Fachhochschule
FH JOANNEUM Gesellschaft mbH

Activity Model Runtime Engine für Python

Diplomarbeit
zur Erlangung des akademischen Grades eines
Diplomingenieurs für technisch-wissenschaftliche Berufe (FH)
eingereicht am
Studiengang Informationsmanagement

Betreuer: FH-Prof. Dipl.-Ing. Peter Salhofer
Firmenbetreuer: Jens W. Klein; BlueDynamics Alliance

eingereicht von: Johannes Raggam
Personenkennzahl: 0710062001

September 2009

FH JOANNEUM
INFORMATIONSMANAGEMENT

BlueDynamics
Alliance

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum Vor- und Nachname

Inhaltsverzeichnis

I Einleitung **2**

1 Gegenstand der Diplomarbeit **3**

1.1 Ausgangslage 3

1.2 Problemstellung 4

1.3 Zielsetzung und Anforderungen 6

1.4 Use Case *PatientInnenversorgung in einem Krankenhaus* 6

2 Aktuelle Ansätze zur Metamodellierung und ausführbaren Modellen **9**

3 Forschungsleitende Ansätze **11**

4 Ziele und Abgrenzungen der Diplomarbeit **12**

II Theoretische Grundlagen **14**

5 Modellierung und Metamodellierung **15**

5.1 Modelle, Modellierungssprachen und Ausführung von Modellen . 15

5.2 Metamodellierung 17

5.3 Unified Modeling Language 19
5.3.1 Einführung 19
5.3.2 Geschichte 19
5.3.3 Aufbau 21
5.3.4 Diagramme der UML 22

6 Modellierungssprachen im Kontext der Prozessmodellierung 25

6.1 Petri-Netze 25
6.1.1 Einführung in Petri-Netze 25
6.1.2 Mögliche Grundstrukturen in Petri-Netzen 27
6.1.3 Analytische Fragestellungen über den Zustand von Petri-Netzen 29
6.1.4 Formale Definition von Petri-Netzen 29
6.1.5 Erweiterungen von Petri-Netzen 30
6.1.6 Praktische Bedeutung von Petri-Netzen 31
6.1.7 Beispiel für das Petrinetz *PatientInnenversorgung in einem Krankenhaus* 31
6.2 UML Aktivitätsdiagramme 33
6.2.1 Activities 33
6.2.2 Actions 35
6.2.3 Object Nodes 37
6.2.4 Control Nodes 37
6.2.5 Activity Edges 40
6.2.6 Tokens 41
6.2.7 Anwendung von Aktivitätsdiagrammen 42

6.2.8 Beispiel für die Aktivität *PatientInnenversorgung in einem Krankenhaus* 43
6.3 Business Process Modeling Notation 45
6.3.1 Hintergrund 45
6.3.2 Tokens als Zustandsmarker 46
6.3.3 Flow Objects 46
6.3.4 Connecting Objects 50
6.3.5 Swimlanes 52
6.3.6 Artifacts 52
6.3.7 Beispiel für das BPMN Diagramm *PatientInnenversorgung in einem Krankenhaus* 54
6.4 Diskussion der vorgestellten Modellierungssprachen 56

III Activity Model Runtime Engine für Python 58

7 Metamodell der Activity Model Runtime Engine für Python 59
7.1 Einführung 59
7.2 Das AMREP Metamodell im Überblick 60
7.2.1 Allgemeine Metamodell Elemente 60
7.2.2 Aktivität 62
7.2.3 Kanten 64
7.2.4 Knoten 64
7.3 Modellvalidierung 66
7.4 Unterschiede zum UML2 Metamodell für Aktivitäten 67
7.5 Konventionen für die Erstellung von Metamodellen für AMREP . 69

8 Interpreter der Activity Model Runtime Engine für Python 71

8.1 Einführung 71

8.2 Implementierung der Runtime Engine 71

8.3 Token-Fluss in der AMREP 77

8.3.1 Allgemeines 77

8.3.2 Daten-Payload 78

8.3.3 Tokens referenzieren Kanten 79

8.3.4 Traverse-To-Completion Problem 79

8.4 Executions als Implementierung von Aktionen 81

8.4.1 Einführung 81

8.4.2 Trennung zwischen Modell und Implementierung 82

8.4.3 Beispiel 83

9 Verwendung der Activity Model Runtime Engine für Python 86

9.1 Modellerstellung mit Python 86

9.2 Verwendung der Runtime 89

9.3 Modellerstellung mit einem UML-Editor 93

10 Zusammenfassung, Ergebnisse und Ausblick 96

IV Anhang 98

Literaturverzeichnis 99

Abbildungsverzeichnis

5.1 Metalevel nach OMG 18

6.1 Stelle in Petrinetzen 26
6.2 Transition in Petrinetzen 26
6.3 Verbindung unterschiedlicher Knotentypen in Petrinetzen 26
6.4 Token in Petrinetzen 27
6.5 Erzeugen von Marken in Petrinetzen 27
6.6 Löschen von Marken in Petrinetzen 27
6.7 Vervielfachung von Marken in Petrinetzen 28
6.8 Synchronisation in Petrinetzen 28
6.9 Quelle für Marken in Petrinetzen 28
6.10 Archiv für Marken in Petrinetzen 28
6.11 Ausschließende Fortsetzungsalternativen in Petrinetzen 28
6.12 Asynchrones Zusammenführen in Petrinetzen 29
6.13 Beispiel für das Petrinetz *PatientInnenversorgung in einem Krankenhaus* 32
6.14 Aktivität mit Parametern und Bedingungen 35
6.15 Aktion mit Vor- und Nachbedingungen 35

6.16 Aktion mit verschiedenen Arten von Pins 37
6.17 Decision Node mit Bedingungen 38
6.18 Merge Node 38
6.19 Fork Node 39
6.20 Join Node 39
6.21 Initial Node 39
6.22 Final Node 40
6.23 Flow Final Node 40
6.24 Beispielsaktivität *PatientInnenversorgung in einem Krankenhaus* 44
6.25 Verschiede Arten von Events in BPMN 47
6.26 Sub-Process in BPMN 48
6.27 Task in BPMN 48
6.28 Databased Exclusive Gateway in BPMN 49
6.29 Eventbased Exclusive Gateway in BPMN 49
6.30 Inclusive Gateway in BPMN 49
6.31 Parallel Gateway in BPMN 50
6.32 Complex Gateway in BPMN 50
6.33 Sequence Flows in BPMN 51
6.34 Message Flow in BPMN 51
6.35 Assoziationen in BPMN 52
6.36 Pool und Lanes in BPMN 53
6.37 Data Object in BPMN 53
6.38 Text Annotation in BPMN 53
6.39 Group in BPMN 54

6.40 PatientInnenversorgung in einem Krankenhaus in BPMN Darstellung 55

7.1 Allgemeine Metamodell Elemente in AMREP 61

7.2 Aktivität, Behavior und Package in AMREP 63

7.3 Kante in AMREP 64

7.4 Knoten in AMREP 65

7.5 Kontrollknoten in AMREP 67

8.1 Klassen der Runtime Engine 73

8.2 Aktivitätsmodell der Start-Methode 74

8.3 Aktivitätsmodell der Next-Methode 76

8.4 Anwendung des Command Pattern für Executions 83

9.1 Modell für den XMI Import 95

Tabellenverzeichnis

8.1 Zusammenhang zwischen Executions und Modellelementen . . 82

Listings

8.1 Definition von Executions in Python 83

9.1 Usecase *Patientenversorgung in einem Krankenhaus* als Python-Modell 86

9.2 Verwendung der Runtime 89

9.3 Import von XMI Dateien 94

Abkürzungsverzeichnis

AMREP: Activity Model Runtime for Python
BE-Netz: Bedingungs-Ereignis-Netz
BMI: Business Modeling & Integration
BPDM: Business Process Definition Meta Model
BPM: Business Process Management
BPMI: Business Process Management Initiative
BPMN: Business Process Modeling Notation
DTF: Domain Task Force
EMF: Eclipse Modeling Framework
EMOF: Essential Meta Object Facility
Engl: Englisch
EPK: Ereignisgesteuerte Prozesskette
Ff: Fort folgend
IM-Netz: Individuelle-Marken-Netz
ITU: International Telecommunication Union
MDA: Model Driven Architecture
MOF: Meta Object Facility
OASIS: Organization for the Advancement of Structured Information Standards
OCL: Object Constraint Language
OMG: Object Management Group
PIM: Platform Independent Model
RFP: Request for Proposal
UML: Unified Modeling Language
Vgl: Vergleiche
WS-BPEL: Web Services Business Process Execution Language
XMI: XML Metadata Interchange
XML: Extensible Markup Language
ZCA: Zope Component Architecture

Danksagung

Die vorliegende Arbeit wäre ohne folgende Personen nicht oder zumindest nicht in dieser Form möglich gewesen.

Ich möchte mich deshalb bedanken: Bei FH-Prof. Dipl.-Ing. Werner Fritz für die Möglichkeit zum Wiedereinstieg in das Studium Informationsmanagement, bei FH-Prof. Dipl.-Ing. Peter Salhofer für die Betreuung und wertvollen Hinweise zur Verbesserung der Diplomarbeit, bei Jens Klein für die externe Betreuung dieser Arbeit und für die spannenden Diskussionen gemeinsam mit Robert Niederreiter zum Thema Runtime Engine und Modellierung im Allgemeinen.

Ganz besonders möchte ich mich auch bei meinen Eltern Barbara und Johann Raggam bedanken.

Vor allem bedanke ich mich bei meiner Partnerin Johanna Muckenhuber für all die aufgebrachte Geduld, ständige Motivation und Inspiration und die Ruhe, die ich bei ihr finden konnte.

Abstract (English)

Business process modeling techniques have been helpful to understand the activities which are done in enterprises. However, traditional information systems for enterprises show little flexibility in adapting business processes found in real environments. The BPEL standard tries to meet these shortcomings but is limited to web services. A more general modeling approach can be found in the standards published by the Object Management Group, specifically the UML standard.

The Activity Diagram specification of the UML standard can be used to model behavioral systems, including business processes. Formal models with sufficient semantics can be executed. The semantics is ensured by a well defined meta model which provides a common basis for executable models. Furthermore, executable models can be interpreted rather than compiled to a target Language. A model interpreter also allows run-time changing of the executed behavior.

In this diploma thesis a prototype was built in the programing language python, which consists of a customized metamodel and a runtime interpreter for activity models. Such a activity model runtime engine does not yet exist for Python. The functionality was proven for a use case by unit tests. A restriction is the limited compatibility to the UML standard for activity diagrams. Nevertheless, the results show that modeled processes can be successfully used with the Activity Model Runtime Engine for Python. More use cases have to be implemented to prove the application of the customized metamodel and token semantics for the respective purpose.

Abstract (Deutsch)

Geschäftsprozess-Modellierungstechniken haben das Verständnis von Abläufen in Unternehmen erhöht. Traditionelle Informationssysteme bieten allerdings wenig Flexibilität, Abläufe, die im realen Umfeld verrichtet werden, zu implementieren. Der Standard BPEL versucht diese Lücke zu schließen, ist aber auf Web-Services beschränkt. Ein generellerer Modellierungsansatz kann in den von der Object Management Group veröffentlichten Standards, speziell dem UML-Standard, gefunden werden.

Die Aktivitätsdiagramm-Spezifikation des UML-Standards kann zur Modellierung des Verhaltens von Systemen inklusive der Modellierung von Geschäftsprozessen verwendet werden. Formale Modelle mit ausreichend semantischer Information können ausgeführt werden. Die Modellsemantik wird durch ein Metamodell sichergestellt, das als Basis für ausführbare Modelle dient. Darüber hinaus können solche Modelle interpretiert werden, anstatt sie in eine Zielsprache zu kompilieren. Ein Modellinterpreter ermöglicht auch das Ändern von Modellen zur Laufzeit.

In dieser Diplomarbeit wurde eine prototypische Implementierung eines angepassten Metamodells und eines Modellinterpreters für Aktivitäten in der Programmiersprache Python erstellt. Eine solche Activity Model Runtime Engine existiert für diese Programmiersprache noch nicht. Die Funktionalität wurde anhand eines Use Case durch Unit-Tests bewiesen. Eine Einschränkung ist die limitierte Kompatibilität zum UML Standard für Aktivitätsdiagramme. Das Ergebnis zeigt aber, dass modellierte Abläufe mit der Activity Model Runtime Engine für Python erfolgreich umgesetzt werden können. Weitere Use Cases müssen noch implementiert werden, um die Anwendbarkeit des angepassten Metamodells und der angepassten Token-Semantik für den jeweiligen Zweck zu prüfen.

The Zen of Python [1]

