ist. Wenn die Sichtbarkeit einer Variablen auf eine Makroinstanz beschränkt wurde wird deren Instanznummer als Parameter verwendet.

```
// Channel * macroInstanceNumber * varname -> (vartype, content)
function variable : LIST * NUMBER * STRING -> LIST
// Channel -> Set[(macroInstanceNumber, varname)]
function variableDefined : LIST -> SET
```

Listing 60: Definitionen für Variablen

Bei jedem Lese- oder Schreibzugriff auf eine Variable wird über die `isMacroLocalVariable` Funktion nachgeschlagen ob die Sichtbarkeit einer Variable eingeschränkt wurde. Die Sichtbarkeit einer Variablen ist auf eine Makroinstanz beschränkt, wenn im Prozessmodel der Name der Variable als Parameter des Makros angegeben wurde oder wenn der Variablenname explizit als makroinstanz-lokal angegeben wurde:

```
derived isMacroLocalVariable(MI, varname) = return boolres in {
  if (MI = 0) then {
    boolres := false
  }
  else {
    boolres := (contains(macroArguments(processIDFor(self), macroID(channelFor(self), MI)), varname)
             or contains(macroVariables(processIDFor(self), macroID(channelFor(self), MI)), varname))
  }
}
```

Listing 61: isMacroLocalVariable

Zum Schreiben von Variablen wird die `SetVar` Regel benutzt:

```
rule SetVar(MI, varname, vartype, value) = {
  if (varname != "$empty" and varname != "$self") then {
    local x := 0 in { // standardmäßig die globale Stelle verwenden
      seq
        if isMacroLocalVariable(MI, varname) then {
          x := MI // makroinstanz-lokale Variable => an der Stelle der Makroinstanz speichern
        }
      next {
        add [x, varname] to variableDefined(channelFor(self))
        variable(channelFor(self), x, varname) := [vartype, value]
      }
    }
  }
}
```

Listing 62: SetVar

Zuerst wird sichergestellt, dass die vordefinierten Variablen `"$empty"` und `"$self"` nicht überschrieben werden. Im Anschluss wird bestimmt an welcher Stelle die Variable in der `variable` Funktion gespeichert werden soll. Dann wird die Variable gespeichert und in `variableDefined` vermerkt.

Mit der `ClearVar` Regel können Variablen wieder gelöscht werden:

```
rule ClearVar(MI, varname) = {
  if (varname != "$empty" and varname != "$self") then {
    local x := 0 in { // standardmäßig die globale Stelle verwenden
      seq
        if isMacroLocalVariable(MI, varname) then {
          x := MI // makroinstanz-lokale Variable => an der Stelle der Makroinstanz löschen
        }
      next {
        remove [c, varname] from variableDefined(channelFor(self))
        variable(channelFor(self), c, varname) := undef
      }
    }
  }
}
```

Listing 63: ClearVar

Wie bei der `SetVar` Regel wird zuerst sichergestellt, dass die besonderen Variablen `"$empty"` und `"$self"` nicht gelöscht werden. Im Anschluss wird bestimmt an welcher Stelle die Variable in der `variable` Funktion gespeichert wurde. Diese Stelle wird zurückgesetzt und die Hinterlegung aus `variableDefined` entfernt.

Mit der `ClearAllVarOfMI` Regel können alle Variablen einer Makroinstanz zurückgesetzt werden:

```
rule ClearAllVarOfMI(MI) = {
  forall x in variableDefined(channelFor(self)) with (nth(x, 1) = MI) do {
    let varname = nth(x, 2) in {
      ClearVar(MI, varname)
    }
  }
}
```

Listing 64: ClearAllVarOfMI

Dazu wird in der `variableDefined` Funktion nachgeschlagen welche Variablen für die entsprechende Makroinstanz definiert wurden.

Zum Laden von Variablen wird die `loadVar` Hilfsfunktion angeboten:

```
derived loadVar(MI, varname) = return listres in {
  if isMacroLocalVariable(MI, varname) then {
    listres := variable(channelFor(self), MI, varname)
  }
  else {
    listres := variable(channelFor(self), 0, varname)
  }
}
```

Listing 65: loadVar

6.4 VarMan

Mit den Operatoren zur Variablenmanipulation ist es möglich den Inhalt von Variablen zu kombinieren, um neue Variablen zu erzeugen oder bestehende zu überschreiben. Neben den üblichen Operationen aus der Mengenlehre gibt es Operatoren zum Extrahieren von Informationen aus Nachrichten, sowie eine Operation zur Auswahl einer Teilmenge durch die Konsolenanwendung.

Bei der Verwendung der Operatoren muss darauf geachtet werden, dass die Datentypen der Quellvariablen übereinstimmen; beispielsweise kann man keine Menge von Nachrichten mit einer Menge von Channeln kombinieren.

Alle Operatoren der Variablenmanipulation werden in der VarMan Aktion zusammengefasst, welche anhand des ersten Arguments die entsprechende Operation aufruft:

```
rule VarMan(MI, n, args) = {
  case nth(args, 1) of
    "clear"                : VarMan_Clear                (MI, n, nth(args, 2))
    "assign"               : VarMan_Assign               (MI, n, nth(args, 2), nth(args, 3))
    "concatenation"        : VarMan_Concatenation        (MI, n, nth(args, 2), nth(args, 3), nth(args, 4))
    "intersection"         : VarMan_Intersection         (MI, n, nth(args, 2), nth(args, 3), nth(args, 4))
    "difference"           : VarMan_Difference           (MI, n, nth(args, 2), nth(args, 3), nth(args, 4))
    "extractContent"       : VarMan_ExtractContent       (MI, n, nth(args, 2), nth(args, 3))
    "extractChannel"       : VarMan_ExtractChannel       (MI, n, nth(args, 2), nth(args, 3))
    "extractCorrelationID" : VarMan_ExtractCorrelationID(MI, n, nth(args, 2), nth(args, 3))
    "selection"            : VarMan_Selection            (MI, n, nth(args, 2), nth(args, 3),
                                                                 nth(args, 4), nth(args, 5))
  endcase
}
```

Listing 66: VarMan

6.4.1 Basisoperationen

Mittels der Assign Operation wird der Inhalt einer Variablen in eine andere Kopiert. Mit der Clear Operation wird eine Variable gelöscht.

```
rule VarMan_Assign(MI, n, A, X) = {
  let a = loadVar(MI, A) in {
    SetVar(MI, X, head(a), last(a))
    SetCompletedAction(MI, n, undef)
  }
}
```

Listing 67: VarMan: Assign

```
rule VarMan_Clear(MI, n, X) = {

  ClearVar(MI, X)
  SetCompletedAction(MI, n, undef)

}
```

Listing 68: VarMan: Clear

Eine Vereinigung von zwei Mengen wird mit der Concatenation Operation durchgeführt, die Schnittmenge wird mit der Intersection Operation gebildet und mit der Difference Operation wird die Restmenge gebildet.

```
rule VarMan_Concatenation(MI, n, A, B, X) = {
  let a = loadVar(MI, A) in
  let b = loadVar(MI, B) in {
    if (a = undef and b = undef) then {
      ClearVar(MI, X)
      SetCompletedAction(MI, n, undef)
    }
    else if (a = undef) then {
      SetVar(MI, X, head(b), last(b))
      SetCompletedAction(MI, n, undef)
    }
    else if (b = undef) then {
      SetVar(MI, X, head(a), last(a))
      SetCompletedAction(MI, n, undef)
    }
    else if (head(a) = head(b) and contains({"MessageSet", "ChannelSet"}, head(a))) then {
      let x = (last(a) union last(b)) in {
        SetVar(MI, X, head(a), x)
        SetCompletedAction(MI, n, undef)
      }
    }
  }
}
```

Listing 69: VarMan: Concatenation

```
  rule VarMan_Intersection(MI, n, A, B, X) = {
    let a = loadVar(MI, A) in
    let b = loadVar(MI, B) in {
      if (a = undef or b = undef) then {
        ClearVar(MI, X)
        SetCompletedAction(MI, n, undef)
      }
      else if (head(a) = head(b)
   and contains(
     {"MessageSet", "ChannelSet"}, head(a)
   ))
      then {
        let x = (last(a) intersect last(b)) in {
          SetVar(MI, X, head(a), x)
          SetCompletedAction(MI, n, undef)
        }
      }
    }
  }
```

Listing 70: VarMan: Intersection

```
rule VarMan_Difference(MI, n, A, B, X) = {
  let a = loadVar(MI, A) in
  let b = loadVar(MI, B) in {
    if (a = undef) then {
      ClearVar(MI, X)
      SetCompletedAction(MI, n, undef)
    }
    else if (b = undef) then {
      SetVar(MI, X, head(a), last(a))
      SetCompletedAction(MI, n, undef)
    }
    else if (head(a) = head(b)
 and contains(
   {"MessageSet", "ChannelSet"}, head(a)
 ))
    then {
      let x = (last(a) diff last(b)) in {
        SetVar(MI, X, head(a), x)
        SetCompletedAction(MI, n, undef)
      }
    }
  }
}
```

Listing 71: VarMan: Difference

6.4.2 Extract

Mit den Extract Operationen werden Inhalte aus Nachrichtenmengen extrahiert. Zur Erinnerung aus dem S-PM Strukturmodell: Eine Nachricht beinhaltet immer einen Absenderchannel, einen Nachrichtentyp, und den Nachrichteninhalt. Sie hat außerdem eine CorrelationID, die standardmäßig 0 ist. Der Nachrichteninhalt besteht analog zu Variablen aus einem Datentyp und dem eigentlichen Inhalt.

```
derived msgChannel(msg)       = nth(msg, 1)
derived msgType(msg)          = nth(msg, 2)
derived msgContent(msg)       = nth(msg, 3)
derived msgCorrelationID(msg) = nth(msg, 4)
derived msgIsReservation(msg) = nth(msg, 5)
```

Listing 72: Nachrichtenkomponenten

Zur Unterscheidung von Nachrichten und Reservierungen wird in dieser Spezifikation eine zusätzliche Angabe „isReservation“ hinzugefügt. Somit ergibt sich in der Umsetzung ein 5-Tupel (Absenderchannel, Nachrichtentyp, Nachrichteninhalt, CorrelationID, isReservation). Für Variablenoperationen mit mit Nachrichten ist das Feld isReservation immer false, da die Nachricht bereits empfangen wurden.

Wurde beispielsweise eine verschachtelte Nachricht empfangen, also eine Nachricht die ein MessageSet mit weiteren Nachrichten beinhaltet, aus deren Inhalt mit der Selection Operation eine Teilmenge gewählt werden soll, so müssen die Inhalte der Nachricht zuerst in eine Variable entpackt werden. Dazu dient die ExtractContent Operation. Mit der ExtractChannel Operation ist es möglich nur die Absenderchannel aus einer Menge von Nachrichten zu entpacken.

Es kann auch die CorrelationID entpackt werden, dabei muss jedoch darauf geachtet werden, dass sich in der Nachrichtenmenge keine unterschiedlichen CorrelationIDs befinden - die Menge sollte daher die Mächtigkeit 1 haben.

```
rule VarMan_ExtractContent(MI, n, A, X) = {
  let a = loadVar(MI, A) in {
    // sicherstellen dass die Variable eine Menge von Nachrichten beinhaltet
    if (head(a) = "MessageSet") then {
      let messages = last(a) in
      let messagesContent = map(messages, @msgContent) in {
        if (| messagesContent | = 1) then {
          // wenn nur eine Nachricht in der Menge ist oder alle Nachrichten den gleichen Inhalt haben
          // kann der Inhalt direkt gespeichert werden
          let content = firstFromSet(messagesContent) in {
            SetVar(MI, X, head(content), last(content))
            SetCompletedAction(MI, n, undef)
          }
        }
        else if (| messagesContent | > 1) then {
          // einen zufällige Inhalt der Nachrichtenmenge für späteren Vergleich merken
          let m1 = firstFromSet(messagesContent) in {
            // sicherstellen dass mindestens dieser Nachrichteninhalt eine Menge enthält
            if (head(m1) = "MessageSet" or head(m1) = "ChannelSet") then {
              // sicherstellen dass alle Nachrichteninhalte den gleichen Datentyp haben
              if (forall m in messagesContent holds (head(m) = head(m1))) then {
                // alle Inhalte extrahieren, vereinigen und speichern
                SetVar(MI, X, head(m1), flattenSet(map(messagesContent, @last)))
                SetCompletedAction(MI, n, undef)
              }
            }
          }
        }
      }
    }
  }
}
```

Listing 73: VarMan: ExtractContent

```
rule VarMan_ExtractChannel(MI, n, A, X) = {
  let a = loadVar(MI, A) in {
      /* sicherstellen dass die Variable eine Menge von Nachrichten beinhaltet */
    if (head(a) = "MessageSet") then {
      let msgs = last(a) in
      // alle Absenderchannel extrahieren
      let msgsChs = map(msgs, @msgChannel) in {
        if (| msgsChs | > 0) then {
          SetVar(MI, X,
            "ChannelSet", msgsChs)
          SetCompletedAction(MI, n, undef)
        }
      }
    }
  }
}
```

Listing 74: VarMan: ExtractChannel

```
rule VarMan_ExtractCorrelationID(MI, n, A, X) = {
  let a = loadVar(MI, A) in {
    if (head(a) = "MessageSet") then {
      let msgs = last(a) in
      // alle CorrelationIDs extrahieren
      let msgsCIDs = map(msgs, @msgCorrelationID) in {
        if (| msgsCIDs | = 1) then {
          SetVar(MI, X,
            "CorrelationID", firstFromSet(msgsCIDs))
          SetCompletedAction(MI, n, undef)
        }
      }
    }
  }
}
```

Listing 75: VarMan: ExtractCorrelationID

6.4.3 Selection

Mit der Selection Operation kann in der Konsolenanwendung eine Teilmenge ausgewählt werden. Beispielsweise können alle Antworten auf eine Angebotsaufforderung in einer Variable gespeichert werden, um dann eine Auswahl zu treffen welche Angebote näher in Betracht bezogen werden sollen. Für die Selection Operation muss neben der Ursprungsmenge angegeben werden wieviele Elemente ausgewählt werden sollen.

```
// Channel * MacroInstanceNumber * NodeID -> ELEMENT
function selectionVartype  : LIST * NUMBER * NUMBER -> STRING
function selectionData     : LIST * NUMBER * NUMBER -> LIST
function selectionOptions  : LIST * NUMBER * NUMBER -> LIST
function selectionMin      : LIST * NUMBER * NUMBER -> NUMBER
function selectionMax      : LIST * NUMBER * NUMBER -> NUMBER
function selectionDecision : LIST * NUMBER * NUMBER -> SET
function selectionResult   : LIST * NUMBER * NUMBER -> SET
```

Listing 76: Definitionen für Selection

In der `selectionVartype` Funktion wird für die Konsolenanwendung gespeichert um welchen Datentyp es sich handelt. In `selectionData` werden die Daten gespeichert, in `selectionOptions` eine menschenlesbare Darstellung der Daten. In `selectionMin` und `selectionMax` wird angegeben wieviele Einträge auszuwählen sind; der Maximalwert kann durch **undef** auf „beliebig viele“ gesetzt werden. In der Konsolenanwendung wird nur die Darstellung der `selectionOptions` verwendet. Die in der Oberfläche getroffene Auswahl der Indizes wird in `selectionDecision` gespeichert. In `selectionResult` werden dann die Daten der ausgewählten Indizes gespeichert, um sie in der Zielvariablen zu speichern.

Da die Selection Funktionalität auch an anderer Stelle verwendet wird ist die `VarMan_Selection` Regel nur eine Hülle zum Aufruf der eigentlichen `Selection` Regel:

```
rule VarMan_Selection(MI, n, srcVarname, dstVarname, minimum, maximum) = {
  let src = loadVar(MI, srcVarname),
      res = selectionResult(channelFor(self), MI, n) in
  if (res = undef) then {
    // es liegt noch kein Ergebnis vor
    Selection(MI, n, src, minimum, maximum)
  }
  else {
    // Ergebnis mit dem Ursprungsdatentyp speichern
    selectionResult(channelFor(self), MI, n) := undef
    SetVar(MI, dstVarname, head(src), res)
    SetCompletedAction(MI, n, undef)
  }
}
```

Listing 77: VarMan: Selection

Für eine menschenlesbare Darstellung werden Nachrichten und Channel in einen String überführt. Dazu werden zwei Helferfunktionen verwendet:

```
derived msgToString(msg) = "'" + msgContent(msg) + "' from '" + msgChannel(msg1) + "'" +
                           " with messageType '" + msgType(msg) + "'" +
                           " and correlationId of '" + msgCorrelationID(msg) + "'"
derived chToString(ch)   = nth(ch, 4) + "@" + nth(ch, 3) + "@" + nth(ch, 2) + "@" + nth(ch, 1)
```

Listing 78: msgToString, chToString

Die `Selection` Regel prüft zuerst ob die Daten für die Konsolenanwendung vorbereitet wurden. Falls nicht wird die Quellmenge in eine Liste umgewandelt, um stabile Indizes zu erhalten und für die Elemente eine menschenlesbare Darstellung zu erstellen. Zusätzlich werden der Datentyp und die Minimal-/Maximalangaben übernommen.

