Definition und Realisierung einer Plattform zur modellbasierten Komposition von Simulationsmodellen

Dissertation
zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
der Fakultät für Informatik und Elektrotechnik
der Universität Rostock

vorgelegt von

Mathias Röhl, geb. am 7.7.1976 in Ludwigslust
aus Rostock

Rostock, 6. Februar 2008

Gutachter:

1. Prof. Dr. rer. nat. Adelinde M. Uhrmacher, Universität Rostock
2. Univ. Prof. Dr. Axel Lehmann, Universität der Bundeswehr München
3. Prof. Dr. Andreas Tolk, Old Dominion University, Norfolk, Virginia, USA

Tag der mündlichen Prüfung: 20. Mai 2008

Zusammenfassung Modular-hierarchische Modellierungsformalismen verfügen über wohldefinierte Ausdrucksmöglichkeiten zur flexiblen Konstruktion von Modellen, sind bisher jedoch vorrangig auf eine zentralisierte Modellentwicklung zugeschnitten. Ansätze zur simulationsbasierten Integration von Modellen unterscheiden demgegenüber explizit zwischen Schnittstellen und Implementierungen, offerieren jedoch wenig Hilfestellung, Modelle zu konstruieren und bieten beschränkte Flexibilität, mit Modellen zu experimentieren. Fortschritte aus dem Bereich komponentenorientierter und modellgetriebener Softwareentwicklung sowie dienstorientierter Architekturen ermöglichen, die Vorteile modellorientierter und simulationsorientierter Kompositionsansätze zu kombinieren.

Diese Arbeit konzipiert, definiert und realisiert eine Plattform zur Komposition von Simulationsmodellen. Mittels der entwickelten Komponentenplattform lassen sich Modelle, die in spezifischen Formalismen definiert sind, mit formalismusunabhängigen Schnittstellenbeschreibungen assoziieren und als parametrisierbare Komponenten verwenden. Schnittstellendefinitionen können in Form von XML-Dokumenten in Datenbanken hinterlegt werden, um Kompositionskandidaten aufzufinden, zu selektieren und zu analysieren.

Ein Metamodell formalisiert die Beschreibungsmittel der Komponentenplattform, so dass sich die Verfeinerung von Schnittstellen in Modelle und die Kompatibilität von Schnittstellen prüfen lassen.

Das Kompositionswerkzeug CoMo (Component Models) realisiert die entwickelte Komponentenplattform in der Programmiersprache Java. Mit CoMo können plattformspezifische Simulationsmodelle aus plattformunabhängigen Kompositionen automatisiert abgeleitet werden. Die Anbindung einer konkreten Simulationsumgebung wird anhand des existierenden Simulationssystems James II demonstriert.

Die entwickelte Komponentenplattform wird zur Evaluation von Dienstvermittlung in mobilen Ad-Hoc-Netzen (MANETs) praktisch angewandt.

Schlagwörter:
Modellkomponenten, Komponierbarkeit, Komposition, Wiederverwendung

Abstract Modular hierarchical modeling formalisms distinghuish clearly between simulation algorithms and model definitions and offer well-defined means to construct models flexibly. At the same time they are particularily aimed at a centralized development of models. In contrast, simulation-based integration approaches clearly separate interface descriptions from model implementations, but offer less support for constructing models and experimenting with them. A carefull combination of concepts from the area of component-oriented software development, model-driven development and service-oriented architectures allows to combine the advantages of model-based and simulation-based composition approaches.

This thesis conceives, defines, and realizes a platform for the composition of simulation models. The platform allows to associate model implementations, defined in specific formalisms, with formalism-independent interface definitions and to deploy models as parametrizable components. Interface definitions may be stored as XML documents in databases to search, select, and analyze composition candidates.

A meta model formalizes the description means of the component platform, such that the refinement of interfaces into model implementations and the compatibility of interfaces can be checked automatically.

The composition tool CoMo (*Component Models*) realizes the platform definition in the Java programming language. CoMo allows to derive platform-specific simulation models from platform-independent compositions. The integration of simulation platforms is demonstrated on the example of the existing simulation system James II.

The developed composition platform is used in a simulation study to evaluate service trading mechanisms in mobile ad-hoc networks.

Keywords:
model components, composability, composition, reuse

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Experimenteller Rahmen Der experimentelle Rahmen beschreibt die Bedingungen einer Experimentdurchführung und damit den Einsatzkontext operationaler Modelle (Zeigler 1984; Yilmaz 2004a)., 19

Interoperabilität Interoperabilität bezeichnet die Fähigkeit eines Systems, Dienste für andere Systeme nutzbar zu machen. Im Kontext der Simulation bezieht sich Interoperabilität auf die Fähigkeit von Simulationssystemen innerhalb eines Simulationsverbundes sinnvoll zu interagieren. (Petty u. Weisel 2003a), 14

Kompatibilität Kompatibilität sagt aus, ob Komponenten zusammengefügt werden können. Es lassen sich die technische, syntaktische, semantische, pragmatische, dynamische und konzeptuelle Dimension der Kompatibilität unterscheiden. (Tolk u. Muguira 2003), 2

Komponente Komponenten erfüllen eine klare Funktion, ausgedrückt durch eine wohldefinierte Schnittstelle und realisiert durch eine Implementierung. Eine Komponente sollte in unvorhergesehenen Kombinationen verwendbar sein und einen ersetzbaren sowie konfigurierbaren Teil eines Systems darstellen. (Verbraeck 2004; Szyperski 2002), 1

Komponierbarkeit Komponierbarkeit drückt das Vermögen aus, Modellkomponenten auszuwählen und in unterschiedlichen Kombination zusammenfügen zu können, so dass spezielle Anforderungen eines Nutzers erfüllt werden. (Petty u. Weisel 2003a; Yilmaz u. Ören 2006), 2

Komposition Komposition ist die Anwendung eines Kompositionsoperators entsprechend einer Kompositionstheorie auf Komponenten in einem bestimmten Kontext. (Szyperski 2003), 1

Kompositionalität Kompositionalität fordert, dass die Bedeutung einer Komposition sich allein aus den Bedeutungen der Teile sowie den Regeln ihrer Kombination bestimmt (Janssen 1997). Teile haben kompositionale Eigenschaften, wenn sich die Semantik einer Komposition aus den Semantiken der verwendeten Komponenten ableiten lässt (Szyperski 2002)., 1

Metamodell Ein Metamodell ist ein Modell einer Modellierungssprache. (Favre 2005), 34

Modell Ein Modell repräsentiert ein reales oder imaginiertes System, so dass durch das Experimentieren mit dem Modell Erkenntnisse über das repräsentierte System gewonnen werden können. (Minsky 1965), 1

Modell, konzeptuelles Ein konzeptuelles Modell stellt eine natürlichsprachliche, logische oder auch formale Repräsentation eines Systems im Rahmen einer bestimmten Fragestellung dar (Sargent 2005)., 1

Modell, operationales Ein operationales Modell ist die Implementierung eines konzeptuellen Modells, mit dem Experimente durchgeführt werden können (Sargent 2005)., 1

Plattform Eine Plattform ist eine Ausführungsumgebung für eine Menge von Modellen (Mellor u. a. 2004). Plattformunabhängig ist ein Modell, dessen Metamodell von einer oder mehreren Plattformspezifikationen abstrahiert. Plattformabhängig ist ein Modell, das Details über Plattformen beinhaltet., 34

Schnittstelle Eine Schnittstelle ist ein Modell einer Implementierung. Schnittstellen sollten genau soviel Informationen enthalten, dass Kompositionen allein auf ihrer Grundlage stattfinden können. (de Alfaro u. Henzinger 2005), 1

Validieren Validieren untersucht, ob ein Modell die Anforderungen für einen speziellen beabsichtigten Gebrauch erfüllt. Validieren prüft den Wert eines Modells aus Sicht des Nutzers und einer speziellen Verwendung. So kann es vorkommen, dass für ein Modell mehrere Validierungen erforderlich werden. (QM-Lexikon 2007a), 18

Verfeinerung Verfeinerung ist eine Beziehung zwischen Modellen, die das erlaubte Maß an Abweichung im Hinblick auf bestimmte Eigenschaften formalisieren. (Schröter 2004), 1

Verifizieren Verifizieren prüft, ob ein Modell festgelegte Forderungen erfüllt. (QM-Lexikon 2007b), 18

Akronyme

BOM *Base Object Model*, 16–18, 26, 46, 72

CBSE *Component-based software engineering*, 29, 113

CCM *CORBA Component Model*, 29–32, 41, 43, 53, 86

CoMo *Component Models*, 3, 4, 81, 82, 85–87, 91, 94–96, 98, 105, 111, 113, 129

CORBA *Common Object Request Broker Architecture*, 29–31, 34

DEVS *Discrete Event System Specification*, 5–12, 20, 25, 26, 31, 47, 48, 51, 53, 57, 60, 61, 70, 75, 82, 94

DIANE Dienste in Ad-Hoc-Netzen, 4, 6, 75, 99, 100, 102

EJB *Enterprise Java Beans*, 86

FOM *Federation Object Model*, 16

HLA *High Level Architecture*, 14–18, 26, 29, 34, 38, 57, 113

IDL *Interface Definition Language*, 29–32, 37, 38, 41–43, 46, 72

James II *Java-based Agent Modeling Environment for Simulation*, 3, 59, 78, 91, 92, 94, 96, 98, 102, 108, 111, 129

JAXB *Java Architecture for XML Binding*, 82, 83, 85, 92, 95, 111

LCIM *Levels of Conceptual Interoparability Model*, 14

LTS *Labelled Transition System*, 23–25, 39, 40, 43, 112

MANET *Mobile Ad-Hoc Network*, 4, 6, 8, 10, 36, 58, 99, 108

MB *Model Base*, 10

MDA *Model Driven Architectur*, 33, 34, 42, 46, 57

OMT *Object Model Template*, 15–17, 58

ORB *Object Request Broker*, 31

OWL-S *Web Ontology Language for Web Services*, 41

PDEVS *Parallel DEVS*, 6–8, 45, 59–61, 66, 69, 75, 77, 78, 81, 87, 91, 92, 95, 96, 98, 102, 111, 113, 116

RDF *Resource Description Framework*, 40, 43, 74

RTI *Run Time Infrastructure*, 15, 38, 58

SAWSDL *Semantic Annotations for WSDL*, 41, 43, 74, 111

SBML *Systems Biology Markup Language*, 14, 26

SCXML *State Chart XML*, 94–96, 100, 101, 111

SES *System Entity Structure*, 10

SOA *Service Oriented Architecture*, 35, 36, 42, 113

SOM *Simulation Object Model*, 16

SysML *Systems Modeling Language*, 33, 34, 43, 48, 51, 53, 60, 63, 79, 129

UML *Unified Modeling Language*, 32–34, 42, 43, 46, 48, 49, 51, 53, 57, 60, 72, 79, 94, 113, 129

URI *Uniform Resource Identifier*, 40, 42, 43, 74, 85, 86, 111

VIMS *Visual Interactive Modelling Systems*, 5, 8, 16

WSDL *Web Service Description Language*, 37, 38, 40–43, 46, 48, 51, 73, 129

WWW *World Wide Web*, 35, 36, 38, 40

W3C *World Wide Web Consortium*, 36, 37, 39, 73, 80, 100, 111, 129

XML *eXtensible Markup Language*, 3, 9, 10, 13, 16–18, 35–38, 40, 42, 43, 45, 46, 72–75, 77, 79–82, 85–87, 89, 91, 92, 94, 96, 100–102, 111, 113, 129

XSD *XML Schema Definition*, 37, 40, 41, 72–75, 78, 80–82, 90–92, 94, 95, 100, 101, 111, 129

XSLT *XSL Transformations*, 77, 85

1 Einleitung

Mit steigender Größe und Komplexität von Simulationsmodellen wächst der Wunsch, diese zumindest teilweise aus vorgefertigten Modellkomponenten zusammensetzen zu können. Modellbausteine wiederzuverwenden, verspricht die Glaubwürdigkeit von Modellen zu erhöhen (Davis u. Anderson 2004) und die Durchführung von Simulationsstudien zu vereinfachen und zu beschleunigen (Davis u. Anderson 2004; Valentin u. a. 2003a;b). Damit sich diese positiven Effekte einstellen, bedarf es jedoch spezieller Techniken.

Kompositionale Techniken

Eine komplexe Fragestellung in einzelne, jeweils leichter handhabbare Teilprobleme zu strukturieren, kann als Ausgangspunkt dienen, Teillösungen unabhängig voneinander zu erstellen und in Form von Komponenten verfügbar zu machen. Um Integrationsaufwände zu minimieren, ist es wünschenswert, nicht sämtliche Details der Komponenten beim Zusammenfügen berücksichtigen zu müssen, sondern deren wesentliche Eigenschaften in besser überschaubaren und leichter handhabbaren Beschreibungen, genannt Schnittstellen, zusammenzufassen.

Die wesentliche Herausforderung bei der Integration von Teillösungen besteht darin, sicherzustellen, dass eine Kombination von Bausteinen in Hinblick auf einen Anwendungszweck sinnvoll ist. Um dies zu ermöglichen, sollte in Kompositionstheorien definiert sein, welche Arten von Eigenschaften relevant sind und welche Mittel zur Kombination genutzt werden. Im Rahmen einer Kompositionstheorie bezeichnet Komposition die Anwendung eines Kompositionsoperators auf Komponenten in einem gegebenen Kontext (Szyperski 2003).

In Anlehnung an den Begriff *building block* (Verbraeck 2004) wird eine **Komponente** im Folgenden als abgeschlossene Einheit verstanden, die für ihre Umgebung nützliche Funktionalität bereitstellt. Die Funktion einer Komponente wird durch eine wohldefinierte Schnittstelle ausgedrückt und durch eine Implementierung realisiert. Eine Komponente ist ein konfigurierbarer und ersetzbarer Teil eines Systems und muss in unvorhergesehenen Kombinationen verwendbar sein (Szyperski 2002).

Das Prinzip der Kompositionalität fordert von Repräsentationen, dass sich die Bedeutung eines komponierten Systems allein aus den Bedeutungen der verwendeten Bausteine und den syntaktischen Regeln des Zusammensetzens bestimmen lässt (Janssen 1997). Grundvoraussetzung dafür ist, dass Implementierungen und Schnittstellen einer Komponente in einer bestimmten Beziehung zueinander stehen. Verfeinerungsrelationen formalisieren das erlaubte Maß an Abweichung zwischen abstrakten und konkreten Beschreibungen hinsichtlich bestimmter Eigenschaften. Generell gilt, dass Schnittstellen genau soviel Informationen beinhalten sollten, um Komponenten allein auf der Grundlage von Schnittstellen nutzen zu können (de Alfaro u. Henzinger 2005).

Komposition von Simulationsmodellen

Simulation nutzt Modelle um Experimente durchzuführen (Becker u. a. 2005). Ein Modell repräsentiert ein reales oder imaginiertes System, so dass durch das Experimentieren mit dem Modell, Erkenntnisse über das repräsentierte System gewonnen werden können (Minsky 1965). Damit ein Modell bezüglich einer bestimmten Fragestellung vertrauenswürdig ist, muss es wesentliche Eigenschaften des repräsentierten Systems widerspiegeln. Welche Systemeigenschaften als wesentlich erachtet werden, hängt von den beabsichtigten Beobachtungen ab. Bildet ein Modell die wesentlichen Eigenschaften eines Systems hinsichtlich einer Fragestellung korrekt ab, bezeichnet man es als valide für diese Fragestellung (Balci u. a. 2002).

Natürlichsprachliche, logische oder auch formale Repräsentationen eines Systems im Rahmen

einer bestimmten Fragestellung werden auch als konzeptuelle Modelle bezeichnet (Sargent 2005). Operationale Modelle müssen darüberhinaus in ausführbarer Form vorliegen, so dass Simulationsläufe durchgeführt werden können.

Für die Gesamtheit der Herausforderungen, die sich bei der Komposition von Simulationsmodellen stellen, wird in der Modellierung und Simulation der Begriff der Komponierbarkeit verwendet (Zimmerman u. a. 2002; SISO 2006a; Yilmaz u. Ören 2006). Nach Petty u. Weisel (2003a) bezeichnet **Komponierbarkeit** die Qualität, zum Komponieren geeignet bzw. im Hinblick auf eine Komposition nützlich zu sein. Komponierbarkeit fordert, Komponenten zu verschiedenen Systemen bzgl. unterschiedlicher Fragestellungen kombinieren und rekombinieren zu können.

Ein notwendiges Kriterium für die Komponierbarkeit ist, dass die verwendeten Komponenten bzgl. ihrer Abstraktionen, Annahmen und Rahmenbedingungen zueinander kompatibel sind (Davis u. Anderson 2004; Overstreet u. a. 2002). Kompatibilität von Komponenten wird auch als ingenieurstechnischer Aspekt der Komponierbarkeit bezeichnet (Weisel u. a. 2004).

Für die Integration von Simulationsmodellen muss insbesondere sichergestellt sein, dass die in einem Modell getätigten Abstraktionen mit den Abstraktionen anderer Modelle verträglich sind (Muguira u. Tolk 2006; Yilmaz u. Ören 2006; Pullen u. a. 2005). Schnittstellen sollten daher beschreiben, welche Systeme durch Modelle repräsentiert werden, welche Annahmen für die Repräsentationen getroffen wurden und welche Anforderungen an den Einsatzkontext gestellt werden (DoD 2003; SCM 2006).

Während Verfeinerung und Kompatibilität unabhängig von einem konkreten Einsatzkontext prüfbar sind, kann Validität nur in Bezug auf einen konkreten Einsatzkontext hergestellt werden (Yilmaz u. Ören 2006). Validität drückt sich nicht im Verhältnis der verwendeten Komponenten untereinander aus, sondern ist durch das Verhältnis des ganzen Simulationsmodells zu einer Fragestellung bestimmt. Ob eine Komposition einen geforderten Zweck erfüllt, wird auch als semantischer Aspekt der Komponierbarkeit bezeichnet (Weisel u. a. 2004).

Um der praktischen Forderung, Komponenten via Bibliotheken aufzufinden, zu selektieren und wiederzuverwenden (Verbraeck 2004; Heisel u. a. 2004; Kasputis u. Ng 2000), gerecht zu werden, bedarf es letztendlich ausreichend deskriptiver Schnittstellenbeschreibungen, auf deren Basis sich sowohl der ingenieurstechnische als auch der semantische Aspekt der Komponierbarkeit in angemessener Zeit analysieren lassen.

1.1 Zielsetzung

Die vorliegende Arbeit zielt auf die Entwicklung einer Plattform zur Komposition operationaler Simulationsmodelle. Abbildung 1.1 visualisiert den angestrebten Zusammenhang zwischen schnittstellenbasierter Komposition und wesentlichen Aktivitäten einer Simulationsstudie (Law u. Kelton 2000). Die Anforderungen an den zu entwickelnden Kompositionsansatz gliedern sich entsprechend der Aktivitäten in:

- *Publizieren, Auffinden & Spezifizieren:* Schnittstellen müssen als Spezifikation für eine Implementierung fungieren und in öffentlichen Bibliotheken hinterlegt werden können. Auf ihrer Basis müssen Modelle ausgewählt sowie Kompositionen erstellt und überprüft werden können. Schnittstellenbeschreibungen sollten unabhängig von einem konkreten Modellierungsformalismus oder Simulationssystem sein.

- *Implementieren & Verifizieren:* Komponenten sollten dezentral und unabhängig voneinander, in unterschiedlichen Implementierungssprachen realisierbar sein. Komponenten sollten sich konfigurieren lassen. Die Verfeinerung von Schnittstellen in Implementierungen sollte formal prüfbar sein.

- *Komponieren & Analysieren:* Auf Grundlage von Schnittstellenbeschreibungen sollen Kompositionen möglichst umfassend und automatisch auf Kompatibilität und Validität überprüfbar

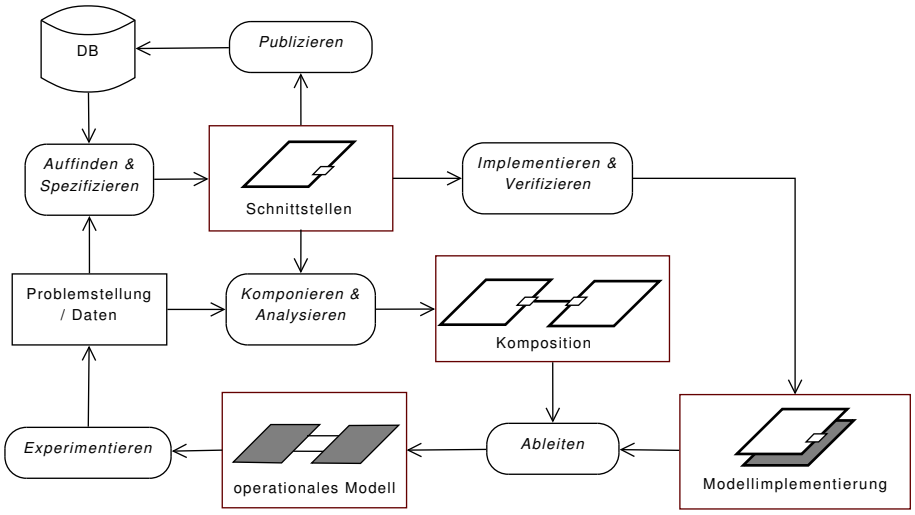

Abbildung 1.1: Aktivitäten und Objekte zur Komposition von Simulationsmodellen

sein.

- *Ableiten & Experimentieren:* Aus einer Komposition muss ein operationales Modell ableitbar sein, das erlaubt, für eine Fragestellung relevante Daten zu beobachten. Das Simulationsmodell sollte möglichst effizient ausführbar und flexibel kontrollierbar sein.

Ziel dieser Arbeit ist ein Ansatz zur Komposition von Simulationsmodellen, in dessen Rahmen sich alle aufgeführten Anforderungen adressieren lassen. Entsprechend der Abbildung 1.1 fallen dabei Schnittstellenbeschreibungen und der Möglichkeit, Kompositionen allein auf der Grundlage von Schnittstellen zu analysieren, die zentralen Rollen zu.

1.2 Methodik

Als Ausgangspunkt der Arbeit werden existierende Ansätze zur Komposition von Simulationsmodellen vorgestellt und in Hinblick auf obige Anforderungen verglichen. Praktische Kompositionslösungen fokussieren vor allem auf die Kompatibilität von Modellen und basieren auf modular-hierarchischen Modellierungsformalismen oder auf Ansätzen zur Kopplung von Simulationssystemen. Komplementiert werden diese durch zwei theoretische Ansätze, mit denen die Validität von zusammengesetzten Modellen formal analysiert werden kann.

Der Wunsch, die Vorteile von modellorientierten und simulationsorientierten Ansätzen zu vereinen, motiviert einen ausführlichen Blick auf Ansätze zur Komposition von Softwaresystemen. Dazu werden grundlegende Konzepte und Techniken komponentenbasierter und modellgetriebener Softwareentwicklung sowie dienstorientierter Architekturen beleuchtet und auf ihre Verwendbarkeit für die Komposition von Simulationsmodellen untersucht.

Den Kernteil der Arbeit bildet die Konzeption und Definition einer Plattform zur modellbasierten Komposition von Simulationsmodellen. Die Syntax von Beschreibungsstrukturen der Plattform wird mengentheoretisch vorgestellt. Parallel DEVS (Zeigler u. a. 2000) fungiert als semantische Domäne und als Grundlage zur Formulierung kompositionaler Eigenschaften. Um Schnittstellen und Komponenten konkret einsetzen zu können, wird die abstrakte Syntax, in Anlehnung an dienstorientierte Architekturen, auf eine Repräsentation in der *eXtensible Markup Language* (XML) abgebildet. Dafür kommen *XML Schema Definitions* (W3C 2004f) zum Einsatz, die mittels UML-Klassendiagrammen visualisiert werden.

Schließlich wird für die Plattformdefinition ein konkretes Werkzeug konzipiert und realisiert. Das Kompositionswerkzeug *Component Models* (CoMo) ermöglicht, Komponenten einzusetzen und ein operationales Modell zu generieren. Dafür wird eine exemplarische Anbindung an das existierende Simulationssystem *Java-based Agent Modeling Environment for Simulation* (James II) umgesetzt (Himmelspach u. Uhrmacher 2007).

Der Vergleich existierender Ansätze und die entwickelten Konzepte werden durchgängig anhand eines Anwendungsbeispiels erläutert. Als Anwendungszenario dient die Simulation eines mobilen Ad-Hoc-Netzwerkes, engl. *Mobile Ad-Hoc Network* (MANET), im Kontext des Projekts Dienste in Ad-Hoc-Netzen (DIANE) (Diane 2007).

1.3 Strukturierung der Ausarbeitung

Der Hauptteil der Arbeit besteht aus fünf Teilen. Das anschließende Kapitel arbeitet den aktuellen Stand zur Komposition von Simulationsmodellen auf. Ansätze aus dem Bereich von Softwarekomponenten und Web Services werden in Kapitel 3 vorgestellt und verglichen. Eine eigene Kompositionsplattform wird in Kapitel 4 konzipiert und definiert. Kapitel 5 stellt das Kompositionswerkzeug CoMo vor. Den Einsatz der entwickelten Komponentenplattform illustriert Kapitel 6 am Anwendungsbeispiel DIANE. Im Anschluss werden die Ergebnisse der Arbeit zusammenfassend dargestellt und Ausblicke auf mögliche Anschlussarbeiten gegeben. Anhang A enthält Beweise zur Plattformdefinition. Die Schemadefinitionen für Schnittstellen, Komponenten und Modelle listet Anhang B auf.

2 Komposition von Simulationsmodellen

Die existierenden Modellrepräsentationen und Simulationswerkzeuge geben die Möglichkeiten vor, in deren Rahmen Simulationsmodelle komponiert werden können. Visuell-interaktive Modellierungssysteme, engl. *Visual Interactive Modelling Systems* (VIMS), dominieren derzeit die „Landschaft“ der Simulationswerkzeuge (Pidd u. Carvalho 2006). VIMS verfügen über grafische Benutzungsoberflächen, in denen Modelle durch Ikonen repräsentiert, am Bildschirm platziert und miteinander verbunden werden können. Modelle lassen sich in Bibliotheken zusammenfassen, die auf spezielle Anwendungsdomänen, wie z.B. Fertigungssysteme oder Logistiksysteme, zugeschnitten sind. In Hinblick auf die Komposition komplexer Simulationsmodelle leiden VIMS jedoch unter einem prinzipiellen Schwachpunkt. VIMS wie Arena (Kelton u. a. 1998) offerieren relativ einfache und domänenspezifische Modellierungskonstrukte (Valentin u. a. 2003a). So stellt es für VIMS eine Herausforderung dar, Modelle aus unterschiedlichen Anwendungsgebieten miteinander zu kombinieren (Pidd u. Carvalho 2006).

Die Alternativen zur werkzeug- und domänenspezifischen Modellierung lassen sich nach drei prinzipiell verschiedenen Herangehensweisen klassifizieren. Es ist möglich, zur Komposition von Simulationsmodellen

- allgemeine, modular-hierarchische Modellierungsformalismen zu nutzen,
- Modelle über die Kopplung von Simulationssystemen zu integrieren und
- Modelle auf abstrakter Ebene mit Formalismen zu analysieren, die unabhängig von konkreten Modellierungssprachen sind.

Diese drei Möglichkeiten werden am Beispiel repräsentativer Vertreter und im Hinblick auf den aufgestellten Anforderungskatalog[1] vorgestellt. Die Begriffe Komponente und Komponierbarkeit werden im Weiteren so genutzt, wie auf Seite 1f. eingeführt.

2.1 Modular-hierarchische Modellierung

Herausragende Vertreter modular-hierarchischer Modellierungsansätze sind die Sprache Modelica und der Formalismus *Discrete Event System Specification* (DEVS). Diese formulieren explizit den Anspruch, komponentenorientiert zu sein (Elmqvist u. a. 2001; Zeigler u. Sarjoughian 1999), verfügen jeweils über eine breite Nutzerbasis sowie regelmäßig stattfindende internationale *Workshops* (Modelica 2006; DEVS 2007).

Modelica (Elmqvist u. a. 2001) ist eine objektorientierte Modellierungssprache. Modelle werden in Modelica als parametrisierbare Klassen definiert, die getypte Ports bekanntgeben. Vorhandene Modelle können über Ports gleichen Typs verbunden und zu komplexeren Modellen zusammengesetzt werden. Die Substitution von Untermodellen beruht auf Vererbungsbeziehungen zwischen Klassen. Modelle werden in Modelica unabhängig von einem konkreten Werkzeug in standardisierter Form repräsentiert und lassen sich so zwischen unterschiedlichen Simulationswerkzeugen austauschen. Modelicas besondere Stärken liegen im Bereich physikalischer Systeme, vor allem mit mechanischen, hydraulischen und elektronischen Aspekten, für die umfangreiche Modellbibliotheken existieren.

[1] siehe Abschnitt 1.1

Vollständig losgelöst von einem konkreten Anwendungsgebiet ist der Formalismus DEVS, an dessen Beispiel im Folgenden die Prinzipien modular-hierarchischer Modellkonstruktion beleuchtet werden.

2.1.1 Der Formalismus DEVS

DEVS wird als universeller Formalismus für die diskret-ereignisorientierte Modellierung propagiert (Zeigler u. a. 2000). Modelle werden in DEVS vergleichbar zu Zustandsautomaten mittels Mengen und Funktionen definiert. Basis ist die explizite Definition von erlaubten Ein- und Ausgaben eines Modells. Die Eingabemenge X und die Ausgabemenge Y werden in der allgemeinen Form des Formalismuses als einfache Mengen definiert. In der Praxis werden X und Y üblicherweise in Ports mit separaten Wertebereichen strukturiert (Zeigler u. a. 2000). Die Kommunikation zwischen Modellen ist dann nur über Kopplungen statthaft, die Modellports gleichen Typs verbinden.

Im Folgenden wird eine parallele Variante von DEVS vorgestellt, die auf Chow und Zeigler (1994) zurückgeht. *Parallel DEVS* (PDEVS) ist ausdrucksäquivalent zu DEVS (Zeigler u. a. 2000, 392f.). Dies gilt auch für andere DEVS-Derivate, bspw. zur Definition variabler Strukturmodelle (Uhrmacher 2001).

Definition 2.1. Ein atomares **PDEVS-Modell** mit Ports ist definiert durch ein Tupel

$$\mathscr{M} = (X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta),$$

wobei

- $X = \{(i, v) | i \in \mathit{InPorts}, v \in X_i\}$ die Menge von Ereignissen, die als Eingaben empfangen werden können, strukturiert durch eine Menge von Eingabeports *InPorts* mit zugehörigen Wertebereichen X_i, für alle Ports $i \in \mathit{InPorts}$,
- $Y = \{(o, v) | o \in \mathit{OutPorts}, v \in Y_o\}$ die Menge von Ereignissen, die als Ausgaben vom Modell produziert werden können, strukturiert in Ports und Wertebereiche,
- S eine Menge von *sequentiellen* Zuständen,
- $\delta_{int} : S \to S$ die interne Zustandsüberführungsfunktion,
- $\delta_{ext} : Q \times X^b \to S$, die externe Zustandsüberführungsfunktion, mit der Multimenge X^b über Elemente von X sowie $Q = \{(s, e) : e \in \mathbb{R}^{\geq 0}, s \in S, 0 \leq e < ta(s)\}$,
- $\delta_{con} : S \times X^b \to S$ die konfluente Überführungsfunktion,
- $\lambda : S \to Y^b$ die Ausgabefunktion, mit Y^b als Multimenge über Elemente von Y,
- $ta : S \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ die Zeitfortschrittsfunktion ist.

PDEVS teilt die Definition der Zustandsüberführung auf drei Funktionen auf. Die interne Zustandsüberführungsfunktion δ_{int} definiert den proaktiven Teil der Dynamik eines Modells. Die externe Überführungsfunktion δ_{ext} definiert, wie auf Eingabeereignisse in Abhängigkeit vom aktuellen Zustand reagiert wird. Tritt ein Konflikt durch Gleichzeitigkeit von interner und externer Zustandsüberführung auf, wird die konfluente Zustandsüberführungsfunktion ausgeführt. Interne Zustandsüberführungen werden nach dem Verstreichen der durch die Zeitfortschrittsfunktion ta definierten Zeit durchgeführt. Ausgaben werden entsprechend der Ausgabefunktion λ in Abhängigkeit vom aktuellen Zustand produziert. PDEVS arbeitet auf einer kontinuierlichen Zeitbasis. S umfasst die Menge der so genannten sequentiellen Zustände. In der Menge Q werden die sequentiellen Zustände aus S mit der Variable e gepaart, die angibt, wieviel Simulationszeit seit der letzten Zustandsüberführung verstrichen ist.

Beispiel 2.1. *Für das Anwendungsbeispiel DIANE*[2] *wird nun ein einfaches Protokoll zur Dienstsuche in MANETs als atomares PDEVS-Modell definiert. Dienstsuche ist in dem Modell Flooding durch das Fluten des gesamten Netzes mit Suchnachrichten realisiert.*

Die Eingabemenge und Ausgabemenge sind in jeweils zwei Ports strukturiert. Die Kommunikation mit einem Nutzermodell kann über die Ports mit den Namen „call“ und „response“ stattfinden. Flooding abstrahiert für die Definition des Wertebereichs von Ports von den internen Strukturen der Nachrichten und nimmt ServiceSearchCall, ServiceSearch usw. als gegebene Mengen an. Über „msgOut“ und „msgIn“ kommuniziert das Modell mit der Netzwerkschicht. Die Zustandsmenge S ist in Unterzustände gegliedert, so dass sie eine Phase, die aktuell zu wartende Zeit und bereits empfangene Dienstsuchen umfasst. Einmal empfangene Dienstsuchen werden in Received gespeichert. Nachrichten, die im fortlaufenden Fluten des Netzwerks zum wiederholten Male eintreffen, werden ignoriert. Damit kann verhindert werden, dass Suchprozesse in Endlosschleifen resultieren.

Formal ist das Modell definiert durch $Flooding = (X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta)$, *mit*

- $X = \{(i, v) | i \in InPorts, v \in X_i\}$, *mit*
  - $InPorts = \{$„*call*“, „*msgIn*“$\}$,
  - $X_{call} = Call = ServiceOfferCall \cup ServiceSearchCall$,
  - $X_{msgIn} = Message = ServiceSearch \cup ServiceSearchResult$,
- $Y = \{(i, v) | i \in OutPorts, v \in Y_i\}$, *mit*
  - $OutPorts = \{$„*response*“, „*msgOut*“ $\}$,
  - $Y_{response} = ServiceSearchResponse$,
  - $Y_{msgOut} = Message$,
- $S = Phase \times \mathbb{R}^{\geq 0} \cup \{\infty\} \times Received \times ToSend$, *mit*
  - $Phase = \{$„*idle*“, „*searching*“, „*waiting*“, „*forwarding*“, „*answering*“$\}$,
  - $Received = \{m : m \subseteq ServiceSearch\}$,
  - $ToSend = Call \cup Message$.

  Ein Element $s \in S$ *wird im Folgenden als* $s = (p, t, R, ts)$ *genutzt, mit* $p \in Phase$, $t \in \mathbb{R}^{\geq 0} \cup \{\infty\}$, $R \in Received$ *und* $ts \in ToSend$.
- $\delta_{int}((p, t, R, ts)) = \begin{cases} (\text{„}waiting\text{“}, 3.0, R, ts) & falls\ p = \text{„}searching\text{“} \\ (\text{„}answering\text{“}, 0, R, ssr) & falls\ p = \text{„}forwarding\text{“} \\ (\text{„}idle\text{“}, \infty, R, ts) & sonst, \end{cases}$

  mit $ssr \in ServiceSearchResult$,
- $\delta_{ext}(((p, t, R, ts), e), x) =$

$$\begin{cases} (\text{„}idle\text{“}, \infty, R, ts) & falls\ \exists (i, m) \in x.m \in ServiceOfferCall \\ (\text{„}searching\text{“}, 0, R, ss) & falls\ \exists (i, m) \in x.m \in ServiceSearchCall \\ (\text{„}forwarding\text{“}, 0, R \cup \{m\}, m) & falls\ \exists (i, m) \in x.m \in ServiceSearch \wedge m \notin R \\ (\text{„}waiting\text{“}, t - e, R, ssr) & falls\ \exists (i, m) \in x.m \in ServiceSearchResult, \end{cases}$$

  mit $ss \in ServiceSearch$ *und* $ssr \in ServiceSearchResponse$,
- $\delta_{con}((s, e), x) = \delta_{ext}((\delta_{int}(s), e), x)$,
- $\lambda((p, t, R, ts)) = \begin{cases} \{(\text{„}msgOut\text{“}, ts)\} & falls\ p = \text{„}searching\text{“} \\ \{(\text{„}response\text{“}, ts)\} & falls\ p = \text{„}waiting\text{“} \\ \{(\text{„}msgOut\text{“}, ts)\} & falls\ p = \text{„}forwarding\text{“} \\ \{(\text{„}msgOut\text{“}, ts)\} & falls\ p = \text{„}answering\text{“}, \end{cases}$

[2] Für eine Einführung siehe Kapitel 6

- $ta((p, t, R, ts)) = t$.

Die Semantik von PDEVS-Modellen ist durch die Abbildung auf eine systemtheoretische Beschreibungsform eindeutig definiert[3]. Die formale Definition erlaubt andere Modellformalismen auf (P)DEVS abzubilden und (P)DEVS als vereinheitlichenden Formalismus zu propagieren (Zeigler u. a. 2000; Vangheluwe 2000).

Zur Modularisierung von Modellstrukturen stellt PDEVS gekoppelte Modelle bereit. Die Definition der Ein- und Ausgabemengen von gekoppelten Modellen entsprechen denen atomarer Modelle. In der allgemeinen Form gekoppelter Modelldefinitionen wird der Zusammenhang zwischen Teilmodellen durch eine Ausgabeübersetzungsfunktion spezifiziert. Die Ausgabeübersetzungsfunktion gibt an, wie Ereignisse von einem Modell zu einem anderen Modell gelangen (Zeigler u. a. 2000). Auf der Basis von Ports lässt sich die Ausgabeübersetzungsfunktion in drei Mengen von Port-zu-Port-Kopplungen aufgliedern. Gekoppelte PDEVS-Modelle können mit Hilfe von Kopplungen als Blockdiagramme definiert werden und eignen sich so für die Modellierung in VIMS.

Definition 2.2. Ein gekoppeltes PDEVS-Modell ist definiert durch ein Tupel

$$\mathscr{N} = (X, Y, D, \{M_d\}, EIC, EOC, IC),$$

mit

- $X = \{(i, v) | i \in InPorts, v \in X_i\}$ die Menge von Eingabeports mit Wertebereichen,
- $Y = \{(i, v) | i \in OutPorts, v \in Y_i\}$ die Menge von Ausgabeports mit Wertebereichen,
- D die Menge der Namen der Untermodelle,
- $\{M_d | d \in D\}$ die Menge an Untermodellen,
- $EIC \subseteq \{(\text{„this“}, p, d, p') | p \in InPorts_{\mathscr{N}}, d \in D, p' \in InPorts_{M_d}\}$ die Menge von Kopplungen, die Eingabeports von $\mathscr{N}$ mit Eingabeports von Untermodellen verbinden,
- $IC \subseteq \{(d, p, d', p') | d, d \in D \wedge p \in OutPorts_{M_d} \wedge p' \in InPorts_{M_{d'}}\}$ die Menge von Kopplungen, die Ausgabeports von Untermodellen mit Eingabeports anderer Untermodelle verbinden und
- $EOC \subseteq \{(d, p, \text{„this“}, p') | d \in D, p \in OutPorts_{M_d}, p' \in OutPorts_{\mathscr{N}}\}$ die Menge von Kopplungen, die Ausgabeports von Untermodellen mit Ausgabeports von $\mathscr{N}$ verbinden.

Es müssen folgende Bedingungen eingehalten werden:

1. Wohlgeformte Untermodelle: $\forall d \in D$ ist M_d ein atomares oder gekoppeltes PDEVS-Modell mit Ports.
2. Keine direkte Rückkopplung: $\forall (d, p, d', p') \in IC.d \neq d'$.
3. Die Wertebereiche der Quellports müssen eine Teilmenge der Wertebereiche der Zielports sein:
   a) $\forall (\text{„this“}, p, d', p') \in EIC.X_{p,\mathscr{N}} \subseteq X_{p',M_{d'}}$
   b) $\forall (d, p, d', p') \in IC.Y_{p,M_d} \subseteq X_{p',M_{d'}}$
   c) $\forall (d, p, \text{„this“}, p') \in EOC.Y_{p,M_d} \subseteq Y_{p',\mathscr{N}}$.

[3] siehe Abschnitt 2.3.1

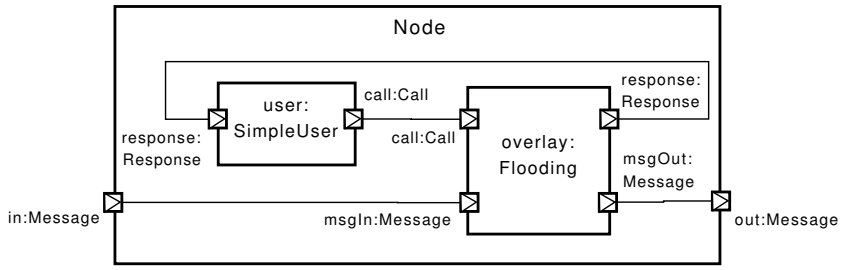

Abbildung 2.1: Gekoppeltes DEVS-Modell eines Knotens

In DEVS und seinen Varianten findet Komposition über gekoppelte Modelldefinitionen statt. Untermodelle können, so lange Ein- und Ausgabeports übereinstimmen, beliebig ausgetauscht werden und atomare Modelle durch gekoppelte Modelle verfeinert werden. Letzteres ist durch das Prinzip der Geschlossenheit unter Kopplung sichergestellt (Zeigler u. a. 2000, 151f.).

Modelle besitzen nicht automatisch Informationen darüber, mit welchen anderen Modellen sie verkoppelt werden. Ports von Modellen werden durch das übergeordnete Modell verbunden. Da Modelle nur über ihre Ports miteinander kommunizieren dürfen, lässt sich die Kompatibilität von Teilmodellen auf Basis der Wertebereiche ihrer Ports überprüfen.

Beispiel 2.2. *Abbildung 2.1 stellt einen Netzwerknoten für MANETs als Blockdiagramm dar. Das atomare DEVS-Modell Flooding aus Beispiel 2.1 wird darin als Untermodell verwendet und mit einem Nutzermodell verbunden. Das Nutzermodell wird als gegeben angenommen. Formal ist ein Knoten definiert als* $Node = (X, Y, D, \{M_{overlay}, M_{user}\}, EIC, EOC, IC)$*, mit*

- $X = \{(i, v) | i \in InPorts, v \in X_i\}$*, mit* $InPorts = \{$*„in“*$\}$ *und* $X_{in} = Message$
- $Y = \{(i, v) | i \in OutPorts, v \in Y_i\}$*, mit* $OutPorts = \{$*„out“*$\}$ *und* $Y_{out} = Message$
- $D = \{$*„user“*, *„overlay“*$\}$
- $M_{overlay} = Flooding$ *und* $M_{user} = SimpleUser$
- $EIC = \{($*„this“*, *„in“*, *„overlay“*, *„msgIn“* $)\}$
- $IC = \{($*„user“*, *„call“*, *„overlay“*, *„call“*$), ($*„overlay“*, *„response“*, *„user“*, *„response“*$)\}$
- $EOC = \{($*„overlay“*, *„msgOut“*, *„this“*, *„out“* $)\}$

Das Modell Node enthält Flooding *und* SimpleUser *als Untermodelle. Das Nutzermodell wird mit dem Dienstprotokoll über die Ports „call“ und „response“ verbunden. Eingehende Ereignisse vom Typ Message werden an „overlay“ delegiert. Von „overlay“ produzierte Ausgaben werden an die Umgebung weitergereicht.*

Es existiert bisher noch kein akzeptierter Standard zum Speichern von DEVS-Modellen. In existierenden DEVS-Tools[4] dominieren proprietäre Formate (MacSween u. Wainer 2004) oder die Definition von Modellen wird mittels Programmiersprachen realisiert (Zeigler u. Sarjoughian 2005; Chen u. Szymanski 2002). Neuere Ansätze (Janoušek u. a. 2006; Martín u. a. 2007) nutzen XML und streben nach einer Standardisierung. Den existierenden Ansätzen ist gemein, dass in den Wertebereichen der Portdefinitionen allein der syntaktische Aspekt auszutauschender Daten erfasst wird.

[4]Eine Übersicht findet sich auf `http://www.sce.carleton.ca/faculty/wainer/standard/tools.htm` (Zugriff am 17. September 2007)

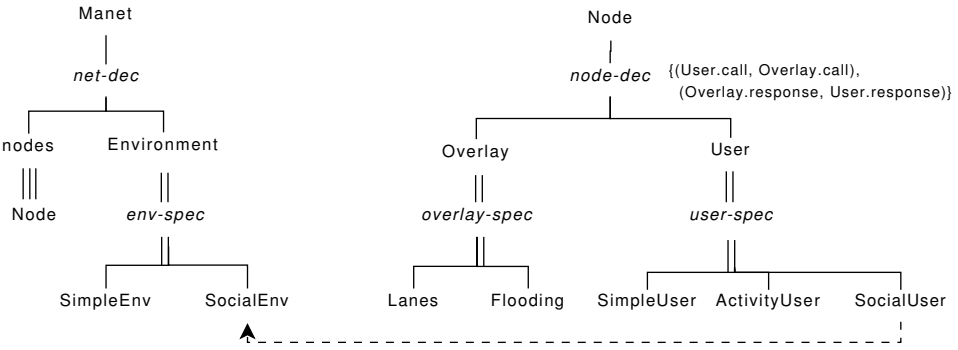

Abbildung 2.2: Dekompositionen und alternative Spezifikationen für das Modell Manet

Mit DEVS erstellte Modelle werden gewöhnlich in einem Simulationswerkzeug zur Ausführung gebracht, wo sie unter Kontrolle des Werkzeugs liegen. Beobachtungsinstrumente lassen sich vergleichsweise einfach und flexibel realisieren. DEVS unterscheidet strikt zischen Modell und Simulator (Zeigler 1984), so dass Modellebeschreibungen, soweit vorhanden, mit verschiedenen Simulationsalgorithmen ausgeführt (Himmelspach u. Uhrmacher 2004) sowie partitioniert (Ewald u. a. 2006) werden können.

Generell, unterscheidet DEVS nicht explizit zwischen öffentlichen und privaten Teilen bei Modelldefinitionen. Schnittstellen lassen sich nicht als eigenständig existierende Dokumente definieren, die separat publiziert werden könnten, sondern sind als Portdefinitionen Bestandteil von Modellbeschreibungen. In den existierenden XML-Repräsentationen definiert jedes Modell die Wertebereiche seiner Ports lokal (Martín u. a. 2007; Janoušek u. a. 2006). Modelle mittels Programmiersprachen zu definieren, ermöglicht demgegenüber auszutauschende Ereignisse in separaten Dokumenten zu definieren, bspw. als Klassen in Java. Werden diese Definitionseinheiten öffentlich zugänglich gemacht, können sie auch von Modellen anderer Modellentwickler genutzt werden.

Ein explizites Schnittstellenkonzept für DEVS schlägt Yilmaz (2004b) vor. Für Modelle lassen sich Rollen spezifizieren, die einen Teil des Modellverhaltens reflektieren. Rollen beschreiben das gewünschte Verhalten einer Komponente in Bezug auf einen Kommunikationspartner. Eine Komponente kann mehrere Rollen ausstellen, die jedoch unabhängig voneinander sein müssen. Eine Rolle basiert auf einer Teilmenge von Eingabe- und Ausgabeports und definiert das Verhalten, das bezüglich dieser Ports sichtbar ist. Praktisch genutzt wird dieser Ansatz bisher[5] nicht.

SES/MB

Einen Teil der Anforderungen an die Komponierbarkeit von Modellen wird nicht von DEVS selbst abgedeckt, sondern von Technologien, die mit DEVS assoziiert sind, wie die *System Entity Structure* (SES) und die *Model Base* (MB), zusammen als SES/MB bezeichnet (Zeigler u. Sarjoughian 2002). SES/MB ermöglicht, Modellfamilien und alternative Modelldefinitionen in Modellbanken zu organisieren, um bspw. die Modellierung von einem System auf unterschiedlichen Abstraktionsebenen, mit unterschiedlichen Kompositionsstrukturen und alternativen Spezialisierungen systematisch verwalten zu können. Simulationsmodelle können nach dem Ausschluss aller Designalternativen einer SES-Struktur, genannt Beschneidung, automatisch generiert werden. Die Beschneidung kann durch Selektionsbedingungen eingeschränkt werden und ein beschnittener SES-Baum kann durch Parameterbelegung direkt auf ein DEVS-Modell abgebildet werden. Für Modellinstanzen resultieren die Parameterwerte in initialen Zustandswerten.

