E2 DANUBIA – Ein web-basiertes Modellierungs- und Entscheidungsunterstützungssystem zur Untersuchung des Globalen Wandels des Wasserkreislaufs im Einzugsgebiet der Oberen Donau - Teilprojekt Informatik

1. Überblick

DANUBIA (Barth et al., 2004) ist ein integratives Simulations- und Entscheidungsunterstützungssystem, das im Rahmen von GLOWA-Danube entwickelt wird. Mit DANUBIA können wasserbezogene Szenarien unter ökologischen und ökonomischen Gesichtspunkten untersucht werden, um Wissenschaftler und Entscheidungsträger beim Entwurf von Strategien für ein nachhaltiges Umweltmanagement zu unterstützen.

In DANUBIA wurden sechzehn Simulationsmodelle aller beteiligten Forschergruppen integriert. Damit können sowohl sektorale als auch interdisziplinäre Fragestellungen unter Einbeziehung gegenseitig abhängiger Prozesse untersucht werden.

Die Entwicklung von DANUBIA basiert auf Methoden der objektorientierten Software-Entwicklung und des Web-Engineering. Dabei spielt in allen Phasen des Entwicklungsprozesses die *Unified Modeling Language* (*UML*, Booch et al., 1999) als gemeinsame Notation, die von allen Projektpartnern zur Beschreibung der integrativen Aspekte des Systems verwendet wird, eine entscheidende Rolle.