Solange in der Konsolenanwendung keine Auswahl getroffen wurde wird auf diese gewartet. Knoten mit niedriger Priorität dürfen nicht ausgeführt werden. Wurde die Auswahl getroffen steht sie als Menge der gewählten Indizes zur Verfügung. Die ausgewählten Indizes der Quellliste werden in `selectionResult` gespeichert.

Dabei wählt die `pickItems` Funktion anhand der Indizes aus dem zweiten Argument die Daten aus der Liste des ersten Arguments aus und gibt eine neue Liste zurück.

```
rule Selection(MI, n, src, minimum, maximum) = {
  if (selectionData(channelFor(self), MI, n) = undef) then {
    // vorbereiten der Daten für die Konsolenanwendung
    if (head(src) = "MessageSet") then {
      let l = toList(last(src)) in {
        selectionData   (channelFor(self), MI, n) := l
        selectionOptions(channelFor(self), MI, n) := map(l, @msgToString)
      }
    }
    else if (head(src) = "ChannelSet") then {
      let l = toList(last(src)) in {
        selectionData   (channelFor(self), MI, n) := l
        selectionOptions(channelFor(self), MI, n) := map(l, @chToString)
      }
    }
    selectionVartype (channelFor(self), MI, n) := head(src)
    selectionMin     (channelFor(self), MI, n) := minimum
    selectionMax     (channelFor(self), MI, n) := maximum
    selectionDecision(channelFor(self), MI, n) := undef
    SetExecutionState(MI, n, REPEAT)
  }
  else if (selectionDecision(channelFor(self), MI, n) = undef) then {
    // warten auf die Konsolenanwendung
    if not(contains(wantInput(channelFor(self), MI, n), "SelectionDecision")) then {
      add "SelectionDecision" to wantInput(channelFor(self), MI, n)
      SetExecutionState(MI, n, DONE)
    }
    else {
      SetExecutionState(MI, n, NEXT)
    }
  }
  else {
    // Ergebnis bestimmen und speichern
    let data = selectionData(channelFor(self), MI, n),
        indices = selectionDecision(channelFor(self), MI, n) in
    let res = pickItems(data, indices) in {
      selectionResult(channelFor(self), MI, n) := res
    }
    // zurücksetzen der Daten
    ResetSelection(MI, n)
    SetExecutionState(MI, n, REPEAT)
  }
}

rule AbortVarMan(MI, n) = {
  ResetSelection(MI, n)
  SetAbortionCompleted(MI, n)
}

rule ResetSelection(MI, n) = {
    selectionVartype (channelFor(self), MI, n) := undef
    selectionData    (channelFor(self), MI, n) := undef
    selectionOptions (channelFor(self), MI, n) := undef
    selectionMin     (channelFor(self), MI, n) := undef
    selectionMax     (channelFor(self), MI, n) := undef
    selectionDecision(channelFor(self), MI, n) := undef
}
```

Listing 79: Selection

6.5 Select Agents

Mit der Select Agents Aktion kann eine explizite Zuordnung von Subjekten und Agenten getroffen werden, um neue Verknüpfungen in einer Variablen zu speichern. Das Ergebnis der Aktion sind Channel, die in einer Variablen gespeichert werden. Dies ist nützlich, um vor dem Senden die Empfänger festzulegen oder um in Prozessen mit einer Dreieckkommunikation eine Auswahl zu einem dritten Subjekt zu treffen, welche an das zweite kommuniziert werden soll. Wird bei einem Senden keine Variable mit Empfängern angegeben so wird automatisch das Verhalten der Select Agents Aktion aufgerufen, um neue Verknüpfungen anzulegen und somit die Empfänger zu bestimmen. Um nicht bei jedem Senden die Auswahl erneut treffen zu müssen, und dabei möglicherweise neue Verknüpfungen zu erzeugen, empfiehlt es sich die getroffene Auswahl in einer Variablen zu speichern, insbesondere um sicherzustellen dass bei jedem darauffolgenden Senden die gleichen Empfänger verwendet werden.

Die Select Agents Aktion legt für die Konsolenanwendung eine Aufgabe in `wantInput` ab und hinterlegt Informationen zu dem Subjekt zu dem Agenten gewählt werden müssen:

```
// Channel * MacroInstanceNumber * NodeID
function selectAgentsDecision  : LIST * NUMBER * NUMBER -> SET
function selectAgentsProcessID : LIST * NUMBER * NUMBER -> STRING
function selectAgentsSubjectID : LIST * NUMBER * NUMBER -> STRING
function selectAgentsCountMin  : LIST * NUMBER * NUMBER -> NUMBER
function selectAgentsCountMax  : LIST * NUMBER * NUMBER -> NUMBER
function selectAgentsResult    : LIST * NUMBER * NUMBER -> SET
```

Listing 80: Definitionen für SelectAgents

In den Funktionen `selectAgentsProcessID` und `selectAgentsSubjectID` wird hinterlegt für welches Subjekt die Agenten gewählt werden sollen, für Interfacesubjekte wird dazu die SubjektID des externen Subjekts bereits mit dem externen Prozessmodell aufgelöst. In `selectAgentsCountMin` und `selectAgentsCountMax` wird hinterlegt wie viele Agenten gewählt werden sollen. In `selectAgentsDecision` speichert die Konsolenanwendung die gewählten Agenten. In `selectAgentsResult` speichert die Select Agents Aktion die Channel der erzeugten Verknüpfungen, damit sie später von der `SelectAgentsAction` Regel zum Speichern in der Variablen verwendet werden können.

Die `SelectAgentsAction` Regel ist eine Hülle zum Aufruf der eigentlichen `SelectAgents` Regel:

```
rule SelectAgentsAction(MI, n, args) = {
  let
    varname  = nth(args, 1),
    sIDLocal = nth(args, 2),
    countMin = nth(args, 3),
    countMax = nth(args, 4) in
  {
    if (selectAgentsResult(channelFor(self), MI, n) != undef) then {
      SetVar(MI, varname, "ChannelSet", selectAgentsResult(channelFor(self), MI, n))
      selectAgentsResult(channelFor(self), MI, n) := undef
      SetCompletedAction(MI, n, undef)
    }
    else {
      SelectAgents(MI, n, sIDLocal, countMin, countMax)
    }
  }
}
```

Listing 81: SelectAgentsAction

Dabei ruft die `SelectAgentsAction` Regel so lange die `SelectAgents` Regel auf, bis diese die erzeugten Verknüpfungen in `selectAgentsResult` hinterlegt hat. Diese werden dann in der Zielvariablen gespeichert.

```
rule SelectAgents(MI, n, sIDLocal, countMin, countMax) = {
  let processID = processIDFor(self),
      PI        = processInstanceFor(self) in
  let resolvedInterface = resolveInterfaceSubject(sIDLocal) in
  let resolvedProcessID = nth(resolvedInterface, 1),
      resolvedSubjectID = nth(resolvedInterface, 2) in
  if (selectAgentsDecision(channelFor(self), MI, n) != undef) then {
    // Verknüpfungen zu der Auswahl bilden; siehe 2. Teil
  }
  else if(hasSizeWithin(predefinedAgents(processID, PI, sIDLocal), countMin, countMax) = true) then {
    selectAgentsDecision(channelFor(self), MI, n) := predefinedAgents(processID, PI, sIDLocal)
    SetExecutionState(MI, n, REPEAT)
  }
  else {
    if not(contains(wantInput(channelFor(self), MI, n), "SelectAgentsDecision")) then {
      add "SelectAgentsDecision" to wantInput(channelFor(self), MI, n)
      selectAgentsProcessID(channelFor(self), MI, n) := resolvedProcessID
      selectAgentsSubjectID(channelFor(self), MI, n) := resolvedSubjectID
      selectAgentsCountMin (channelFor(self), MI, n) := countMin
      selectAgentsCountMax (channelFor(self), MI, n) := countMax
      selectAgentsResult   (channelFor(self), MI, n) := undef
      SetExecutionState(MI, n, DONE)
    }
    else {
      SetExecutionState(MI, n, NEXT)
    }
  }
}
```

Listing 82: SelectAgents, 1. Teil

Zuerst wird überprüft ob bereits eine Auswahl der Agenten getroffen wurde. Ist das nicht der Fall wird überprüft ob im Prozessmodell für das Subjekt eine passende Vorauswahl der Agenten hinterlegt wurde und angewendet. Diese Vorauswahl ist für Prozesse der Testumgebung notwendig, um eine Benutzerinteraktion zu vermeiden. Ansonsten wird eine Aufgabe für die Konsolenanwendung hinterlegt. Ist die Aufgabe bereits hinterlegt muss auf die Konsolenanwendung gewartet werden, es werden keine Knoten mit niedrigerer Priorität erlaubt.

Sobald die Agenten ausgewählt wurden können die zugehörigen Verknüpfungen erzeugt werden. Interne Subjekte werden mittels der `InitializeSubject` Regel initialisiert. Für Interfacesubjekte muss vom zugehörigen Prozess zuerst eine neue Prozessinstanz durch die `StartProcess` Regel angelegt werden. Durch die letzten beiden Parameter `additionalInitializationSubject` und `additionalInitializationAgent` wird dort das Subjekt initialisiert. Anhand der zurückgegebenen Prozessinstanznummer kann der zugehörige Channel gespeichert werden.

In beiden Fällen wird das Subjekt zwar initialisiert, jedoch noch nicht gestartet. Das Starten der Subjekte findet erst nach dem erfolgreichem Senden von Nachrichten statt. Zu beachten ist, dass durch das Starten von externen Prozessen die dortigen Startsubjekte jedoch bereits gestartet werden.

```
rule SelectAgents(MI, n, sIDLocal, countMin, countMax) = {
  let processID = processIDFor(self),
      PI        = processInstanceFor(self) in
  let resolvedInterface = resolveInterfaceSubject(sIDLocal) in
  let resolvedProcessID = nth(resolvedInterface, 1),
      resolvedSubjectID = nth(resolvedInterface, 2) in
  if (selectAgentsDecision(channelFor(self), MI, n) != undef) then {
    local setres1 := {} in
      seq
        foreach agent in selectAgentsDecision(channelFor(self), MI, n) do {
          if (resolvedProcessID = processID) then {
            let ch = [processID, PI, sIDLocal, agent] in {
              InitializeSubject(ch)
              add ch to setres1
            }
          }
          else {
            local numres1 in {
              seq
                numres1 <- StartProcess(resolvedProcessID, resolvedSubjectID, agent)
              next
                let ch = [resolvedProcessID, numres1, resolvedSubjectID, agent] in
                  add ch to setres1
            }
          }
        }
      next
        selectAgentsResult(channelFor(self), MI, n) := setres1

    selectAgentsDecision (channelFor(self), MI, n) := undef
    selectAgentsCountMin (channelFor(self), MI, n) := undef
    selectAgentsCountMax (channelFor(self), MI, n) := undef
    selectAgentsProcessID(channelFor(self), MI, n) := undef
    selectAgentsSubjectID(channelFor(self), MI, n) := undef
    SetExecutionState(MI, n, REPEAT)
  }
  else // Vorbereitungen und Warten auf die Konsolenanwendung; siehe 1. Teil
}
```

Listing 83: SelectAgents, 2. Teil

6.6 Inputpool

Der Inputpool besteht aus Warteschlangen je (Absender-Subjekt-ID, Nachrichtentyp, CorrelationID)-Tupel und ist dem Empfänger-Channel zugeordnet. Die Warteschlangen werden als FIFO Listen umgesetzt.

```
// receiverChannel * senderSubjID * messageType * correlationID -> [msg1, msg2, ...]
function inputPool : LIST * STRING * STRING * NUMBER -> LIST

// receiverChannel -> {[senderSubjID, messageType, correlationID], ..}
function inputPoolDefined : LIST -> SET

// receiverChannel * senderSubjID * messageType * correlationID
function inputPoolClosed : LIST * STRING * STRING * NUMBER -> BOOLEAN
```

Listing 84: Definitionen Inputpool

In der `inputPoolDefined` Funktion wird gespeichert welche Warteschlangen bereits in Verwendung sind.

Mit der `inputPoolClosed` Funktion wird erfasst ob eine Warteschlange geschlossen ist. An der speziellen Stelle `inputPoolClosed(ch, undef, undef, undef)` wird angegeben ob der komplette IP geschlossen ist.

6.6.1 CloseIP und OpenIP

Durch das Öffnen und Schließen von Warteschlangen kann das Empfangen von weiteren Nachrichten verhindert werden. Wird versucht an eine geschlossene Warteschlange zu senden, so blockiert die Send Aktion. Standardmäßig sind alle Warteschlangen und der Inputpool an sich geöffnet, es ist möglich den kompletten Inputpool zu schließen oder nur einzelne Warteschlangen.

Mit der CloseIP Aktion wird eine Warteschlange geschlossen und mit der OpenIP Aktion wird eine Warteschlange geöffnet.

```
rule CloseIP(MI, n, args) = {
  let senderSubjID       = nth(args, 1),
    messageType          = nth(args, 2),
    correlationIDVarname = nth(args, 3) in
  let correlationID = loadCorrelationID(MI, correlationIDVarname) in {
    if (inputPool(channelFor(self), senderSubjID, messageType, correlationID) = undef) then {
      // Warteschlange noch nicht in Verwendung => anlegen
      add [senderSubjID, messageType, correlationID] to inputPoolDefined(channelFor(self))
      inputPool(channelFor(self), senderSubjID, messageType, correlationID) := []
    }
    inputPoolClosed(channelFor(self), senderSubjID, messageType, correlationID) := true
    SetCompletedAction(MI, n, undef)
  }
}
```

Listing 85: CloseIP

Falls die zu schließende Warteschlange noch nicht in Verwendung ist wird sie hier erzeugt, damit sie später beim Öffnen aller Warteschlangen mit erfasst werden kann.

Die `loadCorrelationID` Funktion lädt aus der angegebenen Variablen die CorrelationID. Ist keine Variable angegeben wird 0 verwendet. Beinhaltet die Variable anstatt einer CorrelationID eine Menge von Nachrichten, so wird die CorrelationID daraus extrahiert - wie bei der ExtractCorrelation Operation der VarMan Aktion darf die Nachrichtenmenge keine Nachrichten mit unterschiedlichen CorrelationIDs beinhalten.

Das Öffnen einer Warteschlange wird analog dazu modelliert:

```
rule OpenIP(MI, n, args) = {
  let senderSubjID       = nth(args, 1),
    messageType          = nth(args, 2),
    correlationIDVarname = nth(args, 3) in
  let correlationID = loadCorrelationID(MI, correlationIDVarname) in {
    if (inputPool(channelFor(self), senderSubjID, messageType, correlationID) = undef) then {
      // Warteschlange noch nicht in Verwendung => anlegen
      add [senderSubjID, messageType, correlationID] to inputPoolDefined(channelFor(self))
      inputPool(channelFor(self), senderSubjID, messageType, correlationID) := []
    }
    inputPoolClosed(channelFor(self), senderSubjID, messageType, correlationID) := false
    SetCompletedAction(MI, n, undef)
  }
}
```

Listing 86: OpenIP

6.6.2 CloseAllIPs und OpenAllIPs

Mit der CloseAllIPs Aktion werden der Inputpool an sich sowie alle Warteschlange geschlossen, und mit der OpenAllIPs Aktion werden der Inputpool an sich sowie alle Warteschlangen geöffnet.

```
rule CloseAllIPs(MI, n, args) = {
  inputPoolClosed(channelFor(self), undef, undef, undef) := true

  // zusätzlich jede Warteschlange schließen
  forall key in inputPoolDefined(channelFor(self)) do {
    let sID = nth(key, 1),
      mT    = nth(key, 2),
      cID   = nth(key, 3) in {
      inputPoolClosed(channelFor(self), sID, mT, cID) := true
    }
  }
  SetCompletedAction(MI, n, undef)
}
```

Listing 87: CloseAllIPs

Dazu wird an der besonderen Stelle `inputPoolClosed(ch, undef, undef, undef)` vermerkt dass der komplette IP geschlossen ist. Außerdem werden alle Warteschlangen, die in Verwendung sind, in einer Schleife durchlaufen und geschlossen.

Die OpenAllIPs Aktion wird analog modelliert:

```
rule OpenAllIPs(MI, n, args) = {
  inputPoolClosed(channelFor(self), undef, undef, undef) := false

  // zusätzlich jede Warteschlange öffnen
  forall key in inputPoolDefined(channelFor(self)) do {
    let sID = nth(key, 1),
      mT    = nth(key, 2),
      cID   = nth(key, 3) in {
      inputPoolClosed(channelFor(self), sID, mT, cID) := false
    }
  }
  SetCompletedAction(MI, n, undef)
}
```

Listing 88: OpenAllIPs

6.6.3 IsIPEmpty

Mit der IsIPEmpty Aktion kann überprüft werden ob sich in einer Warteschlange Nachrichten oder Reservierungen befinden. Dazu muss die Aktion zwei ausgehende Kanten haben, die mit `"true"` und `"false"` beschriftet sind.