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren`t special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you`re Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it`s a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let`s do more of those!
>>>
```

[1] "The Zen of Python" formulierte der langjährige Python Entwickler Tim Peters als Zusammenfassung der Entwicklungsprinzipien des Python-Erfinders und *Benevolent Dictator For Life* Guido van Rossum. Sie sind als *Easter Egg* in den Python Interpreter implementiert und können durch den Befehl `import this` ausgegeben werden (vgl. [Peters 2004]).

Teil I

Einleitung

Kapitel 1

Gegenstand der Diplomarbeit

1.1 Ausgangslage

Eine der Anforderungen an moderne betriebliche Informationssysteme ist die Unterstützung von Geschäftsprozessen in Organisationen.

In der Literatur existiert eine Vielzahl von Geschäftsprozessdefinitionen. Schwickert und Fischer haben daraus eine umfassende Definition abgeleitet:

”Der Prozeß ist eine logisch zusammenhängende Kette von Teilprozessen, die auf das Erreichen eines bestimmten Zieles ausgerichtet sind. Ausgelöst durch ein definiertes Ereignis wird ein Input durch den Einsatz materieller und immaterieller Güter unter Beachtung bestimmter Regeln und der verschiedenen unternehmensinternen und -externen Faktoren zu einem Output transformiert. Der Prozeß ist in ein System von umliegenden Prozessen eingegliedert, kann jedoch als eine selbständige, von anderen Prozessen isolierte Einheit, die unabhänig von Abteilungs- und Funktionsgrenzen ist, betrachtet werden” ([Schwickert und Fischer 1996], S.10-11).

Geschäftsprozesse können mithilfe einer Modellierungssprache in Modellen abgebildet werden. Modelle sind vereinfachte Abbildungen realer Zusammenhänge für einen bestimmten Zweck (Siehe dazu auch 5.1, S.15). Modelle, die auf einer formalen Modellierungssprache basieren und in Relation zu einer seman-

tischen Domäne definiert sind können für eine Automatisierung durch Codegenerierung, Ausführung von kompilierten Modellen oder Interpretation[1] durch Maschinen verwendet werden (vgl. [Object Management Group 2008b], S.12). Interpretierte Modelle erlauben innerhalb gewisser Grenzen für zukünftige Programmschritte eine Änderung der Modelle zur Laufzeit.

Der de-facto Software-Modellierungsstandard *Unified Modeling Language* (UML) der *Object Management Group* (OMG) definiert mit den Aktivitätsdiagrammen Elemente zur modellhaften Darstellung dynamischen Verhaltens, die auch für die Modellierung von Geschäftsprozessen verwendet werden können.

Aktivitäten im Sinne der UML sind abgrenzbare Tätigkeiten, die von sogenannten Ereignissen (Events) ausgelöst werden und einen Eingangszustand (Input) in einen Ausgangszustand (Output) überführen. Eine Aktivität stellt eine Verhaltensbeschreibung dar, die aus einer oder mehreren Aktionen besteht. Der Knotentyp Aktion ist ein einzelner Tätigkeitsschritt, dessen Granularität nicht weiter erhöht wird. Eine Aktivität ist ein gerichteter Graph mit Knoten und Kanten, die miteinander verbunden sind. Neben Aktionen gibt es noch weitere Knoten, die den Ablauf steuern, wie zum Beispiel Verzweigungen (Fork), Zusammenführungen (Join und Merge) und Entscheidungen (Decision).

1.2 Problemstellung

Geschäftsprozesse werden in der Regel von AnalystInnen und ManagerInnen mit tiefem Verständnis betrieblicher Zusammenhänge definiert und in Modellen abgebildet. Diese Modelle können zur Dokumentation betrieblicher Abläufe dienen, zur Analyse von Optimierungspotentialen herangezogen werden und bei EDV-basierten Integrationen in ein betriebliches Informationssystem für die automatisierten Unterstützung von Geschäftsprozessen verwendet werden (Siehe auch Kapitel 2, S.9).

[1] "Ein Interpreter ist ein Stück Software, das ein Modell zur Laufzeit auswertet und dabei die Operationen ausführt, die in dem Modell beschrieben sind" ([Stahl u. a. 2007], S.174). Das Interpretieren von Anweisungen einer formalen Sprache entspricht der Zuordnung von syntaktischen Elementen der Sprache zu Elementen der semantischen Domäne (vgl. [Object Management Group 2008b], S.12).

Aufgrund fehlender Unterstützung durch geeignete Werkzeuge war es bisher meist notwendig die in Modellen abgebildeten Zusammenhänge und Sachverhalte manuell in Programmcode zu übersetzen. Bei einer solchen Vorgehensweise kann es aber zu folgenden Problemen kommen:

1. Wenn neue Anforderungen an den Ablauf der Geschäftsprozesse entstehen, müssen die geänderten oder neu erstellten Modelle mühsam in das Informationssystem integriert werden. Es entsteht auch die Gefahr, dass die Modelle mit der Umsetzung in der Software nicht mehr übereinstimmen beziehungsweise nicht mehr synchron sind.

2. Es kann eine semantische Lücke zwischen der Intention der Modellierenden und der Interpretation der technischen Umsetzenden entstehen und Geschäftsprozesse können falsch implementiert werden.

3. Ein weiteres Problem tritt auf, wenn Ausnahmen zu den modellierten und in Software umgesetzten Geschäftsprozessen auftauchen und für solche Ausnahmen keine geeigneten Mechanismen vorgesehen sind. In diesem Fall steht die von der Software geforderte Handlung dem eigentlichen Geschäftsprozess im Wege.

Die Interpretation von Geschäftsprozessmodellen kann diese Probleme wie folgt lösen:

1. Durch die direkte Interpretation von Modellen, die zuvor validiert, getestet und simuliert werden können, tritt das Problem der Synchronisation nicht auf. Das geänderte Verhalten kann direkt nach Freigabe implementiert werden. Durch die Interpretation von Modellen kann ein Neustart des Systems entfallen.

2. Eine semantische Lücke tritt nicht auf, da Modelle nach streng vorgegebenen Regeln erstellt werden und validiert, getestet und simuliert werden können. TechnikerInnen und AnalystInnen verwenden die selben Modelle, die sie gemeinsam und iterativ erstellen können, wobei dieser Punkt auch ohne ausführbare Modelle möglich ist.

3. Interpretierte Modelle können in gewissen Grenzen für zukünftige Prozessschritte ad-hoc geändert werden.

1.3 Zielsetzung und Anforderungen

Diese Diplomarbeit umfasst die Konzeption und Erstellung einer Activity Model Runtime Engine für Python (AMREP).

Eine Activity Engine erlaubt die Integration von Aktivitäten in ein Softwaresystem. Eine Activity Model Runtime Engine interpretiert die in Modellen definierten Aktivitäten zur Laufzeit (Executable Model) und erlaubt darüber hinaus das Verändern der Modelle zur Laufzeit, um sie an unvorhergesehene Anforderungen anzupassen. Eine Activity Engine unterstützt somit Ad-Hoc-Workflows, dessen Regeln während des Ablaufs geändert werden können.

Als Grundlage dient die Spezifikation der Aktivitätsmodellierung von UML2. Grund hierfür ist, dass Aktivitätsdiagramme sich in andere UML Standards integrieren lassen, für dynamische Systeme im Allgemeinen und nicht nur für Geschäftsprozesse definiert sind, dass sie eine breite Akzeptanz in der Industrie genießen und durchgängig formalisiert sind (siehe auch 6.4, S.56).

1.4 Use Case *PatientInnenversorgung in einem Krankenhaus*

Es wird von einem Use Case ausgegangen, anhand dessen die Funktionalität von AMREP definiert und bewiesen wird. Dieser Use Case dient zur Illustration der in späteren Kapiteln vorgestellten Diagrammtypen. Er ist ein fiktiver und stark vereinfachter Ablauf zur Therapie von PatientInnen in einem Krankenhaus. Der Ablauf ist hier in natürlicher Sprache beschrieben:

Start: Der Prozessablauf beginnt und ein/e PatientIn trifft im Krankenhaus ein.

Vorbedingung: Wenn der Prozess angestoßen wird, muss die Vorbedingung ”PatientIn ist eingetroffen” erfüllt sein.

Erstdiagnose: Als erste Aktion wird eine schnelle Erstdiagnose erstellt. Die Diagnose wird anhand des Gesundheitszustandes des/der Patienten/Patientin erstellt. Der Gesundheitszustand kann einen Wert zwischen 0% (PatientIn ist tot) und 100% (PatientIn ist gesund) einnehmen. Ist der Gesundheitszustand unter 30% handelt es sich um eine akute Krankheit oder Verletzung, ansonsten wird die Diagnose ”normaler Gesundheitszustand” gefällt.

Akuttherapie: Ist das Ergebnis der Erstdiagnose, dass der/die PatientIn eine akute Verletzung hat, wird eine Akuttherapie durchgeführt, die beispielsweise lebenserhaltende und stabilisierende Funktion hat. Danach wird der/die PatientIn zur Datenaufnahme weiter geschickt.

Ohne Akuttherapie: Lautet das Ergebnis der Erstdiagnose, dass der/die PatientIn keine akute Verletzung hat, wird er/sie gleich nach der Erstdiagnose zur Datenaufnahme weiter geschickt.

Datenaufnahme: In der Datenaufnahme werden Basisdaten wie Name abgefragt und gespeichert.

Datenüberprüfung und Diagnose: Nach der Datenaufnahme sollen die Überprüfung der Daten und die normale Diagnose parallel erfolgen.

Normale Therapie: Erst wenn Datenüberprüfung und Diagnose durchgeführt wurden, soll eine normale Therapie durchgeführt werden.

Gesundheitszustand zu schlecht: Hat der/die PatientIn nach der Therapie den Gesundheitszustand 90% nicht erreicht, so muss er/sie nochmals die Therapie durchlaufen.

Ende: Der Prozess wird beendet.

Nachbedingung: Bei Beendigung des Prozesses muss die Nachbedingung ”Gesundheitszustand besser oder gleich 90%” erfüllt sein.

Mithilfe dieses Usecases werden die Anforderungen an AMREP getestet. Der Usecase bildet wesentliche Eigenschaften der Prozessmodellierung ab: Es sind Aktionen vorhanden, es werden Entscheidungen anhand von Zuständen getroffen und alternative Pfade gewählt, es gibt Parallelisierungen, Synchronisationen und Zusammenführungen. Um die Möglichkeit der Änderung des

Modells zur Laufzeit zu beweisen, wird der Schritt ”Gesundheitszustand zu schlecht” im Test vorerst nicht abgebildet und zur Laufzeit nach Beendigung des Prozessschrittes ”Normale Therapie” hinzugefügt.

Kapitel 2

Aktuelle Ansätze zur Metamodellierung und ausführbaren Modellen

Stephen A. White untersucht in ”Process Modeling Notations and Workflow Patterns” die Anwendung von Geschäftsprozess Patterns nach Wil van der Aalst et al. [1] auf die Notationssprachen *Business Process Modeling Notation* (BPMN) und UML2 Aktivitätsdiagramme. Er kommt zum Schluss, dass sich beide Notationssprachen eignen, gängige Geschäftsprozessmuster zu modellieren (vgl. [White 2004a]).

Im Bereich der Metamodellierung ist mit dem Eclipse Modeling Framework (EMF) ein umfangreiches Toolkit entstanden, das auch die Erstellung von Modelleditoren und Codegenerierung unterstützt. Mit Ecore existiert eine Meta-Metamodell Implementierung, die an EMOF (Essential MOF) angelehnt ist. Das UML2 Eclipse Projekt hat ein Mapping der OMG UML2-Spezifikation zu Ecore zum Ziel (vgl. [Steinberg u. a. 2009]).

In seiner Dissertation ”A Metamodeling Framework for Software Engineering” stellt Marcus Alanen das Coral Framework als generisches Metamodellierungs-Framework vor, das in den Implementierungssprachen Python und C entwi-

[1] Siehe auch: http://is.ieis.tue.nl/research/patterns/patterns.htm

ckelt wurde. Es implementiert einen eigenen Meta-Metasprachenkern mit dessen Hilfe Metasprachen definiert werden können. Für diesen Kern existieren Mappings auf UML2, XMI, Ecore und anderen Modellierungssprachen in unterschiedlicher Implementierungstiefe (vgl. [Alanen 2007]).

Michelle L. Crane hat in ihrer PhD Thesis "Slicing UML's Three-layer Architecture: A Semantic Foundation for Behavioural Specification" eine Formalisierung der Ausführungssemantik von UML-Aktionen vorgestellt, da die UML-Spezifikation diesen Bereich unzureichend formalisiert hat. Die Arbeit kann als Ergänzung bzw. Alternative zum *Request for Proposal* (RFP) "Semantics of a Foundational Subset for Executable UML Models (FUML)" dienen (vgl. [Object Management Group 2008b]). Zur Validierung der Arbeit wurde von Crane ein Interpreter für Aktionen und Aktivitäten erstellt, der die komplexe Token-Passing-Semantik berücksichtigt. Der Interpreter unterstützt die Konstruktion, Validierung und Ausführung von Aktivitäten (vgl. [Crane 2009]).

Mit der Web Services Business Process Execution Language (WS-BPEL) existiert ein Standard, mit dem die Interaktion zwischen Webservices definiert werden kann und Geschäftsprozesse erstellt werden können (vgl. [OASIS2007 2007], S.8). WS-BPEL basiert auf XML und kann mit anderen Standards aus der WS-* Familie (bsp. WS-Policy) kombiniert werden. WS-BPEL operiert in einem SOA (Service Oriented Architecture) Umfeld (vgl. [Weerawarana u. a. 2005], S.313-315).

Mit jBPM von JBoss existiert eine Open-Source Plattform für ausführbare Prozesssprachen[2]. Die beiden implementierten Prozesssprachen BPEL und jPDL (eine JBoss Eigenentwicklung) trennen die Aktionssemantik von der Modell-Semantik [3]. jBPM in Version 3 verwendet Tokens als Zustandsmarker.

Weitere Plattformen für ausführbare Geschäftsprozesse werden unter anderem von SAP mit der SAP NetWeaver BPM Komponente[4] und von Intalio angeboten. Intalio hat eine BPEL Execution Engine unter dem Namen Apache ODE[5] unter einer Open-Source Lizenz veröffentlicht.

[2]Website: http://www.jboss.org/jbossjbpm/
[3]Website: http://jboss.org/jbossjbpm/jpdl/
[4]Website: http://www.sap.com/platform/netweaver/components/sapnetweaverbpm/index.epx
[5]Website: http://ode.apache.org/

Kapitel 3

Forschungsleitende Ansätze

In den vorhergehenden Kapiteln wurden verschiedene Ansätze der Geschäftsprozessmodellier
und Modellausführung vorgestellt. Folgende Ansätze dienen als Grundlage für
diese Diplomarbeit:

1. Geschäftsprozesse lassen sich mit UML2 Aktivitätsdiagrammen modellieren (vgl. [White 2004a]).
2. Modelle, die auf einer formalen Modellierungssprache basieren, können ausgeführt und interpretiert werden (vgl. [Object Management Group 2008b], S.12).
3. Die Ausführung kann direkt durch Interpretation des Modells erfolgen (vgl. [Crane 2009], S. 180ff.).

Des Weiteren soll diese Diplomarbeit folgende Annahme beweisen:

- Von Maschinen Interpretierte Modelle können zur Laufzeit geändert werden.

Kapitel 4

Ziele und Abgrenzungen der Diplomarbeit

Ziel der Diplomarbeit war die Erstellung einer prototypischen Implementierung eines maschinellen Interpreters für Aktivitätsmodelle in der Programmiersprache Python, der eine Änderung der Modelle zur Laufzeit zulässt und den Standard für UML2 Aktivitätsdiagramme als Grundlage hat. Dieser Interpreter und das zugehörige Metamodell wird in Folge ”Activity Runtime Engine für Python”, kurz AMREP, genannt.

Die Abgrenzungen und Einschränkungen sind:

- Die Diplomarbeit konzentriert sich auf eine prototypische Implementierung. Die Implementierung ist kein fertig einsetzbares Framework, kann aber als Basis für ein solches dienen.
- Die Funktionalität ist durch Unit-Tests bewiesen. Es wird kein graphisches User Interface zur Verfügung gestellt.
- Es wurde ein relevantes Subset des Standards für UML2 Aktivitätsmodelle umgesetzt. Auf eine vollständige Umsetzung des Standards wurde aufgrund der hohen Komplexität verzichtet. Es wurden einige Vereinfachungen durchgeführt, die die Implementierung des Standards im Metamodell nicht mehr vollständig kompatibel zur UML2 Spezifikation macht. Auf die Unterschiede wird im Kapitel 7.4, S.67 hingewiesen.

- Modelle werden als Python-Modelle durch Instantiierung von Python-Metamodellklassen erstellt.
- Es existiert eine XMI Modellimport-Komponente, die einen Import durch kompatible UML2 Modellierungswerkzeuge erstellte Modelle ermöglicht.
- Die Activity Model Runtime Engine ist für die Steuerung des Prozessablaufs zuständig. Die eigentlichen Tätigkeiten bzw. Aktionen wurden als Business Objekte außerhalb der Activity Model Runtime Engine definiert.

Die Activity Runtime Engine für Python wurde in der Programmiersprache Python erstellt und verwendet das Zope Component Architecture[1] (ZCA) Framework. Python und das ZCA Framework sind Kerntechnologien der Partnerfirma BlueDynamics Alliance, für die diese Diplomarbeit erstellt wurde.

Für Python existiert noch keine Software für ausführbare Aktivitätsmodelle. Die BlueDynamics Alliance arbeitet an einem Framework für modellbasierte Datenstrukturen, das von einer Activity Runtime Engine profitieren kann. Dies sind die Grundmotivationen, die zur Konzeption der vorliegenden Arbeit geführt haben.

[1] Die Zope Component Architecture ist eine Python Bibliothek zur Entwicklung wiederverwertbarer Softwarekomponenten durch Interfaces. In Python gibt es kein natives Sprachkonstrukt für Interfaces. Interfaces sind aber für die Erstellung komplexer Software hilfreich, indem sie durch die Beschreibung von Schnittstellen eine lose Kopplung zwischen kooperierenden Komponenten, die diese Schnittstellen implementieren, ermöglichen (vgl. [Muthukadan]).

Teil II

Theoretische Grundlagen

Kapitel 5

Modellierung und Metamodellierung

5.1 Modelle, Modellierungssprachen und Ausführung von Modellen

Modelle sind eine von vielen Möglichkeiten die Umwelt zu beschreiben. Diese dienen dazu, komplexe Zusammenhänge so darzustellen, dass sie einfacher und schneller von Menschen oder Maschinen verstanden werden können. Die Komplexität der Realität erlaubt es aber gleichzeitig nicht, alle Aspekte in Modellen abzubilden. Modelle sind daher vereinfachte Darstellungen realer Sachverhalte.

Modelle dienen vor allem zur Dokumentation von Systemen. Ein System ist ein begrenzter Ausschnitt der realen Welt, deren Bestandteile (Akteure, Objekte, Zusammenhänge, Abläufe, usw.) in einem Modell beschrieben werden können. Das System kann auch über die Systemgrenzen hinaus beeinflusst werden (vgl. [Bernroider und Stix 2006]).

Der Vorgang der Erstellung von Modellen für eine Anwendung wird als Modellierung bezeichnet. Dieser Prozess lässt sich durch die Relation $R(S, P, T, M)$ ausdrücken, wobei S ein Subjekt ist, das zum Zweck P (engl.: *purpose*) für ein

Original T (engl.: *prototype*) ein Modell M entwirft. Zwischen M und T existiert eine Verkürzungsrelation in dem Sinne, dass in M nur jene Details von T abgebildet sind, die aus der Sicht von S bezüglich P relevant erscheinen (vgl. [Claus und Schwill 2006], S.425).

Modelle in diesem Sinne werden mithilfe von formal definierten Modellierungssprachen erstellt. Eine formale Sprache definiert eine Syntax für ein Alphabet, legt also die Sprachregeln fest. Syntaktisch korrekte Ausdrücke sind wohlgeformt (engl.: *well formed*). Durch Validierung des Ausdrucks anhand der syntaktischen Regeln kann überprüft werden, ob der Ausdruck wohlgeformt ist. Die Semantik einer Sprache definiert die inhaltliche Bedeutung syntaktisch korrekt angewandter Ausdrücke. Ausdrücke können jedoch auch semantisch sinnlos sein, indem sie unmögliche oder bedeutungslose Zusammenhänge beschreiben. Die Semantik eines Ausdrucks steht dabei im Kontext zu einer sogenannten semantischen Domäne, dessen Elemente durch den Ausdruck beschrieben werden. Die Semantik kann sich demnach durch verschiedene semantische Domänen ändern (vgl. [Claus und Schwill 2006], S.670-671 und (vgl. [Object Management Group 2008b], S.12)).

Somit eignen sich Modelle, die auf formal definierten Modellierungssprachen basieren, zur Ausführung und Interpretation. Ein Interpreter ist ein Programm, das ein in einer anderen Programmiersprache definiertes Programm einliest und sofort ausführt. Ein Interpreter analysiert hierbei schrittweise Anweisung für Anweisung und führt jede unmittelbar aus. Ein Vorteil von Interpretern ist, dass Programmteile geändert werden und danach direkt ausgeführt werden können, ohne das Programm in die Zielsprache neu übersetzen zu müssen. Programme können auch während der Laufzeit geändert werden. Ein Nachteil ist die längere Rechenzeit bei der Ausführung von Anweisungen (vgl. [Claus und Schwill 2006], S.325).

Die formale Definition einer Modellierungssprache kann durch Metamodelle erfolgen. Der Bereich der Metamodellierung wird im nächsten Abschnitt behandelt.

5.2 Metamodellierung

Ein Metamodell ist ein Modell, das die Regeln und Elemente zur Beschreibung von anderen Modellen definiert. Modelle, die mit den Regeln und Elementen eines Metamodells erstellt worden sind, sind Instanzen dieses Metamodells (vgl. [Rumbaugh u. a. 2005], S.459).

Ein Modell, dass eine Instanz eines Metamodells ist, kann selbst als Metamodell für andere Modelle verwendet werden. Dieses Prinzip kann beliebig oft rekursiv angewandt werden. Die OMG definiert eine vier-stufige Metalevel Hierarchie, wie in Abbildung 5.1 dargestellt und nach folgendem Schema aufgebaut (vgl. [Object Management Group 2009b], S.16-17):

M0: Die Ebene M0 ist die niedrigste und konkreteste Ebene. Hier finden sich die Laufzeitinstanzen

M1: Auf der Ebene M1 existiert das Modell, aus dem Laufzeitinstanzen instantiiert werden

M2: M2 ist die Ebene des Metamodells, das die Modellelemente und Modellierungsregeln der Ebene *M1* definiert

M3: M3 ist die höchste und abstrakteste Ebene, die Ebene des Meta-Metamodells. Höhere Ebenen werden nach der OMG nicht angegeben, da metamodellbasierte Sprachen als reflexiv [1] definiert werden können und so die Möglichkeit haben sich selbst zu beschreiben. Die UML Infrastructure Library und die Meta Object Facility (MOF) sind solche Beispiele reflexiver, sich selbst definierender Sprachen

Das Prinzip der Metaebenen oder -levels kann man in vielen Beispielen in der Informatik finden. Die Beziehung Klasse - Instanz/Objekt besitzt eine zweistufige Metalevel Hierarchie, relationale Datenbanksysteme mit Systemtabellen, Tabellen und Datensätzen eine dreistufige. Die UML benutzt die von der OMG definierte vierstufige Metalevel Hierarchie, auf deren Ebenen sich folgende Bereiche der UML wiederfinden (vgl. [Object Management Group 2006], S.8-9):

M3: MOF (Meta Object Facility) bzw. UML Infrastructure Library

[1] Reflexion ist die Möglichkeit eines Programms, seinen eigenen Status während der Laufzeit abzufragen und zu manipulieren (R.G. Gabriel et al. zitiert nach [Rivard 1996]).

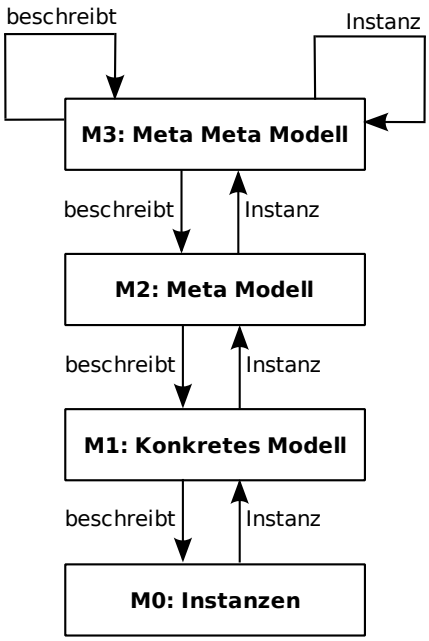

Abbildung 5.1: Metalevel nach OMG (vgl. [Stahl u. a. 2007], S.62)

M2: UML Superstructure Library

M1: Benutzerdefinierte Modelle

M0: Laufzeitinstanzen des Modells

Die OMG hat mit der Meta Object Facility (MOF) einen Standard geschaffen, der als Meta-Metamodell zur Spezifikation von Modellierungssprachen verwendet werden kann (vgl. [Rumbaugh u. a. 2005], S.464). MOF basierte Modelle können in das, mit dem Standard im Zusammenhang stehende, XML Metadata Interchange (XMI) Format übergeführt werden und mit anderen Tools ausgetauscht werden.

5.3 Unified Modeling Language

5.3.1 Einführung

Die Unified Modeling Language (UML) ist eine visuelle Modellierungssprache, die für die Dokumentation, Visualisierung, Spezifikation und Konstruktion von Softwaresystemen eingesetzt werden kann. Sie wurde speziell zur Unterstützung objektorientierter Entwicklungsprozesse geschaffen.

Mit der UML können die statische Struktur und das dynamische Verhalten von Softwaresystemen beschrieben werden. Dafür werden verschiedene Arten von Diagrammen zur Verfügung gestellt.

Die UML ist in erster Linie eine Modellierungssprache, kann aber auch als Programmiersprache eingesetzt werden. Dezidierte Programmiersprachen unterstützen jedoch durch spezielle Sprachkonstrukte den Entwicklungsprozess besser. UML Modelle können aber mithilfe von Codegeneratoren in ausführbare Programme überführt werden und entsprechen somit einer Programmiersprache (vgl. [Rumbaugh u. a. 2005], S.3).

5.3.2 Geschichte

Die Modellierungssprache UML wurde entwickelt, um mehrere Ansätze objektorientierter Entwicklungsmethoden zu vereinheitlichen. Diese Entwicklungsmethoden entstanden mit der zunehmenden Popularität objektorientierter Sprachen. Eine der ersten, generell anerkannten objektorientierten Sprachen war Simula-67 aus dem Jahr 1967, aber eine weite Verbreitung erfuhr erst Smalltalk in den frühen 1980er Jahren. Ab diesem Zeitpunkt entstanden mehrere Methoden zur Unterstützung der Entwicklung objektorientierter Programme.

Im Jahre 1994 arbeiteten James Rumbaugh, Grady Booch und Ivar Jacobson bei der Rational Software Corporation gemeinsam an der Zusammenführung der von ihnen entwickelten Methoden, die zu diesem Zeitpunkt auch zu den meistverbreitetsten gehörten. 1995 wurde ein erster Entwurf der Unified Mo-

deling Language veröffentlicht. 1996 veröffentlichte die OMG ein *Request for Proposal* (RFP) für einen standardisierten Ansatz objektorientierter Modellierung. Gemeinsam mit Autoren anderer Unternehmen wurde von Booch, Jacobson und Rumbaugh im September 1997 ein überarbeiteter Entwurf bei der OMG eingereicht und von dieser im November 1997 einstimmig akzeptiert. Es wurde eine breite Unterstützung von Herstellern von Modellierungswerkzeugen sowie von AnwenderInnen und ForscherInnen der Methoden angekündigt. Die UML hat seitdem andere bis dahin verwendete Methoden weitgehend verdrängt (vgl. [Rumbaugh u. a. 2005], S.4-6).

Die Erfahrungen durch den Einsatz der UML in den folgenden Jahren führte zur Entwicklung der Version 2.0, die im Jahr 2004 veröffentlicht wurde. Die wichtigsten Änderungen sind (vgl. [Rumbaugh u. a. 2005], S.6-7):

- Sequenzdiagramme basieren auf dem Message Sequence Chart Standard der International Telecommunication Union (ITU).
- Die Konzepte der Aktivitätsmodellierung wurden von den Konzepten der Zustandsmodellierung getrennt und Notierungskonzepte aus der Geschäftsprozess-Modellierung eingeführt.
- Vereinheitlichung der Aktivitätsmodellierung mit der Aktionsmodellierung aus der UML Version 1.5.
- Änderungen bezüglich der Kompositions- und Kollaborationsdiagramme.
- Angleichung des UML Core an die MOF.
- Restrukturierung des UML Metamodells zur Vermeidung redundanter Konstrukte.
- Verfügbarkeit von Profilen für domänen- und technologiespezfischen Erweiterungen der UML.

Die aktuelle Version trägt die Versionsnummer 2.2 und ist im Februar 2009 erschienen (vgl. [Object Management Group 2009c]).

5.3.3 Aufbau

UML Infrastructure

Das *core package* stellt einen Metasprachenkern zur Verfügung, der von verschiedenen Metasprachen wie UML und MOF verwendet werden kann. Die *Infrastructure Library* stellt damit eine architektonische Angleichung zwischen UML, MOF und XMI sicher. Das *Core Package* ist der zentrale Bestandteil aller Metasprachen im OMG Umfeld und wird als der architektonische Kern der *Model Driven Architecture*[2] (MDA) Initiative der OMG angesehen.

Um eine Wiederverwertung der Elemente des Core Package zu unterstützen, sind diese in Pakete und Unterpakete organisiert, die einzeln importiert werden können. Um eine Metasprache wie UML zu erstellen, werden die Metaklassen des Core Package instantiiert. Das Core Package ist reflexiv definiert, beschreibt sich also selbst und hängt nicht von einem abstrakteren Metamodell ab (vgl. [Object Management Group 2009b], S.11-13).

Das Paket *Profiles* verwendet Elemente aus dem Core Package und definiert einen Erweiterungsmechanismus für Metamodelle, um diese bestimmten Domänen, Technologien oder Plattformen anzupassen. Profile sind dem Erweiterungsmechanismus der MOF angeglichen, aber einfacher und leichtgewichtiger implementiert (vgl. [Object Management Group 2009b], S.13-14).

UML Superstructure

Die UML Superstructure definiert das eigentliche UML Metamodell. Es besteht wiederum aus mehreren Unterpaketen, die den einzelnen Arten von Diagrammen entsprechen (vgl. [Object Management Group 2009b], S.14-15).

[2] Die Model Driven Architecture beschreibt eine Methode zur Entwicklung von Softwaresystemen, die durchgängig auf Modellen basiert. MDA setzt hierbei auf etablierte OMG Standards auf. Eine zentrale Rolle spielen UML und MOF. MDA trennt die Business-Logik von der Plattform in Form von *Platform Independent Models* (PIM) die über Transformationen in *Platform Specific Models* (PSM) übergeführt werden und so im Zielsystem implementiert werden (vgl. [Stahl u. a. 2007], S.377ff).

Language Units

Die einzelnen Diagrammenarten der UML Superstructure sind als sogenannte Language Units organisiert, die eine Sammlung zusammengehöriger Modellierungskonzepte darstellen, wie beispielsweise Aktivitätsdiagramme und Klassendiagramme (vgl. [Object Management Group 2009c], S.2).

Compliance Level

Die UML definiert vier Compliance Levels, die angeben, wie weit das UML Metamodell in einem Framework implementiert ist. Compliance Levels geben auch den Grad der Interoperabilität zwischen Werkzeugen von Toolherstellern an. Die Einhaltung eines Compliance Levels gibt Aufschluss über die Kompatibilität zu einem anderen Werkzeug. Der Level Null (L0) ist die niedrigste Stufe und entspricht der UML Infrastructure. Der Level Drei (L3) entspricht der kompletten Implementierung des UML Metamodells (vgl. [Object Management Group 2009c], S.2).

5.3.4 Diagramme der UML

Die UML unterscheidet zwischen folgenden Diagrammarten (vgl. [Pilone und Pitman 2005], S.5-7):

Strukturdiagramme

Klassendiagramme: Engl.: *Class Diagrams*. Klassendiagramme stellen die am meisten verwendete Diagrammart in UML dar. Sie erfassen die Details über Klassen und Interfaces im System und deren statischen Beziehungen.

Objektdiagramme: Engl.: *Object Diagrams*. Objektdiagramme nutzen ebenso dieselbe Notation wie Klassendiagramme. Sie stellen konkrete Objekte als Instanzen von Klassen zu einem bestimmten Zeitpunkt der Laufzeit dar.

Packetdiagramme: Engl. *Package Diagrams*. Paketdiagramme sind spezielle Klassendiagramme, die dieselbe Notation nutzen und die Gruppierung von Klassen und Interfaces in Paketen darstellen.