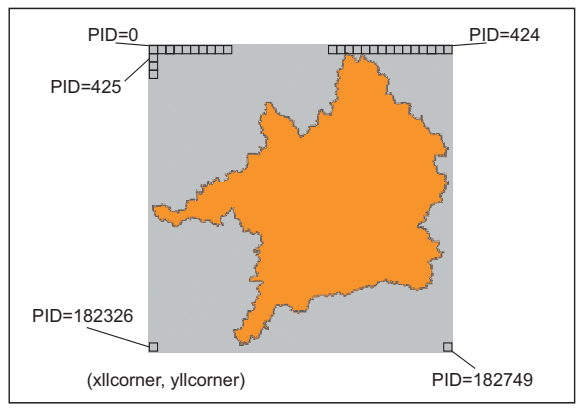

Abbildung E2.2: Das obere Donaugebiet

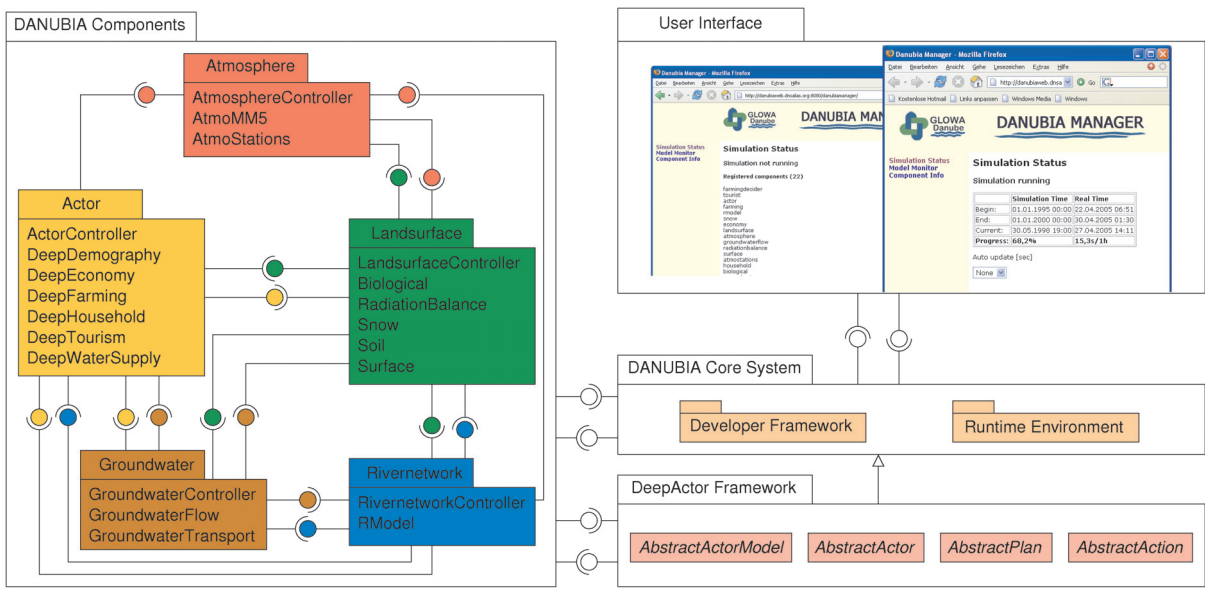

Abbildung E2.1: Die Architektur des DANUBIA-Systems

Die Architektur des DANUBIA-Systems ist in Abbildung E2.1 abgebildet. In der Komponente DANUBIA *Components* sind die Simulationsmodelle der verschiedenen Fachgruppen von GLOWA-Danube realisiert. Das DANUBIA-Kernsystem (Komponente DANUBIA *Core System*) besteht aus einem Entwickler-Framework und einer Laufzeitumgebung (siehe Abschnitt 3). Für die Implementierung sozioökonomischer Modelle stellt DANUBIA das DeepActor-Framework zur Integration von Akteuren bereit (siehe Abschnitt 5).

Aktuell läuft DANUBIA auf einem Rechnercluster mit 56 Prozessoren. Das System kann aber auch für Testzwecke auf einzelnen Rechnern oder kleineren Netzwerken installiert werden.

2. Konzept

2.1 Hauptkomponenten und Schnittstellen

Die sechzehn in DANUBIA integrierten Simulationsmodelle sind gemäß ihrer thematischen Zugehörigkeit in die fünf Hauptkomponenten *Atmosphere, Actor, Landsurface, Groundwater* und *Rivernetwork* gruppiert (siehe Abbildung E2.1). Der Datenaustausch sowohl zwischen den Hauptkomponenten als auch zwischen einzelnen Simulationsmodellen wird durch Schnittstellen spezifiziert. Die Schnittstellen beinhalten für jeden auszutauschenden Parameter den Bezeichner und Datentyp. Die zeitliche Gültigkeit der ausgetauschten Daten wird hiervon getrennt durch ein eigenes zeitliches Koordinationskonzept (siehe Abschnitt 2.3) sichergestellt.

2.2 Raumkonzept

Ein zentraler Aspekt von Umweltsimulationen betrifft die Behandlung des Simulationsraums. In DANUBIA wird der Simulationsraum durch ein zweidimensionales Netz repräsentiert (siehe Abbildung E2.2). Den Zeilen und Spalten dieses Rasters sind mittels konformer konischer Lambert-Projektion geographische Koordinaten zugeordnet. Die Koordinatenpunkte entsprechen denen des Hydrologischen Atlasses der Bundesrepublik Deutschland. Die Länge und Breite der Rasterzellen beträgt jeweils 1000 m, somit ergibt sich eine Rasterfläche von 1 km². Durch Anwendung des objektorientierten Paradigmas erhält der Simulationsraum nicht nur eine statische, sondern auch eine dynamische Struktur, welche im Wesentlichen aus den Prozessen, die an der entsprechenden Stelle des Netzes ablaufen, besteht. Dies führt zu dem Begriff Proxel (ein Akronym für process pixel). Ein Proxel-Objekt wird über eine für das betrachtete Simulationsgebiet eindeutige Identifikationsnummer, der so genannten ProxelID (siehe PID in Abbildung E2.2) identifiziert. Sämtliche Proxel-Objekte werden in einem Tabellenobjekt, der so genannten ProxelTable, verwaltet. Ein allgemeines Proxel-Objekt speichert die für alle Simulationsmodelle relevanten Eigenschaften des beschriebenen Rasterpunktes (Koordinaten, Geländehöhe, Landnutzung, etc.). Durch Spezialisierung werden in den einzelnen DANUBIA-Komponenten den Proxeln weitere, fachspezifische Eigenschaften hinzugefügt.