Die Aktion hat drei Parameter, die in der Knotenbeschriftung gegeben sein müssen: die SubjektID, einen Nachrichtentyp und ein Variablenname für eine CorrelationID. Jeder Parameter kann auf `"undef"` gesetzt werden um ihn als Wildcard zu benutzen. Mit der Knotenbeschriftung `"IsIPEmpty(undef, undef, undef)"` kann somit überprüft werden ob der komplette Inputpool leer ist.

```
rule IsIPEmpty(MI, n, args) = {
  let senderSubjID       = nth(args, 1),
    messageType          = nth(args, 2),
    correlationIDVarname = nth(args, 3) in
  let correlationID = loadCorrelationID(MI, correlationIDVarname) in {
    if (inputPoolIsEmpty(channelFor(self), senderSubjID, messageType, correlationID) = true) then {
      SetCompletedAction(MI, n, "true") // aktiviert dabei die Kante die mit "true" beschriftet ist
    }
    else {
      SetCompletedAction(MI, n, "false") // aktiviert dabei die Kante die mit "false" beschriftet ist
    }
  }
}
```

Listing 89: IsIPEmpty

Die `inputPoolIsEmpty` Funktion zählt die Nachrichten der entsprechenden Warteschlange.

6.7 Send

Das Senden findet in zwei Phasen statt: In der Knotenaktion werden Reservierungen für alle Empfänger abgelegt, um sicherzustellen, dass beim Senden an Multisubjekte die Nachricht an alle gewünschten Empfänger gesendet werden kann. In der Kantenaktion werden diese durch die tatsächliche Nachricht ausgetauscht.

Es darf kein Empfänger *non-proper terminated* sein. Zum Ablegen einer Reservierung in einer Warteschlange darf diese nicht bis zum IP Limit gefüllt sein und die Warteschlange darf auch nicht geschlossen sein. Ist eine dieser Bedingungen für einen Empfänger nicht erfüllt kann die Knotenaktion nicht abgeschlossen werden und somit blockiert die Send Aktion. Für alle anderen Empfänger werden jedoch bereits Reservierungen abgelegt.

In der Kantenaktion werden beim Austausch der Reservierungen durch die Nachricht die Empfänger gestartet.

Die Send Aktion kann, neben den optionalen timeout und cancel Kanten, nur eine einzige ausgehende Kante haben. Es ist nicht vorgesehen eine Nachricht an unterschiedliche Subjekte in einer Aktion zu senden. An der Kante müssen Parameter zum Empfängersubjekt, der Anzahl der Empfänger und der Nachrichtentyp angegeben sein. Optional können über Variablen der Nachrichteninhalt und die Channel der Empfänger angegeben werden. Falls sie nicht angegeben wurden wartet die Send Aktion auf eine Eingabe über die Konsolenanwendung; die Auswahl der Empfängerchannel greift dabei auf die `SelectAgents` Regel zurück.

```
// Channel * MacroInstanceNumber * NodeID -> Set[Channel]
function receivers : LIST * NUMBER * NUMBER -> SET
// Channel * MacroInstanceNumber * NodeID -> STRING
function messageContent : LIST * NUMBER * NUMBER -> LIST
// Channel * MacroInstanceNumber * NodeID -> Set[Channel]
function reservationsDone : LIST * NUMBER * NUMBER -> SET

function nextCorrelationID : -> NUMBER
function nextCorrelationIDUsedBy : NUMBER -> Agents
```

Listing 90: Definitionen Send

Wenn die Channel der Empfänger bestimmt wurden werden sie in der `receivers` Funktion gehalten. In der `messageContent` Funktion wird der Nachrichteninhalt gehalten. In der `reservationsDone` Funktion wird vermerkt für welche Empfänger bereits Reservierungen abgelegt wurden.

In der globalen `nextCorrelationID` Funktion werden die verwendeten CorrelationIDs hochgezählt. In der `nextCorrelationIDUsedBy` Funktion wird analog zu `nextPIUsedBy` gespeichert von welchem ASM Agenten die `nextCorrelationID` Funktion inkrementiert wurde. Dadurch wird verhindert dass in einem ASM Schritt die gleiche CorrelationID von mehreren ASM Agenten verwendet wird.

```
rule StartSend(MI, n) = {
  let processID = processIDFor(self) in
  let e         = firstNormalOutEdgeID(processID, n) in {
    receivers(channelFor(self), MI, n) := undef
    reservationsDone(channelFor(self), MI, n) := {}

    // ggf. Nachrichteninhalt aus Variable laden und merken
    let contentVarname = messageContentVar(processID, firstNormalOutEdgeID(processID, n)) in
      if (contentVarname != undef and contentVarname != "") then
        messageContent(channelFor(self), MI, n) := loadVar(MI, contentVarname)

    // ggf. CorrelationID verwenden und in Variable speichern
    let cIDVarname = messageNewCorrelationVar(processID, e) in
      if (cIDVarname != undef and cIDVarname != "") then {
        SetVar(MI, cIDVarname, "CorrelationID", nextCorrelationID)
        nextCorrelationID := nextCorrelationID + 1
        nextCorrelationIDUsedBy(nextCorrelationID) := self
      }
  }

  SetExecutionState(MI, n, LOWER)
}
```

Listing 91: StartSend

Mit der `StartSend` Regel werden beim Start der Send Aktion die `receivers` und `reservationsDone` Funktionen initialisiert. Wenn eine Variable für den Nachrichteninhalt angegeben wurde wird dieser geladen. Wenn angegeben wurde dass eine neue CorrelationID erzeugt werden soll wird diese in der angegebenen Variable gespeichert und der globale Zähler erhöht.

```
rule PerformSend(MI, n) = {
  if (receivers(channelFor(self), MI, n) = undef) then {
    // Empfänger bestimmen
    SelectReceivers(MI, n)
  }
  else {
    if (messageContent(channelFor(self), MI, n) = undef) then {
      // Nachrichteninhalt bestimmen
      SetMessageContent(MI, n)
    }
    else {
      if (startTime(channelFor(self), MI, n) = undef) then {
        StartTimeout(MI, n)
        SetExecutionState(MI, n, REPEAT)
      }
      else {
        if (|receivers(channelFor(self), MI, n)| = |reservationsDone(channelFor(self), MI, n)|) then {
          if (anyNonProperTerminated(reservationsDone(channelFor(self), MI, n))) = true) then {
            if (shouldTimeout(channelFor(self), MI, n) = true) then {
              SetCompleted(MI, n)
              ActivateTimeout(MI, n)
            }
            else {
              SetExecutionState(MI, n, NEXT)
            }
          }
          else {
            // alle Reservierungen abgelegt und keine Empfänger non-proper terminated
            let e = firstNormalOutEdgeID(processIDFor(self), n) in {
              ActivateEdge(MI, e)
            }
            SetCompleted(MI, n)
          }
        }
        // Anzahl Reservierungen nicht ausreichend
        else if (shouldTimeout(channelFor(self), MI, n) = true) then {
          SetCompleted(MI, n)
          ActivateTimeout(MI, n)
        }
        else {
          DoReservations(MI, n)
        }
      }
    }
  }
}
```

Listing 92: PerformSend

Mit der `PerformSend` Regel wird die Knotenaktion ausgeführt. Zuerst wird auf die Festlegung der Empfänger und des Nachrichteninhalts gewartet. Erst danach wird der Timeout gestartet, damit dieser sich auf die Dauer der Platzierung der Reservierungen bezieht, und nicht bereits auf das Warten der vorherigen Benutzereingaben.

Wenn für alle Empfänger eine Reservierung abgelegt werden konnte wird sichergestellt dass keine Empfänger, die bereits in einem früherem Schritt eine Reservierung erhalten haben, zwischenzeitlich non-proper-terminated wurden. In dem Fall kann die Knotenaktion nicht verlassen werden, ansonsten wird die ausgehende Kante aktiviert und die Knotenaktion damit abgeschlossen.

Falles es Empfänger gibt für die noch keine Reservierungen abgelegt wurden, und kein Timeout vorliegt, wird durch die `DoReservations` Regel versucht diese Reservierungen abzulegen.

```
rule SelectReceivers(MI, n) = {
  let processID  = processIDFor(self) in
  let e          = firstNormalOutEdgeID(processID, n) in
  let receiverID = messageSubjectId(processID, e),
      receiverVarname = messageSubjectVar(processID, e),
      countMin   = messageSubjectCountMin(processID, e),
      countMax   = messageSubjectCountMax(processID, e) in {
    if (receiverVarname != undef and receiverVarname != "") then {
      // Empfänger aus Variable laden, prüfen ob Anzahl passend ist
      let rChs = loadChannelsFromVariable(MI, receiverVarname, receiverID) in {
        if (| rChs | = 0 or (countMin != 0 and | rChs | < countMin)) then {
          // Empfänger aus der Variable ungeeignet
          SetExecutionState(MI, n, NEXT)
        }
        else if (countMax != 0 and | rChs | > countMax)  then {
          // in der Konsolenanwendung eine Teilmenge auswählen
          SelectReceivers_Selection(MI, n, rChs, countMin, countMax)
        }
        else {
          // Empfänger können verwendet werden
          receivers(channelFor(self), MI, n) := rChs
          SetExecutionState(MI, n, REPEAT)
        }
      }
    }
    else {
      // keine Variable mit Empfängern => neue Verknüpfungen erzeugen
      let selection = selectAgentsResult(channelFor(self), MI, n) in
        if (selection = undef) then {
          SelectAgents(MI, n, receiverID, countMin, countMax)
        }
        else {
          receivers(channelFor(self), MI, n) := selection
          selectAgentsResult(channelFor(self), MI, n) := undef
          SetExecutionState(MI, n, REPEAT)
        }
    }
  }
}
```

Listing 93: SelectReceivers

Zur Auswahl der Empfänger wird zuerst überprüft ob eine Variable mit Empfängerchanneln angegeben ist. Ist keine Variable für die Empfänger angegeben, so wird die bereits definierte `SelectAgents` Regel aufgerufen, um neue Verknüpfungen zu erzeugen.

Ist eine Variable angegeben so wird überprüft ob die Anzahl Channel in der Variable der gewünschten Anzahl an Empfängern entspricht. Wenn die untere Schranke auf 0 gesetzt ist bedeutet dies, dass es keine untere Schranke gibt. Die Empfänger dürfen jedoch nicht leer sein. Wenn die obere Schranke auf 0 gesetzt ist bedeutet dies, dass die Anzahl der Empfänger nach oben unbeschränkt ist. Sind in der Variable zu wenig Channel so blockiert die Auswahl und damit die Senden Aktion. Sind in der Variable zu viele Empfängerchannel so muss eine Auswahl der gewünschten Empfänger getroffen werden. Die Auswahl der Empfänger erfolgt durch die `SelectReceivers_Selection` Regel, welche die bereits definierte `Selection` Regel der Variablenmanipulation aufruft:

```
rule SelectReceivers_Selection(MI, n, rChs, minimum, maximum) = {
  let res = selectionResult(channelFor(self), MI, n) in
  if (res = undef) then {
    let src = ["ChannelSet", rChs] in {
      Selection(MI, n, src, minimum, maximum)
    }
  }
  else {
    selectionResult(channelFor(self), MI, n) := undef
    receivers(channelFor(self), MI, n) := res
    SetExecutionState(MI, n, REPEAT)
  }
}
```

Listing 94: SelectReceivers Selection

Wenn der Nachrichteninhalt `messageContent` nicht bereits in `StartSend` durch eine Variable gefüllt wurde muss der Inhalt in der Konsolenanwendung eingegeben werden:

```
rule SetMessageContent(MI, n) = {
  if not(contains(wantInput(channelFor(self), MI, n), "MessageContentDecision")) then {
    add "MessageContentDecision" to wantInput(channelFor(self), MI, n)
    SetExecutionState(MI, n, DONE)
  }
  else {
    SetExecutionState(MI, n, NEXT)
  }
}
```

Listing 95: SetMessageContent

Wenn alle Informationen vorliegen versucht die `DoReservations` Regel die Reservierungen bei allen Empfängern, die noch keine Reservierung erhalten haben, zu platzieren:

```
rule DoReservations(MI, n) = {
  local hasPlacedAnyReservation := false in
  seq
    foreach receiver in receivers(channelFor(self), MI, n)
        with not(exists r2 in reservationsDone(channelFor(self), MI, n) with r2 = receiver) do {
      local hasPlacedReservation := false in
        seq
          hasPlacedReservation <- DoReservation(MI, n, receiver)
        next
          hasPlacedAnyReservation := hasPlacedAnyReservation or hasPlacedReservation
    }
  next
    if (hasPlacedAnyReservation = true) then
      SetExecutionState(MI, n, DONE)
    else
      SetExecutionState(MI, n, NEXT)
}
```

Listing 96: DoReservations

Dabei gibt die `DoReservations` Regel zurück, ob für diesen Empfänger eine Reservierung abgelegt werden konnte. Wurden in einem Durchgang keine Reservierungen abgelegt können andere Knoten mit der gleichen Priorität ausgeführt werden; Knoten mit niedrigerer Priorität werden jedoch nicht erlaubt.

```
rule DoReservation(MI, n, receiverChannel) = {
  if (properTerminated(receiverChannel) != true) then {
    result := false
  }
  else {
    let processID         = processIDFor(self) in
    let e                 = firstNormalOutEdgeID(processID, n),
        senderChannel     = channelFor(self),
        receiverProcessID = processIDOf(receiverChannel) in
    let senderSubjectID   = searchSenderSubjectID(processID, subjectIDFor(self), receiverProcessID),
        msgType           = messageType(processID, e),
        msgCorrelationID  = loadCorrelationID(MI, messageNewCorrelationVar(processID, e)),
        ipCorrelationID   = loadCorrelationID(MI, messageWithCorrelationVar(processID, e)) in
    let reservationMsg    = [senderChannel, messageType(processID, e), {}, msgCorrelationID, true] in {
      seq
        if (inputPool(receiverChannel, senderSubjectID, msgType, ipCorrelationID) = undef) then {
          // noch keine Warteschlange vorhanden => anlegen
          add [senderSubjectID, msgType, ipCorrelationID] to inputPoolDefined(receiverChannel)
          inputPool(receiverChannel, senderSubjectID, msgType, ipCorrelationID) := []
        }
      next
        if (inputPoolIsClosed(receiverChannel, senderSubjectID, msgType, ipCorrelationID) = true) then {
          // Warteschlange ist geschlossen
          result := false
        }
        else if (inputPoolGetFreeSpace(receiverChannel, senderSubjectID, msgType) = 0) then {
          // Warteschlange ist voll
          result := false
        }
        else {
          // alles ok, Reservierung am Ende anfügen
          enqueue reservationMsg into inputPool(receiverChannel, senderSubjectID, msgType, ipCorrelationID)
          add receiverChannel to reservationsDone(channelFor(self), MI, n)
          result := true
        }
    }
  }
}
```

Listing 97: DoReservation

Mit der `searchSenderSubjectID` Funktion wird für das Senden an Subjekte in externen Prozessen die eigene SubjektID im dortigen Prozess nachgeschlagen, indem dort das Interfacesubjekt gesucht wird, welches auf das eigene Subjekt verweist.

Zuerst wird sichergestellt dass die Warteschlange des Empfängers definiert ist. Im Anschluss wird überprüft ob die Warteschlange, oder auch der komplette Inputpool an sich, geschlossen ist, und ob die Warteschlange bereits voll ist. In dem Fall kann keine Reservierung abgelegt werden. Ansonsten wird die Reservierung an das Ende der Warteschlange angefügt und vermerkt dass für diesen Empfänger eine Reservierung abgelegt wurde.

Zum Abbruch der Send Aktion müssen alle bereits platzierten Reservierungen entfernt werden:

```
rule AbortSend(MI, n) = {
  foreach r in reservationsDone(channelFor(self), MI, n) do {
    CancelReservation(MI, n, r)
  }
  SetAbortionCompleted(MI, n)
}
```

Listing 98: AbortSend

Nachdem in der Knotenaktion alle Reservierungen platziert wurden werden in der Kantenaktion die Reservierungen durch die eigentliche Nachricht ausgetauscht.

```
rule PerformEdgeSend(MI, n, e) = {
  // Reservierungen mit der Nachricht austauschen und ggf. Empfänger starten
  foreach r in reservationsDone(channelFor(self), MI, n) do {
    ReplaceReservation(MI, n, r)
    EnsureRunning(r)
  }

  // ggf. Empfänger in Variable speichern
  let storeReceiverVarname = messageStoreReceiverVar(processIDFor(self), e) in
    if (storeReceiverVarname != undef and storeReceiverVarname != "") then {
      SetVar(MI, storeReceiverVarname, "ChannelSet", reservationsDone(channelFor(self), MI, n))
    }

  SetCompletedEdge(MI, n, e)
}
```

Listing 99: PerformEdgeSend

Die `ReplaceReservation` Regel führt dabei den Austausch durch, die `EnsureRunning` Regel startet den Empfänger, wenn er nicht bereits ausgeführt wird. Wurde eine Variable zum Speichern der Empfänger angegeben werden diese nun gespeichert.