Komponentendiagramme: Engl.: *Component Diagrams*. Komponentendiagramme stellen die Organisation und Abhängigkeiten von Komponenten in einem System dar.

Kompositionsstrukturdiagramme: Engl.: *Composite Structure Diagrams*. Kompositionsstrukturdiagramme sind neu in UML2.0. Sie verbinden Klassendiagramme mit Komponentendiagrammen, jedoch ohne die Detaillierungstiefe beider Diagrammarten. Sie dienen dazu, komplexe Zusammenhänge bzw. Patterns darzustellen.

Verteilungsdiagramme: Engl.: *Deployment Diagrams*. Verteilungsdiagramme zeigen, wie Softwaresysteme über verschiedene Hardware verteilt sind und stellen Laufzeitkonfigurationen dar.

Verhaltensdiagramme

Anwendungsfalldiagramme: Engl.: *Use Case Diagrams*. Anwendungsfalldiagramme stellen die funktionalen Anforderungen eines Systems unabhängig von der Implementierung dar.

Aktivitätsdiagramme: Engl.: *Activity Diagrams*. Aktivitätsdiagramme sind der zentrale Inhalt dieser Diplomarbeit. Sie stellen ein Systemverhalten in Form einer Komposition von Aktionen und Kontrollknoten, verbunden durch Kanten, dar.

Zustandsdiagramme: Engl.: *State Machine Diagrams*. Zustandsdiagramme stellen die Zustandsübergänge von einzelnen Klassen oder ganzen Systemen dar.

Interaktionsdiagramme: Engl.: *Interaction Diagrams*. Interaktionsdiagramme ist eine Kategorie von Diagrammarten, die die Kommunikation zwischen Objekten darstellen. Sie Umfassen Sequenzdiagramme, Interaktionsübersichtsdiagramme, Kommunikationsdiagramme und Zeitdiagramme.

Sequenzdiagramme: Engl. *Sequence Diagrams*. Sequenzdiagramme haben die Nachrichten, die zwischen Objekten ausgetauscht werden, zum Fokus. Sie sind die häufigste Form von Interaktionsdiagrammen.

Kommunikationsdiagramme: Engl.: *Communication Diagrams*. Kommunikationsdiagramme sind Spezialfälle von Interaktionsdiagrammen und stellen die Kommunikation von Objekten dar. Der Schwerpunkt liegt dabei mehr auf den Objekten selbst als auf den Nachrichten, die diese austauschen.

Interaktionsübersichtsdiagramme: Engl.: *Interaction overview Diagrams*. Interaktionsübersichtsdiagramme sind vereinfachte Formen von Aktivitätsdiagrammen. Anstelle der Darstellung von Aktionen werden Sequenzdiagramme dargestellt. Der Fokus liegt hierbei auf die Darstellung des Elemente und Nachrichten in Aktivitäten.

Zeitdiagramme: Engl.: *Timing Diagrams*. Zeitdiagramme sind Spezialfälle von Interaktionsdiagrammen und haben die zeitliche Abfolge von Nachrichten zwischen Objekten zum Inhalt.

Kapitel 6

Modellierungssprachen im Kontext der Prozessmodellierung

6.1 Petri-Netze

6.1.1 Einführung in Petri-Netze

Ein Petri-Netz ist ein Modell zur Beschreibung und Analyse von nebenläufigen Prozessen, die in verteilten Systemen mit vielen Komponenten auftauchen (vgl. [Petri und Reisig 2008]). Nebenläufigkeit ist als Begriff allgemeiner gefasst als Parallelität, da nebenläufige Prozesse sequentielle und parallele Ausführung mit einschließen (vgl. [Claus und Schwill 2006], S. 444-445). Die theoretische Basis der Petri-Netze wurde 1962 vom Deutschen Mathematiker und Informatiker Carl Adam Petri beschrieben. Petri-Netze bilden die Basis der meisten Prozessmodell-Sprachen (vgl. [Bernroider und Stix 2006], S. 17).

Ein Petri-Netz ist ein gerichteter Graph aus Knoten, die durch Kanten verbunden sind. Es werden zwei verschiedene Knotentypen unterschieden (vgl. [Bernroider und Stix 2006] S. 17):

Stelle: Engl.: *state*. Alternative Bezeichnung: *Aktion* oder *Bedingung*. Dieser Knotentyp führt bestimmte Aktionen aus oder stellt bestimmte Bedingungen bereit. Stellen, denen Transitionen folgen, heißen *Eingabestellen*.

Stellen, die Transitionen nachgelagert sind, heißen *Ausgabestellen*. Sie werden durch einen Kreis dargestellt. Siehe Abbildung 6.1.

Abbildung 6.1: Stelle

Transition: Engl.: *transition*. Alternative Bezeichnung: *Ereignis*. Dieser Knotentyp stellt einen Zeitpunkt dar, an dem ein Ereignis eintritt und nachfolgende Stellen aktiviert werden. Siehe Abbildung 6.2.

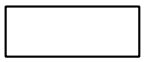

Abbildung 6.2: Transition

In Petri-Netzen können immer nur zwei unterschiedliche Knotentypen durch Kanten verbunden werden. Das impliziert, dass zwischen Aktionen bzw. Bedingungen immer Ereignisse eintreten müssen, die dann weitere Aktionen bzw. Bedingungen auslösen (vgl. [Bernroider und Stix 2006], S. 18). Siehe Abbildung 6.3.

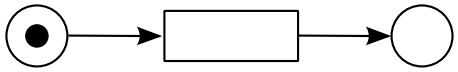

Abbildung 6.3: Verbindung unterschiedlicher Knotentypen

Ein weiteres wichtiges Konzept von Petri-Netzen sind *Marken* (engl.: *tokens*). Diese können sich in Stellen befinden und markieren, dass eine Aktion durchgeführt wird oder eine Bedingung zutrifft. Die Marken werden im Prozessablauf von einer Stelle zur nächsten weitergereicht. Stellen können beliebig viele Marken beinhalten, Kanten jedoch nur jeweils eine Marke zu einem Zeitpunkt weiter transportieren. Marken werden durch einen schwarzen Punkt innerhalb des Stellenkreises dargestellt (vgl. [Bernroider und Stix 2006], S. 17). Siehe Abbildung 6.4.

Die *Markierung* eines Petri-Netzes ist der Zustand zu einem Zeitpunkt, der durch Anzahl und Positionen der Marken beschrieben wird und durch einen

Abbildung 6.4: Stelle mit einem und drei Token

Vektor ausgedrückt werden kann. Die Position i im Vektor entspricht der Position der Stelle im Petri-Netz (vgl. [Bernroider und Stix 2006] S. 17).

Es gelten folgende Regeln für den Prozessablauf in Petri-Netzen (vgl. [Claus und Schwill 2006], S. 500):

1. Eine Transition schaltet (bzw. *feuert*) dann, wenn in jeder Eingabestelle mindestens eine Marke vorhanden ist. Die Transition ist somit *aktiv*.
2. Wenn eine Transition schaltet, wird aus jeder Eingabestelle eine Marke entnommen und für jede Ausgabestelle eine neue Marke produziert. Der Zustand des Netzes ändert sich.

6.1.2 Mögliche Grundstrukturen in Petri-Netzen

In Petri-Netzen können folgende Grundstrukturen auftreten (vgl. [Claus und Schwill 2006], S. 501 und [Bernroider und Stix 2006], S. 19-20):

Erzeugen von Marken: Ein Ereignis tritt ein und eine Marke wird erzeugt und weitergegeben (siehe Abbildung 6.5).

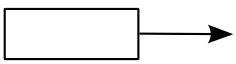

Abbildung 6.5: Erzeugen von Marken

Löschen von Marken: Eine Marke wird einer vorangehenden Stelle entnommen und gelöscht (siehe Abbildung 6.6).

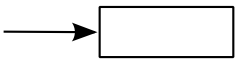

Abbildung 6.6: Löschen von Marken

Vervielfachung von Marken und Beginn einer Nebenläufigkeit: Es werden Marken für alle nachfolgenden Stellen produziert. Eventuell vorhandene Objekte werden vervielfacht (siehe Abbildung 6.7).

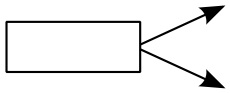

Abbildung 6.7: Vervielfachung von Marken

Synchronisation, Ende der Nebenläufigkeit und Verschmelzung von Objekten: Der Knoten wird aktiv, sobald an jeder Kante Marken eintreffen. Eventuell transportierte Objekte werden verschmolzen (siehe Abbildung 6.8).

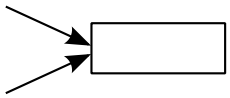

Abbildung 6.8: Synchronisation

Quelle für Marken: Die Stelle kann Marken besitzen, die nachgelagerten Transitionen weitergereicht werden können (siehe Abbildung 6.9).

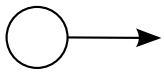

Abbildung 6.9: Quelle für Marken

Archiv für Marken: Die Stelle nimmt Marken von einer vorgelagerten Transition an (siehe Abbildung 6.10).

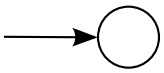

Abbildung 6.10: Archiv für Marken

Ausschließende Fortsetzungsalternativen: Je nachdem welche nachgelagerte Transition aktiv wird, wird der entsprechende Pfad genommen (siehe Abbildung 6.11).

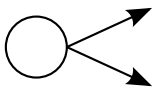

Abbildung 6.11: Ausschließende Fortsetzungsalternativen

Asynchrones Zusammenführen paralleler Aktionen und Sammelstelle: Im Gegensatz zur Synchronisation müssen nicht alle Marken gleichzeitig eintreffen (siehe Abbildung 6.12).

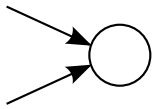

Abbildung 6.12: Asynchrones Zusammenführen

6.1.3 Analytische Fragestellungen über den Zustand von Petri-Netzen

Petri-Netze können nach folgenden Fragestellungen analysiert und klassifiziert werden (vgl. [Claus und Schwill 2006], S. 501):

Terminierung des Netzes: Es wird ausgehend von einem bestimmten Ausgangszustand untersucht, ob die Transitionen nur endlich oft schalten können.

Lebendigkeit des Netzes: Ausgehend von einem bestimmten Ausgangszustand wird untersucht, ob eine Transition t nach einem Schaltvorgang nochmals schalten kann. Wenn das nicht der Fall ist, nennt man die Transition t *tot*.

Verklemmung des Netzes: Engl.: *Deadlock*. Es wird untersucht, ob Situationen auftreten können, in denen keine Transitionen schalten können.

Erreichbarkeitsproblem: Ausgehend von zwei Markierungen M_1 und M_2 wird untersucht, ob eine Schaltfolge existiert, in der das Netz vom Zustand M_1 in M_2 wechseln kann.

Beschränkung des Netzes: Ein Netz heißt beschränkt, wenn bei jeder erreichbaren Markierung höchstens k Marken in jeder Stelle liegen.

Fairness einer Schaltfolge: Eine unendliche Schaltfolge ist dann unfair, wenn es Transitionen gibt, die in dieser Schaltfolge nie berücksichtigt werden, aber unendlich oft schalten könnten.

6.1.4 Formale Definition von Petri-Netzen

Ein Petri-Netz wird durch ein 5-Tupel $P = (S, T, A, E, M)$ formal definiert [Claus und Schwill 2006], S. 502), wobei:

- S eine nicht-leere Menge von Stellen ist
- T eine nicht-leere Menge von Transitionen ist
- $S \cap T = \emptyset$, also Elemente aus S nicht in T enthalten sind
- $A \subset S \times T$ ist die Menge der Kanten, die von Stellen zu Transitionen führen
- $E \subset T \times S$ ist die Menge der Kanten, die von Transitionen zu Stellen führen
- $M : S \rightarrow \mathbb{N}_0$ ist eine Markierungsfunktion, die den Ausgangszustand definiert - also die Anzahl der Marken in jeder Stelle

A und E können zur sogenannten Flussrelation $A \cup E = F$ zusammengefasst werden.

6.1.5 Erweiterungen von Petri-Netzen

Petri-Netze können wie folgt erweitert werden (vgl. [Bernroider und Stix 2006], S. 21):

Stellen in Petri-Netzen können theoretisch unendlich viele Marken besitzen. Kanten können nur jeweils eine Marke transportieren. Durch Gewichtung können Kapazitäten und Durchfluss festgelegt werden. Werden Stellen mit einem Gewicht versehen, wird deren maximale Kapazität festgelegt. Analog dazu wird durch Gewichtung von Kanten die Anzahl der zu transportierenden Marken festgelegt.

Ein Petri-Netz, in dem alle Stellen und Kanten mit eins gewichtet sind, wird *BE-Netz* (Bedingungs-Ereignis-Netz) genannt. Ein solches Netz eignet sich zur Modellierung logischer Schaltungen.

Eine weitere Variation ergibt sich, wenn die Marken voneinander unterschieden werden (Dokument, Person, usw.) oder ihnen bestimmte Attribute (Gewicht, Preis, Kosten, usw.) zugewiesen werden. Ein solches Netz wird *IM-Netz* (Individuelle-Marken-Netz) genannt.

6.1.6 Praktische Bedeutung von Petri-Netzen

Petri-Netze bauen auf der Graphentheorie auf und bilden die Basis der meisten Prozessmodellierungssprachen. Beispielsweise basiert die Ereignisgesteuerte Prozesskette (EPK) auf Petri-Netzen, allerdings sind Ereignisgesteuerte Prozessketten weit weniger formal definiert und eignen sich nicht für die Beschreibung exakter Prozessabläufe (vgl. [Bernroider und Stix 2006], S. 26). Auch bei den Aktivitätsdiagrammen von UML2 finden sich Konzepte aus den Petri-Netzen wieder.

Petri-Netze bieten Möglichkeiten, bestimmte Aussagen über das Netz mathematisch zu beweisen. Es kann beispielsweise bewiesen werden, ob ein Netz einen toten Zustand erreichen kann oder nicht. Durch ihre Granularität und die Art der Darstellung werden Petri-Netze allerdings schnell unübersichtlich und sind für Personen, die mit den Konzepten nicht vertraut sind, wenig intuitiv (vgl. [Bernroider und Stix 2006], S. 22).

6.1.7 Beispiel für das Petrinetz *PatientInnenversorgung in einem Krankenhaus*

Der im Kapitel 1.4, S.6 vorgestellte Usecase einer Patientenversorgung in einem Krankenhaus wird in der Abbildung 6.13 als Petrinetz dargestellt. Die Vor- und Nachbedingungen sind nicht modelliert, da es hierfür keine Entsprechung in Petrinetzen gibt. Es ist ein Token in der Stelle ”Erstdiagnose” dargestellt, der nach Beendigung von der folgenden Transition konsumiert wird.

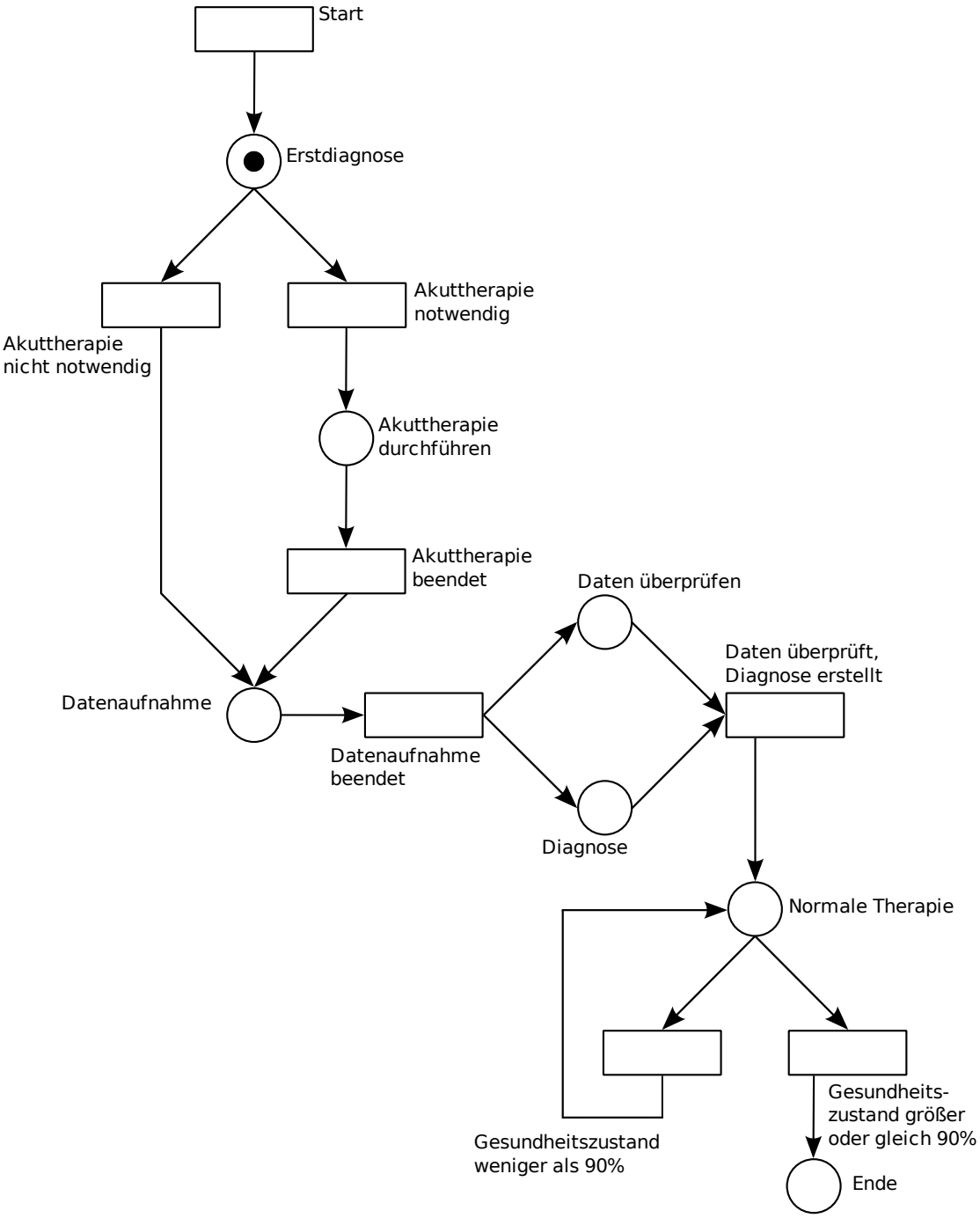

Abbildung 6.13: Beispiel für das Petrinetz *PatientInnenversorgung in einem Krankenhaus*

6.2 UML Aktivitätsdiagramme

Mit Aktivitätsdiagrammen kann das dynamische Verhalten von Anwendungsfällen modelliert werden. Aktivitäten beschreiben Aktionen, die in nebenläufigen Handlungssträngen in einem gerichteten Graph organisiert sind, ihre Synchronisation sowie gegebenenfalls beteiligte Objekte. Neben der Modellierung von Abläufen in Softwaresystemen eignen sich Aktivitätsdiagramme insbesondere auch für die Beschreibung und Analyse von realen Geschäftsprozessen (vgl. [Bernroider und Stix 2006], S.79).

In UML1 waren Aktivitätsdiagramme Spezialfälle von Zustandsdiagrammen und dadurch in der Ausdrucksstärke zur Modellierung von Abläufen beschränkt.
In UML2 wurden die Metamodelle von Zustandsdiagrammen und Aktivitätsdiagrammen getrennt, wobei Aktionen weiterhin als gemeinsame Basis verwendet werden. Aktivitätsdiagramme in UML2 basieren auf den Konzepten von Petri-Netzen und sind durch moderne Geschäftsprozess-Modellierungssprachen beeinflusst (vgl. [Rumbaugh u. a. 2005], S.157).

6.2.1 Activities

Eine Aktivität ist ein Verhalten, dass sich aus mehreren Aktionen zusammensetzt. Eine Aktion ist eine Tätigkeit, die im Diagramm nicht weiter zerlegt wird (vgl. [Pilone und Pitman 2005], S.104).

Wenn eine Aktivität das Verhalten eines *classifiers* [1] beschreibt, wird der Classifier als der *Kontext* der Aktivität bezeichnet. Die Aktivität hat dann Zugriff auf alle Attribute und Methoden des Classifiers. Wenn Business-Prozesse modelliert werden, werden diese Daten als prozessrelevante Daten bezeichnet (vgl. [Pilone und Pitman 2005], S.105).

Es gibt drei Möglichkeiten, wie Aktivitäten auftreten können (vgl. [Hitz u. a.

[1] Ein *classifier* ist eine Abstrakte Basisklasse im Metamodell von UML, von der viele andere Klassen (über eine spezialisierte Classifier-Hierarchie) abgeleitet werden. *Classifier* beschreiben strukturelle und verhaltensmäßige Eigenschaften wie Namensraum, Typ, Operationen und Attribute. Die Metaklasse *class*, die in Klassendiagrammen zum Einsatz kommt, ist ein Beispiel für einen Classifier (vgl. [Hitz u. a. 2005], S.390 und [Rumbaugh u. a. 2005], S.222).

2005], S.188):

Im Kontext eines Classifiers, als Methode des Classifiers: Die Aktivität wird dann ausgeführt, wenn die entsprechende Methode des Classifiers aufgerufen wird.

Im Kontext eines Classifiers, diesem direkt zugeordnet: Die Ausführung der Aktivität beginnt mit der Instantiierung des Classifiers. Am Ende der Aktivität wird das Classifier-Objekt gelöscht. Umgekehrt, wenn das Classifier-Objekt vor dem Ende der Aktivität gelöscht wird, wird die Aktivität unterbrochen.

Ohne Zuordnung zu einem Classifier: Die Aktivität kann auch "autonom" definiert werden, ohne einem Classifier zugeordnet zu sein. In diesem Fall muss die vordefinierte Aktion *CallBehaviorAction* verwendet werden, um die Aktivität zu starten.

Zu einer Aktivität können die involvierten Eingabeparameter angegeben werden (vgl. [Pilone und Pitman 2005], S.105) sowie Vor- und Nachbedingungen definiert werden, die jeweils vor Ausführung oder bei Beendigung gelten müssen. Diese Bedingungen werden mit den Schlüsselwörtern *precondition* und *postcondition* notiert, denen jeweils die Definition der Bedingung als Pseudocode, *Object Constraint Language* (OCL) Ausdruck [2] oder beschreibender Bedingungstext folgt. Die UML Spezifikation schreibt hier keine Form für die Notation der Bedingung vor, da dies als Implementierungsdetail angesehen wird (vgl. [Pilone und Pitman 2005], S.106-107). Abbildung 6.14 stellt eine Aktivität mit zwei Eingangsparametern, einem Ausgangsparameter, einer Precondition und einer Postcondition dar.

Weiters bilden Aktivitäten einen Namensraum, der die Sichtbarkeit der Elemente der Aktivität beschränkt (vgl. [Hitz u. a. 2005], S.189).

[2] Die *Object Constraint Language* (OCL) ist eine Spezifikation der OMG, die in Verbindung mit UML eine Möglichkeit bietet, in Modellen Bedingungen und Logiken auszudrücken. OCL wurde bereits mit UML 1.4 eingeführt. Mit der UML 2.0 Spezifikation wurde die OCL 2.0 Spezifikation auf Basis von MOF bzw. UML formal definiert. Bestandteile von OCL sind Typen (Boolean, Integer, Real, String und jeder Classifier des betroffenen UML Modells), Operatoren mit Rangfolge, If-Else-Bedingungen, Variablendefinitionen usw. OCL ist aber eine *Query-Only* Sprache und kann keine Kontrollflüsse definieren, keine Programmlogik ausdrücken und das Modell nicht verändern (vgl. [Pilone und Pitman 2005], S.192-200).

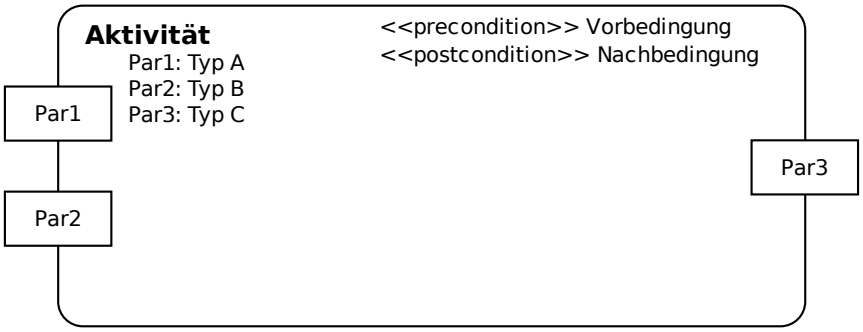

Abbildung 6.14: Aktivität mit Parametern und Bedingungen

6.2.2 Actions

Aktionen sind Aktivitätsknoten und die elementaren Bausteine einer Aktivität. Sie repräsentieren die eigentlichen Tätigkeiten. Eine Aktion kann als eine der vordefinierten UML Aktionen modelliert werden (siehe weiter unten) oder selbst in Pseudocode bzw. in der Implementierungssprache definiert werden.

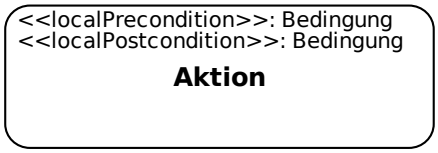

Abbildung 6.15: Aktion mit Vor- und Nachbedingungen

Wie auch für Aktivitäten können für Aktionen Vor- und Nachbedingungen definiert werden. Diese werden durch die Schlüsselwörter *localPrecondition* und *localPostcondition* angegeben. Die Definition der Bedingung erfolgt gleich wie bei Aktivitäten (vgl. [Pilone und Pitman 2005], S.105-107). Abbildung 6.15 zeigt eine solche Aktion mit definierten Vor- und Nachbedingungen.

Aktionen können durch den lesenden oder schreibenden Zugriff auf Objekte bzw. durch den Aufruf anderen Verhaltens den Zustand des Systems verändern. Die Aktion steht immer im Kontext einer Verhaltensbeschreibung in Form einer Aktivität oder einer Transition in einem Zustandsdiagramm und ist nur durch diese verfügbar. Die Verhaltensbeschreibung legt fest, wann eine Aktion mit welchen Parametern ausgeführt wird.

Aktionen sind atomar, können aber unterbrochen werden wobei der Ursprungs-

zustand des Systems wie vor Beginn der Aktion wieder hergestellt wird.

Aktionen können sowohl Eingabewerte entgegennehmen und verarbeiten wie auch Ausgabewerte zur Verfügung stellen. Diese Werte werden als Kopien verarbeitet wobei Kopien von Objektreferenzen wiederum auf das Originalobjekt zeigen.

Die Ausgabewerte können entsprechend zweier Strategien zur Verfügung gestellt werden: gleichzeitig oder, gemäß ihrer Verfügbarkeit, einzeln. Welche dieser Strategien gewählt wird ist eine Frage der Implementierung. Zusätzlich wird in jedem Fall nach Beendigung der Aktion an jeder Kontrollflusskante ein Kontrolltoken angeboten.

In UML 2.0 existieren 44 vordefinierte, sprachunabhängige Aktionen. Diese können aufgrund ihrer Granularität auf eine beliebige Zielsprache abgebildet werden. Mithilfe der *OpaqueAction* kann eine Aktion auch direkt in der Implementierungssprache definiert werden.

Für die meisten vordefinierten Aktionen gibt es keine eigenen Notationsregeln. Es können beliebige Namen in die Aktionsrechtecke geschrieben werden. Damit eine Zuordnung vom anwendungsspezifischen Aktionsnamen auf die jeweilige konkrete vordefinierte Aktion erfolgen kann, wird empfohlen, eigene Konventionen zu erstellen und diese konsequent zu befolgen (bsp. *Dokument anlegen* für *CreateObjectAction*).

Die vordefinierten Aktionen werden in folgende Kategorien eingeteilt:

Kommunikationsbezogene Aktionen: Übermitteln von Objekten und Signalen, Aufrufen von Verhalten und Operationen, Empfang von Ereignissen.

Objektbezogene Aktionen: Erzeugen und Löschen von Objekten, Aufruf von Objektverhalten, Reflexionsfunktionen.

Strukturmerkmals- und variablenbezogene Aktionen: Setzen und Löschen einzelner oder aller Werte von strukturellen Merkmalen und Variablen.

Linkbezogene Aktionen: Erzeugen und Löschen von Links und Navigation auf Basis von Links.

Für die Beschreibung der einzelnen Aktionen siehe Hitze et al. (vgl. [Hitz u. a. 2005], S.222 ff.)

6.2.3 Object Nodes

Ein Objektknoten stellt die Existenz eines Objektes dar, das von einer Aktion produziert wird und von einer anderen konsumiert wird (vgl. [Rumbaugh u. a. 2005], S.487). Sie können auch Signale repräsentieren (vgl. [Pilone und Pitman 2005], S.113).

Für Objektknoten kann angegeben werden, in welchem Status sich das Objekt im Objektknoten befinden muss, von welchem Typ die Objekte sein müssen sowie die maximale Anzahl an Tokens, die in einem Objektknoten erlaubt sind (vgl. [Object Management Group 2009c], S.393).

Objektknoten sind abstrakte Klassen (vgl. [Object Management Group 2009c], S.393), die sich letztendlich in folgenden Pins spezialisieren:

Input Pin: Eingabepin für Aktionen.

Output Pin: Ausgabepin für Aktionen.

Value Pin: Werteingabepin für Aktionen. Für Wert-Pins kann ein Wert definiert werden.

Die Abbildung 6.16 zeigt verschiedene Arten von Pins. ”Aktion B” hat auf der linken Seite zwei Eingabepins, wobei der obere Pin ein ”Value Pin” mit dem Wert ”8” ist. Auf der linken Seite werden zwei Ausgabepins gezeigt, wobei bei dem unteren Pin eine alternative Darstellung gewählt ist. Er ist mit dem Pin der nachfolgenden Aktion zusammengefasst und wird als Objektknoten dargestellt.

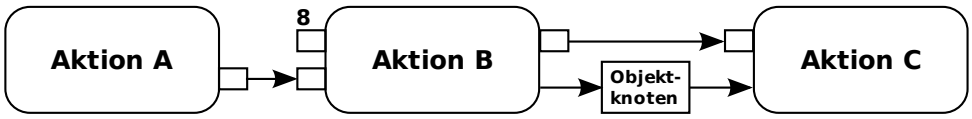

Abbildung 6.16: Aktionen mit verschiedenen Arten von Pins

6.2.4 Control Nodes

Kontrollknoten koordinieren den Kontrolltoken- und Objekttokenfluss zwischen anderen Knoten. Es können folgende Arten von Kontrollknoten unterschieden

werden (vgl. [Rumbaugh u. a. 2005], S.292-296):

Decision: Ein Entscheidungsknoten hat eine eingehende und mehrere ausgehende Kanten und steuert den Fluss aufgrund von Kantenbedingungen, die auf den ausgehenden Kanten notiert werden. Eintreffende Tokens werden auf maximal eine ausgehende Kante weitergeleitet, auch wenn die Kantenbedingungen für mehrere ausgehende Kanten zutreffen. Auf einer Kante kann die spezielle Kantenbedingung ”else” notiert werden, deren Pfad traversiert wird, wenn keine der anderen Bedingungen erfüllt werden konnte (siehe Abbildung 6.17).

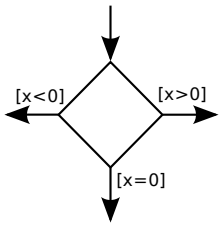

Abbildung 6.17: Decision Node mit Bedingungen

Merge: Ein Zusammenführungsknoten führt ein oder mehrere alternative Pfade zusammen. Wenn auf einer der eingehenden Kanten Tokens bereitstellt sind, werden diese auf die ausgehende Kante weitergeleitet. Es findet also keine Synchronisation statt. Die Tokens werden aber auch nicht vereinigt, sondern einzeln weitergeleitet, wodurch sich die Anzahl nebenläufiger Tokens nicht reduziert (siehe Abbildung 6.18).

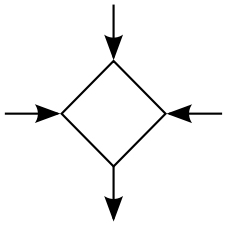

Abbildung 6.18: Merge Node

Fork Node: Dieser Knoten hat eine eingehende Kante und mehrere ausgehende Kanten. Wenn ein Token auf der eingehenden Kante bereitgestellt wird, wird dieser auf alle ausgehenden Kanten kopiert. Ein *fork node* erhöht somit die Anzahl der nebenläufigen Tokens (siehe Abbildung 6.19).

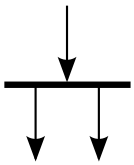

Abbildung 6.19: Fork Node

Join Node: Ein Synchronisationsknoten hat mehrere eingehenden Kanten und eine ausgehende Kante. Stehen an allen eingehenden Kanten Tokens zur Verfügung, werden diese zusammengeführt und ein Token für die ausgehende Kante produziert. Somit reduziert dieser Knotentyp die Anzahl nebenläufiger Tokens (siehe Abbildung 6.20).

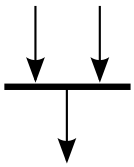

Abbildung 6.20: Join Node

Initial Node: Ein Startknoten hat keine eingehenden Kanten und mindestens eine ausgehende Kante. Ein Startknoten ist der standardmäßige Startpunkt für eine Aktivität. Wenn eine Aktivität gestartet wird, werden die *initial nodes* aktiviert und produzieren für jede ausgehende Kante ein Token. Aktivitäten können aber auch anders gestartet werden, beispielsweise durch Bereitstellung von Tokens an Aktivitätseingangsparameter (vgl. [Weilkiens und Oestereich 2004], S.93). Siehe Abbildung 6.21.

Abbildung 6.21: Initial Node

Activity Final Node: Ein Aktivitätsendknoten hat eine oder mehrere eingehende Kanten aber keine ausgehende Kante. Der Knoten wird aktiviert, sobald ein Token auf einer der eingehenden Kanten eintrifft (Oder-Semantik) und die gesamte Aktivität wird terminiert, unabhängig davon, wieviele aktive Tokens sich noch in der Aktivität befinden (vgl. [Weilkiens und Oestereich 2004], S.93-94). Alle Rückgabewerte, die die Aktivität definiert hat, werden in einem Paket zusammengefasst und zurückgegeben (siehe Abbildung 6.22).

Abbildung 6.22: Final Node

Flow Final Node: Ein Ablaufendknoten hat eine oder mehrere eingehende aber keine ausgehenden Kanten. Alle Tokens, die diesen Knoten erreichen, werden konsumiert und zerstört. Die Aktivität wird dadurch nicht beendet (siehe Abbildung 6.23).

Abbildung 6.23: Flow Final Node

Conditional Node: Ein *conditional node* besitzt eine oder mehrere eingehende Kanten, eine oder mehrere ausgehende Kanten und zwei oder mehrere *clauses* (Bedingungen). Für diesen Knoten gibt es keine graphische Notation. Die Bedingungen bestehen aus einem Test, der evaluiert wird, und einem Codeteil, der ausgeführt wird. Wird der Knoten aktiviert, indem alle eingehenden Kanten Tokens zur Verfügung stellen, und werden die Tests von einem oder mehreren *clauses* erfüllt, wird eine dieser *clauses* ausgeführt. Die Wahl ist nicht deterministisch. Der Rückgabewert wird für die Produktion eines Tokens verwendet (vgl. [Rumbaugh u. a. 2005], S.275-276).

Loop Node: Ein Schleifenknoten wird solange eine Bedingung erfüllt wird ausgeführt. Für diesen Knoten gibt es keine graphische Notation.

6.2.5 Activity Edges

Kanten stellen Sequenz-Beziehungen zwischen zwei Knoten dar, die auch Daten beinhalten können. Eine Kante hat immer eine Quelle *source* und ein Ziel *target*. Der Zielknoten kann solange nicht ausgeführt werden, bis der Quellknoten die Ausführung beendet hat, ein Token bereitgestellt hat, dieses die Kante traversieren kann und alle anderen Bedingungen des Quellknotens ebenfalls erfüllt werden. Kanten können boolesche Bedingungen (*guards*) zugewiesen werden, die erfüllt werden müssen, damit die Kante traversiert werden kann. Weiters kann ein Gewicht definiert werden, welches die minimale Anzahl an Tokens angibt, die über die Kante zum selben Zeitpunkt fließen müssen.

Das abstrakte Konstrukt Aktivitätskante wird in zwei Ableitungen spezialisiert:

Control Flow: Über eine Kontrollflusskante können Kontrolltokens fließen. Sie können nicht mit *object nodes* verbunden werden.

Object Flow: Über eine Objektflusskante können Objekttokens fließen. Sie müssen mit *object nodes* verbunden werden.

6.2.6 Tokens

Tokens sind Steuerungsmarker, die von Aktivitätsknoten erzeugt und verbraucht werden können. [Weilkiens und Oestereich 2004] unterscheiden zwischen Objekt- und Kontrolltokens, wobei Objekttokens zusätzlich zur Steuerungsmarkierung weitere Informationen in Form von *classifiers* oder beliebigen anderen Daten beinhalten können. Das Vorhandensein von zusätzlichen Informationen wird durch Objektknoten dargestellt, also Ein- und Ausgangs-Pins an Aktionen bzw. Objektknoten zwischen Aktionen und Ein- und Ausgabeparametern an Aktivitäten (vgl. [Weilkiens und Oestereich 2004], S.90).

Tokens definieren den Zustand der Aktivität. Das Konzept der Tokens ist der Petri-Netz-Theorie entlehnt. In einem Aktivitätsdiagramm mit Nebenläufigkeit können mehrere Tokens gleichzeitig vorhanden sein (vgl. [Rumbaugh u. a. 2005], S.654).

Tokens werden von Objektknoten aufgenommen bzw. an Eingabeparameter-Pins der Aktionen bereitgestellt. Aktionen werden erst dann ausgeführt, wenn genügend Tokens an allen Eingabe-Pins zur Verfügung stehen. Für Pins kann hierbei eine Zustandsbedingung (*guard condition*) definiert werden, die erfüllt sein muss damit das Token aufgenommen wird, andernfalls wird es zerstört. Die Zustandsbedingung wird als als boolescher Ausdruck in eckigen Klammern unter dem Namen des Parameter-Pins angegeben (vgl. [Hitz u. a. 2005], S.190 und [Pilone und Pitman 2005], S.111).

Tokens werden über Aktivitätskanten zwischen Aktivitätsknoten weitergegeben. An diesen Kanten kann ein Gewicht definiert werden, dass die minimale Anzahl der vorhandenen Tokens definiert. Das Gewicht kann den speziel-

len Wert *all* haben, durch den alle verfügbaren Tokens ohne minimale Einschränkung vom Aktivitätsknoten konsumiert werden (vgl. [Rumbaugh u. a. 2005], S.680-681 und [Pilone und Pitman 2005], S.111).

Tokens werden von Aktivitätsknoten vor deren Ausführung verbraucht. Nach Beendigung der Ausführung werden neue Tokens erzeugt und an den Ausgangspins bereitgestellt (vgl. [Rumbaugh u. a. 2005], S.655).

Für Tokens gelten zusammengefasst folgende Regeln:

- Eine Aktion startet erst, wenn an allen eingehenden Kanten genügend Tokens zur Verfügung stehen. Das bewirkt eine implizite Synchronisation und entspricht einer Und-Logik.
- Ist eine Aktion beendet, werden an allen ausgehenden Kanten Tokens bereitgestellt. Das bewirkt eine implizite Aufspaltung und entspricht einer Und-Logik.
- Ein Token kann von einer Aktion zur nächsten fließen, wenn an der vorherigen Aktion ein Token zur Verfügung steht, die folgende Aktion bereit ist das Token aufzunehmen und die Anzahl der Tokens mindestens dem Kantengewicht entspricht.

6.2.7 Anwendung von Aktivitätsdiagrammen

Mit der UML Version 2 wurden Aktivitätsdiagramme stark überarbeitet, was vor allem ihren Einsatz zum Modellieren von Geschäftsprozessen erleichtert hat. Mit Aktivitätsdiagrammen kann beliebiges Ablaufverhalten modelliert werden. Die Einbettung in den UML Standard und Interoperabilität mit anderen Diagrammarten aus der UML machen Aktivitätsdiagramme zu einen mächtigen und flexiblen Werkzeug.