2.3 Zeitkonzept

Ein Simulationsmodell berechnet während der gesamten Simulationszeit zu diskreten Zeitpunkten Daten, die den jeweils gültigen Zustand des modellierten Systems beschreiben. Der Abstand zweier solcher Zeitpunkte ist pro Modell konstant und wird Modellzeitschritt genannt. Die Länge der Modellzeitschritte reicht in DANUBIA von einer Stunde (z.B. bei den Atmosphären- und Landoberflächenmodellen) bis zu einem Monat (bei den meisten Akteurmodellen). In einer integrativen Simulation werden mehrere Modelle mit unterschiedlichen Zeitschritten gekoppelt, die zyklisch Berechnungen durchführen und zur Laufzeit untereinander Daten austauschen. Um verlässliche Simulationsergebnisse zu erhalten, müssen beim Datenaustausch folgende Bedingungen erfüllt sein:

- Die ausgetauschten Daten müssen in einem stabilen Zustand sein, es darf also nicht gleichzeitig lesend und schreibend auf die Daten zugegriffen werden.
- Jedes Modell muss bei einer Datenanfrage Daten erhalten, die bezüglich seiner eigenen lokalen Modellzeit gültig sind.

Um diese Anforderungen zu erfüllen, verfügt DANUBIA über eine Komponente zur zeitlichen Koordination der einzelnen beteiligten Modelle, den so genannten *Timecontroller* (Hennicker & Ludwig, 2005 und 2006). Für die Modelle selbst ergibt sich daraus folgender Lebenszyklus (siehe Abbildung E2.3):

waitForGetData: Warten auf die Freigabe zum Einlesen von Daten von anderen Modellen durch den *Timecontroller*
getData: Einlesen von Daten von anderen Modellen
compute: Berechnen der beim nächsten Simulationszeitpunkt gültigen Daten
waitForProvide: Warten auf die Freigabe zur Bereitstellung der neu berechneten Daten für andere Modelle durch den Timecontroller
provide: Bereitstellen der neu berechneten Daten für andere Modelle

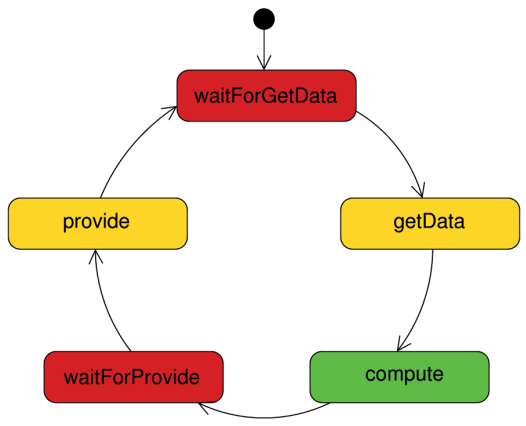

Abbildung E2.3: Lebenszyklus eines gekoppelten Simulationsmodells

3. DANUBIA-Kernsystem

Das DANUBIA-Kernsystem besteht im Wesentlichen aus einem Entwickler-Framework und einer Laufzeitumgebung (siehe Abbildung E2.4).

3.1 Entwickler-Framework

Das Entwickler-Framework unterstützt die Modellentwickler durch die Bereitstellung von Klassen und Schnittstellen, die von den Modellentwicklern benutzt werden, um ihre Modellimplementierungen in DANUBIA zu integrieren.

Zum einen sind dies so genannte Basisklassen, die gemäß dem objektorientierten Vererbungsprinzip spezialisiert werden müssen. Die wichtigste dieser Basisklassen ist *AbstractModel*, von der die konkreten Modellimplementierungen in DANUBIA abgeleitet werden. Diese Basisklasse enthält bereits die Implementierung des Lebenszyklus eines Simulationsmodells (siehe Abschnitt 2.3) und bildet die Grundlage für die Anbindung des Modells an die Laufzeitumgebung (siehe Abschnitt 3.2). In die Kategorie der Basisklassen fällt auch die Klasse *AbstractProxel*, die das in Abschnitt 2.2 beschriebene Raumkonzept realisiert.