Dazu wird die Reservierung in der Warteschlange gesucht und ersetzt:

```
rule ReplaceReservation(MI, n, receiverChannel) = {
  let processID         = processIDFor(self) in
  let e                 = firstNormalOutEdgeID(processID, n),
      senderChannel     = channelFor(self),
      receiverProcessID = processIDOf(receiverChannel) in
  let senderSubjectID   = searchSenderSubjectID(processID, subjectIDFor(self), receiverProcessID),
      msgType           = messageType(processID, e),
      msgContent        = messageContent(channelFor(self), MI, n),
      msgCorrelationID  = loadCorrelationID(MI, messageNewCorrelationVar(processID, e)),
      ipCorrelationID   = loadCorrelationID(MI, messageWithCorrelationVar(processID, e)) in
  let reservationMsg    = [senderChannel, msgType, {}, msgCorrelationID, true],
      message           = [senderChannel, msgType, msgContent, msgCorrelationID, false],
      IP = inputPool(receiverChannel, senderSubjectID, msgType, ipCorrelationID) in
  let i  = getIndexOfMessage(IP, reservationMessage) in {
    inputPool(receiverChannel, senderSubjectID, msgType, ipCorrelationID) := setnth(IP, i, message)
  }
}
```

Listing 100: ReplaceReservation

Zum Entfernen von Reservierungen im Falle eines Abbruchs wird sie aus der Warteschlange gestrichen:

```
rule CancelReservation(MI, n, receiverChannel) = {
  let processID         = processIDFor(self) in
  let e                 = firstNormalOutEdgeID(processID, n),
      senderChannel     = channelFor(self),
      receiverProcessID = processIDOf(receiverChannel) in
  let senderSubjectID   = searchSenderSubjectID(processID, subjectIDFor(self), receiverProcessID),
      msgType           = messageType(processID, e),
      msgCorrelationID  = loadCorrelationID(MI, messageNewCorrelationVar(processID, e)),
      ipCorrelationID   = loadCorrelationID(MI, messageWithCorrelationVar(processID, e)) in
  let reservationMsg    = [senderChannel, msgType, {}, msgCorrelationID, true],
      IP = inputPool(receiverChannel, senderSubjectID, msgType, ipCorrelationID) in
  let i  = getIndexOfMessage(IP, reservationMsg) in {
      inputPool(receiverChannel, senderSubjectID, c, ipCorrelationID) := dropnth(IP, i)
  }
}
```

Listing 101: CancelReservation

6.8 Receive

Das Empfangen von Nachrichten besteht aus zwei Phasen: In der Knotenaktion der Receive Aktion werden die Warteschlangen des Inputpools regelmäßig auf Nachrichten überprüft und dementsprechend die ausgehenden Kanten freigeschaltet. Durch die Kantenwahl wird die Knotenaktion beendet und in der Kantenaktion werden die Nachrichten aus der Warteschlange entfernt.

Neben den optionalen timeout und cancel Kanten können beliebig viele ausgehende Kanten modelliert werden, welche entsprechende Parameter zu den zu empfangenen Nachrichten haben müssen. An jeder Kante müssen mindestens die Parameter zum Sendersubjekt und Nachrichtentyp angegeben sein. Optional können weitere Parameter verwendet werden um die Anzahl der Sender von Multisubjekten zu bestimmen, zur Verwendung von CorrelationIDs, oder um die empfangenen Nachrichten in einer Variablen zu speichern. Je Kante kann jeweils nur ein Sendersubjekt und ein Nachrichtentyp angegeben werden - das Empfangen von mehreren Nachrichten mit unterschiedlichen Absendersubjekten oder unterschiedlichen Nachrichtentypen ist nicht in einem Schritt möglich.

Liegen zu mehreren Kanten passende Nachrichten vor so muss der Benutzer eine Kante auswählen deren Nachrichten empfangen werden sollen. Über Kantenprioritäten kann diese Auswahl eingeschränkt werden. Liegt eine Nachricht einer Kante mit der auto-Eigenschaft vor wird die Knotenaktion direkt beendet. Wenn keine Nachrichten empfangen werden können wird die Ausführung von Knoten mit niedrigerer Priorität erlaubt.

```
rule PerformReceive(MI, n) = {
  if (startTime(channelFor(self), MI, n) = undef) then {
    StartTimeout(MI, n)
    SetExecutionState(MI, n, REPEAT)
  }
  else {
    if (shouldTimeout(channelFor(self), MI, n) = true) then {
      SetCompleted(MI, n)
      ActivateTimeout(MI, n)
    }
    else {
      // es liegt kein Timeout vor
      seq
        // ausgehende Kanten aktivieren bzw. deaktivieren
        forall e in outEdgesNormal(processIDFor(self), n) do {
          CheckIP(MI, n, e)
        }
      next
        let enabledOutEdges = outEdgesEnabled(channelFor(self), MI, n) in {
          if (|enabledOutEdges| = 0) then {
            // keine ausgehende Kante aktiviert, Knoten mit niederigerer Priorität erlauben
            SetExecutionState(MI, n, LOWER)
          }
          else {
            seq
              if (|enabledOutEdges| = 1) then {
                // gibt es genau eine ausgehende Kante, die freigeschaltet und auto ist?
                let e = firstFromSet(enabledOutEdges) in {
                  if (edgeIsAuto(processIDFor(self), e) = true) then {
                    ActivateEdge(MI, e)
                  }
                }
              }
            next
              if (exists e in enabledOutEdges with isActive(MI, n, e)) then {
                // Kantenwahl wurde getroffen
                SetCompleted(MI, n)
              }
              else {
                // keine ausgehende Kante aktiviert, Benutzerwahl notwendig
                SelectEdge(MI, n)
              }
          }
        }
    }
  }
}
```

Listing 102: PerformReceive

In der Knotenaktion wird zuerst der Timeout gestartet und im jedem Durchgang überprüft. Im ersten Durchgang gibt `shouldTimeout` immer **`false`** zurück, da durch den Ausführungszustand REPEAT diese erste Prüfung noch im gleichen ASM Schritt stattfindet und somit die Startzeit noch der aktuellen Zeit entspricht, was ein Empfangen mit timeout=0 ermöglicht. Anschließend wird zu jeder normalen ausgehende Kante der Inputpool mit der `CheckIP` Regel überprüft ob entsprechende Nachrichten vorliegen, und dementsprechend die zugehörige Kante aktiviert bzw. deaktiviert. Ist genau eine ausgehende Kante aktiv und bei dieser die auto-Eigenschaft angegeben so wird die Knotenaktion automatisch erfolgreich beendet; sind mehrere ausgehende Kanten aktiv so muss eine Entscheidung in der Konsolenanwendung getroffen

werden.

Die CheckIP Regel prüft zu der entsprechenden Kante die zugehörige Warteschlange des Inputpools und aktiviert oder deaktiviert die Kante entsprechend:

```
rule CheckIP(MI, n, e) = {
  let processID         = processIDFor(self) in
  let senderSubjectID   = messageSubjectId(processID, e),
      msgType           = messageType(processID, e),
      countMin          = messageSubjectCountMin(processID, e),
      ipCorrelationID   = loadCorrelationID(MI, messageWithCorrelationVar(processID, e)) in
  let availableMsgs     = availableMessages(channelFor(self), senderSubjectID, msgType, ipCorrelationID) in
    if (|availableMsgs| >= messageSubjectCountMin(processID, e)) then
      EnableEdge(MI, e)
    else
      DisableEdge(MI, e)
}
```

Listing 103: CheckIP

Dabei lädt die availableMessages Funktion alle Nachrichten aus der Warteschlange des (senderSubjectID, msgType, ipCorrelationID) Tupels, entfernt aus diesen geladenen Nachrichten Reservierungen und reduziert anschließend Nachrichten von gleichen Sendern auf die jeweils älteste.

Durch die Reduzierung wird sichergestellt, dass beim Empfangen von Multisubjekten die Nachrichten auch tatsächlich von unterschiedlichen Absendern stammen, sollten mehrere Nachrichten des gleichen Senders vorliegen.

Die Kantenaktion besteht aus dem Empfangen der Nachrichten:

```
// Channel * MacroInstanceNumber * NodeID -> Set[Messages]
function receivedMessages : LIST * NUMBER * NUMBER -> SET

rule PerformEdgeReceive(MI, n, e) = {
  let processID        = processIDFor(self) in
  let senderSubjectID  = messageSubjectId(processID, e),
      sChsVarname      = messageSubjectVar(processID, e),
      msgType          = messageType(processID, e),
      countMax         = messageSubjectCountMax(processID, e) in {
      ipCorrelationID  = loadCorrelationID(MI, messageWithCorrelationVar(processID, e))
      msgStoreVarname  = messageStoreMessagesVar(processID, e) in
  seq
    InputPool_Pop(MI, n, senderSubjectID, msgType, ipCorrelationID, countMax)
  next
    if (msgStoreVarname != undef and msgStoreVarname != "") then
      let msgs = receivedMessages(channelFor(self), MI, n) in
        SetVar(MI, msgStoreVarname, "MessageSet", msgs)
  SetCompletedEdge(MI, n, e)
}
```

Listing 104: PerformEdgeReceive

Die InputPool_Pop Regel entfernt bis zu countMax Nachrichten aus der Warteschlange des (senderSubjectID, msgType, ipCorrelationID) Tupels; nach den gleichen Regeln wie die availableMessages Funktion . Sollten sich mehr Nachrichten in der Warteschlange befinden verbleiben diese dort. Die entfernten Nachrichten werden in der receivedMessages Funktion gespeichert damit diese im Anschluss in einer Variable gespeichert werden können.

6.9 Modal Split / Modal Join

Mittels der Modal Split Aktion werden parallele Ausführungspfade gestartet, welche in der Modal Join Aktion wieder vereinigt werden.

```
rule ModalSplit(MI, n, args) = {
  foreach e in outEdgesNormal(processIDFor(self), n) do {
    AddNode(MI, n, MI, target(processIDFor(self), e))
  }
  RemoveNode(MI, n, MI, n)
  SetExecutionState(MI, n, DONE)
}
```

Listing 105: ModalSplit

Dazu aktiviert die Modal Split Aktion alle ihre Folgeknoten und deaktiviert sich selbst.

```
// Channel * MacroInstanceNumber * joinNode -> Number
function joinCount : LIST * NUMBER * NUMBER -> NUMBER

rule ModalJoin(MI, n, args) =
  let splitCount = nth(args, 1) in {
  seq
    // bisher zusammengeführte Ausführungspfade zählen
    if (joinCount(channelFor(self), MI, n) = undef) then {
      joinCount(channelFor(self), MI, n) := 1
    }
    else {
      joinCount(channelFor(self), MI, n) := joinCount(channelFor(self), MI, n) + 1
    }
  next
    // sind alle Ausführungspfade zusammengeführt?
    if (joinCount(channelFor(self), MI, n) >= splitCount) then {
      // Zähler zurücksetzen und normal fortfahren
      joinCount(channelFor(self), MI, n) := undef
      SetCompletedAction(MI, n, undef)
    }
    else {
      // eigenen Knoten aus den aktiven Knoten streichen
      RemoveNode(MI, n, MI, n)
      SetExecutionState(MI, n, DONE)
    }
}
```

Listing 106: ModalJoin

In der `joinCount` Funktion zählt die Modal Join Aktion mit wie oft sie aktiviert wurde. Der Wert wird verglichen mit der Anzahl der durch die Modal Split Aktion erzeugten Ausführungspfade. Dazu muss der Modal Join Aktion als Argument die Anzahl der Ausführungspfade übergeben werden. Diese Anzahl kann automatisch beim Parsen des Prozessmodells erfasst werden und muss somit nicht explizit im Prozess modelliert werden.

In der Funktion `joinCount` wird mitgezählt wie oft die Modal Join Aktion bereits aufgerufen wurde. Sind noch weitere Pfade aktiv so entfernt sich der Knoten selbst; ist es der letzte Ausführungspfad wird die weitere Ausführung fortgesetzt.

6.10 Cancel Aktion

Mit der Cancel Aktion können andere Aktionen abgebrochen werden. Die ausgehenden Kanten der Cancel Aktion werden mit der KnotenID der abzubrechenden Aktion beschriftet, die ausgehenden Kanten können sowohl Kantenprioritäten als auch eine auto-Eigenschaft haben. Der Knoten der abzubrechenden Aktion muss über eine Cancel-Kante verfügen, die jedoch auch versteckt sein kann.

Die ausgehenden Kanten der Cancel Aktion werden abhängig davon, ob die abzubrechende Aktion aktiv ist oder nicht, freigeschaltet. Die Cancel Aktion prüft regelmäßig ihre ausgehenden Kanten und aktiviert bzw. deaktiviert diese abhängig davon ob der jeweils angegebene Knoten aktiv ist.

```
rule Cancel(MI, n, args) = {
  seq
    // Freischaltung der ausgehenden Kanten
    forall e in outEdgesNormal(processIDFor(self), n) do {
      CheckCancel(MI, n, e)
    }
  next
    let enabledOutEdges = outEdgesEnabled(channelFor(self), MI, n) in {
      if (|enabledOutEdges| = 0) then {
        // keine ausgehenden Kanten freigeschaltet
        SetExecutionState(MI, n, LOWER)
      }
      else {
        seq
          // ist eine automatische Kantenwahl möglich?
          if (|enabledOutEdges| = 1) then {
            let e = firstFromSet(enabledOutEdges) in {
              if (edgeIsAuto(processIDFor(self), e) = true) then {
                edgeDecision(channelFor(self), MI, e) := true
              }
            }
          }
        next
          // wurde eine Kante aktiviert?
          choose e in enabledOutEdges with isActive(MI, n, e) do {
            SetCompletedAction(MI, n, edgeName(processIDFor(self), e))
          }
          ifnone {
            // keine ausgehende Kante aktiviert, Benutzerwahl notwendig
            SelectEdge(MI, n)
          }
      }
    }
  }
```

Listing 107: Cancel

Zuerst wird mit dem Aufruf der `CheckCancel` Regel die Freischaltung der ausgehenden Kanten aktualisiert.

Gibt es keine freigeschalteten Kanten dürfen weitere Knoten ausgeführt werden, insbesondere auch Knoten mit einer niedrigeren Priorität.

Wenn es genau eine freigeschaltete Kante gibt, welche die auto-Eigenschaft hat, wird diese automatisch aktiviert.

Gibt es eine aktivierte Kante wird die Knotenaktion beendet, um in der Kantenaktion die entsprechende Aktion abzubrechen.

Falls es keine Kante gibt die aktiviert wurde muss in der Konsolenanwendung eine gewählt werden. Es können zwar weitere Knoten ausgeführt werden, nicht jedoch mit geringerer Priorität.

```
rule CheckCancel(MI, n, e) = {
  let processID = processIDFor(self) in
  let eName = edgeName(processID, e) in
  let nCancel = nodeID_Named(processID, eName) in {
    // gibt es einen aktiven Knoten mit der Knoten-ID der Kantenbeschriftung?
    if (contains(activeNodes(channelFor(self), MI), nCancel) = true) then {
      EnableEdge(MI, e)
    }
    else {
      DisableEdge(MI, e)
    }
  }
}
```

Listing 108: CheckCancel

Mittels der `CheckCancel` Regel wird überprüft, ob es zu der jeweiligen ausgehenden Kante der Cancel Aktion einen entsprechenden aktiven Knoten gibt. In dem Fall wird die Kante aktiviert, andernfalls deaktiviert.

```
rule PerformEdgeCancel(MI, n, e) = {
  let eName = edgeName(processIDFor(self), e) in
  let nCancel = nodeID_Named(processIDFor(self), eName) in {
    // aktiviere den Abbruch des aktiven Knotens mit der Knoten-ID der Kantenbeschriftung
    cancelDecision(channelFor(self), MI, nCancel) := true
    SetCompletedEdge(MI, n, e)
  }
}
```

Listing 109: ModalJoin

Die Kantenaktion aktiviert den Abbruch des zugehörigen Knotens.

6.11 CallMacro

Die CallMacro Aktion startet eine neue Instanz eines Makros und kontrolliert deren Ausführung.

Die Knotenaktion ruft dazu nach dem Anlegen der Makroinstanz die `MacroBehaviour` Regel für diese Instanz auf um sie auszuführen.

Bei einem Abbruch der CallMacro Aktion wird die zugehörige Makroinstanz abgebrochen. Der Abbruch der CallMacro Aktion ist erst abgeschlossen wenn die Instanz keine aktiven Knoten mehr hat.

Zur Verwaltung der aufgerufenen Makroinstanz werden zwei Funktionen verwendet:

```
// Channel * macroInstanceNumber * NodeID -> macroInstanceNumber
function callMacroChildInstance : LIST * NUMBER * NUMBER -> NUMBER

// Channel * macroInstanceNumber -> result
function macroTerminationResult : LIST * NUMBER -> ELEMENT
```

Listing 110: Definitionen CallMacro

In der `callMacroChildInstance` Funktion wird die Makroinstanznummer des aufgerufenen Makros hinterlegt. Die `macroTerminationResult` Funktion wird von der aufgerufenen Makroinstanz bei ihrer Terminierung beschrieben, um einerseits die CallMacro Aktion über die Terminierung zu informieren und andererseits um optional ein Ergebnis zurückzugeben, welches über die Aktivierung der ausgehenden Kante der CallMacro Aktion entscheidet.