Beispiel 2.3. *Abbildung 2.2 zeigt eine SES-Struktur für ein MANET-Modell. Das Modell Manet ist zusammengesetzt aus (symbolisiert durch einen senkrechten Strich, der mit dem Aspekt net-spec*

[5] Stand Dezember 2006

Kriterium	*Grad*	*Bemerkungen*
Spezifiz., Publiz. & Auffinden		
Austauschformat	∘	Definition in XML partiell möglich, jedoch nicht standardisiert
Separate Schnittstelle	-	Keine separaten Schnittstellendefinitionen, Ports sind Teil der Modelle.
Implementieren & Verifizieren		
Modellierung	+	Modellierungsformalismus mit strikter Trennung zwischen Modell und Simulator sowie formaler operationaler Semantik
Parametrisierung	∘	begrenzt realisierbar, durch Setzen initialer Zustandswerte atomarer Modelle
Verfeinerung	∘	Modular-hierarchischer Formalismus mit Geschlossenheit unter Kopplung
Komponieren & Analysieren		
Technische Ebene	∘	Vielzahl an Modelltransformationen mit DEVS als Zielformalismus möglich
Syntax	+	Ereignisports definieren die Syntax von auszutauschenden Datenelementen vollständig.
Semantik	-	nicht explizit erfasst
Pragmatik	-	nicht explizit erfasst
Dynamik	∘	theoretisch definierbar mit Yilmaz' dynamischen Rollenbeschreibungen
Konzepte	-	nicht explizit erfasst
Validität	∘	Verwaltung von Kompositionsstrukturen mit SES; theoretisch analysierbar mit Zeiglers Hierarchie der Systembeschreibungen (vgl. Abschnitt 2.3.1)
Experimentieren	+	breite Toolunterstützung teilweise mit alternativen Ausführungsalgorithmen, Partionierung und Instrumentierung

Tabelle 2.1: Abdeckung der Anforderungen durch DEVS

beschriftet ist) einer Menge von Knoten und einem Umgebungsmodell. Für das Umgebungsmodell besteht die Alternative (symbolisiert durch zwei senkrechte Striche und beschriftet mit env-spec), durch SimpleEnv oder SocialEnv realisiert zu werden. Drei senkrechte Striche symbolisieren eine spezielle Form der Dekomposition, so dass durch nodes eine Menge von Knoten repräsentiert werden. Ein Knoten ist wiederum strukturiert in User und Overlay. Das Kopplungsschema ist nur für die Dekomposition mit dem Namen `node-dec` *explizit angegeben. Die gestrichelte Linie repräsentiert eine Selektionsbedingung, so dass, falls der soziale Nutzer in einem Simulationsmodell genutzt wird, auch das soziale Umgebungsmodell zum Einsatz kommen muss.*

Die formale Grundierung und Ausdrucksstärke, kombiniert mit einer breiten Toolunterstützung, prädestinieren DEVS als Referenzformalismus für die *lokale*, modular-hierarchische Modellierung. SES erweitert die Fähigkeiten von DEVS, um die Möglichkeit, alternative Kompositionsstrukturen zu verwalten. Ebenso wie DEVS, spezifiziert SES jedoch sämtliche Abhängigkeiten zentralisiert und top-down. Die Komposition von Simulationsmodellen aus unabhängig voneinander entwickelten Komponenten wird im Umfeld von DEVS bisher nicht unterstützt. Tabelle 2.1 veranschaulicht[6], wie die aufgestellten Anforderungen durch DEVS abgedeckt werden[7].

[6] Die Abdeckungsgrade sind symbolisiert durch: keine: −, partiell: ∘ und hoch: + (hoch).

[7] Die Aufgliederung des Anforderungsbereichs „Komponieren & Analysieren" folgt der Diskussion zu Beginn des Abschnitts 2.2

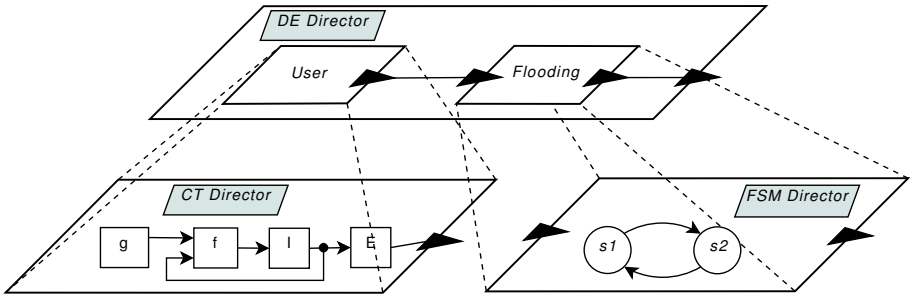

Abbildung 2.3: Ein Netzwerkknoten mit heterogenen Modellbeschreibungen

2.1.2 Heterogene modular-hierarchische Modellierung in Ptolemy

Der Wunsch, heterogene Beschreibungen von Teilmodellen zu integrieren, stellt sich unter anderem im Bereich der eingebetteten Systeme. Einzelne Teile können unterschiedliche Charakteristika aufweisen, die jeweils die Verwendung verschiedener Formalismen nahelegen. Ptolemy (Eker u. a. 2003) ist ein Projekt, das sich aus den Anforderungen zur Modellierung eingebetteter Systeme motiviert. Ptolemy ist jedoch als allgemeines Rahmenwerk zur Komposition von heterogenen Modellen konzipiert und im Softwareprodukt Ptolemy II inklusive graphischer Benutzungsschnittstelle implementiert (Brooks u. a. 2007). Anders als bei DEVS-Modellen wird die Heterogenität von Teilmodellen in Ptolemy nicht mittels eines vereinheitlichenden Formalismus' aufgelöst.

Ansatzpunkt für Ptolemy ist die Modularisierung von nebenläufigen Systembeschreibungen durch die Unterscheidung von Kommunikation und Berechnung. Anders als bei Schnittstellen für Software, bei denen Methodenaufrufe mit einem Transfer der Kontrolle einhergehen, sollen Schnittstellen für Modelle nur den Datenfluss in Modellen bekanntgeben (Lee u. Neuendorffer 2004a). Modelle werden in Ptolemy als nebenläufige, miteinander kommunizierende Systeme aufgefasst. Nach Lee und Neuendorffer (2004b) basiert das akteursorientierte Modellierungskonzept von Ptolemy II auf dem von Hewitt (1977) eingeführten und von Agha u.a. (1997) adaptierten Begriff des Akteurs, engl. *actor*.

Akteure stellen in Ptolemy wohldefinierte Schnittstellen aus, die von internem Verhalten abstrahieren. Ein Akteur definiert Kommunikationspunkte, genannt Ports, und Konfigurationspunkte, genannt Parameter. Ein Model besteht in Ptolemy II aus einer Menge von Akteuren, Ports, Parametern und Kopplungen. Mittels Kopplungen wird der Datenfluss zwischen Akteuren modelliert. Ähnlich zu DEVS lassen sich Modelle modular-hierarchisch konstruieren und dürfen nur über Ports miteinander kommunizieren. Mittels Ports und Kopplungen lässt sich bei der Modellierung syntaktische Kompatibilität sicherstellen (Lee u. Neuendorffer 2004b).

Anders als in DEVS wird das Port-basierte Schnittstellenkonzept nur als syntaktische Grundlage aller Modelle gesetzt. Die verschiedenen Modellierungsformalismen in Ptolemy teilen sich die syntaktischen Beschreibungsmittel von Ports und Akteuren, sind jedoch mit jeweils eigenen Berechnungsmodellen assoziiert. Berechnungsmodelle definieren für einen Formalismus, wann Teilmodelle ausgeführt werden und wann Kommunikation entlang von Kopplungen stattfindet. In Ptolemy II sind auf diese Weise mehr als zehn Modellierungsformalismen, bezeichnet als Domänen, integriert (Eker u. a. 2003).

Beispiel 2.4. *Eine vereinfachte Darstellung eines heterogenen Knotenmodells mit drei unterschiedlichen Modellierungsformalismen findet sich in Abbildung 2.3. Der Ereignisgenerator E bildet die kontinuierlichen Signale des Nutzermodells in diskrete Ereignisse ab, die zum Modell Flooding geleitet werden und dort Zustandveränderungen eines endlichen Zustandsautomatens hervorrufen. Node integriert die beiden Teilmodelle in der diskret-ereignisorientierten Modellierungsdomäne.*

Ptolemy nimmt eine Sonderrolle innerhalb der modular-hierarchischen Ansätze ein, da es die Kom-

Kriterium	*Grad*	*Bemerkungen*
Spezifiz., Publiz. & Auffinden		
Austauschformat	+	Vollständiges Austauschformat für Modelle in XML
Separate Schnittstelle	-	Keine separaten Schnittstellendefinitionen, Ports sind Teil der Modelle.
Implementieren & Verifizieren		
Modellierung	+	Modellierungsformalismen mit strikter Trennung zwischen Modell und Simulator sowie formaler operationaler Semantik
Parametrisierung	+	Explizites Parametrisierungskonzept; Parameter sind auch zur Laufzeit änderbar
Verfeinerung	∘	Modular-hierarchische Verfeinerung von Modellen möglich
Komponieren & Analysieren		
Technische Ebene	+	Modelldefinitionen unterschiedlicher Formalismen kombinierbar
Syntax	+	Ereignisports definieren die Syntax von auszutauschenden Datenelementen vollständig.
Semantik	-	nicht explizit erfasst
Pragmatik	-	nicht explizit erfasst
Dynamik	∘	theoretisch möglich durch dynamisches Typsystem; Wird aufgrund des hohen Berechnungsaufwands nicht genutzt.
Konzepte	-	nicht explizit erfasst
Validität	-	nicht explizit erfasst
Experimentieren	+	graphische Benutzungsschnittstelle inklusive Modelleditoren und Werkzeugen zur Auswertung von Simulationsläufen

Tabelle 2.2: Abdeckung der Anforderungen durch Ptolemy

patibilität von Berechnungsmodellen prüft, um heterogene Modelle zu integrieren. Die Grundidee ist dabei, das Verhältnis unterschiedlicher Berechnungsmodelle zu formalisieren. In Ptolemy II wurde ein Typsystem für dynamische Beschreibungen auf Basis von Interface-Automaten (de Alfaro u. Henzinger 2001) integriert (Lee u. Xiong 2004). Die Interface-Automaten dienen dazu, das Kommunikationsverhalten der unterschiedlichen Akteurstypen sowie Domänen zu spezifizieren, um über ihre Kompatibilität zu befinden. Zwei Automaten sind formal dann kompatibel, falls es eine Kommunikationsfolge zwischen ihnen gibt, die nicht zu illegalen Zuständen führt. Ein Zustand gilt in einer Komposition als illegal, falls ein Automat eine Ausgabe produziert, die von einem anderen Automaten nicht akzeptiert werden kann. Des Weiteren gelten alle Zustände, von denen der illegale Zustand durch interne Übergänge oder die Produktion von Ausgaben erreicht werden kann, auch als illegal. Kompatibilität von Interface-Automaten sagt aus, dass erfolgreiche Kommunikation zwischen Automaten möglich ist, jedoch nicht, dass Kommunikation in jedem Fall erfolgreich sein muss.

Die Kompatibilität von Akteursmodellen mittels Interface-Automaten zu prüfen, ist bereits für kleinere Modelle berechnungsintensiv. Aus diesem Grund wurde sich in Ptolemy für eine Kompatibilitätsprüfung entschieden, die nur prüft, ob die Kommunikationsprotokolle der genutzten Formalismen zueinander kompatibel sind (Lee u. Xiong 2004). Nur wenn ein Akteurstyp kompatibel zu einer Formalismusdomäne ist, kann der Akteur in dieser Domäne sinnvoll eingesetzt werden (Eker u. a. 2003). Es wird jedoch nicht geprüft, ob das implementierte Verhalten jedes einzelnen Modells kompatibel ist.

Auch mit Ptolemy bleibt jedoch ein grundlegendes Hindernis im Hinblick auf die Komponen-

tenorientierung ungelöst. Datentypen werden lokal definiert und Modelle als ganzes veröffentlicht (Brooks u. a. 2007). Eine zusammenfassende Bewertung von Ptolemy gibt Tabelle 2.2.

Modular-hierarchische Formalismen bieten damit nur eine begrenzte Unterstützung, um dezentral und unabhängig voneinander entwickelte Modelle zu integrieren. Dafür bedarf es der expliziten Differenzierung zwischen Schnittstellenbeschreibung und Modellimplementierung.

2.2 Kopplung von Simulationssystemen

Alternativ zum Austausch von Modellen zwischen Simulationssystemen, lassen sich Modelle auf der Ebene von Simulationssystemen integrieren. Modelle interagieren dann nicht in einer bestimmten Formalismusdomäne miteinander, sondern als Softwareeinheiten. Dieser Ansatz bietet den Vorteil, sehr heterogene Modelle integrieren zu können.

Die ingenieurstechnische Dimension von Komponierbarkeit manifestiert sich bei der Integration operationaler Modelle an dem Problem der Interoperabilität. Interoperabilität bezeichnet die Fähigkeit eines Systems, Dienste für andere Systeme nutzbar zu machen. Für die Kopplung von operationalen Modellen wird die Kompatibilität von Modellen über die Interoperabilität von Simulationssystemen hergestellt.

Für die Komponierbarkeit operationaler Modelle existiert ein Referenzmodell zur Unterscheidung unterschiedlicher Grade von Interoperabilität (Tolk u. Muguira 2003). Das Referenzmodell *Levels of Conceptual Interoparability Model* (LCIM) differenziert in seiner aktuellen Form (Turnitsa 2005; Tolk 2006) zwischen folgenden Ebenen der Interoperabilität:

0 *Keine:* Systeme sind nur einzeln nutzbar.

1 *Technische:* Es existiert ein Protokoll zur Interaktion zwischen den beteiligten Systemen.

2 *Syntaktische:* Die Systeme nutzen kompatible Formate zur Beschreibung aller Daten, die ausgetauscht werden.

3 *Semantische:* Die Bedeutung der auszutauschenden Daten ist jeweils definiert, z.B. durch Referenzierung von Standardformaten oder Ontologien.

4 *Pragmatische:* Die Kontexte der Datennutzung sind wechselseitig bekannt.

5 *Dynamische:* Die internen Zustandsänderungen der Systeme werden gegenseitig bekanntgegeben.

6 *Konzeptuelle:* Die Systeme stellen eine formale, implementierungsunabhängige Spezifizierung aller verwendeten Annahmen, Einschränkungen und Abstraktionen bereit.

LCIM liefert damit einen umfassenden Bewertungsrahmen für Grade an Interoperabilität, der sich in Bezug auf die Integration von Modellen auch für die Differenzierung von Kompatibilität nutzen lässt.

Im Kontext modular-hierarchischer Formalismen werden bisher die syntaktische und dynamische Dimension von Kompatibilität hervorgehoben (Lee u. Xiong 2004; Yilmaz 2004b). Die semantische und die konzeptuelle Ebene werden nur implizit über die Definition von Eingabe- und Ausgabemengen bzw. mittels Kommentaren in Modellfdefinitionen kodiert. Dort, wo Modelle unabhängig voneinander entwickelt und wiederverwendet werden sollen, stellt sich die Frage, ob Daten den gleichen Ausschnitt aus der Realität beschreiben und somit von allen beteiligten Modellen gleich interpretiert werden. In domänenspezifischen Austauschformaten, wie bspw. der *Systems Biology Markup Language* (SBML) für molekular-biologische Modelle (Hucka u. a. 2004), wird auch die semantische und konzeptuelle Dimension der Kompatibilität zunehmend berücksichtigt (Le Novère u. a. 2005).

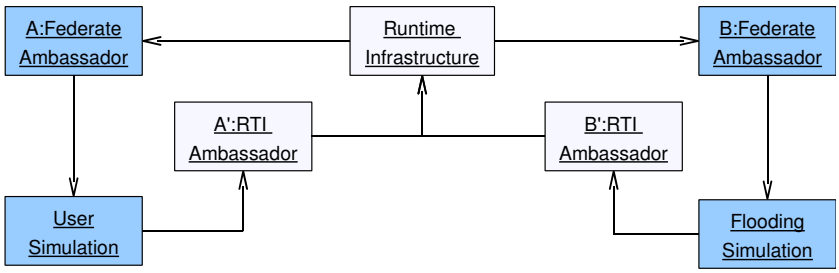

Abbildung 2.4: Kopplung der Simulationsmodelle `User` und `Flooding` über die HLA

Im Bereich der simulationsbasierten Komposition hat sich das Verständnis der unterschiedlichen Kompatibilitätsebenen sogar weiter ausdifferenziert. Aufsetzend auf der Semantik von Daten gilt es, deren pragmatische Relevanz zu berücksichtigen (Hofmann 2004). Die Pragmatik eines Modells drückt sich darin aus, welche Rolle Datenelemente für den Ablauf eines Modells spielen. Selbst Ereignissen, denen auf semantischer Ebene die gleichen Bedeutungen zugeordnet werden, können in Modellen zu unterschiedlichen Konsequenzen führen (Davis u. Tolk 2007).

Gänzlich zu unterscheiden von dem semantischen Aspekt der Kompatibilität ist die semantische Dimension der Komponierbarkeit. Semantische Kompatibilität bezieht sich darauf, ob auszutauschende Datenobjekte von den beteiligten Modellen mit der gleichen Bedeutung assoziiert werden. Der semantische Aspekt der Komponierbarkeit betrifft die Frage, ob ein komponiertes Modell in Hinblick auf eine Fragestellung valide ist.

Um Interoperabilität zwischen Simulationssystemen herzustellen, wird gewöhnlich schrittweise vorgegangen und zuerst an der technischen Ebene angesetzt.

2.2.1 HLA

Die *High Level Architecture* (HLA) ist von der IEEE (2000) standardisiert und darauf ausgelegt, Interoperabilität von Simulationsmodellen, auch über Grenzen von Organisationen hinweg, zu ermöglichen (Morse u. a. 2006).

Ein grundlegendes Konzept der HLA nennt sich *federate* und kann ein Simulationsmodell, ein Datenkollektor, eine Visualisierung oder auch eine Schnittstelle zu einem menschlichen Akteur sein. Ein oder mehrere *Federates* können zum Erreichen eines bestimmten Ziels zu einer *Federation* zusammengeschlossen werden. Alle *Federates* interagieren in einer HLA-Simulation über die *Run Time Infrastructure* (RTI). Die RTI stellt allgemeine Dienste zur Interaktion zwischen den *Federates* bereit und ermöglicht damit, die Implementierungen von Teilmodellen zu entkoppeln.

Beispiel 2.5. *Abbildung 2.4 zeigt die mögliche Kopplung einer Nutzersimulation mit einer Simulation zur Dienstvermittlung mittels HLA. Für beide Federates existiert ein Ambassador, der es der RTI erlaubt, mit dem jeweiligen Federate zu interagieren.*

Die RTI ist das Mittel, um technische Interoperabilität herzustellen. *Federates* können dabei auch auf unterschiedliche Orte verteilt sein. Diese Abstraktionsleistung geht jedoch mit einem Verlust an Ausführungseffizienz und Experimentkontrolle einher. So ist das Verteilungsmuster in der Regel vorgegeben und die Kommunikation über die RTI benötigt im Vergleich zu einem Funktionsaufruf in einer Programmiersprache ca. 100 mal mehr Zeit (Möller u. Dahlin 2006).

Für die syntaktische Interoperabilität bedarf es einer einheitlichen Definition von Datenstrukturen, die über die RTI ausgetauscht werden sollen. Die Daten werden für die HLA mittels eines Beschreibungsstandards namens *Object Model Template* (OMT)definiert. Mit der OMT werden unter anderem die Struktur für und die Zugriffsmodi auf Datenobjekte tabellarisch spezifiziert. Ein gemeinsames Verständnis von Daten kann über Lexika hergestellt werden.

Call (S)	ServiceOfferCall (S)
	ServiceSearchCall (S)
Response (P)	ServiceSearchResponse (P)
Message (PS)	ServiceSearch (PS)
	ServiceSearchResponse (PS)

Tabelle 2.3: Klassenstruktur für Datenobjekte von `Flooding`

Category	Information
Name	Flooding
Type	SOM
Version	1.0
Modification Date	2007-05-25
Purpose	Service Trading in MANETs
Application domain	Mobile Ad-Hoc Networks
POC	M. Röhl
POC Organization	University of Rostock
POC Telephone	+49 381 498 7614
POC Email	`mroehl@informatik.uni-rostock.de`
References	M. Röhl, B. König-Ries and A. M. Uhrmacher: An Experimental Frame for Evaluating Service Trading in Mobile ad-hoc Networks, in Mobilität und Mobile Informationssysteme (MMS 2007), volume 104, pp. 37–48, Lect. Notes Inform., 2007
Other	none

Tabelle 2.4: Identifikationstabelle für das Simulationsmodell `Flooding`

Beispiel 2.6. *Tabelle 2.3 zeigt den Teil einer OMT-Klassenstruktur für das Modell Flooding. Klassen werden darin Berechtigungen zugeordnet. S steht dabei für einen lesenden Zugriff, engl.* Subscribe *und P für einen schreibenden Zugriff, engl.* Publish*. Fluten kann auf Attribute von Call zugreifen. Attribute von Responses können publiziert werden. Der Zugriff auf ServiceSearch und ServiceSearchResult ist bidirektional möglich.*

Die OMT bietet eine Reihe weiterer Spezifikationsmöglichkeiten, um den Datenaustausch für Federates und innerhalb von Fedarations zu spezifizieren. So können mittels Interaktionstabellen für Klassen die Ereignisse, die ein Federate produzieren bzw. empfangen kann, definiert werden. Interaktionsklassen geben Aktionen vor, die von den Federates durchgeführt werden können.

Schnittstellen sind mit der OMT nicht als Teil von Modellenimplementierungen sondern in separaten Dokumenten definiert. Auf der Ebene von Federates spricht man von *Simulation Object Model* (SOM) und auf der Ebene von einer Federation von *Federation Object Model* (FOM). Mittels FOM und SOM werden vorrangig die syntaktische und semantische Ebene der Interoperabilität adressiert. Zusätzlich können eine Reihe von Metadaten in einer Identifikationstabelle spezifiziert werden. Für das Modell `Flooding` ist ein Beispiel in Tabelle 2.4 aufgeführt.

Die HLA bietet mittels expliziter Schnittstellenspezifikationen prinzipiell die Möglichkeit, Schnittstellen in deklarativer Form in öffentlichen Datenbanken zu hinterlegen, um unabhängig voneinander entwickelte Modelle, auf Grundlage abstrakter Beschreibungen zu komponieren. Die HLA wird bisher jedoch vorrangig im militärischen Bereich angewandt (Boer u. a. 2006a;b).

Die Möglichkeiten der HLA zur Komposition von Simulationsmodellen sind in Tabelle 2.5 zusammengefasst. Generell wird gefordert, dass die HLA mit softwaretechnischen Entwicklungen des freien Marktes schritthalten muss (Tolk 2002). So wird angestrebt, Fortschritte aus dem Bereich

Kriterium	*Grad*	*Bemerkungen*
Spezifiz., Publiz. & Auffinden		
Austauschformat	+	standardisiertes Austauschformat
Separate Schnittstelle	+	separate Dokumente
Implementieren & Verifizieren		
Modellierung	-	beschränkt auf die Beschreibung von Schnittstellen; keine Definition der Modellimplementierungen
Parametrisierung	○	indirekt über geteilte Objekte und Variablen möglich
Verfeinerung	-	kein formales Maß
Komponieren & Analysieren		
Technische Ebene	+	generelle Architektur zur Integration beliebiger Simulationssysteme
Syntax	+	Vollständige Definition auszutauschender Datenelemente.
Semantik	○	informal in Lexika
Pragmatik	○	informal in Metadaten beschreibbar
Dynamik	-	Zusammenhang zwischen Operationen nicht als Ablauf definierbar
Konzepte	○	informal in Metadaten
Validität	○	informal in Metadaten
Experimentieren	○	geringe Unterstützung durch kommerzielle Simulationswerkzeuge; eingeschränkte Kontrolle der Experimentdurchführung

Tabelle 2.5: Abdeckung der Anforderungen durch HLA

dienstorientierter Architekturen (Blais u. a. 2005; Pullen u. a. 2005; Möller u. Dahlin 2006) sowie der modelgetriebenen Entwicklung (Tolk 2004; Parr u. Keith-Magee 2004) der Modellierung und Simulation zuzuführen[8].

Die HLA wird entsprechend weiterentwickelt und erlaubt als HLA Evolved, FOMs zu modularisieren, so dass diese leichter wiederverwendet und flexibel in Federations geladen werden können (Möller u. Löfstrand 2007; Möller u. a. 2007a). Mit Blick auf die Komposition ermöglicht ein zu der HLA kompatibler Ansatz, die Ebenen der Interoperabilität weitreichender und modellbasiert abzudecken.

2.2.2 BOM

Das *Base Object Model* (BOM) ist ein standardisiertes Rahmenwerk (SISO 2006a), das auf die Wiederverwendung und die Komponierbarkeit von Simulationsmodellen ausgerichtet ist. BOMs repräsentieren wiederverwendbare Interaktionsmuster von Simulationsmodellen in einer XML-basierten Repräsentation (Gustavson u. Chase 2004).

BOM erweitert die Metadatenbeschreibung von HLA. Für ein Modell können Zweck, bisherige Nutzungen und bekannte Einschränkungen dokumentiert werden. In den Identifikationsdaten von BOMs können Schlagwörtern entsprechend einer Taxonomie angegeben und Ikonen für die Nutzung in VIMS bereitgestellt werden.

Mittels BOM lassen sich die in einem Simulationsmodell verwendeten Konzepte sowie die konzeptuellen Zusammenhänge zwischen Modellen beschreiben. BOM-Spezifikationen definieren den dynamischen Zusammenhang zwischen konzeptuellen Entitäten über Interaktionsmuster, engl. *pattern of interplay*. Ein Interaktionsmuster beschreibt eine Abfolge von Ereignissen, die zwischen

[8] Für eine ausführliche Diskussion siehe Abschnitt 3.2 bzw. 3.3

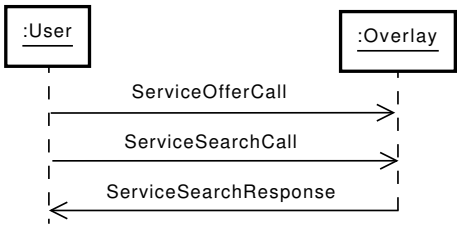

Abbildung 2.5: Muster der Zusammenarbeit mit BOM

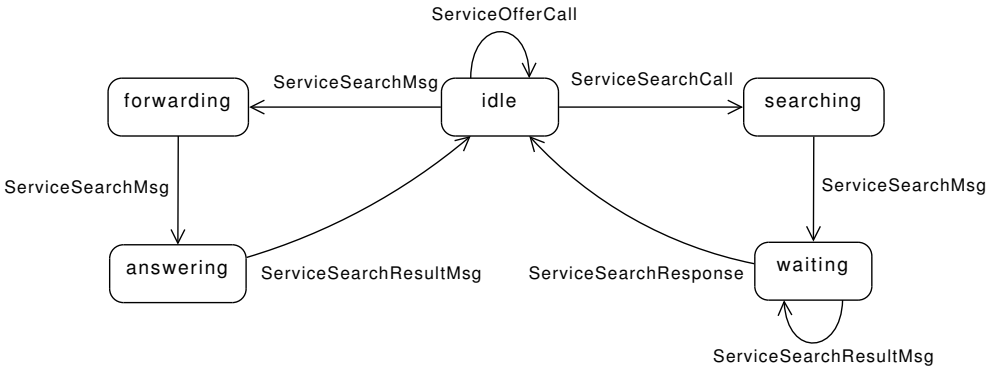

Abbildung 2.6: BOM-Zustandsmaschine für das Modell `Flooding`

konzeptuellen Entitäten zum Erreichen einer bestimmten Zielstellung ausgetauscht werden.

Das Gegenstück zu den Interaktionsmustern bilden Zustandsmaschinen, die wichtige interne Zustände und Bedingungen von Modellen reflektieren. Die Zustandsmaschinen definieren den Einfluss von Interaktionen auf die Modelle, die an Interaktionsmustern beteiligt sind. Zustandsübergänge in den Automaten können dabei mit mehreren Interaktionsmustern verknüpft werden.

Zur Visualisierung des Zustandsverhaltens und von Interaktionsmustern eignen sich UML Zustandsmaschinen bzw. Sequenzdiagramme (SISO 2006b). Vollständig definieren lassen sich BOMs aber nur mittels XML oder in Tabellen.

Beispiel 2.7. *Ein einfaches Muster der Zusammenarbeit zwischen einem Nutzermodell und einem Modell für die Dienstsuche veranschaulicht Abbildung 2.5. Von einem Dienstmodell wird erwartet, dass es erst Nachrichten vom Typ ServiceOfferCall empfängt und dann auf ServiceSearchCall mit ServiceSearchResponse antwortet. Abbildung 2.6 visualisiert einen Zustandsautomaten für das Modell Flooding.*

Die Beschreibung von Schnittstellen ist mit BOM theoretisch unabhängig von der HLA. Praktisch ist die Terminologie von BOM an die HLA angelehnt, die Abbildung konzeptueller Modelle auf die OMT zugeschnitten und die Benutzung im Rahmen empfohlener HLA-Praktiken (IEEE 2003) eingeordnet. Für die Entwicklung von HLA-Simulationen lassen sich BOMs als eine Repräsentation für konzeptuelle Modelle nutzen (Möller u. a. 2007b). Die Modellierung kann mittels eines graphischen Werkzeugs durchgeführt werden (SimVentions 2008). Tabelle 2.6 stellt die Möglichkeiten von BOM zusammenfassend dar.

Insgesamt bietet BOM die umfangreichste Beschreibung von Schnittstellen unter den praktischen Kompositionsansätzen. In der BOM-Spezifikation (SISO 2006a) ist jedoch nicht formal definiert, wie die Kompatibilität von Interaktionsmustern zu prüfen ist. Anstehende Entwicklungen zielen entsprechend auf die Automatisierbarkeit von Kompositionsprozessen mittels BOM (Moradi u. a. 2006).

Kriterium	*Grad*	*Bemerkungen*
Spezifiz., Publiz. & Auffinden		
Austauschformat	+	standardisiertes Austauschformat in XML
Separate Schnittstelle	+	separate Dokumente
Implementieren & Verifizieren		
Modellierung	○	Definition konzeptueller Modelle möglich
Parametrisierung	○	indirekt über geteilte Objektattribute und Parameter von Interaktionsklassen möglich
Verfeinerung	-	keine Verfeinerungsrelation
Komponieren & Analysieren		
Technische Ebene	+	„profitiert“ von den Möglichkeiten der HLA durch Abbildung auf OMT
Syntax	+	Vollständige Definition auszutauschender Daten
Semantik	○	informale Definitionen in Lexika
Pragmatik	○	informal in Metadaten beschreibbar
Dynamik	○	Zustandsmaschinen und Muster der Zusammenarbeit; keine formale Kompatibilitätsprüfung
Konzepte	○	informal in Metadaten
Validität	○	informal in Metadaten
Experimentieren	○	Graphisches Werkzeug zur Definition von BOMs; eingeschränkte Kontrolle der Experimentdurchführung

Tabelle 2.6: Abdeckung der Anforderungen durch BOM

2.3 Validität von Kompositionen formal analysieren

Die bisher vorgestellten Ansätze zur Komposition berücksichtigen den Aspekt der Validität von Modellkompositionen nicht oder nur informell über natürlichsprachliche Metadaten. Validieren erfordert, eine Komposition auf einen konkreten Einsatzkontext zu beziehen. Während Validieren sich mit der Frage beschäftigt, ob ein Modell bezüglich einer Fragestellung korrekt ist, prüft Verifizieren, ob ein Modell, bspw. ein operationales Modell, hinsichtlich einer Spezifikation, bspw. einem konzeptuellen Modell, richtig erstellt wurde.

Der Kontext des Einsatzes beschreibt die Bedingungen der Experimentdurchführung und wird in der Modellierung und Simulation auch experimenteller Rahmen, engl. *experimental frame*, genannt (Zeigler 1984; Yilmaz 2004a).

Ein formaler Ansatz zum Validieren ist, Modelle zu einem Referenzsystem in Beziehung zu setzen. Das Referenzsystem kann ein reales oder imaginiertes Modell sein, das für eine bestimmte Fragestellung als perfekt gilt. Validität lässt sich so als Grad der Überdeckung zwischen gewolltem und erstelltem Modell formalisieren. Ein Modell ist valide, wenn es in einem konkreten experimentellen Rahmen das gleiche Verhalten produziert wie das Referenzsystem unter den gleichen Bedingungen. Diese Methode des Validierens verfolgen sowohl Zeigler (1984) als auch Petty u.a. (2005). Indem beide Ansätze sowohl die Validität als auch die Komposition von Modellen formalisieren, lässt sich mit ihnen über die Validität von Kompositionen befinden. Anders als modular-hierarchische Modellierungsformalismen und Schnittstellenbeschreibungen aus dem Kontext der simulationsbasierten Ansätze fokussieren Zeigler und Petty nicht auf die syntaktischen Modellierungssmittel *eines* konkreten Formalismus sondern auf Beschreibungsstrukturen, die als semantische Domäne für Modelle allgmein fungieren können. Bei Zeigler sind das zeitbasierte Systembeschreibungen und im Falle von Petty berechenbare Funktionen.

2.3.1 Zeiglers Hierarchie von Systemspezifikationen

Zur Spezifikation von Systemen auf unterschiedlichen Abstraktionsebenen führt Zeigler (1976) ein systemtheoretisches Gerüst, genannt *Hierarchy of System Specifications*, ein. Die Hierarchie von Systembeschreibungen unterscheidet unterschiedliche Ebenen, auf denen Wissen über Systeme existieren kann — von abstrakten Verhaltenbeschreibungen bis zu stark strukturierten Systemdefinitionen. Die Ebenen sind nicht primär als syntaktische Mittel zur Spezifikation dieser Systeme gedacht, sondern können als semantische Domäne fungieren, auf die sich konkrete Modellierungsformalismen abbilden lassen. Modelle, Simulatoren und auch Referenzsysteme lassen sich auf diese Weise gleichermaßen mit einer formalen Semantik ausstatten. Es können sowohl formale Beziehungen zwischen Beschreibungen auf gleicher Ebene als auch zwischen Beschreibungen auf unterschiedlichen Ebenen hergestellt werden.

Die Ebenen

Grundlage der Beschreibungsmittel für Systeme bildet eine Zeitbasis $\langle T, < \rangle$, wobei T die Menge möglicher Zeitwerte darstellt und $<$ eine Ordnungsrelation über Elemente aus T ist. Auf Ebene 0 der Hierarchie wird ein System als reine *black box* beschrieben, die eine Zeitbasis mit einer Menge möglicher Eingaben X und einer Menge von Ausgabewerten Y verknüpft.

Für die Simulation interessiert das Verhalten eines Systems über die Zeit. Mittels Zeitfunktionen $f : T \to A$, auch Signale oder Trajektorien genannt, kann der Verlauf von Variablen über die Zeit beschrieben werden. Variablen aus A können dabei Eingaben, Ausgaben oder Zustände repräsentieren. Zeitfunktionen, die f auf eine Teilmenge von T beschränken, heißen Segmente. Die Menge aller Segmente über A und T wird mit (A, T) symbolisiert.

Die Beschreibungsform auf Ebene 1 erweitert die Beschreibung von Ebene 0 um eine Menge von erlaubten Eingabesegmenten $\Omega \subseteq (X, T)$ und eine Eingabe-Ausgabe-Relation $R \subseteq \Omega \times (Y, T)$. Ein Element $(\omega, \rho) \in R$ wird als Eingabe-Ausgabe-Paar bezeichnet und formalisiert die Vorstellung, dass, als Resultat auf die Eingabe eines Segments ω, das System ein Segment ρ produziert. R beinhaltet alle möglichen Paare (ω, ρ), die durch Experimentieren zu beobachten sind, ohne dass Annahmen über den Zustand von S getroffen werden. Differentialgleichungen sind ein konkreter Formalismus, um Wissen über Systeme auf Ebene 1 zu spezifizieren, wobei $T = \mathbb{R}^{\geq 0}$ und sowohl X als auch Y als Kreuzprodukte über $\mathbb{R}^{\geq 0}$ darstellbar sind.

Je nach innerem Zustand kann ein System unterschiedlich auf Eingaben reagieren. Verfügt man über Wissen über den initialen Zustand des Systems, kann die globale Relation R auf Ebene 2 in eine Menge von Funktionen $F = \{f_1, \ldots : f \subseteq \Omega \times (Y, T)\}$ zerlegt werden, so dass für jedes Eingabesegment ω eindeutig ein Ausgabesegment $\rho = f_i(\omega)$ identifizierbar ist. Ein Beispiel für Modelle auf Ebene 2 sind Differentialgleichungen mit Anfangsbedingungen.

Bezieht man das Wissen über die Zwischenzustände des Systems, und nicht nur über den Anfangszustand, in die Systembeschreibung mit ein, gelangt man auf Ebene 3 zu der Beschreibungsform *I/O System*. Die Funktionsmenge F kann dann durch eine Kombination aus Zustandsüberführungsfunktion $\Delta : S \times \Omega \to S$ und Ausgabefunktion $\Lambda : S \times X \to Y$ ersetzt werden, die in Abhängigkeit von einem initialen Zustand, den Endzustand sowie die Ausgaben des Systems eindeutig definieren.

Ebene 3 stellt unter anderem die Möglichkeiten bereit, die Semantik von DEVS-Modellen zu definieren. Das, durch ein DEVS-Modell $\mathscr{M}$, spezifizierte Verhalten wird mittels einer Systemstruktur $\mathscr{S} = \langle T, X, \Omega, Y, S, \Delta, \Lambda \rangle$ entfaltet, dessen Zustandsmenge $S = Q_{\mathscr{M}}$ durch die Menge der totalen Zustände von $\mathscr{M}$ gegeben ist. Die Ein- und Ausgabedefinitionen finden in $\mathscr{S}$ nicht mehr nur auf Basis der Wertebereiche statt, sondern assoziieren Werte mit der Zeit, so dass $\Omega \subseteq (X, T)$ die Menge der erlaubten Eingabesegmente definiert. Die Überführungsfunktion $\Delta : Q \times \Omega \to Q$ lässt sich für jedes Tupel, bestehend aus $q \in Q$ und einem möglichen Eingabesegment $\omega \in \Omega$, aus δ_{int}, δ_{ext}, und δ_{con} konstruieren. Λ ist die Ausgabefunktion und produziert Ausgabesegmente entsprechend der Modellausgabefunktion λ.

Auf der nächsten Ebenen der Hierarchie werden die globalen Funktionen Δ und Λ durch iterative

Ebene	*Name*	*Beschreibt*
0	I/O Observation Frame (IO)	Statische Schnittstelle eines Systems
1	I/O Relation Observation (IORO)	Menge möglichen Systemverhaltens für eine Menge von Stimuli: Das System als Datenquelle.
2	I/O Function Observation (IOFO)	Eindeutiges Systemverhalten für eine Eingabe
3	I/O System (IOS)	Globale Zustandsveränderungen und Systemausgaben für Eingaben
4	Iterative Specification	Lokale Zustandsveränderungen
5	Structured System specification	Strukturierung von Zuständen und Funktionen
6	Nonmodular coupled multicomponent system	System als Menge interagierender Komponenten mit eigenen Zuständen und Funktionen.
7	Modular coupled network of systems	Komponenten sind selbst Systeme mit Eingabe- und Ausgabeschnittstelle

Tabelle 2.7: Zeiglers Hierarchie von Systembeschreibungen, nach (Zeigler u. a. 2000, 132)

Funktionsdefinitionen δ bzw. λ ersetzt. Über die Strukturierung von Zuständen und Überführungsfunktionen eines Systems auf Ebene 5, gelangt man auf Ebene 6 zu zusammengesetzten Systemen in nichtmodularer Form und auf Ebene 7 schließlich zu vollständig modularen Systembeschreibungen. Einen Überblick der Systembeschreibungen gibt Tabelle 2.7.

Beziehungen zwischen den Ebenen

Sowohl horizontale als auch vertikale Beziehungen sind in der Hierarchie der Systembeschreibungen gut formalisiert. Mit dem Wissen über ein System auf Ebene i lässt sich eine Beschreibungsform auf Ebene $i-1$ generieren. Mit Ansteigen der Ebene wechselt man von Verhaltensbeschreibungen zu Strukturdefinitionen. Von abstrakten zu konkreteren Beschreibungsformen zu gelangen, ist jedoch nicht trivial und erfordert zusätzliche Informationen (Zeigler 1976).

Für horizontale Beziehungen zwischen Systembeschreibungen auf gleicher Ebene werden sogenannte Morphismen definiert. Ein Morphismus ist eine eigenschaftserhaltende Abbildung zwischen Systembeschreibungen (Zeigler 1976, 261). Zwei Systembeschreibungen auf Ebene 1 können bspw. als gleich betrachtet werden, wenn die Mengen ihres jeweils beobachtbaren Verhaltens zueinander isomorph sind. Morphismen auf Ebene 2 setzen verhaltensäquivalente Systeme zueinander in Beziehung (Zeigler 1984, 48), die auf gleiche Eingaben mit gleichen Ausgaben reagieren. Auf Ebene 3 werden darüber hinaus die internen Zustände zweier Systeme zueinander in Beziehung gesetzt.

Morphismen lassen sich zum Verifizieren operationaler Modelle nutzen. Ein Simulator zur Ausführung eines bestimmten Modells macht, zusätzlich zu den Schritten eines Modells, interne Verwaltungsschritte. Ein Simulator führt ein Modell korrekt aus, wenn das durch den Simulator produzierte Verhalten gleich dem durch das Modell spezifizierte Verhalten ist (Zeigler 1984).

Mittels Morphismen lässt sich auch über die Validität eines Modells befinden (Zeigler 1984). Ein Modell kann dann als valide aufgefasst werden, wenn für den Ausschnitt aller möglichen Trajektorien, der durch den experimentellen Rahmen vorgegeben ist, das Modell äquivalent zum Referenzsystem ist.

Entsprechend der Ebenen eins bis drei unterscheidet Zeigler replikative, voraussagende und strukturelle Validität (Zeigler 1984). Können alle Beobachtungen, die mit dem Referenzsystem möglich sind, auch mit dem modellierten System getätigt werden, gilt das modellierte System als replikativ valide. Voraussagende Validität formalisiert die Vorstellung, dass das modellierte System nicht nur die gleiche Menge an Beobachtungen erlaubt, sondern auch für gleiche Eingaben gleiche Ausgaben produziert. Strukturelle Validität erfordert zusätzlich einen Morphismus auf Ebene 3 vom Referenzsystem zum modellierten System, der angibt, dass beide Systeme über eine ähnliche interne

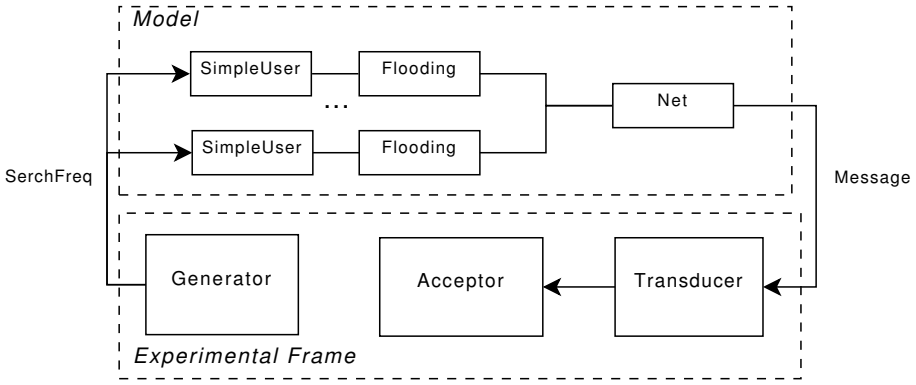

Abbildung 2.7: Experimenteller Rahmen

Abarbeitung verfügen.

Beispiel 2.8. *Die Kopplung eines experimentellen Rahmens mit einem Modell Manet zeigt Abbildung 2.7. Ein Generator beeinflusst die Suchfrequenz im Nutzermodell durch die Induktion von SearchFreq. Suchanfragen produzieren im Protokoll zur Dienstvermittlung Netzwerknachrichten. Diese werden als Ausgaben der Simulation beobachtet und dem experimentellem Rahmen zugeführt, der daraus bspw. die durchschnittliche Antwortzeit für Suchantworten ermitteln kann. Ein Modell ist in diesem experimentellem Rahmen replikativ valide, wenn jedem möglichem Eingabesegment des Referenzsystems ein Eingabesegment des Modells zugeordnet werden kann, jedes Ausgabesegment des Referenzsystems auf ein Ausgabesegment des Modells abgebildet werden kann und für jedes Paar, das im Referenzsystem beobachtet werden kann, die Abbildung des Paares im Modell beobachtbar ist.*

Da es nahezu unmöglich ist, Modelle zu erstellen, die vollständig gleiches Verhalten wie Referenzsysteme produzieren, führt Zeigler approximierende Morphismen ein, die ein bestimmtes Maß an Abweichung tolerieren. Welche Form der Validität und welches Fehlermaß angemessen ist, hängt von den Zielen einer Simulationsstudie ab (Zeigler u. Sarjoughian 2002). Allerdings stellt das Finden eines Referenzsystems mit Ansteigen der Ebene eine zunehmend unlösbare Herausforderung dar.

Für die Validität eines Modells ist es unerheblich, ob es komponiert wurde, oder ob es ein monolithisches Modell ist. Entscheidend ist, ob die Systembeschreibung eines Modells zu einem Referenzsystem auf der gewünschten Ebene in Beziehung gesetzt werden kann. Durch die formalisierte Beziehung zwischen den Ebenen lässt sich eine modulare Systemspezifikation stets eindeutig auf eine Beschreibung der ersten, zweiten und dritten Ebene abbilden, für die Validität formal definiert ist.

Die Nutzung von potentiell kontinuierlichen Zeitbasen erschwert jedoch die Analyse von Systembeschreibungen. So fehlt es an Werkzeugen, mittels denen sich Systembeschreibungen automatisiert vergleichen lassen. Tabelle 2.8 fasst die Möglichkeiten von Zeiglers Ansatz zusammen.

2.3.2 Theorie der Semantischen Komponierbarkeit

Es existiert ein alternativer Ansatz (Petty u. a. 2005), um die Validität von Simulationsmodellen formal zu analysieren. Im Unterschied zu Zeiglers Systemverständnis wird jedoch nicht auf einer kontinuierlichen sondern einer diskreten Zeitbasis analysiert. Modelle werden in dem Ansatz, im Einklang mit einem grundlegenden Konzept der theoretischen Informatik, als berechenbare Funktionen verstanden.

Kriterium	*Grad*	*Bemerkungen*
Spezifiz., Publiz. & Auffinden		
Austauschformat	-	kein Austauschformat
Separate Schnittstelle	∘	Eine Beschreibung auf unterer Ebene lässt sich als separate Schnittstelle für eine Systembeschreibung auf höherer Ebene verwenden.
Implementieren & Verifizieren		
Modellierung	∘	Die Ebenen definieren semantische Domänen für Modellierungsformalismen — nicht die Modellierungskonstrukte selbst.
Parametrisierung	-	keine Parameter
Verfeinerung	∘	Vollständig formalisierte Beziehungen zwischen Systembeschreibungen (sowohl auf gleichen als auch auf unterschiedlichen Ebenen)
Komponieren & Analysieren		
Technische Ebene	-	von der technischen Ebene wird vollständig abstrahiert
Syntax	+	Als Mengen definierbar
Semantik	-	nicht explizit definierbar
Pragmatik	-	nicht explizit definierbar
Dynamik	+	vollständig formale Beschreibung der Dynamik auf Ebene 3 der Hierarchie
Konzepte	-	nicht explizit definierbar
Validität	+	vollständig formalisiert
Experimentieren	-	Von der Experimentdurchführung wird vollständig abstrahiert.

Tabelle 2.8: Abdeckung der Anforderungen durch Zeiglers Hierarchie der Systembeschreibungen

Formal ist ein Modell M als berechenbare Funktion $f : S \times I \to S \times O$ definiert. S stellt dabei eine nichtleere Menge von Zuständen, I eine Eingabemenge und O eine Ausgabemenge dar (Petty u. a. 2005). Um die Darstellung zu vereinfachen, sind S, I und O als Mengen über endliche Vektoren von ganzen Zahlen definiert. Komplexere Zustände und Ereignisse können für die formale Analyse auf diese grundlegenden Mengen abgebildet werden. Eine Simulation ist als Sequenz von Berechnungsschritten eines Modells definiert, wobei die Zustandswerte eines Berechnungsschrittes dem Modell im nächsten Schritt als Berechnungsgrundlage dienen (Petty u. a. 2005).

Als Formalismus zur Beschreibung von Berechnungszusammenhängen finden beschriftete Transitionssysteme, engl. *Labelled Transition System* (LTS), Verwendung. Ein LTS ist ein Tupel $T = \langle S, \Sigma, \to \rangle$. S bezeichnet dabei die Menge von Zuständen, Σ die Menge von Beschriftungen, und $\to \subseteq S \times \Sigma \times S$ definiert eine Überführungsrelation. LTS verallgemeinern endliche Automaten, indem sie nicht notwendigerweise Anfangs- und Endzustände haben und S, Σ und $\to$ nicht notwendigerweise endlich sind. Hat ein Transitionssystem genau einen Anfangszustand $s_0 \in S$, definiert man das LTS als Viertupel $T = \langle S, s_0, \Sigma, \to \rangle$.

Für ein Modell M bezeichnet $M_S : S \times I \to S$ das Zustandsmodell, falls gilt, dass $\forall x \in S \times I . M_s(x) = s \Leftrightarrow M(x) = (s, o)$. Gegeben ein Zustandsmodell M_s, erfasst das deterministische LTS $L(M) = \langle S, I, M_S \rangle$ die Berechnungsschritte für M.

Beispiel 2.9. *Im oberen Teil der Abbildung 2.8 ist ein Ausschnitt der Berechnungsschritte für ein vereinfachtes Modell* `Flooding`*, genannt F, dargestellt. Für die Eingaben i_0, i_1, i_2, i_3 und die Ausgaben o_0, o_1, o_2, o_3 sind die repräsentierten Ereignisse mit aufgeführt. Im ersten Schritt berechnet F, für den Anfangszustand s_0 und die Eingabe i_0, die Ausgabe o_0 sowie den Folgezustand s_0, so dass $F(s_0, i_0) = (s_0, o_0)$. Der untere Teil der Abbildung enthält den Überführungsgraph für F.*

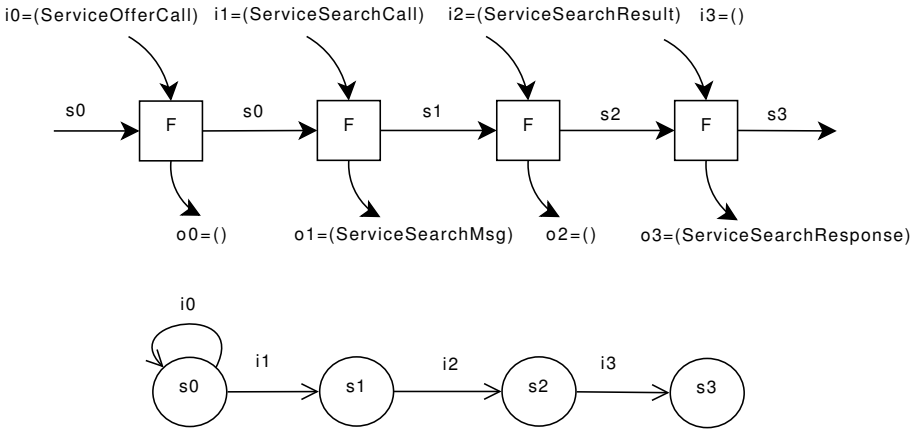

Abbildung 2.8: Berechnungsschritte für vereinfachtes Flooding-Modell mit assoziiertem LTS

Zwei Modelle können als gleich betrachtet werden, wenn sie ähnliche Berechnungsschritte unter gleichen Eingaben liefern. In Übereinstimmung mit Zeiglers Ansatz wird Validität in Bezug auf ein Referenzmodell definiert. Die Validität eines Modells M wird mittels einer Validitätsrelation V ausgedrückt, die M zu einer Annäherung M' eines perfekten Modells M^* in Beziehung setzt. M ist valide unter der Validitätsrelation V, falls ein V existiert, das $L(M)$ mit $L(M')$ in Beziehung setzt (Weisel u. a. 2003).