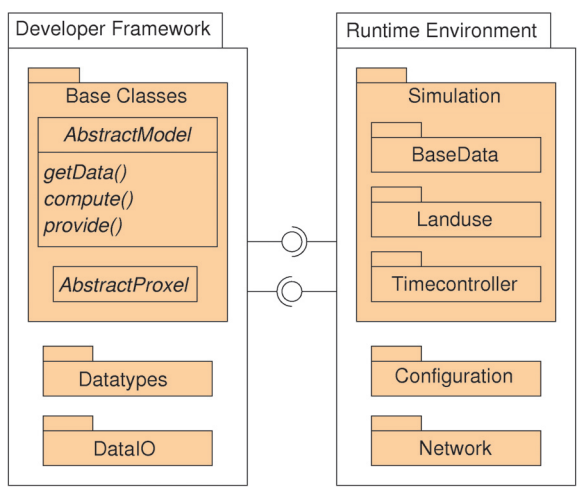

Abbildung E2.4: Das DANUBIA-Kernsystem

Weitere Klassen, die von den Modellentwicklern verwendet werden, sind z. B. die Implementierungen der in DANUBIA gemeinsam genutzten Datentypen und Werkzeuge zur Vor- und Nachbearbeitung von Eingabe- und Ausgabedaten. Mit diesen Werkzeugen können z. B. Daten zwischen üblichen GIS-Formaten und dem DANUBIA-eigenen binären Datenformat konvertiert werden, sowie zeitliche und räumliche Aggregierungen oder Mittelwertbildungen durchgeführt werden.

3.2 Laufzeitumgebung

Die Laufzeitumgebung ermöglicht einerseits integrierte Simulationsläufe des gesamten DANUBIA-Systems auf einem Rechnercluster, andererseits Testläufe von kleineren Modellkonfigurationen auf einzelnen Rechnern oder kleineren lokalen Netzwerken. Sie besteht im Wesentlichen aus den im Folgenden beschriebenen Komponenten.

Die zentrale Komponente der Laufzeitumgebung ist die Komponente *Simulation* mit ihren Subkomponenten *BaseData*, *Landuse* und *Timecontroller*. Während die Komponente *Simulation* selbst für die Verwaltung der Simulationsmodelle während einer Simulation zuständig ist, haben die Subkomponenten folgende Aufgaben: die Komponente *BaseData* dient der Initialisierung der allgemein relevanten Eigenschaften der Proxel (siehe Abschnitt 2.2), die Komponente *Landuse* verwaltet die während eines Simulationslaufs veränderliche Landnutzung des Simulationsgebiets und stellt diese den beteiligten Simulationsmodellen aktuell bereit. Schließlich realisiert die Komponente *Timecontroller* die in Abschnitt 2.3 beschriebene zeitliche Koordination der einzelnen Simulationsmodelle während eines integrativen Simulationslaufs.

Die Komponente *Configuration* dient zur Erstellung von Simulationskonfigurationen. Sie interagiert zu diesem Zweck mit der Benutzerschnittstelle (siehe Abschnitt 4). Die Verteilung der einzelnen Systemteile über ein Netzwerk ist Aufgabe der Komponente *Network*. Insbesondere ist sie für die Verbindung der korrespondierenden Datenimport- und Datenexportschnittstellen der an einer integrativen Simulation beteiligten Modelle zuständig.

4. Benutzerschnittstelle

Mit der DANUBIA-Web-Schnittstelle können verteilte Simulationsläufe konfiguriert, gesteuert und überwacht werden (siehe Komponente *UserInterface* in Abbildung E2.1). Je nach Zustand des Simulationssystems bietet die Benutzerschnittstelle verschiedene Ansichten.

Vor dem Start eines Simulationslaufs werden die Namen der angemeldeten Simulationsmodelle angezeigt. Ist die Simulationskonfiguration vollständig, kann die Simulation gestartet werden.