Durch die Flexibilität und Vielseitigkeit von Aktivitätsdiagrammen ergeben sich auch deren Schwächen. Der Modellierungsvorgang kann relativ aufwändig sein, da oft eine Vielzahl an Elementen verwendet und konfiguriert werden muss, um bestimmte, vergleichsweise einfache Ziele zu erreichen. Die Ausdrucksstärke der visuellen Darstellung ist gegenüber spezialisierten Diagrammarten, wie beispielsweise die Business Process Modeling Notation, die im

nächsten Kapitel vorgestellt wird, vergleichsweise geringer.

6.2.8 Beispiel für die Aktivität *PatientInnenversorgung in einem Krankenhaus*

Abbildung 6.24 zeigt den im Kapitel Kapitel 1.4, S.6 vorgestellten Usecase für eine PatientInnenversorgung in einem Krankenhaus. Es wurden keine Pins oder Eingabeparameter notiert, obwohl dies für eine solche Aktivität für den Zugriff auf den/die PatientIn oder die Krankenakte sinnvoll sein kann. Die Aktivität und alle in ihr enthaltenen Elemente haben Zugriff auf den/die PatientIn, wenn die Aktivität im Kontext einer Klasse mit entsprechenden Attributen für eine/n Patient/In aufgerufen wird. Wenn der/die PatientIn die Krankenakte besitzt, kann über den/die PatientIn auf diese zugegriffen werden. In dem Fall müssen keine Pins zwischen Datenaufnahme und Datenüberprüfung notiert werden. Bei der Verwendung von Pins müsste entweder die Aktion ”Diagnose” ebenfalls einen eingehenden Objektfluss enthalten oder auf den Fork Knoten verzichtet werden, da alle ein- und ausgehenden Kanten eines Fork Knoten vom selben Typ sein müssen. Weitere Bedingungskriterien können in der *UML Superstructure Specification* nachgelesen werden (vgl. [Object Management Group 2009c], S.295ff.)

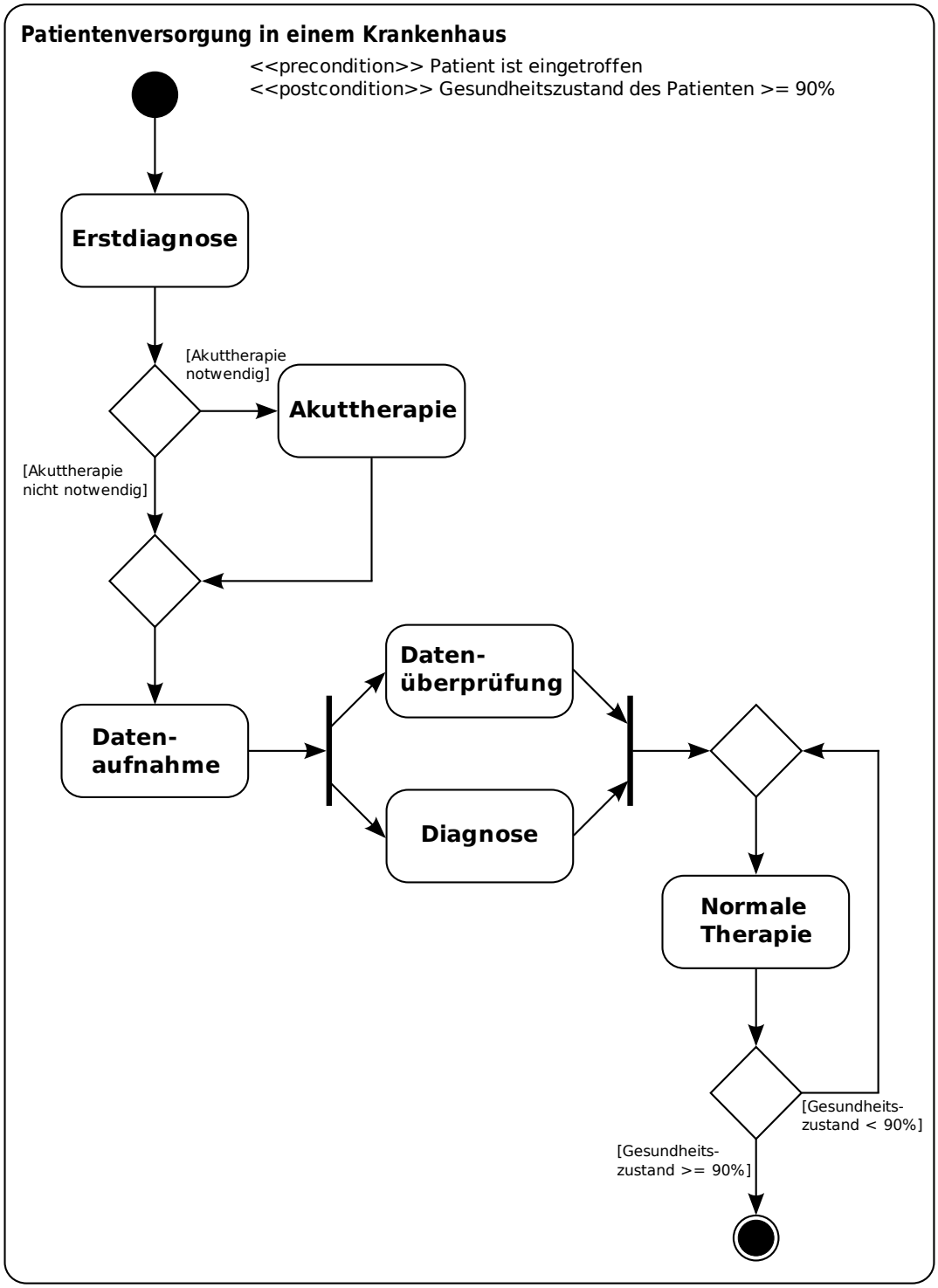

Abbildung 6.24: Beispielsaktivität *PatientInnenversorgung in einem Krankenhaus*

6.3 Business Process Modeling Notation

6.3.1 Hintergrund

Die Business Process Modeling Notation (BPMN) ist eine graphische Notationssprache zur Beschreibung von Geschäftsprozessen.

Das Ziel der Entwicklung des BPMN Standards ist eine allgemein verständliche Notation von Geschäftsprozessen zu bieten, die als gemeinsame Sprache zwischen verschiedenen Organisationseinheiten wie Technik und Betriebswirtschaft in einem Unternehmen dienen kann und somit Lücken zwischen Prozess-Analyse und Definition und Implementierung schließt.

Ein weiteres Ziel der BPMN ist die Bereitstellung einer Möglichkeit zur Visualisierung von XML basierten Prozessausführungssprachen wie die *Web Services Business Process Execution Language* (WS-BPEL) (vgl. [Object Management Group 2009a], S.1). Die BPMN Spezifikation beschreibt auch ein informelles Mapping von BMPN zu WS-BPEL (vgl. [Object Management Group 2009a], S.143ff).

Die BPMN wurde der Öffentlichkeit im Mai 2004 von der *Business Process Management Initiative* (BPMI) in der Version 1.0 präsentiert (vgl. [White 2004a], S.1). Im Juni 2005 haben die BPMI und die *Object Management Group* (OMG) die Zusammenlegung ihrer Aktivitäten im Bereich des *Business Process Management* in der *Business Modeling & Integration* (BMI) *Domain Task Force* (DTF) verkündet. Die Entwicklung der BPMN wird nun von der OMG koordiniert. Der aktuelle Standard trägt die Version 1.2 und ist im Jänner 2009 erschienen.

Als ein Grund für den Zusammenschluss wird das Fehlen eines Meta-Metamodells für den Standard BPMN genannt, was eine Reihe praktischer Probleme (Speicherung, Datenaustausch, Transformation, Versionierung) mit sich bringt, die die OMG mit MOF-basierten Sprachen schon gelöst hat (vgl. [Gilbert 2007]). Es existiert noch kein formalisiertes Meta-Metamodell für die BPMN. Der im November 2008 verabschiedete Standard *Business Process Definition Meta Model* (BPDM) soll dieses Problem beheben. BPMN beschreibt ein Metamo-

dell und Serialisierungs-Modell für BPMN und basiert selbst auf dem OMG Standard MOF (vgl. [Object Management Group 2008a], S.1).

Die Standards BPDM und BPMN sollen in der kommenden Version 2.0 des BPMN Standards zusammengeführt werden (vgl. [Gilbert 2007]). Von Juni 2007 bis Februar 2008 wurde ein Request for Proposal (RFP) ausgerufen (vgl. [Object Management Group 2007]). Trotz der Ähnlichkeiten zwischen Aktivitätsdiagrammen und der BPMN sieht das RFP keine Angleichung oder Zusammenführung der beiden Standards vor (vgl. [Object Management Group 2007], S.23).

6.3.2 Tokens als Zustandsmarker

Wie in den UML Aktivitätsdiagrammen werden bei der BPMN Tokens als Zustandsmarker verwendet. Ein Token ist dabei ein beschreibendes Konstrukt und wie bei den UML Aktivitätsdiagrammen kein Modellelement. Tokens haben eine eindeutige Identität, damit multiple Tokens voneinander unterschieden werden können (vgl. [Object Management Group 2009a], S.290).

6.3.3 Flow Objects

Event

Ein Event ist ein Ereignis, das während des Ablaufs eines Geschäftsprozesses eintreten kann oder ausgelöst wird. Im Sinne der BPMN werden nur jene Events betrachtet, die den Prozessfluss oder den zeitlichen Ablauf beeinflussen. Events können eine Ursache (engl. *Trigger*) haben und empfangen (engl. *Catching*) werden, wie zum Beispielder Start eines Prozesses bei Empfang einer Nachricht. Andererseits können Events auch ein Ergebnis (engl. *Result*) produzieren indem Sie einen Event auslösen (engl. *Throwing*), wie zum Beispiel das Senden einer Nachricht bei Beendigung eines Prozesses. Es werden drei Typen von Events unterschieden: *Start*, *Intermediate* und *End*. Start Events beginnen einen Prozessfluss und können nur Ereignisse empfangen,

Intermediate Events können während des Prozessablaufs passieren und Ereignisse empfangen oder senden und End Events beenden den Prozessfluss und können nur Ergeignisse senden. Events werden durch einen Kreis dargestellt, wobei in der Mitte des Kreises ein Zeichen platziert werden kann, das eine Unterscheidung verschiedener Eventarten erlaubt. Die Stärke des Striches gibt hierbei an, ob es sich um Start, Intermediate oder End Events handelt (vgl. [White 2004a], S.2 und [Object Management Group 2009a], S.35ff.)).

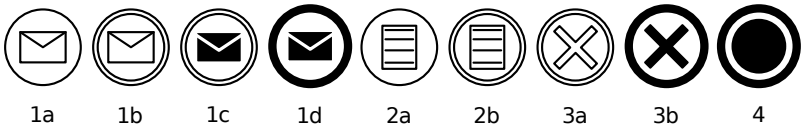

Abbildung 6.25: Verschiede Arten von Events

In der Abbildung 6.25 werden verschiedene Arten von Events aufgelistet. Events 1a - 1d sind Events, die Nachrichten senden und Empfangen können. 1a ist hierbei ein Start Event, 1b ein Intermediate Catching, 1c ein Intermediate Throwing und 1d ein End Event. Die Events 2a und 2b sind *Conditional Events*, die auf geänderte Geschäftsbedingungen reagieren, wobei 2a ein Start und 2b ein Intermediate Catching Event ist. Der Event 3a ist ein Intermediate Catching Event und reagiert auf abgebrochene Transaktionen, während 3b ein End Event ist und den Abbruch von Transaktionen anstößt. Der Event 4 ist ein End Event, der den sofortigen Abbruch des Prozesses bewirkt.

Activity

Eine Aktivität ist die Arbeit, die in einem Geschäftsprozess verrichtet wird. Der Start von Aktivitäten wird nicht nur durch eingehende Tokens angestoßen, sondern kann auch von Events, Nachrichten oder anderen Faktoren beeinflusst werden (vgl. [Object Management Group 2009a], S.98). Es können folgende Typen von Aktivitäten unterschieden werden (vgl. [Object Management Group 2009a], S.52):

Process Ein Prozess hat in der BPMN keine eigene graphische Darstellung, sondern setzt sich aus mehreren Flow Objects zusammen. Er ist also die Gesamtheit des dargestellten Ablaufs.

Sub-Process Ein Subprozess ist eine zusammengesetzte Aktivität und wird als abgerundetes Rechteck mit einem Plus-Zeichen dargestellt. Vom Betrachtungspunkt des Subprozesses selbst ist dieser einem Prozess gleichzusetzen (vgl. [Object Management Group 2009a], S.56). Siehe Abbildung 6.26.

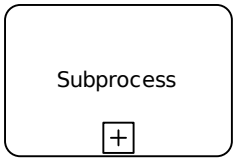

Abbildung 6.26: Sub-Process

Task Ein Task ist eine atomare Aktivität, die in einem Prozess verrichtet wird. Tasks werden verwendet, wenn die Arbeit in einem Prozess nicht auf eine feinere Granularität herunter gebrochen wird. In der Regel verrichtet eine Person, eine Maschine oder ein Programm die Arbeit, wenn der Task ausgeführt wird. Es werden drei Kategorien unterschieden: *Loop*, *Multi-Instance Loop* und *Compensation*. Ein Task wird als abgerundetes Rechteck dargestellt, wobei ein Zeichen die Kategorie angeben kann (vgl. [Object Management Group 2009a], S.64). Siehe Abbildung 6.27.

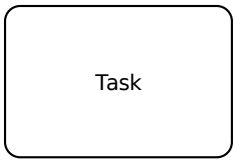

Abbildung 6.27: Task

Gateway

Gateways steuern den Fluss in Prozessen indem sie ihn aufspalten oder zusammenführen. Ein Gateway wird als Raute dargestellt. Um die unterschiedlichen Arten voneinander unterscheiden zu können, werden entsprechenden Zeichen verwendet (vgl. [Object Management Group 2009a], S.70). Gateways können den Fluss aufteilen (engl. *Split*) oder zusammenführen(engl. *Merge*). Es werden folgende Gateways voneinander unterschieden:

Exclusive Gateway: *Split:* Wird das Exclusive Gateway durch einen Token einer eingehenden Kante aktiviert, kann dieser nur einen der ausgehen-

den Pfade traversieren. Die Entscheidung basiert auf Kantenbedingungen. Wenn keine Bedingungen zutreffen, kann eine als *Default* markierte ausgehende Kante (siehe Abbildung 6.33, c) traversiert werden. *Merge:* Ein Token auf einer eingehenden Kante wird auf die ausgehenden Kante weitergeleitet. Hier findet keine Synchronisation statt (vgl. [Object Management Group 2009a], S.73-80). Die Situation mehrerer gleichzeitig eintreffender Tokens wird nicht explizit beschrieben. Aufgrund der XOR Semantik ist jedoch zu erwarten, dass nur ein eintreffender Token zu einem Zeitpunkt weitergeleitet werden kann (vgl. [Object Management Group 2009a], S.75-76). Bei *Data-Based Exclusive Gateways* wird die Entscheidung aufgrund von Kantenbedingungen getroffen, die auf Prozessdaten zugreifen können (siehe Abbildung 6.28).

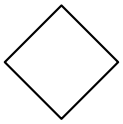

Abbildung 6.28: Databased Exclusive Gateway

Bei *Event-Based Exclusive Gateways* wird die Entscheidung durch unterschiedliche Events ausgelöst. Auf den ausgehenden Kanten dieser Entscheidungsknoten können keine Bedingungen definiert werden und es kann keine Kante als *Default* markiert werden (siehe Abbildung 6.29).

Abbildung 6.29: Eventbased Exclusive Gateway

Inclusive Gateway: *Split:* Tokens können eine oder mehrere ausgehende Kanten traversieren, basierend auf Kantenbedingungen. Eine ausgehende Kante, die als *Default* markiert ist (siehe Abbildung 6.33, c), wird nur traversiert, wenn keine anderen Bedingungen zutreffen. Kann keine Kante aufgrund der Kantenbedingungen traversiert werden, wird das Modell als invalide angesehen. Für die Semantik dieses Entscheidungsknoten gibt es eine alternative Darstellungsform als Task mit ausgehenden Entscheidungskanten (6.33, b)) . *Merge:* Es werden die Tokens der eingehenden Kanten synchronisiert, aber maximal ein Token pro eingehender Kante (vgl. [Object Management Group 2009a], S.80-83). Siehe Abbildung 6.30.

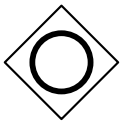

Abbildung 6.30: Inclusive Gateway

Parallel Gateway: Ermöglicht die Synchronisation und Erzeugung von parallelen Flüssen (vgl. [Object Management Group 2009a], S.85-86). Siehe Abbildung 6.31.

Abbildung 6.31: Parallel Gateway

Complex Gateway: Wird verwendet, um Situationen zu modellieren, die mithilfe der anderen Gateways nicht modelliert werden können. Es können komplexe Bedingungen auf Basis verbaler Sprache definiert werden. Als Evaluierungsgrundlage können der Status der eingehenden Kanten und Daten des Prozesses herangezogen werden. Auch können Split- und Merge-Verhalten definiert werden (vgl. [Object Management Group 2009a], S.83-85). Siehe Abbildung 6.32.

Abbildung 6.32: Complex Gateway

Inclusive- und Parallel-Gateway synchronisieren eingehende Kanten im Merge-Fall.

6.3.4 Connecting Objects

Sequence Flow

Eine Sequence Flow Kante ist eine gerichtete Kante mit einer Quelle und einem Ziel, verbindet Flow Objects und definiert so eine geordnete Sequenz

(siehe Abbildung 6.33, a).

Kanten können Bedingungen besitzen, erfüllt werden müssen, damit Token diese Kante traversieren können. Die Bedingung muss evaluiert werden, bevor der Token für die Kante produziert wird. Kanten mit Bedingungen müssen entweder Entscheidungsknoten oder Aktivitäten als Quelle haben. Im Falle von Aktivitäten kann somit die Semantik von Inclusive Gateways modelliert werden. Die Kante muss hierbei aber wie in Abbildung 6.33 b dargestellt, mit einer kleinen Raute am Quellknoten notiert werden.

Kanten können für die Entscheidungsknotensemantik der Data Based Exclusive Gateways und Inclusive Gateways als *Default* markiert werden (siehe Abbildung 6.33, c). Eine solche Kante wird dann traversiert, wenn keine anderen Bedingungen zutreffen (vgl. [Object Management Group 2009a], S.97-98). Siehe Abbildung 6.33.

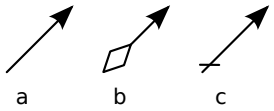

Abbildung 6.33: Sequence Flows

Message Flow

Ein Message Flow zeigt den Nachrichtenfluss zwischen verschiedenen Entitäten, die diesen Nachrichten senden oder empfangen können (vgl. [Object Management Group 2009a], S.99-100). Siehe Abbildung 6.34.

Abbildung 6.34: Message Flow

Association

Eine Assoziation verbindet Informationen und Artefakte mit Flow Objects. Gerichtete Assoziationen können Aktivitäts-Input und -Output von Datenobjekten anzeigen. Weiters werden Assoziationen für die Verbindung von Text Annotationen mit Modellelementen verwendet (vgl. [Object Management Group 2009a], S.101-102). Abbildung 6.35 a zeigt eine ungerichtete, b eine gerichtete und c eine bidirektional gerichtete Assoziation.

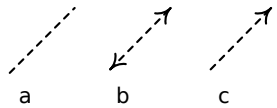

Abbildung 6.35: Assoziationen

6.3.5 Swimlanes

Mithilfe von Swimlanes können Aktivitäten unterteilt bzw. organisiert werden (vgl. [Object Management Group 2009a], S.86-91).

Pool

Ein Pool repräsentiert einen *Participant* bzw. Teilnehmer eines Geschäftsprozesses, eine spezifische Entität (zum Beispiel ein Unternehmen) oder eine allgemeine Rolle (Käufer, Verkäufer, Produzent, etc.). Ein Pool kann auch als *Black Box* ohne Flow Objects dargestellt werden. Dabei ist es aber möglich, einen Message Flow zwischen Black Boxes zu modellieren, indem die Kanten jeweils nur bis zum äußeren Rand des Pools gezeichnet werden.

Lane

Lanes sind alle weiteren Subpartitionen eines Pools und werden zur Organisation und Unterteilung von Aktivitäten verwendet. Die Semantik ist hierbei nicht vorgegeben, sondern kann selbst festgelegt werden (siehe Abbildung 6.36).

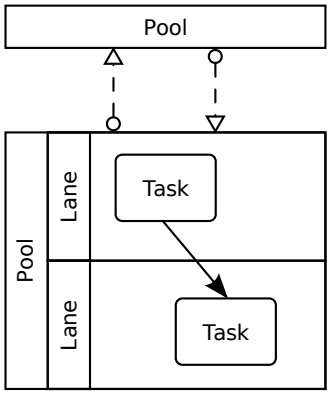

Abbildung 6.36: Pool und Lanes

6.3.6 Artifacts

Die BPMN ermöglicht mit den Artefakten, zusätzliche Informationen zu modellieren, die nicht direkt im Zusammenhang mit dem Sequence Flow oder Message Flow stehen. In der BPMN Version 1.2 werden drei Artefakte unterstützt. Toolhersteller können weitere Artefakte anbieten (vgl. [Object Management Group 2009a], S.92).

Data Object

Datenobjekte sind in der BPMN keine Flow Objects, da sie keine Auswirkungen auf den Sequence Flow oder Message Flow haben. Datenobjekte können physische wie elektronische Objekte darstellen. Sie zeigen wie Dokumente, Daten und andere Objekte während des Ablaufs verwendet und aktualisiert werden (vgl. [Object Management Group 2009a], S.93). Siehe Abbildung 6.37.

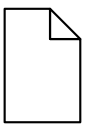

Abbildung 6.37: Data Object

Text Annotation

Mit Text Annotationen können zusätzliche Informationen zum Modell bereitgestellt werden, die bei der Interpretation des Modells behilflich sein können (vgl. [Object Management Group 2009a], S.94). Siehe Abbildung **??**.

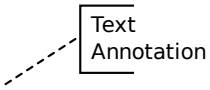

Abbildung 6.38: Text Annotation

Group

Mithilfe von Gruppen steht neben den Swimlanes ein weiterer Mechanismus zur informellen Gruppierung und Kategorisierung von Flow Objects zur Verfügung (vgl. [Object Management Group 2009a], S.95). Siehe Abbildung 6.39.

Abbildung 6.39: Group

6.3.7 Beispiel für das BPMN Diagramm *PatientInnenversorgung in einem Krankenhaus*

In Abbildung 6.40 wird der im Kapitel 1.4, S.6 vorgestellte Usecase für eine PatientInnenversorgung in einem Krankenhaus als BPMN Diagramm dargestellt. Es wurden die Zuständigkeiten, die in der Beschreibung des Usecase nicht definiert sind, exemplarisch in Pools und Swimmlanes organisiert. Die Ähnlichkeit mit der Darstellung als Aktivitätsdiagramm sind offensichtlich da viele Elemente eine gleiche oder ähnliche Darstellungsweise haben. Einerseits liegt das an der Einfachheit des Beispiels, in dem keine komplexen Geschäftsprozesse modelliert wurden, andererseits sind die Darstellungsweisen von Aktivitätsdiagrammen und BPMN Diagrammen per Definitionen und

absichtlich sehr ähnlich. Die Unterschiede liegen im Detail und werden im folgenden Kapitel diskutiert.

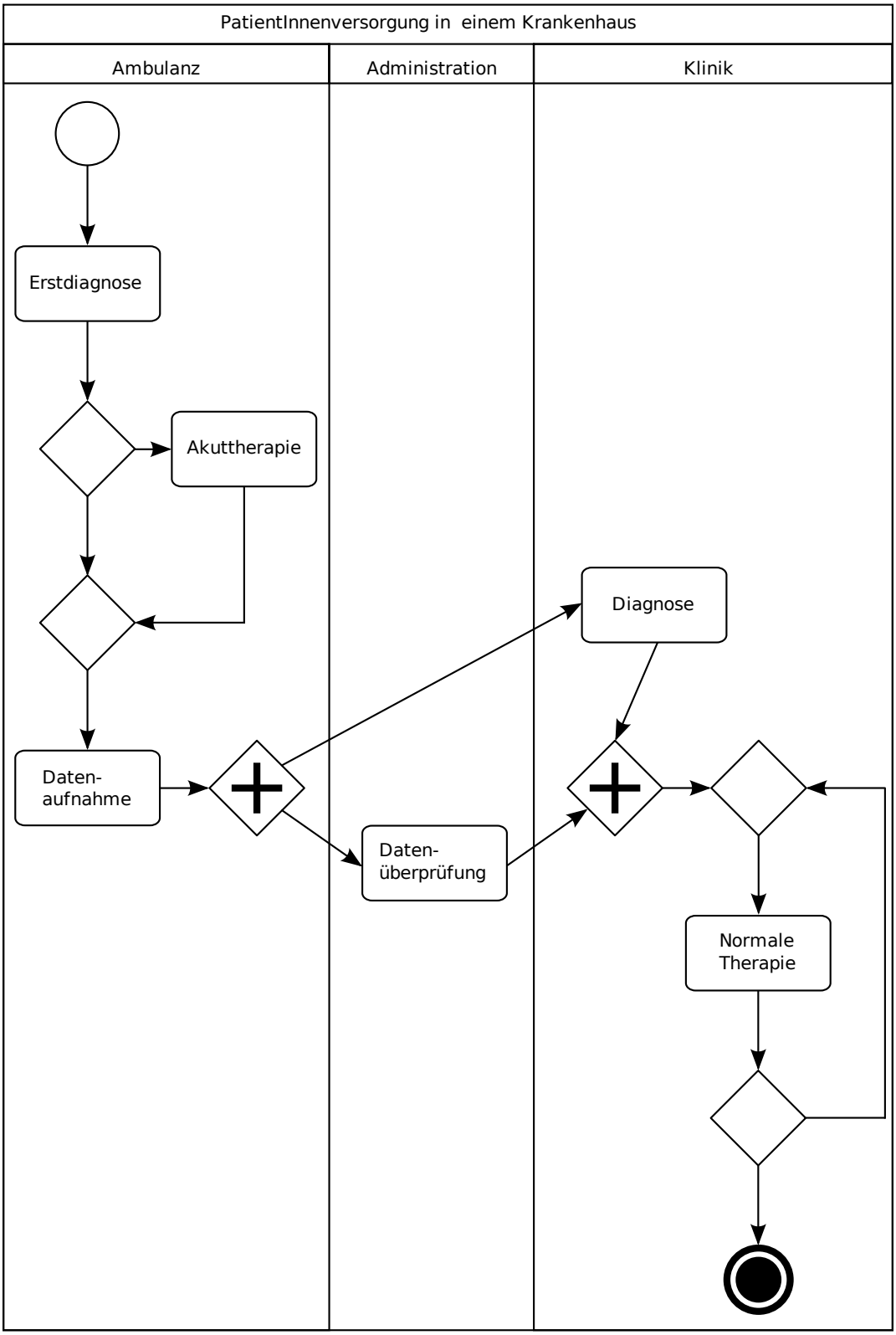

Abbildung 6.40: PatientInnenversorgung in einem Krankenhaus

6.4 Diskussion der vorgestellten Modellierungssprachen

Petrinetze können als Grundlage der meisten Prozessmodellierungssprachen angesehen werden, wie zum Beispiel die zuvor vorgestellten Aktivitätsdiagramme und BPMN. Petrinetze haben das Konzept nebenläufiger Prozesse eingeführt, die durch Tokens koordiniert werden. Sie besitzen Möglichkeiten zur Aufteilung, Synchronisation, Zusammenführung und der Wahl alternativer Pfade. Die ursprüngliche Form unterstützt aber keine typisierten Tokens. Es gibt aber zahlreiche Erweiterungen zu den Petrinetzen, die sie für bestimmte andere Anwendungsgebiete optimieren.

Aktivitätsdiagramme seit der UML Version 2 sind an Petrinetze angelehnt. Grundlegende Sprachkonstrukte wie Decision, Fork, Join und Merge lassen sich auch mit Petrinetzen umsetzen. Aktivitätsdiagramme haben aber eine große Zahl an Erweiterungen und semantischen Besonderheiten, die in dieser Form in Petrinetzen nicht existieren. Die Arbeiten von Störrle und Hausmann (Towards a Formal Semantics of UML 2.0 Activities) sowie von Schattkowsky und Förster (On the Pitfalls of UML 2 Activity Modeling) arbeiten diese Unterschiede heraus (vgl. [Störrle und Hausmann 2005] und [Schattkowsky und Förster 2007]). Die größere Vielfalt an Elementen in Aktivitätsdiagrammen ist einerseits den Anforderungen der Geschäftsprozessmodellierung, andererseits den Anforderungen von Software- und Hardwaredesign geschuldet [Schattkowsky und Förster 2007]. Die Einbettung in den gesamten UML Standard und der Anspruch eines universellen Werkzeugs zur Verhaltensmodellierung von Systemen ist eine große Stärke der Aktivitätsdiagramme. Dies bedingt aber eine große Komplexität, die eine Umsetzung des Standards in Werkzeugen erschwert. Deshalb wurden Compliance Levels von der OMG definiert, die eine Teilweise Umsetzung des Standards ermöglichen und trotzdem Kompatibilität auf dem entsprechenden Compliance Level garantieren (siehe 5.3.3, S.22).

Die Konzepte der BPMN sind denen von Aktivitätsdiagrammen sehr ähnlich. Stephen A. White hat in der Arbeit ”Process Modeling Notations and Workflow

Patterns" 21 häufig in Geschäftsprozessen auftretende Patterns analysiert und ihre Umsetzbarkeit mit BPMN und mit Aktivitätsdiagrammen untersucht ((vgl. [White 2004b]). Es konnte dabei nur ein Pattern identifiziert werden, für den es in Aktivitätsdiagrammen keine direkte Entsprechung gab, der aber durch Kombination anderer Elemente ebenso umgesetzt werden konnte. Da sich die BPMN aber nur auf das Feld der Geschäftsprozessmodellierung stützt, können viele Prozesse klarer verständlich und weniger ausführlich modelliert werden. Beide Modellierungssprachen unterstützen das Abbilden von Daten, die im Prozess verwendet werden, wobei in der BPMN diese Artefakte aber nicht den Prozessfluss steuern. Die BPMN Spezifikation sieht vor, dass Artefakte von Toolherstellern oder Anwendern erweitert werden können. In UML kann das Metamodell durch Stereotypen erweitert werden, für die auch eine graphische Darstellung durch Bilder definiert werden kann. Beide Standards unterstützen die Modellierung von Ressourcen und Zuständigkeiten durch Partitionen (UML) beziehungsweise Pool und Swimlanes (BPMN). Die Spezifikation der BPMN basiert im Gegensatz zu den Aktivitätsdiagrammen nicht auf einen formalisierten Metamodell und hat auch keinen Standard zur Serialisierung und zum Austausch von Modellen definiert. Dieser Umstand schafft Kompatibilitätsprobleme beim Austausch von Modellen zwischen Werkzeugen, die es mit UML basierten Werkzeugen nicht geben sollte. Diese Probleme wurden erkannt und werden voraussichtlich mit dem kommenden Standard der BPMN, Version 2, gelöst (vgl. [Object Management Group 2007]).

Die AMREP basiert auf dem Standard für UML Aktivitätsdiagramme, da die Interoperabilität mit anderen Diagrammarten der UML und mit UML kompatiblen Werkzeugen, das mit der MOF kompatible Metamodell der UML und die breite Anwendbarkeit für eine Vielzahl an Einsatzszenarien als wesentliche Vorteile gelten. Dennoch lag die Implementierung des gesamten Standards für Aktivitätsdiagramme außerhalb der Möglichkeiten dieser Diplomarbeit. Es wurde ein Metamodell mit einem Subset an Elementen aus dem UML Metamodell definiert, wobei für einige Elemente auch eine andere Semantik definiert wurde und Vereinfachungen getroffen wurden. Deshalb ist das AMREP Metamodell nicht vollständig kompatibel zum UML Metamodell. Welche dieser Vereinfachungen vorgenommen wurden und warum diese Entscheidungen getroffen

wurden, wird in den nächsten Kapiteln erläutert.

Teil III

Activity Model Runtime Engine für Python

Kapitel 7

Metamodell der Activity Model Runtime Engine für Python

7.1 Einführung

Dieses Kapitel stellt die Python Implementierung des UML Metamodells für die *Activity Model Runtime Engine für Python* (AMREP) vor und beschreibt die Unterschiede und Vereinfachungen, die getroffen wurden. Diese Python Implementierung des Metamodells ist der Kern der AMREP und ermöglicht die Definition von Python-Modellen, die alle die selben Eigenschaften teilen und so für die Software interpretierbar sind.

Die Entstehungsgeschichte der AMREP basiert auf der Arbeit mit UML Modellierungswerkzeugen im Allgemeinen und Überlegungen zu UML Aktivitätsdiagrammen im Besonderen, die vor allem seit der UML Version 2 das dynamische Verhalten von Systemen in vielfältigen Einsatzszenarien modellieren lässt. Das Python Metamodell der AMREP basiert somit auf dem aktuellen UML Standard für Aktivitätsdiagramme, der Version 2.2.

Das Python Metamodell der AMREP ist aber keine direkte Implementierung des UML Metamodells in Python, sondern orientiert sich daran und verwendet ein Subset der im UML Standard definierten Elemente, Attribute und Assoziationen sowie eine angepasste Semantik. Ziel war es, ein einfaches, aber

dennoch flexibles Metamodell zur Verfügung zu stellen, ohne die Gesamtheit und Tiefe der Vererbungshierarchie des UML Metamodells umzusetzen. Es basiert nicht auf einem Meta-Metamodell der Ebene *M3* (siehe Kapitel 5.2), da es zum Zweck der Erstellung einer Activity Runtime genügt, das Metamodell direkt in der Implementierungssprache Python umzusetzen, ohne Elemente einer höheren Abstraktionsebene zu instantiieren.

Die Begriffe AMREP Metamodell und `activites.metamodel` werden synonym verwendet. `activites.metamodel` ist der Name des Python-Paketes, in dem die Python Implementierung des Metamodells umgesetzt ist.

7.2 Das AMREP Metamodell im Überblick

In diesem Kapitel wird das Metamodell anhand dessen Darstellung als UML Klassendiagrammen erläutert.

Die Attribute werden unter den Klassennamen angegeben. Es existieren konkrete und abstrakte Klassen. Jede dieser Klassen besitzt das Attribut *abstract*, wobei in den konkreten Klassen `abstract = False` zugewiesen wurde und das Attribut in der UML Darstellung nicht angeführt wurde.

Die Assoziationen werden durch benannte Beziehungslinien dargestellt, wie in der UML gebräuchlich. Das Attribut, durch das die Assoziation einer Klasse abgefragt werden kann, wird jeweils am anderen Ende der Beziehungslinie angegeben.

Die Methoden der Metamodelle sind wie üblich unter den Attributen durch einen horizontalen Strich getrennt dargestellt.

7.2.1 Allgemeine Metamodell Elemente

In der Abbildung 7.1 werden die Modellelemente im Überblick dargestellt, wobei Subklassen dieser Elemente in den Folgenden Abschnitten behandelt werden.

Element ist die Superklasse aller Modellelemente und definiert deren grundlegenden Eigenschaften. Es ist von *Node* abgeleitet und erbt von dieser die Möglichkeit, hierarchische Datenstrukturen aufzubauen und auf enthaltene Elemente auf verschiedene Weise zuzugreifen. Eine oft verwendete Methode ist ein Filter, der alle Elemente, die ein bestimmtes Interface implementieren, zurück liefert. Node Elemente besitzen eine eindeutige ID, die ebenfalls für den Zugriff verwendet werden kann und einen Namen.

Element definiert die Methode *check_model_constraints()* , die von abstrakten wie konkreten Metamodellelementen verwendet wird, um Bedingungen für valide Modelle zu implementieren, wie sie von der UML Spezifikation vorgegeben werden. Es ist eine Konvention, dass in allen Modellelementen in der Methode *check_model_constraints()* die selbe Methode der Superklasse aufgerufen wird, so dass alle Modellbedingungen, die in der Vererbungshierarchie definiert sind, überprüft werden. Diese Methode wird nicht in alle Subklassen des Metamodells überschrieben.

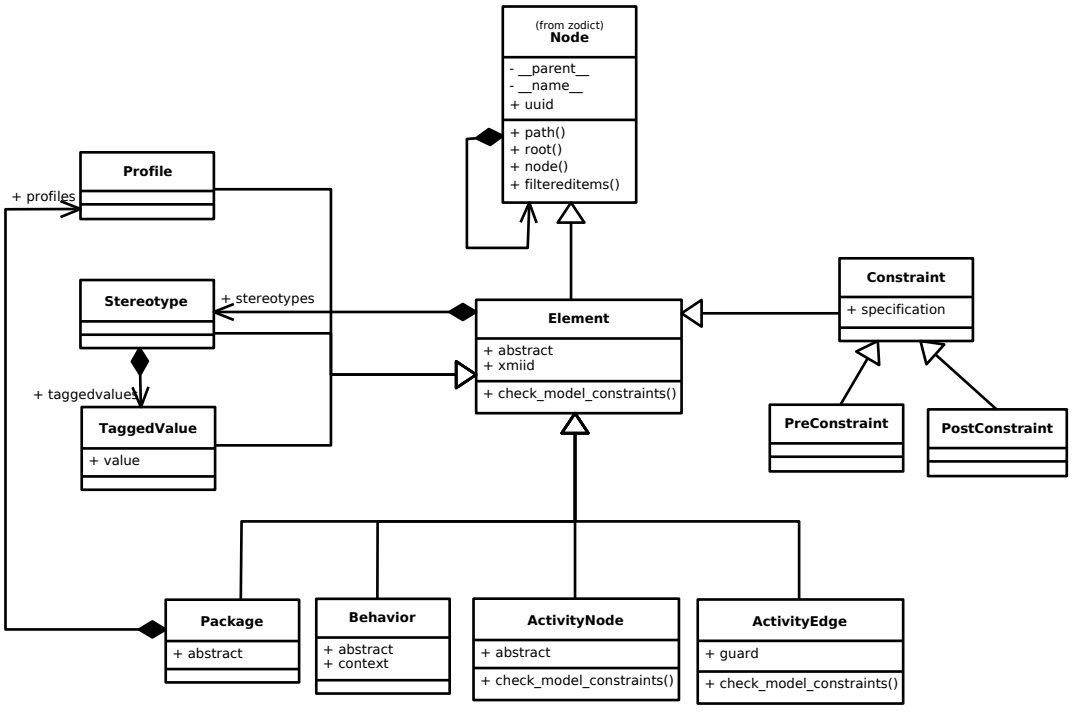

Abbildung 7.1: Allgemeine Metamodell Elemente

Das Modellelement *Constraints* erlaubt über dessen konkrete Subklassen

die Definition von Vor- und Nachbedingungen für die Modellelemente *Activity* und *Action* (siehe Kapitel **??**, S.**??**). Die Spezifikation der Bedingung wird als Python-Ausdruck im Attibut *specification* angegeben und zur Laufzeit ausgewertet. Hierbei stehen die Daten aus dem gerade aktiven Tokens zur Verfügung (siehe Kapitel 8.3, S.77). Kann die Bedingung zur Laufzeit nicht erfüllt werden, so wird das Modell als nicht valide beziehungsweise *ill formed* angesehen und ein Laufzeitfehler ausgelöst (vgl. [Rumbaugh u. a. 2005], S. 285).

Die Implementierung des UML Erweiterungsmechanismus über Profile unterscheidet sich stark vom Standard. Profile werden nicht als eigenes Metamodell zur Erweiterung von Modellen definiert, sondern direkt im Modell integriert. Diese Einschränkung macht Profile inkompatibel zum Standard. Dieses Problem wurde aber in Kauf genommen und kann in einer zukünftigen Weiterentwicklung von AMREP gelöst werden. Profile werden in AMREP dafür verwendet, um Python-Module in denen *Executions* [1] definiert sind zu laden. Zum Laden des Moduls wird hierbei der Name des Profils benutzt, der somit ein vom Python Interpreter erreichbares Modul benennen muss (siehe Kapitel 8.1, S.83). Profile sind in *Package* Elementen enthalten.

Stereotypen benennen den Namen der zu ladenden Execution und können *TaggedValues* über die Kompositionsbeziehung *taggedvalues* beinhalten. Stereotypen können für beliebige Modellelemente aufgrund der Kompositionsbeziehung *stereotypes* der Klasse *Element* definiert werden Die Runtime Engine verwendet allerdings nur Stereotypen, die für Elemente des Typs *Action* definiert wurden. Mit Hilfe von *TaggedValues* können zusätzliche statische Informationen modelliert werden, die den *Executions* zur Laufzeit übergeben werden und von diesen verwendet werden können.

[1] Executions sind in AMREP Klassen, die die Aktionsimplementierungen bereitstellen. Executions werden hierbei über ein Command Pattern von der Runtime geladen und sind dadurch von den modellierten Aktionen getrennt.

7.2.2 Aktivität

Mit der Metamodellklasse *Activity* können UML Aktivitäten modelliert werden (siehe Abbildung 7.2). Sie wird über die abstrakte Klasse *Behavior* von *Element* abgeleitet und erbt über Behavior die Möglichkeit Vor- und Nachbedingungen mithilfe der Kompositionsbeziehungen[2] *preconditions* und *postconditions* zu definieren. Die Kompositionsbeziehung *preconditions* referenziert Klassen des Typs *PreConstraint* , *postconditions* hingegen *PostConstraints* .

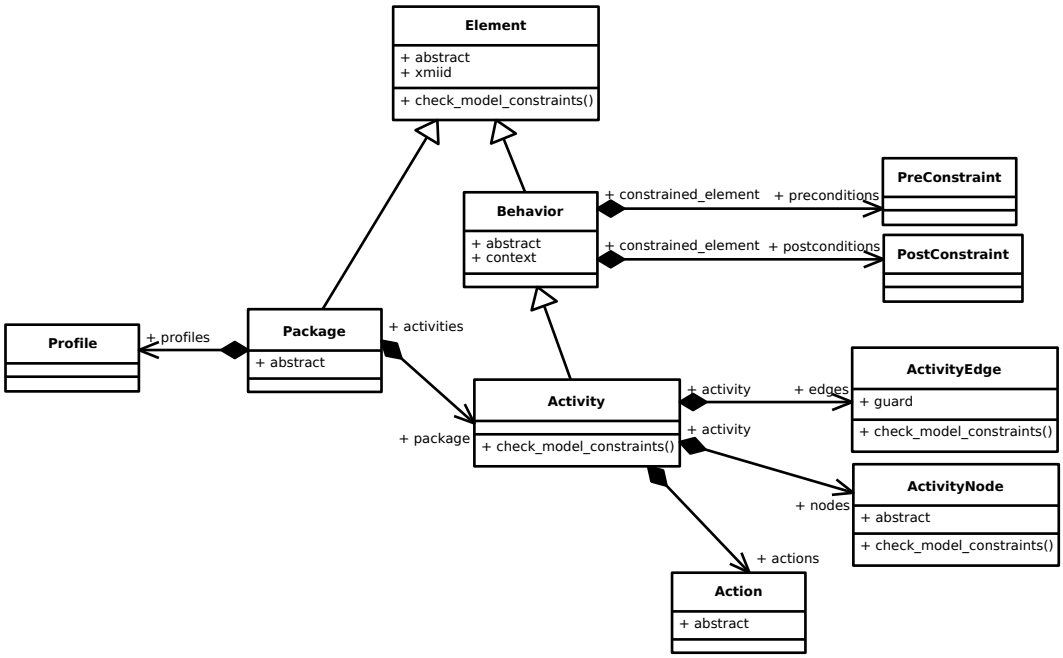

Abbildung 7.2: Aktivität, Behavior und Package

Eine Aktivität besitzt eine Menge von Knoten und Kanten, die über die Kompositionsbeziehungen *nodes* , *edges* und *actions* referenziert werden. Die Kompositionsbeziehung *actions* ist hierbei nicht im UML Standard definiert sondern wurde zum Zweck eines schnellen Zugriffs in die Python Implementierung des Metamodells für AMREP aufgenommen.

[2] Kompositionen stellen in UML Klassendiagrammen starke Beziehungen zwischen Klassen dar. Sie sind die stärkste Art von Assoziationen und erlauben das Modellieren von Teil-Ganzes Beziehungen (vgl. [Pilone und Pitman 2005], S.27). Im UML Standard werden die Beziehungen als Assoziationen angeführt. Hier werden Sie aber entsprechend der Implementierung genauer benannt.

Eine Aktivität muss immer in einem *Package* definiert sein, welches zur Gruppierung von Modellelementen dient. Ein Package kann Profile über die Kompositionsbeziehung *profiles* besitzen, die in der AMREP Implementierung zum Laden von *Executions* dienen.

7.2.3 Kanten