Für LTS existieren eine Reihe von Relationen, über die Systeme zueinander in Beziehung gesetzt werden können (van Glabbeek 2005). Bisimulation gilt als die am besten entscheidbare (berechenbare) Äquivalenzrelationen über LTS (Moller u. a. 2004). Zudem ist sie die am strengsten unterscheidende Verhaltensrelation (van Glabbeek 2005). Bisimulation formalisiert die Vorstellung, diejenigen Systeme als gleich zu betrachten, die in jeglichem Experiment, das auf Beobachtung beruht, ununterscheidbar sind (Milner 1990).

Bisimulation setzt diejenigen Zustände von zwei LTS miteinander in Beziehung, die mit jeweils gleichen Aktionen beschriftet sind und darüber hinaus zu Nachfolgezuständen führen, die wieder über Bisimulation in Beziehung stehen.

Es ist häufig sinnvoll, zwischen internen Aktionen und sichtbaren Berechnungsschritten eines Systems zu unterscheiden. Interne Aktionen sind für Beobachter unsichtbar und werden in LTS mit τ symbolisiert. Mit der schwachen Bisimulation lassen sich, im Unterschied zur starken Bisimulation, Systeme auch dann als äquivalent betrachten, wenn sie zusätzliche interne Aktionen ausführen. Schwache Bisimulation erlaubt bspw., eine Schnittstelle für eine Implementierung zu konstruieren, die von internen Details abstrahiert, jedoch als verhaltensäquivalent zur Implementierung betrachtet werden kann. Die Unterscheidung von beobachtbaren und nicht beobachtbaren Aktionen ist im Kontext von LTS das Mittel, Systeme auf unterschiedlichen Abstraktionsebenen zu betrachten. Im Unterschied zu Zeiglers Hierarchie von Systembeschreibungen bieten LTS damit ein universelles Spezifikationsmittel zur Beschreibung von Systemen auf unterschiedlichen Abstraktionsebenen. Im Rahmen der Theorie der semantischen Komponierbarkeit findet ausschließlich die schwache Bisimulation Verwendung.

Beispiel 2.10. *Im vorigen Beispiel inklusive Abbildung 2.8 stellt i_3 für das LTS F eine leere Eingabe dar. Für einen Beobachter ist die Änderung von s_2 zu s_3 nicht wahrnehmbar, da sie unabhängig von einer Interaktion erfolgt. Die Ausgabe o_3 ist erst beim Verlassen von s_3 zu beobachten. In Abbildung 2.9 ist die interne Zustandsänderung i_3 entsprechend durch τ ersetzt. Für i_0, i_1, i_2 sind die durch sie repräsentierten Aktionen dargestellt.*

Die Zustände s_0 bis s_3 können nun mit den Zuständen t_0, t_1, t_2 eines anderen LTS in Relation

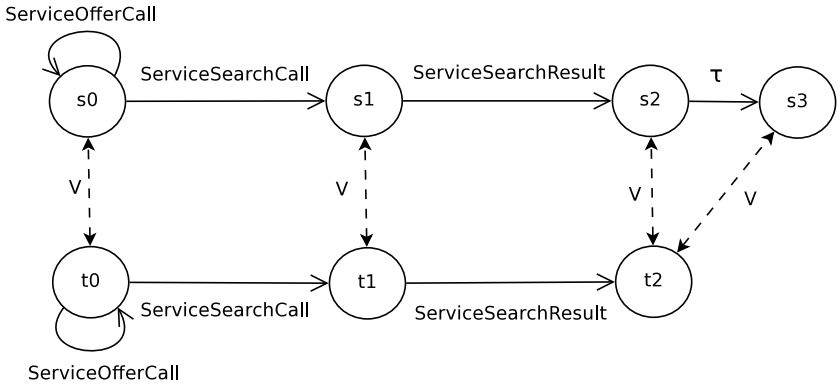

Abbildung 2.9: Beispiel für schwache Bisimulation zwischen zwei LTS

gesetzt werden, so dass diese Relation, symbolisiert durch V, eine schwache Bisimulation ist.

Mit der Repräsentation von Modellen durch Funktionen kann die Komposition von Modellen auf Basis der Verknüpfung von Funktionen definiert werden. Gegeben die Funktionen $F : X' \rightarrow Y'$ und $G : X \rightarrow Y$, wobei $X' \subseteq S' \times I'$, $Y' \subseteq S' \times O'$, $X \subseteq S \times I$ und $Y \subseteq S \times O$, existiert die syntaktische Komposition $F \circ G : X'' \rightarrow Y''$ genau dann, wenn $S = S'$, wobei $X'' \subseteq S \times (I \times I')$ und $Y'' \subseteq S \times (O \times O')$. Syntaktische Komponierbarkeit stellt sicher, dass zwei Modelle zueinander kompatibel sind. Die Kompatibilität von zwei Modellen gibt noch keine Auskunft über die Validität des Gesamtmodells. Aufbauend auf der syntaktischen Komposition sind zwei Modelle F und G semantisch komponierbar, falls $F \circ G$ existiert und für eine Validitätsrelation V sowie für ein als perfekt geltendes Modell $F^* \circ G^*$ zwei Modelle F', G' existieren, so dass $F' \circ G' \subseteq F^* \circ G^*$ und $L(F \circ G) \Leftrightarrow_V L(F' \circ G')$.

Die praktische Anwendung der Theorie der semantischen Komponierbarkeit steht vor zwei wesentlichen Herausforderungen. Zum einen sind perfekte Modelle in der Realität schwer zu finden und zum anderen stellt Bisimulation für konkrete Anwendungen hohe Anforderungen an die Ähnlichkeit von Beschreibungen. Über Validität mittels Bisimulation zu befinden, entspricht Zeiglers höchster Form der Validität: der strukturellen Validität. Zur Abmilderung der Diskriminierungsstärke von Bisimulationen bietet die Theorie der semantischen Komponierbarkeit parameterisierte Bisimulationen (Petty u. Weisel 2003b). Metriken erlauben bspw. ein wohldefiniertes Maß an Abweichung für einzelne Berechnungsschritte oder auch ganze Trajektorien (Petty u. Weisel 2003b). Tabelle 2.9 fasst die Möglichkeiten der Theorie der semantischen Komponierbarkeit zusammen.

Bisimulation über LTS sind ein vielversprechendes Kriterium, um über die Validität von Modellen zu befinden. Bisimulation vereint Ausdrucksstärke mit relativ guter Berechenbarkeit (Moller u. a. 2004). Darüber hinaus ermöglichen LTS der Modellierung und Simulation von Fortschritten in der theoretischen Informatik zu profitieren. So gibt es auch für DEVS den Versuch die Semantik mittels LTS zu definieren (Posse 2004).

Zu beachten bleibt, dass Validieren sich in der Praxis nicht auf einen einmaligen Schritt oder eine einzelne Methode beschränken lässt. Validierung sollte den Modellerstellungsprozess mittels einer Vielzahl von Methoden kontinuierlich begleiten (Balci 1997).

2.4 Zusammenfassung und Vergleich

Tabelle 2.10 stellt die Stärken und Schwächen der vorgestellten Ansätze noch einmal vergleichend dar. Zeiglers Hierarchie der Systembeschreibungen ist abgekürzt als HSB und die Theorie der semantischen Komponierbarkeit als TSK.

Kriterium	*Grad*	*Bemerkungen*
Spezifiz., Publiz. & Auffinden		
Austauschformat	-	kein Austauschformat
Separate Schnittstelle	○	Mittels schwacher Bisimulation lassen sich Schnittstellen für Modelle als separate Beschreibungen konstruieren.
Implementieren & Verifizieren		
Modellierung	○	Abstraktion von konkreten Modellierungssprachen; allgmein gültig jedoch praktisch schwer nutzbar
Parametrisierung	-	keine Parameter
Verfeinerung	○	Vollständig formalisierte Beziehungen zwischen Systembeschreibungen
Komponieren & Analysieren		
Technische Ebene	-	von der technischen Ebene wird vollständig abstrahiert
Syntax	+	Als Mengen definierbar
Semantik	-	nicht explizit definierbar
Pragmatik	-	nicht explizit definierbar
Dynamik	+	Vollständig formale Beschreibung der Dynamik durch Bisimulation
Konzepte	-	nicht explizit definierbar
Validität	+	Vollständig formalisiert
Experimentieren	-	von der Experimentdurchführung selbst wird vollständig abstrahiert

Tabelle 2.9: Abdeckung der Anforderungen durch die Theorie der semantischen Komponierbarkeit

Modular-hierarchische Formalismen unterscheiden nicht explizit zwischen öffentlichen und privaten Beschreibungsmitteln für Modelle und sind damit nicht für eine entkoppelte Entwicklung geeignet. Während Ptolemy zumindest ein vollständiges Austauschformat für Modelle bereithält, befindet sich eine standardisierte Repräsentation für DEVS noch in der Entwicklung. Publizierbar und austauschbar sind Modelldefinitionen jedoch auch mit Ptolemy nur als Ganzes.

Die Stärken modular-hierarchischer Formalismen liegen darin, Modelle flexibel zu konstruieren und mit Hilfe entsprechender Simulationswerkzeuge kontrollierte Experimente durchzuführen. Simulationsbasierte Ansätze bieten kaum Unterstützung beim Modellieren und eine beschränkte Kontrolle beim Experimentieren.

In Hinblick auf eine schnittstellenbasierte Komposition sind Ansätze auf Basis der HLA zu bevorzugen. Schnittstellen lassen sich als vollständig separate Dokumente publizieren. Die HLA löst zudem die technische Dimension der Kompatibilität am flexibelsten und ermöglicht eine Vielzahl an Implementierungssprachen.

Während sich die syntaktische Ebene allgemein gut erfassen lässt, findet der semantische Aspekt der Kompatibilität in Modellierungsformalismen keine Beachtung. Die semantische Ebene wird bisher vorrangig durch domänenspezifische Austauschformate wie der SBML (Hucka u. a. 2004) unterstützt. Es existieren für DEVS und Ptolemy Konzepte, um Kompatibilität von Modellen auf dynamischer Ebene zu prüfen. Diese finden jedoch aufgrund des hohen Berechnungsaufwands keine Anwendung. BOM stellt Zustandsautomaten zur Verfügung, um den dynamischen Aspekt von Modellen zu deklarieren, jedoch keinen Algorithmus zur Analyse von Kompositionen. Die konzeptuelle Ebene sowie Validitätskriterien lassen sich mit BOM zumindest strukturiert erfassen.

Formale Mittel zum Verifizieren von Verfeinerungen und zum Validieren von Kompositionen stellen die Stärken der theoretischen Ansätze dar. Zeiglers Hierarchie der Systembeschreibungen liefert die Mittel, um die Validität eines Modells hinsichtlich eines Referenzmodells zu überprüfen. Ebenfalls auf Basis von Referenzmodellen bietet die Theorie der Semantischen Komponierbar-

	DEVS	Ptolemy	HLA	BOM	HSB	TSK
Spezifizieren, Publizieren & Auffinden						
Austauschformat	○	+	+	+	-	-
Separate Schnittstelle	-	-	+	+	○	○
Implementieren & Verifizieren						
Modellierung	+	+	-	○	○	○
Parametrisierung	○	+	○	○	-	-
Verfeinerung	○	○	-	-	+	+
Komponieren & Analysieren						
Technische Ebene	○	+	+	+	-	-
Syntax	+	+	+	+	+	+
Semantik	-	-	○	○	-	-
Pragmatik	-	-	○	○	-	-
Dynamik	○	○	-	○	+	+
Konzepte	-	-	○	+	-	-
Validität	○	-	○	○	+	+
Experimentieren	+	+	○	○	-	-

Tabelle 2.10: Ausgewählte Kompositionsansätze für Simulationsmodelle im Vergleich

keit eine Alternative zu der Hierarchie von Zeigler. Mittels der Abbildung von Modellverhalten auf beschriftete Transitionssysteme verspricht die Analyse rechnerisch handhabbar zu bleiben. Die Praxistauglichkeit der formalen Ansätze konnte bisher jedoch nicht anhand realer Simulationsmodelle untermauert werden.

Insgesamt decken existierende Kompositionsansätze jeweils nur Teilbereiche der Anforderungen ab. Es existiert bisher kein Ansatz mit dem Teile von Simulationsmodellen in Modellierungsformalismen erstellt, auf Grundlage von expliziten Schnittstellenbeschreibungen komponiert sowie formal analysiert werden können.

3 Komposition von Software

Komponentenorientierung weckt in der Softwareentwicklung die Erwartung, Softwaresysteme in kürzeren Entwicklungszeiten zu produzieren, komplexe Softwaresysteme besser handhaben und so die Qualität von Softwaresystemen allgemein verbessern zu können (Fröberg 2002). Als Teilbereiche der Informatik stellen sich für die Komposition von Modellen und Software verwandte Herausforderungen (Bartholet u. a. 2004), auch wenn gelegentlich die speziellen Eigenschaften von Simulationsmodellen gegenüber Softwaresystemen — Modelle vereinfachen Systeme mit Hinblick auf bestimmte Fragestellungen — hervorgehoben werden (Davis u. Anderson 2004; Zeigler u. Sarjoughian 2002). Ähnlich zur Kompatibilität von Modellen werden sowohl für Softwarekomponenten (Beugnard u. a. 1999) als auch für Web Services (Medjahed u. Bouguettaya 2005) Ebenen der Kompatibilität unterschieden. Für Modelle, Software und auch Web Services gilt es gleichermaßen, Integrationsprobleme auf syntaktischer, semantischer, dynamischer und qualitativer Ebene zu lösen. Wie im vorigen Kapitel bereits angedeutet, orientieren sich Kompositionsansätze für Simulationsmodelle, insbesondere im Kontext der HLA, an Fortschritten bei der Komposition von Software.

Prominentes Beispiel für ein gemeinsam genutztes Prinzip ist die Objektorientierung. Objektorientierung ist in der Simulationsgemeinschaft akzeptiert, sowohl als Modellierungskonzept (Zeigler 1990; Elmqvist u. a. 2001) als auch für die Implementierung von Simulationssystemen (Pidd u. Carvalho 2006).

Aus softwaretechnischer Sicht wird die objektorientierte Entwicklung jedoch als alleine nicht ausreichend betrachtet, um heterogene Hardwareplattformen zu überbrücken und von Implementierungssprachen sowie der physikalischen Verteilung von Software zu abstrahieren. Objektorientierte Kompositionsoperatoren wie Aggregation und Substitution auf Basis von Vererbungsbeziehungen sind nicht kompositional (Szyperski 2002; Bergmans u. a. 2003). Objektorientierung bietet selbst kaum Hilfestellung rigide[1] und fragile[2] Strukturen in Softwarearchitekturen zu vermeiden (Martin 2003). Auch Entwurfsmuster (Gamma u. a. 1995) und Prinzipien agiler Softwareentwicklung (Martin 2003) adressieren diese Herausforderungen nur teilweise. Sie zielen vorrangig auf die Modularisierung des Entwurfs und nicht explizit darauf, Softwaresysteme am Einsatzort zu integrieren.

Die komponentenbasierte Entwicklung von Software, engl. *Component-based software engineering* (CBSE), zielt explizit darauf, Software unter Verwendung vorfabrizierter Einheiten zu konstruieren (Szyperski 2003). Softwarekomponenten sollen in Einsatzkontexten austauschbar sein, so dass weder sie noch die Kontexte dafür verändert werden müssen (Szyperski 2002). Im Folgenden werden am Beispiel einer konkreten Komponentenplattform grundlegende Prinzipien von CBSE erläutert und auf die Modellkomposition bezogen.

3.1 CORBA — Eine Plattform für Softwarekomponenten

Die *Common Object Request Broker Architecture* (CORBA) ist eine Technologie zur Umsetzung verteilter, objektorientierter Anwendungen (Siegel 2001). Das grundlegende Konzept von CORBA ist das CORBA-Objekt, welches die Heterogenität bezüglich Rechner, Programmiersprachen und Verteilung vor Entwicklern und Anwendern versteckt. Damit nimmt CORBA für Softwaresysteme eine ähnliche Rolle ein wie die HLA für Simulationssysteme. Seit Version 3.0 umfasst CORBA

[1] Teile lassen sich nicht unabhängig voneinander verändern

[2] Änderungen an einer Stelle resultieren in Fehlverhalten an anderer, konzeptuell unabhängiger Stelle

```
typedef long HopCount;
typedef short AddPart;
typedef struct Address {
  AddPart p1;
  AddPart p2;
  AddPart p3;
  AddPart p4;
}
typedef struct Message {
  HopCount hops;
  Address from;
  Address to;
}
interface ISender {
  void send(Message msg);
}
component Protocol {
  requires ISender transport;
}
```

Abbildung 3.1: Definition einer Kontextbedingung mit der Corba-IDL

das *CORBA Component Model* (CCM), welches das CORBA-Objektmodell erweitert und einen standardisierten Weg anbietet, Komponenten einzusetzen (Wang u. a. 2001).

Um von Dritten zur Komposition verwendet werden zu können, müssen die Anforderungen und Angebote einer Komponente klar spezifiziert werden. Mittels Schnittstellen lassen sich direkte Abhängigkeiten auf andere Komponenten vermeiden. Die Implementierung von Komponenten wird hinter Schnittstellenbeschreibungen versteckt und Konsistenzchecks finden allein auf Basis der Schnittstellenbeschreibungen statt. Die Existenz einer Schnittstelle ist nicht an die Existenz einer bestimmten Komponente gebunden (Szyperski 2002). Verschiedene Komponenten können so gleiche Schnittstellenbeschreibungen referenzieren. Komponentenplattformen stellen gewöhnlich spezielle Definitionssprachen für Schnittstellen, engl. *Interface Definition Language* (IDL), bereit.

Zur Beschreibung der Schnittstellen von CORBA-Objekten dient CORBA-IDL. CORBA-IDL erlaubt die Beschreibung von Schnittstellen samt Operationen, Parametern und Datentypen unabhängig von einer konkreten Programmiersprache. IDL-Beschreibungen fungieren als Verträge zwischen Anbietern und Nutzern von Funktionalitäten. Aus einer IDL-Definition wird mittels eines IDL-Compilers zum einen ein Stub für die Nutzerseite generiert und zum anderen ein Skeleton für die Anbieterseite. CORBA unterstützt eine Vielzahl an Implementierungssprachen.

CORBA-Komponenten erweitern CORBA-Objekte um vier Porttypen, die abstrakte Interaktionspunkte definieren. Von einer Komponente benötigte Schnittstellen werden als *receptacle* bezeichnet und in Diagrammen durch Halbkreise repräsentiert. Bereitgestellte Schnittstellen heißen *facet* und werden durch geschlossene Kreise symbolisiert. Im Bereich der Software findet Kommunikation vorrangig als Aufruf von Methoden statt und Schnittstellen fassen im Wesentlichen eine Menge von Methoden zusammen. Eine Schnittstelle zu implementieren, heißt, die in der Schnittstelle deklarierten Methoden zu definieren und zum Aufrufen zur Verfügung zu stellen. CORBA-Komponenten können jedoch auch Ereignisquellen und -senken deklarieren, die asynchrone Verbindungen ermöglichen. Während sich Methoden mit CCM jedoch in Schnittstellen gruppieren lassen, um als angebotene oder benötigte Schnittstelle bekannt gegeben zu werden, lassen sich asynchrone Kommunikationsendpunkte nur einzeln ausstellen.

Beispiel 3.1. *Abbildung 3.1 listet die Definition einer Schnittstelle zur Transportvermittlung in CORBA-IDL.* `Address` *ist als komplexer Typ auf Basis des einfachen Datentyps* `short` *definiert. Die Schnittstelle* `ISender` *deklariert eine Methode zum Versenden des komplexen Typs* `Message`*.*

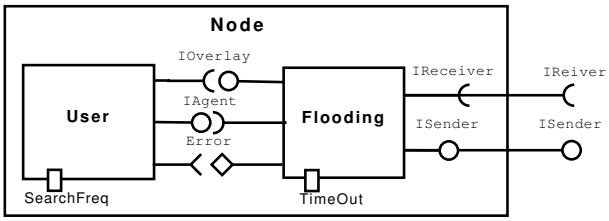

Abbildung 3.2: Definition eines Knotens als zusammengesetzte Komponente

Die Komponente `Protocol` *stellt die Schnittstelle* `ISender` *schließlich als* receptacle *mit dem Namen* `transport` *aus.*

Neben der Fähigkeit wohldefinierte Schnittstellen auszustellen, sollten Komponenten in unterschiedlichen Anwendungskontexten einsetzbar sein. CORBA-Komponenten lassen sich zu diesem Zweck mit Attributen ausstatten (OMG 2002). Komponenten können so am Einsatzort konfiguriert werden.

In Kompositionen werden Komponenten über ihre angebotenen und angeforderten Schnittstellen verbunden. Auf gleicher Ebene müssen jeweils komplementäre Schnittstellen, d.h. eine Facet und eine Receptacle gleichen Typs, verbunden werden. Kompositionen können in CCM selbst als Komponente aufgefasst werden. Verbindungen können so auch zwischen einer Komponente und ihren Teilkomponenten definiert werden, wobei gleichartige Schnittstellen desselben Typs verbunden werden müssen. Somit erlaubt CCM modular-hierarchische Konstruktion ähnlich zu DEVS. Während in DEVS Kommunikationsendpunkte einzeln miteinander verbunden werden, lassen sich mit CCM Kommunikationsfähigkeiten in Schnittstellen zusammenfassen und komplexe Kommunikationsendpunkte miteinander verbinden.

Beispiel 3.2. *Abbildung 3.2 zeigt die grafische Repräsentation der zusammengesetzen Komponente* `Node`*. Ein Knoten beinhaltet die zwei Komponenten* `User` *und* `Flooding`*, die entsprechend der Schnittstellendefinitionen* `IAgent` *und* `IOverlay` *verbunden sind.* `Flooding` *besitzt die Kontextbedingung* `IReceiver` *und bietet* `ISender` *an. Beide Schnittstellen werden an* `Node` *delegiert.* `User` *und* `Flooding` *stellen jeweils einen Parameter aus. Des Weiteren bietet* `Flooding` *eine Ereignisquelle zum Propagieren von Fehlersituationen, die mit einer entsprechenden Ereignissenke der Komponente* `User` *verbunden ist.*

Für Softwarekomponenten lässt sich Kompatiblitätsprüfung auf die Feststellung beschränken, ob zwei Komponenten dieselbe Schnittstellenbeschreibung, einmal als *receptacle* und einmal als *facet*, referenzieren oder nicht. Für Kompatibilität muss sichergestellt werden, dass alle *receptacles* mit *facets* des gleichen Typs verbunden sind.

IDLs für Softwarekomponenten beschränken Schnittstellendefinitionen weitestgehend auf Signaturbeschreibungen und verbleiben damit auf der syntaktischen Ebene. Es existieren Vorschläge auf Basis des π-Kalküls (Canal u. a. 2001; 2003), auf Grundlage von Petri-Netzen (Hameurlain 2005) sowie mittels Automaten (de Alfaro u. Henzinger 2001), die Dynamik von Komponenten zu formalisieren. Diese haben bisher aber keinen Eingang in praktisch relevanten Ansätzen gefunden (Lau u. Wang 2006).

Während IDLs für Softwarekomponenten sich in ihren syntaktischen Mitteln ähneln, ist eine generelle Architektur für Softwarekomponenten derzeit nicht in Sicht. Softwarekomponenten sind immer an eine spezielle Komponentenplattform gebunden, für die sie entwickelt und auf der sie eingesetzt werden. Dies ermöglicht einen relativ geringen Laufzeitüberbau von Komponentenplattformen, prädestiniert Komponenten aber für den Einsatz innerhalb gewisser Grenzen, bspw. in Firmen, in denen die Verwendung einer bestimmten Plattform garantiert ist.

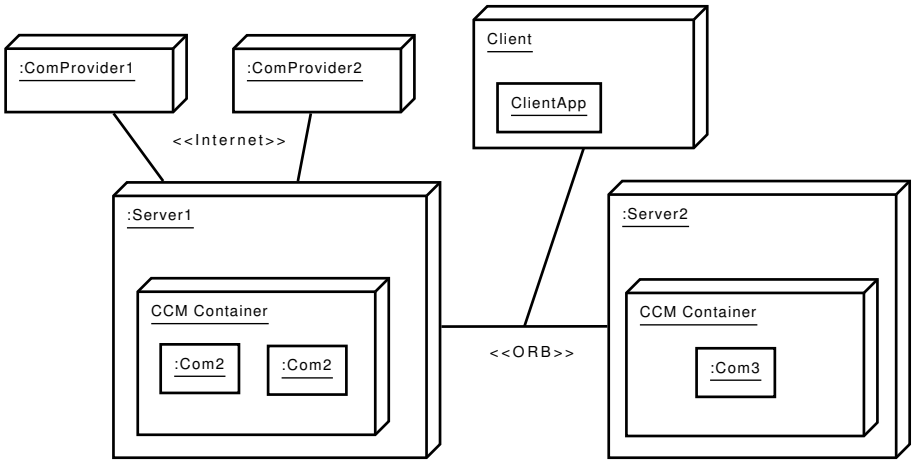

Abbildung 3.3: Entwicklung und Einsatz von CCM-Komponenten

Die Integration auf technischer Ebene leistet in CORBA-Systemen der *Object Request Broker* (ORB). Der ORB (Neubauer u. a. 2004) vermittelt Operationsaufrufe und Ergebnisszustellungen zwischen CORBA-Objekten. CORBA-Objekte werden dafür mit eindeutige Objektreferenzen versehen, die vom konkreten Aufenthaltsort abstrahieren. Für die Kommunikation zwischen ORB und Objektimplementierung wird ein Objektadapter benötigt, der die mögliche Heterogenität der verwendeten Implementierungssprachen überbrückt.

Abbildung 3.3 zeigt eine beispielhafte Verteilung eines CORBA-Systems. Komponenten werden als fertige Pakete von einem Komponentenanbieter erworben und im ORB-Verbund eingesetzt. Bei CCM ist die Verteilung zum Zeitpunkt der Nutzung losgelöst von der Verteilung der Komponentenentwickler. Nicht der Komponentenhersteller instantiiert Komponenten, sondern derjenige, der die Komponente erworben hat. Komponenten können wie klassische Softwarepakete von Anbietern vertrieben und beim Kunden installiert werden. CORBA-Komponenten sind auf unterschiedlichen Zielplattformen ausführbar. Entsprechend der zu unterstützenden Zielplattformen, kann eine Komponente mehrere ausführbare Softwareeinheiten enthalten.

Simulationswerkzeuge sind Softwaresysteme, die über spezielle Funktionalität verfügen, bspw. um Simulationszeit zu kontrollieren, Zufallsereignisse zu generieren und Laufzeitverhalten beobachten zu können (Page u. Kreutzer 2005). Ansätze für Softwarekomponenten lassen sich nach Erweiterung um simulationsspezifische Funktionalitäten auch zur Komposition von Simulationsmodellen nutzten. Praktisch existieren für die Komposition von Simulationsmodellen bereits Lösungen auf Basis von Softwarekomponenten, sowohl für spezielle Domänen wie Netzwerksimulationen (Shen 1996; Cholkar u. Koopman 1999) als auch domänunabhängig für modular-hierarchische Formalismen (Kim u. Kang 2004). Modelle werden in diesen Ansätzen in Programmiersprachen definiert.

Sind Modellbeschreibungen von Simulationalgorithmen getrennt, können Modelle nicht direkt auf einer bestimmten Hardwareplattform ausgeführt werden, sondern erfordern spezielle Simulationswerkzeuge. Während die Heterogenität der Einsatzplattformen für Softwarekomponenten überschaubar ist, existieren für Modelle eine Vielzahl möglicher Simulationswerkzeuge. Plattformunabhängige Simulationskomponenten dadurch zu schaffen, dass für sämtliche potentiellen Simulationswerkzeuge Implementierungen in den Komponenten enthalten sind, ist entsprechend aufwendig. Für Modellkomponenten stellt sich damit die Frage, wie von der Heterogenität der Modellierungsformalismen sinnvoll abstrahiert werden kann. UML ist ein Ansatz, der auf diese Frage eine systematische Antwort gibt.

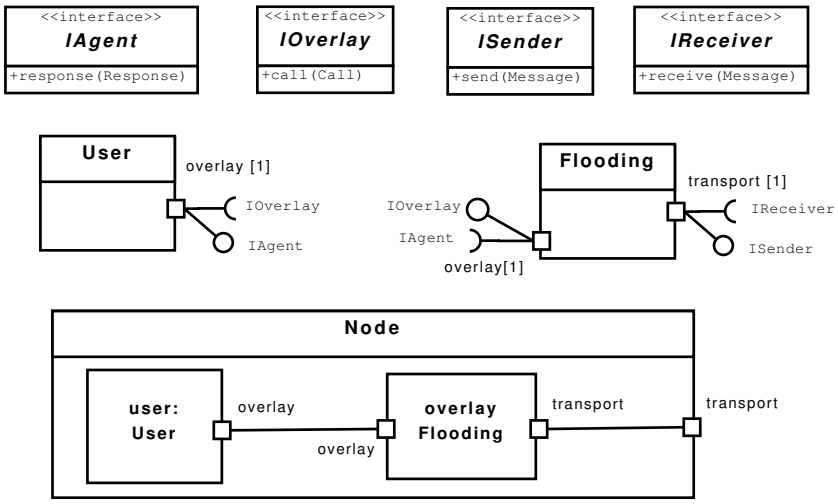

Abbildung 3.4: Schnittstellen, strukturierte Klassen und Kompositionsstrukturen in der UML

3.2 UML, SysML und MDA

Die *Unified Modeling Language* (UML) ist als allgemeine Modellierungssprache konzipiert. Sie eignet sich zur Modellierung verschiedener Arten von Systemen sowie unterschiedlicher Aspekte eines Systems. Galt das bisher vor allem für objektorientierte Systeme, wurden mit Version 2.x (OMG 2005) die Möglichkeiten zum Beschreiben von Komponenten und Kompositionen grundlegend überarbeitet.

Im Hinblick auf die Komponentenorientierung wurden in UML 2.x Kompositionsstrukturdiagramme, engl. *Composite Structure Diagram*, aufgenommen (Oestereich 2006). Kompositionsstrukturen beschreiben die interne Konfiguration einer Komponente oder Klasse und die Beziehungen zwischen ihren Teilen (Oestereich 2006). Kompositionsstrukturen überlappen in ihren Ausdrucksmöglichkeiten zum Teil mit klassischen Komponentendiagrammen der UML — die in ihren Ausdrucksmöglichkeiten wiederum der IDL für CCM ähneln. Mit der neuen Diagrammart lassen sich Kompositionsstrukturen jedoch feiner und flexibler beschreiben. Kompositionstrukturen erlauben einem Port mehrere Schnittstellen, auch mit unterschiedlichen Richtungen, zuzuweisen und Verbindungsmultiplizitäten zu spezifizieren.

Beispiel 3.3. *Abbildung 3.4 zeigt die Schnittstellendefinitionen* `IAgent`*,* `IOverlay`*,* `ISender` *und* `IReceiver`*. Die Klassen* `User` *und* `Flooding` *nutzen diese Schnittstellen, um benötigtes und bereitgestelltes Verhalten über Ports bekanntzugeben.* `User` *und* `Flooding` *werden schließlich in der Kompositionsstruktur von* `Node` *verwendet.* `User` *und* `Flooding` *sind über ihren jeweiligen* `overlay`*-Port verbunden. Die zusammengesetzte Komponente* `Node` *delegiert den Port* `transport` *der Subkomponente* `Flooding` *zu ihrem eigenen Port* `transport`*.*

Die Beschreibungsmittel der UML für Komponenten und Kompositionen eignen sich vorrangig zur Definition von methodenorientierter Kommunikation. Für Modellkomponenten ist die Repräsentation von Ereignisports durch Methodendeklarationen nicht intuitiv. Eingabeports und Ausgabeports lassen sich in Form von Methoden bspw. einzeln als *facet* bzw. *receptacle* repräsentieren (Zinoviev 2005).

Die methodenorientierte Sichtweise von UML wird auch im Bereich des Systems-Engineering als Beschränkung wahrgenommen und hat die Entwicklung der *Systems Modeling Language* (SysML) motiviert. SysML (OMG 2007a) basiert auf UML 2.1 und bietet, zusätzlich zu methodenorientierten Standardports von UML, sogenannte *flow ports*, über die asynchron kommuniziert werden kann. In

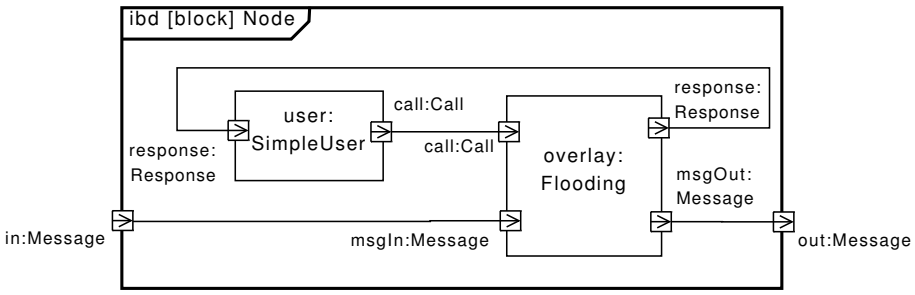

Abbildung 3.5: Knoten als Blockdiagramm in SysML

SysML lassen sich damit Blockdiagramme beschreiben, wie sie in der Modellierung und Simulation gebräuchlich sind.

Beispiel 3.4. *Abbildung 3.5 veranschaulicht Node als internes Blockdiagramm in SysML. Die Spezifikationen der Flüsse sind nicht aufgeführt.*

Um SysML für die Definition von Simulationsmodellen nutzen zu können, bedarf es einer eindeutigen Semantik. SysML erbt als UML-Profil das semantische Grundgerüst der UML.

MDA, Plattformen und Semantik

UML-Modelle fanden in der Vergangenheit vorrangig in der Entwurfs- und Dokumentationssphase Verwendung. Im Kontext der modellbasierten Entwicklung, engl. *Model Driven Architectur* (MDA) beginnt sich die Rolle der UML in der Softwareentwicklung zu wandeln. Die UML fungiert dort als plattformunabhängiges Beschreibungsmittel (Selic 2004b). Die Idee der MDA zielt auf die Minimierung der Kosten von Entwurfsänderungen durch das Ersetzen manueller Implementierungsarbeit durch automatisierte Transformationsprozesse. Änderungen werden an plattformunabhängigen Modellen vorgenommen. Die plattformspezifische Implementierung wird, entsprechend der einmal definierten Transformationsregeln, automatisch aktualisiert (Mellor u. a. 2004).

Der Wunsch nach plattformunabhängigen Spezifikationen besteht auch im Bereich der Modellierung und Simulation, bspw. um konzeptuell stimmige Simulationen zusammenzufügen (Tolk 2004). UML wird bereits direkt zur Modellierung (Page u. Kreutzer 2005) sowie zur Generierung von HLA-Anbindungen aus UML-Modellen (Parr u. Keith-Magee 2004) genutzt. Für Modelica wurde ein MDA-basierter Ansatz zur Definition von Modellen vorgeschlagen (Larsson 2006).

Plattformunabhängigkeit ist in der MDA relativ. Während CORBA plattformunabhängig bzgl. Implementierungssprachen ist, erlaubt MDA von konkreten Komponentenplattformen wie CORBA zu abstrahieren (Schmoelzer u. a. 2004). Eine Plattform kann allgemein als Ausführungsumgebung für eine Menge von Modellen aufgefasst werden (Mellor u. a. 2004). Modelle werden mittels Modellierungssprachen definiert. Welche Modelle mit einer Modellierungsprache ausgedrückt werden können, lässt sich durch Metamodelle festlegen. Ein Metamodell ist somit ein Modell einer Modellierungsprache (Vangheluwe u. de Lara 2002; Favre u. Nguyen 2005). Mittels Metamodellen lässt sich also die Menge von Modellen definieren, die auf einer Plattform ausgeführt werden können. Dies gilt für Softwaremodelle und Simulationsmodelle gleichermaßen.

Auf Grundlage von Metamodellen kann nun der Begriff der Plattformunabhängigkeit präzisiert werden. Plattformunabhängig ist ein Modell, dessen Metamodell von einer Plattformspezifikation oder mehreren Plattformspezifikationen abstrahiert. Plattformabhängig ist ein Modell, das Details über Plattformen beinhaltet.

Mit Blick auf die MDA wurden die Beschreibungsmittel der UML in Version 2 erweitert und die Semantik präzisiert (Selic 2004a; Hogg 2004). Abbildung 3.6 zeigt einen vereinfachten Ausschnitt des syntaktischen Teils des Metamodells für UML-Komponenten. Die UML definiert Komponenten als modulare Einheiten mit wohldefinierten Schnittstellen. Komponenten werden als spezielle

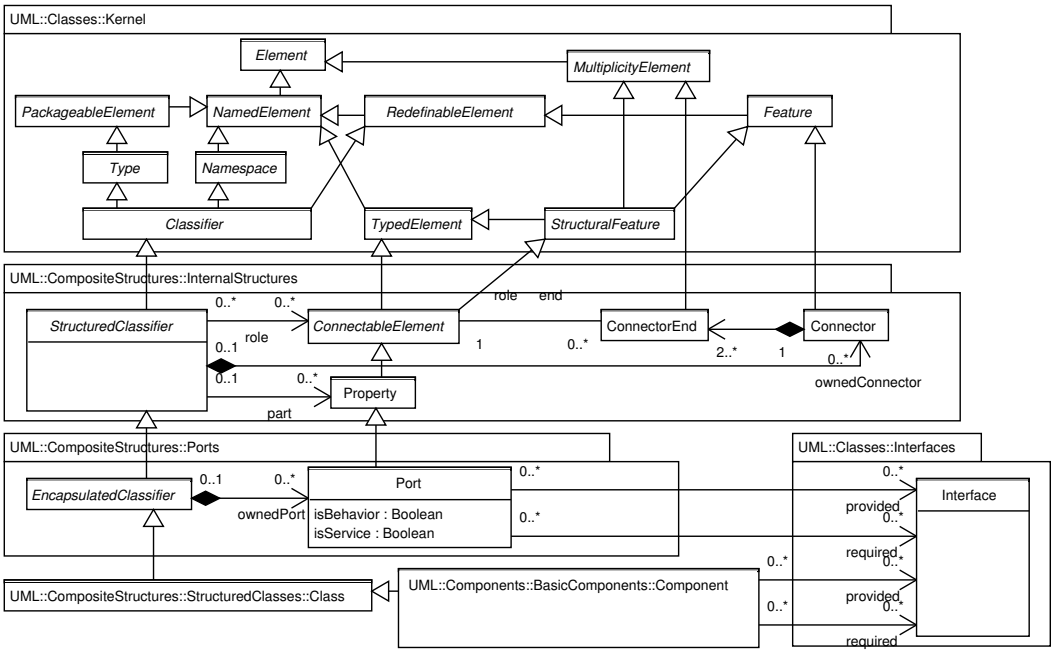

Abbildung 3.6: Abstrakte Syntax für UML-Komponenten, vereinfacht nach (OMG 2006)

Klassen aufgefasst, die einen austauschbaren Teil eines Systems realisieren und die intern über eine oder mehrere Klassen implementiert sind. Um eine möglichst breite Anwendung zu ermöglichen, ist die Semantik von UML bewusst an bestimmten Stellen, genannt semantische Variationspunkte, variabel gehalten (Selic 2004b). Ein Beispiel für einen semantischen Variationspunkt ist der Anschluss von mehreren Verbindungen an einen Port. Variabel ist dabei, ob Nachrichten über einen oder alle angeschlossenen Verbindungen weitergeleitet werden (OMG 2005, 177). UML lässt auch offen, was zwei verbindbare Elemente kompatibel macht (OMG 2007b, 176).

Ein beträchtlicher Teil der Semantik von UML und SysML ist natürlichsprachlich formuliert. Der UML wird daher auch in Version 2 noch unzureichende Semantik vorgeworfen, sowohl allgemein (Harel u. Rumpe 2004; Zhan u. a. 2004; O'Keefe 2006) als auch konkret für Komposita (Oliver u. Luukkala 2006). Bisher dringt die UML mit Kompositionsstrukturdiagrammen nur syntaktisch in den Bereich von Sprachen vor, mittels derer sich Zusammenhänge in Architekturen durch Verbindungen abstrakter Ports formal beschreiben und analysieren lassen (Medvidovic u. Taylor 2000). Es existieren jedoch verstärke Bemühungen die Semantik der UML, zumindest für einen Teil der Sprachelemente, vollständig zu formalisieren (Broy u. a. 2006).

Dienstorientierte Architekturen stellen eine praktische Alternative dar, um Funktionalität komponentenorientiert und plattformunabhängig zu realisieren.

3.3 Web Services und Semantic Web

Das klassische *World Wide Web* (WWW) wurde zur Verknüpfung statischer Informationsangebote konzipiert. In dem Maße, wie über das WWW auch Geschäftsprozesse abgewickelt werden sollen, steigt die Notwendigkeit, dynamische Netzinhalte anbieten und automatisch nutzen zu können. Netzinhalte werden danach nicht als reine Informationen gesehen, sondern als funktionale Einheiten. Die angebotene Funktionalität wird als Dienst bezeichnet. Logik von Prozessabläufen in

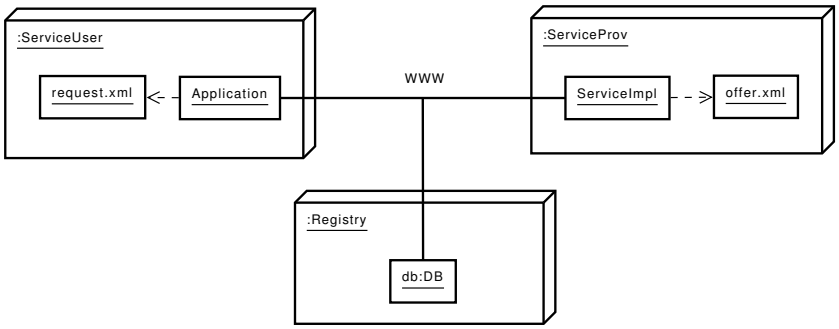

Abbildung 3.7: Verteilungsmodell dienstorientierter Architekturen

kleineren, wiederverwendbaren Einheiten umzusetzen, ist das Gebiet dienstorientierter Architekturen (Erl 2005), engl. *Service Oriented Architecture* (SOA).

Abbildung 3.7 veranschaulicht das prinzipielle Verteilungsmodell dienstorientierter Architekturen. Ein Anbieter erstellt eine Dienstbeschreibung in XML und gibt sie einem Verzeichnisdienst, der *Registry*, bekannt. Unabhängig davon kann ein potentieller Nutzer eine Beschreibung des von ihm gewünschten Dienstes erstellen und von der Registry passende Angebote erfragen. Entspricht eine gefundene Dienstbeschreibungen den Anforderungen des Nutzers, kann dieser den Dienst durch die in der Dienstbeschreibung definierten Bindungen aufrufen. Der Verzeichnisdienst ist ein optionales Element in der Architektur, da Dienstbeschreibungen auch direkt zwischen Anbieter und Nutzer ausgetauscht werden können.

Im Unterschied zum Verteilungsmodell von Softwarekomponenten werden Implementierungen von Diensten nicht an unterschiedliche Orte verbracht. Dienstimplementierungen sind vollständig hinter den Grenzen eines Anbieters versteckt und mit einem festen Ort assoziiert. Allein die Beschreibungen von Diensten werden veröffentlicht. Die Verteilung von Diensten ist nach Kriterien von Geschäftsprozessen gewählt und auf eine maximale funktionale Kapselung gerichtet. Die starke Entkopplung von Implementierungen wird jedoch mit einem, im Vergleich zu Softwarekomponenten, erhöhtem Ausführungssaufwand erkauft.

Das Experimenten mit Modellen stellt spezielle Anforderungen an die Modellrepräsentationen zum Zeitpunkt der Ausführung, bspw. in Hinblick auf die Ausführungseffizienz aber auch hinsichtlich der Beobachtung von Simulationsläufen. Die Kontrolle über die Ausführung von Modellkomponenten sollte soweit möglich sein, dass bspw. ein Simulationsmodell erst nach Analyse des gesamten Modells verteilt und ggf. zur Laufzeit umverteilt wird.

In einer SOA-Lösung zur Simulation von MANETs müssten bspw. die Modellkomponente für die Dienstvermittlung und die Nutzerkomponente über das WWW verbunden werden und Daten in Form von XML austauschen. Für die Simulation von MANETs wird jedoch eine Vielzahl von Knoten benötigt, auf denen jeweils ein Nutzer- und ein Protokollmodell ausgeführt wird. Die Verteilung des Simulationsmodells, z.B. auf Grundlage physikalischer Nähe der MANET-Knoten[3] ist in Hinblick auf die Ausführung sinnvoller als die Verteilung der Modelle nach ihrer Funktionalität.

Anknüpfungspunkte für die Modellierung und Simulation ergeben sich jedoch aus dem Ansatz, Schnittstellen von Diensten strikt XML-basiert zu beschreiben. Datenbeschreibungen auf Basis von XML gelten als robust, erweiterbar und geeignet, auch komplexe Datenstrukturen zu repräsentieren (Harold 2002). XML-Dokumente besitzen eine Baumstruktur, die sich aus geschachtelten Knoten zusammensetzt. XML bietet bis zu einem gewissen Maß selbstbeschreibende Dokumente, in denen Element- und Attributnamen einen informellen Hinweis auf die Bedeutung des Inhalts geben. Des Weiteren integrieren sich XML-basierte Speicherformate gut in Datenbanken und erleichtern den Austausch von Modellen über das Internet.

[3] Physikalische Nähe von MANET-Knoten impliziert erhöhtes Kommunikationsaufkommen

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="unihro/diane/base/service" ...>
  <xs:complexType name="Call">
    <xs:choice>
      <xs:element name="call" type="ServiceOfferCall"/>
      <xs:element name="call" type="ServiceRevokeCall"/>
      <xs:element name="call" type="ServiceSearchCall"/>
    </xs:choice>
  </xs:complexType>
  <xs:complexType name="ServiceSearchCall">
    <xs:sequence>
      <xs:element name="inquirer" type="Address"/>
      <xs:element name="serv" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Address">
    <xs:attribute name="p1" type="xs:integer"/>
    <xs:attribute name="p2" type="xs:integer"/>
    <xs:attribute name="p3" type="xs:integer"/>
    <xs:attribute name="p4" type="xs:integer"/>
  </xs:complexType> ...
</xs:schema>
```

Abbildung 3.8: Definition von `Call` mit XSD

Ein Mittel, um dienstorientierte Architekturen zu realisieren, sind *Web Services*. Web Services lösen die drei grundlegenden Operationen dienstorientierter Architekturen: Publizieren, Auffinden und Aufrufen.

Der Architekturvorschlag des *World Wide Web Consortium* (W3C) für Web Services (W3C 2004d) besteht aus aufeinander aufbauenden Schichten, die unterschiedliche Ebenen der Interoperabilität adressieren. Anders als die Plattformen für Softwarekomponenten kombiniert die Web Service Architektur des W3C offene, zueinander kompatible Beschreibungsmittel und befreit Entwickler von der Bindung an konkrete Einsatzplattformen. Im Folgenden werden Beschreibungsmittel für die syntaktische, semantische und dynamische Ebene vorgestellt.

3.3.1 Syntax

Auf syntaktischer Ebene erlauben Schemasprachen Einschränkungen zu definieren, gegen die XML-Daten geprüft werden können. So können Auftretensbeschränkungen und Reihenfolgen von Elementen festgelegt werden. *XML Schema Definition* (XSD) erlaubt auch den Inhalt von Elementen und Attributen durch Typisierungen und Wertebereiche vorzugeben (W3C 2004f). XSD ist durch das W3C standardisiert und erlaubt, Definitionen in Namensräumen zu strukturieren.

XSD ist nicht darauf beschränkt, im Kontext von Software genutzt zu werden, sondern auch für die Modellierung und Simulation prinzipiell geeignet. Mittels XSD lassen sich die Struktur und der Inhalt eines XML-Dokuments standardisiert und plattformunabhängig definieren.

Beispiel 3.5. *Abbildung 3.8 listet die XML-Schemadefinition von* `Call` *als komplexen Typ. Ein Call kann veranlassen, einen Dienst anzubieten, zu suchen oder eine Veröffentlichung zurückzuziehen. Die drei Alternativen sind jeweils wieder als komplexe Typen definiert. Eine Dienstsuche besteht zum Beispiel aus der Adresse des suchenden Knotens und dem Namen des zu suchenden Dienstes. Adressen enthalten primitive Datentypen als Blätter.*

Schemasprachen bilden die Grundlage, um den syntaktischen Teils eines Web Services zu beschreiben. Der Standard (W3C 2006c) zur Beschreibung einer Schnittstelle heißt *Web Service Description*

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl" ...>
  <types><import namespace="unihro/diane/base/service"/></types>
  <interface name="ServiceReq" >
    <operation name="call" pattern="http://www.w3.org/2006/01/wsdl/in-only">
      <input messageLabel="msgIn" element="service:Call" />
    </operation> ...
  </interface>
  <binding name="serviceSOAPBinding" interface="ServiceReq"
      type="http://www.w3.org/2006/01/wsdl/soap">
    <operation ref="call"
        wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/> ...
  </binding>
  <service name="serviceReqService" interface="tns:ServiceReq">
    <endpoint name="serviceReqEndpoint" binding="tns:ServiceSOAPBinding" address
        ="http://www.example.com/2004/protocol"/>
  </service>
</description>
```

Abbildung 3.9: Definition von `ServiceReq` mit WSDL

Language (WSDL). WSDL definiert die Schnittstelle eines Dienstes mittels einer Menge abstrakter Operationen, die jeweils bestimme Funktionalitäten anzeigen (W3C 2006b) und nimmt damit eine ähnliche Rolle wie IDLs bei Softwarekomponenten ein. Mit WSDL werden Operationssignaturen und Fehlerwerte aufbauend auf Typdefinitionen deklariert.