Nach dem Start einer Simulation lässt sich der zeitliche Fortschritt sowohl einzelner Modelle als auch der des integrierten Modellverbunds anzeigen. Der Fortschritt wird in Echtzeit und in Simulationszeit dargestellt. Weiterhin stellt die Benutzerschnittstelle detaillierte Informationen zu den einzelnen Simulationsmodellen zur Verfügung. Dazu gehören beispielsweise Performanzinformationen wie durchschnittliche Prozessor- und Speicherbelastung, aber auch Metadaten wie Autor oder Version eines Simulationsmodells.

5. DeepActor-Framework

Das DeepActor-Framework erweitert das von DANUBIA bereitgestellte Entwickler-Framework (siehe Abschnitt 3.1) durch zusätzliche Basisklassen und Schnittstellen. Es realisiert eine gemeinsame konzeptionelle Grundlage des in GLOWA-Danube entwickelten DeepActor-Ansatzes und ermöglicht den sozioökonomischen Simulationsmodellen aus GLOWA-Danube (Hauptkomponente *Actor* in Abbildung E2.1), Entscheidungsprozesse sogenannter tiefer Akteure explizit zu modellieren. Der DeepActor-Ansatz konkretisiert den Ansatz zur agentenbasierten Simulation in Sozialwissenschaften (Gilbert & Troitzsch, 2005), der wiederum auf Agentenkonzepten aus der (verteilten) künstlichen Intelligenz (Norvig & Russel, 2003; Weiss, 1999) beruht. Wir verwenden den Begriff Akteur, um Verwechslungen mit dem zumindest konzeptionell verwandten Begriff des Software-Agenten zu vermeiden. Ein Software-Agent ist ein Programm mit den Eigenschaften autonom, reaktiv, pro-aktiv und interagierend. Diese Eigenschaften werden für Software-Agenten auf technischer Ebene interpretiert, beispielsweise die Eigenschaft autonom durch die Zuordnung eines eigenen Threads oder Prozesses je Agent. Im Gegensatz hierzu sind Eigenschaften solcher Art für Akteure auf konzeptioneller Ebene zu interpretieren und damit immer abhängig von der Implementierung innerhalb eines konkreten Simulationsmodells. Ein Akteur repräsentiert handelnde Entitäten des Simulationsraums, beispielsweise Privathaushalte, Landwirte oder Betriebe touristischer Infra- und Suprastruktur. Simulationsmodelle, die das DeepActor-Framework verwenden, werden als DeepActor-Modelle bezeichnet.

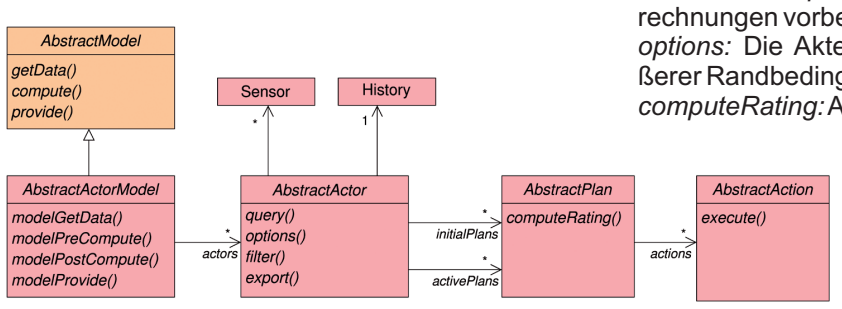

Abbildung E2.5: Statische Struktur des DeepActor-Frameworks

Die grundlegenden Modellierungselemente eines DeepActor-Modells sind in Abbildung E2.5 dargestellt. Die von DANUBIA vorgegebene Basisklasse *AbstractModel* wird durch eine spezielere Basisklasse *AbstractActorModel* verfeinert. Hinzu kommen Basisklassen für Akteure, Pläne und Aktionen. Im Folgenden wird die statische Struktur aus Abbildung E2.5, und damit die strukturelle Konzeption des DeepActor-Ansatzes von GLOWA-Danube, näher erläutert. Die dynamischen Aspekte des Ansatzes werden in den Abschnitten 5.1 und 5.2 erläutert.