Da in der AMREP Implementierung keine Objektknoten existieren (siehe 7.4, S.67), gibt es auch keine Unterscheidung zwischen Kontroll- und Objektkanten. Das Element *ActivityEdge* ist somit konkret und direkt instantiierbar. Kanten verbinden Konten über eine Quelle (Assoziation *source*) und ein Ziel (Assoziation *target*). Die Assoziation ist beidseitig navigierbar, wodurch für Knoten die eingehenden (Assoziation *incoming_edges*) und ausgehenden (Assoziation *outgoing_edges*) Kanten abfragbar sind. Über die Kompositionsbeziehung *edges* von Aktivitäten können alle in ihr definierten Kanten abgefragt werden. Umgekehrt ist für eine Kante die zugehörige Aktivität über das Attribut *activity* bekannt (siehe Abbildung 7.3).

Das Attribut *guard* erlaubt die Definition einer Kantenbedingung, die zur Laufzeit erfüllt werden muss, damit Tokens über diese Kanten traversieren können. Die Bedingung wird als Python Ausdruck angegeben, der auf die Daten Zugriff hat, die im Token gespeichert werden, falls die Kantenbedingung zutrifft und der Token produziert werden kann.

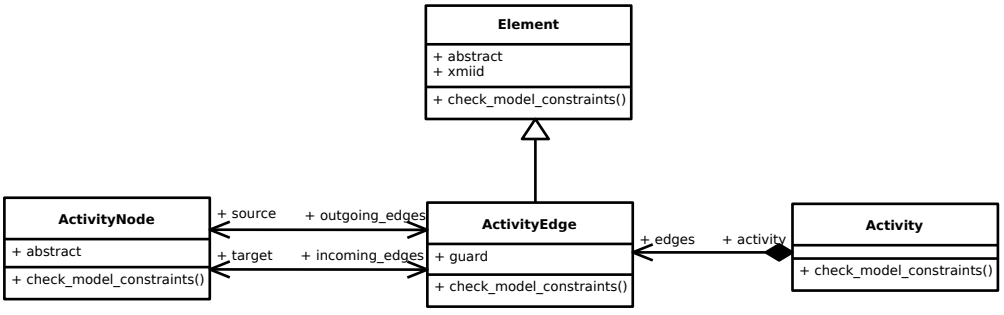

Abbildung 7.3: Kante

7.2.4 Knoten

In Abbildung 7.4 wird ein Überblick über die verschiedenen Knotentypen in AMREP geboten. Abbildung 7.5 zeigt die unterschiedlichen Kontrollknoten.

Der größte Unterschied der UML Metamodellimplementierung in AMREP zum Standard ist das Fehlen von Objektknoten und damit einhergehend die fehlende Unterscheidung zwischen Objekt- und Kontrollkanten. Diese Entscheidung ermöglichte wesentliche Vereinfachungen in der Implementierung der Runtime Engine rief aber auch eine Inkompatibilität mit dem Standard hervor. Dies ist in 7.4, S.67 näher erläutert.

Über die Kompositionsbeziehung *nodes* von Activity können alle Knoten beziehungsweise *ActivityNodes* einer Aktivität abgefragt werden.

Für die Subklasse *Action* können Vor- und Nachbedingungen über die Kompositionsbeziehungen *postconditions* und *preconditions* definiert werden. Diese Klasse ist in der Subklasse *OpaqueAction* spezialisiert, die zur Modellierung von Aktionen in AMREP verwendet werden muss. OpaqueAction besitzt anders als im UML Standard die Attribute *body* und *language* nicht, da das Verhalten nicht direkt in der Aktion, sondern in Executions definiert ist.

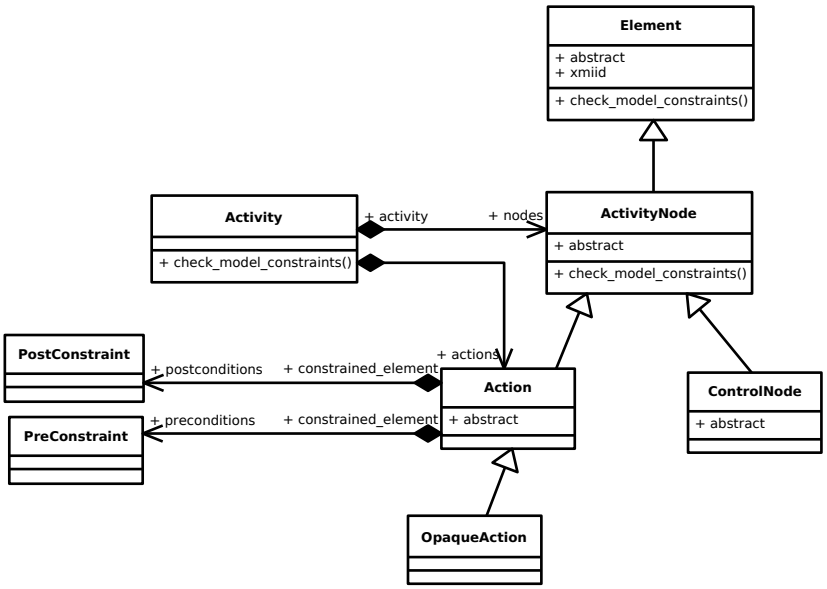

Abbildung 7.4: Knoten

Kontrollknoten steuern den Prozessablauf und sind in Abbildung 7.5 dargestellt.

InitialNodes sind die Startpunkte von Aktivitäten. Es können mehrere InitialNode Elemente definiert werden, die jeweils einen Token für jede ausgehende Kante bei Start der Aktivität erzeugen. Diese Knoten können keine eingehenden Kanten besitzen.

FinalNode ist in ActivityFinalNode und FlowFinalNode spezialisiert. Beide können mehrere eingehende aber keine ausgehende Kanten besitzen. Während ActivityFinalNode die Aktivität als ganzes beendet, konsumiert FlowFinalNode nur den jeweiligen Token und beendet somit den entsprechenden Ablauf, wobei andere Tokens auf dem jeweiligen Zweig nicht gelöscht werden.

DecisionNode ist ein Kontrollknoten, der aufgrund von Kanten-Bedingungen Tokens für eine von mehreren ausgehenden Kanten produziert. Dies entspricht einem Exklusiv-Oder Verhalten.

Elemente vom Typ ForkNode erzeugen Nebenläufigkeit indem sie für jede ausgehende Kante einen Token mit dem selben Daten produzieren.

Ein JoinNode synchronisiert eingehende Kanten, indem ein Token für die ausgehende Kante produziert wird, wenn an allen eingehenden Kanten ein Token konsumiert werden kann. Es wird somit die Anzahl nebenläufiger Tokens reduziert.

MergeNodes bringen mehrere eingehende Flüsse zusammen, synchronisieren aber nicht. Entgegen der UML Spezifikation wird bei einem MergeNode nur ein Token für die ausgehende Kante produziert, auch wenn mehrere Tokens an den eingehenden Kanten konsumiert worden sind. Dieses Verhalten entspricht einer Und Semantik.

7.3 Modellvalidierung

Die Funktion zur Validierung des Modells *validate(node)* ist im selben Modul definiert, in dem die Python Implementierung des Metamodells definiert

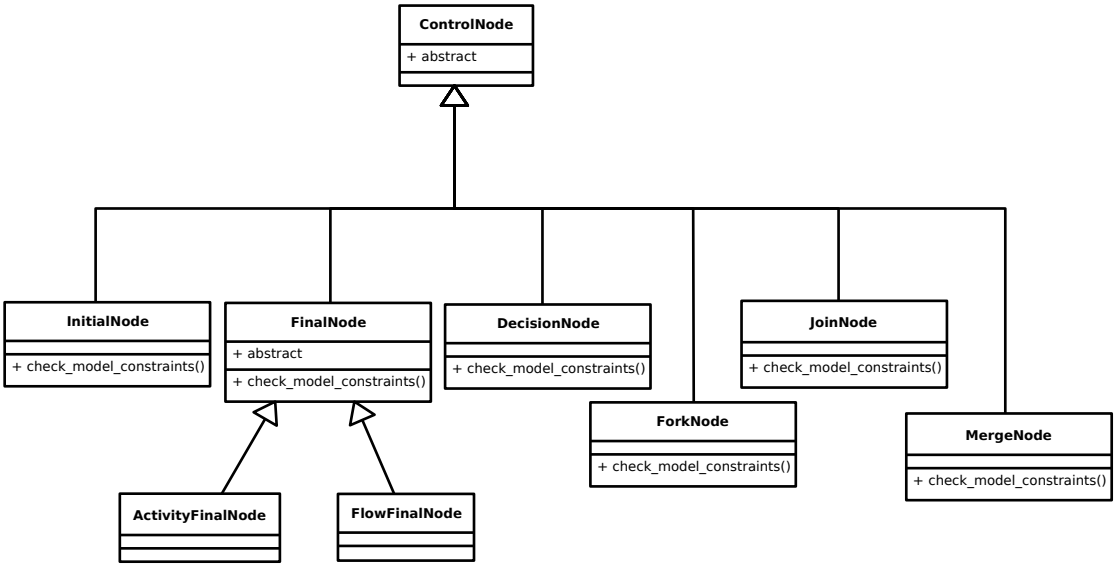

Abbildung 7.5: Kontrollknoten

ist (*activities.metamodel*). Diese Funktion erwartet als Parameter ein Modellelement und ruft dann die Methode *check_model_contraints()* des Modellelements sowie rekursiv jedes darin enthaltene Modellelement auf. Wird der Funktion eine Activity oder ein Package übergeben, wird somit das gesamte Modell validiert. Ist das Modell nicht valide, beispielsweise da eingehende Kanten in einer InitialNode modelliert worden sind, wird ein Fehler erzeugt.

7.4 Unterschiede zum UML2 Metamodell für Aktivitäten

Aufgrund der Komplexität der UML Spezifikation konnte im Rahmen dieser Diplomarbeit nur ein Subset der Metamodellelemente von Aktivitätsdiagrammen umgesetzt werden. Dieses Subset und weitere Vereinfachungen orientieren sich an den Anforderungen, die sich aus dem im Kapitel 1.4, S.6 skizzierten Usecase und den Zielen und Abgrenzungen aus dem Kapitel 4, S.12 ergeben.

Als Aktion steht nur OpaqueAction zur Verfügung, da die Implementierung von

Aktionen zur Gänze durch Executions erfolgt (siehe Kapitel 8.1, S.83). Executions können hierbei kleine Bausteine sein, die elementare Tätigkeiten verrichten oder komplexe Programme mit Userinteraktion und Ausführung weiterer Aktivitätsmodelle.

Executions werden über Profile geladen und über Stereotypen den Aktionen zugewiesen. Die Implementierung von Stereotypen und Profilen ist hierbei nur lose am UML Standard angelehnt, da die Implementierung als Erweiterungsmechanismus von Metamodellen den Rahmen der Diplomarbeit gesprengt hätte, die erforderte Funktionalität aber auch in dieser vereinfachten Form zur Verfügung gestellt werden kann.

Objektknoten wurden nicht implementiert, wodurch die Datensicht auf Modelle nicht modelliert werden kann. Modellrelevante Daten werden über die Startmethode der Runtime Engine in den Prozessablauf eingebracht und von Executions gegebenenfalls manipuliert oder es werden von ihnen neue Daten erzeugt. Das Anbieten aller notwendigen Daten liegt hierbei in der Verantwortung des Prozesses, der die Ausführung des Modells startet. Im Modell kann aber durch Angabe von Vorbedingungen für Aktivitäten das Vorhandensein notwendiger Daten überprüft werden und ein Fehler somit vorzeitig ausgelöst werden, wenn die Bedingungen nicht zutreffen. Die Personen, die Modelle erstellen, müssen somit Kenntnis über von Executions erforderte Daten besitzen. Dies wäre aber auch im Falle einer Umsetzung laut Standard der Fall gewesen, da bestimmte Aktionen bestimmte Daten benötigen und diese durch Objektkanten und Objektknoten modelliert werden müssen. Durch den Verzicht auf Objektknoten konnte das Token Competition Problem, wie im Kapitel 8.3.4, S.79 beschrieben, umgangen werden. Das im selben Kapitel beschriebene Traverse-to-Completion Problem tritt durch diese Entscheidung und eine geänderte Tokenflusssemantik nicht auf. Die Implementierung der Runtime konnte dadurch wesentlich vereinfacht werden, wobei die Eingangs definierten Ziele weiterhin erreicht werden können. Die direkte Kompatibilität mit UML Aktivitätsdiagrammen anderer Werkzeuge wurde somit aber aufgegeben. Modelle für AMREP müssen nach bestimmten Konventionen erstellt werden, die im Kapitel 7.5, S.69 beschrieben werden.

Durch die fehlende Implementierung von Objektknoten gibt es auch keine Un-

terscheidung zwischen Objektkanten und Kontrollkanten. Kanten nehmen beide Funktionen wahr und besitzen einen Daten-Payload, der auch leer sein kann.

Namen von Attributen und Assoziationen wurden teilweise geändert und in den in Python gebräuchlichen Standards notiert (vgl. [van Rossum und Warsaw 2001]).

Erweiterte Konzepte von UML Aktivitätsdiagrammen wie Partitionen, Gruppen, Ausnahmebehandlung, Datenspeicher, Erweiterungsknoten und Erweiterungsregionen, Unterbrechungsregionen, Schleifenknoten, Signale und andere wurden nicht umgesetzt, da sie nicht Teil der Anforderungen waren.

Die Python Implementierung des Metamodells für AMREP ist eine prototypische Implementierung, die in verschiedenen Einsatzszenarien erprobt werden und gegebenenfalls erweitert werden kann. Eine weitere Angleichung an den UML Standard wäre von Vorteil, da eine bessere Interoperabilität mit anderen UML Werkzeugen und Frameworks hergestellt werden könnte.

7.5 Konventionen für die Erstellung von Metamodellen für AMREP

Bei der Modellerstellung für AMREP müssen folgende Konventionen eingehalten werden:

- Es können nur die Modellelemente Profile, Stereotype, TaggedValue, PreConstraint, PostConstraint, Package, Activity, ActivityEdge, OpaqueAction, InitialNode, ActivityFinalNode, FlowFinalNode, DecisionNode, ForkNode, JoinNode und MergeNode verwendet werden.
- Aktionen müssen mit dem Modellelement OpaqueAction modelliert werden.
- Damit Aktionen ausgeführt werden können, müssen sie mit einem Stereotyp versehen werden, der genau dem Namen einer Execution entspricht.

- Die benötigten Executions müssen durch die Angabe von Profilen geladen werden. Profile müssen in Packages definiert werden. Der Name eines Profils muss dem Namen eines Moduls entsprechen, das durch den Python Interpreter importiert werden kann. Weitere Informationen über die Erstellung von Executions sind im Kapitel 8.1 angegeben.

- Es können keine Objektknoten verwendet werden und keine Pins für Aktionen definiert werden.

- Zur Modellierung von Kanten muss das Modellelement ActivityEdge verwendet werden. In UML Editoren müssen Kontrollkanten verwendet werden, die durch den XMI Importer in ActivityEdge Elemente übersetzt werden.

- Guard Bedingungen sowie Pre- und Postconstraints müssen als Python Ausdrücke notiert werden. Der Zugriff auf Variablen aus dem Daten-Payload der Tokens erfolgt durch direkte Angabe des Variablennamens.

- Wird ein durch den Daten-Payload eines Tokens referenziertes Objekt durch eine Execution geändert, ist dieses Objekt auch für andere Tokens geändert deren Daten-Payload das selbe Objekt referenzieren (siehe Kapitel 8.3.2, S.78).

- Die Prinzipien ”Token Competition” und ”Traverse to Completion” sind nicht umgesetzt. Es wird für jede ausgehende Kante, deren Kantenbedingungen zutreffen, ein Token produziert, unabhängig davon, ob der Zielknoten den Token konsumieren kann oder nicht (siehe Kapitel 8.3.4, S.79).

- Wenn Tokens von einer Kante konsumiert werden, wird deren Daten-Payload zusammengeführt. Sind hierbei zwei gleiche Schlüssel definiert, die unterschiedliche Objekte referenzieren, wird ein Fehler ausgelöst und die Aktivität abgebrochen.

- Für FinalNodes gilt die Und-Semantik. Es müssen daher an allen eingehenden Kanten Token zur Verfügung stehen, damit der Knoten aktiv wird und die Aktivität beziehungsweise den Tokenfluss abbricht. Dieses Verhalten ist in der UML Spezifikation nicht explizit definiert. Das Problem kann umgangen werden, indem Für FinalNodes nur eine eingehende Kante modelliert wird.

Kapitel 8

Interpreter der Activity Model Runtime Engine für Python

8.1 Einführung

Der Modellinterpreter[1] beziehungsweise die Runtime Engine, wie diese in AMREP genannt wird, ist für die Ausführung von Aktivitäten zuständig. Aktivitäten werden als Python Modelle geladen und Schritt für Schritt abgearbeitet. Der Interpreter erlaubt die Unterbrechung der Aktivität sowie die Änderung der Python Modelle zur Laufzeit.

Dieses Kapitel erläutert die Funktionsweise und Implementierung der Runtime Engine.

8.2 Implementierung der Runtime Engine

Die Runtime Engine ist im Packet *activities.runtime* implementiert. Die involvierten Klassen werden in der Abbildung 8.1 dargestellt.

Im Klassendiagramm ist ersichtlich, dass das Adapter Pattern in drei Fällen

[1] Zur Definition eines Interpreters siehe Kapitel 5.1, S.16 .

zum Einsatz kommt. Der für das Interface IActionInfo registrierte Adapter adaptiert Knoten vom Typ Action und liefert ein ActionInfo Objekt mit Informationen aus der adaptierten Aktion zurück. Der Adapter ITaggedValueDict zur Adaption von Stereotypen wird verwendet, um ein Dictionary mit TaggedValues des adaptierten Stereotyps zu erzeugen. ITokenFilter ist ein sogenannter Multiadapter[2], der zwei Objekte vom Typ TokenPool und ActivityEdge adaptiert und wird verwendet, um alle Tokens zu einer bestimmten Kante auszulesen[3].

Der Kern der Runtime Engine ist die Klasse *ActivityRuntime* . Bei der Instantiierung wird der *__init__* Methode die auszuführende Aktivität als Parameter übergeben und die Klasse *TokenPool* instantiiert, die alle aktiven Tokens während der Laufzeit verwaltet.

Die Runtime Engine wird mit der Methode *start* gestartet. Als optionaler Parameter kann ein Daten Dictionary[4] übergeben werden, dessen Schlüssel (engl. *Keys*) den im Modell verfügbaren Variablennamen entsprechen sollen und dessen Werte beliebige Python Objekte[5] sein können.

Bei Aufruf der Start-Methode werden folgende Schritte ausgeführt (siehe Abbildung 8.2 als Darstellung als Aktivitätsmodell):

1. Es wird überprüft, ob *TokenPool* aktive Tokens enthält und ein Fehler ausgelöst und der Vorgang abgebrochen falls dies der Fall ist, da die Aktivität bereits ausgeführt wird.

2. Es werden die Vorbedingungen der Aktivität überprüft und ein Fehler ausgelöst, wenn diese nicht erfüllt werden können.

3. Es wird über alle Profile des Package iteriert, in der die Aktivität definiert

[2] Multiadapter werden in der Zope Component Architecture verwendet, um zwei Objekte zu adaptieren und unterschiedliches Verhalten, abhängig von beiden Objekten, zur Verfügung zu stellen. Multiadapter werden in ZCA basierten Web Frameworks häufig verwendet, um unterschiedliche Ansichten (engl. *Views*) für ein Objekt, abhängig vom *Request* bereit zu stellen.

[3] Dieser Adapter trägt den Namen *tokenfilter* und ist deshalb in Kleinbuchstaben notiert, da er als Funktion und nicht als Klasse implementiert ist.

[4] In Python ist ein *dictionary* ein Hashtable, dessen Elemente Objektreferenzen und in Konsequenz beliebige Datentypen beinhalten können (vgl. [Martelli 2006], S.44). Ein Dictionary ist einem Array anderer Programmiersprachen sehr ähnlich.

[5] In Python wird jeder Wert inklusive Zahlen, Zeichen und Funktionen als Objekt repräsentiert.

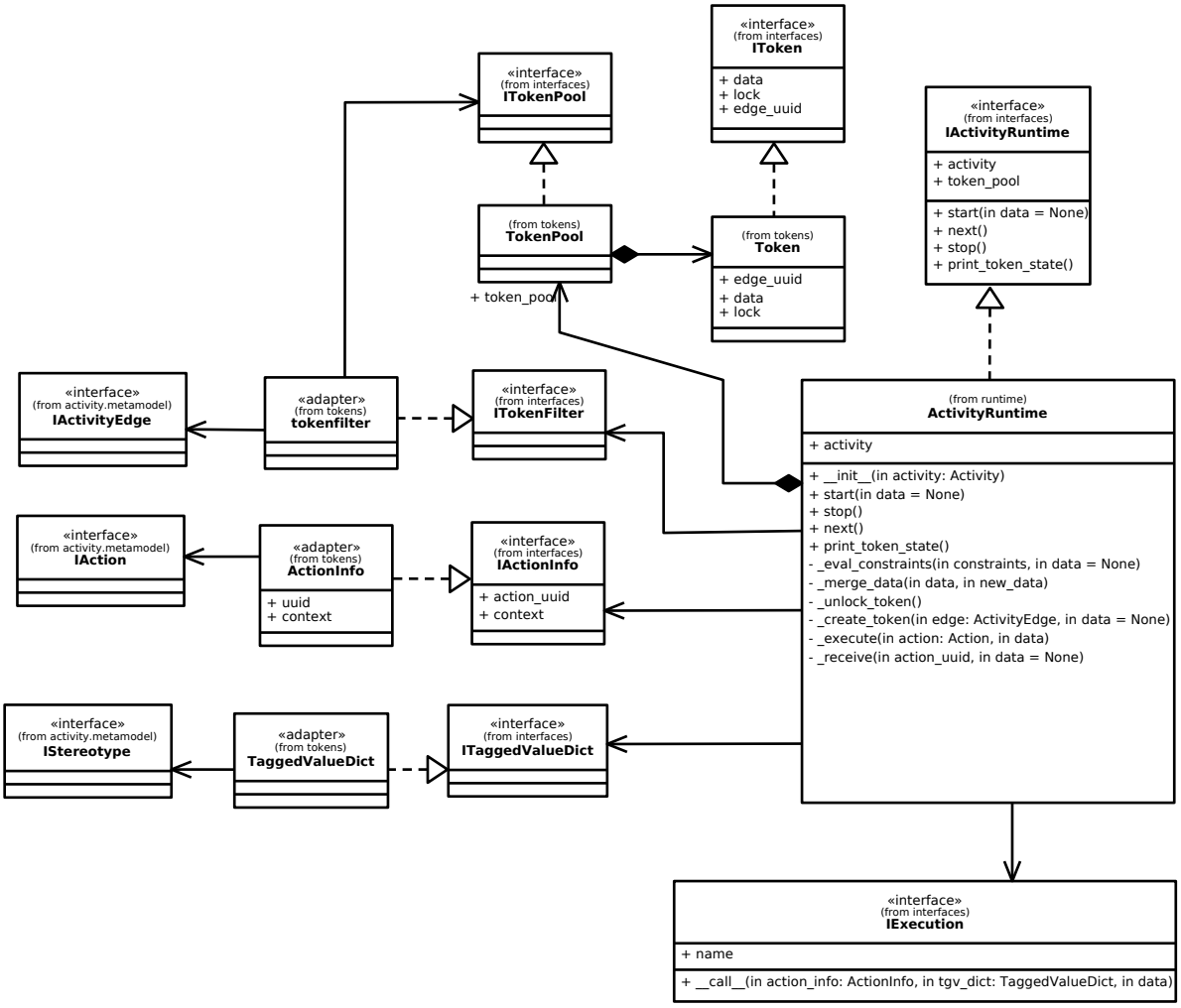

Abbildung 8.1: Klassen der Runtime Engine

ist. Der Name jedes Profils wird verwendet, um ein gleichnamiges Python Modul zu importieren.

4. Für jede ausgehende Kante jeder InitialNode wird ein Token mit den übergebenen Daten produziert.

5. Es werden alle Tokens entsperrt, damit diese im nächsten Schritt von Knoten konsumiert werden können

Neu erzeugte Tokens werden deshalb gesperrt, damit diese im aktuellen Schritt keine Kanten traversieren können. Andernfalls wäre es möglich, dass neu pro-

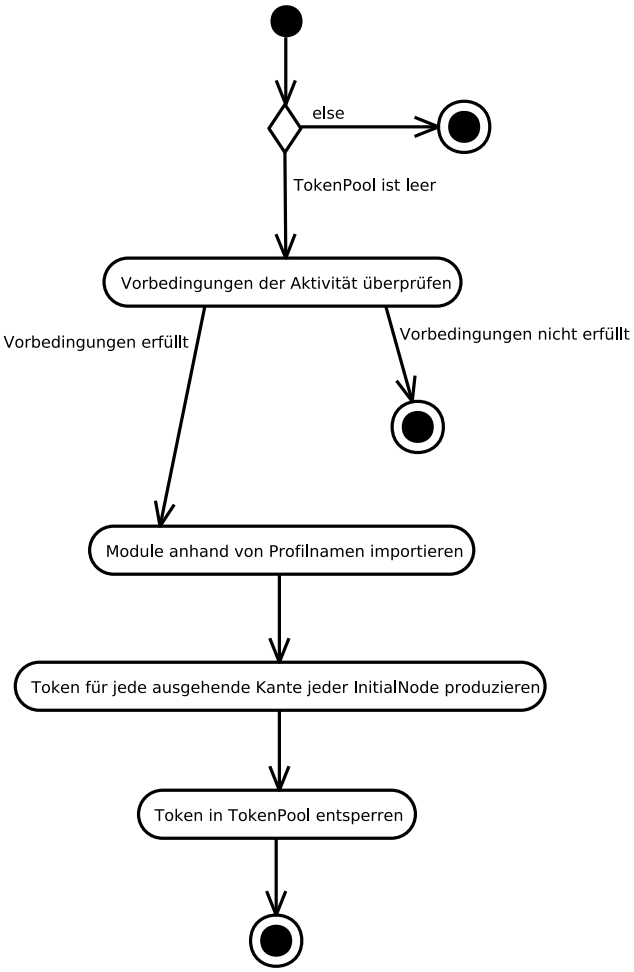

Abbildung 8.2: Aktivitätsmodell der Start-Methode

duzierte Tokens die gesamte Aktivität in einem Iterationsschritt durchlaufen können. Dieses Verhalten ist auch für nebenläufige Aktivitäten von Bedeutung, da andernfalls ein Zweig bis zum Ende durchlaufen werden könnte, ohne dass sich Tokens in einem anderen Zweig aufgrund von Knotenbedingungen bewegen würden.

Nachdem die Aktivität gestartet wurde kann mit der *next* Methode die Aktivität weiter abgearbeitet werden. Es werden die Schritte wie in Abbildung 8.3 dargestellt ausgeführt.

Die Next-Methode iteriert über alle in der Aktivität definierten Knoten. Falls der Knoten der aktuellen Iteration ein InitialNode ist, oder dieser aufgrund feh-

lender Tokens nicht ausgeführt werden kann, wird zur nächsten Iteration gesprungen. Für jeden Knoten werden alle Tokens aus den eingehenden Kanten gelöscht und der Daten-Payload zusammengeführt. Wenn der Knoten eine Aktion ist, wird diese ausgeführt (siehe auch 8.1, S.83). Anschließend werden für alle ausgehenden Kanten Tokens produziert, wenn die Kantenbedingungen dies zulassen. Konnten keine Tokens produziert werden und ist eine Kante mit der speziellen Kantenbedingung ”else” vorhanden, wird für diese Kante ein Token produziert. Wenn der aktuelle Knoten vom Typ ”FinalNode” ist, werden Rückgabedaten aus dem Daten-Payload erzeugt. Nachdem über alle Knoten iteriert worden ist, werden die Tokens im TokenPool entsperrt. Falls ein Token zuvor einen Knoten vom Typ ActivityFinalNode erreicht hat, werden alle Tokens aus dem TokenPool gelöscht. Ist der TokenPool leer, so ist die Aktivität beendet und die Nachbedingungen können überprüft werden. Ist ein Rückgabewert vorhanden, wird dieser nun zurück gegeben.

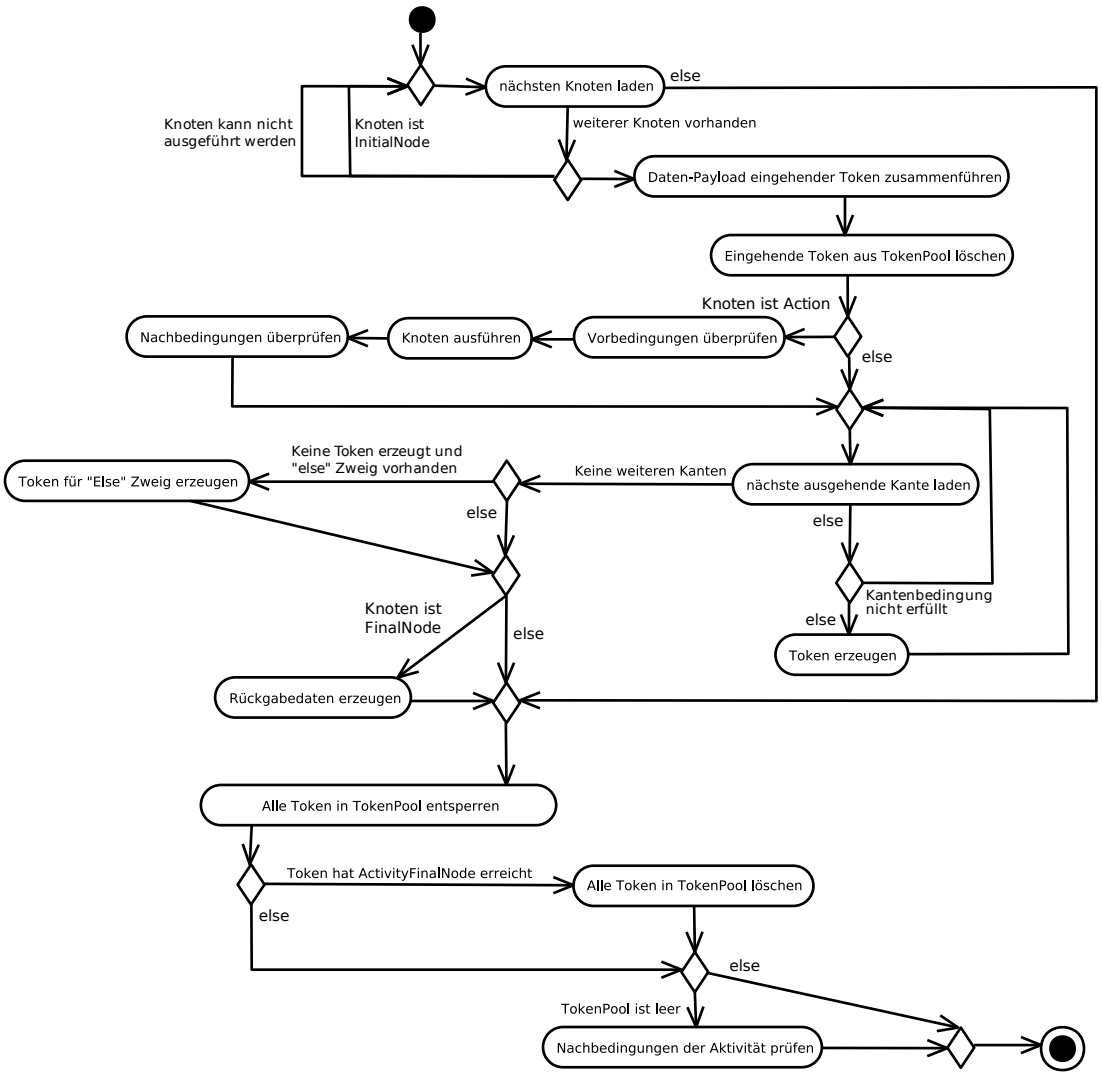

Abbildung 8.3: Aktivitätsmodell der Next-Methode

8.3 Token-Fluss in der AMREP

8.3.1 Allgemeines

Das Konzept der Tokens basiert auf dem Konzept der *Marken* aus den Petri-Netzen (vgl. [Rumbaugh u. a. 2005], S.654). Tokens sind Runtime-Markierungen, die den Zustand der Aktivität beschreiben.

Knoten werden ausgeführt wenn genügend Tokens an den eingehenden Kanten zur Verfügung stehen. Bei allen Knoten außer *MergeNode* müssen an allen eingehenden Kanten Tokens vorhanden sein damit diese konsumiert werden können und der Knoten ausgeführt werden kann. Dies entspricht einer impliziten Und-Semantik. Im Falle von Knoten des Typs *MergeNode* muss nur ein Token an einer der eingehenden Kanten zur Verfügung stehen. Dies entspricht einer impliziten Oder-Semantik. Nach der Ausführung des Knotens werden Tokens für alle ausgehenden Kanten erzeugt (implizite Und-Semantik). Hiervon ausgenommen sind Knoten vom Typ *DecisionNode*, die nur für die erste mögliche ausgehende Kante ein Token produzieren, wo die Kantenbedingung es zulässt. Welche Kante dies ist, kann nicht vorhergesagt werden. Das Verhalten ist in diesem Fall nicht deterministisch. Da die Implementierung auf einem *ordered dictionary* basiert, ist die gewählte Kante von der Reihenfolge der Definitionen im Modell abhängig. Das Verhalten von Knoten vom Typ *DecisionNode* entspricht einer impliziten Exklusiv-Oder-Semantik (XOR).

Tokens aus eingehenden Kanten werden von der Runtime Engine für Knoten, die ausgeführt werden, gelöscht. Zuvor wird der Daten-Payload vom Token kopiert und mit dem Daten-Payload anderer Tokens vermengt (*merge*). Nach der Ausführung des Knotens werden Tokens mit dem Daten-Payload erneut produziert. Tokens existieren also immer nur solange, bis ein nächster Knoten ausgeführt wird.

Es ist hierbei erwähnenswert, dass Tokens nicht Teil des Metamodells sind. Weiters referenzieren nicht direkt auf Modellelemente vom Typ *ActivityEdge* , sondern speichern die *uuid* der Kante. Somit können Modell und TokenPool getrennt voneinander gespeichert werden.

8.3.2 Daten-Payload

In der AMREP-Implementierung gibt es keine Unterscheidung zwischen Objekt- und Kontrolltoken. Tokens besitzen immer ein Attribut (*data*), welches bei Initialisierung auf ein leeres Python-Dictionary gesetzt wird. Demzufolge besitzt ein Token in der AMREP-Implementierung immer einen Daten-Payload, der entweder leer ist oder mit Schlüsselwort-/Elementpaaren befüllt. Das Schlüsselwort entspricht dem Namen der Variablen, die bei der Auswertung von Kantenbedingungen und Vor- und Nachbedingungen für Aktionen und Aktivitäten zur Verfügung stehen. Als Elemente können beliebige Python Objekte verwendet werden. Dieser Daten-Payload entspricht den Werten bzw. Objekten, die mit Objekttokens über Objektkanten fließen.

Im Falle von Kontrollknoten werden die eventuell zusammengeführten Daten weitergereicht, also nicht verändert in den Tokens der ausgehenden Kanten gespeichert. Im Falle von Aktionen wird der Daten-Payload den *Executions* übergeben, die diesen gegebenenfalls verändern und zurückgeben. Dieser Rückgabewert wird als neuer Daten-Payload für Tokens auf den ausgehenden Kanten verwendet. Es hängt also von der Implementierung der *Executions*) ab, ob Werte im Daten-Payload aus vorhergehenden Aktionen in nachfolgenden Aktionen noch zur Verfügung stehen (siehe Kapitel 8.1, S.83).

Bei Variablenzuweisungen werden in Python Referenzen auf Objekte der Variable zugewiesen. Python unterscheidet zwischen veränderbaren (engl. *mutable*) und unveränderbaren (engl. *immutable*) Objekten. Zahlen, Zeichen und Strings sind Beispiele für unveränderbare Objekte. Operationen auf diesen Objekten erzeugen neue Objekte. Veränderbare Objekte sind zum Beispiel Instanzen selbst definierter Klassen. Wird ein veränderbares Objekt eines Daten-Payloads manipuliert, so ist dieses Objekt für jeden Daten-Payload der Tokens, die dieses Objekt referenzieren, geändert. Wird einem Schlüssel aber ein neues Objekt zugewiesen, wird nur die Referenz des auf dieses Objekt für den Daten-Payload des entsprechenden Tokens geändert. Die Daten-Payloads anderer Tokens mit dem selben Schlüssel-Bezeichner werden dadurch nicht geändert.

Wenn in einem Knoten, der ausgeführt wird, die Token konsumiert werden, wird versucht, die Daten der Tokens zusammenzuführen. Es wird hierfür ein

neues Dictionary erzeugt, in das alle Schlüssel-/Elementpaare kopiert werden. Haben zwei Schlüssel den selben Bezeichner, wird geprüft, ob die referenzierten Objekte die selbe Identität haben. Ist das der Fall wird ein entsprechender Schlüssel im Dictionary angelegt. Wenn die referenzierten Objekte unterschiedlich sind, wird ein Fehler erzeugt und die Ausführung der Aktivität abgebrochen. Dieser Umstand muss bei der Modellierung und Implementierung von *Executions* berücksichtigt werden. Es wird in vielen Einsatzszenarien nicht wünschenswert sein, dass die Aktivität in einem solchen Fall abgebrochen wird. Ein solcher Fehler kann abgefangen werden und zum Beispiel durch Benutzerinteraktion gelöst werden.

8.3.3 Tokens referenzieren Kanten

In der AMREP Implementierung werden Token auf Kanten referenziert. Ist ein Token bekannt, kann somit die zugehörige Kante abgefragt werden und über die Kante der Quell- und Zielknoten ermittelt werden. Die *uuid* der Kante ist ein obligatorischer Parameter beim Instantiieren des Tokens. Tokens werden nach Ausführung eines Knoten von der Runtime Engine für jede ausgehende Kante produziert, bei der die Kantenbedingungen zutreffen.

Tokens werden in einer Liste gespeichert[6] wodurch das First-In First-Out (FIFO) Prinzip umgesetzt ist. Werden die Tokens einer Kante abgefragt, werden die ältesten für diese Kante produzierten Tokens zuerst zurückgeliefert.

8.3.4 Traverse-To-Completion Problem

Nach der UML2-Spezifikation kann ein Token erst dann eine Kante traversieren, wenn eine nachfolgende Aktion den Token aufnehmen kann und die Kantenbedingungen, Ziel- und Quellknotenbedingungen zutreffen. Dabei muss der Zielknoten eine Aktion oder ein Objektknoten sein, da Kontrollknoten keine Tokens halten können (vgl. [Object Management Group 2009c], S.319). Dieses

[6] Eine Instanz der Klasse *TokenPool*, die vom Python-Builtin *list* erbt.

Token-Fluss Prinzip wird nach Conrad Bock *Traverse-To-Completion* genannt (vgl. [Bock 2004], S.35).

Eine weitere Regel des Tokenfluss nach der UML2-Spezifikation ist, dass Tokens nur über eine Kante traversieren können, auch wenn ein Objektknoten mehrere ausgehende Kanten besitzt. Dieses Prinzip wird nach Conrad Bock *Token Competition* genannt (vgl. [Bock 2004], S.35 und [Object Management Group 2009c], S.312). Dies entspricht einer *XOR* Semantik. Die Entscheidung, über welche der ausgehenden Objektkanten der Token traversiert kann nicht vorausgesagt werden (vgl. [Crane und Dingel 2008]).

Die Prinzipien *Traverse-To-Completion* und *Token Competition* sollen sicherstellen dass keine Deadlock-Situation bzw. Verklemmung entsteht. Sie verhindern, dass Knoten angebotene Tokens akzeptieren, obwohl sie nicht ausgeführt werden können, da die notwendigen Bedingungen nicht zutreffen und somit anderen Knoten den Konsum des Tokens entziehen (vgl. [Object Management Group 2009c], S.312).

Das Prinzip *Traverse-To-Completion* erschwert die Implementierung einer Runtime, da bei jedem Durchlauf getestet werden muss, ob ein traversieren des Tokens möglich ist bevor das Token weitergereicht wird. Dabei müssen alle anderen Abhängigkeiten wie nebenläufige Token-Flüsse mit berücksichtigt werden. Mit zunehmender Komplexität müssten immer mehr Testfälle durchgespielt werden, was zu einer negativen Beeinflussung der Performance des Systems führen würde.

In der AMREP Implementierung der Runtime Engine werden diese Prinzipien nicht berücksichtigt. Es wird für jede ausgehende Kante Tokens produziert, unabhängig davon, ob ein nachfolgender Knoten diesen Token unmittelbar konsumieren kann oder nicht. Dieses Verhalten ist beabsichtigt, da in der AMREP Implementierung davon ausgegangen wird, dass jeder Zielknoten einer Kante, deren Kantenbedingungen zutreffen, auch ausgeführt werden soll. Dieses geänderte Verhalten ist bei der Modellierung ebenfalls zu berücksichtigen.

8.4 Executions als Implementierung von Aktionen

8.4.1 Einführung

Im AMREP Metamodell ist nur die Aktion OpaqueAction als konkrete, instantiierbare Aktion definiert. Die Granularität der in der UML Spezifikation definierten Aktionen sind im AMREP Metamodell nicht umgesetzt. Stattdessen wird die Ausführungslogik getrennt in Python-Callables[7] implementiert, die hier *Executions* genannt werden.

Executions können granulare Aufgaben erledigen sowie umfassende Tätigkeiten ausführen, die Benutzerinteraktionen und den Aufruf von weiteren Aktivitäten beinhalten können.

Executions sind dabei keine Artefakte, die von einer Person erstellt werden, die Aktivitäten modelliert und nicht mit der Programmierung vertraut ist. Executions und Modellprofile, mithilfe denen Executions geladen werden, werden vielmehr von einer Entwicklungsabteilung bereitgestellt. Dabei kann ein Pool an Executions für verschiedene Einsatzzwecke, Problemstellungen und Domänen, angelegt werden. Dieser Pool an Executions kann durch Modellprofile organisiert beziehungsweise kategorisiert werden. Personen, die Aktivitäten modellieren, können sich dann aus bereitgestellten Profilen bedienen, diese an *Packages* , in denen *Activities* definiert sind, anwenden und Modellelemente mit *Stereotypen* annotieren, um so die Tätigkeiten, die Aktionen ausführen sollen, zu definieren.

In der Tabelle 8.1 wird der Zusammenhang zwischen Aktivitätsmodellelementen und Executions dargestellt. In den Zeilen sind die Elemente einer Execution angegeben und in den Spalten die Modellelemente. Das Modellelement Profil definiert das Python Paket beziehungsweise Modul, in dem Executions definiert sind. Stereotypen geben den Namen der für eine Aktion zu ladenden Execution an. Mit Tagged Values können schließlich Übergabeparameter modelliert werden, die gemeinsam mit dem Daten-Payload von Tokens und dem

[7] In Python sind *callable types* Instanzen, die Funktionsaufrufoperationen in der Form `function-object(arguments)` unterstützen. Dies können normale Funktionen, Typen, Objekte und Methoden von Objekten sein (vgl. [Martelli 2006], S.45 und S.73).

Kontext der Aktivität an die Execution übergeben werden.

	Profil	Stereotyp	Tagged Value	Daten-Payload	Aktivitätskontext
Python Paket	X				
Execution		X			
Parameter			X	X	X

Tabelle 8.1: Zusammenhang zwischen Executions und Modellelementen

8.4.2 Trennung zwischen Modell und Implementierung

Executions sind getrennt vom Modell implementiert. Mithilfe von Stereotypen, die für Knoten des Typs *OpaqueAction* definiert sind, werden die Executions geladen und von der Runtime gestartet. Executions implementieren das Interface *IExecutions* und werden in der Zope Component Registry als *Named-Utility* registriert. Der Name der Named-Utility entspricht dabei genau dem Namen des Stereotyps, der für die *OpaqueAction* angewandt wird. Wird nun eine *OpaqueAction* ausgeführt und ist ein Stereotyp für diese vorhanden, wird der Name des Stereotyps für das Laden der Execution aus der Registry verwendet.

Die Runtime übernimmt das Registrieren der Executions beim Aufruf der *start*-Methode. Hierbei wird über alle Profile, die für das Package der Aktivität definiert sind, iteriert. Der Name jedes Profils wird der Python-Funktion *__import__* übergeben, die ein Modul importiert, das genau dem Namen des Profils entspricht. Dieses Modul muss dann die *Executions* als Named Utilities mit der Methode *registerUtility* des *GlobalSiteManager* aus dem Modul *zope.component* registrieren.

Wird eine Execution aufgerufen, wird ihr ein *ActionInfo* Objekt mit Informationen über die ausgeführte Aktion, ein *TaggedValueDict* Objekt, das alle Tagged Values der Aktion beinhaltet und die zusammengeführten Daten der Tokens der Aktion übergeben.

Die beschriebene Trennung zwischen Modell und der Aktionsimplementierung wird im wesentlichen mit einer Variante des *Command Pattern* ermöglicht. In

der Abbildung 8.4 werden die involvierten Akteure dargestellt.

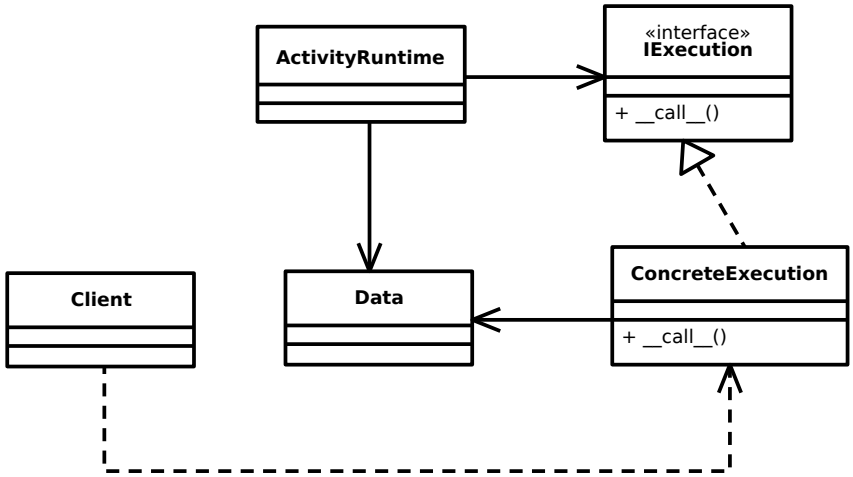

Abbildung 8.4: Anwendung des Command Pattern für Executions

Der *Client* im Sinne des Command Patterns, der für die Instantiierung der konkreten Implementierung der Executions (nach dem Command Pattern: *ConcreteCommand*) zuständig ist, ist das Python Modul, in dem die Executions als Named-Utilities registriert werden. Im Unterschied zum Command Pattern besitzt der Client aber keine Referenz auf die zu manipulierenden Daten, die nach der Nomenklatur des Command Patterns dem Reciever entsprechen. Diese Referenz besitzt die ActivityRuntime, die dem *Invoker* des Command Patterns entspricht. Für ActivityRuntime ist die Implementierung der Execution unbekannt. Sie lädt mithilfe des Stereotyps der Aktion die konkrete Implementierung des Execution aus der Zope Component Registry und ruft diese direkt auf.

8.4.3 Beispiel

Im folgenden Listing 8.1 wird beispielhaft gezeigt, wie AMREP-Executions definiert werden können.