WSDL-Beschreibungen bestehen aus vier wesentlichen Teilen. Die prinzipiell übertragbaren Nachrichtenarten werden im Element *types* definiert. Das Element *interface* beschreibt die bereitgestellten Operationen. Eine Operation wird als Menge auszutauschender Nachrichten definiert und übernimmt damit eine ähnliche Funktion wie *receptacles* und *facets* von Softwarekomponenten. Im Abschnitt *binding* wird definiert, mittels welchem Nachrichtenformat und Übertragungsprotokoll Nachrichten ausgetauscht werden sollen. Das Element *service* gibt schließlich an, wo der Dienst lokalisiert ist und zugegriffen werden kann.

Beispiel 3.6. *In Abbildung 3.9 ist die Definition der Schnittstelle* `ServiceReq` *mittels WSDL aufgeführt. Die XSD-Typen aus Beispiel 3.5 werden importiert und auf ihrer Basis wird die Operation* `call` *definiert. Die Definitionen von* `response` *und möglicher Fehlersituationen wurden ausgelassen.*

Während Softwarekomponenten in geschlossenen Systemen zum Einsatz kommen, sind Web Services für offene Umgebungen wie das WWW konzipiert. Softwarekomponenten können als einsetzbare Einheiten Bestandtteil anderer Komponenten werden. Web Services sind hingegen an einen festen Ort gebunden und interagieren mit wechselnden Partnern. WSDL-Dokumente definieren, anders als IDLs für Softwarekomponenten, eine Schnittstelle für *eine* Implementierung. Für die Nutzung von Diensten muss überprüft werden, ob Schnittstellenbeschreibungen zueinander kompatibel sind. Da Web Services in Schnittstellen jeweils ihre Implementierung beschreiben, erfordert Kompatibilitätsprüfung ggf. den Vergleich von Typdefinitionen, die unterschiedliche Schemasprachen nutzen.

Technologien aus dem Bereich dienstorientierter Architekturen bieten allgemein neue Möglichkeiten für die Modellierung und Simulation (Tolk u. a. 2006). Konkret wird versucht, Fortschritte aus diesem Bereich zu nutzen, um Simulationssysteme zu koppeln (Blais u. a. 2005; Pullen u. a. 2005) und auch, um Modelle eines bestimmten Formalismuses auszuführen (Kim u. Kang 2005; Mittal u. a. 2007).

Eine WSDL-Erweiterung der HLA stellt die HLA Evolved Web Service API dar (Möller u. Dahlin

2006). HLA Evolved liefert eine spezielle WSDL-Schnittstelle für HLA, die in Kombination mit klassischen HLA-Anbindungen genutzt werden kann. Die Anbindung über WSDL ist jedoch mit einem beträchtlichen Mehraufwand verbunden. Der Laufzeitaufwand für den Datenaustausch über die WSDL-Schnittstelle erhöht sich um den Faktor 30, verglichen mit der Standardversion der RTI (Möller u. Dahlin 2006). Da die technischen Mittel zur Realisierung von Verbindungen transparent sind, können die Vorteile der HLA (höhere Ausführungseffizienz im Vergleich zu einem XML-basiertem Datenaustausch) und die Vorteile von Web Services (stärke Entkopplung der Systeme) individuell bewahrt werden. Dies erleichtert die Koordination komplexer Federations (Möller u. a. 2007c).

3.3.2 Dynamik

Für die zeitlich verschränkte Ausführung mehrerer Web Services werden zwei Modi unterschieden. *Orchestrierung* integriert eine vorhandene Menge von Diensten über einen zentralen Koordinator, der als Fassade fungiert und andere Dienste aufruft. Orchestrierung schafft einen neuen Dienst durch Koordinierung des Kontroll- und Datenflusses zwischen einer Menge existierender Dienste. Orchestrierung kann als hierarchische Konstruktion interpretiert werden, bei dem ein Web Service als Fassade nach außen fungiert. Beschreiben lässt sich Orchestrierung, aufbauend auf WSDL, mit der *Web Services Business Process Execution Language*, abgekürzt WS-BPEL (OASIS 2007).

Im Unterschied zur Orchestrierung gibt es bei der *Choreographie* keine zentrale Instanz zur Koordination, sondern nur die Rollenbeschreibungen der beteiligten Entitäten. Choreographie widmet sich der Problematik, ob Web Services zueinander kompatibel sind. Choreographie definiert die Abfolge und Bedingungen des Nachrichtenaustausches zwischen unabhängigen Web-Services (W3C 2004e). Der Verschlag des W3C zur Standardisierung von Choreographiebeschreibungen heißt *Web Services Choreography Description Language*, abgekürzt WS-CDL, (W3C 2005).

Die beiden Standards zur Orchestrierung und Choreographie eignen sich nicht direkt zur formalen Analyse (Bordeaux u. Salaün 2005). Vorteilhafter ist es, die öffentlichen Schnittstellenbeschreibungen auf formale Sprachen abzubilden (ter Beek u. a. 2006). Um Interaktion von Web Services zu analysieren, existieren Ansätze mittels Automaten (de Alfaro u. Henzinger 2005), Petrinetzen (Hamadi u. Benatallah 2003) und Prozessalgebren (Salaün u. a. 2004). Letztere werden bspw. genutzt, um die Dynamik von Diensten allgemein zu analysieren (Cámara u. a. 2006), über die Kompatibilität von Diensten zu befinden (Liu u. a. 2005) und, bei leichten Kompatibilitätsabweichungen, automatisch Adaptoren zu generieren (Brogi u. a. 2004).

Prozessalgebren können die drei grundlegenden Arten von Komposition in dynamischen Systemen ausdrücken (Baeten 2005). Mittels sequentieller Komposition lassen sich Systembeschreibungen aneinanderreihen. Die alternative Komposition beschreibt den Kontrollfluss und die parallele Komposition die Zusammenstellung konkurrierend laufender Prozesse in einem System. Algebren ermöglichen Kongruenzen zu definieren und damit, für die dynamische Ebene, Kompositionalität vollständig zu formalisieren. Kompositionalität ist jedoch insbesondere für Verhaltensbeschreibungen herausfordernd, wo alternative und parallele Komposition kombiniert werden sollen (van Glabbeek 2005).

Als Standard zum Beschreiben der operationalen Semantik von Prozessalgebren haben sich LTS durchgesetzt (Aceto u. a. 2005). Für die Analyse von Kompatibilität mittels LTS lassen sich verschiedene Arten dynamischer Kompatibilität definieren (Bordeaux u. a. 2005). Auf der Basis der schwachen Bisimulation, lässt sich bspw. Kompatibilität als gegensätzliches Verhalten von A und B durch $A \sim \overline{B}$ formalisieren (Bordeaux u. a. 2005). $\overline{B}$ erhält man dabei aus B, indem alle Eingaben durch gleichnamige Ausgaben ausgetauscht und alle Ausgaben durch gleichnamige Eingaben ersetzt werden. Etwas weniger restriktiv ist es, zu erlauben, dass ein Prozess mindestens die Nachrichten seines Gegenübers akzeptieren muss, aber auch weitere empfangen darf. Eine dritte Anforderung kann sein, dass die Komposition einen Endzustand erreichen muss. Sinnvoll sind sowohl für Web Services als auch für Simulationsmodelle Kompatibilitätsdefinitionen, die auf Terminie-

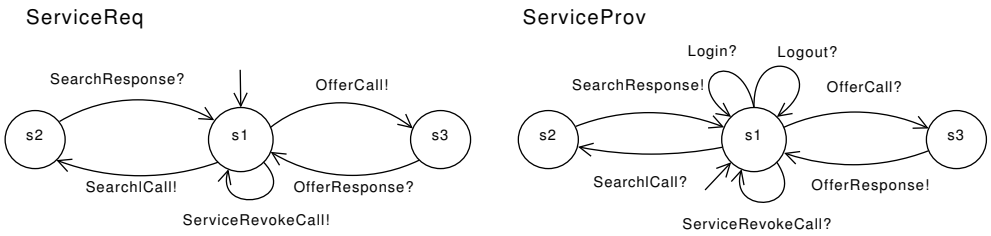

Abbildung 3.10: Kompatible Dynamik von ServiceReq und ServiceProv

rungsforderungen verzichten und stattdessen Interaktionen als kompatibel ausweisen, falls sie nicht fehlschlagen.

Beispiel 3.7. *Abbildung 3.10 visualisiert* `ServiceReq` *und* `ServiceProv` *als LTS. Die Komposition erreicht keinen Endzustand. ServiceReq und ServiceProv sind auch nicht komplementär, da* `ServiceProv` *in der Lage ist, Calls vom Typ Login und Logout zu empfangen, wobei ServiceReq diese nicht zu senden vermag. ServiceProv ist jedoch kompatibel zu ServiceReq, wenn man fordert, dass mindestens alle gesendeten Nachrichten empfangen werden müssen. Aus Sicht von ServiceProv ist das Senden von Login und Logout optional.*

Die konkrete Syntax von Beschreibungen für Web Services auf formale Sprachen abzubilden, schafft Anknüpfungspunkte für die Modellierung und Simulation. Durch die Abstraktion ist nicht mehr relevant, ob das abgebildete System einen Dienst oder ein Modell repräsentiert. Diese These wird auch dadurch gestützt, dass LTS bereits als semantische Domäne für Modellierugsformalismen verwendet werden (Petty u. a. 2005; Posse 2004).

Die Komposition steht jedoch vor einer weiteren Herausforderung. Dynamische Beschreibungen sind nur sinnvoll miteinander in Beziehung zu setzen, wenn auszutauschende Daten eindeutig definiert sind. Neben der Dynamik gilt es auch die Semantik des Informationsflusses zu berücksichtigen.

3.3.3 Semantik

Klassische WWW-Technologien sind auf die Präsentation von Informationen für Menschen zugeschnitten. Das Semantic Web vollzieht die Evolution von manuell zu automatisch auswertbaren Repräsentationen (Berners-Lee u. a. 2001). Dafür müssen, zusätzlich zu den Daten selbst, auch deren Bedeutung explizit und möglichst umfassend zur Verfügung gestellt werden.

Die Grundlagen des Semantic Web bilden Unicode als Konvention zur international einheitlichen Repräsentation von Zeichen und *Uniform Resource Identifier* (URI) als Mittel zur eindeutigen Identifizierung von Objekten. URIs ermöglichen beliebige Kombinationen aus lokalen und globalen Verweisen, so dass Dokumente transparent im globalen Maßstab verteilt werden können. Über XML, Namensräume und XML-Schema lassen sich Beschreibungsformen des Semantic Web mit anderen XML-basierten Standards integrieren und kombinieren.

Der Kernbereich des Semantik Web beginnt, wo die Beschreibungsmöglichkeiten von XML und XSD enden. Mittels Schemasprachen lassen sich Typen definieren, die Aufschluss darüber geben, wie Daten mittels XML repräsentiert werden. Sie liefern keine formalen Mittel, um diese Daten mit Dingen aus der realen Welt in Beziehung zu setzen.

In Anlehnung an die Grammatik natürlicher Sprachen, lassen sich Aussagen in dem *Resource Description Framework* (RDF) aus Subjekt, Prädikat und Objekt zusammensetzen (W3C 2004c). Alle drei Elemente werden in RDF mittels URIs referenziert. Mittels RDF lassen sich so bspw. Ontologien definieren. Ontologien stellen Konzepte in einen Bedeutungszusammenhang, um sie gemeinsam nutzen zu können (Shadbolt u. a. 2006).

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  ...
  <rdfs:Class rdf:about="http://...sne/ndl/ip#IPAddress">
    <rdfs:isDefinedBy rdf:resource="http://...sne/schema/ip.rdf"/>
    <rdfs:label xml:lang="en">IP address</rdfs:label>
    <rdfs:comment>An IP address, either IPv4 or IPv6.</rdfs:comment>
    <rdf:type rdf:resource="http://...sne/ndl/layer#LabelType"/>
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
  </rdfs:Class>
  <rdfs:Class rdf:about="http://...sne/ndl/ip#IPv4Address">
    <rdfs:isDefinedBy rdf:resource="http://...sne/schema/ip.rdf"/>
    <rdfs:label xml:lang="en">IPv4 address</rdfs:label>
    <rdfs:comment>An IPv4 address in dotted decimal notation.</rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://...sne/ndl/ip#IPAddress"/>
  </rdfs:Class> ...
</rdf:RDF>
```

Abbildung 3.11: IP-Adresse in RDF, adaptiert von (van der Ham u. a. 2006)

RDF bietet durch die konsequente Nutzung von URIs eine dezentrale Lösung zur Beschreibung von Typsystemen und damit die Voraussetzung, unabhängig voneinander entwickelte Komponenten zueinander in Beziehung zu setzen. Damit kann die semantische Dimension der Kompatibilität erfasst werden. Eine Herausforderung liegt in der Entwicklung und Standardisierung von Ontologien.

In der Modellierung und Simulation gelten Ansätze des Semantic Web allgemein als vielversprechend (Fishwick u. Miller 2004; Blais u. a. 2005). RDF lässt sich direkt auch zur Beschreibung von Modellen nutzen (Miller u. Baramidze 2005).

Beispiel 3.8. *Abbildung 3.11 zeigt ein RDF-Definition, die eine IPv4-Adresse als Unterklasse einer IP-Adresse definiert. IPAddress selbst ist abgeleitet von dem Standard-RDF-Typen* `Literal`*. Die Definition wurde, leicht verändert, aus der Network Description Language (van der Ham u. a. 2006; SNE 2006) übernommen.*

Web Services stellen semantische Beschreibungen vor eine neue Herausforderung. Wo Informationsverarbeitung in Prozesse eingebettet ist, reichen statische Semantikbeschreibungen für das automatische Prozessieren nicht aus. Bieten Web Services den Übergang von statischen zu dynamischen Netzinhalten und das Semantic Web den Übergang von syntaktischen zu semantischen Inhalten, zielen Semantic Web Services auf die semantische Beschreibung dynamischer Inhalte (Daskalova u. Atanasova 2005). Im Sinne dienstorientierter Architekturen liegt ein Schwerpunkt darauf, Dienste basierend auf öffentlichen Beschreibungen automatisch auffindbar zu gestalten (Alesso 2004). Zusätzlich zu WSDL müssen Eigenschaften und Fähigkeiten eines Dienstes beschrieben werden, so dass sein Zweck automatisch inferiert werden kann. Komposition umfasst hier das Selektieren und Kombinieren von Web Diensten, um bestimmte Zielstellungen zu erreichen.

Die *Web Ontology Language for Web Services* (OWL-S) ist ein Standard (W3C 2004b), der semantische und dynamische Beschreibungen kombiniert. OWL-S umfasst drei wesentliche Teile. Die Angebote und Anforderungen sowie nichtfunktionale Eigenschaften werden im *profile* beschrieben. Im *process*-Abschnitt wird spezifiziert, wie ein Dienst arbeitet. Der Abschnitt setzt Eingaben und Ausgaben mit Vor- und Nachbedingungen in Beziehung und gibt ggf. an, aus welchen Teilprozessen ein Dienst besteht. Das *grounding* definiert, wie ein Dienst genutzt werden kann und stellt damit die Beziehung zu WSDL her.

Die Beschreibungsmöglichkeiten für Prozessstrukturen in OWL-S duplizieren teilweise die Funktionen von Standards für Web Services. Demgegenüber synthetisieren *Semantic Annotations for*

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="unihro/diane/com/service/v2"
    targetNamespace="unihro/diane/com/service/v2"
    xmlns:net="unihro/diane/base/xmlSchemas/net"
    xmlns:sawsdl="http://www.w3.org/2002/ws/sawsdl/spec/sawsdl#">
  <xs:import namespace="unihro/diane/base/xmlSchemas/net"/>
  <xs:complexType name="Call"
                  sawsdl:modelReference="unihro.diane.base.Call">
    <xs:choice>
      <xs:element name="logInCall" type="LogInCall"/>
      <xs:element name="serviceSearch" type="ServiceSearchCall"/> ...
    </xs:choice>
  </xs:complexType>
  <xs:complexType name="ServiceSearchCall" sawsdl:modelReference="...">
    <xs:sequence>
      <xs:element name="inquirer" type="net:Adress"/>
      <xs:element name="serv" type="xs:string"/>
    </xs:sequence>
  </xs:complexType> ...
</xs:schema>
```

Abbildung 3.12: Typdefinitionen in XSD inklusive semantischer Annotationen

WSDL (SAWSDL) semantische und dynamische Beschreibungen aus der Perspektive von Web Services (W3C 2006a). SAWSDL erweitert XSD- und WSDL-Beschreibungen um Annotationen, die auf semantische Informationen verweisen. Damit integriert sich SAWSDL nahtlos in die Architektur für Web Services.

Beispiel 3.9. *Abbildung 3.12 listet die Definition des komplexen Typs* `Call` *mittels XSD inklusive SAWSDL-Attribut auf. Die Definition ist, bis auf die Erweiterung mit dem Attribute modelRef, identisch mit Beispiel 3.5.*

Zur Analyse semantischer Prozessbeschreibungen wird KI-Planung eingesetzt (Srivastava u. Koehler 2003). Zur Reduktion der Komplexität semantischer Analyse existieren Vorschläge zur Unterscheidung von Phasen (Agarwal u. a. 2005). Alternativ kann auch der dynamische Anteil der Komposition variabel gestaltet werden (Küster u. König-Ries 2006). Reine Prozessbeschreibungen sind im Allgemeinen jedoch leichter auszuwerten als Planungsprobleme (Cámara u. a. 2005).

Semantische Prozessbeschreibungen werden in der Modellierung und Simulation bisher nur vereinzelt genutzt (Uhrmacher u. a. 2005; Röhl u. a. 2007b). Insgesamt lässt sich, wie für Softwaretechniken allgemein, auch für semantische Beschreibungssprachen feststellen, dass sie für die Modellierung und Simulation nützlich, jedoch alleine nicht ausreichend sind, um Modelle inklusive verwendeter Konzepte zu beschreiben (Tolk 2006).

3.4 Zusammenfassung und Diskussion

Komponentenorientierte Softwareentwicklung lindert Defizite der Objektorientierung in Hinblick auf die Entwicklung und den Einsatz von Softwareeinheiten. In Komponentenplattformen wie CCM wird strikt zwischen Schnittstelle und Implementierung unterschieden. Softwarekomponenten sind parametrisierbar und modular-hierarchisch konstruierbar. Mit Softwarekomponenten lassen sich die technische und syntaktische Ebene der Komposition gut abdecken, indem IDLs von konkreten Programmiersprachen abstrahieren und Komponentenplattformen Anbindungen an unterschiedliche Programmiersprachen bereitstellen.

Mit UML steht ab Version 2 eine plattformunabhängige Beschreibungssprache für Kompositionsstrukturen bereit. UML abstrahiert von der spezifischen Syntax unterschiedlicher IDLs sowie von konkreten Einsatzplattformen. Die Beziehung zu konkreten Einsatzplattformen kann im Rahmen der MDA formalisiert werden. Plattformspezifische Implementierungen lassen sich zumindest teilweise generieren.

An die Stelle homogener Middleware für Softwarekomponenten treten bei dienstorientierten Architekturen XML-basierte Standards. Mit Web Services können Organisationsgrenzen zur Laufzeit leichter überwunden und sehr heterogene Systeme integriert werden. Im Kontext von SOA existiert mit WSDL ein akzeptierter Standard zur Schnittstellenbeschreibung, dessen Funktionsumfang, in Kombination mit XML-Schemasprachen, mit dem von IDLs vergleichbar ist. Die besondere Stärke von Web Services und Semantic Web liegt dabei in dem geschichteten, modularen Aufbau ihrer Architektur aus sich gegenseitig ergänzenden Standards. So lassen sich eine Reihe von XML-basierten Standards mit WSDL kombinieren, um bspw. den semantischen und dynamischen Aspekt der Kompatibilität abzudecken. Letztendlich gilt aber auch für Web Services, dass die vollständige Beschreibung mittels einer adäquaten, formal definierten und genügend ausdrucksstarken Sprache ein offenes Problem ist (Bordeaux u. Salaün 2005).

Mit Blick auf die Anforderungen an Kompositionsansätze für Simulationsmodelle lassen sich folgende Punkte festhalten:

- *Austauschformat:* Datenformate auf Basis von XML eignen sich prinzipiell zum Publizieren, Speichern und Anfragen in Datenbanken. XML ermöglicht in Kombination mit URIs, Beschreibungen flexibel zu verteilen und ist für eine dezentrale Entwicklung prädestiniert.

- *Separate Schnittstelle:* IDLs für Softwarekomponenten, UML und WSDL können eine Menge abstrakter Kommunikationsendpunkte bekannt geben, über die Funktionalität bereitgestellt bzw. gefordert wird. Eine Komponente kann so mehrere Rollen ausstellen, die jeweils unterschiedliche Aspekte des gesamten Kommunikationspotentials beschreiben.

- *Modellierung:* Die Kompositionsstrukturdiagramme der UML und die Beschreibungsmöglichkeiten von CCM ähneln, den in der Modellierung und Simulation gebräuchlichen, Blockdiagrammen. IDLs und auch UML sind jedoch vorrangig methodenorientiert. SysML erweitert UML um ereignisorientierte Portkonzepte. Der Nutzung von UML und SysML zum Erstellen ausführbarer Modelle steht ihre nur teilweise formalisierte Semantik im Wege.

- *Technik:* Weder das Einsatzmodell von Softwarekomponenten noch von Web Services ist für Modellkomponenten optimal. Die Vielzahl an Modellierungssprachen und Simulationswerkzeugen verhindert, Modellkomponenten mit direkt ausführbaren Implementierungen auszustatten, wie es für Softwarekomponenten üblich ist. Die Architektur von Web Service ist nicht darauf ausgelegt, eine Vielzahl feingranularer Modelle miteinander zu kombinieren und ggf. zur Laufzeit die Lastverteilung zu ändern.

- *Syntax & Semantik:* Während IDLs für Softwarekomponenten vorrangig auf die Komposition methodenorientierter Einheiten zielen und sich auf die syntaktische Dimension beschränken, lassen sich Standards wie URIs, XML, XML-Schema, SAWSDL und RDF miteinander kombinieren und direkt auch für die Beschreibung von Modellkomponenten nutzen.

- *Dynamik:* Formale Analysetechniken sind für Web Services ausgereifter als für Simulationsmodelle. Jedoch sind Prozessalgebren und LTS in ihrer Verwendung nicht auf Software begrenzt, sondern lassen sich auch für Simulationsmodelle nutzen. Welche konkrete Algebra für Simulationsmodelle sinnvoll ist, ist derzeit noch ungeklärt.

- *Experimentieren:* Das Einsatzmodell von Softwarekomponenten ist in Hinblick auf die Ausführunggeschwindigkeit adäquater für Simulationsmodelle als der Ansatz von Web Services.

Als einsetzbare Einheiten können Modellkomponenten flexibler beobachtet und kontrolliert werden.

Diese Evaluierungsergebnisse werden im Folgenden den Rahmen bilden, um einen Ansatz zur Komposition von Simulationsmodellen zu konzipieren.

4 Eine Plattform zur modellbasierten Komposition von Simulationsmodellen

Modelle sollen in Zeit und Raum verteilt entwickelt werden können. In spezifischen experimentellen Kontexten müssen sie jedoch flexibel einsetzbar und kontrolliert ausführbar sein. Gleichzeitig sollten Schnittstellenbeschreibungen so beschaffen sein, dass sie in Datenbanken veröffentlicht sowie auf ihrer Basis Modelle ausgewählt und komponiert werden können.

Im Folgenden wird ein Ansatz zur Komposition von Simulationsmodellen entwickelt, der die theoretische Fundierung modular-hierarchischer Modellierungsformalismen mit den Vorteilen vollständig separater Schnittstellenbeschreibungen kombiniert. Abbildung 4.1 stellt die zu entwickelnden Beschreibungsmittel in Beziehung zu existierenden Modellierungsformalismen. In Kompositionen stellt sich das Problem heterogene Modelle zu integrieren, bspw. AM und BM, definiert in den Modellierungsformalismen α und β. Die zu entwickelnden Beschreibungsmittel für Schnittstellen, Komponenten und Kompositionen sollen unabhängig von konkreten Modellierungsformalismen verwendbar sein.

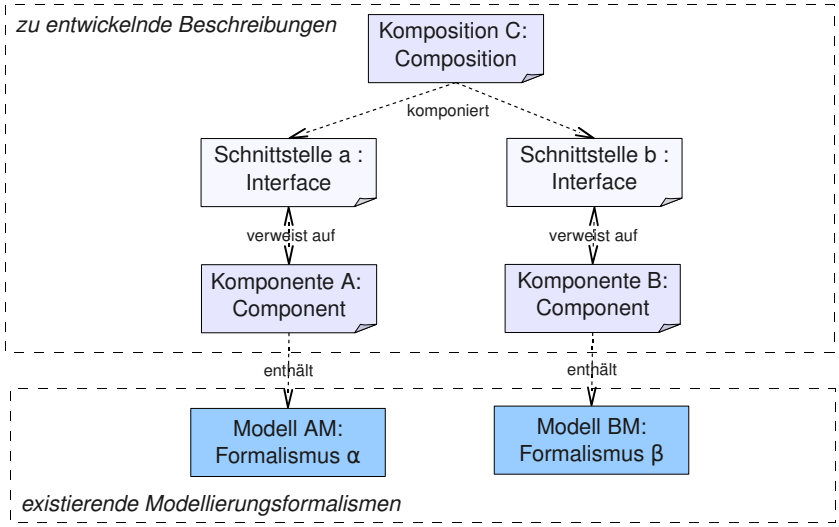

Abbildung 4.1: Beziehung zwischen existierenden und zu entwickelnden Beschreibungen

Die Entwicklung vollzieht sich in vier Schritten:

1. Konzeption und Definition syntaktischer Strukturen (abstrakte Syntax) für Schnittstellen, Komponenten und Kompositionen.
2. Definition der Semantik von Komponenten und Kompositionen. Dies geschieht mittels einer Funktion, die Komponenten auf den Modellierungsformalismus PDEVS abbildet.
3. Formulierung von Anforderungen an Kompositionen, wie Kompatibilität, Verfeinerung und Vollständigkeit, auf deren Basis sich für ein Simulationsmodell positive Eigenschaften garantieren lassen.

4. Bereitstellung einer konkreten Syntax für die Beschreibungsmittel der Plattform in XML.

Die Ergebnisse der ersten beiden Schritte bilden zusammengenommen ein Metamodell einer Kompositionsplattform. Im dritten Schritt werden die formalen Mittel definiert, mit denen sich Kompositionen alleine auf der Grundlage von Schnittstellen und den Regeln des Zusammensetzens analysieren lassen. Der letzte Schritt ermöglicht, die Beschreibungseinheiten konkret zu nutzen und sie in Form von XML-Dokumenten in Datenbanken zu verwalten.

4.1 Beschreibungsmittel

Die Definition eines Metamodells für die zu entwickelnden Beschreibungsmittel bedarf selbst auch eines Formalismuses. UML wurde für den Zweck der Metamodellierung und speziell im Kontext von MDA in Version 2.0 überarbeitet. In der Praxis wird UML verwendet, um bspw. Metamodelle für XML-basierte Sprachen zu definieren (Bernauer u. a. 2004). In der Modellierung und Simulation nutzt die BOM-Spezifikation (SISO 2006a) Klassendiagramme der UML zur Repräsentation von Syntaxstrukturen. Der UML-Kern ist jedoch komplex und unübersichtlich und es existiert Kritik an der rekursiven und halb-formalen Definition der UML-Semantik (Favre 2004; Broy u. a. 2006). Die abstrakte Syntax des Metamodells wird aus diesen Gründen mengentheoretisch definiert. In Abschnitt 4.4 nimmt UML dann die Rolle des Bindeglieds zwischen abstrakter und konkreter Syntax ein.

4.1.1 Typen

Für eine Komponentenplattform ist entscheidend, wie Spezifikationen zusammenhängen, wie sie auf Dokumente aufgeteilt und wie sie verteilt werden können.

Je mehr einzelne Komponenten voneinander entkoppelt sind, desto weniger Kontextbedingungen müssen bei einer Komposition berücksichtigt werden und desto weitreichender gestalten sich deren potentielle Einsatzmöglichkeiten. Andererseits vereinfachen Referenzen auf standardisierte Definitionen die Integrationsbemühungen, insofern sie in allen beteiligten Komponenten Verwendung finden bzw. zu diesen kompatibel sind. Die Minimierung von Abhängigkeiten ist letztendlich immer nur begrenzt möglich, da der Verweis auf unabhängige, zentrale Definitionen die einzige Möglichkeit darstellt, Komponenten zueinander und zu einem Einsatzkontext in Beziehung zu setzen. Objektive Bewertung erfordert ein unabhängiges Referenzsystem. Darum gilt es, bei der Entwicklung von Komponenten Verweise auf nicht standardisierte Definitionen sowie gegenseitige Abhängigkeiten zu minimieren. Wo die Grenze zwischen lokalen Definitionen auf der einen Seite und referenzierten Definitionen auf der anderen Seite verläuft, hängt von der konkreten Anwendung ab, bspw. von der Frage, ob für das Anwendungsgebiet standardisierte Ontologien existieren. Im Allgemeinen benötigt man daher Technologien, die es erlauben, je nach Bedarf zwischen dezentralen und zentralen Definitionen wählen zu können.

Diesem Anspruch genügen modular-hierarchische Modellierungsformalismen bisher nicht. Modellierungsformalismen abstrahieren von der Verteilung einzelner Modellteile und fassen alle Definitionen als lokal verfügbar auf. Damit sind sie vor allem auf eine Entwicklung von Modellen zugeschnitten, in der alle Typdefinitionen unter einer zentralen Kontrolle liegen. Die praktisch verfügbaren syntaktischen Mittel zur Definition von Modellen heben diese Beschränkung teilweise durch die Verwendung von Programmiersprachen auf (Zeigler u. Sarjoughian 2005; Brooks u. a. 2007). So können Datentypen in Java als separate Definitionseinheiten mit einer eindeutigen Kombination aus Paket- und Klassennamen versehen, in JAR-Paketen verfügbar gemacht und damit von unterschiedlichen Modellimplementierungen referenziert werden.

In der Softwarewelt haben sich fortschrittlichere Konzepte durchgesetzt. Grundlage für Schnittstellendefinitionen von Softwarekomponenten und Web Services sind Mittel, um Datentypen zu definieren, ob direkt wie in IDLs oder indirekt wie mit XML-Schemasprachen im Kontext von

WSDL. Bei Softwarekomponenten wie auch Web Services haben sich global eindeutige Namen etabliert, um Komponenten mit Identität zu versehen und transparent auf sie zugreifen zu können. Analog soll für das Metamodell der Komponentenplattform die Modularisierung von Definitionen explizit vorgesehen werden, so dass Datentypen referenzierbar und verteilbar werden. Dementsprechend bilden nicht einfache Mengen die Grundlage des Metamodells sondern referenzierbare Typen.

Definition 4.1. Ein **Typ** ist ein Paar $\mathscr{T} = (id, car)$, mit einem qualifizierten Namen $id \in QName$ und einer Menge von Werten $car \subseteq UVal$, genannt Trägermenge. Für den Zugriff auf die Elemente von $\mathscr{T}$ lassen sich $id_{\mathscr{T}}$ bzw. $car_{\mathscr{T}}$ nutzen.

Dabei wird eine Menge von qualifizierten Bezeichnern $QName \subseteq String$ sowie ein Universum möglicher Werte $UVal$ als gegeben angenommen, wobei $String$ die Menge aller Zeichenketten repräsentiert. Ein qualifizierter Name $qname \in QName$ kann bspw. als Kombination aus Namensraum und lokalem Namen, getrennt durch ':', repräsentiert werden.

Beispiel 4.1. *Im Beispiel 2.1 wurde die Eingabemenge $X_{call} = Call$ definiert. Diese lässt sich als Trägermenge des Typs $CallType$ = („unihro/diane/base/service:Call“, Call) nutzen, so dass dort, wo die Menge Call benötigt wird, sie mittels „unihro/diane/base/service:Call“ referenziert werden kann.*

Typdefinitionen lassen sich erst sinnvoll nutzen, wenn eindeutig zwischen ihnen anhand qualifizierter Namen unterschieden werden kann. Im Folgenden wird $Type$ als Menge von Typen genutzt, so dass für alle $\mathscr{T}, \mathscr{T}' \in Type$ aus $id_{\mathscr{T}} = id_{\mathscr{T}'}$ folgt, dass $\mathscr{T} = \mathscr{T}'$.

Um Typdefinitionen für Modellkomponenten verwenden zu können, müssen Typen zueinander in Beziehung gesetzt werden können. In der Typtheorie formalisieren Untertypen das Prinzip der sicheren Substitution (Pierce 2002). Sichere Substitution drückt aus, dass überall dort, wo ein Element von einem Typ $\mathscr{T}$ genutzt werden kann, auch ein Untertyp von $\mathscr{T}$ verwendbar ist. Im Folgenden wird die Relation $S \subseteq Type \times Type$ genutzt, die für zwei Typen $\mathscr{T}, \mathscr{T}' \in Type$ angibt, ob die Trägermenge des ersten Typs eine Teilmenge der Trägermenge des zweiten Typs ist:

$$(\mathscr{T}, \mathscr{T}') \in S \Leftrightarrow car_{\mathscr{T}} \subseteq car_{\mathscr{T}'}.$$

Falls $(\mathscr{T}, \mathscr{T}') \in S$ schreiben wir auch $\mathscr{T} \sqsubseteq \mathscr{T}'$ und sagen $\mathscr{T}$ ist ein Untertyp von $\mathscr{T}'$.

Der Umgang mit Typen lässt sich durch eine weitere Hilfsfunktion vereinfachen. Gegeben $Type$ als Menge aller Typdefinitionen sowie ein spezielles Symbol $\perp$, das für „undefiniert“ steht, dereferenziert die Funktion $drefT : QName \to Type \cup \{\perp\}$ qualifizierte Namen zu Typdefinitionen, so dass

$$drefT(qname) \stackrel{def}{=} \begin{cases} \mathscr{T} & \text{falls } \exists \mathscr{T} \in Type.id_{\mathscr{T}} = qname \\ \perp & \text{sonst.} \end{cases}$$

Im Folgenden wird davon ausgegangen, dass die syntaktische, semantische, pragmatische und konzeptuelle Dimension von Daten in den Definitionen von Trägermengen enthalten ist und es wird vorerst davon abstrahiert, wie Trägermengen konkret definiert werden. Dieser Frage wird im Zuge der Definition einer konkreten Syntax in Abschnitt 4.4 nachgegangen.

4.1.2 Ereignisports und Rollen

Typen werden jetzt verwendet, um zu definieren, welche Arten von Nachrichten zwischen Modellen ausgetauscht werden dürfen. In modular-hierarchischen Ansätzen, wie Modelica, DEVS und Ptolemy, stellen Modelle Ereignisports aus, um anzuzeigen, dass Ereignisse eines bestimmten Typs empfangen bzw. gesendet werden können. Ein Ereignisport wird hier nicht auf Basis lokaler Mengendefinitionen formalisiert, sondern mittels Referenzen auf Typdefinitionen.

Abbildung 4.2: Eine Rolle mit zwei gerichteten Ereignisports

Definition 4.2. Ein **Ereignisport** ist ein Tupel $e = (name, tid, inp)$, mit einem Namen $name \in String$, einem Verweis auf eine Typdefinition $tid \in QName$ und einer Richtungsangabe $inp \in \mathbb{B}$. Für den Zugriff auf die Elemente von e werden $name_e$, tid_e bzw. inp_e genutzt.

Dabei ist $\mathbb{B} = \{true, false\}$. Falls $inp_e = true$ wird e als Eingabeport bezeichnet und falls $inp_e = false$ als Ausgabeport.

Im vorigen Kapitel wurde festgestellt, dass es sich im Bereich der Software durchgesetzt hat, logisch zusammengehörende atomare Interaktionspunkte in Abstraktionseinheiten zu strukturieren. Analog zur Gruppierung von Methoden in Schnittstellendefinitionen sollten auch Ereignisports so zusammengefasst werden können, dass sie einen Teil der Interaktionsmöglichkeiten einer Modellimplementierung deklarieren. In der Modellierung und Simulation findet diese Sichtweise bisher kaum Anwendung. Ausnahmen bilden die zusammengesetzten *flowports* von SysML und das Konzept dynamischer Rollenbeschreibungen für DEVS (Yilmaz 2004b).

Während die Definition von Operationen in WSDL-Definitionen vollständig enthalten sind, erlauben strukturierte Klassen der UML 2.x einen flexibleren Umgang mit abstrakten Interaktionspunkten. Eine strukturierte Klasse kann mehrere abstrakte Interaktionspunkte, genannt Ports, ausstellen, die jeweils auf Rollendefinitionen[1] verweisen. Rollentypen können so unabhängig von referenzierenden Klassen definiert und damit auch für unterschiedliche Klassen verwendet werden.

Rollen extrahieren die Schnittstelleninformationen bezüglich einer bestimmten Abstraktion aus der Modelldefinition. Sie deklarieren eine Menge von gerichteten Ereignisports, die in einem logischen Zusammenhang stehen.

Definition 4.3. Eine **Rolle** ist ein Paar $\mathscr{R} = (id, EP)$ mit einem qualifizierten Namen $id \in QName$ und einer endlichen Menge von Ereignisports EP, so dass $\forall e, e' \in EP$ gilt

1. Eindeutige Namen: $name_e = name_{e'} \Rightarrow e = e'$
2. Korrekte Typreferenzen: $drefT(tid_e) \neq \bot$

Die Elemente von $\mathscr{R}$ werden auch als $id_{\mathscr{R}}$ und $EP_{\mathscr{R}}$ bezeichnet.

Analog zur Definition der Menge *Type* wird *Role* als Menge für Rollen mit eindeutigen Ids genutzt, so dass für zwei Rollen $\mathscr{R}, \mathscr{R}' \in Role$ aus $id_{\mathscr{R}} = id_{\mathscr{R}'}$ folgt, dass $\mathscr{R} = \mathscr{R}'$. Die Funktion $drefR(qname)$ liefert analog zu $drefT$ eine Rolle $\mathscr{R}$ mit $id_{\mathscr{R}} = qname$ bzw. $\bot$, falls eine Rolle mit dem angegebenem Identifizierer nicht existiert.

Beispiel 4.2. *In Abbildung 4.2 ist eine Rolle dargestellt, die das Interaktionspotential eines Protokolls zur Dienstvermittlung in Hinblick auf die Kommunikation mit einem Nutzermodell definiert. Die Rolle deklariert einen Eingabeport zum Empfang von Calls und einen Ausgabeport zum Verschicken von Responses. Die formale Definition lautet $\mathscr{R} = (id, EP)$, mit*

- *id = „unihro/diane/base/service:ServiceProv“ und*
- *$EP = \{$(„call“, „unihro/diane/base/service:Call“, $true$),*
  („response“, „unihro/diane/base/service:Response“, $false$).$\}$

[1] In UML werden diese Rollendefinitionen als Schnittstellen bezeichnet.

Voraussetzung für die Korrektheit von $\mathscr{R}$ ist, dass Typdefinitionen für „Call“ und „Response“ im Namensraum „unihro/diane/base/service“ existieren und zugreifbar sind. In der graphischen Repräsentation ist der Namensraum aus Gründen der Übersichtlichkeit nicht aufgeführt, sondern nur der lokale Name des Datentyps angegeben.

4.1.3 Schnittstellen

Methodenorientierte Schnittstellendefinitionen betonen den Unterschied zwischen aufrufendem und aufgerufenem System. Aus der Art, wie eine Softwarekomponente eine Schnittstelle referenziert, lässt sich ableiten, ob die Schnittstelle eine Anforderung oder ein Angebot ausdrückt. Der Initiator einer Interaktion stellt Bedingungen an seinen Einsatzkontext. Eine Softwarekomponente, die Methoden einer anderen Komponente aufrufen möchte, gibt dies in Form einer *receptacle* an, die anzeigt, dass von einem Interaktionspartner eine Schnittstelle gleichen Typs als *facet* erwartet wird. Diese Kontextbedingungen müssen zum Zeitpunkt des Einsatzes durch Angebote komplementiert werden.

In einer ersten Adaption von UML-Komponentendiagrammen auf Modellkomponenten wurden Rollenbeschreibungen auch für Modellkomponenten mittels *receptacle* und *facet* ausgestellt (Röhl 2006). Als *receptacle* wurde dort der potentielle Initiator einer Kommunikation gewählt. Für ereignisorientierte Rollen lässt sich jedoch nicht immer ein eindeutiger Richtungssinn angeben. Der Datenaustausch von ereignisorientierten Modellen findet nicht über den Aufruf von Methoden statt, die Rückgabewerte liefern, sondern über den Austausch separater Ereignisse. Rollen können zudem Eingabeports *und* Ausgabeports deklarieren, wobei das Senden eines Ereignisses nicht notwendigerweise dazu führt, dass ein Antwortereignis empfangen wird. Zum Beispiel erfordert das erfolgreiche Versenden von Nachrichten in Netzwerken, dass potentielle Interaktionspartner Ereignisports zum Empfang von Nachrichten besitzen. Die Teilnehmer in einem Netzwerk nehmen für sich wechselseitig komplementäre Rollen als Sender und Empfänger ein. Wer von den Netzwerknoten eine Interaktion initiiert, ist nicht allein aus der Verbindungsstruktur ableitbar. Um *receptacle* und *facet* für diskret-ereignisorientierte Schnittstellen eindeutig zu nutzen, bliebe nur, Ausgabeports als Anforderungen und Eingabeports als Angebote atomar zu deklarieren (Zinoviev 2005). Ereignisports ließen sich dann nicht mehr in Rollen gruppieren.

In den Kompositionstrukturen von UML findet sich ein flexiblerer Ansatz Rollen auszustellen. Abstrakte Interaktionspunkte können dort mit Multiplizitäten versehen werden, die spezifizieren, wie oft sie mindestens verbunden werden müssen und höchstens verbunden werden dürfen. Multiplizitäten ermöglichen damit, Rollen auch wechselseitig als Anforderung auszustellen, indem abstrakte Interaktionspunkte auf beiden Seiten mit einer minimalen Multiplizität ≥ 1 versehen werden.

Multiplizitäten sind für Modellkomponenten aus einem weiteren Grund wünschenswert. Gewöhnlich finden sich in Simulationsmodellen einzelne Modelle, die starke Abstraktionen enthalten und mit einer Vielzahl anderer Modelle verkoppelt sind. Ein typisches Beispiel sind Modelle für räumliche Umgebungen. Das Umgebungsmodell fungiert dabei als zentrales Modell an das eine Menge anderer Modelle, jeweils mit dem gleichen Kopplungsschema, angeschlossen ist. Multiplizitäten können in diesem Fall die Wiederholung von Rollendefinitionen für einzelne Interaktionspartner ersparen.

Definition 4.4. Ein **Port** ist ein Tupel $p = (name, rid, min, max)$, mit einem Bezeichner $name \in String$, einem Verweis auf eine Rolle $rid \in QName$ sowie einer minimalen und maximalen Anzahl von Verbindungsmöglichkeiten $min \in \mathbb{N}^+$ und $max \in \mathbb{N}^+ \cup \{\infty\}$. Für p muss gelten

1. Existenz der Rolle: $drefR(rid) \neq \bot$
2. Geordnete Multiplizitäten: $min \leq max$

Für den Zugriff auf die Elemente eines Ports p werden $name_p$, rid_p, min_p und max_p genutzt.

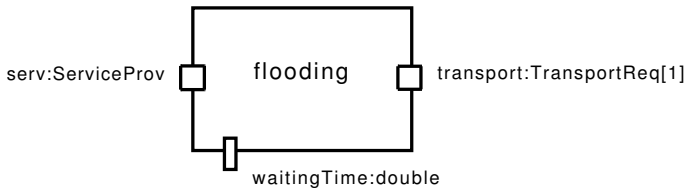

Abbildung 4.3: Schnittstelle für das Modell *Flooding*

Ports referenzieren Rollen, die ihrerseits eine Menge von Ereignisports umfassen. In Hinblick auf die Komposition definieren Ports die Angebote und Anforderungen eines Modells für bzw. an andere Modelle. Für eine Komposition lassen sich Ports nutzen, um über die Kompatibilität von Modellen zu befinden. Ports enthalten jedoch keine Information über die Eignung eines Modells für eine bestimmte Fragestellung.

Ein Grundsatz der Modellbildung lautet: andere Fragestellung, anderes Modell. Vollständige Wiederverwendung von Modellen ist nur möglich, wenn dem Einsatzkontext einer Komponente bei ihrer Erstellung bewusst oder unbewusst entsprochen wurde. Dies engt den Wiederverwendungswert von Modellen stark ein. Höhere Grade an Wiederverwendung eröffnet die Parametrisierung von Modellkomponenten, wie sie bspw. Modelica und Ptolemy anbieten. Ein Modell wird damit nicht für eine einzige, ganz bestimmte Fragestellung entwickelt, sondern als Klasse von Modellen, die für eine Menge ähnlicher Fragestellungen potentiell verwendbar ist.

Definition 4.5. Ein **Parameter** ist ein 3-Tupel $p = (name, tid, value)$, mit einem Namen $name \in String$, einem Verweis auf einen Datentypen $tid \in QName$ und einem Wert $value \in UVal$. Für $\mathscr{T} = drefT(tid)$ muss gelten

1. Existente Typdefinition: $\mathscr{T} \neq \bot$
2. Korrekter Wertebereich: $value \in car_{\mathscr{T}}$

Basierend auf Ports und Parametern lassen sich nun Schnittstellen für Modellkomponenten definieren.

Definition 4.6. Eine **Schnittstelle** ist ein Tupel $\mathscr{I} = (id, impl, Params, Ports)$, mit einem Identifizierer $id \in QName$, einem Verweis auf eine Implementierung $impl \in QName$, einer endlichen Menge von Parametern $Params$ und einer endlichen Menge von Ports $Ports$, so dass gilt

1. Eindeutige Portnamen: $\forall p, p' \in Ports.name_p = name_{p'} \Rightarrow p = p'$
2. Eindeutige Parameternamen: $\forall p, p' \in Params.name_p = name_{p'} \Rightarrow p = p'$

Für den Zugriff auf die Elemente einer Schnittstelle $\mathscr{I}$ werden $id_{\mathscr{I}}$, $impl_{\mathscr{I}}$, $Params_{\mathscr{I}}$ bzw. $Ports_{\mathscr{I}}$ genutzt.

Schnittstellendefinitionen binden jeweils genau eine konkrete Implementierung[2] an sich. Wird bei der Suche nach Kompositionskandidaten eine Schnittstelle als passend identifiziert, lässt sich mittels der Referenz *impl* eine Implementierung für *diese* Schnittstelle beschaffen.

Analog zu Typen und Rollen repräsentiert *Iface* die Menge aller Schnittstellendefinitionen, so dass alle $\mathscr{I} \in Iface$ über eindeutige Identifizierer verfügen. Des Weiteren sei $drefI(qname) = \mathscr{I}$, falls $\exists \mathscr{I} \in Iface.id_{\mathscr{I}} = qname$ bzw. $\bot$, falls nicht.

[2] siehe die Definition von Komponenten in Abschnitt 4.1.6

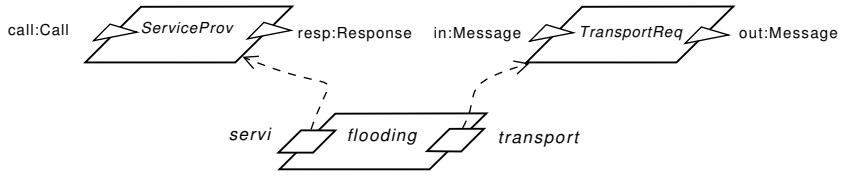

Abbildung 4.4: Ports von Schnittstellen referenzieren Rollen, die Ereignisports enthalten

Beispiel 4.3. *Die Rolle* `ServiceProv` *aus Beispiel 4.2 dient nun zur Definition einer Schnittstelle für ein Flooding-Modell. Die Schnittstelle stellt zwei Ports aus und bietet einen Parameter zur Konfiguration der Wartezeit bei Dienstsuchen an. Formal lässt sich die Schnittstelle definieren als* $\mathscr{I} = (id, impl, Params, Ports)$*, mit*

- $id =$ *„unihro/diane/com/flooding:interface“*
- $impl =$ *„unihro/diane/com/flooding:implementation“*
- $Params = \{(\text{„}waitingTime\text{“}, \text{„}double\text{“}, 1.0)\}$
- $Ports = \{(\text{„}serv\text{“}, \text{„}unihro/diane/base/service:ServiceProv\text{“}, 0, 1),$
  $(\text{„}transport\text{“}, \text{„}unihro/diane/base/transport:TransportReq\text{“}, 1, 1)\}$

Während der Port „transport“ mittels einer minimalen Multiplizität von eins eine Kontextanforderung ausdrückt, muss der Port „serv“ nicht notwendigerweise verbunden werden. Warum dies für ein Flooding-Modell sinnvoll ist, erläutert Beispiel 4.10.

Abbildung 4.3 visualisiert die Schnittstelle in Anlehnung an die Repräsentation von strukturierten Klassen mit UML. Aus Gründen der Übersichtlichkeit werden in der visuellen Repräsentation die Namensräume nicht explizit angeben. Der Doppelpunkt trennt den Portnamen vom Typnamen. In Abbildungen können Multiplizitäten in eckigen Klammern angegeben werden. Falls nicht angegeben, wird $min = 0$ *und* $max = 1$ *angenommen.*

Schnittstellen fassen, ähnlich zu strukturierten Klassen der UML, eine Menge von Ports zusammen, die abstrakte Interaktionspunkte für eine Komponente deklarieren. Darüber hinaus geben Schnittstellen Parameter bekannt, über die Komponenten für konkrete Einsatzkontexte konfiguriert werden können. Rollen typisieren abstrakte Interaktionspunkte und fassen, eine Menge gerichteter Ereignisports zusammen. Abbildung 4.4 verdeutlicht den Zusammenhang zwischen Schnittstellen, Rollen und Ereignisports. Schnittstellen enthalten Ports, die Rollen referenzieren und durch diese getypt werden. Eine Rolle enthält wiederum eine Menge von Ereignisports, die auf Typdefinitionen verweisen.

4.1.4 Bindungen

Öffentliche Schnittstellenbeschreibungen fungieren als Verträge zwischen Modellkomponenten. In Schnittstellen gegebene Versprechen müssen sich letztendlich in entsprechenden Modellimplementierungen wiederfinden. Im Rahmen der Kompositionsplattform müssen Modelle so definierbar sein, dass sie mit ihrer Umgebung allein über gerichtete Ereignisports interagieren und keine direkten Abhängigkeiten zu anderen Modelle aufweisen. Modular-hierarchische Modellformalismen, wie DEVS, Modelica, SysML oder auch die akteursbasierten Modellierungsdomänen von Ptolemy sind dafür prinzipiell geeignet.