Man beachte, dass abstrakte Elemente des Frameworks in der Regel optional in einem DeepActor-Modell zu konkretisieren sind. Je nach Art und Komplexität des zu realisierenden Simulationsmodells lassen sich so einfache reaktive Akteure genauso wie komplexe lernfähige Akteure auf der gemeinsamen Grundlage des DeepActor-Frameworks umsetzen.

Die konkrete Ableitung der Basisklasse *AbstractActorModel* dient im einfachsten Fall der Administration und Initialisierung der konkreten Akteure eines DeepActor-Modells. Die Akteure verfügen über Sensoren zur Wahrnehmung ihrer Umgebung, im Speziellen zum Import von Proxeldaten und zum Import von Daten anderer Akteure des gleichen Modells. Zusätzlich verfügen Akteure über eine History zur Speicherung der Entscheidungen vorangegangener Simulationsschritte. Pläne repräsentieren die Handlungsoptionen eines Akteurs und enthalten jeweils eine Menge von Aktionen, die Effekte einer Planumsetzung explizit modellieren. Die Menge der initialen Pläne (*initialPlans*) ist die gesamte Menge an Plänen, die einem Akteur über den Simulationsverlauf zur Verfügung steht. Die Menge der aktiven Pläne (*activePlans*) ist diejenige Teilmenge initialer Pläne, für deren Umsetzung sich ein Akteur in einem gegebenen Zeitschritt entschieden hat.

5.1 Lebenszyklus eines DeepActor-Modells

Der in Abbildung E2.6 dargestellte Lebenszyklus eines DeepActor-Modells verfeinert den Zyklus eines DANUBIA-Modells (siehe Abbildung E2.3). Zentral ist hierbei die Unterscheidung zwischen Modell- und Akteurberechnung. Erstere repräsentiert die Makroebene der Simulation, die mindestens für den koordinierten Datenaustausch (*modelGetData, modelProvide*) mit gekoppelten Simulationsmodellen benötigt wird, letztere simuliert die Mikroebene einzelner Entscheidungen (*filter, options*), deren Effekte von der Modellberechnung auf Makroebene berücksichtigt werden. Die Zustände *getData, compute* und *provide* aus Abbildung E2.3 werden durch das DeepActor-Framework unter Verwendung der abstrakten Operationen aus Abbildung E2.5 realisiert. Ein konkretes DeepActor-Modell liefert hierfür konkrete Implementierungen wie folgt:

modelGetData und *query:* Das Modell importiert Daten von gekoppelten Simulationsmodellen und die Akteure fragen Sensoren ab.
modelPreCompute: Das Modell kann Akteurberechnungen vorbereiten.
options: Die Akteure aktivieren die gemäß äußerer Randbedingungen umsetzbaren Pläne.
computeRating: Alle aktiven Pläne berechnen ein Bewertungsattribut.
filter: Die Menge aktiver Pläne wird aufgrund akteurspezifischer Kriterien weiter reduziert.
execute: Die Aktionen der verbliebenen aktiven Pläne werden ausgeführt.
modelPostCompute: Das Modell kann Akteurberechnungen verarbeiten.
export und *modelProvide:* Die Akteure und das Modell exportieren Daten für andere Akteure bzw. für gekoppelte Simulationsmodelle.