```
1 from activities.runtime.interfaces import IExecution
2 from zope.interface import implements
3 from zope.component import getGlobalSiteManager
4
```

```
5  class Execution1(object):
6      implements(IExecution)
7      name = "execution1"
8
9      def __call__(self, action_info, stereotype_info, data):
10         # value of key test is changed
11         # rest of data is passed through
12         data['test'] = not data['test']
13         print "data['test'] = " + str(data['test'])
14         return data
15
16 class Execution2(object):
17     implements(IExecution)
18     name = "execution2"
19
20     def __call__(self, action_info, stereotype_info, data):
21         # data is replaced
22         return {'info': "execution2 has completed"}
23
24 # registration
25 gsm = getGlobalSiteManager()
26 gsm.registerUtility(component=Execution1(), name=Execution1.name)
27 gsm.registerUtility(component=Execution2(), name=Execution2.name)
```

Listing 8.1: Definition von Executions in Python

5-14 Definition einer Execution mit dem Namen ”execution1”. Der Name der Klasse ist nicht von Bedeutung. Die Klasse implementiert das IExecution-Interface und stellt deswegen das Klassenattribut `name` und die `__call__`-Methode mit der im Interface definierten Signatur bereit[8]. In diesem Beispiel verändert die __call__-Methode den Wert des Dictionary-Schlüssels ”test” der Variable `data`, aber gibt `data` ansonsten der aufrufenden *Action* unverändert zurück. Alle anderen Schlüssel des Dictionaries bleiben hier unverändert.

16-22 Definition einer weiteren Execution mit dem Namen ”execution2”. Hier wird in der `__call__`-Methode der aufrufenden Aktion ein neues *Dictionary* zurückgegeben, was zur Folge hat, dass die von der Aktion produzierten Tokens einen neuen Daten-Payload zugewiesen bekommen.

25-27 Registrierung der beiden Executions. Die Angabe des Interfaces ist dabei nicht notwendig, da die Executions jeweils nur ein Interface imple-

[8]Es sei hierbei angemerkt, dass der Python Sprachkern keine Interfaces definiert. Sie sind eine Erweiterung des Zope Component Frameworks. Es wird auch nicht überprüft, ob die Implementierung der Interfaces mit ihrer Deklaration übereinstimmen. Interfaces werden hauptsächlich dazu verwendet, um eine lose Kopplung der Komponenten zu erzielen und registrierte Komponenten anhand ihrer Interfaces anzusprechen.

mentieren und das implementierte Interface hier eindeutig ermittelt werden kann. Die Executions können danach anhand ihres Namens und des Interfaces von der Registry geholt werden.

Kapitel 9

Verwendung der Activity Model Runtime Engine für Python

9.1 Modellerstellung mit Python

Im folgenden wird der im Kapitel 1.4, S.6 skizzierte Usecase hier als Python Modell nachgebildet. Es folgen Erklärungen der wichtigsten Programmteile.