Analog zu den Element *Binding* in WSDL müssen deklarierte Ereignisports mit konkreten Modellports assoziiert werden. Eine Schwierigkeit ergibt sich daraus, dass Schnittstellenports über

Multiplizitäten verfügen, die anzeigen, wie oft ein Port Verbindungen eingehen kann. Mehrfache Verbindungen eines Schnittstellenports bringen Konflikte in der Bindung von abstrakten an konkrete Ereignisports mit sich. Während in UML-Kompositionsdiagrammen die Semantik von Mehrfachverbindungen als Variationspunkt offen gelassen wird, sollen deklarierte Ereignisports eindeutig Modellports zugeordnet werden können.

Definition 4.7. Eine **Bindung** ist eine endliche Tupelmenge $\mathscr{B} = \{(port, decl, pos, impl)\}$, mit jeweils einem Portnamen $port \in String$, dem Namen des in einer Rolle deklarierten Ereignisports $decl \in String$, einer Positionsangabe $pos \in \mathbb{N}^+$ und dem Namen des zugewiesenen Modellports $impl \in String$. Für alle $b, b' \in \mathscr{B}$ muss gelten, dass $b = b'$ genau dann, wenn $port_b = port_{b'} \wedge decl_b = decl_{b'} \wedge pos_b = pos_{b'}$.

Mittels Bindungen lassen sich deklarierte Ereignisports konkreten Implementierungen zuordnen. Da Schnittstellenports über Multiplizitäten verfügen, ist es für eindeutige Bindungen erforderlich, die Position innerhalb der Multiplizitäten anzugeben.

Die Beziehung zwischen deklarierten und implementierenden Ereignisports wird nun durch eine Hilfsfunktion definiert. Die Funktion *impl* liefert für den Namen eines Schnittstellenports pn, den Namen eines Ereignisports epn, eine Position pos und eine Bindung $\mathscr{B}$ den Namen des implementierenden Ports, so dass

$$impl(pn, epn, pos, \mathscr{B}) \stackrel{def}{=} \begin{cases} impl_b & \text{falls } \exists b \in \mathscr{B}.port_b = pn \wedge decl_b = epn \wedge pos_b = pos \\ epn & \text{sonst.} \end{cases}$$

Falls für die Kombination aus Portnamen und Ereignisportnamen eine Bindung existiert, gibt *impl* den entsprechenden Namen des implementierenden Ports zurück. Existiert keine Bindung, wird der Name des deklarierten Ereignisports genutzt. Dadurch ist sichergestellt, dass *impl* für sämtliche deklarierten Positionen definiert ist, insbesondere für ∞.

Beispiel 4.4. *Eine Modellimplementierung für Fluten wird nun an die Schnittstelle $\mathscr{I}$ aus Beispiel 4.3 gebunden. Das Modell besitzt die Eingabeports „call“ und „msgIn“ sowie die Ausgabeports „response“ und „msgOut“. Für die Schnittstelle Flooding ergibt sich die Bindung*

$$\mathscr{B} = \{(„transport“, „in“, 1, „msgIn“), („transport“, „out“, 1, „msgOut“)\}$$

So ist z.B. $impl(„transport“, „in“, 1, \mathscr{B}) = „msgIn“$ und $impl(„serv“, „call“, 1, \mathscr{B}) = „call“$. Für den Port „call“ bedarf es keiner expliziten Bindung, da der deklarierte Name und der Name des implementierenden Ports zusammenfallen. Den Zusammenhang zwischen den abstrakten Interaktionspunkten und den konkreten Modellports veranschaulicht Abbildung 4.5. Den Ereignisports von Rollen werden über Bindungen Ereignisports von Modellimplementierungen zugeordnet. Die vier dickgestrichelten Linien repräsentieren die Zuordnung.

4.1.5 Kompositionen

Eine Schnittstellendefinition beinhaltet keine Implementierungsdetails. Schnittstellen können als eigenständige Definitionen in öffentlichen Datenbanken hinterlegt und die Analyse von Kompositionskandidaten allein auf ihrer Basis durchgeführt werden. In Kompositionen sollen Komponenten dann über ihre Schnittstellen genutzt werden. Dazu bedarf es eines qualifizierten Namens und, entsprechend der Anforderungen eines konkreten Einsatzkontextes, einer Menge von Parameterwerten.

Definition 4.8. Eine **Schnittstellenreferenz** ist ein Tupel $iref = (name, iid, Params)$ mit einem Namen $name \in String$, einem Verweis auf eine Schnittstellendefinition $iid \in QName$ sowie einer endlichen Menge von Parametern $Params$. Für $\mathscr{I} = drefI(iid)$ muss gelten

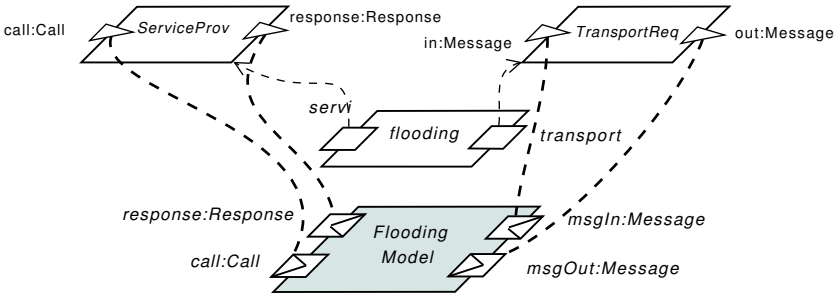

Abbildung 4.5: Bindung von deklarierten Ereignisports an Ports von Modellimplementierungen

1. Existente Schnittstelle: $\mathscr{I} \neq \bot$
2. Eindeutige Parameternamen: $\forall p, p' \in Params.name_p = name_{p'} \Rightarrow p = p'$
3. Existente und konsistente Parameter:

$$\forall p \in Params. \exists p' \in Params_{\mathscr{I}}.name_p = name_{p'} \wedge tid_p \sqsubseteq tid_{p'}$$

Von Schnittstellenreferenzen gelangt man zu Kompositionen, indem Verbindungen zwischen referenzierten Schnittstellen hergestellt werden. Verbindungen zwischen Schnittstellen werden mittels Konnektoren ausgedrückt, die auf Positionen innerhalb der Multiplizitäten von Schnittstellenports verweisen.

Definition 4.9. Ein **Konnektor** ist ein Tupel $k = (if, port, pos)$, mit dem Namen einer Schnittstelle $if \in String$, dem Namen eines Ports $port \in String$ und einer Position $pos \in \mathbb{N}^+$.

Ein Konnektor verweist demnach auf eine bestimmte Portposition einer Schnittstelle. Beziehungen zwischen Schnittstellen lassen sich so über die Paarung von jeweils zwei Konnektoren ausdrücken.

Definition 4.10. Eine **Kompositionsverbindung** ist ein Paar $c = (start, end)$, wobei *start* und *end* jeweils Konnektoren sind, die den Anfangspunkt bzw. den Endpunkt einer Verbindung darstellen. Für c muss gelten, dass $if_{start} \neq if_{end}$.

Eigenschaften von Kompositionen sollen sich allein auf Grundlage der Eigenschaftsbeschreibungen der Teile sowie den Regeln des Zusammensetzens bestimmen lassen. Die Eigenschaften der Teile werden in Schnittstellen beschrieben und die Regeln des Zusammensetzens werden durch Kompositionsverbindungen ausgedrückt.

Definition 4.11. Eine **Komposition** ist ein Paar (Sub, Con), mit einer endlichen Menge von Schnitttstellenreferenzen *Sub* und einer endlichen Menge von Kompositionsverbindungen *Con*. Es muss gelten

1. Eindeutige Namen in Schnittstellenreferenzen: $\forall s, s' \in Sub.name_s = name_{s'} \Rightarrow s = s'$

Analog zu modular-hierarchischen Modellierungsformalismen sowie externer und interner Blockdarstellung in SysML ist für Modellkomponenten die Verfeinerung von Schnittstellen in Kompositionsstrukturen sinnvoll, so dass Modelle auf Basis von Schnittstellen hierarchisch konstruiert werden können. Eine Komposition sollte selbst Ports ausstellen können, über die sich Beziehungen zu den Ports der komponierten Teile herstellen lassen.

Der spezielle Bezeichner „this“ kann in Kompositionen genutzt werden, um Teile einer Komposition hierarchisch zu verbinden. Damit sind, wie in UML, CCM und auch DEVS, Verbindungen

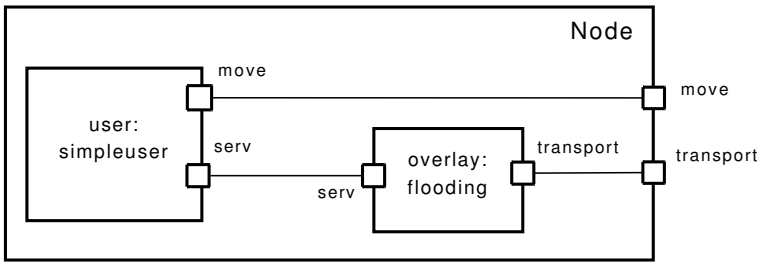

Abbildung 4.6: Verbindung von Schnittstellen über Ports

sowohl zwischen Systemen auf gleicher Ebene, als auch zwischen einem zusammengesetztem System und seinen Teilen möglich. In der Sprache von UML werden Verbindungen auf gleicher Ebene als assemblierende Verbindungen bezeichnet. Für $c = (start, end)$ ist

$$assembly(c) \Leftrightarrow name_{start} \neq \text{„this“} \wedge name_{end} \neq \text{„this“}$$

Falls $assembly(c)$ nicht gilt, wird eine Verbindung als delegierend bezeichnet.

Beispiel 4.5. *Abbildung 4.6 zeigt, wie im Rahmen eines Netzwerkknotens, die Schnittstellenreferenzen für einen einfachen Nutzer und für ein Protokoll zur Dienstvermittlung hierarchisch komponiert und verbunden werden. Die Komposition selbst ist wieder als Schnittstelle interpretierbar und delegiert die Ports move und transport.*

$$\begin{aligned} Sub &= \{ \quad (\text{„}this\text{“}, \text{„}unihro/diane/com/node{:}interface\text{“}, \emptyset), \\ &\qquad (\text{„}user\text{“}, \text{„}unihro/diane/com/simpleuser{:}interface\text{“}, \emptyset), \\ &\qquad (\text{„}overlay\text{“}, \text{„}unihro/diane/com/flooding{:}interface\text{“}, \emptyset) \ \} \\ Con &= \{ \quad ((\text{„}user\text{“}, \text{„}serv\text{“}, 1), (\text{„}overlay\text{“}, \text{„}serv\text{“}, 1)), \\ &\qquad ((\text{„}user\text{“}, \text{„}move\text{“}, 1), (\text{„}this\text{“}, \text{„}move\text{“}, 1)), \\ &\qquad ((\text{„}overlay\text{“}, \text{„}transport\text{“}, 1), (\text{„}this\text{“}, \text{„}transport\text{“}, 1)) \ \} \end{aligned}$$

Die erste Kompositionsverbindung ist assemblierend, da $name_{start} \neq$ „this“ und $name_{end} \neq$ „this“. Die zweite und die dritte Kompositionsverbindung sind delegierend, da $name_{end}$ in beiden Fällen gleich „this“ ist.

4.1.6 Komponenten

Schnittstellen deklarieren neben Ports eine Menge von Parametern. Um eine Komponente als Implementierung einer Schnittstelle definieren zu können, müssen konkrete Parametrisierungen ausgewertet werden. Zur Definition von Komponenten bedarf es eines Mechanismuses, der die Interna einer Komponente entsprechend einer Menge von Parametern verändert. Diese Aufgabe übernimmt für eine Komponente ein Konfigurator.

Definition 4.12. Ein **Konfigurator** ist eine Funktion, die ein Tupel $(Params, Sub, Con, \mathscr{M}, \mathscr{B})$ auf ein Tupel $(Sub', Con', \mathscr{B}', \mathscr{M}')$ abbildet. $Params$ ist eine Menge von Parametern. (Sub, Con) und (Sub', Con') sind jeweils eine Komposition. $\mathscr{M}$ und $\mathscr{M}'$ sind Modelldefinitionen. $\mathscr{B}$ und $\mathscr{B}'$ sind Bindungen.

Komponenten lassen sich nun mittels Konfiguratoren kompakt repräsentieren.

Definition 4.13. Eine **Komponente** ist ein Tupel $\mathscr{C} = (id, if, \kappa, \mathscr{M}, \mathscr{B}, Sub, Con)$, mit einem Bezeichner $id \in QName$, einem Verweis auf eine Schnittstelle $if \in QName$, einem Konfigurator κ,

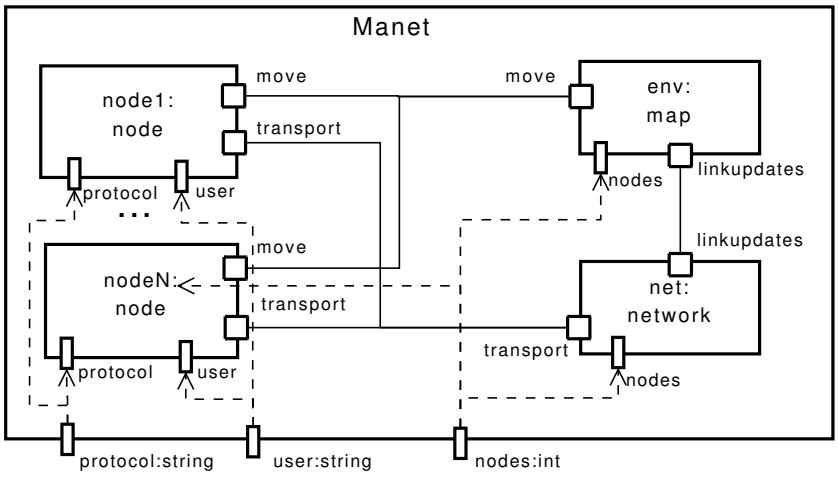

Abbildung 4.7: Einfluss von Parametern auf die Komponente Manet

einem Modell $\mathscr{M}$ und einer Menge von Bindungen $\mathscr{B}$. (Sub, Con) ist eine Komposition. Es müssen folgende Bedingungen gelten:

1. Existente Schnittstelle: $drefI(if) \neq \bot$
2. Konsistenter Verweis in der Schnittstelle: $id = impl_{\mathscr{I}}$, für $\mathscr{I} = drefI(if)$

Falls $Sub_{\mathscr{C}} = \emptyset$ ist, wird $\mathscr{C}$ als atomare Komponente bezeichnet und falls $Sub_{\mathscr{C}} \neq \emptyset$, als zusammengesetzte Komponente.

Komponenten sollen allein über parametrisierte Schnittstellenreferenzen genutzt werden. Für eine Menge von Komponenten Com wird dementsprechend keine Dereferenzierungsfunktion für qualifizierte Namen von Komponenten definiert, sondern eine Instantiierungsfunktion. Eine Komponente wird instantiiert, indem der Konfigurator der Komponente auf eine Menge von Parametern entsprechend einer Schnittstellenreferenz $iref = (name, iid, Params)$ angewendet wird, so dass

$$instC(iref) \stackrel{def}{=} \begin{cases} (id_{\mathscr{C}}, if_{\mathscr{C}}, \bot, \mathscr{M}', \mathscr{B}', Sub', Con') & \text{falls } \exists \mathscr{C} \in Com.id_{\mathscr{C}} = impl_{\mathscr{I}} \\ \bot & \text{sonst,} \end{cases}$$

wobei $(Sub', Con', \mathscr{M}', \mathscr{B}') = \kappa_{\mathscr{C}}(Params_{iref}, Sub_{\mathscr{C}}, Con_{\mathscr{C}}, \mathscr{M}_{\mathscr{C}}, \mathscr{B}_{\mathscr{C}})$ und $\mathscr{I} = drefI(iid_{iref})$.

Komponenten umfassen demnach veränderliche und unveränderliche Elemente. Unveränderlich sind $id_{\mathscr{C}}$, $if_{\mathscr{C}}$ und $\kappa_{\mathscr{C}}$. Die Menge der Schnittstellenreferenzen, Kompositionsverbindungen und Bindungen sowie die Modellimplementierung können während der Instantiierung durch den Konfigurator einer Komponente ($\kappa_{\mathscr{C}}$) verändert werden.

Beispiel 4.6. *Abbildung 4.7 veranschaulicht die Parameterabbildung von Komponenten anhand der zusammengesetzten Komponente Manet. Der Einfluss von Parametern ist in der Abbildung durch gestrichelte Linien visualisiert. Durchgehende Linien zeigen Kompositionsverbindungen.*

Fest enthalten in $Manet = (id, if, \kappa, \mathscr{M}, \mathscr{B}, Sub, Con)$ *sind zwei Unterkomponenten, so dass* $Sub = \{s_1, s_2\}$*, mit*

$$\begin{aligned} s_1 &= (\text{„}env\text{“}, \text{„}unihro/diane/com/map{:}interface\text{“}, \emptyset) \\ s_2 &= (\text{„}net\text{“}, \text{„}unihro/diane/com/network{:}interface\text{“}, \emptyset) \end{aligned}$$

Die beiden Unterkomponenten sind mittels einer Kompositionsverbindung verbunden, so dass $Con = \{c_1\}$*, mit*

$$c1 = ((\text{„}env\text{“}, \text{„}linkupdates\text{“}, 1), (\text{„}net\text{“}, \text{„}linkupdates\text{“}, 1))$$

Die Schnittstelle $\mathscr{I}$ *von Manet stellt drei Parameter aus — die Anzahl von Netzwerknoten, den Typ des in jedem Knoten zu verwendenden Nutzers und den Typ des zu nutzenden Protokolls zur Dienstvermittlung.*

Angenommen, die Komponente Manet soll mit den Parametern $Params = \{p_1, p_2, p_3\}$ *instantiert werden, wobei*

$$\begin{aligned} p_1 &= (\text{„}nodes\text{“}, \text{„}int\text{“}, 100) \\ p_2 &= (\text{„}user\text{“}, \text{„}string\text{“}, \text{„}unihro/diane/com/simpleuser:interface\text{“}) \\ p_3 &= (\text{„}protocol\text{“}, \text{„}string\text{“}, \text{„}unihro/diane/com/flooding:interface\text{“}) \end{aligned}$$

Die Instantiierung der Komponente Manet über eine parametrisierte Schnittstellenreferenz liefert $instC(\text{„}manet\text{“}, id_{\mathscr{I}}, \{p_1, p_2, p_3\}) = (id, if, \bot, \mathscr{M}', \mathscr{B}', Sub', Con')$*. Im Zuge der Instantiierung delegiert* $\kappa(\{p_1, p_2, p_3\})$ *den Parameter* p_1 *an die Schnittstellenreferenzen für „env“ und „net“ und fügt 100 Schnittstellenreferenzen für Netzwerknoten ein, so dass* $Sub' = \{s'_1, s'_2, s'_3, \ldots, s'_{102}\}$*, wobei*

$$\begin{aligned} s'_1 &= (\text{„}env\text{“}, \text{„}unihro/diane/com/map:interface\text{“}, \{p_1\}) \\ s'_2 &= (\text{„}net\text{“}, \text{„}unihro/diane/com/network:interface\text{“}, \{p_1\}) \\ s'_3 &= (\text{„}node1\text{“}, \text{„}unihro/diane/com/node:interface\text{“}, \{p_2, p_3\}) \\ &\ldots \\ s'_{102} &= (\text{„}node100\text{“}, \text{„}unihro/diane/com/node:interface\text{“}, \{p_2, p_3\}) \end{aligned}$$

Jedem Knoten wird ein individueller Name zugewiesen und als Parameter mitgegeben, welches Protokoll- und welches Nutzermodell eingesetzt werden soll. Die Kompositionsverbindungen werden erweitert zu $Con' = \{c_1, c'_2, \ldots, c'_{101}, c'_{102}, \ldots, c'_{201}\}$*, wobei*

$$\begin{aligned} c'_2 &= ((\text{„}node1\text{“}, \text{„}transport\text{“}, 1), (\text{„}net\text{“}, \text{„}transport\text{“}, 1)) \\ c'_{101} &= ((\text{„}node100\text{“}, \text{„}transport\text{“}, 1), (\text{„}net\text{“}, \text{„}transport\text{“}, 100)) \\ c'_{102} &= ((\text{„}node1\text{“}, \text{„}move\text{“}, 1), (\text{„}env\text{“}, \text{„}move\text{“}, 1)) \\ c'_{201} &= ((\text{„}node100\text{“}, \text{„}move\text{“}, 1), (\text{„}env\text{“}, \text{„}move\text{“}, 1)) \end{aligned}$$

Des Weiteren ist $\mathscr{B}' = \mathscr{B}$ *und* $\mathscr{M}' = \mathscr{M}$*.*

Damit stehen alle syntaktischen Mittel zur Definition von Schnittstellen, Komponenten und Kompositionen bereit. Komponenten können Subkomponenten über die qualifizierten Namen ihrer Schnittstellendefinitionen referenzieren, in Schnittstellenreferenzen konkrete Parameterwerte setzen und Subkomponenten entsprechend ihrer publizierten Ports verbinden. Namens- und Multiplizitätskonflikte zwischen deklarierten Ereignisports und tatsächlich vorhandenen Modellports lassen sich mittels Bindungen auflösen. Konfiguratoren verändern die Interna von Komponenten entsprechend einer Menge von Parametern.

In konkreten Einsatzkontexten kommt es darauf an, aus Kompositionen ausführbare Modelle für eine konkrete Fragestellung abzuleiten.

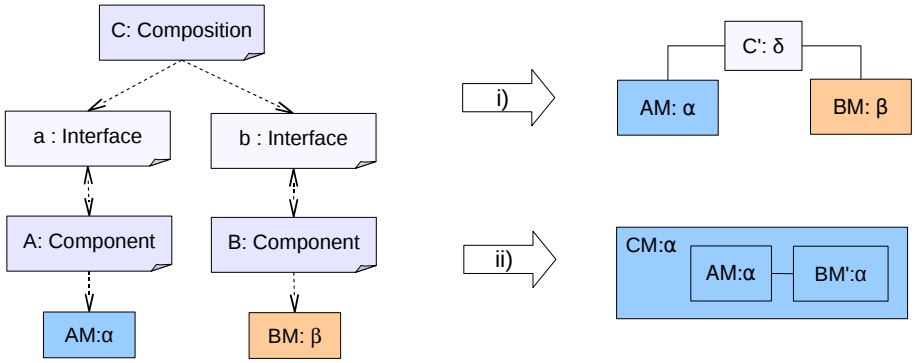

Abbildung 4.8: Möglichkeiten zur Abbildung einer Komposition auf Simulationsmodelle

4.1.7 Simulationsmodelle

Komponenten verbergen die Formalismen, in denen Modelle definiert sind. Atomare Komponenten beinhalten jeweils eine Modellimplementierung, die in einem bestimmten Modellierungsformalismus vorliegt. Für eine zusammengesetzte Komponente ist es möglich, dass ihre Unterkomponenten unterschiedliche Modellierungsformalismen nutzen. Enthalten Unterkomponenten tatsächlich heterogene Modellimplementierungen stellt sich die Frage, ob die genutzten Formalismen kompatibel sind. Zum Zeitpunkt der Ausführung eines Simulationsmodells muss Kompatibilität für die Teilmodelle auch auf technischer Ebene hergestellt werden.

Integrationsformalismus und Zielplattform

Heterogene Modellrepräsentationen lassen sich in Simulationsmodellen über zwei verschiedene Ansätze integrieren. Entsprechend Kapitel 2 besteht eine Möglichkeit darin, die Heterogenität von Teilmodellen beizubehalten. Zur Ausführung benötigt man dann Werkzeuge wie Ptolemy, die es erlauben, Modelle unterschiedlicher Formalismen kombiniert auszuführen. Werden nicht nur unterschiedliche Modellierungsformalismen verwendet, sondern auch verschiedene Simulationssysteme integriert, bedarf es Middleware wie HLA. Eine zweite prinzipielle Herangehensweise besteht in der Transformation der Teilmodelle in einen Superformalismus, bspw. eine Variante von DEVS, der die Ausdrucksmöglichkeiten aller in der Komposition vorkommenden Formalismen vereinigt (Sarjoughian 2006).

Abbildung 4.8 veranschaulicht die beiden Alternativen im Kontext der entwickelten Komponentenplattform. In der Komposition C werden das Modell AM, welches im Modellierungsformalismus α definiert ist, und das Modell BM, das im Formalismus β vorliegt, zusammengeführt. Variante i) belässt die Modelle in ihren Formalismen. Für die Variante ii) wird Modell BM vom Formalismus β in α transformiert. Mit beiden Varianten müssen die in C definierten Kompositionsverbindungen zwischen den Unterkomponenten als Verbindungskanäle zwischen den Teilmodellen im Simulationsmodell realisiert werden. Die erste Variante erfordert keine Transformation, steigert jedoch die Heterogenität im Zielmodell, da die Integration der Teilmodelle gewöhnlich in einer dritten Domäne stattfindet, bspw. mittels HLA. Mit der zweiten Variante lassen sich Kompositionsstrukturen als weiterer Modellierungsformalismus behandeln, der auf einen Superformalismus abgebildet werden kann.

Nachteilig für Variante ii) ist, dass Modelle in einer Repräsentation vorliegen müssen, die Transformation erlaubt. Modelle können jedoch nicht nur in unterschiedlichen Formalismen, sondern auch für unterschiedliche Simulationswerkzeuge definiert sein. Für einen Formalismus können Repräsentationen genutzt werden, die auf die Ausführung durch ein spezielles Werkzeug ausgerichtet sind. Modelle, die Annahmen über die Ausführungsplattform treffen, werden als plattformspezifi-

sche Modelle bezeichnet[3]. Im Sinne der MDA werden Modelle, die an kein konkretes Simulationssystem bzw. Programmiersprache gebunden sind, als plattformunabhängig bezeichnet.

Simulationsmodelle werden wiederum auf konkreten Plattformen (mit speziellen Simulationswerkzeugen) ausgeführt. In der Modellierung und Simulation sind sowohl modellorientierte (Integration in einem Werkzeug) als auch simulationsorientierte Integrationsansätze (Integration verschiedener Werkzeuge) theoretisch berechtigt und praktisch relevant. In der Zusammenfassung von Kapitel 3 wurde zudem festgestellt, dass die Einsatzmodelle von Softwarekomponenten, modellgetriebener Entwicklung und dienstorientierter Architekturen jeweils spezifische Vor- und Nachteile bieten und keines eindeutig für die Modellierung und Simulation zu bevorzugen ist. Im Rahmen der entwickelten Kompositionsplattform wird es daher vermieden, eine Integrationsweise und eine Zielplattform fest vorzugeben. Um ausführbare Simulationsmodelle aus Kompositionen ableiten zu können, soll sich stattdessen eine beliebige Kombination aus einem Formalismus, der als Integrationsdomäne fungiert, sowie einer Zielplattform angeben lassen.

Die erste Variante in Abbildung 4.8 ließe sich bspw. formulieren mittels OMT als Integrationsformalismus (In der Abbildung als δ bezeichnet) und RTI als Zielplattform. In der zweiten Variante ist α der Zielformalismus und ein beliebiges Simulationssystem, mittels dem sich α-Modelle ausführen lassen, eine gültige Zielplattform.

Konfiguration von Beobachtungsinstrumenten

Um ein ausführbares Simulationsmodell zu erhalten, genügt es, den ingenieurstechnischen Aspekt der Komponierbarkeit zu betrachten. Die Kommunikation zwischen Modellen lässt sich mit den entwickelten Beschreibungen bereits über Kompositionsverbindungen erfassen. Beispielsweise bedarf die Simulation eines Protokolls zur Dienstvermittlung in MANETs einer Reihe von Hilfsmodellen, die als Umgebung für das zu untersuchende Modell fungieren. Beim Experimentieren komplementieren diese Modelle das Untersuchungsmodell, indem sie mit diesem in fortwährender Interaktion stehen. Umgebungsmodelle legen die potentiell möglichen Interaktionsszenarien mit dem Untersuchungsmodell zielgerichtet fest (Prenninger u. Pretschner 2004). Solche Modelle lassen sich im Rahmen der vorgestellten Kompositionsplattform auch als Komponenten definieren und zusammen mit dem Untersuchungsmodell komponieren.

Nützlich ist ein Simulationsmodell jedoch erst, falls es für eine konkrete Fragestellung valide ist. In Kapitel 2 wurde das Konzept des experimentellen Rahmens vorgestellt, in dem sich Validität explizit erfassen lässt. Ein Simulationsmodell wurde dort als System beschrieben, das mit einem experimentellen Rahmen kommuniziert. Diese Interaktion ist zu unterscheiden von der Kommunikation, die zwischen Modellteilen stattfinden. Im Gegensatz zur Interaktion zwischen Teilen eines Simulationsmodells ist für eine Fragestellung die Interaktion zwischen dem Simulationsmodell als Ganzem und einem Beobachter von Interesse.

Für die entwickelte Kompositionsplattform ergibt sich der experimentelle Rahmen als Kombination aus einer Menge von Parameterwerten und Beobachtungsinstrumenten. Parameter konfigurieren eine Komponente und Beobachtungsinstrumente liefern gewünschte Ausgabedaten. Durch das Variieren von Parametern und Beobachtungsinstrumenten können so unterschiedliche Simulationsläufe mit der gleichen Komponente durchgeführt werden.

Definition 4.14. Ein **Simulationslauf** ist ein Tupel $run = (iref, formalism, platform, obs)$, wobei $iref$ eine Schnittstellenreferenz, $formalism \in String$ ein Bezeichner für einen Integrationsformalismus, $platform \in String$ der Bezeichner für eine Zielplattform und $obs \in String$ ein Verweis auf eine Instrumentationsfunktion ist, die das Modell im Integrationsformalismus und auf der Zielplattform mit Beobachtungsinstrumenten ausstattet. Es muss gelten

1. $\mathscr{C} = instC(iref)$ ist eine Komponente.
2. $\mathscr{I} = drefI(iid_{iref})$ besitzt keine Abhängigkeiten: $\forall p \in Ports_{\mathscr{I}}.min_p = 0$.

[3] vgl. Abschnitt 3.2 zur UML und MDA

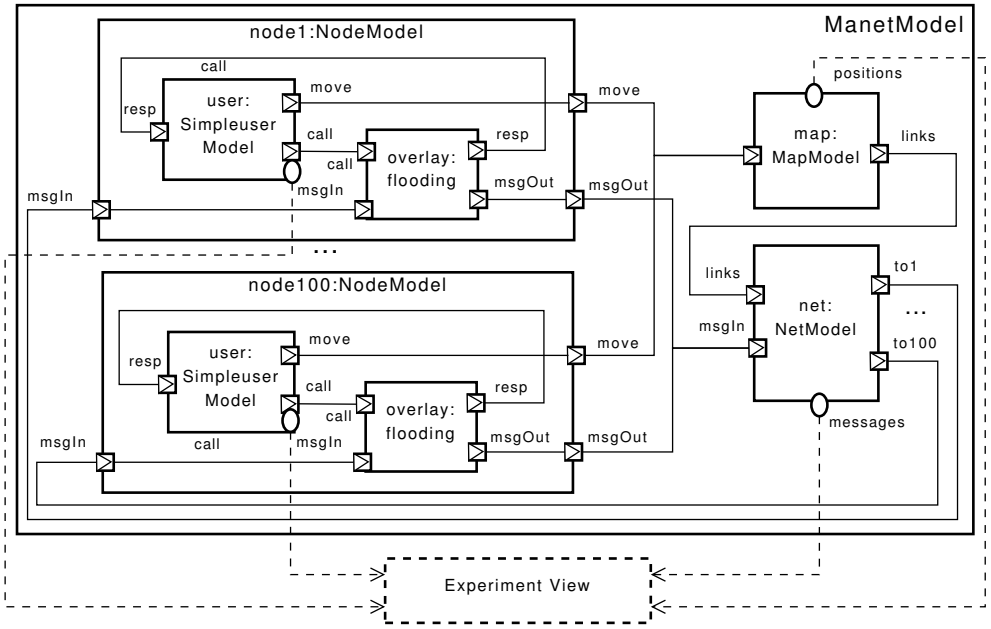

Abbildung 4.9: Struktur eines abgeleiteten Simulationsmodells mit Beobachtungsinstrumenten

Entsprechend der Definition von *instC* auf Seite 55 instantiiert ein Simulationslauf eine Komponente über ihre Schnittstelle, für eine Menge von Parametern und unter Ausführung des Konfigurators der Komponente. Die instantiierte Komponente wird für ein konkretes Simulationswerkzeug *(platform)* in einer speziellen Repräsentation *(formalism)* aufbereitet. Anschließend kann das Simulationsmodell, durch die im Experiment definierte Instrumentationsfunktion, mit Beobachtungsinstrumenten ausgestattet werden.

Die Instrumentierung von Modellen erfordert flexible Mechanismen. Strukturelle Validierung von Modellen verlangt bspw. danach, nicht nur die Modellinteraktion sondern auch interne Zustandsänderungen von Modellen sichtbar zu machen. Es existieren bisher keine plattformunabhängen Beschreibungsmittel, die diese Flexibilität bieten (Dalle u. Mrabet 2007). Die Instrumentationsfunktion selbst wird darum vorerst nicht mit in die Plattformdefinition aufgenommen. Stattdessen dient ein Verweis auf eine Instrumentationsfunktion dazu, die Konfiguration auf der Zielplattform zu initiieren. Der Konfigurator muss als spezifische Lösung für eine konkrete Zielplattform implementiert werden.

Beispiel 4.7. *Ein Simulationslauf mit der Komponente Manet (siehe Beispiel 4.6) soll 100 Knoten umfassen, jeweils mit Flooding als Protokoll zur Dienstvermittlung und SimpleUser als Nutzermodell. Die Komposition soll im Formalismus PDEVS mittels des Simulationssystems James II (Himmelspach u. Uhrmacher 2004) ausgeführt werden. Dies lässt sich spezifizieren über*

- $iref = (\text{„}manet\text{“}, \text{„}unihro/diane/com/manet:interface\text{“}, \{p_1, p_2, p_3\})$, *wobei*
  - $p_1 = (\text{„}nodes\text{“}, \text{„}int\text{“}, 100)$
  - $p_2 = (\text{„}user\text{“}, \text{„}string\text{“}, \text{„}unihro/diane/com/simpleuser:interface\text{“})$
  - $p_3 = (\text{„}protocol\text{“}, \text{„}string\text{“}, \text{„}unihro/diane/com/flooding:interface\text{“})$
- $formalism = \text{„}PDEVS\text{“}$
- $platform = \text{„}James\ II\text{“}$

- *obs instrumentiert das Simulationsmodell so, dass die Dienstnachfragen aller Nutzer, die Position der Knoten und die verschickten Netzwerknachrichten beobachtet werden können.*

Abbildung 4.9 zeigt das aus der Komponente `Manet` *gebildete PDEVS-Modell mit angehängten Beobachtungsinstrumenten.*

4.1.8 Syntax — Quo vadis?

Es wurden allgemeine Beschreibungsmittel für Schnittstellen, Komponenten und Kompositionen entwickelt, die klassische Modellierungsformalismen komplementieren. Der Fokus lag dabei auf einer möglichst flexiblen Untergliederung der Definitionseinheiten und einer vollständig schnittstellenbasierten Beschreibung von Kompositionen.

Um in Schnittstellen wahlweise eigene oder existente (und evtl. standardisierte) Definitionen verwenden zu können, wurden Typen, Rollen, Schnittstellen und Komponenten als referenzierbare Definitionseinheiten eingeführt. Schnittstellendefinitionen geben komplexe abstrakte Interaktionpunkte, genannt Rollen, bekannt. In Kompositionen können Komponenteninstanzen so, ähnlich wie in UML und SysML, entsprechend ihrer deklarierten Interaktionspunkte miteinander verbunden werden. Kompositionsverbindungen bilden den Schlüssel zur (späteren) Analyse der Kompatibilität von Teilmodellen. Komponenten können Unterkomponenten enthalten und eignen sich damit zur modular-hierarchischen Konstruktion komplexer Modelle.

Komponenten offerieren über ihre Schnittstellen Parameter, die von Konfiguratoren zum Zeitpunkt der Instantiierung ausgewertet werden. Eine Komponente realisiert auf diese Weise eine Klasse von Modellen und kann für konkrete Einsatzkontexte konfiguriert werden. Mit den entwickelten Beschreibungsmitteln lässt sich der experimentelle Rahmen als Kombination aus einer Menge von Parametern und einer spezifischen Instrumentierung des operationalen Modells formulieren.

Die mengentheoretische Definition der abstrakten Syntax eröffnet nun die Möglichkeit, die entwickelten Beschreibungsmittel mit einer formalen Semantik auszustatten. Die formale Semantikdefinition wird im Anschluss die Grundlage zur automatischen Analyse von Kompositionen bilden.

4.2 Semantik

Kompositionen werden als Bestandteil zusammengesetzter Komponenten mittels Schnittstellenreferenzen und Kompositionsverbindungen zwischen Schnittstellenports ausgedrückt. Bisher ist nicht eindeutig definiert, was diese Beschreibungen repräsentieren, d.h. welche Bedeutung mit ihnen assoziiert ist. Im Folgenden wird Komponenten und Kompositionen eine formale Semantik zugeordnet.

Während die syntaktischen Mittel der Kompositionsplattform sich nicht auf einen Integrationsformalismus und eine Zielplattform festlegen, sondern erlauben, diese frei zu wählen, bedarf die Definition der Semantik einer wohldefinierten semantischen Domäne. Von den beiden oben beschriebenen Möglichkeiten zur Abbildung von Kompositionen auf Simulationsmodelle bietet die zweite Variante (vgl. Abbildung 4.8) den Vorteil, die Semantik von Kompositionsstrukturen mittels *eines* Zielformalismuses erfassen zu können.

In Kapitel 2 wurde DEVS in einer parallelen Variante als allgemeiner Modellierungsformalismus zur modular-hierarchischen Konstruktion von Modellen vorgestellt. Die modular-hierarchischen Beschreibungsmittel gekoppelter PDEVS-Modelle korrespondieren mit denen der entwickelten Kompositionstrukturen. So dient der Formalismus PDEVS nun als semantische Domäne für (zusammengesetzte) Komponenten.

Definition 4.15. Sei $\mathscr{C} = (id, if, \kappa, \mathscr{M}, \mathscr{B}, Sub, Con)$ eine Komponente. Das durch $\mathscr{C}$ spezifizierte PDEVS-Modell wird gebildet durch

$$model(\mathscr{C}) = \begin{cases} \mathscr{M}' & \text{, falls } Sub = \emptyset \\ (X, Y, D, \{M_d\}, EIC, IC, EOC) & \text{, falls } Sub \neq \emptyset \end{cases}$$

wobei $\mathscr{M}' = pdevs(\mathscr{M})$, $X = X_{\mathscr{M}'}$ und $Y = Y_{\mathscr{M}'}$ sowie

$$\begin{aligned} D &= \bigcup_{s \in Sub} name_s \\ \{M_d\} &= \bigcup_{s \in Sub} M_{name_s}, \text{mit } M_{name_s} = model(\mathscr{C}_{name_s}) \text{ und } \mathscr{C}_{name_s} = instC(s) \\ EIC &= \{(fm, fp, tm, tp) \in Couplings(Sub', Con) | fm = \text{„this“} \wedge tm \neq \text{„this“}\} \\ EOC &= \{(fm, fp, tm, tp) \in Couplings(Sub', Con) | fm \neq \text{„this“} \wedge tm = \text{„this“}\} \\ IC &= \{(fm, fp, tm, tp) \in Couplings(Sub', Con) | fm \neq \text{„this“} \wedge tm \neq \text{„this“}\}. \end{aligned}$$

Dabei ist $Sub' = Sub \cup \{(\text{„this“}, if, \emptyset)\}$ die Menge der Schnittstellenreferenzen auf Unterkomponenten inklusive einer Referenz auf die Schnittstelle von $\mathscr{C}$ selbst.

Die Funktion *model* bildet Unterkomponenten auf Untermodelle und zusammengesetzte Komponenten rekursiv auf gekoppelte PDEVS-Modelle ab. Um die semantische Abbildung tatsächlich auf eine Zieldomäne beschränken zu können, werden Modellimplementierungen, die in anderen Formalismen vorliegen, mittels der Funktion *pdevs* in den Formalismus PDEVS transformiert. DEVS und PDEVS sind als universelle Formalismen (Zeigler u. a. 2000, 391ff.) prinzipiell geeignet, um als Zielformalismus für eine Vielzahl an Quellformalismen zu fungieren (Vangheluwe 2000). Die Funktion *pdevs* formalisiert die Beziehungen zwischen PDEVS und anderen existierenden Modellierungsformalismen. Für die semantische Abbildung wird von der konkreten Realisierung dieser Funktion abstrahiert.

Im Rahmen dieser Arbeit liegt der Fokus auf der Semantik von Kompositionsverbindungen. Die flexiblen Ausdrucksmöglichkeiten der entwickelten Kompositionsstrukturen stellen für die formale Semantikdefinition eine Herausforderung dar. Die Funktion *model* nutzt zur Abbildung von Kompositionsverbindungen die Funktion *Couplings*, die für eine Komposition (Sub, Con) eine Menge von Modellkopplungen konstruiert. Durch die Iteration über alle Kompositionsverbindungen einer Komposition ergibt sich die Menge von Modellkopplungen als

$$Couplings(Sub, Con) \stackrel{def}{=} \bigcup_{c \in Con} Coup(Sub, c),$$

wobei für jede Kompositionsverbindung $c = (k, k') = ((if, port, pos), (if', port', pos'))$ und

$$Coup(Sub, c) \stackrel{def}{=} \begin{cases} MC(c, s, s') & \text{falls } \exists s, s' \in Sub.name_s = if \wedge name_{s'} = if' \\ \bot & \text{sonst.} \end{cases}$$

Coup ordnet jeder Kompositionsverbindung c die zwei Schnittstellenreferenzen aus Sub zu, die durch c verbunden werden und delegiert die Konstruktion an eine weitere Funktion, namens MC, die für eine Kompositionsverbindung c und zwei Schnittstellenreferenzen s, s' eine Menge von Modellkopplungen konstruiert. Die Definition von MC erweist sich als aufwendig, da Kompositionsverbindungen zwei syntaktisch nützliche Abstraktionen verschränken:

- Schnittstellenports repräsentieren über Rollen eine Menge von Ereignisports und
- Schnittstellenports erlauben es, mittels Multiplizitäten eine Rolle mehrfach zu verbinden.

Einzelne Positionen innerhalb der Multiplizität von Schnittstellenports lassen sich über Bindungen unterschiedlichen Implementierungsports zuordnen.

Entsprechend dieser beiden Punkte wird die Funktion MC in zwei Schritten entwickelt:

1. Definition der Hilfsfunktion EC, die aus einer Kompositionsverbindung, die zwei Schnittstellenports miteinander verbindet, eine Menge abstrakter atomarer Verbindungen zwischen Ereignisports von Rollen konstruiert.
2. Definition der Funktion MC selbst, die atomare Verbindungen zwischen deklarierten Ereignisports (die Ergebnisse von EC) unter der Nutzung von Bindungen auf Modellkopplungen abbildet, so dass die Ereignisports der implementierenden Modelle verbunden werden.

4.2.1 EC: Bildung atomarer Verbindungen aus Kompositionsverbindungen

Kompositionsverbindungen setzen Schnittstellenports miteinander in Beziehung. Schnittstellenports sind durch Verweise auf Rollen getypt. Die Semantik einer Kompositionsverbindung hängt von den beiden Rollen ab, die von den verbundenen Schnittstellenports referenziert werden. Rollen fassen wiederum eine Menge von Ereignisports zusammen.

Syntaktisch wird eine atomare Verbindung, ähnlich zu Kompositionsverbindungen (vgl. Definition 4.10), mittels Konnektoren ausgedrückt, so dass eine atomare Verbindung Ereignisports von Rollen aufeinander bezieht.

Definition 4.16. Eine **atomare Verbindung** ist ein Tupel $ec = (k, n, k', n')$, mit zwei Konnektoren $k = (if, port, pos), k' = (if', port', pos')$ und zwei Namen für Ereignisports n, n'. Für ec muss gelten, dass $if \neq if'$.

Direkte Rückkopplung ist für atomare Verbindungen untersagt, da der Name der Startreferenz (if) ungleich des Namens der Zielreferenz (if') sein muss.

In Typsystemen fordert man für gerichtete Kommunikationskanäle, dass der Typ eines sendenden Ports ein Untertyp des empfangenden Ports sein muss (Pierce 2002). So lässt sich sicherstellen, dass ein Quellport nur Ereignisse sendet, die vom Zielport akzeptiert werden können. Dies steht im Einklang mit den Anforderungen an Kopplungen für zusammengesetzte Systembeschreibungen (Zeigler u. a. 2000, 130).

Um Ereignisports aus einer Rolle $\mathscr{R}$ sinnvoll mit Ereignisports aus einer anderen Rolle $\mathscr{R}'$ in Beziehung setzen zu können, sind für Verbindungen insgesamt drei Fälle zu unterscheiden, darunter zwei Arten delegierender Kompositionsverbindungen. Einmal können zusammengesetzte Komponenten Schnittstellenports an ihre Unterkomponenten delegieren. Zum anderen können Schnittstellenports von Unterkomponenten an zusammengesetzte Komponenten delegiert werden. Der dritte Fall deckt assemblierende Kompositionsverbindungen ab. Für assemblierende Kompositionsverbindungen müssen auf atomarer Ebene Ausgabeports mit Eingabeports in Beziehung gesetzt werden.

Um die nachfolgenden Definitionen zu vereinfachen, wird eine Hilfsfunktion *role* eingeführt, die für einen Konnektor $k = (if, port, pos)$ und eine Schnittstelle $\mathscr{I}$ die referenzierte Rolle bestimmt als

$$role(\mathscr{I}, k) \stackrel{def}{=} \begin{cases} drefR(rid_p) & \text{falls } \exists p \in Ports_{\mathscr{I}}.name_p = port_k \wedge min_p \leq pos_k \leq max_p \\ \bot & \text{sonst.} \end{cases}$$

Atomare Verbindungen mit passender Richtung und Typsicherheit werden als wohlgeformt bezeichnet.

Definition 4.17. Eine atomare Verbindung $ec = (k, n, k', n')$ ist für zwei Schnittstellenreferenzen s, s' **wohlgeformt**, falls für $\mathscr{I} = drefI(name_s)$, $\mathscr{I}' = drefI(name_{s'})$, $\mathscr{R} = role(\mathscr{I}, k)$ und $\mathscr{R}' = role(\mathscr{I}', k')$ gilt, dass $\exists e \in EP_{\mathscr{R}}, e' \in EP_{\mathscr{R}'}.name_e = n \wedge name_{e'} = n'$, so dass

1. Konsistente Bezeichner: $if_k = name_s \wedge if_{k'} = name_{s'}$
2. Passende Richtungen:

$$\begin{aligned} if_k = \text{„this“} &\Rightarrow inp_e \wedge inp_{e'} \\ if_{k'} = \text{„this“} &\Rightarrow \neg inp_e \wedge \neg inp_{e'} \\ if_k \neq \text{„this“} \wedge if_{k'} \neq \text{„this“} &\Rightarrow \neg inp_e \wedge inp_{e'} \end{aligned}$$

3. Typsicherheit: $drefT(tid_e) \sqsubseteq drefT(tid_{e'})$

Für wohlgeformte atomare Verbindungen wird die Notation $wellformed(ec, s, s')$ verwendet.

Im Rahmen der semantischen Abbildung wird nun das Ziel verfolgt, aus Kompositionsverbindungen wohlgeformte atomare Verbindungen zu konstruieren. Kompositionsverbindungen werden dementsprechend so interpretiert, dass sie die Ereignisports zweier Rollen $\mathscr{R}, \mathscr{R}'$ zueinander in Beziehung setzen. Die Menge der Paare wird als $matches(\mathscr{R}, \mathscr{R}')$ bezeichnet, wobei

$$\begin{aligned} matches(\mathscr{R}, \mathscr{R}') \stackrel{def}{=} \{&(f,t) \in String \times String | \\ &\exists e \in EP_{\mathscr{R}}, e' \in EP_{\mathscr{R}'}.name_e = f \wedge name_{e'} = t \wedge inp_e \neq inp_{e'} \\ &\wedge (inp_e \Rightarrow drefT(tid_{e'}) \sqsubseteq drefT(tid_e)) \wedge (inp_{e'} \Rightarrow drefT(tid_e) \sqsubseteq drefT(tid_{e'}))\}. \end{aligned}$$

Dabei bezeichnet f immer einen Ereignisport aus $\mathscr{R}$ und t einen Ereignisport aus $\mathscr{R}'$. Falls f einen Eingabeport repräsentiert, steht t für einen Ausgabeport. Steht f für einen Ausgabeport, repräsentiert t einen Eingabeport. Ein Ausgabeport darf nur Ereignistypen senden, die vom verbundenen Eingabeport empfangen werden können.

Bezogen auf die drei möglichen Richtungen von Verbindungen (einmal assemblierend und zwei Möglichkeiten delegierender Verbindungen), drückt *matches* nur für assemblierende Kompositionsverbindungen sinnvolle Beziehungen aus. Delegierende Kompositionsverbindungen sollten nicht Ereignisports mit entgegengesetzten, sondern mit gleichen Richtungen in Beziehung setzen. Eingabeports sollten auf Eingabeports und Ausgabeports auf Ausgabeports bezogen werden. Dafür ist es sinnvoll, komplementäre Rollen bilden zu können, wie es bspw. in SysML möglich ist[4]. Im Folgenden bezeichnet $\overline{\mathscr{R}}$ das Komplement einer Rolle $\mathscr{R}$, so dass alle Ereignisports in $\overline{\mathscr{R}}$ die gleichen Namen und Typen wie in $\mathscr{R}$ besitzen, jedoch die entgegengesetzten Richtungen:

$$\overline{\mathscr{R}} \stackrel{def}{=} (\bot, EP_{\overline{\mathscr{R}}}), \text{mit } EP_{\overline{\mathscr{R}}} = \{(name_e, tid_e, \neg inp_e) | (name_e, tid_e, inp_e) \in EP_{\mathscr{R}}\}.$$

Da der Name einer Rolle keinen Einfluss auf *matches* hat, benötigt das Komplement keinen Namen.