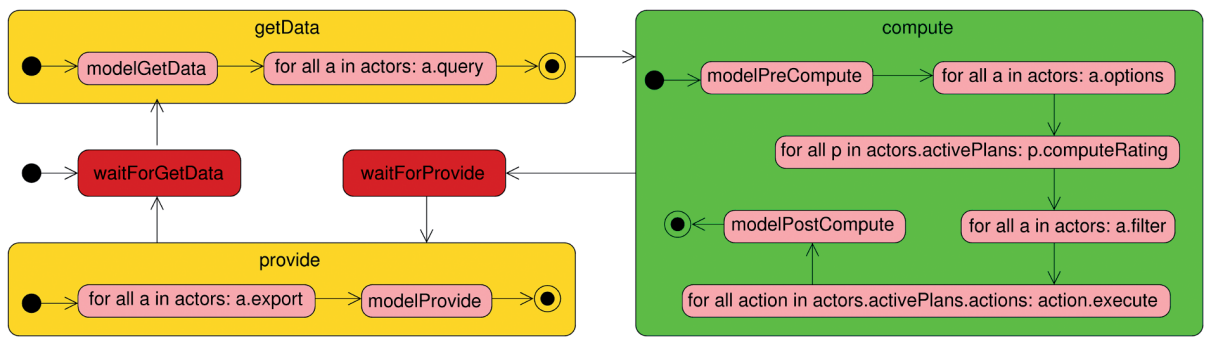

Abbildung E2.6: Lebenszyklus eines DeepActor-Modells als Verfeinerung von Abbildung E2.3

5.2 Entscheidungsprozess eines tiefen Akteurs

Die Entscheidung eines Akteurs umfasst zwei Schritte: *options* – zur Bestimmung der aktiven Planmenge im gegebenen Simulationsschritt und *filter* – zur Reduktion der aktiven Planmenge auf diejenigen Pläne, die tatsächlich ausgeführt werden sollen. Im ersten Schritt werden Pläne deaktiviert, die zum gegenwärtigen Simulationszeitpunkt aufgrund äußerer Rahmenbedingungen, auf die der Akteur keinen Einfluss hat, nicht ausführbar sind. Im zweiten Schritt berücksichtigt ein Akteur seine implizit vorhandenen Ziele und Präferenzen und bestimmt damit die in diesem Zeitschritt auszuführende Menge aktiver Pläne. Zwischengeschaltet ist die Berechnung eines optionalen Bewertungsattributs (*computeRating*). Der resultierende Wert kann als Grundlage der Planauswahl im *filter* Schritt verwendet werden. Damit lässt sich beispielsweise ein Auswahlalgorithmus basierend auf einer Multiattribute Utility Theory (Norvig & Russel, 2003) realisieren.

Ziele, die eine Entscheidung eines Akteurs motivieren, sind kein expliziter Bestandteil des DeepActor-Ansatzes. Stattdessen muss die konkrete Realisierung des *filter* Schritts die Ziele des jeweiligen Akteurs berücksichtigen.

Autoren

R. Hennicker, S. Janisch, A. Kraus, M. Ludwig
Institut für Informatik,
Ludwig-Maximilians-Universität München

Literatur

Barth, M., Hennicker, R., Kraus, A. & Ludwig, M. (2004): *DANUBIA: An Integrative Simulation System for Global Change Research in the Upper Danube Basin.* Cybernetics and Systems, Vol. 35 (7-8), pp. 639-666. Taylor&Francis.

Booch, G., Rumbaugh, J. & Jacobson, I. (1999): *The Unified Modeling Language User Guide*. Addison Wesley, Object Technology Series.

Gilbert, N. & Troitzsch, K. G. (2005): *Simulation for the Social Scientist.* 2.Ed. Open University Press.

Hennicker, R. & Ludwig, M. (2005): *Property-Driven Development of a Coordination Model for Distributed Simulations.* Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2005), LNCS 3535, pp. 290-305. Springer.

Hennicker, R. & Ludwig, M. (2006): *Design and Implementation of a Coordination Model for Distributed Simulations.* In: Mayr, H.C. and Breu, R. (editors): Proc. Modellierung 2006 (MOD'06), Volume P-82 of Lect. Notes Informatics, pp. 83-97. Gesellschaft für Informatik.

Norvig, P. & Russell, S. J. (2003): *Artificial Intelligence : A Modern Approach*. 2.Ed. Prentice Hall.

Weiss, G. (Ed.) (1999): *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* The MIT Press.