Die Knotenaktion hat zwei Phasen: Zuerst wird die Makroinstanz angelegt; anschließend wird das Verhalten der Makroinstanz ausgeführt und ihr Ausführungsergebnis übertragen als Ausführungsergebnis der Knotenaktion:

```
rule CallMacro(MI, n, args) = {
  let childInstance = callMacroChildInstance(channelFor(self), MI, n) in {
    // wurde bereits eine Instanz angelegt?
    if (childInstance = undef) then {
      // eine Instanz des Makros anlegen
      let mIDNew = searchMacro(head(args)),
          MINew  = nextMacroInstanceNumber(channelFor(self)) in {
        nextMacroInstanceNumber(channelFor(self)) := MINew + 1
        callMacroChildInstance(channelFor(self), MI, n) := MINew

        activeNodes(channelFor(self), MINew) := []
        AddNode(MI, n, MINew, macroStartnode(processIDFor(self), mIDNew))

        seq
          macroID(channelFor(self), MINew) := mIDNew
        next
          if (|macroArguments(processIDFor(self), mIDNew)| > 0) then {
            InitializeMacroArguments(MI, n, mIDNew, MINew, tail(args))
          }

        SetExecutionState(MI, n, DONE)
      }
    }
    else {
      // die Makroinstanz ist vorhanden, siehe zweiten Teil
    }
  }
}
```

Listing 111: CallMacro, erster Teil

Anhand der `callMacroChildInstance` Funktion wird überprüft ob bereits eine Makroinstanz angelegt wurde. Ist das nicht der Fall wird `nextMacroInstanceNumber` inkrementiert und der aktuelle Wert in `callMacroChildInstance` gespeichert. Die aktiven Knoten der neuen Makroinstanz werden mit dem Startknoten des Makros vorbelegt. In der Funktion `macroID` wird die MakroID der Makroinstanz hinterlegt. Verwendet das aufgerufene Makro Argumente so werden diese geladen und für die neue Instanz kopiert:

```
rule InitializeMacroArguments(MI, n, mIDNew, MINew, macroArgumentsValues) = {
  local
    listres1 := macroArguments(processIDFor(self), mIDNew),
    listres2 := macroArgumentsValues in
  {
    while (|listres1| > 0) do {
      let varnameDst = head(listres1),
          varnameSrc = head(listres2) in
      let var = loadVar(MI, varnameSrc) in
      {
        SetVar(MINew, varnameDst, nth(var, 1), nth(var, 2))
      }
      listres1 := tail(listres1)
      listres2 := tail(listres2)
    }
  }
}
```

Listing 112: InitializeMacroArguments

Da die SetVar Regel auf die macroID Funktion zurückgreift um nachzuschlagen, ob die Variable makroinstanz-lokal ist, muss der Aufruf der InitializeMacroArguments Regel erfolgen, nachdem die MakroID gespeichert wurde. Zum Kopieren der Variablen werden alle Makroargumente in einer Schleife durchgegangen um die Quellvariable aus der Makroinstanz der CallMacro Aktion zu laden und in der neuen Makroinstanz zu speichern. Da die SetVar Regel die variableDefined Funktion verändert ist hier eine **while** Schleife mit inkrementellen ASM Schritten nötig.

In der zweiten Phase der CallMacro Aktion wird die zuvor angelegt Makroinstanz ausgeführt:

```
rule CallMacro(MI, n, args) = {
  let childInstance = callMacroChildInstance(channelFor(self), MI, n) in {
    // wurde bereits eine Instanz angelegt?
    if (childInstance = undef) then {
      // eine Instanz des Makros anlegen, siehe ersten Teil
    }
    else {
      // die Makroinstanz ist vorhanden
      let terminationResult = macroTerminationResult(channelFor(self), childInstance) in {
        // prüfen ob sie terminiert ist
        if (terminationResult != undef) then {
          // die Makroinstanz ist terminiert => Knotenaktion abschließen
          callMacroChildInstance(channelFor(self), MI, n) := undef
          if (terminationResult = true) then {
            SetCompletedAction(MI, n, undef)
          }
          else {
            SetCompletedAction(MI, n, terminationResult)
          }
        }
        else
          // die Makroinstanz ist noch nicht terminiert
          seq
            MacroBehaviour(childInstance)
          next
            let state = macroExecutionState(channelFor(self), childInstance) in {
              if (state = DONE) then {
                SetExecutionState(MI, n, DONE)
              }
              else if (state = NEXT) then {
                SetExecutionState(MI, n, NEXT)
              }
              else if (state = LOWER) then {
                SetExecutionState(MI, n, LOWER)
              }
            }
      }
    }
  }
}
```

Listing 113: CallMacro, zweiter Teil

Zuerst wird in der Funktion macroTerminationResult überprüft ob die angelegte Makroinstanz beendet wurde. Ist der Wert **true** so bedeutet dies, dass ein Endzustand ohne Beschriftung erreicht wurde. Wenn jedoch ein beschrifteter Endzustand erreicht wurde trägt die End Aktion diese Beschriftung in die macroTerminationResult Funktion ein, und der Wert wird für die SetCompletedAction Regel als Argument verwendet, um anzugeben dass die CallMacro Aktion über die Kante mit der gleichen Beschriftung verlassen werden muss. Die Funktion callMacroChildInstance wird zurückgesetzt um die CallMacro Aktion in einer Schleife ausführen zu können.

Wenn die Makroinstanz noch nicht beendet wurde wird die MacroBehaviour Regel für die Instanz aufgerufen und das Ausführungsergebnis aus der macroExecutionState Funktion für den eigenen Ausführungszustand übernommen.

Werden innerhalb der aufgerufenen Makroinstanz Knoten mit einer niedrigeren Priorität erlaubt so überträgt sich dies auf die CallMacro Aktion. Der Abbruch der CallMacro Aktion bewirkt einen Abbruch der Makroinstanz:

```
rule AbortCallMacro(MI, n) = {
  let childInstance = callMacroChildInstance(channelFor(self), MI, n) in {
    if (|activeNodes(channelFor(self), childInstance)| > 0) then {
      AbortMacroInstance(childInstance, undef)
      SetExecutionState(MI, n, DONE)
    }
    else {
      callMacroChildInstance(channelFor(self), MI, n) := undef
      SetAbortionCompleted(MI, n)
    }
  }
}
```

Listing 114: AbortCallMacro

Zum Abbruch der angelegten Makroinstanz wird überprüft ob in dieser noch aktive Knoten vorhanden sind. In dem Fall wird die `AbortMacroInstance` Regel aufgerufen, welche den Abbruch der Knoten einleitet. Sind keine Knoten mehr aktiv ist der Abbruch angeschlossen und die Funktion `callMacroChildInstance` wird wie nach dem Abschluss der Knotenaktion zurückgesetzt.

Die `AbortMacroInstance` Regel leitet den Abbruch aller Knoten einer Makroinstanz ein. Wird der Regel ein optionaler `ignoreNode` Parameter übergeben so wird dieser Knoten nicht abgebrochen - dies wird später von der End Aktion verwendet, um sich nicht selbst abzubrechen.

```
rule AbortMacroInstance(MIAbort, ignoreNode) = {
  foreach currentNode in activeNodes(channelFor(self), MIAbort) do {
    if (currentNode != ignoreNode) then {
      add [MIAbort, currentNode] to killNodes(channelFor(self))
    }
  }
  ClearAllVarOfMI(MIAbort) // alle Variablen dieser Makroinstanz zurücksetzen
}
```

Listing 115: AbortMacroInstance

Zur Einleitung der Knotenabbrüche werden alle aktiven Knoten der übergebenen Makroinstanz in der Funktion `killNodes` hinterlegt, damit diese zu Beginn des nächsten ASM Schrittes beim Aufruf der `SubjectBehaviour` Regel sofort beendet werden, da diese vor dem Aufruf der `MacroBehaviour` Regel des Mainmakros alle `killNodes` mittels der `KillBehaviour` Regel abbricht. Außerdem werden alle Variablen der aufgerufenen Makroinstanz entfernt.

6.12 End

Mit der End Aktion wird eine Makroinstanz beendet. Wird dabei das Mainmakro beendet so terminiert das Subjekt.

```
rule StartEnd(MI, n) = {
  // direkt mit PerformEnd fortfahren, globalen ASM Schritt vermeiden
  SetExecutionState(MI, n, REPEAT)
}
```

Listing 116: StartEnd

```
rule PerformEnd(MI, n) = {
  if (|activeNodes(channelFor(self), MI)| > 1) then {
    AbortMacroInstance(MI, n)
    SetExecutionState(MI, n, DONE)
  }
  else {
    // terminiert das Mainmakro?
    if (MI = 1) then {
      ClearAllVarOfMI(0) // alle globalen Variablen zurücksetzen
      ClearAllVarOfMI(1) // alle Variablen des Mainmakros zurücksetzen
      let proper = inputPoolIsEmpty(channelFor(self), undef, undef, undef) in {
        properTerminated(channelFor(self)) := proper
      }
      program(self) := undef
      remove self from asmAgents
    }
    else {
      ClearAllVarOfMI(MI) // alle Variablen dieser Makroinstanz zurücksetzen
      let res = head(nodeActionArguments(processIDFor(self), n)) in {
        if (res = undef) then
          macroTerminationResult(channelFor(self), MI) := true
        else
          macroTerminationResult(channelFor(self), MI) := res
      }
    }
    RemoveNode(MI, n, MI, n)
    SetExecutionState(MI, n, DONE)
  }
}
```

Listing 117: AbortMacroInstance

Zum Beenden der Makroinstanz wird zuerst überprüft ob neben dem Knoten dieser End Aktion weitere Knoten aktiv sind. In dem Fall wird analog zum Abbruch der CallMacro Aktion die `AbortMacroInstance` Regel aufgerufen welche den Abbruch aller anderen Knoten einleitet.

Sind keine anderen Knoten aktiv wird unterschieden ob es sich bei dem Makro um das Mainmakro handelt. In dem Fall terminiert das Subjekt und die Ausführung des ASM Agenten. Mit der `inputPoolIsEmpty` Funktion wird überprüft ob sich noch Nachrichten oder Reservierungen in einer Warteschlange befinden und entsprechend in der `properTerminated` Funktion vermerkt ob das Subjekt proper terminated ist.

Falls sich die End Aktion dagegen in einem aufgerufenen Makro befindet wird in der `macroTerminationResult` Funktion für die aufrufende CallMacro Aktion hinterlegt dass die Makroinstanz nun beendet ist. Wenn die End Aktion mit einem Ergebnis beschriftet ist wird diese Beschriftung hinterlegt, ansonsten nur `true`. Beim Beenden der Makroinstanz werden alle Variablen der Instanz zurückgesetzt, beim Beenden des Mainmakros zusätzlich alle globalen Variablen.

In diesem Kapitel wurde die Spezifikation der Ausführungseinheit mit einer Ausführungssemantik aller Aktionstypen vervollständigt. Es wurde die Spezifikation aus [12] als Grundlage genommen, nach CoreASM übertragen und an das Strukturmodell des vierten Kapitels angepasst und erweitert.

Die Einführung der aktiven Knoten auf Ebene einer Makroinstanz hat dazu geführt, dass der Aufruf der Regeln zum Abbruch einer Knotenaktion in die `Behaviour` Regel verschoben wurde. Die Regeln zum Senden und Empfangen von Nachrichten, sowie die Internal Action, konnten vom Ansatz her übernommen werden, es mussten jedoch weitere Regeln ergänzt werden, damit sie ausführbar sind und mit der Konsolenanwendung verknüpft werden konnten. Die Regeln zur Variablenmanipulation wurden so ausgearbeitet, dass sie ausführbar sind. Die Regeln zum Öffnen und Schließen von Warteschlangen des Inputpools wurden angepasst. Die Spezifikationen der Select Agents, Modal Split, Modal Join, Cancel, CallMacro und End Aktionen wurden komplett neu entworfen.

Im nächsten Kapitel wird die in diesem und vorherigem Kapitel entworfene Spezifikation auf Fehlerfreiheit untersucht und die Verwendung aller S-PM Sprachelemente demonstriert.

7 Machbarkeitsnachweis

In den beiden vorherigen Kapiteln 5 und 6 wurde eine Ausführungssemantik für das in Kapitel 4 vorgestellte S-PM Strukturmodell spezifiziert. Zum Entwurf der Spezifikation wurden Abstrakte Zustandsmaschinen (ASM) verwendet, dabei wurde der CoreASM Dialekt genutzt, um eine Ausführung der Spezifikation mit dem CoreASM Interpreter zu ermöglichen. Zur Unterstützung einer interaktiven Prozessvalidierung und einer Testumgebung wurde bereits eine Schnittstelle für Ein- und Ausgaben definiert.

In diesem Kapitel wird die Spezifikation systematisch auf Vollständigkeit und Fehlerfreiheit untersucht. Zur automatisierten Ausführung von Komponenten- und Integrationstests wurde eine Testumgebung auf Basis des CoreASM TestEngineDrivers entwickelt. Die erfolgreiche Ausführung von drei Integrationstests im Scala Build-Management Tool ist in Abbildung 7.1 zu sehen. Die Systemtests erfolgten durch eine abstrake Prozessausführung, die mit einer in Scala entwickelten Konsolenanwendung interaktiv gesteuert werden kann, wie in Abbildung 7.2 gezeigt wird.

In den Komponententests wird die Ausführung einzelner Aktionen in vordefinierten Situationen überprüft. Für jeden Testfall werden die Inhalte der Inputpools und die Belegung von Variablen definiert, der Testfall wird akzeptiert, wenn die entsprechende Aktion über eine bestimmte Kante verlassen wurde. Beispielsweise muss eine Send Aktion über eine Timeoutkante verlassen werden, wenn der Inputpool des Empfängers gefüllt ist.

In den Integrationstests wird die Ausführung ganzer Prozessmodelle überprüft, welche durch die Verwendung von automatischen Kanten und Timeoutkanten ohne Benutzerinteraktionen lauffähig sind. Die Testumgebung der Integrationstests wurde so gestaltet, dass sie einen Testfall akzeptiert, wenn die abstrakte Ausführung des Prozessmodells in einer vorgegebenen Anzahl an ASM Schritten in einer End Aktion ohne Knotenbeschriftung terminiert. Der Testfall wird als ungültig erkannt, wenn ein Subjekt mit einer End Aktion terminiert, der als Knotenbeschriftung eine Fehlermeldung angegeben wurde.

Zur Sicherstellung der Vollständigkeit der Spezifikation und dem korrekten Zusammenspiel mit der Konsolenanwendung werden Beispielprozesse mit der Konsolenanwendung ausgeführt. Neben Prozessmodellen der S-BPM Community wird auf Prozessmodelle aus dem PolyEnergyNet Forschungsprojekt (PEN) [15] zurückgegriffen. Mit der Verwendung der S-PM Methodik wurden in dem PEN Projekt Energietechniker und Softwareentwickler zur Erforschung und exemplarischen Realisierung von resilienten Ortsnetzen (Holonen) zusammengebracht. Ein Überblick des dort entworfenen Prozessmodells mit 17 Subjekten befindet sich am Ende dieses Kapitels in Abbildung 7.9.

```
sbt:root> asm/testOnly de.tkip.asm.test.semantic.TestTransitions2Tests  -- -z GraphML
[info] TestTransitions2Tests:
[info] - TestMacroArgumentsGraphML.casm
[info] - TestModalSplitLoopGraphML.casm
[info] - TestSendMinMaxGraphML.casm
[info] Run completed in 16 seconds, 140 milliseconds.
[info] Total number of tests run: 3
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 3, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

Abbildung 7.1: Testumgebung

```
> process load processes/ba-thesis/Zustandsabfrage.graphml

> process start Zustandsabfrage.graphml

Available actions:

 1) Select Agent for Subject 'Holoner' in Process 'Zustandsabfrage.graphml'
 (PI 1) and start execution


> 1

Agent name for 'Holoner': Holoner1

Available actions:

Process 'Zustandsabfrage.graphml' (PI 1)
 Subject 'Holoner' executed by Agent 'Holoner1'
  Macro 'Zustandsabfrage' (MI 2)
   Node 'Sende' (send)
    1) Select 2 to 4 Agents for Subject 'Holonkoordinator'


> 1

Agent name (1/4) for 'Holonkoordinator': HK1

Agent name (2/4) for 'Holonkoordinator': HK2

Agent name (3/4) for 'Holonkoordinator' (leave empty to use only 2 agents):


Available actions:

Process 'Zustandsabfrage.graphml' (PI 1)
 Subject 'Holoner' executed by Agent 'Holoner1'
  Macro 'Zustandsabfrage' (MI 2)
   Node 'Sende' (send)
    1) Set Message Content ("HZ-Abfrage") for Message to {Channel(Zustandsa
bfrage.graphml,1,Holonkoordinator,HK1), Channel(Zustandsabfrage.graphml,1,H
olonkoordinator,HK2)}