Gegeben seien nun eine Kompositionsverbindung $c = (start, end)$ und zwei Schnittstellenreferenzen s, s'. Die referenzierten Schnittstellen ergeben sich aus $\mathscr{I} = drefI(iid_s)$ und $\mathscr{I}' = drefI(iid_{s'})$, und ihre Rollen durch $\mathscr{R} = role(\mathscr{I}, start)$ und $\mathscr{R}' = role(\mathscr{I}', end)$.

Die drei möglichen Arten von Kompositionsverbindungen lassen sich nun für *matches* anwenden, so dass für $c = (start, end)$ und $start = (if, port, pos)$:

$$M(c, s, s') \stackrel{def}{=} \begin{cases} matches(\overline{\mathscr{R}}, \mathscr{R}') & \text{falls } if_{start} = \text{„this“} \\ matches(\mathscr{R}, \overline{\mathscr{R}'}) & \text{falls } if_{end} = \text{„this“} \\ matches(\mathscr{R}, \mathscr{R}') & \text{falls } if_{start} \neq \text{„this“} \wedge if_{end} \neq \text{„this“}. \end{cases}$$

Mittels der Elemente von M lässt sich eine Menge atomarer Verbindungen konstruieren, so dass

$$EC(c, s, s') \stackrel{def}{=} \bigcup_{m \in M(c,s,s')} ec(start_c, end_c, m, \mathscr{R}, \mathscr{R}')$$

[4]Das Attribut *conjugated* invertiert für einen SysML-*flowport* die Richtungen aller Ereignisflüsse.

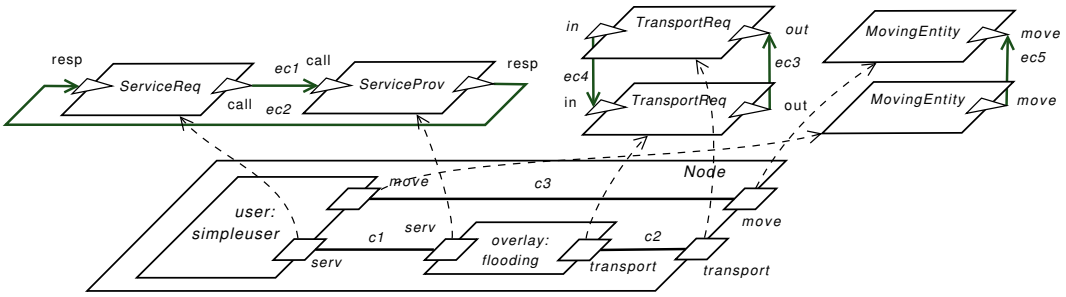

Abbildung 4.10: Konstruktion atomarer Verbindungen aus Kompositionsverbindungen

Für jedes Match $m = (f, t)$ wird nun eine atomare Verbindung gebildet. Dazu muss, je nach Art des Startkonnektors (ob der Konnektor von einer komponierenden oder komponierten Schnittstelle ausgeht) und in Abhängigkeit von der Richtung des ersten Ereignisports (der Port mit dem Namen f im gebildeten Paar $m = (f, t)$), die Richtung der atomaren Verbindung angepasst werden, so dass

$$ec(k, k', m, \mathscr{R}, \mathscr{R}') \stackrel{def}{=} \begin{cases} (k, f, k', t) & \text{falls } \exists e \in EP_{\mathscr{R}}.name_e = f \wedge if_k = \text{„this“} \wedge inp_e \\ (k, f, k', t) & \text{falls } \exists e \in EP_{\mathscr{R}}.name_e = f \wedge if_k \neq \text{„this“} \wedge \neg inp_e \\ (k', t, k, f) & \text{falls } \exists e \in EP_{\mathscr{R}}.name_e = f \wedge if_k \neq \text{„this“} \wedge inp_e \\ (k', t, k, f) & \text{falls } \exists e \in EP_{\mathscr{R}}.name_e = f \wedge if_k = \text{„this“} \wedge \neg inp_e \\ \bot & \text{sonst} \end{cases}$$

Mit ec ist das Ende der verschachtelten Funktionsaufrufe erreicht. Jeder Aufruf von ec liefert eine atomare Verbindung bzw. $\bot$, falls keine Verbindung konstruiert werden kann.

Beispiel 4.8. *Für die Komposition eines Netzwerkknotens visualisiert Abbildung 4.10 die Konstruktion atomarer Verbindungen. Der untere Teil der Abbildung zeigt die Komponente Node, wie in Beispiel 4.5 eingeführt. Node enthält zwei Unterkomponenten, so dass* $Sub' = \{s1, s2, sthis\}$, *mit*

$$\begin{aligned} s1 &= (\text{„}user\text{“}, \text{„}unihro/diane/com/simpleuser{:}interface\text{“}, \emptyset), \\ s2 &= (\text{„}overlay\text{“}, \text{„}unihro/diane/com/flooding{:}interface\text{“}, \emptyset) \\ sthis &= (\text{„}this\text{“}, \text{„}unihro/diane/com/node{:}interface\text{“}, \emptyset) \end{aligned}$$

Node enthält weiterhin die drei Kompositionsverbindungen

$$\begin{aligned} c1 &= ((\text{„}user\text{“}, \text{„}serv\text{“}, 1), (\text{„}overlay\text{“}, \text{„}serv\text{“}, 1)) \\ c2 &= ((\text{„}overlay\text{“}, \text{„}transport\text{“}, 1), (\text{„}this\text{“}, \text{„}transport\text{“}, 1)) \\ c3 &= ((\text{„}user\text{“}, \text{„}move\text{“}, 1), (\text{„}this\text{“}, \text{„}move\text{“}, 1)) \end{aligned}$$

In Abbildung 4.10 verweisen die gestrichelten Pfeile von den Schnittstellenports auf die referenzierten Rollen, die ihrerseits im oberen Teil der Abbildung zu sehen sind. Aus den Kompositionsverbindungen ergeben sich die atomaren Verbindungen durch $EC(c1, s1, s2) = \{ec1, ec2\}$,

$EC(c2, s2, sthis) = \{ec3, ec4\}$ *und* $EC(c3, s1, sthis) = \{ec5\}$*, mit*

$$\begin{aligned}
ec1 &= ((\text{„}user\text{“}, \text{„}serv\text{“}, 1), \text{„}call\text{“}, (\text{„}overlay\text{“}, \text{„}serv\text{“}, 1), \text{„}call\text{“}) \\
ec2 &= ((\text{„}overlay\text{“}, \text{„}serv\text{“}, 1), \text{„}resp\text{“}, (\text{„}user\text{“}, \text{„}serv\text{“}, 1), \text{„}resp\text{“}) \\
ec3 &= ((\text{„}overlay\text{“}, \text{„}transport\text{“}, 1), \text{„}out\text{“}, (\text{„}this\text{“}, \text{„}transport\text{“}, 1), \text{„}out\text{“}) \\
ec4 &= ((\text{„}this\text{“}, \text{„}transport\text{“}, 1), \text{„}in\text{“}, (\text{„}overlay\text{“}, \text{„}transport\text{“}, 1), \text{„}in\text{“}) \\
ec5 &= ((\text{„}user\text{“}, \text{„}move\text{“}, 1), \text{„}move\text{“}, (\text{„}this\text{“}, \text{„}move\text{“}, 1), \text{„}move\text{“})
\end{aligned}$$

Die Leistung von EC besteht darin, die Untergliederung einer Kompositionsverbindung zwischen Schnittstellenports auf einzelne Verbindungen zwischen Ereignisports von Rollen abzubilden.

4.2.2 MC: Konstruktion von Modellkopplungen aus atomaren Verbindungen

Die durch EC gebildeten atomaren Verbindungen dienen nun als Grundlage, um Modellkopplungen zu erstellen. Eine **Modellkopplung** wird als Viertupel $mc = (fm, fp, tm, tp)$ ausgedrückt, wobei $fm, fp, tm, tp \in String$ das Quellmodell, den Quellport, das Zielmodell bzw. den Zielport einer Kopplung bezeichnen. Der Zusammenhang zwischen abstrakten Verbindungen und Modellkopplungen ergibt sich durch die Bindungen der beiden beteiligten Komponenten.

Aus einer Kompositionsverbindung c und zwei Schnittstellenreferenzen s, s' wird eine Menge von Modellkopplungen gebildet durch

$$MC(c, s, s') \stackrel{def}{=} \bigcup_{ec \in EC(c,s,s')} mc(ec, s, s', \mathscr{B}, \mathscr{B}').$$

Die Funktion MC iteriert über alle durch EC gebildeten atomaren Verbindungen und ordnet den Schnittstellenreferenzen s und s' die in den entsprechenden Komponenten definierten Bindungen $\mathscr{B}$ und $\mathscr{B}'$ zu.

Für eine atomare Verbindung $ec = (k, n, k', n')$, zwei Schnittstellenreferenzen s, s', mit $s = (name, iid, Params)$ und $s' = (name', iid', Params')$, und zwei Bindungen $\mathscr{B}, \mathscr{B}'$, konstruiert die Funktion mc eine Modellkopplung, so dass für $k = (if, port, pos)$ und $k' = (if', port', pos')$
$mc((k, n, k', n'), s, s', \mathscr{B}, \mathscr{B}') \stackrel{def}{=}$

$$\begin{cases}
(if, impl(port, n, pos, \mathscr{B}), if', impl(port', n', pos', \mathscr{B}')) & \text{falls } if = name \wedge if' = name' \\
(if, impl(port', n', pos', \mathscr{B}'), if', impl(port, n, pos, \mathscr{B})) & \text{falls } if = name' \wedge if' = name \\
\bot & \text{sonst.}
\end{cases}$$

Entsprechend der Schnittstellennamen if und if' ordnet mc die zwei Schnittstellenreferenzen s, s' und die beiden Bindungen $\mathscr{B}, \mathscr{B}'$ den Konnektoren k, k' zu. Auf diese Kombination wird dann jeweils die Funktion $impl$ angewendet, die für einen deklarierten Ereignisport und eine Bindung, den Namen des implementierenden Ereignisports liefert (siehe die Definition der Funktion $impl$ im Kontext von Bindungen auf Seite 52).

Beispiel 4.9. *Gegeben sei eine Komposition* (Sub, Con)*, mit* $Sub = \{s_1, s_2, s_3\}$ *und* $Con = \{c_1, c_2\}$*, wobei*

$$\begin{aligned}
s_1 &= (node1, node, \emptyset) \\
s_2 &= (node2, node, \emptyset) \\
s_3 &= (net, net, \{(nodes, 2)\}) \\
c_1 &= ((\text{„}node1\text{“}, \text{„}transport\text{“}, 1), (\text{„}net\text{“}, \text{„}transport\text{“}, 1)) \\
c_2 &= ((\text{„}node2\text{“}, \text{„}transport\text{“}, 1), (\text{„}net\text{“}, \text{„}transport\text{“}, 2))
\end{aligned}$$

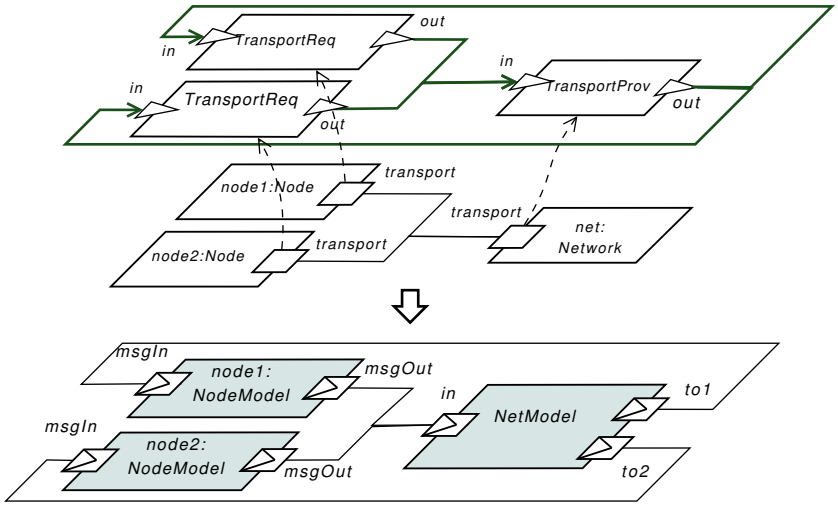

Abbildung 4.11: Konstruktion von Modellkopplungen

Die Konstruktion von konkreten Modellkopplungen aus abstrakten Verbindungen veranschaulicht Abbildung 4.11. Im oberen Teil der Abbildung ist die Komposition inklusive der Verweise auf Rollen und den konstruierten atomaren Verbindungen dargestellt. Die atomaren Verbindungen wurden gebildet durch $EC(c_1, s_1, s_3) = \{ec_1, ec_2\}$ *und* $EC(c_2, s_2, s_3) = \{ec_3, ec_4\}$*, wobei*

$$\begin{aligned} ec_1 &= ((„node1“, „transport“, 1), „out“, („net“, „transport“, 1), „in“) \\ ec_2 &= ((„net“, „transport“, 1), „out“, („node1“, „transport“, 1), „in“) \\ ec_3 &= ((„node2“, „transport“, 1), „out“, („net“, „transport“, 1), „in“) \\ ec_4 &= ((„net“, „transport“, 2), „out“, („node1“, „transport“, 1), „in“) \end{aligned}$$

Für die Bindungen

$$\begin{aligned} \mathscr{B}_{node1} &= \{(„transport“, „in“, 1, „msgIn“), („transport“, „out“, 1, „msgOut“)\} \\ \mathscr{B}_{node2} &= \{(„transport“, „in“, 1, „msgIn“), („transport“, „out“, 1, „msgOut“)\} \\ \mathscr{B}_{net} &= \{(„transport“, „out“, 1, „to1“), („transport“, „out“, 2, „to2“)\} \end{aligned}$$

ergeben sich die Modellkopplungen

$$\begin{aligned} mc(ec_1, s_1, s_3, \mathscr{B}_{node1}, \mathscr{B}_{net}) &= („node1“, „msgOut“, „net“, „in“) \\ mc(ec_2, s_1, s_3, \mathscr{B}_{node1}, \mathscr{B}_{net}) &= („net“, „to1“, „node1“, „msgIn“) \\ mc(ec_3, s_2, s_3, \mathscr{B}_{node2}, \mathscr{B}_{net}) &= („node2“, „msgOut“, „net“, „in“) \\ mc(ec_4, s_2, s_3, \mathscr{B}_{node2}, \mathscr{B}_{net}) &= („net“, „to2“, „node2“, „msgIn“) \end{aligned}$$

So resultieren die Bindungen der Netzwerkkomponente in Modellkopplungen, die einzelne Knotenmodelle durch unterschiedliche Ausgabeports ansprechen. Die resultierende Modellstruktur wurde bereits in Abbildung 4.9 gezeigt.

Auf Grundlage von MC lässt sich die Abbildung von Komponenten auf Modelle in geschlossener Form angeben (siehe Definition 4.15). Mit der Funktion *model* steht die formale Definition bereit, um eine (zusammengesetzte) Komponente auf ein PDEVS-Modell abzubilden. In Beispiel 4.7 findet sich das Ergebnis für die Anwendung von *model* auf die Komponente Manet. Die resultierende

Modellstruktur zeigt Abbildung 4.9.

Die formal aufwändige Semantikdefinition für Kompositionsverbindungen befähigt nun dazu, Eigenschaften eines Simulationsmodells aus den Eigenschaften einer Komposition abzuleiten.

4.3 Kompositionale Eigenschaften

Eine wesentliche Motivation für Kompositionsansätze gründet im Wunsch, Korrektheit durch Konstruktion zu gewährleisten. Eigenschaften auf zusammengesetzter Ebene sollen sich aus den Eigenschaftsbeschreibungen der Teile und den Regeln des Zusammenfügens automatisch ableiten lassen. Im Folgenden werden Bedingungen an die beteiligten Komponenten und die Kompositionsstrukturen formalisiert, so dass für ein, aus ihnen abgeleitetes, Simulationsmodell nützliche Eigenschaften garantiert werden können:

- *Kompatibilität* formalisiert, ob Komponenten über ihre Schnittstellenports zusammengefügt werden können. Setzen Kompositionsverbindungen kompatible Rollen zueinander in Beziehung, liefert EC wohlgeformte atomare Verbindungen.
- *Verfeinerung* formalisiert die Beziehung zwischen Schnittstellendefinitionen und Modellimplementierungen. Implementierungen müssen ihre Schnittstellen verfeinern, damit sich (mittels MC) aus wohlgeformten atomaren Verbindungen wohlgeformte Modellkopplungen bilden lassen.
- *Vollständigkeit* erfasst die Anforderungen von Komponenten an Einsatzkontexte. Sind alle Abhängigkeiten in einer Komposition erfüllt, lässt sich sicherstellen, dass auch die Anforderungen der Modellimplementierungen erfüllt sind.
- *Korrektheit* fasst die Anforderungen an Komponenten zusammen, so dass sich mittels *model* wohlgeformte Simulationsmodelle bilden lassen.

4.3.1 Kompatibilität

Ob sich aus einer Kompositionsverbindung wohlgeformte atomare Verbindungen bilden lassen, hängt von der Beziehung zwischen den beiden referenzierten Rollen der verbundenen Schnittstellenports ab. Die Kompatibilität von Rollen wird dementsprechend auf die, in den Rollen enthaltenen, Ereignisports zurückgeführt.

Definition 4.18. Kompatibilität von Rollen ist eine Relation $K \subseteq \mathit{Role} \times \mathit{Role}$, so dass für zwei Rollen $\mathscr{R}, \mathscr{R}' \in \mathit{Role}$ das Paar $(\mathscr{R}, \mathscr{R}') \in K$ genau dann, falls

1. $\forall e \in EP_{\mathscr{R}}.\exists e' \in EP_{\mathscr{R}'}$, so dass

$$\begin{aligned} &\mathit{inp}_e \Rightarrow \neg \mathit{inp}_{e'} \wedge \mathit{dref}(\mathit{tid}_e) \sqsupseteq \mathit{dref}(\mathit{tid}_{e'}) \\ &\neg \mathit{inp}_e \Rightarrow \mathit{inp}_{e'} \wedge \mathit{dref}(\mathit{tid}_e) \sqsubseteq \mathit{dref}(\mathit{tid}_{e'}) \end{aligned}$$

2. $\forall e' \in EP_{\mathscr{R}'}.\exists e \in EP_{\mathscr{R}}$, so dass

$$\begin{aligned} &\mathit{inp}_{e'} \Rightarrow \neg \mathit{inp}_e \wedge \mathit{dref}(\mathit{tid}_{e'}) \sqsupseteq \mathit{dref}(\mathit{tid}_e) \\ &\neg \mathit{inp}_{e'} \Rightarrow \mathit{inp}_e \wedge \mathit{dref}(\mathit{tid}_{e'}) \sqsubseteq \mathit{dref}(\mathit{tid}_e) \end{aligned}$$

Falls $(\mathscr{R}, \mathscr{R}') \in R$ werden $\mathscr{R}$ und $\mathscr{R}'$ als **kompatibel** bezeichnet, repräsentiert durch $\mathscr{R} \sim \mathscr{R}'$.

Kompatibilität von Rollen drückt aus, ob für jeden Ausgabeport der einen Rolle jeweils ein Eingabeport der anderen Rolle existiert, so dass der Typ des Ausgabeports ein Untertyp des Eingabeports ist. Falls zwei Rollen $\mathscr{R}, \mathscr{R}'$ kompatibel sind, lässt sich die Menge $matches(\mathscr{R}, \mathscr{R}') = \{(s,t) \in String \times String\}$ so bilden, dass für alle Paare die Ausgabeports Untertypen der Eingabeports verwenden.

Für Kompositionsverbindungen gilt es jedoch drei Fälle zu unterscheiden: 1. assemblierende, 2. delegierende, die von der übergeordneten Komponente zur Unterkomponente verlaufen und 3. delegierende, die von einer Unterkomponente zur übergeordneten Komponente delegieren. Im Rahmen delegierender Kompositionsverbindungen bedarf es komplementär-kompatibler Rollen (vgl. Abschnitt 4.2.1). Analog zu wohlgeformten atomaren Verbindungen lässt sich mittels der Kompatibilitätsbeziehung zwischen Rollen formulieren, wann eine Kompositionsverbindung wohlgeformt ist.

Definition 4.19. Eine Kompositionsverbindung $c = (start, end)$ ist für zwei Schnittstellenreferenzen s, s' **wohlgeformt**, falls für $\mathscr{R} = role(drefI(iid_s), start)$ und $\mathscr{R}' = role(drefI(iid_{s'}), end)$ gilt

1. Referenzierte Rollen existieren: $\mathscr{R} \neq \bot \wedge \mathscr{R}' \neq \bot$
2. Kompatible Rollen:

$$\begin{aligned} if_{start} = \text{„this“} &\Rightarrow \overline{\mathscr{R}} \sim \mathscr{R}' \\ if_{end} = \text{„this“} &\Rightarrow \mathscr{R} \sim \overline{\mathscr{R}'} \\ assembly(c) &\Rightarrow \mathscr{R} \sim \mathscr{R}' \end{aligned}$$

Für eine Kompositionsverbindung c und zwei Schnittstellenreferenzen s, s' drückt $wellformed(c, s, s')$ aus, dass c wohlgeformt ist.

Die erste Bedingung stellt insbesondere sicher, dass Kompositionsverbindungen Ports im Rahmen ihrer Multiplizität verbinden. Wohlgeformte Kompositionsverbindungen garantieren, dass sich mittels EC wohlgeformte atomare Verbindungen (vgl. Definition 4.17) konstruieren lassen und damit der erste Schritt der semantischen Abbildung zu sinnvollen Ergebnissen führt.

Lemma 1. *Gegeben seien eine Kompositionsverbindung $c = (start, end)$ und zwei Schnittstellenreferenzen s, s'. Ist c wohlgeformt, konstruiert EC wohlgeformte atomare Verbindungen:*

$$\begin{aligned} wellformed(c, s, s') \Rightarrow \forall ec \in EC(c, s, s'). \quad & start_{ec} = start_c \wedge wellformed(ec, s, s') \\ & \vee start_{ec} = end_c \wedge wellformed(ec, s', s) \end{aligned}$$

Beweis. Siehe Anhang A.1. □

Kompatibilität ist das Kriterium, das in Kompositionen von den Regeln des Zusammensetzens (Kompositionsverbindungen) eingehalten werden muss. Kompositionalität fordert jedoch auch von den Teilen, die zusammengesetzt werden, die Einhaltung bestimmter Spielregeln. Die wichtigste ist, dass Implementierungen ihre Schnittstellen verfeinern.

4.3.2 Verfeinerung

Um wohlgeformte Modellkopplung zu garantieren, muss zusätzlich zu wohlgeformten Kompositionsverbindungen sichergestellt sein, dass die Implementierungen konsistent zu ihren Schnittstellen sind. Diese Vorstellung formalisieren Verfeinerungsrelationen. Prinzipiell lassen sich drei Arten von Verfeinerungsrelationen unterscheiden (Schröter 2004):

- *bewahrende:* Alle in einer Schnittstelle spezifizierten Eigenschaften gelten auch für die Implementierung.
- *reflektierende*: Alle relevanten Eigenschaften, die nicht in der Schnittstelle deklariert sind, dürfen auch nicht für die Implementierung gelten.
- *äquivalente:* Die Schnittstelle spezifiziert genau die gleichen (relevanten) Eigenschaften wie die Implementierung.

Eine bewahrende Verfeinerung fordert von einer Implementierung, dass sie alle in der Schnittstelle deklarierten Eigenschaften umsetzt. Damit erfassen bewahrende Verfeinerungen die Angebote von Komponenten bzw. die Anforderungen potentieller Einsatzkontexte. Reflektierende Verfeinerungen fordern hingegen von einer Implementierung, dass unerwünschte Eigenschaften, die nicht in der Schnittstelle deklariert sind, auch nicht in der Implementierung vorkommen dürfen. Positiv formuliert heißt das, dass alle relevanten Eigenschaften der Implementierung in der Schnittstelle reflektiert sein müssen. Damit erfassen reflektierende Verfeinerungen die Kontextanforderungen einer Implementierung in der Schnittstelle.

Die Definition von Verfeinerungsrelationen wird gewöhnlich vereinfacht, indem Verfeinerung als binäre Relation über Systeme gleichen Typs definiert wird (de Alfaro u. Henzinger 2005). Schnittstellen und Modellimplementierungen werden im entwickelten Metamodell jedoch in unterschiedlichen Sprachen repräsentiert. Zudem haben Bindungen Einfluss auf das Verhältnis von Schnittstellen und Implementierungen. Für die bewahrende Verfeinerung ergibt sich die folgende Definition.

Definition 4.20. Sei $\mathscr{I}$ eine Schnittstelle, $\mathscr{M}$ ein PDEVS-Modell und $\mathscr{B}$ eine Bindung. Das Modell $\mathscr{M}$ und die Bindung $\mathscr{B}$ stellen eine **bewahrende Verfeinerung** für die Schnittstelle $\mathscr{I}$ dar, falls $\forall p \in Ports_{\mathscr{I}}.\forall pos \in [1, max_p]$, für die Rolle $\mathscr{R} = drefR(rid_p)$, für alle $e \in EP_{\mathscr{R}}$, für $ip = impl(name_p, name_e, pos, \mathscr{B})$ gilt:

$$\begin{aligned} inp_e &\Rightarrow ip \in InPorts_{\mathscr{M}} \wedge car_{dref(tid_e)} = X_{ip,\mathscr{M}} \\ \neg inp_e &\Rightarrow ip \in OutPorts_{\mathscr{M}} \wedge car_{dref(tid_e)} = Y_{ip,\mathscr{M}}. \end{aligned}$$

Verfeinert das Modell $\mathscr{M}$ eine Schnittstelle $\mathscr{I}$ unter Bindung $\mathscr{B}$ so, dass die Eigenschaften von $\mathscr{I}$ bewahrt bleiben, drückt dies die Notation $\mathscr{I} \succ_p (\mathscr{M}, \mathscr{B})$ aus.

Bewahrende Verfeinerung fordert dementsprechend von einer Modellimplementierung, dass alle in der Schnittstelle deklarierten Ereignisports an existente Modellports mit gleichem Wertebereich gebunden sind[5].

Modellkopplungen werden im Folgenden als Kopplungen für PDEVS-Modelle erstellt. Entsprechend der Definition gekoppelter PDEVS-Modelle (vgl. Definition 2.2) muss für eine Modellkopplung $mc = (fm, fp, tm, tp)$ sichergestellt werden, dass $fm \neq tm$ und $XY_{fm,fp} \subseteq XY_{tm,tp}$, wobei $XY_{m,p} = X_{m,p}$, falls $fp \in InPorts_m$ bzw. $XY_{m,p} = Y_{m,p}$ falls $fp \in OutPorts_m$. Erfüllt eine Modellkopplung mc diese Bedingungen, wird dafür im Folgenden der Ausdruck $wellformed(mc)$ genutzt.

Auf Grundlage der formalen Definition von bewahrender Verfeinerung lässt sich sicherstellen, dass MC wohlgeformte Modellkopplungen bildet.

Lemma 2. *Gegeben seien eine Kompositionsverbindung c, zwei Schnittstellenreferenzen s, s', mit $\mathscr{I} = drefI(s)$ und $\mathscr{I}' = drefI(s')$, zwei Bindungen $\mathscr{B}, \mathscr{B}'$ und zwei PDEVS-Modelle $\mathscr{M}, \mathscr{M}'$. Werden die beiden Schnittstellen durch ihre Modelle und Bindungen bewahrend verfeinert und durch*

[5] Die Forderung nach gleichen Wertebereichen (und nicht einschließenden Wertebereichen) liegt darin begründet, dass Ports von gekoppelten Modellen sowohl als Quelle als auch als Ziel von Sendeoperationen fungieren.

eine wohlgeformte Kompositionsverbindung verbunden, dann sind alle durch MC konstruierten Modellkopplungen wohlgeformt:

$$\mathscr{I} \succ_p (\mathscr{M}, \mathscr{B}) \wedge \mathscr{I}' \succ_p (\mathscr{M}', \mathscr{B}') \wedge wellformed(c, s, s') \Rightarrow \forall mc \in MC(c, s, s').wellformed(mc)$$

Beweis. Siehe Anhang A.2. □

Für eine wohlgeformte Kompositionsverbindung konstruiert MC wohlgeformte Modellkopplungen, falls die beteiligten Modelle ihre Schnittstellen verfeinern. Schnittstellen werden dadurch zu aussagekräftigen Beschreibungen.

Das Gegenstück zur bewahrenden Verfeinerung ist die reflektierende Verfeinerung. Um Kompositionen allein auf der Grundlage von Schnittstellenbeschreibungen zu analysieren, muss sichergestellt werden, dass alle wesentlichen Anforderungen der Modellimplementierungen in den Schnittstellen repräsentiert sind. In modular-hierarchischen Formalismen wie DEVS ist es nicht möglich, Anforderungen eines Modells an andere Modelle zu spezifizieren. Der Einsatzkontext eines Modells ist nicht verpflichtet, bestimmte Ports des Modells zum Zeitpunkt des Einsatzes zu verbinden. Gekoppelte DEVS-Modelle sind bereits wohlgeformt, falls alle Kopplungen kompatible Ports verbinden. Dass die Kopplungen existieren, lässt sich allerdings nicht sicherstellen.

Für eine Modellimplementierung sollte jedoch festlegbar sein, welche Ports zum Zeitpunkt des Einsatzes verbunden werden müssen. Im Folgenden bezeichnet $required(\mathscr{M})$ für ein Modell $\mathscr{M}$ die Menge zu verbindender Ports.

Definition 4.21. Eine Modellimplementierung $\mathscr{M}$ und eine Bindung $\mathscr{B}$ stellen für eine Schnittstellendefinition $\mathscr{I}$ eine **reflektierende Verfeinerung** dar, falls $\forall ip \in required(\mathscr{M}).\exists p \in Ports_{\mathscr{I}}.\exists e \in EP_{drefR(rid_p)}.\exists pos \in [1, min_p]$, so dass $ip = impl(p, e, pos, \mathscr{B})$. Für reflektierende Verfeinerungen schreiben wir $\mathscr{I} \succ_r (\mathscr{M}, \mathscr{B})$.

Reflektierende Verfeinerung ist gegeben, falls jeder benötigte Modellport an eine Deklaration in der Schnittstelle gebunden ist, die verbunden werden muss. Damit kann sichergestellt werden, dass alle zu verbindenden Modellports in einer Schnittstelle repräsentiert sind.

Beispiel 4.10. *Das Modell* `Flooding` *erfordert, dass zum Zeitpunkt des Einsatzes die Ports zum Versenden und Empfangen von Netzwerknachrichten verbunden werden. Die Interaktion über die Ports „msgOut“ und „msgIn“ stellt eine Kontextanforderung des Modells Flooding dar. Beide Ports sollen in jedem Einsatzkontext verbunden werden. Wenn Flooding eine Nachricht verschickt, erwartet es eine Antwort. Für das Funktionieren von Flooding ist es dagegen nicht erforderlich, dass calls eintreffen oder produzierte Ereignisse vom Typ response von einem anderen Modell tatsächlich empfangen werden. Für*

- $\mathscr{I} = (\ldots, Ports)$*, mit* $Ports = \{(„serv“, „unihro/diane/base/service:ServiceProv“, 0, 1), („transport“, „unihro/diane/base/transport:TransportReq“, 1, 1)\}$*,*
- $\mathscr{R}_{transport} = („unihro/diane/base/transport:TransportReq“, EP)$*, mit* $EP = \{(„in“, „unihro/diane/base/transport:Message“, true), („out“, „unihro/diane/base/transport:Message“, false)\}$*,*
- $\mathscr{B} = \{(„transport“, „in“, 1, „msgIn“), („transport“, „out“, 1, „msgOut“)\}$ *und*
- $required(\mathscr{M}) = \{„msgIn“, „msgOut“\}$

gilt $\mathscr{I} \succ_r (\mathscr{M}, \mathscr{B})$*, da die beiden zu verbindenden Implementierungsports „msgIn“ und „msgOut“ mittels* $\mathscr{B}$ *so an* $\mathscr{I}$ *gebunden sind, dass sie als Kontextanforderungen in Kompositionen sichtbar sind. Sie sind an die deklarierten Ereignisports des Schnittstellenports „transport“ gebunden, der eine minimale Multiplizität von eins besitzt.*

Auf Basis bewahrender und reflektierender Verfeinerungen lässt sich äquivalente Verfeinerung formalisieren als

$$\mathscr{I} \succ (\mathscr{M}, \mathscr{B}) \stackrel{def}{=} \mathscr{I} \succ_r (\mathscr{M}, \mathscr{B}) \wedge \mathscr{I} \succ_p (\mathscr{M}, \mathscr{B}).$$

Eine Schnittstelle drückt dann alle wesentlichen Anforderungen einer Implementierung aus und ein Modell implementiert alle angekündigten Fähigkeiten einer Schnittstelle.

4.3.3 Vollständige Kompositionen

Reflektierende Verfeinerung drückt aus, ob alle Anforderungen einer Modellimplementierung in der Schnittstelle deklariert sind. In Kompositionen kommt es darauf an, dass deklarierte Anforderungen erfüllt werden.

Definition 4.22. Eine Komposition (Sub, Con) ist **vollständig**, falls

1. Korrekte Kompositionsverbindungen: $\forall c \in Con. \exists s, s' \in Sub. wellformed(c, s, s')$
2. Erfüllte Abhängigkeiten: $\forall s \in Sub$ mit $\mathscr{I} = drefI(iid_s)$ und $\forall p \in Ports_{\mathscr{I}}$ und für alle $m \in [1, min_p]$ existiert eine Kompositionsverbindung $(start, end) \in Con$ für die $if_{start} = name_s \wedge port_{start} = name_p \wedge pos_{start} = m$ oder für die $if_{end} = name_s \wedge port_{end} = name_p \wedge pos_{end} = m$

Für vollständige Kompositionen wird die Notation $complete(Sub, Con)$ verwendet.

Um vollständig zu sein, muss in einer Komposition jeder Port einer referenzierten Schnittstelle für alle Positionen verbunden sein, die kleiner oder gleich der minimalen Multiplizität sind. Weiterhin wird gefordert, dass alle Kompositionsverbindungen wohlgeformt sind. Erfüllt eine Komposition diese Anforderungen, lassen sich Modellkopplungen konstruieren, so dass alle erforderlichen Ports verbunden sind.

Stellen ein Modell $\mathscr{M}$ und Bindung $\mathscr{B}$ für eine Schnittstelle eine reflektierende Verfeinerung innerhalb einer vollständigen Kompostion dar, in der alle Schnittstellen durch Modelle und Bindungen bewahrend verfeinert werden, so sind in der durch *Couplings* konstruierten Menge an Modellkopplungen alle erforderlichen Ports von $\mathscr{M}$ verbunden.

Lemma 3. *Gegeben sei eine Kompostion (Sub, Con), eine Menge von Modellen $\{\mathscr{M}_i\}$, eine Menge von Bindungen $\{\mathscr{B}_i\}$ und eine Menge von Schnittstellendefinitionen $\{\mathscr{I}_i = drefI(s_i) | s_i \in Sub\}$. Des Weiteren sei $\mathscr{I}_j \in \{\mathscr{I}_i\}$, $\mathscr{M}_j \in \{\mathscr{M}_i\}$ sowie $\mathscr{B}_j \in \{\mathscr{B}_i\}$.*

$$\begin{aligned} &complete(Sub, Con) \wedge \mathscr{I}_j \succ_r (\mathscr{M}_j, \mathscr{B}_j) \wedge \forall \mathscr{I}_i. \mathscr{I}_i \succ_p (\mathscr{M}_i, \mathscr{B}_i) \Rightarrow \\ &\quad \forall ip \in required(\mathscr{M}_j). \exists (fm, fp, tm, tp) \in Couplings(Con, Sub). \quad fm = name_{\mathscr{M}_j} \wedge fp = ip \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee tm = name_{\mathscr{M}_j} \wedge tp = ip \end{aligned}$$

Beweis. Siehe Anhang A.3. □

So lässt sich formal prüfen, ob alle Anforderungen der implementierenden Modelle innerhalb einer Komposition erfüllt sind.

4.3.4 Korrekte Komponenten

Basierend auf der Verfeinerung von Schnittstellen in Modelle und vollständigen Kompositionen lässt sich nun formalisieren, wann eine (zusammengesetzte) Komponente alle Anforderungen erfüllt, um einsetzbar zu sein.

Definition 4.23. Eine Komponente $\mathscr{C}$ ist **korrekt**, ausgedrückt durch $correct(\mathscr{C})$, falls

1. Verfeinerung der Schnittstelle: $\mathscr{I} \succ (\mathscr{M}_{\mathscr{C}}, \mathscr{B}_{\mathscr{C}})$, mit $\mathscr{I} = drefI(if_{\mathscr{C}})$

2. Vollständige Komposition: $complete((Sub_{\mathscr{C}} \cup \{(\text{„this“}, if, \emptyset)\}, Con_{\mathscr{C}}))$
3. Korrekte Unterkomponenten: $\forall s \in Sub_{\mathscr{C}}.correct(instC(s))$
4. Azyklische Kompositionen: Keine Unterkomponente referenziert $\mathscr{C}$.

Eine Komponente ist korrekt, falls sie ihre Schnittstelle verfeinert. Korrektheit ist für zusammengesetzte Komponenten rekursiv definiert. Für die Überprüfung einer Komposition müssen also alle Subkomponenten bis zu den Blättern (atomare Komponenten) instantiiert werden. Atomare Komponenten sind bereits korrekt, falls $\mathscr{I} \succ (\mathscr{M}, \mathscr{B})$, da Bedingung 2 bis 4 automatisch erfüllt sind.

Theorem 1. *Gegeben sei eine zusammengesetzte Komponente $\mathscr{C}$. Falls $\mathscr{C}$ korrekt ist, sind in $model(\mathscr{C})$ alle erforderlichen Ports sämtlicher Untermodelle durch wohlgeformte Kopplungen mit kompatiblen Gegenstücken verbunden.*

Beweis. Für korrekte Komponenten ist sichergestellt, dass alle Unterkomponenten korrekt sind (sichergestellt durch Bedingung 3 von Definition 4.23) und alle Modellimplementierungen ihre Schnittstelle äquivalent verfeinern (erfüllt durch Bedingung 1 von Definition 4.23). Nach Lemma 3 konstruiert *Couplings* unter diesen Voraussetzungen für eine vollständige Komposition (gegeben durch Bedingung 2 von Definition 4.23) Modellkopplungen, so dass für jedes Untermodell M_d alle $p \in required(M_d)$ wohlgeformt verbunden sind. Bedingung 4 von Definition 4.23 stellt sicher, dass $model(\mathscr{C})$ terminiert. □

Die Schwierigkeit für Komponentendefinitionen besteht darin, dass eine Komponente im Rahmen der publizierten Parameter unterschiedlich instantiiert werden kann. Um Komponenten zu verifizieren, müsste für alle erlaubten Parameterkonfigurationen gezeigt werden, dass die instantiierte Komponente korrekt ist. Bei der Definition einer Komponente sollte daher abgewogen werden, wieviel Flexibilität sie benötigt.

4.4 Konkrete Repräsentation in XML

Die abstrakte Syntax und die Semantik der entwickelten Beschreibungseinheiten wurden formal definiert. Um Schnittstellen und Komponenten praktisch nutzen zu können, ist die Abbildung auf eine konkrete Repräsentation nötig. Ein Ergebnis des vorigen Kapitels war, dass XML-basierte Formate gegenüber proprietären IDLs zu bevorzugen sind. Die Definition einer XML-basierten Syntax für die entwickelten Beschreibungseinheiten nutzt im Folgenden XSD. Aus Gründen der Übersichtlichkeit werden Schemadefinitionen mittels UML-Klassendiagrammen veranschaulicht.

Es existieren unterschiedliche Ansätze XSD mittels UML darzustellen (Bernauer u. a. 2004). Motiviert durch den Wunsch, die UML-Definitionen eindeutig und automatisch auf XSD abbilden zu können, legen diese Ansätze unter anderem die Reihenfolge der Elemente fest und nutzen Stereotype. Da die abstrakte Syntax sämtlicher Beschreibungseinheiten bereits mengentheoretisch definiert ist, müssen die Klassendiagramme im Rahmen dieser Arbeit keine normative Funktion erfüllen. Entsprechend wird im Folgenden darauf verzichtet, die Reihenfolge der Elemente in der Visualisierung eindeutig festzulegen sowie die Art von Knoten durch Stereotype anzuzeigen. Zur Visualisierung der XSD-Dokumente werden im Folgenden XML-Elemente als Klassen und XML-Attribute als Attribute von Klassen dargestellt. Unterelemente werden per Aggregations-Beziehung zwischen Klassen repräsentiert. Alle aggregierten Elemente sind Teil eines XSD-Dokuments. Eine Klasse, die nicht aggregiert wurde, repräsentiert dementsprechend das Wurzelelement eines XSD-Dokuments. Die konkreten und vollständigen XSD-Dokumente befinden sich in Anhang B.

Im Metamodell wurde die abstrakte Syntax von Rollen, Schnittstellen, Modellen, Komponenten und Simulationsläufen in kompakter Form definiert. Für die konkrete Syntax werden Beschreibungen mit z.T. aussagekräftigeren Bezeichnern versehen und ausgewählte Elemente unterstrukturiert.

Menge im Metamodell	Realisiert durch
$\mathbb{B}$	`xsd::boolean`
$\mathbb{N}$	`xsd::int`
QName	`xsd::QName`
String	`xsd::NCName`, `xsd:string`
UVal	`xsd:any`

Tabelle 4.1: Abbildungen grundlegender Mengen auf XML-Schema

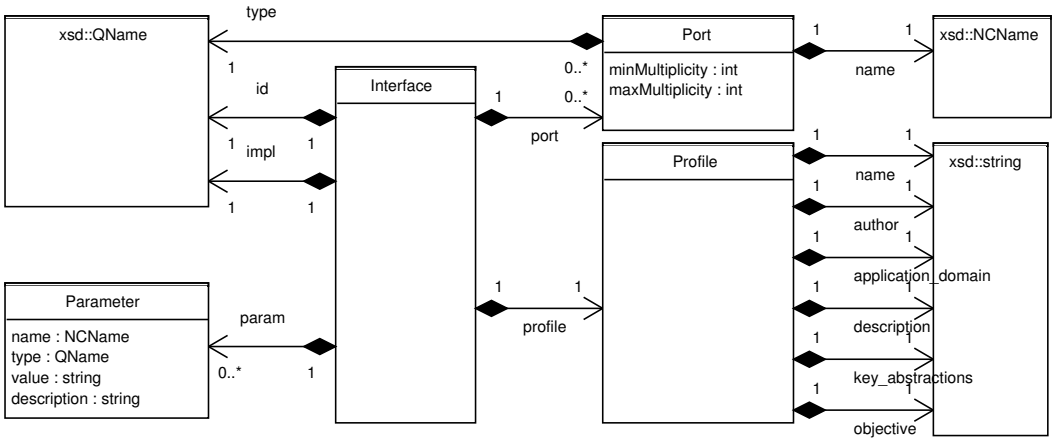

Abbildung 4.12: Struktur von Schnittstellenbeschreibungen

Einfache Mengen, wie $\mathbb{N}$ und qualifizierte Namen, werden direkt auf Datentypen von XSD, entsprechend Tabelle 4.1, abgebildet.

4.4.1 Öffentliche Beschreibungen

Abbildung 4.12 visualisiert die Struktur von XML-Dokumenten für Schnittstellendefinitionen als UML-Klassendiagramm. Die vollständige Syntaxdefinition mittels XML-Schema befindet sich in Anhang B.2. Dem Metamodell entsprechend, besteht die Definition aus einem eindeutigen Identifizierer, einer Menge von Parametern und einer Menge von Ports, die mittels qualifizierter Namen (xsd:QName) auf Rollendefinitionen verweisen. Zusätzlich zur abstrakten Syntax von Schnittstellen, können Schnittstellendokumente ein Element `profile` enthalten, in dem Metadaten ähnlich zu denen von BOM, angegeben werden können. Da sich diese Daten vorrangig informal analysieren lassen, fanden sie keinen Eingang ins Metamodell.

Beispiel 4.11. *Abbildung 4.13 zeigt die XML-Version der Schnittstellendefinition für die Komponente `Flooding`. Die Schnittstelle enthält eine Profilbeschreibung und bietet einen Parameter zur Konfiguration der zeitlichen Obergrenze bei Dienstsuchen an. Die Rolle `ServiceProv` wird über den Port `protocol` bereitgestellt.*

Schnittstellen referenzieren Rollendefinitionen zur Typisierung von Ports. Dies geschieht mittels qualifizierter Namen. Rollen sind wiederum in eigenen Dokumenten definiert. Analog zu Schnittstellen veranschaulicht Abbildung 4.14 die konkreten Syntaxstrukturen für Rollen[6]. Zusätzlich zur

[6]Die vollständige Definition in XML-Schema ist in Anhang B.1 gelistet.

```
<interface xmlns="http://www.informatik.uni-rostock.de/cosa/publici"
           xmlns:flooding="unihro/diane/com/flooding">
  <id>flooding:interface</id>
  <profile>
    <name>Flooding</name>
    <application_domain> MANET simulation</application_domain>
    <description>A very simple protocol for searching services in MANETs. Services
        are searched by sucessively broadcasting requests to neighbouring nodes.
       No overlay structure is maintained.</description>
    <objective>Service trading</objective>
    <key_abstractions>none</key_abstractions>
    <author>Mathias Roehl</author>
  </profile>
  <param name="wait4SearchT" type="http://www.w3.org/2001/XMLSchema:double"
      value="3.0" description="time to wait for service search results
      [seconds]"/>
  <port minMultiplicity="0" maxMultiplicity="1">
    <name>protocol</name>
    <type>flooding:ServiceProv</type>
  </port> ...
  <impl>flooding:component</impl>
</interface>
```

Abbildung 4.13: Schnittstellendefinition für Flooding

Tupeldefinition von Rollen in der abstrakten Syntax, werden in der konkreten Syntax die Elemente *description* und *types* zur Strukturierung genutzt. Der Import von Typdefinitionen ist so, analog zu WSDL, als abgegrenzter Teil einer Schnittstellendefinition definierbar.
Im vorigen Kapitel wurden die Vorteile von XSD als Standard zur Definition der syntaktischen Dimension von Web Service vorgestellt. XSD wird jetzt, neben der Nutzung für die Definition der konkreten Syntax des Metamodells selbst, für Typdefinitionen der Rollen genutzt. Ähnlich zu WSDL können Schemadefinitionen in Rollen importiert werden. Während in WSDL-Dokumenten beliebige Schemasprachen zum Einsatz kommen können, beschränken Rollenbeschreibungen sich auf den Import von Typdefinition im W3C-Standard XSD. Dies vereinfacht die Kompatibilitätsprüfung von Rollen, da nicht Typdefinitionen in unterschiedlichen Schemasprachen miteinander verglichen werden müssen. Anders als in WSDL werden Rollen und Schnittstellen in jeweils eigenen XML-Dokumenten definiert.

XSD bringt eine Reihe einfacher Datentypen mit und erlaubt, zusammengesetzte Datentypen aus einfachen zu konstruieren. Mit XSD lassen sich komplexe Typen wahlweise lokal definieren oder bereits existierende Definitionen über URIs referenzieren. Weiterhin bietet XML Anknüpfungspunkte zu Spezifikationen jenseits der syntaktischen Ebene. So ist mittels SAWSDL-Attributen der Verweis auf semantische Definitionen innerhalb von Ontologien möglich. Damit wird die Brücke zu semantischen Beschreibungssprachen wie RDF geschlagen.

Im Element *role* ist die eigentliche Definition der Rolle als Menge von gerichteten Ereignisports enthalten. Dabei werden deklarierte Ports durch Verweise auf importierte Schemata getypt, die ihrerseits entsprechend SAWSDL über `anyAttribute` auf semantische Definitionen verweisen können.

Beispiel 4.12. *Abbildung 4.15 listet die Definition der Rolle* `ServiceProv` *aus Beispiel 4.2 in XML. Die Schemadefinitionen des Namensraums „unihro/diane/base/service“ werden importiert und dazu genutzt, die Möglichkeiten zum Versenden von* `Call`*-Ereignissen und zum Empfangen von* `Response`*-Ereignissen bekanntzugeben.*

Schnittstellen beschränken sich bisher auf die Deklaration des statischen Teils von Modellimplemen-

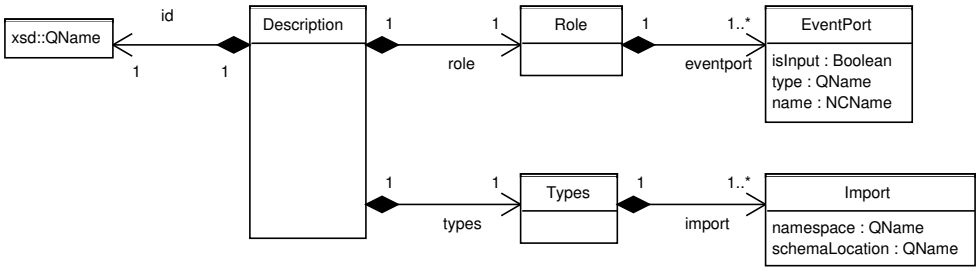

Abbildung 4.14: Dokumentstruktur für Rollendefinitionen in XML

```
<description xmlns="http://www.informatik.uni-rostock.de/cosa/role"
             xmlns:flooding="unihro/diane/com/flooding">
  <id>flooding:ServiceProv</id>
  <types>
    <import namespace="unihro/diane/base/service"/>
  </types>
  <role xmlns:serv="unihro/diane/base/service">
    <eventport name="call" isInput="true" type="serv:Call"/>
    <eventport name="response" isInput="false" type="serv:Response"/>
  </role>
</description>
```

Abbildung 4.15: Definition der Rolle ServiceProv auf Basis importierter Typen

tierungen. Sie umfassen bisher keine Deklaration des Interaktionspotentials von Modellen. Die Unterteilung von Schnittstellen in Rollen verträgt sich jedoch gut mit dynamischen Analysetechniken, die von internen Aktionen abstrahieren, bspw. mittels schwacher Bisimulation. XML-Dokumente für Rollendefinitionen lassen sich, analog zu den semantischen Annotationen, um Verweise auf dynamische Beschreibungen erweitern. Eine konkrete Syntax zur Deklaration des dynamischen Aspekts von Modellen zu finden, stellt jedoch eine wesentliche Herausforderung dar[7].