```
1  import activities.metamodel as mm
2
3  class Patient(object):
4      def __init__(self, name, health):
5          self.name = name
6          self.health = health # int btw. 0 (dead) and 100 (healthy
               )
7
8  profile = mm.Profile('activities.test.hospital')
9
10 model = mm.Package('hospital-admission')
11 model[profile.__name__] = profile
12 model['main'] = mm.Activity()
13 act = model['main']
14
15 act['pc1'] = mm.PreConstraint(specification='patient is not None'
       )
16 act['pc2'] = mm.PreConstraint(specification='patient.health > 0')
17 act['po1'] = mm.PostConstraint(specification='patient.health >=
       90')
18
19 act['start'] = mm.InitialNode()
20 act['d1'] = mm.DecisionNode()
```

```
act['m1'] = mm.MergeNode()
act['f1'] = mm.ForkNode()
act['j1'] = mm.JoinNode()
act['end'] = mm.ActivityFinalNode()

act['first diagnosis'] = mm.OpaqueAction()
act['first diagnosis']['diagnosis'] = mm.Stereotype(profile=
    profile)

act['acute therapy'] = mm.OpaqueAction()
act['acute therapy']['therapy'] =\
    mm.Stereotype(profile=profile)
act['acute therapy']['therapy']['variation'] =\
    mm.TaggedValue(value="acute")

act['data acquisition'] = mm.OpaqueAction()
act['data acquisition']['data-acquisition'] =\
    mm.Stereotype(profile=profile)

act['data verification'] = mm.OpaqueAction()
act['data verification']['data-verification'] =\
    mm.Stereotype(profile=profile)

act['diagnosis'] = mm.OpaqueAction()
act['diagnosis']['diagnosis'] =\
    mm.Stereotype(profile=profile)

act['normal therapy'] = mm.OpaqueAction()
act['normal therapy']['therapy'] =\
    mm.Stereotype(profile=profile)
act['normal therapy']['therapy']['variation'] =\
    mm.TaggedValue(value="normal")


act['1'] = mm.ActivityEdge(\
    source=act['start'],\
    target=act['first diagnosis'])
act['2'] = mm.ActivityEdge(\
    source=act['first diagnosis'],\
    target=act['d1'])
act['3'] = mm.ActivityEdge(\
    source=act['d1'],\
    target=act['acute therapy'],
    guard="diagnosis == 'acute'")
act['4'] = mm.ActivityEdge(\
    source=act['d1'],\
    target=act['m1'],
    guard="diagnosis == 'normal'")
act['5'] = mm.ActivityEdge(\
    source=act['acute therapy'],\
    target=act['m1'])
```

```
71 act['6'] = mm.ActivityEdge(\
72     source=act['m1'],\
73     target=act['data acquisition'])
74 act['7'] = mm.ActivityEdge(\
75     source=act['data acquisition'],\
76     target=act['f1'])
77 act['8'] = mm.ActivityEdge(\
78     source=act['f1'],\
79     target=act['data verification'])
80 act['9'] = mm.ActivityEdge(\
81     source=act['f1'],\
82     target=act['diagnosis'])
83 act['10'] = mm.ActivityEdge(\
84     source=act['data verification'],\
85     target=act['j1'])
86 act['11'] = mm.ActivityEdge(\
87     source=act['diagnosis'],\
88     target=act['j1'])
89 act['12'] = mm.ActivityEdge(\
90     source=act['j1'],\
91     target=act['normal therapy'])
92 act['13'] = mm.ActivityEdge(\
93     source=act['normal therapy'],\
94     target=act['end'])
95
96 mm.validate(model)
```

Listing 9.1: Usecase *Patientenversorgung in einem Krankenhaus* als Python-Modell

1 Import des Package `activites.metamodel` und Zuweisung des Alias `mm` für einen schnellen Zugriff. Das `__init__.py`-Modul des Packages importiert alle konkreten Metaklassen und alle notwendigen Methoden, weshalb dieser Import den Zugriff darauf erlaubt.

3-6 Definition der Hilfsklasse Patient, die Patientenrelevante Daten beinhaltet.

8 Definition eines Profils mit dem Namen eines Moduls, das vom Python Interpreter importiert werden kann.

10 Instantiierung der Metaklasse *Package*.

11 Dem Modellelement *Package* wird ein Profil zugewiesen. Da alle Metamodellelemente von *Node* erben und die *Dictionary*-API unterstützen, muss dem Modellelement ein Schlüssel zugewiesen werden. In diesem Fall muss dem Konstruktor von *Profil* aber kein Name übergeben werden, da der Schlüssel als Name für das Node-Element verwendet wird. Der

Schlüssel bzw. Name des Profils muss einem Package entsprechen, das geladen werden kann, und das Executions registriert (siehe auch 8.1, S.83).

12 Die Aktivität mit dem Namen ”main” wird dem Package zugewiesen.

13 Es wird eine Variable definiert, die in weiterer Folge einen komfortablen Zugriff auf die Aktivität erlaubt.

15-17 Vor- und Nachbedingungen für die Aktivität werden definiert.

20-51 Definition der Modellelemente, Zuweisung von Stereotypen und Definition von TaggedValues.

54-94 Definition der Kanten und Verbindung mit den Knoten.

96 Validierung des Modells.

9.2 Verwendung der Runtime

Die zuvor modellierte Aktivität wird im folgenden ausgeführt und manipuliert. Der gesamte Testcase inklusive Definition der Executions, ausführlich mit Kommentaren versehen, ist im Paket *activities.test.hospital* zu finden.