> 1

Receivers: {Channel(Zustandsabfrage.graphml,1,Holonkoordinator,HK1), Channe
l(Zustandsabfrage.graphml,1,Holonkoordinator,HK2)}
Message Content ("HZ-Abfrage"): Abfrage1
```

Abbildung 7.2: Konsolenanwendung: Anfang der abstrakten Ausführung des Prozessmodells „Zustandsabfrage“

Das Prozessmodell in Abbildung 7.3 ist ein Integrationstest und besteht aus einem einzigen Subjekt „A“, welches Nachrichten in zwei parallelen Ausführungspfaden an sich selbst sendet und empfängt. Der Testfall wurde entworfen um das korrekte Verhalten der Modal Split und Modal Join Aktionen zu erfassen. Mit dem Datenobjekt wird für die Testumgebung hinterlegt, dass das Subjekt „A“ von dem Agenten „a“ ausgeführt werden soll. Damit wird die Agentenwahl in der Konsolenanwendung vermieden.

Zuerst werden zwei „executeLoop“ Nachrichten im Inputpool hinterlegt, welche als Zähler für eine Schleifenbedingung genutzt werden. In den ersten beiden Durchgängen empfängt die „loop“ Aktion diese Nachrichten und der Schleifenkörper wird ausgeführt. Der dritte Versuch eine „executeLoop“ Nachricht zu empfangen schlägt fehl und führt über die Timeoutkante direkt in den Endzustand. Der Prozess wird nun als fehlerfrei erkannt.

Im Schleifenkörper werden durch die Modal Split Aktion zwei Ausführungspfade aktiviert. Der rechte Pfad wird durch das Empfangen einer „unblock“ Nachricht blockiert. Im linken Ausführungspfad wird ebendiese Nachricht gesendet. Am Ende des Schleifenkörpers wird nach der Zusammenführung der Ausführungspfade überprüft ob sich eine „unblock“ Nachricht im Inputpool befindet, was zu dem Fehler „unblock received after Join“ führen würde.

Das Erreichen des End-Zustandes mit der Fehlermeldung kann auf einen Fehler in der Spezifikation der Modal Join Aktion deuten, falls diese die Ausführungspfade inkorrekt zusammenführt oder nicht auf das Beenden aller Pfade wartet. Die Aktionen zur Zusammenführung der Ausführungspfade und zur Überprüfung ob sich eine „unblock“ Nachricht noch im Inputpool befindet werden mit einer höheren Priorität ausgeführt, um in einem solchen Fehlerfall deren Ausführung anstelle der „block“-Aktion des rechten Pfades zu erzwingen. Wenn die Ausführungssemantik der Modal Join Aktion korrekt ist haben die Knotenprioritäten keine Auswirkung, da an dieser Stelle keine weiteren Knoten aktiv sein können.

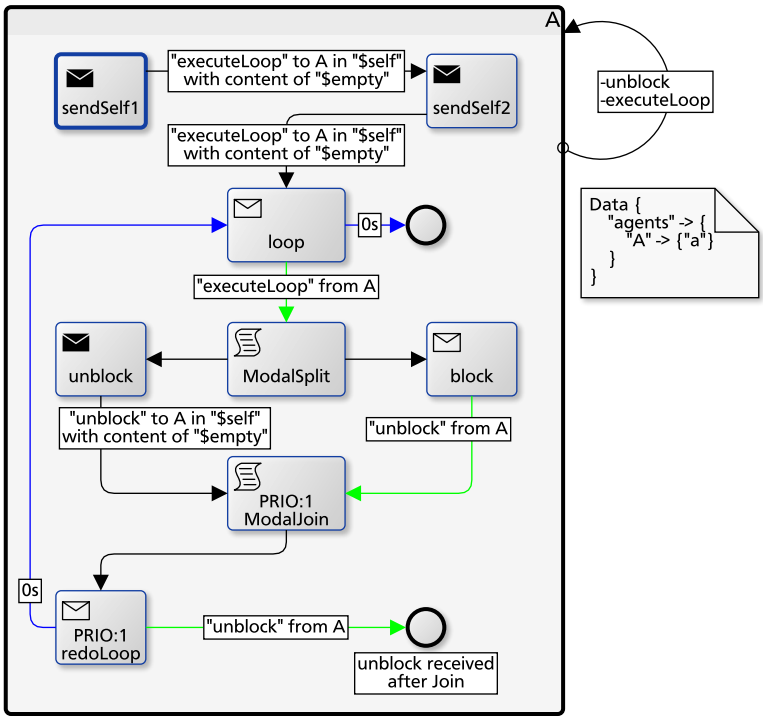


Abbildung 7.3: Integrationstest: Modal Split und Modal Join, zur Sicherstellung der Ausführbarkeit in einer Schleife.

```
// Channel * MacroInstanceNumber * joinNode -> Number
function joinCount : LIST * NUMBER * NUMBER -> NUMBER

rule ModalJoin(MI, n, args) =
  // ...
    // sind alle Ausführungspfade zusammengeführt?
    if (joinCount(channelFor(self), MI, n) >= splitCount)
    then { // Zähler zurücksetzen und normal fortfahren
      joinCount(channelFor(self), MI, n) := undef
      SetCompletedAction(MI, n, undef)
    }
  // ...
```

Listing 118: ModalJoin Regel (Ausschnitt)

Durch die zweifache Ausführung des Schleifenkörpers wird zudem sichergestellt dass die Modal Join Aktion mehrfach ausführbar ist, indem sie nach dem Abschluss der Knotenaktion ihre Zwischenwerte zurücksetzt.

Liegt in der Spezifikation der Modal Join Aktion jedoch ein Fehler vor, beispielsweise wenn die `joinCount` Funktion beim Abschluss der Aktion nicht zurückgesetzt werden würde, dann würde die Aktion bei dem zweiten Durchgang direkt beim Eintreffen des ersten Ausführungspfades beendet werden ohne auf den zweiten Ausführungspfad zu warten. Dieser Fehler wird durch das Prozessmodell in Abbildung 7.3 entdeckt, indem die „redoLoop“ Aktion durch ihre höhere Priorität anstatt der „block“ Aktion ausgeführt wird und nun die „unblock“ Nachricht empfängt.

7.2 Zustandsabfrage

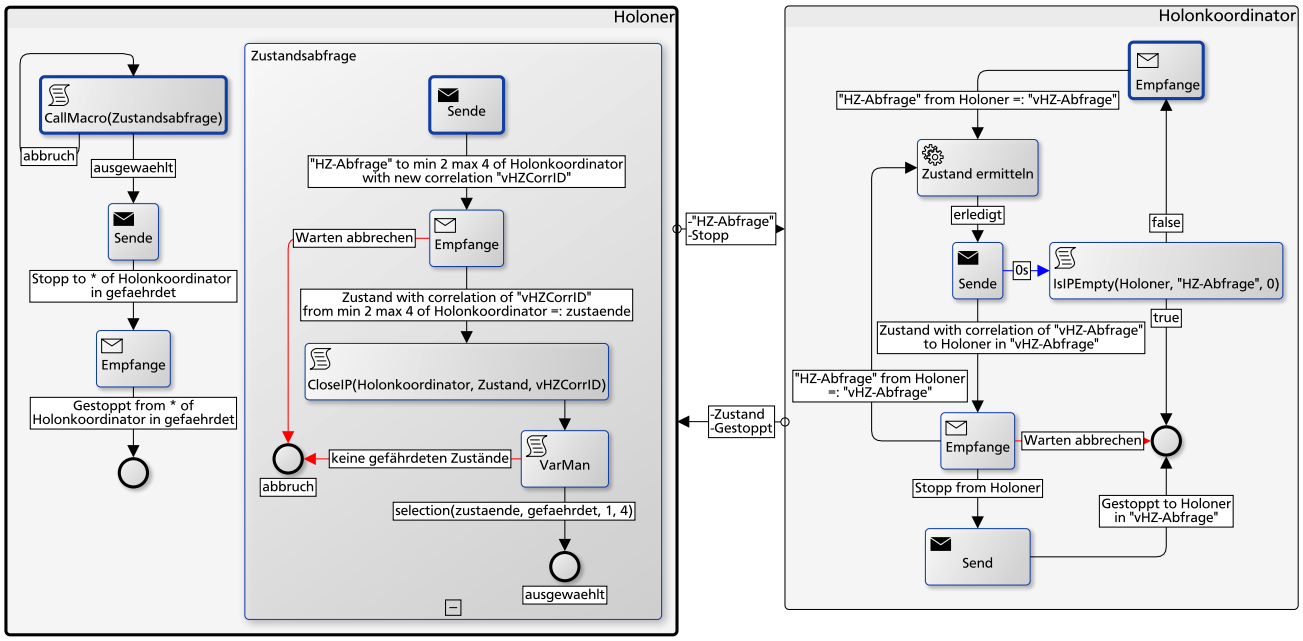


Abbildung 7.4: Zustandsabfrage, aus [16]. Kommunikation mit dem Multisubjekt „Holonkoordinator“. Beispiel zu CorrelationIDs, Inputpool Aktionen, Auswahl der Teilmenge einer Variablen und mehrerer Endzustände eines Makros.

Der Prozess in Abbildung 7.4 ist ein vereinfachtes Beispiel aus dem PEN-Projekt. Dort werden zur Bildung von resilienten Ortsnetzen Energieerzeuger und -verbraucher in Holone gruppiert. Jedes Holon hat einen Holonkoordinator, das Subjekt „Holonkoordinator“ wird daher als Multisubjekt verwendet. Bei Änderungen oder Störungen im Energienetz werden die Holone je nach Bedarf umgebildet oder notfalls gestoppt. Dazu findet eine regelmäßige Zustandsabfrage aller Holonkoordinatoren statt. Der Start und Stopp der Holonkoordinatoren wird durch das Singlesubjekt „Holoner“ koordiniert.

Der Ablauf der Zustandsabfrage ist als Makro modelliert und wird dauerhaft aufgerufen. Für jede Abfrage wird eine neue CorrelationID erzeugt, damit sichergestellt ist, dass sich die Antworten immer auf die aktuellste Abfrage beziehen. Nach dem Empfangen der Zustände wird die Warteschlange der aktuellen CorrelationID geschlossen, um verspätete Antworten zu verhindern. Versucht ein Holonkoordinator, nachdem der Holoner die Warteschlange zu einer veralteten Abfrage geschlossen hat, seinen Zustand zu senden blockiert daher seine Send Aktion. Sie wird nun automatisch über die Timeoutkante verlassen. Im Anschluss prüft der Holonkoordinator, ob eine neuere Nachricht zur Zustandsabfrage vorhanden ist, um entweder diese zu empfangen oder um sein Verhalten zu beenden, um erst beim Erhalt der nächsten Abfrage wieder aktiv zu werden.

Der Holoner überprüft nun die empfangenen Antworten auf gefährdete Zustände. Wurden gefährdete Zustände festgestellt und ausgewählt werden den Holonkoordinatoren mit einem gefährdetem Zustand Stopp-Nachrichten gesendet, und es wird auf die Bestätigung des Stopps gewartet. Der Prozess ist damit abgeschlossen.

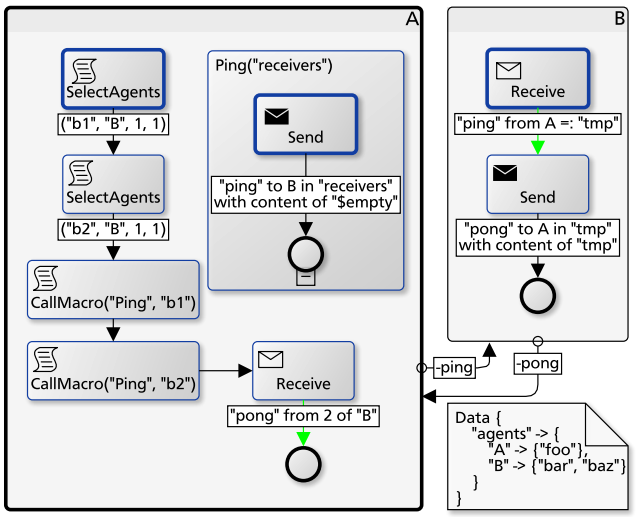


Abbildung 7.5: Integrationstest: CallMacro mit Parametern, aus [16]. A sendet nacheinander an unterschiedliche Agenten des Multisubjekts B.