4.4.2 PDEVS-Modelle

Für DEVS wurden jüngst Vorschläge zur XML-Repräsentation unterbreitet (Janoušek u. a. 2006; Martín u. a. 2007), die jedoch noch nicht standardisiert sind und das Problem der Komposition nicht explizit adressieren. Das größte Manko ist bisher, dass noch keine befriedigende Lösung zur plattformunabhängigen Spezifikation der Zustandsüberführungsfunktionen atomarer Modelle existiert. Die XML-Formate schränken die Ausdrucksstärke in den Funktionsdefinitionen entweder drastisch ein (Martín u. a. 2007) oder verwenden XML-Formate, die eine XML-Kodierung von Programmiersprachen darstellen, bspw. mittels JavaML, (Janoušek u. a. 2006).

Im Rahmen der Anwendung DIANE werden komplexe Zustandsüberführungen benötigt. Aus diesem Grund wurde ein eigenes XML-Format für DEVS-Modelle entwickelt, in dem Zustandüberführungen als Implementierungen in Java eingebunden werden können (Röhl u. Uhrmacher 2005). Java-Implementierungen werden aus den XML-Dokumenten über voll qualifizierte Klassennamen referenziert. Auf die Definition von Zustandsüberführungsfunktionen in Java wird in Abschnitt 5.4 näher eingegangen.

Um Modelldefinitionen im Rahmen der entwickelten Komponentenplattform nutzen zu können,

[7] siehe den Abschnitt 3.3.2 zur Deklaration und Analyse des dynamischen Aspekts von Web Services

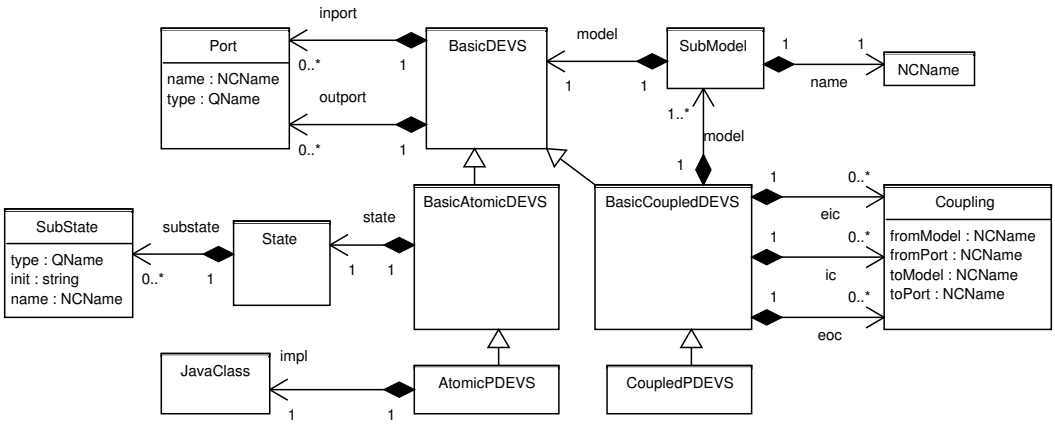

Abbildung 4.16: Struktur von XML-Dokumenten zur Definition von PDEVS-Modellen

```
<atomic xmlns="http://www.informatik.uni-rostock.de/cosa/model/pdevs"
        xmlns:serv="unihro/diane/base/service"
        xmlns:trans="unihro/diane/base/transport"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <inport type="serv:Call">call</inport>
  <inport type="trans:Message">receive</inport>
  <outport type="trans:Message">send</outport>
  <outport type="serv:Response">response</outport>
  <state>
    <substate type="xsd:double" init="0">wait4SearchT</substate>
    <substate type="xsd:string" init="0">phase</substate> ...
  </state>
  <impl>unihro.diane.com.motion.v1.Model</impl>
</atomic>
```

Abbildung 4.17: Modellimplementierung für die Komponente Flooding

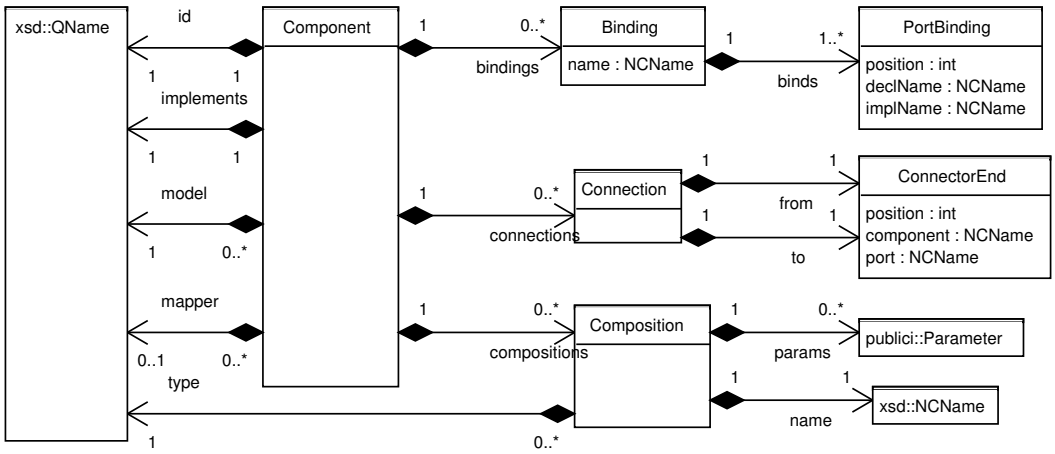

Abbildung 4.18: Syntax für Komponentenbeschreibungen

müssen die Angebote und Anforderungen eines Modells in Schnittstellen ausdrückbar sein. Im Metamodell wird dies formal mittels der Verfeinerungsrelation $\succ$ erfasst. Um $\succ$ für konkrete Modellrepräsentationen prüfen zu können, verweisen die Definitionen von Eingabe- und Ausgabeports auf XSD-Typen.

Die Struktur von XML-Dokumenten zur Definition von PDEVS-Modellen zeigt Abbildung 4.16. Die Syntax für PDEVS-Modelle ist unabhängig von der entwickelten Komponentenplattform verwendbar. Modelle können darin eigenständig definiert werden und sind auch ohne explizite Schnittstellenbeschreibungen verwendbar. Automatisiert komponieren lassen sie sich jedoch erst, wenn sie mit den entwickelten Schnittstellenbeschreibungen assoziiert werden.

Beispiel 4.13. *Abbildung 4.17 zeigt die Definition des atomares PDEVS-Modell für das Protokoll Fluten aus Beispiel 2.1 in XML.*

4.4.3 Komponenten

Abbildung 4.18 visualisiert die Struktur für konkrete Komponentendefinitionen. Eine Komponente verfügt selbst über einen qualifizierten Namen (*id*) und verweist auf ihre Schnittstelle mittels eines qualifizierten Namens. Des Weiteren werden Modellimplementierungen, Konfiguratoren und Unterkomponenten jeweils über einen qualifizierten Namen referenziert. Komponenten können eine Menge von Bindungen (*Binding*), Kompositionsverbindungen (*Connection*) und Unterkomponenten (*Composition*) umfassen. Bindungen sind nach Namen von Schnittstellenports gruppiert. Ein Element vom Typ *Binding* enthält eine Menge von Portbindungen (*Portbinding*) für einen Schnittstellenport. Eine Kompositionsverbindung umfasst zwei Konnektoren (*ConnectorEnd*), die jeweils auf Ports von Unterkomponenten verweisen. Kompositionen (*Composition*) besitzen einen Namen und eine Menge von Parametern, die auf die Unterkomponenten, zum Zeitpunkt ihrer Instantiierung, angewendet werden.

Für die konkrete Syntax von Komponenten ist zu beachten, dass Komponenten in deklarativer Form vorliegen. Die Komponenten sind selbst nicht direkt ausführbar. Komponenten beinhalten jedoch eine Funktion κ, die den Inhalt der XML-Definition manipulieren können muss. Es wird eine konkrete Syntax für κ benötigt, die möglichst nicht plattformspezifisch ist. XML-Daten zu manipulieren und zu transformieren, ist die Domäne von *XSL Transformations* (XSLT). XSLT ist selbst XML-basiert und standardisiert (W3C 1999).

```
<component xmlns="http://www.informatik.uni-rostock.de/cosa/component"
           xmlns:node="unihro/diane/com/node">
  <id>node:component</id>
  <implements>node:interface</implements>
  <mapper>node:NodeMapper</mapper>
  <model>node:model</model>
  <binding name="transport">
    <bind declName="in" implName="msgIn" position="1"/>
    <bind declName="out" implName="msgOut" position="1"/>
  </binding>
  <composition xmlns:flooding="unihro/diane/com/flooding">
    <type>flooding:interface</type>
    <name>overlay</name>
  </composition> ...
  <connection>
    <from component="this" port="transport"/>
    <to component="overlay" port="transport"/>
  </connection> ...
</component>
```

Abbildung 4.19: Die zusammengesetzte Komponente Node

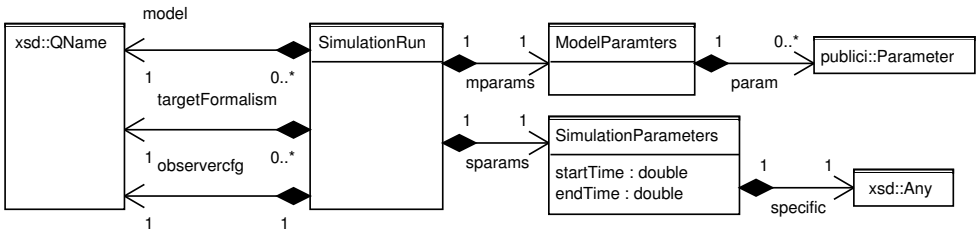

Abbildung 4.20: Beschreibung von Simulationsläufen

Beispiel 4.14. *Abbildung 4.19 listet die XML-Definition der zusammengesetzten Komponente* `Node`*, die zwei Unterkomponenten enthält (visualisiert in Abbildung 4.5). Die Komposition des Protokolls* `Flooding` *ist im XML-Listing aufgeführt. Für die Nutzerkomponente verläuft die Komposition analog. Der Schnittstellenport* `overlay` *der Unterkomponente Flooding wird durch eine Kompositionsverbindung an einen eigenen ausgestellten Port delegiert. Dementsprechend müssen die beiden Ereignisports der Rolle* `TransportReq`*, wie in der Komponente Flooding, an implementierende Modellports (msgIn, msgOut) gebunden werden (siehe Beispiel 4.4).*

4.4.4 Simulationsläufe

Ein Simulationslauf verweist auf eine Modellkomponente mittels eines qualifizierten Namens. Für diese Modellkomponente können eine Menge von Parameterwerten definiert werden. Des Weiteren gibt ein Simulationslauf eine Simulationsplattform sowie einen Zielformalismus vor und verweist auf eine externe Entität, die das Simulationsmodell mit Beobachtern instrumentiert. Abbildung 4.20 gibt einen Überblick über die XSD-Struktur (siehe Anhang B.4).

Nicht alle Simulationsparameter sind plattformunabhängig spezifizierbar. So lassen sich zu verwendende Algorithmen und Datenstrukturen — ob ein Modell bspw. sequentiell oder parallel ausgeführt werden soll oder ob spezielle Ereignisschlagen zum Einsatz kommen sollen — nur im Rahmen der Möglichkeiten eines speziellen Werkzeugs auswählen. Darum wird die Struktur von Simulationsparametern nicht vorgegeben, sondern diese sind in freier Form mittels `xsd:Any` definierbar.

```
<experiment xmlns="http://www.informatik.uni-rostock.de/cosa/experiment"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:exp="unihro/diane/experiment"
  xmlns:net="unihro/diane/com/manet">
  <id>exp:experiment</id>
  <model>net:interface</model>
  <mparams>
    <param name="nodes" type="xsd:int" value="100" description="number of network
        nodes to be simulated"/>
    <param name="protocol" type="xsd:string" value="Flooding" description="Service
         trading protocol to be used for each node"/>
    <param name="user" type="xsd:string" value="SimpleUser" description="User
        model to be used inside each node"/>
  </mparams>
  <targetFormalism>
    http://www.informatik.uni-rostock.de/cosa/model/dynpdevs
  </targetFormalism>
  <platform>http://www.informatik.uni-rostock.de/cosa/jamesii</platform>
  <observercfg>unihro.diane.experiments.v05.ObsVis</observercfg>
  <sparams>
    <startTime>0.0</startTime>
    <endTime>32400</endTime>
    <processor>
      simulator.pdyndevsflatsequential.DynDEVSSequentialProcessorFactory
    </processor>
  </sparams>
</experiment>
```

Abbildung 4.21: Definition eines Simulationslaufs

Beispiel 4.15. *Abbildung 4.21 definiert einen Simulationslauf mit der Komponente Manet als Wurzelmodell (*model*). Für Manet werden die Anzahl der Netzwerkknoten (*param „nodes“*) und das in jedem Knoten zu verwendende Nutzermodell (*param „user“*) sowie Vermittlungsprotokoll (*param „protocol“*) vorgegeben. Als Zielformalismus dient PDEVS auf der Plattform James II. Ein qualifizierter Klassenname definiert den Konfigurator für die Beobachtungen (*observercfg*). Als Simulationsparameter (*sparams*) sind die Start- und Endezeit sowie ein spezieller sequentieller Simulator (*processor*) angegeben.*

4.5 Zusammenfassung

Im Rahmen dieses Kapitels wurde eine Plattform für Modellkomponenten konzipiert und formal definiert. Mit den Beschreibungsmitteln der Plattform können Modelle räumlich und zeitlich verteilt entwickelt, als XML-Dokumente in Datenbanken verwaltet sowie modular-hierarchisch komponiert werden.

Die entwickelten Beschreibungsmittel vereinen zwei Vorteile existierender Kompositionsansätze, die bisher so nicht kombinierbar sind. Modular-hierarchische Modellierungsformalismen, die strikt zwischen Modelldefinitionen und Simulationsalgorithmen trennen und flexible Modellierungskonstrukte bereitstellen, lassen sich im Rahmen der entwickelten Kompositionsplattform als Definitionssprachen für Modellimplementierungen nutzen. Gleichzeitig sind Schnittstellenbeschreibungen vollständig von den Modelldefinitionen separiert und erlauben, Kompositionen allein auf Grundlage öffentlicher Beschreibungen zu analysieren.

Die syntaktischen Strukturen der entwickelten Beschreibungsmittel orientieren sich an den Ausdrucksmöglichkeiten von Kompositionsstrukturen der UML und der SysML. Im Gegensatz zu diesen, sind die hier vorgestellten Kompositionsstrukturen auf die ereignisorientierte Modellierung

zugeschnitten und wurden mit einer formalen Semantik ausgestattet.

Auf Grundlage der expliziten Semantikdefinition für Komponenten konnten Eigenschaften wie die Kompatibilität verbundener Rollen, die Verfeinerung von Schnittstellen in Modellimplementierungen und die Korrektheit von Komponenten formal gefasst werden. Die Definition von bewahrender und reflektierender Verfeinerung ermöglicht, sowohl die Anforderungen an Komponenten als auch die Kontextanforderungen von Komponenten formal zu prüfen. Sind Komponentendefinitionen korrekt, lassen sich aus ihnen wohlgeformte Simulationsmodelle ableiten.

Die konkrete Syntax der entwickelten Beschreibungsmittel wurde im W3C-Standard XSD als XML-basiertes Austauschformat definiert. Darüber hinaus bildet, in Anlehnung an die Architektur von Web Services, XSD die Grundlage für Typdefinitionen in Schnittstellen, die sich so nahtlos mit semantischen Annotationen versehen lassen.

Um in der Praxis von der entwickelten Komponentenplattform zu profitieren, bedarf es eines konkreten Werkzeugs, das eine Komposition instantiiert und analysiert, sowie ein ausführbares Simulationsmodell für eine Zielplattform erstellt.

5 Das Kompositionswerkzeug CoMo

Die im vorigen Kapitel entwickelte Komponentenplattform definiert Mittel zur Spezifikation von Modellkomponenten. Um mit den Beschreibungsmitteln erstellte Rollen, Schnittstellen und Komponenten praktisch nutzbar zu machen, bedarf es einer Realisierung der Plattform. Im Folgenden wird die Realisierung der Komponentenplattform im Rahmen des Kompositionswerkzeugs CoMo beschrieben. Implementiert in der Programmiersprache Java kann CoMo

- auf Definitionen in XML mit Java zugreifen und sie für eine zielgerichtete Verarbeitung nutzbar machen,
- Komponenten aus XML-Dokumenten mit heterogenen Modelldefinitionen instantiieren, Kompositionen analysieren und in PDEVS-Modelle abbilden,
- Typdefinitionen in XSD in Hinblick auf Datenbankanfragen und Kompatibilitätschecks automatisiert vergleichen
- ein ausführbares Modell für eine konkrete Simulationsplattform ableiten.

Den Ausgangspunkt zur Umsetzung von CoMo bildet die, mittels XML-Schema definierte, Syntax der entwickelten Beschreibungsmittel.

5.1 Verarbeiten von schemagebundenen XML-Daten

Klassische Ansätze zur Verarbeitung von XML-Daten in Software basieren auf dem Document Object Model (DOM) oder der Simple API for XML (SAX). Mit DOM (W3C 2004a) wird ein XML-Dokument als Ganzes in einen Objektbaum einer Programmiersprache transformiert. Der Baum reflektiert dabei die Struktur und den Inhalt eines XML-Dokuments direkt. Auf Datenelemente kann in dem Baum beliebig zugegriffen werden. SAX arbeitet hingegen ereignisbasiert (Megginson 2008). Mit SAX bewegt man sich sequentiell vom Anfang eines Dokuments zu seinem Ende. Für jeden gefundenen Knoten im Dokument wird ein Ereignis produziert. Objektstrukturen müssen auf Basis dieser Ereignisse manuell konstruiert werden.

Sowohl DOM als auch SAX erlauben XML-Daten innerhalb von Anwendungen zu verarbeiten. Der Zugriff auf Daten geschieht jedoch nahe an der Grundstruktur von XML, auf der Basis von verschachtelten Elementen und ihrer Attribute. Für die Realisierung von Softwaresystemen ist es jedoch wünschenswert, Programmlogik auf der Abstraktionsebene der entsprechenden Anwendungsdomäne zu implementieren und nicht auf Ebene des zugrundeliegenden Speicherformats, wie DOM und SAX ihn bereitstellen. Bezogen auf Komponentendefinitionen möchte man mit Schnittstellen, Rollen, Typen und Parametern arbeiten. Letzterer Ansatz zur Verarbeitung von XML wird datenzentriert genannt (McLaughlin 2002).

Einen datenzentrierten Ansatz zur Verarbeitung von XML-Inhalten in Softwaresystemen bietet das Binden von Verarbeitungslogik an Schemadefinitionen, genannt *data binding*, (Brodkin 2001; Bourret 2005). Gegeben eine Menge an Schemadefinitionen generiert ein Übersetzer Quellcode in einer bestimmten Programmiersprache, der die Struktur der Schemata reflektiert. Mittels des generierten Quellcodes lassen sich XML-Daten im Speicher bearbeiten und automatisch von XML-Dokumenten in Objektstrukturen der jeweiligen Programmiersprache und zurück konvertieren. Fehleranfällige manuelle Implementierungsarbeiten zum Lesen und Schreiben von XML-Dokumenten können so entfallen.

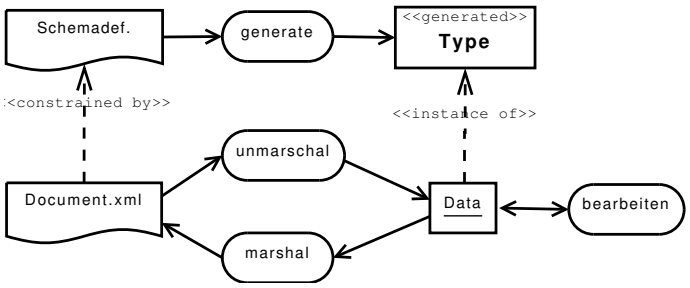

Abbildung 5.1: Bindung von XML-Daten an Schemadefinitionen

Die generelle Vorgehensweise zum Arbeiten mit XML-Daten auf Basis von XSD verdeutlicht Abbildung 5.1 (Röhl u. Uhrmacher 2005). *Marshalling* bezeichnet dabei die Tranformation von Objektbäumen in XML-Dokumente. XML-Dokumente zu lesen und Objektstrukturen mittels generierter Klassen im Arbeitsspeicher anzulegen, wird als *Unmarshalling* bezeichnet. Im Zuge des Unmarshallings können XML-Daten gegen Schemadefinitionen geprüft werden. Marshalling und Unmarshalling stehen in den generierten Klassen automatisch zur Verfügung.

Fehler in Instanzdokumenten, ob bspw. bestimmte Elemente oder Attribute vorhanden sind, werden mit DOM und SAX erst zur Laufzeit erkannt. Mit der Bindung von Implementierungen an XSD wirken sich Änderungen am Schema hingegen direkt auf die generierten Klassen aus. Typänderungen ziehen dann Fehler beim Übersetzen verarbeitender Klassen nach sich und können somit leichter erkannt werden. Die Reduzierung von manueller Implementierungsarbeit und automatischer Typsicherheit beim Arbeiten mit XML-Daten wirkt sich besonders dann vorteilhaft aus, wenn sich Schemata in der Entwicklung befinden, wie es bspw. für den Formalismen DEVS zutrifft.

Um Datenverarbeitung an Schemata zu binden, existieren Werkzeuge für unterschiedliche Programmiersprachen, bspw. für Java (Ort u. Mehta 2003) und C++ (Ware 2005). Für CoMo findet die *Java Architecture for XML Binding* (JAXB) von Sun (2005) Verwendung, das seit Version 6 Bestandteil der Standardedition von Java ist (Sun 2008).

Beispiel 5.1. *Gegeben die Schemadefinition für Modellkomponenten, die im vorigen Kapitel entwickelt wurde*[1] *und in Anhang B.3 vollständig gelistet ist, generiert JAXB (Sun 2005) Java-Klassen, wie in Abbildung 5.2 dargestellt. Die Beziehungen zwischen den Klassen reflektieren die Struktur das Schemas für Komponentendefinitionen direkt. Einfache XSD-Typen, wie Zeichenketten und qualifizierte Namen, sind auf entsprechende Standardklassen von Java abgebildet. Komplexe XSD-Typen resultieren in neuen Klassendefinitionen, die Struktur, Abhängigkeiten und Vererbungsbeziehungen des Schemadokuments bewahren.*

Die konkrete Syntaxdefinitionen für die Beschreibungsmittel der entwickelten Plattform dienen somit als Vorlage, um Quellcode in Java zu generieren, mittels dem Rollen, Schnittstellen, Komponenten, PDEVS-Modelle und Simulationsläufe aus XML gelesen und in XML geschrieben werden können. Für CoMo wurden mit JAXB Java-Klassen für sämtliche Schemadefinitionen von Anhang B generiert.

Flexibler Umgang mit generierten Klassen

Die generierten Klassen können direkt zum Lesen, Editieren und Schreiben von XML-Dokumenten genutzt werden, ohne sich mit den Details der XML-Codierung auseinandersetzen zu müssen. Damit erfüllen sie aber nur einen Teil der Anforderungen zum Verarbeiten von XML-Daten.

Schemasprachen besitzen eine jeweils spezifische Ausdruckstärke. Es lassen sich evtl. nicht alle Einschränkungen, die man für XML-Daten spezifizieren möchte, ausdrücken. Bspw. erlaubt XSD

[1] vgl. Abbildung 4.18

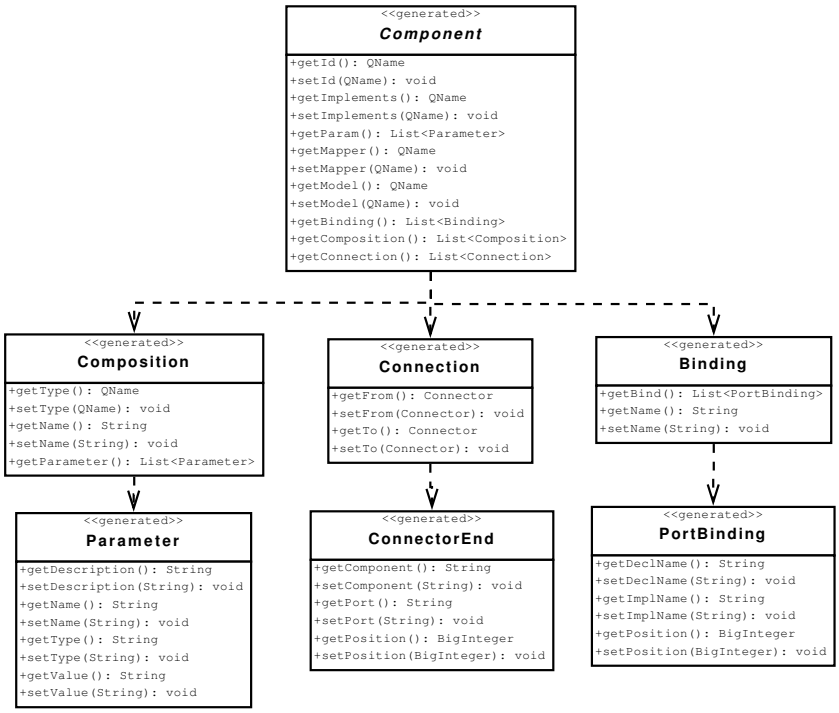

Abbildung 5.2: Aus der Schemadefinition für Komponenten generierte Klassen

nur einfache Einschränkungen zwischen Mengen von Elementen. Die Korrektheit von Daten unterliegen im Metamodell aber zusätzlichen Bedingungen. Möglich ist auch, dass eine vorgegebene Schemadefinition, bspw. ein Standard, nicht alle Möglichkeiten einer Schemasprache ausreizt.

In Hinblick auf die generierten Klassen stellt sich die Frage, wo die Logik zum Überprüfen zusätzlicher Bedingungen implementiert werden kann. Direkte Änderungen der generierten Klassen gehen zu dem Zeitpunkt verloren, an dem eine veränderte Schemadefinition in der erneuten Generierung der Klassen resultiert. Nach jeder Änderung an Schemadefinitionen müssten die generierten Klassen, erneut um die zusätzliche Funktionalität bereichert werden. Damit ginge ein zentraler Vorteil des generativen Ansatzes verloren.

Umgehen lässt sich dieses Dilemma, indem die gewünschte Funktionalität in einer separaten Klasse implementiert wird. Die einfachste Möglichkeit dazu ist, von der generierten Klasse zu erben und entsprechende Methoden zu überschreiben bzw. neue hinzuzufügen. Da Java jedoch keine Mehrfachvererbung von Klassen erlaubt, schränkt dies die Flexibilität beim Erstellen der zusätzlichen Klassen stark ein.

Flexibler ist es, eine eigenständige Klassenhierarchie zu erstellen, die strukturell der generierten Klassenhierarchie entspricht. Es werden zwei Entwurfsmuster (Gamma u. a. 1995) genutzt, um generierte Klassen und zusätzliche Implementierungen zu verknüpfen. Adapter verstecken die JAXB-Implementierung hinter einer Schnittstelle, die auf den Verarbeitungskontext zugeschnitten ist. Dekorierer integrieren Programmlogik, um Bedingungen zu überprüfen, die über die schemaeigenen hinausgehen.

Beispiel 5.2. *Abbildung 5.3 verdeutlicht das Zusammenspiel einer generierten Klasse mit einer angepassten Schnittstelle, einem Adapter und einem Dekorierer. Die Schnittstelle* `IComData` *stellt Methoden bereit, die auf die Nutzung in der Konfigurationsphase zugeschnitten sind. IComData erlaubt Bindungen, Kompositionen und Kompositionsverbindungen zu Komponenten hinzuzufügen.*

Ein Ausschnitt der Implementierung des Dekorierers für Komponentendaten ist in Abbildung 5.4

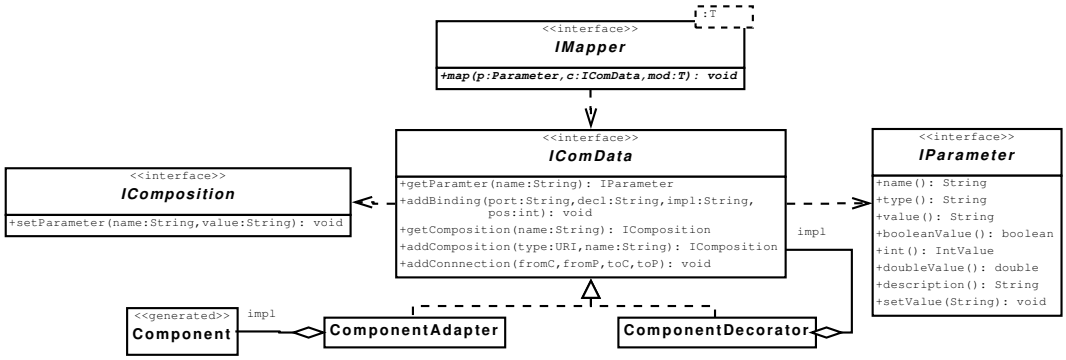

Abbildung 5.3: Erweiterung generierten Codes mittels Adaptern und Dekorierern

```
public class ComDataDecorator implements IComData {
  ...
  public void addConnection(String fromC, String fromP, int fromPos,
                            String toC,String toP, int toPos) {
    IComposition fromComponent = impl.getComposition(fromC);
    IComposition toComponent = impl.getComposition(toC);
    if ((fromComponent != null && toComponent != null)
        || (fromC == "this" && toComponent != null)
        || (toC == "this" && fromComponent != null)) {
      impl.addConnection(fromC, fromP, fromPos, toC, toP, toPos);
    } else {
      throw new InternalError(...);
    }
  }
}
```

Abbildung 5.4: Überprüfen erweiterter Einschränkungen im Dekorierer

```
public class ComponentAdapter implements IComData {
  public void addConnection(String fromC, String fromP, int fromPos,
                            String toC, String toP, int toPos) {
    Connector fromCon = factory.createConnector();
    fromCon.setComponent(fromC);
    fromCon.setPort(fromP);
    fromCon.setPosition(BigInteger.valueOf(fromPos));
    Connector toCon = factory.createConnector();
    toCon.setComponent(toC);
    toCon.setPort(toP);
    toCon.setPosition(BigInteger.valueOf(toPos));
    Connection newCon = factory.createConnection();
    newCon.setFrom(fromCon);
    newCon.setTo(toCon);
    comImpl.getConnection().add(newCon);
  }
```

Abbildung 5.5: Adaption generierter Klassen auf anwendungsorientierte Schnittstellen

```
public class NodeMapper implements IMapper<Object> {
  public final void map(Parameter params, IComData com, Object modelData) {
    // compose user according to parameter <user>
    QName userType = params.getParameter("user").qname();
    IComposition user = com.addComposition(userType, "user");

    // delegate <id> parameter to user model
    String idStr = com.getParameter("id").value();
    user.setParameter("id", idStr);

    // compose trading protocol according to parameter <overlay>
    QName protocolType = com.getParameter("overlay").qname();
    com.addComposition(protocolType, "overlay");
  }
}
```

Abbildung 5.6: Implementierung eines Konfigurators für die Komponente Node

gelistet. Nur falls die referenzierten Subkomponenten tatsächlich existieren, wird eine Kompositionsverbindung hinzugefügt.

Abbildung 5.5 zeigt einen Ausschnitt des entsprechenden Adapters. Mittels einer von JAXB generierten Fabrik werden dort die generierten Klassen instantiiert und initialisiert.

Adapter ermöglichen im Zusammenspiel mit Dekorierern, die Details der XML-Anbindung vollständig hinter einer anwendungsspezifischen Schnittstelle zu verbergen. Damit können die Vorteile des generativen Ansatzes mit der Flexibilität und Ausdrucksstärke von Programmiersprachen verknüpft werden (Röhl u. Uhrmacher 2005).

Konfiguratoren

Modelle für konkrete Einsatzkontexte konfigurierbar zu gestalten, stellt für plattformunabhängige Modellkomponenten eine besondere Herausforderung dar. Komponentenimplementierungen benötigen zu diesem Zweck einen ausführbaren Teil, der entsprechend einer konkreten Parametrisierung, die interne Struktur einer Komponentendefinition anpasst.

Für die Instantiierung von Komponenten bedarf es der konkreten Ausgestaltung von Konfiguratoren. Konfiguratoren verändern XML-Daten von Komponenten und Modellen in Abhängigkeit von Parametern. Im Metamodell wurden Konfiguratoren als ausführbare aber plattformunabhängige Funktionen konzipiert. XSLT (W3C 1999) erlaubt Konfiguratoren in plattformunabhängiger Weise als XML-Dokument umzusetzen. Gebundene Datenobjekte lassen sich jedoch auch nutzen, um XSLT-Verarbeitung mit Java zu imitieren.

Konfiguratoren nutzen dekorierte Datenobjekte, so dass alle Änderungen auf den gebundenen XML-Daten stattfinden. In Abbildung 5.3 ist die Schnittstelle `IMapper` dargestellt, die von Komponentenentwicklern implementiert werden kann, um Konfiguratoren für eine bestimmte Komponente in Java zu realisieren.

Beispiel 5.3. *Die Implementierung des Konfigurators der Node-Komponente ist in Abbildung 5.6 gelistet. Der Parameter* `user` *wird ausgelesen und eine Unterkomponente entsprechenden Typs eingefügt. Der Parameter* `id` *wird an den Nutzer delegiert. Eine Protokollkomponente wird analog zur Nutzerkomponente auf Basis eines gesetzten Parameters komponiert.*

5.2 Instantiieren von Komponenten

CoMo wird nun als erweiterbares Rahmenwerk entworfen, das auf die Komposition von Modellkomponenten aus XML-Dokumenten zugeschnitten ist.

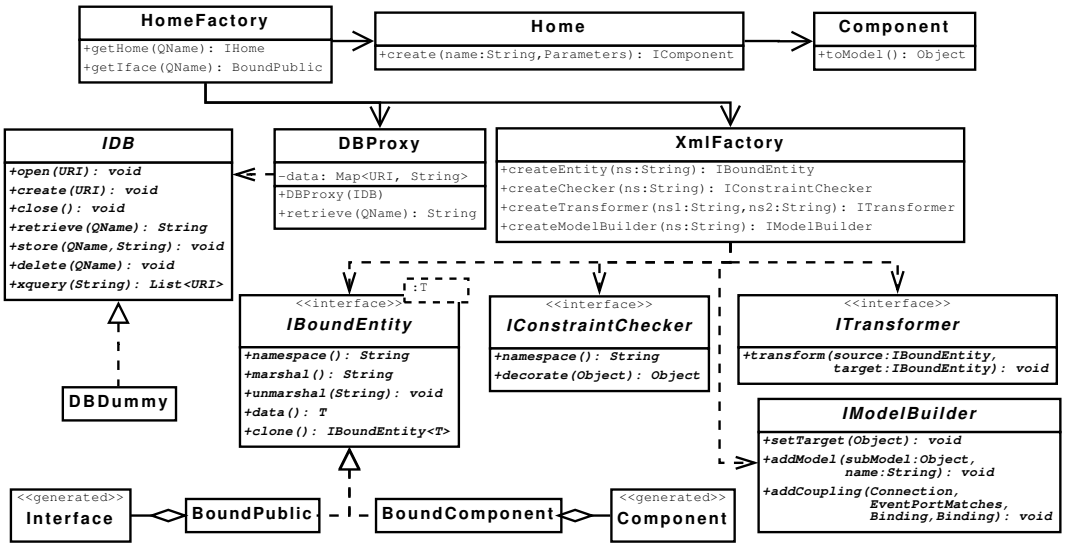

Abbildung 5.7: Grundlegende Abstraktionen von CoMo und ihre Beziehungen

Um Modellkomponenten entkoppelt voneinander entwickeln zu können, sind Komponenten durch XML-Dokumente definiert und über URI-basierte Referenzen miteinander verknüpft. Zum Zeitpunkt der Komposition müssen Komponenten konfiguriert, referenzierte Unterkomponenten instantiiert und die Verbindungen zwischen Ports von Unterkomponenten auf Kompatibilität überprüft werden.

Entsprechend obiger Diskussion werden in CoMo sämtliche Dateneinheiten an Schemata gebunden, gegen die sie geprüft werden können. Die Beziehung zwischen Daten und Schemata wird in CoMo durch die zentrale Schnittstelle `IBoundEntity` ausgedrückt. Sie stellt den Zugriff auf die Java-Repräsentation von Daten mittels der Methode *data()* bereit. Die Methoden *unmarshal()* und *marshal()* erlauben zwischen der Java- und der XML-Repräsentation der Daten zu wechseln.

Für jede Komponentendefinition in XML soll eine Menge von Modellinstanzen erzeugbar sein. In *Enterprise Java Beans* (EJB) (Sun 2003) und CCM (OMG 2002) wird einem Komponententyp eine Fabrik, genannt *Home*, zugeordnet, mittels der Instanzen erzeugt werden können. Analog stellt die Komponentenfabrik *Home* in CoMo den Zusammenhang zwischen Komponentendefinitionen in XML und der Auswertungslogik in Java her (Röhl u. Uhrmacher 2006). XML-Dokumente für Komponenten stehen in einer 1:1-Beziehung zu Komponentenfabriken. Alle Komponenten eines bestimmen Typs werden von einer Fabrik erzeugt. Dafür wird *Home* mit den Java-Repräsentationen der Schnittstelle, der Komponente und des Modells initialisiert. Für eine Menge von Parametern erzeugt eine Komponentenfabrik Instanzen von *Component*. Sämtliche Kompositionslogik ist in *Component* implementiert.

Zum Zugriff auf Komponenten unterschiedlichen Typs wird eine globale Fabrik benötigt, die *Home*-Instanzen für Schnittstellenreferenzen bereitstellt. Analog zu CCM fungiert *EntityResolver* als zentraler Zugriffspunkt, um Fabriken für Komponenten zu erzeugen und bildet damit den Ausgangspunkt für alle Instantiierungen. Die globale Fabrik stellt den Zugriff auf Komponentendefinitionen eines bestimmten Typs über qualifizierte Namen bereit. Abbildung 5.7 visualisiert die grundlegenden Klassen und ihre Abhängigkeiten.

XML-Dokumente müssen aus Datenbanken aufgefunden, durchsucht und geladen werden können. Für die Definition der Kompositionslogik soll von den konkreten Speicherorten der Spezifikationseinheiten abstrahiert werden und der Zugriff auf sie mittels qualifizierter Namen möglich sein. `IDB` definiert eine entsprechende Schnittstelle für Datenbanksysteme, so dass auf Schnittstellen und

Komponenten, entsprechend der Plattformdefinition, über qualifizierte Namen zugegriffen werden kann. Implementierungen von IDB stellen den Zugriff auf XML-Daten (und damit Realisierungen der *dref-Funktionen aus der Plattformdefinition im vorangegangenen Kapitel*) sowie weitere Operationen bereit, um Daten hinzuzufügen, zu löschen und mittels XQuery-Anweisungen anzufragen. Mit `DBDummy` wurde eine prototypische Lösung realisiert, die XML-Dokumente lokal im Dateisystem verwaltet. Alternativ können Datenbankanbindungen implementiert werden, die Dokumente mittels URI aus dem Netz laden. `DBProxy` fungiert als allgemeiner Zwischenspeicher für bereits geladene Daten, so dass entfernt liegende Daten nur einmal geladen werden müssen.

In der Realisierung der Komponentenplattform sollen heterogene Modellimplementierungen möglich sein. Um einen Modellformalismus zu integrieren, müssen Implementierungen für folgende Schnittstellen erstellt und in der `XmlFactory` (siehe Abbildung 5.7) bekanntgegeben werden:

- *IBoundEntity:* Implementierungen müssen Definitionen aus XML-Dokumenten eines bestimmten Namensraums lesen, validieren und schreiben können.
- *ITransformer:* Implementierungen von `ITransformer` transformieren ein Modell von einer Quellrepräsentation in eine Zielrepräsentation.
- *IConstraintChecker:* Diese Schnittstelle fungiert als Erweiterungpunkt, um gebundene Entitäten eines bestimmten Namensraums mit zusätzlicher Funktionalität zu dekorieren.
- *ModelBuilder:* Für jeden Modellierungsformalismus, der als Zielformalismus genutzt werden soll, muss ein Transformator implementiert werden, der Kompositionsstrukturen auf Modellstrukturen abbildet.

Um Modelle auszuführen, muss eine weitere gebundene Entität bereitgestellt werden, die plattformspezifische Modellrepräsentationen umfasst. Des Weiteren bedarf es eines Transformators, der plattformunabhängige in plattformspezifische Modelle überführt.

Von XML-Dokumenten zu Komponentenfabriken Abbildung 5.8 veranschaulicht den Instantiierungsprozess. Eine Anwendung erfragt von der globalen Fabrik ein *Home* für den qualifizierten Namen einer Schnittstelle. Für die Schnittstelle wird ein XML-Dokument aus einer Datenbank geladen. Die XML-Daten werden mittels der Methode `unmarshal` in Java-Objekte umgewandelt. Die qualifizierten Namen der Komponentenimplementierung werden gelesen. Für die Implementierung und die Modelldefinition werden, analog zur Schnittstelle, XML-Dokumente aus der Datenbank geladen und die entsprechenden Java-Objekte erzeugt. XmlFactory erstellt gebundene Entitäten entsprechend des Namensraums der Modellimplementierung. Eine Komponentenfabrik wird dann mit Schnittstelle, Komponentendefinition und Modelldefinition instantiiert.

Instantiieren und Konfigurieren einer Komponente Abbildung 5.9 veranschaulicht den Ablauf zur Instantiierung und Konfiguration einer Komponente. Home erstellt Komponenteninstanzen für eine Menge von Parametern. Jede Komponenteninstanz erhält eine duplizierte Komponenten- sowie Modelldefinition, so dass in der Konfigurationsphase individuelle Datenobjekte verändert werden. Die Schnittstellendefinition bleibt für jede Komponenteninstanz unverändert und wird nicht dupliziert. Entsprechend des Namensraums des implementierten Modells wird das Modellobjekt dekoriert. Für die Komponentendaten ist der Dekorierer vorgegeben. Die in der Komponentendefinition angegebene Konfiguratorklasse wird instantiiert und dessen Methode *map()* mit dem Komponenten- und Modelldaten sowie den Parametern aufgerufen.

Abbildung von Komponenten auf Modelle Im vorigen Kapitel wurde die Semantik von Komponenten mit Hilfe des Modellierungsformalismuses PDEVS definiert. Die entsprechende Implementierung in CoMo veranschaulicht Abbildung 5.10. Falls die Modellimplementierung einer Kompo-

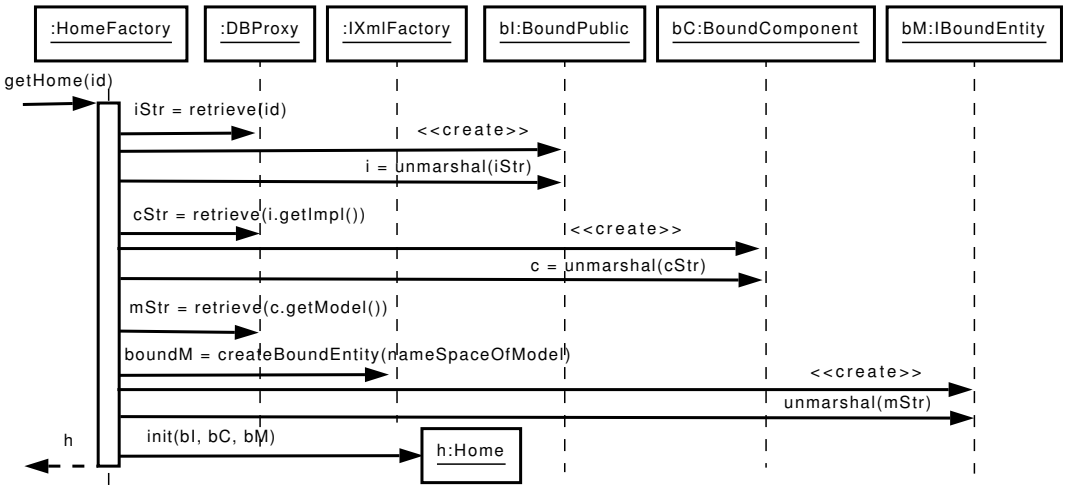

Abbildung 5.8: Instantiierung einer Komponentenfabrik aus XML-Daten

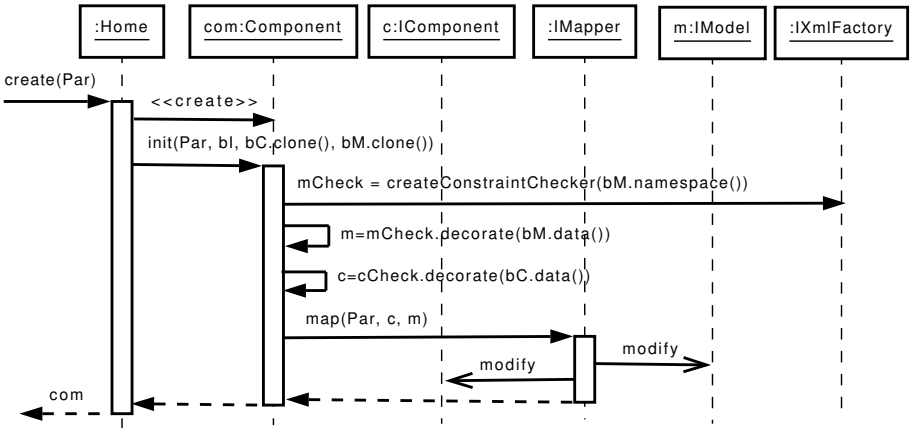

Abbildung 5.9: Instantiierung und Konfiguration einer Komponente

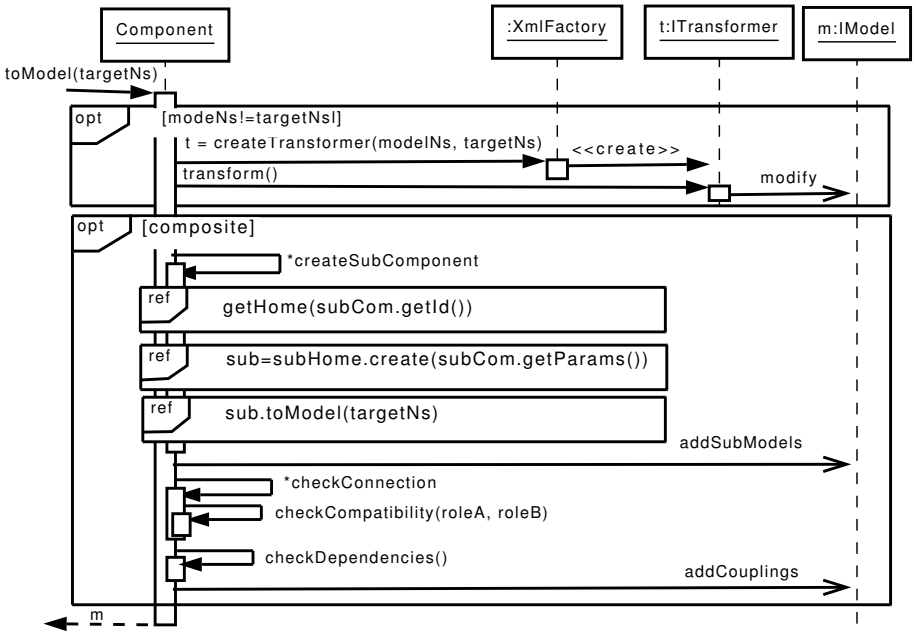

Abbildung 5.10: Abbildung von Komponenten auf Modelle

nente nicht bereits im gewünschten Zielformalismus vorliegt, wird ein entsprechender Modelltransformator zur Abbildung genutzt. Für atomare Komponenten ist die Modelltransformation damit abgeschlossen.

Im Falle einer zusammengesetzten Komponente müssen die Komponentenfabriken für sämtliche Subkomponenten entsprechend Abbildung 5.8 instantiiert werden. Mittels Komponentenfabriken können dann Komponenteninstanzen instantiiert und für eine Menge von Parametern konfiguriert werden (vgl. Abbildung 5.9). Für jede Unterkomponente wird rekursiv *toModel* ausgeführt und die Modelle als Untermodelle zum Zielmodell hinzugefügt.

Nach Definition im Metamodell wird eine Komposition als vollständig angesehen, falls alle benötigten abstrakten Interaktionspunkte mit kompatiblen Gegenstücken verbunden wurden. Alle Verbindungen zwischen Unterkomponenten müssen daher auf Kompatibilität geprüft werden. Die Kompatibilitätsprüfung findet für die referenzierten Rollen verbundener Komponentenports statt. In *checkDependencies()* wird schließlich überprüft, ob zu verbindenden Ports tatsächlich verbunden sind.

Die Transformation von Kompositionsstrukturen in Modellstrukturen hängt von den Ergebnissen der Kompatibilitätsprüfung ab. So sind die Ergebnispaare eines Rollenvergleichs die Grundlage, um Komponentenverbindungen in Modellkopplungen zu überführen.

5.3 Kompatibilitätsprüfung

Die Flexibilität der XML-basierten Schnittstellenbeschreibungen stellt sich zum Zeitpunkt der Integration von Komponenten als Herausforderung dar (Röhl u. Morgenstern 2007). Unabhängig voneinander entwickelte Beschreibungen können bezüglich Syntax und Semantik variieren. So können sich die Namen von Attributen und Elementen oder auch die Reihenfolge von Elementen unterscheiden. Umgehen lassen sich diese Variationen durch den Rückgriff auf gemeinsame Definitionen, bspw. standardisierte Beschreibungen. Falls dies nicht möglich oder wünschenswert ist, muss, um Kompositionskandidaten auffinden zu können, für unabhängig voneinander entwickelte

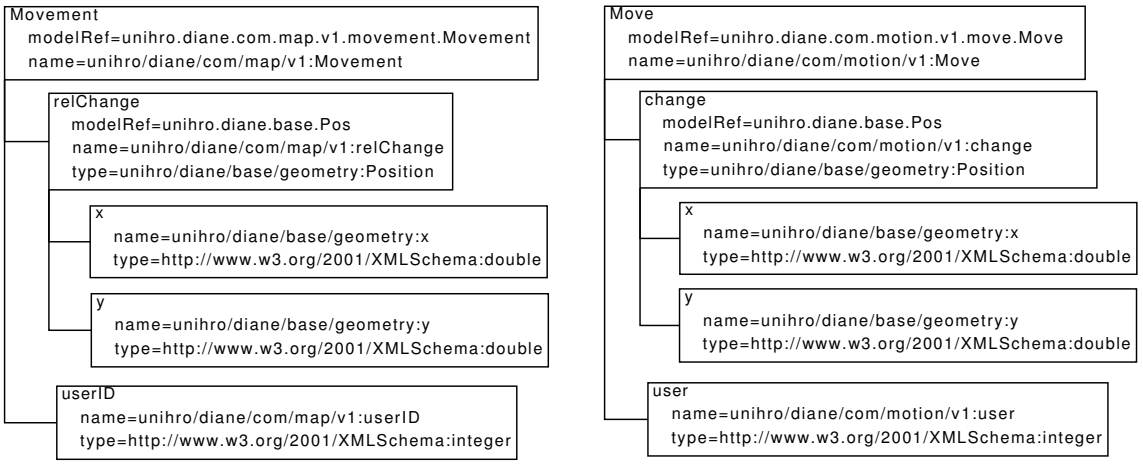

Abbildung 5.11: Interne Repräsentation von XSD-Typen als Graphen

Typen entschieden werden, ob sie kompatibel sind oder nicht. Zum Zeitpunkt der Komposition gilt es darüber hinaus syntaktische Variationen für kompatible Typen aufzulösen.