```
1  Model- and Runtime Import and Instantiation of the Runtime
2  ==========================================================
3      >>> from activities.test.hospital.model import model, Patient
4      >>> from activities.runtime.runtime import ActivityRuntime
5      >>> ar = ActivityRuntime(model['main'])
6
7  Testing very high disease
8  =========================
9      >>> ar.start({'patient':Patient("Mister Patient", 10)})
10     >>> ar.ts()
11     1: <...Token...>, data: {'patient': <...Patient...>}
12
13     >>> ar.next()
14     'executing: "first diagnosis"'
15     >>> ar.ts()
16     2: ... data: {'patient': <...Patient...>, 'diagnosis': 'acute
           '}
17
18     >>> ar.next()
19     >>> ar.ts()
20     3: ...
```

```
21
22      >>> ar.next()
23      'executing: "acute therapy"'
24      >>> ar.ts()
25      5: ...
26
27      >>> ar.next()
28      >>> ar.ts()
29      6: ...
30
31      >>> ar.next()
32      'executing: "data acquisition"'
33      >>> ar.ts()
34      7: ... data: {'patient': <...Patient...>, 'name': 'Mister
            Patient', 'diagnosis': 'acute'}
35
36      >>> ar.next()
37      >>> ar.ts()
38      8: ...
39      9: ...
40
41      >>> ar.next()
42      'executing: "data verification"'
43      'executing: "diagnosis"'
44      >>> ar.ts()
45      10: ... data: {'patient': <...Patient...>, 'name': 'Mister
            Patient'}
46      11: ... data: {'patient': <...Patient...>, 'name': 'Mister
            Patient', 'diagnosis': 'normal'}
47
48      >>> ar.next()
49      >>> ar.ts()
50      12: ... data: {'patient': <...Patient...>, 'name': 'Mister
            Patient', 'diagnosis': 'normal'}
51
52      >>> ar.next()
53      'executing: "normal therapy"'
54
55  We are short before the end of the graph
56      >>> ar.ts()
57      13: ...
58
59  Let's check our patient's health
60      >>> ar.token_pool[0].data['patient'].health
61      80
62
63
64  Thats way too less! Our patient is not healthy enough
65  but our model is not prepared for this case.
66  We have to change our model at runtime
67  with active tokens in place!
```

```
68
69
70 Changing the model at runtime
71 ==============================
72 We have access to the activity stored in the runtime engine.
73     >>> ar.activity
74     <Activity ...>
75
76 Let's add an DecisionNode to redirect the token to
77 "normal therapy" if health < 90
78     >>> from activities.metamodel.elements import DecisionNode
79     >>> from activities.metamodel.elements import MergeNode
80     >>> from activities.metamodel.elements import ActivityEdge
81     >>> ar.activity['d2'] = DecisionNode()
82     >>> ar.activity['m2'] = MergeNode()
83     >>> ar.activity['12'].target = ar.activity['m2']
84     >>> ar.activity['13'].target = ar.activity['d2']
85     >>> ar.activity['14'] = ActivityEdge(\
86     ...     source=ar.activity['m2'],\
87     ...     target=ar.activity['normal therapy'])
88     >>> ar.activity['15'] = ActivityEdge(\
89     ...     source=ar.activity['d2'],\
90     ...     target=ar.activity['m2'],\
91     ...     guard="patient.health < 90")
92     >>> ar.activity['16'] = ActivityEdge(
93     ...     source=ar.activity['d2'],\
94     ...     target=ar.activity['end'],\
95     ...     guard="patient.health >= 90")
96     >>> from activities.metamodel.elements import validate
97     >>> validate(ar.activity)
98
99
100 Our token state didn't change
101     >>> ar.ts()
102     13: ...
103
104 Going back to MergeNode "m2"
105     >>> ar.next()
106     >>> ar.ts()
107     15: ...
108
109     >>> ar.next()
110     >>> ar.ts()
111     14: ...
112
113     >>> ar.next()
114     'executing: "normal therapy"'
115     >>> ar.ts()
116     13: ...
117
118     >>> ar.next()
```

```
119     >>> ar.ts()
120     16: ...
121
122     >>> ar.token_pool[0].data['patient'].health
123     120
124
125
126 Testing Postconditions
127 ======================
128 There is one more thing untested: Postcondition checks.
129
130 Unluckily, our patient died...
131     >>> ar.token_pool[0].data['patient'].health = 0
132
133 And our postcondition cannot be fulfilled...
134     >>> ar.next()
135     Traceback (most recent call last):
136     ...
137     ActivityRuntimeError: PostConstraint not fulfilled: "patient.
            health >= 90"
```

Listing 9.2: Verwendung der Runtime

3 Import des Modells und der Klasse, die den Patienten repräsentiert.

4 Import der Runtime Engine

5 Instantiierung der Runtime Engine

9 Start der Runtime Engine mit dem Patienten als Übergabeparameter. Der Tokenstatus inklusive dem Daten-Payload wird mit dem Befehl *ts()* angezeigt. Die Zahl am Beginn zeigt hierbei den Namen der Kante an, die der Token referenziert.

13 Aufruf der nächsten Iteration. Die erste Diagnose wird durchgeführt.

22 Aufgrund des niedrigen Gesundheitswertes wird eine Akute Therapie durchgeführt. Die Entscheidung wurde aufgrund der Kantenbedingungen wie im Modell definiert getroffen. Im Modell wurde ”acute therapy” mit dem TaggedValue ”variation = acute” versehen. In der Execution ist ein Logik implementiert, die anhand dieses TaggedValues den Gesundheitszustand entsprechend verbessert. Das Modellelement ”normal therapy” ist mit dem TaggedValue ”variation = normal” konfiguriert und unternimmt dementsprechend andere Maßnahmen.

36-39 Aufteilung des Tokenfluss durch den Forknode.

41-46 Abarbeitung paralleler Aktionen. Die Execution der Aktion ”data acquisition” löscht dabei das Diagnoseergebnis aus dem Daten-Payload des Tokens. Ansonsten würde bei der Synchronisation und dem Zusammenführen des Daten-Payloads ein Konflikt entstehen, da zwei unterschiedliche Werte mit dem selben Schlüssel zusammengeführt würden.

70-97 Da im Modell die Schleife, die den Patienten die Therapie solange durchlaufen lässt, bis der Gesundheitszustand besser als 90% ist, nicht implementiert ist, wird diese in diesen Zeilen zur Laufzeit modelliert.

126-137 Demonstration der Evaluierung von Nachbedingungen für die Aktivität.

9.3 Modellerstellung mit einem UML-Editor

Da sich das Metamodell der AMREP an der UML2-Spezifikation orientiert, ist es prinzipiell möglich Modelle mit einem UML2-Editor zu erstellen und anschließend mit dem Package `activities.transform.xmi` zu importieren. Dabei müssen aber die im Kapitel 7.5, S.69 angegebenen Besonderheiten beachtet werden.

Die Tests im Package `activities.transform.xmi` zeigen, dass ein in XMI definieres Modell zu einem AMREP-Modell transformiert werden kann. Das Listing 9.3 zeigt, wie Modelle mit dem XMI Importer importiert werden können. Die Abbildung 9.1 stellt das Modell, das importiert wurde dar.

Der XMI Import Mechanismus hat folgende Einschränkungen:

- Als UML-Editor wurde der Eclipse UML2-Editor des *Model Development Tools*-Projekt verwendet[1]. Es wurde das aktuelle Modeling Package der Eclipse-Galileo-Distribution (3.5) verwendet[2].
- Der Eclipse UML2-Editor erlaubt keine Definition von *guard*-Bedingungen auf Kanten und keine *preconditions* und *postconditions* für Aktivitäten und Aktionen.

[1] Website: http://www.eclipse.org/modeling/mdt/

[2] Website: http://www.eclipse.org/downloads/packages/eclipse-modeling-tools-includes-incubating-components/galileor

- Die Definition von Profilen erfolgt in einer seperaten XMI Datei. Das verwendete Transformations Framework *AGX*[3] unterstützt keine XMI-Referenzen über verschiedene Dateien hinweg. Es können somit keine Profile importiert werden und keine Stereotypen auf Modellelementen angewandt werden.

```
XMI to Activities Transform
===========================
Lookup transform and read source and target
  >>> from agx.core.interfaces import ITransform
  >>> from zope.component import getUtility
  >>> transform = getUtility(ITransform, name="xmi2act")

Import handler to register them.
  >>> from activities.transform.xmi import handler

  >>> import os
  >>> sourcepath = os.path.join(\
  ...     os.path.dirname(__file__),\
  ...     'data', 'activities-testmodel.uml')

  >>> xml = transform.source(sourcepath)
  >>> xml
  <XMLNode object ...>

  >>> from agx.core import Processor
  >>> processor = Processor('xmi2act')
  >>> pkg = processor(xml, transform.target())
  >>> from activities.metamodel.elements import validate
  >>> validate(pkg)

  >>> act = pkg.activities[0]
  >>> act
  <Activity...>

  >>> act.actions
  [<... 'action1' ... 'action2' ... 'action3' ...>]

  >>> act.nodes
  [<... 'action1' ... 'action2' ... 'action3' ... 'initial node'
     ... 'activity final node' ... 'flow final node' ... 'fork
     node' ... 'merge node' ... 'join node' ... 'decision node'
     ...>]

  >>> act.edges
  [<...'1' ...'2' ...'3' ...'7' ...'5' ...'10' ...'11' ...'6'
     ...'9' ...'8' ...'4' ...>]
```

[3]Website: http://svn.plone.org/svn/archetypes/AGX/

Listing 9.3: Import von XMI Dateien

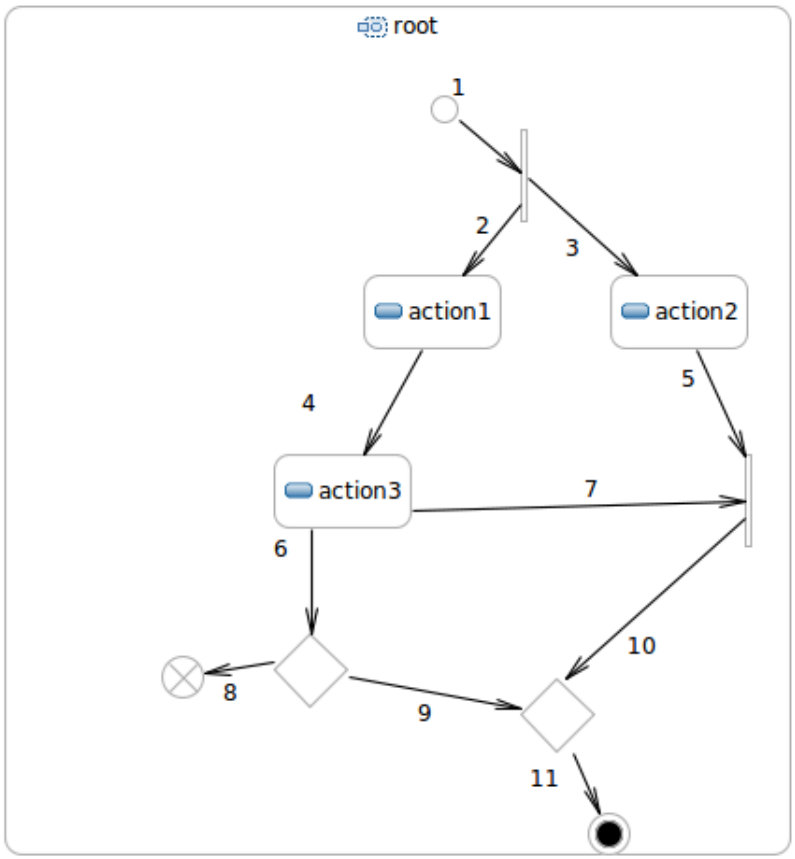

Abbildung 9.1: Modell für den XMI Import

Ein vollständiges XMI Import Framework für AMREP Modelle konnte aufgrund von Inkompatibilitäten mit den verwendeten Tools nicht implementiert werden. Weiters wäre eine solche Implementierung über den Rahmen und die Ziele der Diplomarbeit hinausgegangen.

Kapitel 10

Zusammenfassung, Ergebnisse und Ausblick

In dieser Diplomarbeit auf Grundlage der Erkenntnisse aus den vorgestellten Prozessmodellierungssprachen und den Grundlagen zur Modellierung ein Interpreter für Aktivitätsmodelle präsentiert und eine zugehörige vereinfachte und geänderte Python Implementierung des UML Metamodells für Aktivitätsdiagramme vorgestellt.

Es konnte gezeigt werden, dass der Eingangs skizzierte Usecase mit der Activity Model Runtime Engine für Python umsetzbar ist, die wesentlichsten Sprachkonstrukte zur Prozessmodellierung unterstützt und eine Änderung des Modells zur Laufzeit möglich ist.

Der vorgestellte Aktivitätsinterpreter hat Potential zur Weiterentwicklung und Integration in andere Frameworks und Projekte. Er ist auch mit diesem Ziel vor Augen entwickelt worden.

Speziell können folgende Bereiche als mögliche Erweiterungen genannt werden:

- Angleichung der Python Implementierung des Metamodells an die aktuelle UML Spezifikation.
- Erweiterung der Python Implementierung des Metamodells um nicht implementierte Elemente von UML, insbesondere Ausnahmebehandlung mit *ExceptionHandler*, Unterbrechung von Teilen der Aktivität durch *Inter-*

ruptibleActivityRegion, Senden und Empfangen von Signalen mit *SendSignalAction* und *AcceptEventAction* sowie die Unterstützung der Modellierung von Zuständigkeiten und Verantwortungsbereiche mithilfe von *ActivityPartition*.

- Verbesserung des XMI Import Mechanismus.
- Unterstützung asynchroner Executions und Zugriff auf gemeinsame Ressourcen.
- Visualisierung des Tokenstatus zur Laufzeit.

Für den produktiven Einsatz von AMREP ist es notwendig, Tests mit den speziellen Anforderungen in diesen Einsatzszenarien zu entwickeln. Beispielsweise ist noch kein Test entwickelt worden, der das Verhalten von parallelen Ausführungen von Aktivitäten überprüft.

Die AMREP wurde im Hinblick auf mögliche zukünftige Erweiterungen entwickelt. Es ist zu Erwarten, dass für einen produktiven Einsatz im Rahmen einer größeren Plattform bestimmte Anpassungen zu treffen sind. Die AMREP wurde aber als Referenzimplementierung und Studienobjekt entwickelt. Ein produktiver Einsatz wird den wahren Nutzen der *Activity Model Runtime Engine für Python* zeigen.

Teil IV

Anhang

Literaturverzeichnis

[OASIS2007 2007] *Web Services Business Process Execution Language Version 2.0*. April 2007. – URL http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf. – Zugriffsdatum: 05.08.2009. – oasis-wsbpel-v2.0.pdf

[Alanen 2007] ALANEN, M.: *A Metamodeling Framework for Software Engineering*. Turku, Finland, Åbo Akademi University, Dissertation, 2007. – URL http://tucs.fi/publications/insight.php?id=phdAlanen07a. – alanen-MFSE-DISS89.pdf

[Bernroider und Stix 2006] BERNROIDER, E. ; STIX, V.: *Grundzüge der Modellierung: Anwendungen für die Softwareentwicklung*. Wien : Facultas, 2006

[Bock 2004] BOCK, C.: UML 2 Activity and Action Models Part 4: Object Nodes. In: *Journal of Object Technology* 3 (2004), January, S. 27–41. – URL http://www.jot.fm/issues/issue_2004_01/column3. – Zugriffsdatum: 2009-09-14. – bock2004.html

[Claus und Schwill 2006] CLAUS, V. ; SCHWILL, A.: *Informatik A-Z: Fachlexikon für Studium, Ausbildung und Beruf*. Mannheim : Duden, 2006

[Crane 2009] CRANE, M. L.: *Slicing UML's Three-layer Architecture: A Semantic Foundation for Behavioural Specification*. Kingston, Ontario, Canada, Queen's University, Dissertation, January 2009. – URL http://michellelcrane.com/stuff/publications/Crane_PhD_Thesis.pdf. – Crane_PhD_Thesis.pdf

[Crane und Dingel 2008] CRANE, M. L. ; DINGEL, J.: Towards a UML virtual

machine: implementing an interpreter for UML 2 actions and activities. New York, NY, USA : ACM, 2008. – Crane-Dingel-CASCON-2008.pdf

[Gilbert 2007] GILBERT, P.: *Specification Inflation, Hyperventilation and Ultimately Consolidation*. April 2007. – URL http://blog.lombardicto.com/2007/04/index.html. – Zugriffsdatum: 04.08.2009. – gilbert-bpmn2.html

[Hitz u. a. 2005] HITZ, M. ; KAPPEL, G. ; KAPSAMMER, E. ; RETSCHITZEGGER, W.: *UML @ Work: Objektorientierte Modellierung mit UML2*. Heidelberg : dpunkt.verlag, 2005

[Martelli 2006] MARTELLI, A.: *Python in a Nutshell (In a Nutshell (O'Reilly))*. Sebastopol, California, USA : O'Reilly Media, Inc., 2006

[Muthukadan] MUTHUKADAN, B.: *A Comprehensive Guide to Zope Component Architecture*. – URL http://www.muthukadan.net/docs/zca.html. – Zugriffsdatum: 2009-09-10. – zca-0.5.7.pdf

[Object Management Group 2006] OBJECT MANAGEMENT GROUP: *Meta Object Facility (MOF) Core Specification*. January 2006. – URL http://www.omg.org/spec/MOF/2.0. – Zugriffsdatum: 09.04.2009. – omg-mof-2.0-core-specification–06-01-01.pdf

[Object Management Group 2007] OBJECT MANAGEMENT GROUP: *Business Process Model and Notation (BPMN) 2.0 Request For Proposal*. Juni 2007. – URL http://www.bpmn.org/Documents/BPMN%202-0%20RFP%2007-06-05.pdf. – Zugriffsdatum: 09.04.2009. – omg-BPMN-2-0-RFP-07-06-05.pdf

[Object Management Group 2008a] OBJECT MANAGEMENT GROUP: *Business Process Definition MetaModel, Volume II: Process Definitions*. November 2008. – URL http://www.omg.org/spec/BPDM/1.0/PDF. – Zugriffsdatum: 09.04.2009. – omg-BPDM-08-11-04.pdf

[Object Management Group 2008b] OBJECT MANAGEMENT GROUP: *Semantics of a Foundational Subset for Executable UML Models (FUML)*. November 2008. – URL http://www.omg.org/spec/FUML/1.0/Beta1/. – Zugriffsdatum: 10.08.2009. – omg-fuml–08-11-03.pdf

[Object Management Group 2009a] OBJECT MANAGEMENT GROUP: *Business Process Modeling Notation (BPMN)*. 2009. – URL http://www.omg.org/spec/BPMN/1.2. – Zugriffsdatum: 03.08.2009. – omg-BPMN-1.2–09-01-03.pdf

[Object Management Group 2009b] OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.0*. February 2009. – URL http://www.omg.org/spec/UML/2.2/Infrastructure. – Zugriffsdatum: 09.04.2009. – omg-uml-2.2-infrastructure–09-02-04.pdf

[Object Management Group 2009c] OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language (OMG UML), Superstructure*. 2009. – URL http://www.omg.org/spec/UML/2.2/Superstructure. – Zugriffsdatum: 09.04.2009. – omg-uml-2.2-superstructure–09-02-02.pdf

[Peters 2004] PETERS, T.: *The Zen of Python*. 2004. – URL http://www.python.org/dev/peps/pep-0020/. – Zugriffsdatum: 2009-09-13. – PEP-20.html

[Petri und Reisig 2008] PETRI, Carl A. ; REISIG, Wolfgang: *Petri net*. http://scholarpedia.org/, 2008. – URL http://scholarpedia.org/article/Petri_net. – Zugriffsdatum: 2009-09-14. – petri-reisig-Petri-net.html

[Pilone und Pitman 2005] PILONE, D. ; PITMAN, N.: *UML 2.0 in a Nutshell*. Sebastopol, California, USA : O'Reilly Media, 2005

[Rivard 1996] RIVARD, F.: Smalltalk: a Reflective Language. San Francisco, California : Reflection '96, April 1996. – URL http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/rivard/rivard.html. – Zugriffsdatum: 2009-08-06. – rivard.html

[van Rossum und Warsaw 2001] ROSSUM, G. van ; WARSAW, B.: *Style Guide for Python Code*. 2001. – URL http://www.python.org/dev/peps/pep-0008/. – Zugriffsdatum: 2009-09-13. – PEP-8.html

[Rumbaugh u. a. 2005] RUMBAUGH, J. ; JACOBSON, I. ; BOOCH, G.: *The Unified Modeling Language Reference Manual, Second Edition*. Boston, Massachusetts, USA : Addison-Wesley, Pearson Education, 2005

[Schattkowsky und Förster 2007] SCHATTKOWSKY, T. ; FÖRSTER, A.: On the Pitfalls of UML 2 Activity Modeling. In: *MISE '07: Proceedings of the International Workshop on Modeling in Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2007. – schattkowsky-foerster–29530008.pdf

[Schwickert und Fischer 1996] SCHWICKERT, A. C. ; FISCHER, K.: *Der Geschäftsprozeß als formaler Prozeß - Definition, Eigenschaften und Arten*. Bd. Nr. 4/1996. Mainz : Lehrstuhl für Allg. BWL und Wirtschaftsinformatik, Johannes Gutenberg-Universität, 1996. – schwickert-fischer–gp-formaler-prozess.pdf

[Stahl u. a. 2007] STAHL, T. ; VÖLTER, M. ; EFFTINGE, S. ; HAASE, A.: *Modellgetriebene Softwareentwicklung*. Heidelberg : dpunkt.verlag, 2007

[Steinberg u. a. 2009] STEINBERG, D. ; BUDINSKY, F. ; PATERNOSTRO, M. ; MERKS, E.: *EMF: Eclipse Modeling Framework (2nd Edition) (Eclipse)*. Addison-Wesley Longman, Amsterdam, January 2009

[Störrle und Hausmann 2005] STÖRRLE, H. ; HAUSMANN, J. H.: Towards a Formal Semantics of UML 2.0 Activities. In: LIGGESMEYER, P. (Hrsg.) ; POHL, K. (Hrsg.) ; GOEDICKE, M. (Hrsg.): *Software Engineering* Bd. 64, GI, 2005, S. 117–128. – SE2005-Stoerrle-Hausmann-ActivityDiagrams.pdf. – ISBN 3-88579-393-8

[Weerawarana u. a. 2005] WEERAWARANA, S. ; CURBERA, F. ; LEYMANN, F. ; STOREY, T. ; FERGUSON, D. F.: *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, March 2005

[Weilkiens und Oestereich 2004] WEILKIENS, T. ; OESTEREICH, B.: *UML 2 Zertifizierung: Test-Vorbereitung zum OMG Certified UML Professional (Fundamental)*. Heidelberg : dpunkt.verlag, 2004

[White 2004a] WHITE, S. A.: Introduction to BPMN. (2004), May. – URL http://www.bpmn.org/Documents/Introduction%20to%20BPMN.pdf. – Zugriffsdatum: 04.08.2009. – white–Introduction-to-BPMN.pdf

[White 2004b] WHITE, S. A.: Process Modeling Notations and Workflow Patterns. (2004), January. – URL http://www.bpmn.org/Documents/Notations%20and%20Workflow%20Patterns.pdf. – Zugriffsdatum: 08.08.2009. – white–Notations-and-Workflow-Patterns.pdf