```
Process Transitions {
Data {
  //..
  "A TestSendTimeoutC BeforeInit" -> [
    ["StoreMessage", "$A", "$B",
       "sendTimeoutC", ["Text", "pre1"], 1],
    ["StoreMessage", "$A", "$B",
       "sendTimeoutC", ["Text", "pre2"], 2]
  ],
  //..
}
Subject A {
  StartSubject := true
  Macro Main {
    START: Action -> END
    //..
    TestSendTimeoutC: Send {
      ["sendTimeoutC" to B in "channelB"
        with content of "$empty"]
          -> TestSendTimeoutCFail
      [timeout (0)]
          -> TestSendTimeoutCSuccess
    }
    //..
  }
}
Subject B {
  InputPool := 2
  //..
}}
```

Listing 119: Komponententests (Ausschnitt)

7.3 CallMacro mit Parametern

Der Prozess in Abbildung 7.5 ist ein weiterer Integrationstest. Er besteht aus einem Subjekt „A“ und einem Multisubjekt „B“. In diesem Testfall werden Makro-Argumente getestet, indem ein Makro mit zwei unterschiedlichen Variablen mit Empfängern aufgerufen wird. Für die Testumgebung ist im Datenobjekt hinterlegt, dass das Subjekt „A“ durch den Agenten „foo“ ausgeführt werden soll. Für das Subjekt „B“ werden zwei Agenten „bar“ und „baz“ hinterlegt, so dass die Testumgebung diese automatisch bei den Select Agents Aktionen auswählt, ohne dass eine Benutzerinteraktion nötig wäre. Damit ist sichergestellt, dass sich in den Variablen „b1“ und „b2“ Channel mit unterschiedlichen Agenten befinden.
Der Test in diesem Prozess besteht darin, dass die Receive Aktion zwei „pong“ Nachrichten von unterschiedlichen Absendern des Multisubjektes „B“ empfangen muss. Das ist dann der Fall, wenn die Inhalte der Variablen „b1“ und „b2“ korrekt in die Variable „receivers“ des „Ping“ Makros kopiert wurden. Das Subjekt „B“ empfängt automatisch die „ping“ Nachricht und reagiert sofort mit einer „pong“ Nachricht. Würden beide Nachrichten von „A“ zum gleichen Agenten des Subjektes „B“ gesendet werden, dann würden auch beide Antworten von diesem Agenten im Inputpool von „A“ ankommen. Die Receive Aktion ist jedoch so definiert, dass „from 2“ nur Nachrichten von unterschiedlichen Absendern empfängt.

7.4 Komponententests

Für die Komponententests wird eine textuelle Prozessbeschreibung verwendet. Diese Testumgebung überprüft einzelne Aktionen drauf, dass deren Ausführung fehlerfrei abläuft und die Aktion über die richtige ausgehende Kante verlassen wird. `"$A"` und `"$B"` werden von der Testumgebung durch die Channel der jeweiligen Agenten ersetzt.
In Listing 118 wird ein Ausschnitt der Komponententests gezeigt, mit dem das Senden an einen Empfänger mit bereits gefülltem Inputpool überprüft wird. Zuerst werden zwei Nachrichten des Subjekts „A“ in den Inputpool des Subjekts „B“ gelegt, die den Nachrichtentyp „sendTimeoutC“ haben. Die erste Nachricht hat den Inhalt „pre1“ mit der CorrelationID 1 und die zweite Nachricht hat den Inhalt „pre2“ mit der CorrelationID 2.
Da das Subjekt B ein IP-Limit von 2 hat ist die Warteschlange nun gefüllt. Die Ausführung der „TestSendTimeoutC“ Aktion kann keine weitere „sendTimeoutC“ senden und wird wie erfordert über die Timeoutkante verlassen.

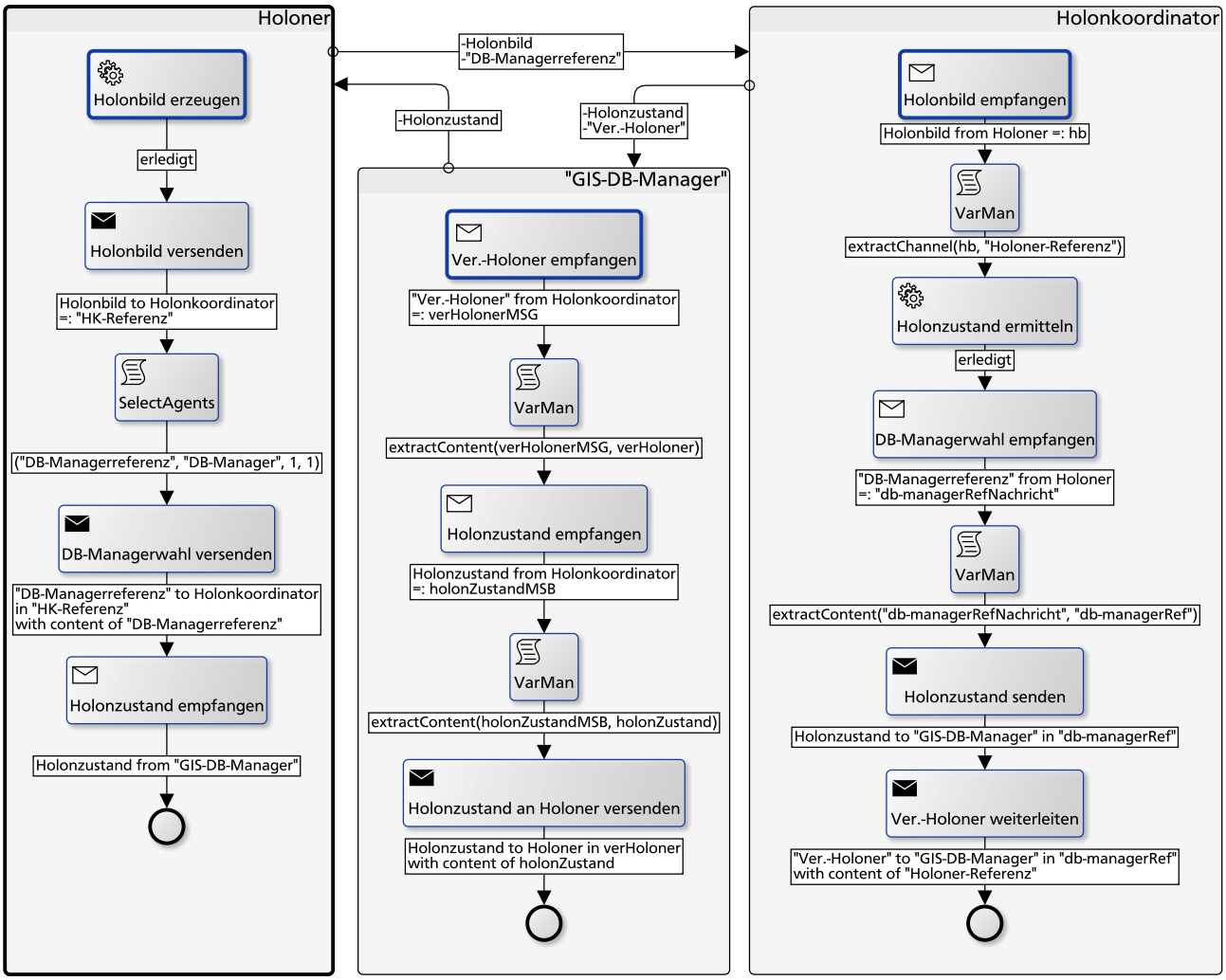


Abbildung 7.6: Mobility of Channels Beispiel, aus [16]. Ein Channel zum GIS-DB-Manager wird vom Holoner bestimmt und an den Holonkoordinator gesendet.

In dem Prozess der Abbildung 7.6 wird eine Dreieckkommunikation mit dem Mobility of Channels Konzept gezeigt. Auch dieser Prozess stammt aus dem PEN-Projekt und wurde vereinfacht; beispielsweise wird hier darauf verzichtet den Holonkoordinator als Multisubjekt zu modellieren, um die Vorauswahl des Agenten für das GIS-DB-Manager-Subjekt besser darzustellen. In diesem Prozess wird ebenfalls der Zustand des Holons abgefragt, diesmal jedoch über ein drittes Subjekt zur Zwischenspeicherung gesendet.

Am Anfang bestimmt das Holoner-Subjekt das aktuelle Holonbild und sendet es an den Holonkoordinator, welcher zuerst die Referenz zum Holoner in der Variablen „Holoner-Referenz“ speichert und dann den eigenen Holonzustand ermittelt. Währenddessen wählt der Holoner einen Agenten für das GIS-DB-Manager Subjekt aus und sendet den erzeugten Channel ebenfalls an den Holonkoordinator.

Nach der Ermittlung des Holonzustandes empfängt der Holonkoordinator die Nachricht mit dem Channel zum GIS-DB-Manager, entpackt diesen in die Variable „db-managerRef“ und sendet diesem seinen aktuellen Holonzustand. Zum Abschluss sendet der Holonkoordinator den Channel des Holoners an den GIS-DB-Manager, damit dieser den Holonzustand weiterleiten kann.

Der GIS-DB-Manager empfängt zuerst die Nachricht mit dem Channel des verantwortlichen Holoners und entpackt diesen in die Variable „verHoloner“. Dann wird die Nachricht mit dem Holonzustand empfangen und ihr Inhalt in die Variable „holonZustand“ entpackt, damit diese an den Holoner gesendet werden kann. Schlussendlich empfängt der Holoner den Holonzustand und der Prozess ist abgeschlossen. Hierbei ist zu sehen, dass Nachrichten mit einem unterschiedlichem Nachrichtentyp in einer anderen Reihenfolge empfangen werden können als sie gesendet wurden.

7.6 Externe Prozesse und IP-Limit

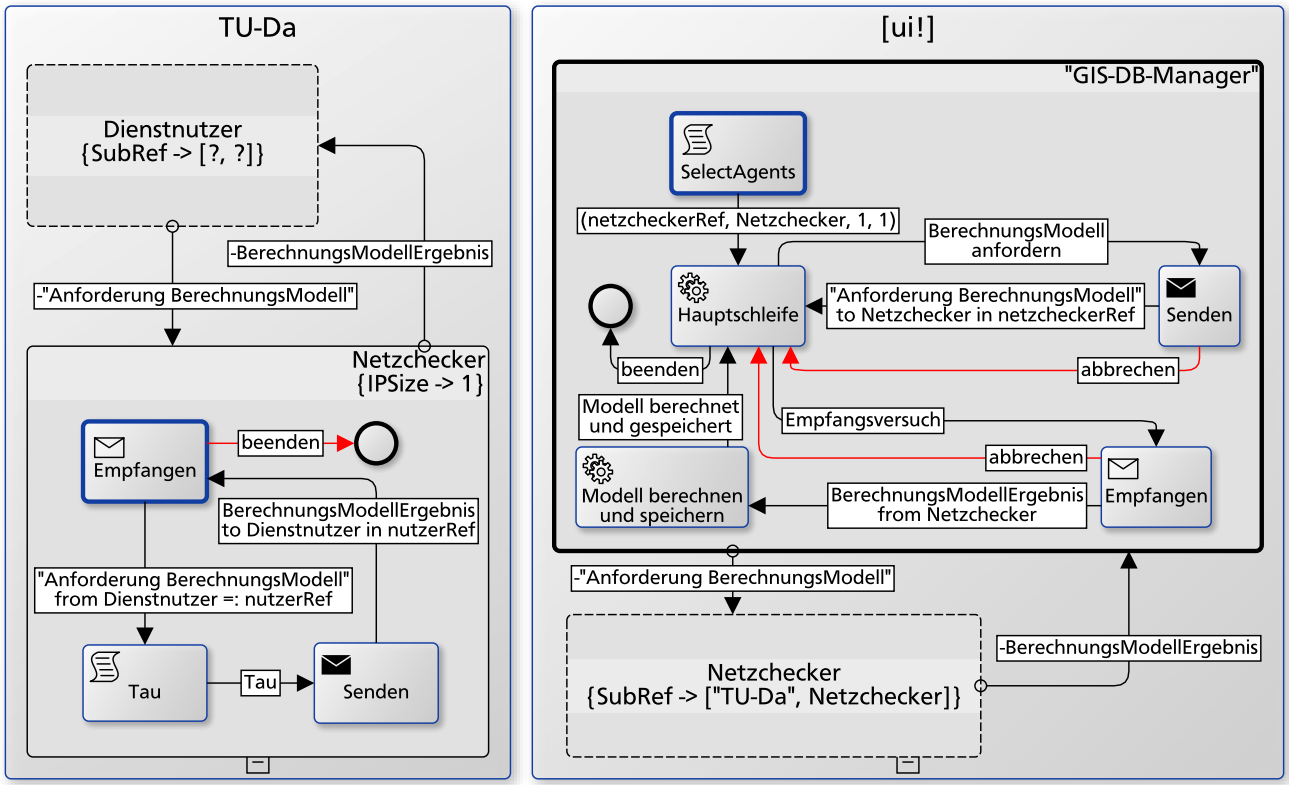


Abbildung 7.7: Extern und IP-Limit, aus [16]. Durch die Limitierung des Inputpools kann der Service Prozess nur eine Anfrage puffern.

In dem Prozessnetzwerk der Abbildung 7.7 wird die Einbindung externer Prozesse, sowie die Limitierung des Inputpools, dargestellt. Zu sehen sind zwei Prozessmodelle unterschiedlicher Organisationen: „TU-Da“ der TU Darmstadt auf der linken Seite und „[ui!]“ der Urban Software Institute GmbH auf der rechten Seite. Die Prozessmodelle sind durch Interfacesubjekte, die am getrichelten Rahmen erkennbar sind, verbunden.

Der „TU-Da“ Prozess wurde als Service Prozess entworfen, so dass er mit beliebigen anderen externen Prozessen verknüpft werden kann. Dies ist daran zu erkennen, dass die Referenz des Interfacesubjekts „Diestnutzer“ offen gelassen wurde. Der Inputpool des internen Subjekts „Netzchecker“ ist auf eine Nachricht je Warteschlange beschränkt, um einen Stau von Anfragen zu verhindern. Der angebotene Dienst besteht aus der Erzeugung von Berechnungsmodellen; da der tatsächliche Vorgang für die Betrachtung von außen nicht relevant ist wird die Tau Aktion als Platzhalter des eigentlichen Verhaltens verwendet.

Im „[ui!]“ Prozess wird der Service Prozess durch das Interfacesubjekt „Netzchecker“ eingebunden, welches auf das externe Subjekt „Netzchecker“ im Prozess „TU-Da“ verweist. Das Subjekt „GIS-DB-Manager“ ist das Startsubjekt und beginnt mit der Auswahl eines Agenten für das Netzchecker Subjekt. In der Hauptschleife stehen drei Aktionen zur Verfügung: es kann ein neues Berechnungsmodell angefordert werden, es kann ein Berechnungsmodell empfangen werden und das Verhalten kann beendet werden. Durch diese Schleife ist es möglich mehrere Berechnungsmodelle anzufordern, das Senden wird jedoch blockiert wenn sich bereits eine Nachricht im Inputpool des Netzcheckers befindet, die noch nicht abgerufen wurde. In dem Fall kann das Senden abgebrochen werden. Wird in der Hauptschleife gewählt, dass versucht werden soll ein Berechnungsmodell zu empfangen, kann es dazu kommen, dass noch keine Nachricht vorliegt, in dem Fall wird auf eine Nachricht gewartet. Alternativ kann auch hier durch einen Abbruch zur Hauptschleife zurückgekehrt werden. Liegt ein Berechnungsmodell vor, so kann es empfangen und im nächsten Schritt in der GIS Datenbank gespeichert werden.

7.7 Überwachter Systemstart

Der Prozess in Abbildung 7.8 zeigt die Verwendung von Kanten- und Knotenprioritäten, sowie der Cancel Aktion in Kombination mit versteckten Kanten.

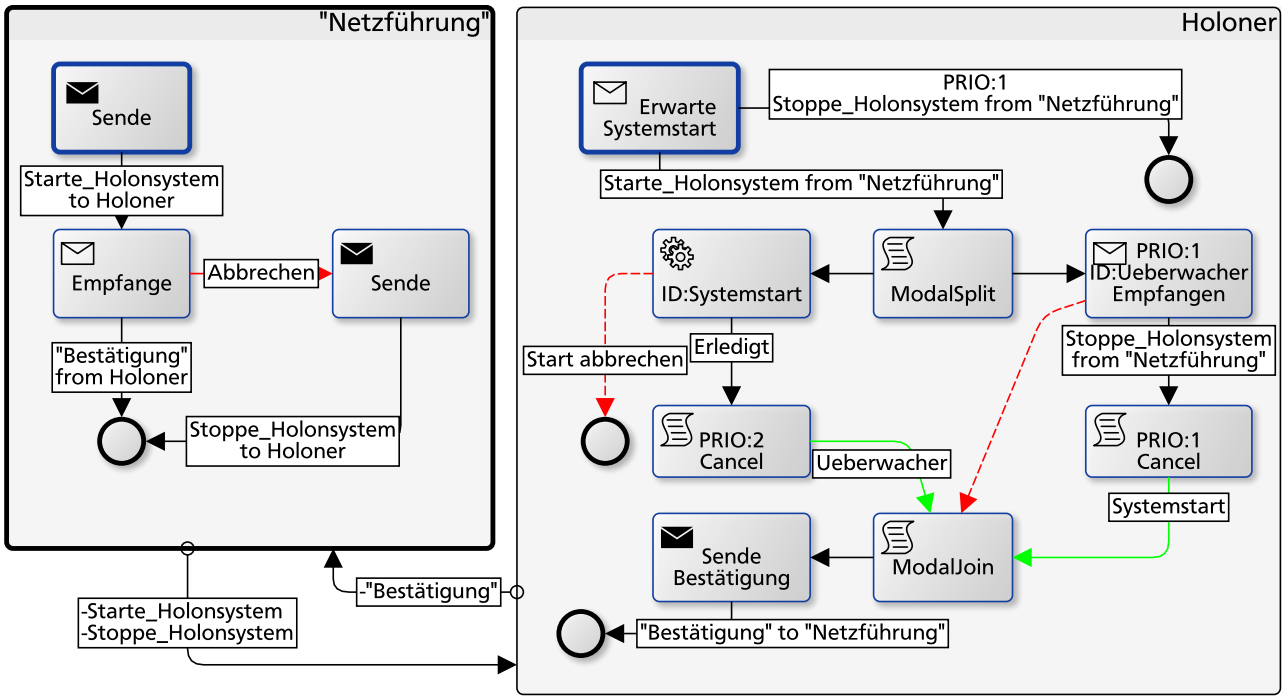


Abbildung 7.8: Systemstart, aus [16]. Erzwungener Abbruch der Internal Action „Systemstart“ bei Erhalt der Nachricht „Stoppe_Holonsystem“.

Das Netzführungssubjekt gibt dem Holoner-Subjekt die Anweisung einen Systemstart durchzuführen. Das Starten eines Holonsystems ist eine länger dauernde Aktion, die hier vereinfacht als Internal Action mit der ID „Systemstart“ modelliert ist. Während der Durchführung des Systemstarts muss der Holoner auf eine „Stoppe_Holonsystem“ Nachrichten reagieren können, um den Systemstart abzubrechen.

Falls das Holoner-Subjekt die Startnachricht beim Eintreffen der Stoppnachricht noch nicht abgerufen hat soll kein Systemstart durchgeführt werden. Dazu hat die „Erwarte Systemstart“ Aktion zwei ausgehende Kanten, wobei die „Stoppe_Holonsystem“ Kante eine höhere Priorität hat. Durch diese Kantenpriorität ist es nicht möglich eine „Starte_Holonsystem“ Nachricht zu empfangen sobald eine „Stoppe_Holonsystem“ Nachricht vorliegt.

Um gleichzeitig den Systemstart durchzuführen und auf eine Nachricht zu reagieren werden mit der Modal Split Aktion zwei parallele Ausführungspfade gestartet. Im linken Pfad wird der Systemstart durchgeführt, während im rechten Pfad mit der Receive Aktion auf eine „Stoppe_Holonsystem“ Nachricht gewartet wird. Die Receive Aktion hat eine höhere Knotenpriorität und wird somit von der `MacroBehaviour` Regel immer zuerst ausgewertet. Sobald eine Nachricht vorliegt wird der Systemstart pausiert, die Cancel-Aktion bricht nun automatisch den Systemstart ab und das Holoner-Subjekt terminiert. Da der Abbruch nicht durch den Agenten ohne vorliegen einer Stoppnachricht abgebrochen werden darf ist die „Start abbrechen“ Kante versteckt. Ebenso ist die Cancel-Kante der Receive Aktion des rechten Pfades versteckt, damit die Überwachung nicht vorzeitig vom Agenten beendet wird.

Wenn keine Stoppnachricht eingetroffen ist und der Systemstart durchgeführt wurde muss die Überwachung beendet werden. Dazu bricht die Cancel Aktion des linken Pfades den „Ueberwacher“ Knoten ab. Beide Ausführungspfade vereinigen sich in der Modal Join Aktion, um im Anschluss die Bestätigung des Systemstarts zu senden.

Mit den in diesem Kapitel untersuchten Prozessen wird gezeigt, dass das in Kapitel 4 entwickelte S-PM Strukturmodell geeignet ist auch komplexe Prozesse zu erfassen und dass die Spezifikation der Ausführungssemantik in den Kapiteln 5 und 6 mit CoreASM ausführbar ist, das komplette Strukturmodell abdeckt und durch die Konsolenanwendung eine interaktive Validierung anhand einer abstrakten Prozessausführung möglich ist. In Tabelle 7.1 wird ein Überblick über die in den jeweiligen Prozessmodellen verwendeten S-PM Sprachelemente gegeben. In Abbildung 7.9 wird mit dem SID ein Überblick des im PEN Projekt entworfenen Prozessmodells gegeben. Im folgenden Kapitel wird diese Arbeit zusammengefasst und ein Ausblick auf zukünftige Arbeiten gegeben.

Abbildung:	2.3	7.3	7.4	7.5	7.6	7.7	7.8
Internal Action	X		X		X	X	X
Tau						X	
Send / Receive	X	X	X	X	X	X	X
CorrelationID			X				
IP Limit						X	
CloseIP / OpenIP / IsIPEmpty			CloseIP / IsIPEmpty				
Modal Split / Modal Join	X	X					X
CallMacro			End-Parameter	Argumente			
VarMan			selection		extract		
Select Agents				X	X	X	
Multisubjekte			X	X			
Subject Restart	X		X			X	
externe Subjekte / Service Processes						X	
Mobility of Channel					X		
Cancel Aktion							X
Knotenprioritäten		X					X
Kantenprioritäten							X
versteckte Kanten							X
timeout-Kanten		X	X				
auto-Kanten		X		X			X
cancel-Kanten			X			X	X

Tabelle 7.1: Unterstützung der Sprachelemente

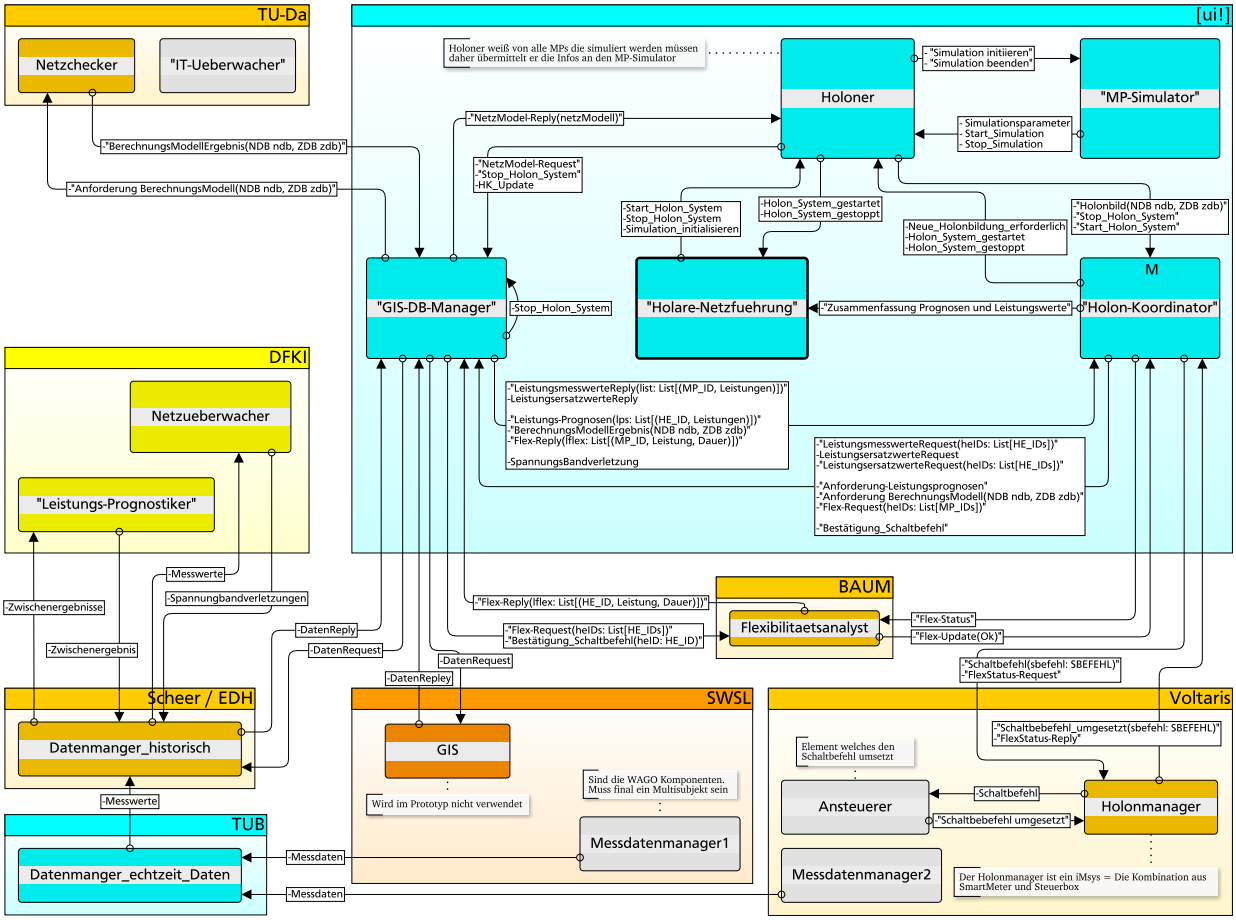


Abbildung 7.9: Subject Interaction Diagram des in PEN entworfenen Prozessmodells, aus [16]

8 Zusammenfassung und Fazit

In dieser Arbeit wurde eine Ausführungssemantik für subjektorientierte Prozessmodelle mit Abstrakten Zustandsmaschinen spezifiziert, die mit dem CoreASM Framework ausgeführt werden kann und durch eine Konsolenanwendung eine interaktive Prozessvalidierung ermöglicht.

Nach der Vorstellung der für diese Arbeit nötigen Grundlagen zum Subjektorientierten Prozessmanagement (S-PM) und zu Abstrakten Zustandsmaschinen (ASM) im zweiten Kapitel wurde im dritten Kapitel der aktuelle Forschungs- und Technikstand zur abstrakten Ausführung von S-PM Prozessmodellen betrachtet. Dort wurde festgestellt, dass zwischen S-PM Prozess-Engines auf der einen Seite und formalen Arbeiten auf der anderen Seite eine Lücke existiert, die durch vier Anforderungen beschrieben wurde.

Im vierten Kapitel wurde unter Berücksichtigung der Anforderungen ein Strukturmodell für S-PM Prozessmodelle entwickelt, bei dem Sprachelemente aus mehreren theoretischen und praktischen Arbeiten zusammengeführt wurden. Zur unterbrechungsfreien Ausführung von kritischen Abschnitten im Subjektverhalten wurde das Strukturmodell um Knotenprioritäten ergänzt, welche außerdem zum priorisierten Empfangen von Nachrichten zur Ausnahmebehandlung verwendet werden. Um bei der Behandlung von Ausnahmesituationen modellieren zu können, ob das unterbrochene Verhalten anschließend fortgeführt oder abgebrochen werden soll, wurde eine neuartige Cancel Aktion entworfen.

Die gewählte Umsetzung zur Ausnahmenbehandlung durch eine priorisierte Receive Aktion mit dem anschließenden Abbruch durch eine Cancel Aktion erfüllt zwar die dritte Anforderung, eine zukünftige Arbeit sollte jedoch weitergehend untersuchen, ob die Modellierung vereinfacht werden kann. Beispielsweise könnte das Observer-Konzept mit einer Angabe, ob das unterbrochene Verhalten fortgesetzt werden soll oder nicht, erweitert und in das Strukturmodell aufgenommen werden. Das Obserververhalten könnte potentiell auch mit dem Ansatz aus [13] durch eine priorisierte Schicht umgesetzt werden.

Da ein Prozessmodell so gestaltet sein kann, dass eine Select Agents Aktion mehrfach hintereinander in einer Schleife aufgerufen wird, muss sichergestellt werden, dass die Auswahl immer neuer Agenten nicht zu potentiell unendlich vielen aktiven Subjekten führt.

In [22] wurde das Problem dadurch umgangen, dass die Aktion blockiert, bis alle Instanzen des Subjektes terminiert sind. In dieser Arbeit wurde bereits der dort als Alternative skizzierte Ansatz eines „lazy“ Send umgesetzt, da ein ASM Agent erst beim tatsächlichen Senden und nicht bereits bei der Agentenwahl gestartet wird – dies löst die Problematik der Unbeschränktheit jedoch nicht. Eine zukünftige Arbeit, die sich näher mit der Verifikation auf Interaction Soundness befasst, muss daher untersuchen, ob eine Blockade der Select Agents Aktion notwendig ist oder ob es beispielsweise ausreicht, die Gesamtzahl der zu einem Zeitpunkt ausgeführten Instanzen eines Multisubjektes zu limitieren.

Im fünften und sechsten Kapitel wurde die Ausführungssemantik für das S-PM Strukturmodell mit Abstrakten Zustandsmaschinen spezifiziert. Der Entwurf der Spezifikation im CoreASM Dialekt hat es ermöglicht auf eine separate Implementierung zu verzichten, da sie direkt mit dem CoreASM Interpreter ausgeführt werden kann.

Durch die Verwendung von [12] als Grundlage dieser Arbeit konnten Ansätze zur Umsetzung der Aktionen zum Senden und Empfangen übernommen werden. Zur Unterstützung von Makros, parallelen Abläufen innerhalb eines Subjektes und der Knotenprioritäten mussten jedoch größere Anpassungen vorgenommen werden: vor allem wurde eine Liste aktiver Knoten auf Ebene einer Makroinstanz hinzugefügt, sowie die `MacroBehaviour` Regel, welche die auszuführenden Knoten auswählt.

In der Umsetzung des Inputpool Limits wurde festgestellt, dass in [12] eine potentielle Unendlichkeit beim Senden von Nachrichten mit einer CorrelationID besteht, da die Anzahl an CorrelationIDs unbeschränkt ist. Das Problem wurde in dieser Arbeit durch eine Anpassung der Definition und Semantik der Limitierung gelöst.

In Kapitel 7 wurde die Spezifikation, und somit auch das Strukturmodell, anhand der abstrakten Ausführung von Beispielprozessen auf Vollständigkeit und Fehlerfreiheit überprüft. Für eine kontinuierliche automatisierte Überprüfung der Spezifikation wurde eine Testumgebung für Komponenten- und Integrationstests entwickelt.

Zur Durchführung der interaktiven Prozessvalidierung wurde eine Konsolenanwendung mit Scala entwickelt, die über ein eigens entwickeltes CoreASM Plugin, das eine Akka-Schnittstelle zum Zugriff auf ASM Funktionen anbietet, mit der Ausführungseinheit verbunden wurde.

Bei der abstrakten Ausführung des Prozessmodells aus dem PEN Projekt als Abschluss des Machbarkeitsnachweises wurde festgestellt, dass die Performance der Ausführungseinheit darunter leidet, dass CoreASM dauerhaft die Programme aller ASM Agenten auswertet, viele Subjekte jedoch auf eine Benutzereingabe, einen Timeout oder den Empfang

einer Nachricht warten und demnach die Auswertung von deren ASM Agenten keine Änderungen bewirken. Eine Unterstützung von Event Driven ASMs [2, Kapitel 6.5] durch den CoreASM Interpreter wäre hier wünschenswert, alternativ könnte eine zukünftige Arbeit untersuchen, ob die Verwendung eines anderen ASM Interpreters oder eines Compilers möglich ist und eine Verbesserung der Performance liefert.

Die Ergebnisse dieser Arbeit ermöglichen eine Validierung von subjektorientierten Prozessmodellen anhand einer formalen Ausführungssemantik unter Erfüllung aller gestellten Anforderungen und stärken damit das formale Fundament des Subjektorientierten Prozessmanagements.

Weitergehend wird im Lebenszyklus eines Prozessmodells zur Qualitätssicherung vor der Validierung eine Verifikation benötigt, die auf der gleichen formalen Ausführungssemantik basiert, so dass die abstrakte Ausführung eines Prozessmodells in beiden Phasen exakt gleich erfolgt und somit übertragbare Ergebnisse liefert.

Um strukturelle Fehler in einem Prozessmodell zu verhindern ist eine Verifikation auf Structural Soundness notwendig. In [22, Kapitel 4] wurden bereits Bedingungen für einen strukturell fehlerfreien Prozess gestellt. Diese Regeln müssen jedoch an das in dieser Arbeit verwendete Modell angepasst werden. Beispielsweise muss für die CallMacro Aktion sichergestellt werden, dass für alle ausgehenden Kanten des Aktionsknotens eine entsprechende End Aktion innerhalb des aufzurufenden Makros existiert.

Nach der Prüfung auf strukturelle Fehler muss eine Verifikation auf Interaction Soundness erfolgen, die mögliche Verklemmungen bei der Interaktion von Subjekten ausschließt. In [12, Kapitel 5.2] wurde bereits eine Verifikationseinheit mit Abstrakten Zustandsmaschinen entworfen. Die dort verwendete Definition zum Aufbau des zu untersuchenden Zustandsraumes ist, unter anderem durch die Modellunterschiede wie der aktiven Knoten in einer Makroinstanz, nicht mehr anwendbar. Daher muss untersucht werden, ob der dortige Ansatz weiterhin als Grundlage geeignet ist, um eine Verifikationseinheit auf Basis der in dieser Arbeit spezifizierten Ausführungssemantik zu konstruieren.

Literaturverzeichnis

[1] Albert Fleischmann, Werner Schmidt, Christian Stary, Stefan Obermeier, and Egon Börger. *Subject-oriented business process management*. Springer Publishing Company, Incorporated, 2012.

[2] Egon Börger and Robert Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer Science & Business Media, 2012.

[3] Roozbeh Farahbod. CoreASM: an extensible modeling framework & Tool environment for high-level design and analysis of distributed systems. *Ph. D. dissertation*, 2009.

[4] Stephan Borgert and Max Mühlhäuser. A S-BPM Suite for the Execution of Cross Company Subject Oriented Business Processes. In *S-BPM ONE-Scientific Research*, pages 161–170. Springer, 2014.

[5] Johannes Kotremba, Stefan Raß, and Robert Singer. Distributed Business Processes - A Framework for Modeling and Execution, September 2013. arXiv:1309.3126.

[6] URL `https://i2pm.net/interest-groups/ueber-flow`. Abgerufen am 01.10.2018.

[7] Florian Strecker, Reinhard Gniza, Thomas Hollosy, and Florian Schmatzer. Business-Actors as base components of complex and distributed software applications. In *Proceedings of the 8th International Conference on Subject-oriented Business Process Management*, page 9. ACM, 2016.

[8] URL `https://www.metasonic.de/metasonic-flow`. Abgerufen am 01.10.2018.

[9] Stephan Borgert, Joachim Steinmetz, and Max Mühlhäuser. ePASS-IoS 1.1: Enabling Inter-enterprise Business Process Modeling by S-BPM and the Internet of Services Concept. In *International Conference on Subject-Oriented Business Process Management*, pages 190–211. Springer, 2011.

[10] Nikoleta Hadzhiivanova. A Pi-calculus Based Semantics for S-BPM Processes. Bachelor-Thesis, TU Darmstadt, Juli 2012.

[11] Egon Börger. A Subject-Oriented Interpreter Model for S-BPM. *Appendix in: A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier, E. Börger: Subjektorientiertes Prozessmanagement, Hanser-Verlag, München*, 2011.

[12] Manuel Bandmann. Spezifikation einer Ausführungs-und Verifikationseinheit mit Abstract State Machines für die Subject-Oriented Business Process Management Modellierungsnotation. Diplomarbeit, TU Darmstadt, Juli 2014.

[13] Matthes Elstermann, Detlef Seese, and Albert Fleischmann. Using the arbitrator pattern for dynamic process-instance extension in a work-flow management system. In *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, pages 323–326. Springer, 2012.

[14] Harald Lerchner and Christian Stary. An open S-BPM runtime environment based on abstract state machines. In *Business Informatics (CBI), 2014 IEEE 16th Conference on*, volume 1, pages 54–61. IEEE, 2014.

[15] URL `http://www.polyenergynet.de/`. Abgerufen am 01.10.2018.

[16] Stephan Borgert. private Korrespondenz, 2018.

[17] Albert Fleischmann. PASS - A Technique for Specifying Communication Protocols. In *Proceedings of the IFIP WG6.1 Seventh International Conference on Protocol Specification, Testing and Verification VII*, pages 61–76, Amsterdam, The Netherlands, The Netherlands, 1987. North-Holland Publishing Co. ISBN 0-444-70293-8. URL `http://dl.acm.org/citation.cfm?id=645831.670083`.

[18] Albert Fleischmann. *Distributed Systems: Software Design and Implementation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1994. ISBN 0387573828.

[19] Albert Fleischmann, Stephan Borgert, Matthes Elstermann, Florian Krenn, and Robert Singer. An overview to S-BPM oriented Tool Suites. *S-BPM ONE*, pages 30–31, 2017.

[20] Albert Fleischmann, Stefan Oppl, Werner Schmidt, and Christian Stary. *Ganzheitliche Digitalisierung von Prozessen*. Springer, 2018. ISBN 978-3-658-22647-3.

[21] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999. ISBN 0521658691.

[22] Richard Stein. Reduktion des Zustandsraums für die Verifikation von subjektorientierten Geschäftsprozessmodellen. Bachelor-Thesis, TU Darmstadt, Oktober 2013.

[23] Arne Link. Verteilte Modellierung und Ausführung subjektorientierter Geschäftsprozessmodelle. Bachelor-Thesis, TU Darmstadt, 2015.

[24] URL `https://github.com/CoreASM/coreasm.core/blob/8814b99/org.coreasm.engine/test-rsc/without_test_class/DiningPhilosophers.casm`. Abgerufen am 01.10.2018.

[25] Egon Börger, Peter Päppinghaus, and Joachim Schmid. Report on a practical application of ASMs in software design. In *Abstract State Machines-Theory and Applications*, volume 1912, pages 361–366. Springer, 2000.

[26] URL `https://bitbucket.org/inkytonik/kiama`. Abgerufen am 01.10.2018.

[27] URL `https://archive.codeplex.com/?p=asml`. Abgerufen am 01.10.2018.

[28] URL `https://casm-lang.org/`. Abgerufen am 01.10.2018.

[29] URL `https://github.com/coreasm/coreasm.core`. Abgerufen am 01.10.2018.

[30] URL `https://www.reactivemanifesto.org/glossary#Back-Pressure`. Abgerufen am 01.10.2018.

Erklärung zur Abschlussarbeit gemäß § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, André Wolski, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Bachelorarbeit stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Bei einer Bachelorarbeit des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, den 23. April 2019

Unterschrift