Schnittstellenbeschreibungen basieren im Wesentlichen auf Definitionen in XSD. Abbildungen zwischen Schemata zu finden, ist ein grundlegendes Problem in der Integration von Datenbankanwendungen (Do u. a. 2003). Unter Verwendung von Schemaabbildungen kann syntaktische Gleichheit von Schnittstellen zu einer weniger restriktiven Beziehung abgeschwächt werden (Walsh 2004). Für die Kompatibilitätsprüfung orientieren wir uns daher an Ansätzen zum *schema matching*. Zwei Typen aus zwei verschiedenen Schemata können dann als kompatibel betrachtet werden, wenn eine Abbildung zwischen diesen beiden Typen hergestellt werden kann.

In der Praxis existieren verschiedene Ansätze zur Schemaabbildung. Die Grundlage bilden jedoch typischerweise Vergleiche von Zeichenfolgen, Phonetik, Synonymen oder Strukturen. Die Leistungsfähigkeit einzelner Vergleiche ist für praktische Anwendungen meist zu beschränkt. Bessere Ergebnisse können durch die Kombination unterschiedlicher Vergleichsmodule erzielt werden (Do u. Rahm 2002). Eine modulare Architektur ist gerade dort sinnvoll, wo möglichst automatische Lösungen angestrebt werden.

Im Rahmen der Diplomarbeit von Stefan Morgenstern (2007) wurden eine Reihe von Vergleichsmodulen implementiert und eine Architektur zu deren flexiblen Kombination realisiert. Angelehnt ist die Architektur an das Schema-Matching-Werkzeug Coma++ (Aumueller u. a. 2005). Zu verwendende Vergleichsmodule können ausgewählt und konfiguriert werden, sowie eine Menge von kompatiblen Typen vom Nutzer vordefiniert werden, um den Vergleichsprozess zu unterstützen. Für den Vergleich werden Schemadefinitionen als gerichtete azyklische Graphen, englisch *directed acyclic graph* (DAG), repräsentiert.

Beispiel 5.4. *Zur Komposition eines Netzwerkmodells werden Knotenkomponenten mit einer Komponente verbunden, die das physikalische Umgebungsmodell repräsentiert. Im Rahmen dieser Verbindungen wird Movement von der Komponente Node genutzt, um anzuzeigen, dass sie als bewegliche Einheiten aufzufassen sind. In einem unabhängig davon entwickeltem Umgebungsmodell wird der Typ Move verwendet, der sich von Movement in syntaktischen Details unterscheidet. Abbildung 5.11 zeigt die Repräsentation der XSD-Typen* `Movement` *und* `Move` *als gerichtete, azyklische Graphen. In der Abbildung ist die Richtung einer Kante mittels Einrückung visualisiert.*

Im Anwendungsbereich der Modellierung und Simulation gilt es eine Besonderheit zu beachten: Ein Simulationsmodell kann eine Menge von Modellinstanzen gleichen Typs beinhalten. Die Kompatibilitätsprüfung sollte erlauben, bereits gefundenen Abbildungen zwischen Typen wiederzuverwen-

den. Die implementierte Kompatibilitätsprüfung ermittelt daher zuerst, ob die zu überprüfenden Typen bereits auf Kompatibilität untersucht wurden. Nur falls dies nicht der Fall ist, werden die zwei Typen entsprechend ihrer Schemadefinitionen in DAG-Knoten transformiert. Die Kompatibilitätsprüfung finden dann entsprechend der Konfiguration mit einer Menge von Matchern statt. Konfigurationen definieren, welche Matcher für welche Informationen zu nutzen sind, sowie welche Gewichte für die Kombination der einzelnen Ergebnisse verwendet werden sollen.

Standardmäßig wird in CoMo ein Vergleichsmodul verwendet, das die Kindknoten von zwei DAG-Knoten miteinander vergleicht und ein Modul, das den Editierabstand (Levenshtein-Distanz) für Namen feststellt. Für den Vergleich der Kindknoten wird jeder Kindknoten des einen Typs mit jedem Kindknoten des anderen Typs verglichen. An Blattknoten wird dann der Typ und der Name der Knoten verglichen. Einfache XSD-Typen können direkt aufeinander abgebildet werden.

Ergebnis der Kompatibilitätsprüfung ist ein Wert zwischen 0 und 1, der den Grad der Kompatibilität angibt. Liegt der Wert über einer bestimmten Grenze, bspw. 0,5[2], werden zwei Typen als kompatibel betrachtet. Nach Abschluss der Prüfung werden die ermittelten Kompatibilitätswerte zur Wiederverwendung gespeichert.
Verbundene Komponentenports müssen für alle Ereignisports ihrer Rollendeklarationen kompatible Gegenstücke aufweisen. Falls nicht, gilt die entsprechende Verbindung und damit die gesamte Komposition als syntaktisch nicht korrekt. Ist die Kompatibilitätsprüfung hingegen erfolgreich, können auf ihrer Grundlage Modellkopplungen entsprechend der Definition im Metamodell gebildet werden.

5.4 Kopplung mit dem Simulationssystem James II

Das Endprodukt einer Komponenteninstantiierung ist ein PDEVS-Modell entsprechend des XML-Schemas in Anhang B.5. Die plattformunabhängige Repräsentation ist für die Ausführung des komponierten Modells jedoch nicht sinnvoll (Röhl 2006). Zum Zweck der Ausführung gilt es, die Modellrepräsentation auf Laufzeit- und Speichereffizienz auszurichten. Die Anbindung eines Simulationssystems an CoMo bedarf eines Transformators, der deklarative PDEVS-Modelle in ausführbare Modelle überführt.

Als Zielplattform für komponierte Modelle fungiert im Folgenden James II (Himmelspach u. Uhrmacher 2007), mit dem PDEVS-Modelle effizient ausgeführt werden können. So erlaubt James II unter anderem, PDEVS-Modelle mit unterschiedlichen Simulatoren, bspw. sequentiell, parallel oder auch gemischt, auszuführen. Des Weiteren lassen sich Simulationsmodelle in James II flexibel mit Beobachtungsmöglichkeiten versehen.

Zwischen Modellen auszutauschende Ereignisse sind in XSD definiert und die Modelldefinitionen referenzieren in ihren Eingabe- und Ausgabeports XSD-Typen. Ob Schnittstellen zueinander kompatible Typen nutzen, wird für Kompositionen geprüft. Als wesentliche Herausforderung für die Generierung eines konsistenten Simulationsmodells stellt sich jedoch, wie unabhängig voneinander entwickelte Implementierungen auf technischer Ebene integriert werden können. Um den Zusammenhang zwischen deklarierten und implementierten Ereignissen herzustellen, existieren folgende Alternativen:

- Modelle erstellen XML-Repräsentationen von Nachrichten, die über die Modellkopplungen verschickt werden. Dies erhöht den Laufzeitbedarf der Simulation, da für jedes gesendete Ereignis eine Transformation in XML und für jedes empfangene Ereignis eine Transformation in Java-Objekte durchgeführt werden müsste.
- Die Modellimplementierungen nutzen Java-Implementierungen für Nachrichten. Zu komponierende Modelle müssen dann jedoch die gleichen Ereignisklassen nutzen, um miteinander

[2]Zum Auffinden von Kompositionskandidaten ist es ggf. sinnvoll mit einem niedrigerem Schwellenwert zu starten und ihn sukzessive zu erhöhen.

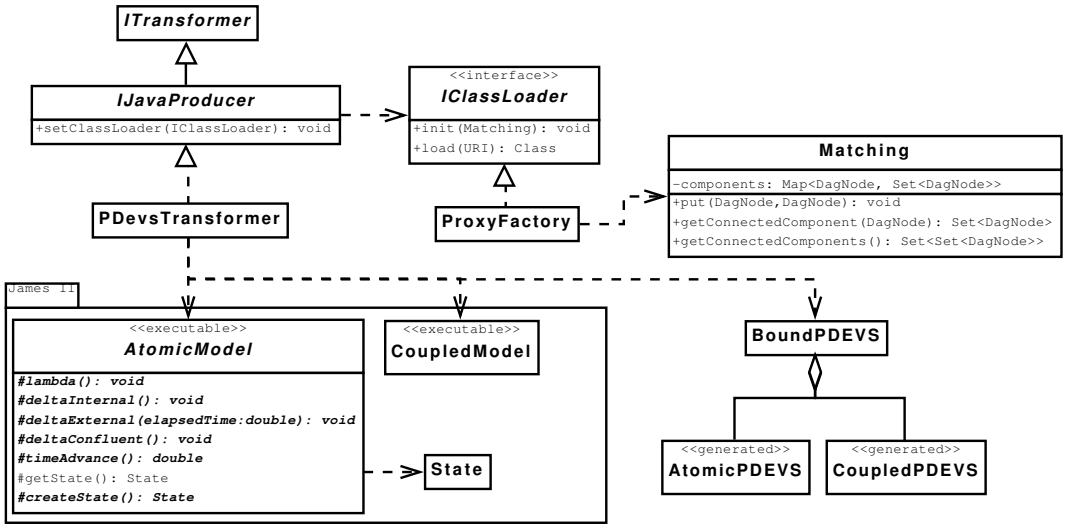

Abbildung 5.12: Klassen zur Erstellung eines Simulationsmodells für James II

kommunizieren zu können. Damit unterlaufen Modelle die starke Entkopplung von Komponenten über Schnittstellen und ein entscheidender Vorteil der Komponentenplattform, Modelle unabhängig voneinander entwickeln zu können, geht verloren.

- Jede Komponente definiert lokal Schnittstellen in Java, über die es auf Ereignisse zugreift. Die tatsächliche Implementierung einer Nachricht muss dann die Schnittstellen aller Modelle implementieren, die entsprechende Ereignisse senden oder empfangen können. Dies ist aufwendiger, jedoch für eine Komposition vor der Ausführung möglich.

Für die dritte Variante lässt sich der generative Ansatz zur Bindung von Datenobjekte an Schemadefinitionen nutzen. Sämtliche Nachrichtentypen sind für eine Modellkomponente im Rahmen ihrer Rollen mit XSD definiert. Mit JAXB lassen sich die Schnittstellen direkt aus und vollständig konsistent zu den XSD-Typen generieren. Die Ergebnisse der Kompatibilitätsprüfung stellen andererseits bereits den Zusammenhang zwischen kompatiblen Typen her. Mittels Java-Proxies (Anft u. Friese 2003) lassen sich automatisch Implementierungen für kompatible Schnittstellen instantiieren. Abbildung 5.12 verdeutlicht den Zusammenhang zwischen dem Endprodukt der Komposition und ausführbaren Modellklassen von James II.

Beispiel 5.5. *Abbildung 5.13 gibt einen Überblick über die Verschränkung der Definitionseinheiten einer Komposition in XML und Java. Typdefinitionen in XSD werden genutzt, um den statischen Anteil von Modellimplementierungen plattformunabhängig und unabhängig von anderen Modellen definieren zu können. Porttypen für Modelle werden aus XSD-Definitionen mittels JAXB generiert und Implementierungen zur Laufzeit über eine Proxy-Fabrik zur Verfügung gestellt. So ist die Implementierung für die Java-Schnittstelle Move erst nach der Kompatibilitätsprüfung verfügbar.*
Einen Ausschnitt der Java-Implementierung für das Bewegungsmodell listet Abbildung 5.14.

Um den technischen Aspekt der Kompatibilität von der Ausführungsphase zu trennen, wurde ein zweistufiger Transformationsprozess vorgestellt. Im ersten Schritt werden Implementierungen von Komponenten und Kompositionsstrukturen in gekoppelte PDEVS-Modelle transformiert. Im zweiten Schritt werden die PDEVS-Modelle samt Ereignisklassen für ein spezielles Simulationswerkzeug, bspw. James II, aufbereitet. Java-Datenobjekte werden aus plattformunabhängigen XSD-Beschreibungen generiert. Transformationen zwischen XML-Daten und Datenrepräsentationen in Java werden so zur Laufzeit vermieden.

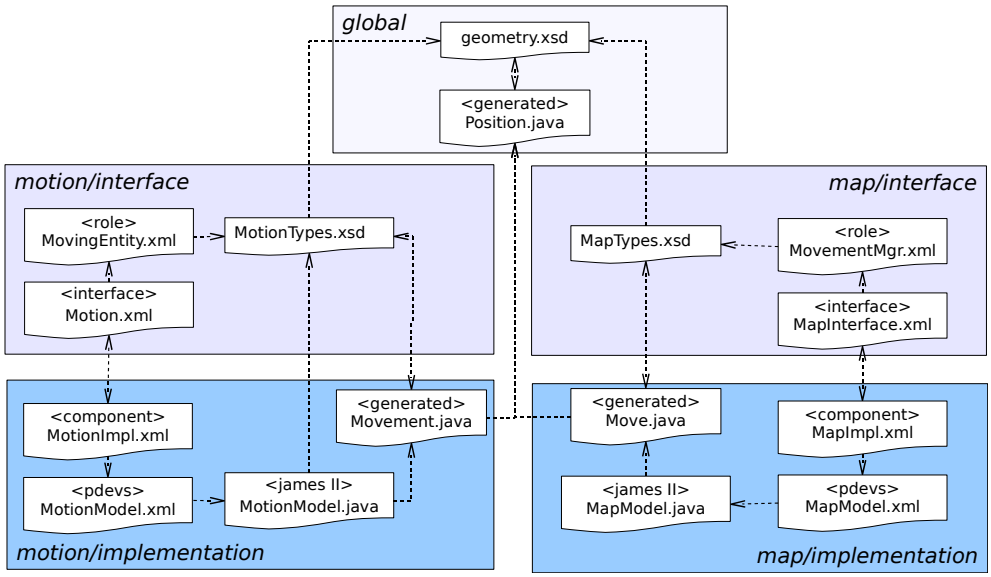

Abbildung 5.13: Abhängigkeiten in Kompositionen

```
package unihro.diane.com.motion.v1;
...
import unihro.diane.com.motion.v1.movement.Movement;
public class Model extends AtomicModel<State> {
  private static final ProxyFactory factory = ProxyFactory.instance();
  public void lambda() {
    final State s = getState();
    switch (s.phase()) {
    case INIT:
      break;
    case MOVING:
      Position moveTo = s.pathToWalk.get(0);
      Movement movement = factory.create(Movement.class);
      movement.setChange(moveTo);
      getOutPort("move").write(movement);
      break; ...
    } } ... }
```

Abbildung 5.14: Definition eines atomaren PDEVS-Modells für James II mit Proxies

```
public void instrumentModel(IModel simModel) { ...
  IBasicCoupledModel rootM = (IBasicCoupledModel) simModel;
  NetworkObserver networkObserver = new NetworkObserver();
  ServiceObserver serviceObs = new ServiceObserver();
  Iterator<IBasicDEVSModel> modelIt = rootM.getSubModelIterator();
  while (modelIt.hasNext()) {
    IBasicDEVSModel curSubModel = modelIt.next();
    if (curSubModel.getName().equals("network")) {
      BasicAtomicModel<NetworkState> netM =
      (BasicAtomicModel<NetworkState>) curSubModel;
      NetworkState netState = netM.getState();
      netState.registerObserver(networkObserver);
    }
    // observe service search calls of users
    else if (curSubModel.getName().indexOf("node") == 0) {
      IBasicCoupledModel node = (IBasicCoupledModel) curSubModel;
      IBasicDEVSModel curUser = node.getModel("user");
      IPort callPort = curUser.getOutPort("call");
      callPort.registerObserver(serviceObs);
    } }
  ...
}
```

Abbildung 5.15: Konfiguration von Beobachtungsintrumenten

Wurde aus einer Komposition ein operationales Modell für James II erstellt, können Beobachtungsinstrumente konfiguriert werden. Eine beispielhafte Konfiguration von Beobachtungsinstrumenten zeigt Abbildung 5.15.

Die Kopplung von James II an CoMo ist unidirektional. Ausführbare Modelle haben keinen Bezug mehr zu XML und können ohne entsprechenden *Overhead* ausgeführt werden (Röhl u. Uhrmacher 2005; Himmelspach u. Röhl 2008).

5.5 Prototypische Integration von Statecharts

Modelle in der Syntax von James II zu definieren, ermöglicht diese im Vergleich zu XML-Repräsentationen effizient auszuführen. Plattformspezifische Modellrepräsentationen erschweren jedoch Modellkomponenten über Werkzeuggrenzen hinaus wiederzuverwenden. Modelle sind als Implementierung in einer Programmierssprache für einen bestimmten Simulator plattformspezifisch.

Eine Alternative zu plattformspezifischen Modellen stellen Modelldefinitionen dar, die keine Annahmen über die Zielplattform treffen und möglichst vollständig in XML verfasst sind. Modelle können so in Datenbanken plattformunabhängig vorgehalten werden und für die Ausführung in plattformspezifische Repräsentationen überführt werden.

Für DEVS existiert jedoch noch kein XML-basiertes Standardformat. Anders ist die Situation für mit DEVS verwandte Formalismen wie bspw. Statecharts (Harel 1987), für die gleich mehrere Austauschformate existieren. Statecharts weisen eine enge Verbindung zu DEVS auf (Schulz u. a. 2000; Borland u. Vangheluwe 2003). In vergangenen Simulationsstudien mit James und James II haben sich Statecharts zur Konzeption und Dokumentation atomarer DEVS-Modelle bewährt (Uhrmacher u. a. 2002; Röhl u. Uhrmacher 2005).

Mit *State Chart XML* (SCXML) liegt ein Vorschlag zur Repräsentation von Zustandsautomaten vor (W3C 2007), der nicht so komplex wie die Zustandmaschinen der UML (OMG 2006) ist und für den eine Schemadefinition in XSD existiert. Im Gegensatz zur Repräsentation von Zustandsmaschinen mit UML und XMI, die auf den Austausch zwischen Computern ausgelegt sind, zielt

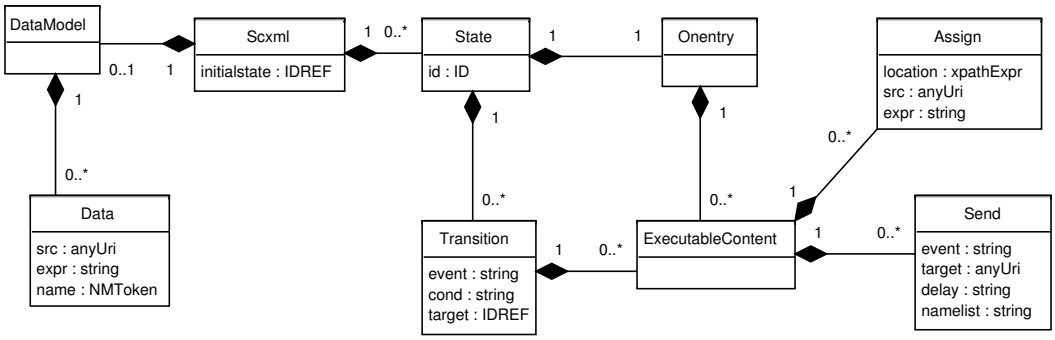

Abbildung 5.16: Verwendete Teilmenge von SCXML

SCXML auch darauf, von Menschen direkt erstellt zu werden.

Die zentralen Beschreibungsmittel von SCXML enthält Abbildung 5.16. Ein SCXML-Dokument kann explizit ein Datenmodell definieren, das eine Menge von Zustandsvariablen beschreibt. Hauptsächlich besteht ein Statechart aus Zuständen. Zustände können Zustandsübergänge und zusätzliche Aktionen bspw. für das Eintreten oder Verlassen eines Zustandes beinhalten. Mit Zustandsüberführungen kann ein ausführbarer Inhalt assoziiert werden, wie das Zuweisung von Werten zu Zustandsvariablen oder das Senden von Ereignissen.

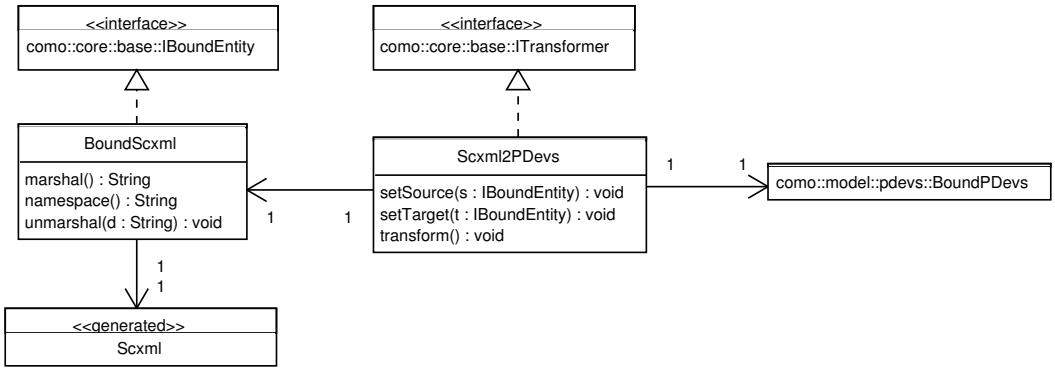

Abbildung 5.17: Integration von SCXML als gebundene Entitäten

Abbildung 5.17 veranschaulicht die Integration von SCXML in CoMo. Um SCXML-Definitionen mit CoMo verarbeiten zu können, wurde mit JAXB Java-Code aus dem standardisierten XSD-Dokument generiert. Eine Implementierung von *IBoundEntity* stellt Lese- und Schreiboperation zur Verfügung und delegiert an die generierten Klassen. Zur Ausführung im Rahmen von Kompositionen wird ein Transformator benötigt, der SCXML-Definitionen in PDEVS-Modelle transformiert. `Scxml2PDevs` produziert als prototypische Implementierung dieses Transformators atomare PDEVS-Modelle. Komponenten, die eine Modelldefinition in SCXML enthalten, können somit in Kompositionen mit PDEVS-Modellen kombiniert werden.

Beispiel 5.6. *Abbildung 5.18 visualisiert ein Bewegungsmodell als Statechart. Die Definition des Modells in SCXML listet Abbildung 5.19. Das Modell wartet im Zustand* idle *auf eine Eingabe.*

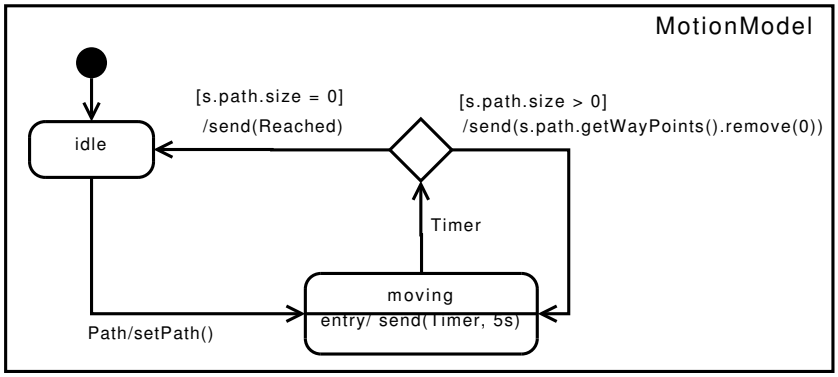

Abbildung 5.18: Definition des Bewegungsmodells als Statechart

Empfängt das Modell einen Pfad, wechselt es in den Zustand moving*. Bei jedem Eintreten in den Zustand* moving *wird ein Ereignis produziert, das Bewegung repräsentiert. Nach verstreichen von fünf Zeiteinheiten (getriggert durch eine spezielle Sendeoperation im Zustand* moving*) wird überprüft, ob das Ende des Pfads erreicht ist. Falls dies nicht der Fall ist, tritt das Modell erneut in den Zustand* moving *ein. Ist das Ende des Pfads erreicht, produziert das Modell ein Ereignis vom Typ* Reached *und geht in den Zustand* idle *über.*

Abbildung 5.20 zeigt einen Auszug des Codes, der aus dem SCXML-Modell für James II generiert wurde. Empfangbare Ereignisse werden als Eingabeports interpretiert und zu sendende Ereignisse als Ausgabeports. Da Statecharts keine Portnamen besitzen, repräsentiert ein Ereignistyp gleichzeitig den Portnamen.

In Statecharts ist es möglich, Transitionen mit Ausgabeereignissen zu versehen. Mit PDEVS können jedoch nur Ausgaben produziert werden, bevor eine interne oder konfluente Zustandsüberführung durchgeführt wird. Aus diesem Grund muss für Sendeoperationen eine separate Zustandsüberführung ausgelöst werden. Im Beispiel trifft dies für die Phasen *moving-sending* und *idle-sending* zu.

Für die generierten Modellimplementierungen erzeugt die Proxy-Fabrik Ereignisse, die gesendet werden sollen. Damit ist sichergestellt, dass Kompatibilität, die auf Kompositionsebene überprüft wurde, im operationalen Modell tatsächlich gegeben ist.
Über die Nutzung von Java hinaus treffen die SCXML-Modelle keine Annahmen über die Ausführungsplattform. In Kompositionen können SCXML-Modelle mit PDEVS-Modellen für James II kombiniert werden.

5.6 Zusammenfassung

CoMo realisiert die im vorigen Kapitel entwickelte Komponentenplattform in der Programmiersprache Java. Das Werkzeug CoMo führt unabhängig voneinander entwickelte Modellkomponenten auf Grundlage von öffentlich zugänglichen Schnittstellenbeschreibungen in XML zu einem ausführbaren Simulationsmodell zusammen.

XML-Dokumente werden in CoMo strikt nach Prinzipien des *data binding* verarbeitet. Sämtliche Beschreibungseinheiten der Komponentenplattform sind an Schemadefinitionen gebunden, aus denen Java-Klassen zum Lesen und Schreiben von XML-Dokumenten generiert werden. Die beiden Entwurfsmuster Adapter und Dekorierer kombinieren die Vorteile des generativen Ansatzes mit den spezifischen Anforderungen zum Zugriff auf die Daten entsprechend der Kompositionslogik.

```
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0" initialstate="idle">
 <datamodel>
  <data name="path" src="unihro/diane/com/motion/v2/PathProc:Path"/>
  <data name="posChange" src="example/base/geometry:Position"/>
  <data name="move" src="unihro/diane/com/motion/v2:Move" expr="create"/>
 </datamodel>
 <state id="idle">
  <transition event="unihro/diane/com/motion/v2/PathProc:Path" target="moving">
   <assign location="path" expr="_eventdata"/>
  </transition>
 </state>
 <state id="moving">
  <onentry> <send event="Timer" delay="5"/> </onentry>
  <transition event="Timer" cond="s.path.getWayPoints().size() > 0"
              target="moving">
   <assign location="posChange" expr="s.path.getWayPoints().remove(0)"/>
   <assign location="move" expr="setChange(s.posChange)"/>
   <send event="unihro/diane/com/motion/v2:Move" namelist="move"/>
  </transition>
  <transition event="Timer" cond="s.path.getWayPoints().size() == 0"
              target="idle">
   <send event="unihro/diane/com/motion/v2/PathProc:Reached"/>
  </transition>
 </state>
</scxml>
```

Abbildung 5.19: Repräsentation des Modells *Motion* als Statechart in SCXML

```
public class class1 extends AtomicModel<james.core.model.State> {
  private static final ProxyFactory factory = ProxyFactory.instance();
  class StateImpl extends james.core.model.State {
    String phase = "idle";
    Move move = factory.create(Move.class); ...}
  public void lambda() {
    StateImpl s = (StateImpl) getState();
    if (s.phase.equals("moving_sending")) {
      IPort outport = getOutPort("Move");
      outport.write(s.move);}
    if (s.phase.equals("idle_sending")) {
      IPort outport = getOutPort("Reached");
      outport.write(s.reached); } }
  public void deltaInternal() {
    StateImpl s = (StateImpl) getState(); String newPhase = null;
    if ((s.phase.equals("moving")) && (s.path.getWayPoints().size() > 0)) {
      s.posChange = s.path.getWayPoints().remove(0);
      s.move.setChange(s.posChange);
      newPhase = "moving_sending"; }
    if (s.phase.equals("moving_sending")) { newPhase = "moving"; }
    if ((s.phase.equals("moving")) && (s.path.getWayPoints().size() == 0)) {
      s.reached.setPosition(s.position);
          newPhase = "idle_sending"; }
    if (s.phase.equals("idle_sending")) { newPhase = "idle";}
   s.phase = newPhase;}
...}
```

Abbildung 5.20: Auszug aus generiertem Code für James II

CoMo ist als erweiterbares Rahmenwerk konzipiert, dessen zentraler Erweiterungspunkt die Möglichkeit zum Einbindung unterschiedlicher Modellierungsformalismen darstellt. Am Beispiel von SCXML wurde prototypisch aufgezeigt, wie neue Modellformalismen in CoMo integriert werden können.

Die Kompatibilität von Schnittstellen bzw. Rollen wurde in Anlehnung an Schemamatching realisiert. Damit können leichte syntaktische Unterschiede in Schnittstellendefinitionen erkannt und ggf. automatisch überbrückt werden. Semantische Annotationen werden in CoMo bisher nur als qualifizierte Namen verglichen, die jedoch nicht dereferenziert werden, um ihre Inhalte miteinzubeziehen.

Mit der Abbildung von Kompositionsstrukturen auf PDEVS-Modelle ist es möglich, Kompositionen als „normale“ Modelle auszuführen. Die Vorteile des entwickelten Kompositionsansatzes können so bspw. mit den flexiblen Ausführungsmitteln des Simulationssystems James II kombiniert werden.

6 Fallbeispiel DIANE

Computernetzwerke auf Basis drahtloser Kommunikation mobiler Knoten werden als mobile Ad-Hoc-Netze, engl. *Mobile Ad-Hoc Networks* (MANETs), bezeichnet. Erscheinende und verschwindende sowie frei bewegliche Netzwerkknoten induzieren dynamische Änderungen in der Verbindungstopologie. Darüber hinaus sind Bandbreite und Energie in MANETs beschränkt. So gelten insbesondere zuverlässige und schnelle Verbindungen für MANETs als Herausforderung (Corson u. Macker 1999).

Unabhängig vom Stromnetz operierende Endgeräte besitzen nur beschränkte Resourcen und Möglichkeiten. Komplexe Operationen erfordern in MANETs ggf. die Kooperation von Knoten. Dafür müssen Ressourcen und Potentiale bekanntgegeben werden und auffindbar sein. Falls keine feste Infrastruktur existiert, ist eine zentrale Kontrolle von Netzwerkoperationen nicht möglich und die Knoten müssen selbst Infrastrukturoperationen übernehmen. Eine Möglichkeit ist, dies im Stil dienstorientierter Architekturen zu lösen, so dass Infrastrukturoperationen automatisiert und transparent für den Endnutzer realisiert werden können.

Das DIANE-Projekt (Diane 2007) widmet sich der Problematik zur Dienstvermittlung und Diensterbringung in MANETs. Dazu werden Dienstbeschreibungen und Methoden zum Abgleich von Beschreibungen, Anreizmechanismen und verteilte Reputationssysteme entwickelt (Küster u. a. 2007). Im Rahmen von DIANE besteht Bedarf zur experimentellen Evaluation von Protokollen und Konzepten (Klein u. a. 2004).

Simulationsstudien im Kontext von MANETs nutzen traditionell spezialisierte Simulationsysteme in Form von Netzwerksimulatoren (Kurkowski u. a. 2005). Netzwerksimulatoren konzentrieren sich auf die unteren Schichten der Protokollhierarchie in Netzwerken, um bspw. Routingprotokollen zu evaluieren (Andel u. Yasinac 2006). Simulationen von MANETs nutzen meist vereinfachte Modelle für Mobilitätsverhalten (Tan u. a. 2002), Dienstnutzungsverhalten (König-Ries u. a. 2006) und zur Funkausbreitung (Cavilla u. a. 2004).

Für DIANE stellen sich Fragen nach dem Aufwand-Nutzen-Verhältnis von Protokollen zur Dienstvermittlung im Kontext unterschiedlicher Nutzermodelle (Röhl u. a. 2007a). Insbesondere interessiert, wie sich Nutzermodelle mit unterschiedlichen Dienstverhalten und Bewegungsmustern auf Protokolle zur Dienstvermittlung auswirken. Nutzerverhalten und Vermittlungprotokolle sollten möglichst unabhängig voneinander modelliert werden können. Zur Evaluation im Rahmen von Simulationsmodellen ist es jedoch wünscheswert, Modellkomponenten zur Dienstvermittlung und Nutzerkomponenten flexibel kombinieren zu können.

In realen MANETs finden, entsprechend des OSI-Referenzmodells (Tanenbaum u. van Steen 2002), sämtliche Netzwerkaktivitäten, mit Ausnahme der Funkausbreitung, in den einzelnen Netzwerkknoten statt. Da im Kontext von DIANE vor allem die höheren Schichten von Interesse sind, werden die unteren vier OSI-Schichten in einem zentralen Modell zusammengefasst.

Abbildung 6.1 zeigt ein konzeptuelles Modell eines MANETs bestehend aus Knoten, die sich auf einem Campus bewegen und mittels Dienstprotokollen Nachrichten austauschen. Die Grundstruktur enthält drei unterschiedliche Arten von Modellen: Nutzermodell, Vermittlungsprotokoll und ein Modell für die Umgebung von Netzwerkknoten. Ein Nutzer und ein Vermittlungsprotokoll werden jeweils in einem Knoten zusammengefasst. Zur Kommunikation dienen folgende Datenstrukturen:

Call Vom Nutzer initiierte Aktionen zur Bekanntgabe eines Dienstes *(ServiceOfferCall)*, zum Widerruf eines Dienstes *(ServiceRevokeCall)* und zur Suche eines Dienstes *(ServiceSearchCall)*

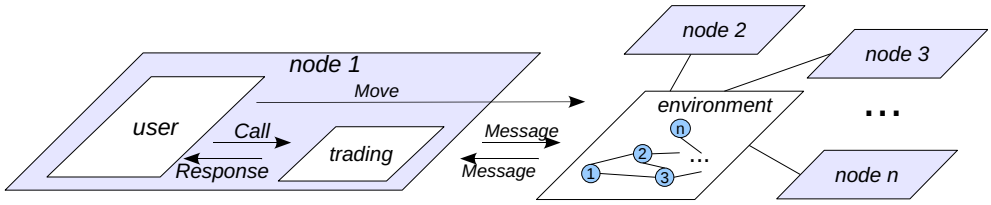

Abbildung 6.1: Konzeptuelles Modell von MANETs für DIANE

Response Antworten des Vermittlungsprotokolls auf Aktionen des Nutzers, bspw. auf eine Dienstsuche mit *ServiceSearchResponse*

Message Nachrichten beliebigen Inhalts, die über das Netzwerk versandt werden.

Move Bewegungsinformationen von Nutzern in einer zweidimensionalen Umgebung

Das konzeptuelle Modell dient nun als Ausgangspunkt, ein Simulationsmodell für DIANE mit den Beschreibungsmitteln der entwickelten Kompositionsplattform zu realisieren.

6.1 Definition einer Modellkomponente

In Beispiel 5.6 wurde ein SCXML-Modell zur Generierung von Bewegungsinformationen in zweidimensionalen Umgebungen vorgestellt.

Um dieses Modell als Modellkomponente zu nutzen, müssen alle Angebote und Anforderungen des Modell veröffentlicht werden. Das Bewegungsmodell erwartet Pfadinformationen von seiner Umgebung und generiert Bewegungsereignisse. Diese beiden funktionalen Aspekte lassen sich in Rollen kodieren und in Schnittstellenports als Kontextanforderung bzw. angebotene Funktionalität ausstellen. Abbildung 6.2 veranschaulicht den Zusammenhang zwischen der Schnittstelle `Motion`, den beiden referenzierten Rollen `MovingEntity` und `PathProc` sowie der Modelldefinition `MotionModel`.

Die konkrete XML-Syntax zur Definition der gesamten Komponente `Motion` ist in Abbildung 6.3 dargestellt. Gestrichelte rote Pfeile symbolisieren die Verweise über qualifizierte Namen. Der öffentliche Teil der Komponente besteht aus einem Schnittstellendokument, zwei Rollenbeschreibungen (`PathProc` und `MovingEntity`) und drei Schemadokumenten (eines zur Definition von `Move`, eines für `Path` & `Reached`; `Position` wird aus einem dritten Dokument importiert). Die Implementierung der Modellkomponente besteht aus zwei XML-Dokumenten und einer Java-Klasse. Das erste XML-Dokument enthält die Modelldefinition in dem Formalismus SCXML (W3C 2007). Das zweite XML-Dokument beschreibt den formalismusunabhängigen Teil der Implementierung.

In Hinblick auf die Anforderungen an Kompositionsansätze für Modellkomponenten bietet diese Beschreibung der Komponente *Motion* eine Reihe von Vorteilen:

Austauschformat Rollen und Schnittstellen sind komplett in XML verfasst. Typdefinitionen werden im W3C-Standard XSD definiert. Einzelne Schnittstellenteile können als Ausgangspunkt fungieren, um Kompositionskandidaten aufzufinden. Um bspw. ein Modell zu identifizieren, das Pfadinformationen für `Motion` liefert, genügt es, eine zu `PathProc` komplementäre Rolle zu finden.

Separate Schnittstelle Die öffentliche Beschreibung ist vollständig von der Implementierung getrennt. Der Zusammenhang zwischen Schnittstelle und Implementierung wird ausschließlich über qualifizierte Namen hergestellt. Darüberhinaus ist die Schnittstellebeschreibung selbst

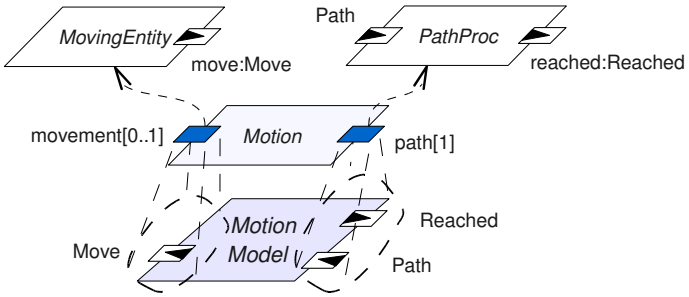

Abbildung 6.2: Die Komponente Motion

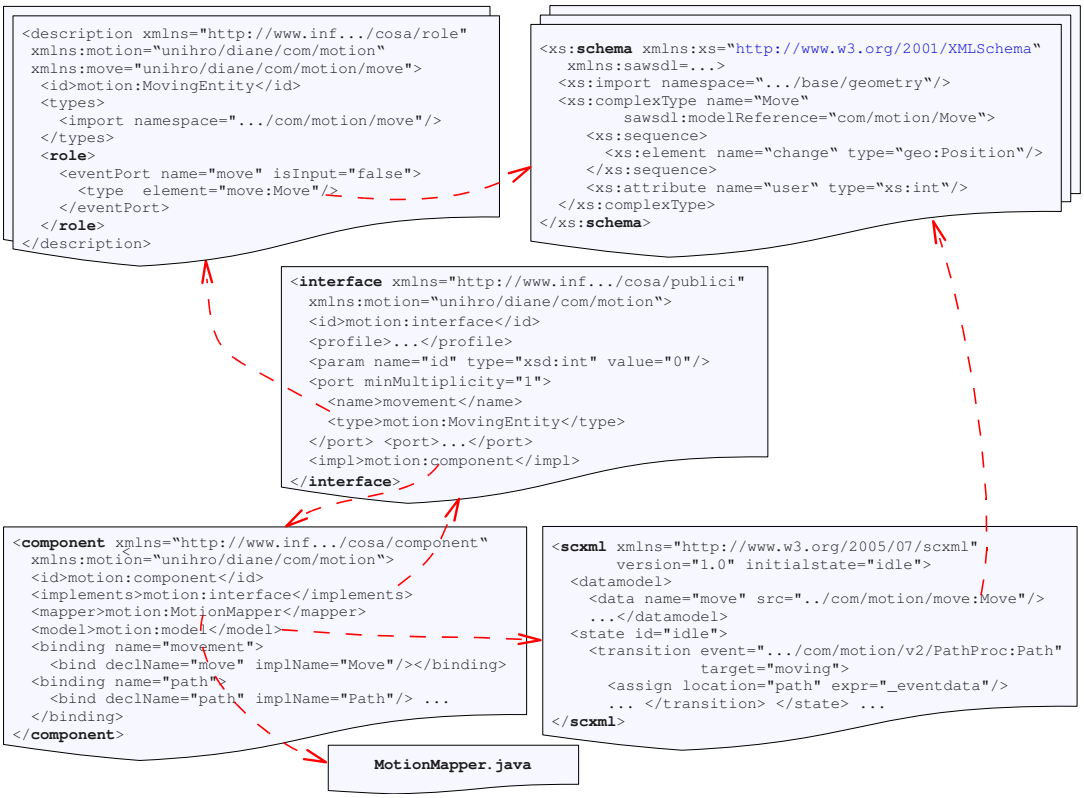

Abbildung 6.3: Zusammenhang zwischen den XML-Dokumenten für Motion

in separate Dokumente unterteilt. Jede Rolle liegt in einem eigenen XML-Dokument. Typdefinitionen in XSD liegen in separaten Dokumenten, die selbst wiederum durch Namensräume strukturiert werden können. Auf diese Weise lassen sich, je nach Verfügbarkeit, eigene, fremde und ggf. standardisierte, Typ- und Rollendefinitionen referenzieren. In der Schemadefinition des Typs `Move` wird bspw. der Namensraum `.../geometry` importiert, um den Typ `Position` wiederzuverwenden.

Modellierung Das abgebildete SCXML-Modell ist eine eigenständige Modelldefinition, die auch unabhängig von der entwickelten Kompositionsplattform nutzbar ist.

Parametrisierung Die Schnittstelle `Motion` gibt einen Konfigurationspunkt (`id`) bekannt. Der Kon-

figurator wertet den Parameter `id` aus, und setzt eine Zustandsvariable in der Modelldefinition auf den entsprechenden Wert. Bewegungsereignisse können so vom Modell mit seiner `id` „signiert“ werden.

Verfeinerung Formale Verfeinerungsrelationen definieren, ob eine Implementierung eine Schnittstelle verfeinert und bilden damit die Voraussetzung, Kompositionen zu analysieren. `Motion` wird bspw. durch `MotionModel` verfeinert. Für alle deklarierten Ereignisports existiert ein Modellport in der Implementierung[1]. Damit das Modell Bewegungsinformationen generieren kann, benötigt es Pfadinformationen. Die Schnittstelle stellt dies als Anforderung aus (die minimale Multiplizität des Ports `path` ist eins).

6.2 Komposition

Kompositionen werden ausschließlich über Schnittstellenreferenzen und Kompositionsverbindungen zwischen ihnen ausgedrückt. Abbildung 6.4 veranschaulicht die konkrete XML-Syntax zur Definition von Kompositionen am Beispiel einer zusammengesetzten Komponente, die einen Nutzer repräsentiert. In der Komponente `User` kommt die Komponente `Motion` zum Einsatz. Des Weiteren komponiert `User` eine Aktivitätskomponente, die das Verhalten des Nutzers koordiniert und die Bewegungskomponente mit Pfadinformationen versorgt. Die Beziehung zwischen den beiden Unterkomponenten wird durch eine einzige Kompositionsverbindung ausgedrückt. Abbildung 6.5 veranschaulicht die Kompositionsverbindung und die durch sie verbundenen Rollen. Die entwickelten Beschreibungsmittel erleichtern das

Komponieren und Analysieren Die genutzte Aktivitätskomponente durch eine andere zu ersetzen, bedarf allein der Änderung der Schnittstellenreferenz für die Komposition mit dem Namen *activity*. Ob dann die alternative Aktivitätskomponente syntaktisch in den Einsatzkontext der Nutzerkomponente passt, lässt sich anhand der Kompositionsverbindungen automatisch überprüfen. Zu diesem Zweck wurde die Kompatibilität von Rollen formal definiert und auf Basis von XML-Schemamatching implementiert. Das Prinzip der Kompatibilitätsprüfung ist auf die semantische Kompatibilität übertragbar und die Voraussetzungen dafür wurden in den Typdefinitionen bereits angelegt (semantische Annotationen). Die pragmatische und konzeptuelle Kompatibilität lassen sich manuell anhand von Metadaten auswerten.

Innerhalb einer Komposition stellt die formale Eigenschaft *Vollständigkeit* sicher, dass die Anforderungen aller verwendeten Komponenten erfüllt sind. Die abgebildete Komposition ist so bspw. nicht vollständig, da die, durch den Port *movement*, deklarierte Anforderung der Komponente *Motion* nicht erfüllt ist. Diese Komposition bedarf weiterer Komponenten.

Experimentieren Die explizite semantische Abbildung von Komponenten auf PDEVS-Modelle garantiert unter der Voraussetzung korrekter Komponenten, dass sich Kompositionen in operationale PDEVS-Modelle abbilden lassen. Realisiert wurde die Anbindung des Simulationssystems James II. Simulationsmodelle lassen sich dort flexibel instrumentieren und mittels alternativer Simulationsalgorithmen ausführen.

6.3 Komposition von Manet

In Hinblick auf die Realisierung des oben abgebildeten konzeptuellen Modells wurde eine kleine Modellbibliothek für DIANE realisiert. Abbildung 6.6 visualisiert die zusammengesetzte Komponente *Manet* samt Unterkomponenten und Kompositionsalternativen. *Manet* selbst beinhaltet eine

[1] Für Statecharts lassen sich die Namen von Ereignissen als Portnamen interpretieren

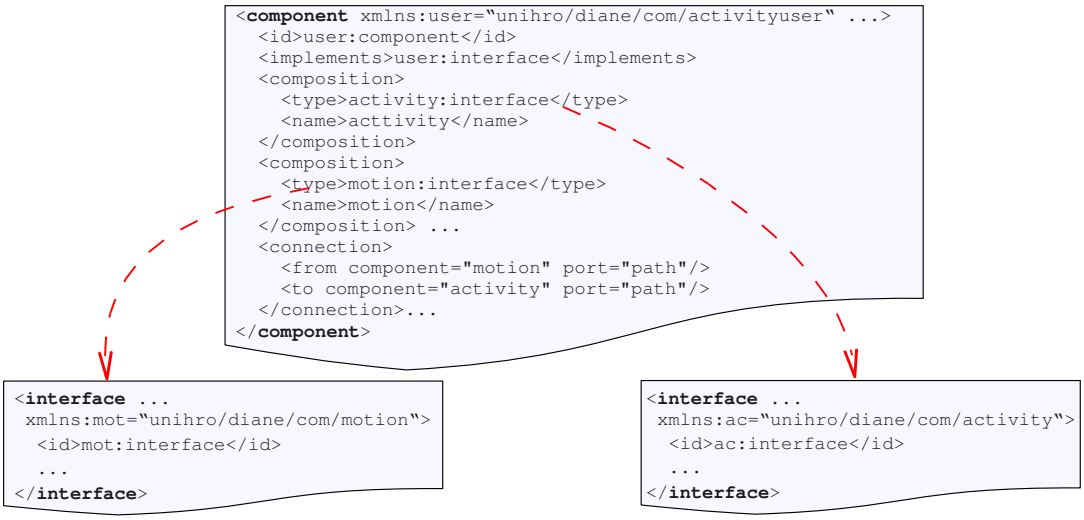

Abbildung 6.4: Beschreibung einer Komposition als zusammengesetzte Komponente

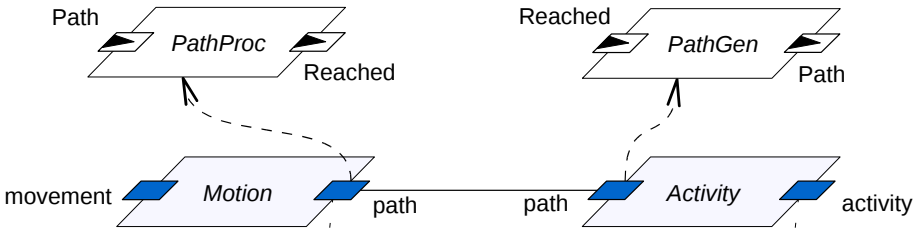

Abbildung 6.5: Kompositionsverbindungen und Kompatibilität

Menge von Netzwerkknoten und drei Komponenten, die unterschiedliche Aspekte der Umgebung von Knoten repräsentieren.

Die Repräsentation der Umgebung ist aufgeteilt in eine räumliche, eine netzwerktechnische und eine soziale Komponente. *Map* repräsentiert ein bestimmtes geographisches Gebiet, das Wege und Gebäude enthält. Die räumliche Umgebung verfolgt Bewegungen von Netzwerkknoten und ermittelt auf Grundlage der Knotenpositionen die physikalische Netztopologie. *Network* modelliert die Transportschicht des Netzwerks. *Network* nimmt gesendete Nachrichten von Knoten entgegen, ermittelt eine Route und stellt die Nachrichten den Empfängern zu. Grundlage für die Routenberechnung ist die physikalische Netzwerktopologie, über deren Änderungen *Network* von *Map* informiert wird. Das soziale Umgebungsmodell fungiert als zentralisierte Kommunikationsverbindung für zwischenmenschliche Handlungen der Netzwerknutzer. Auf Grundlage von physikalischer Nähe und gemeinsamen Zielen initiiert *SocialEnv* Gruppenbildung von Nutzern.

Die Komponente *Manet* stellt Parameter aus, um eine bestimmte Anzahl von Netzwerkknoten zu erstellen und den Typ des Nutzermodells sowie des Dienstvermittlungsprotokolls für jeden Knoten vorzugeben. Ein Knoten ist wiederum eine zusammengesetzte Komponente, die eine Nutzerkomponente und eine Protokollkomponente gruppiert. Die Komponente *trading* delegiert ihren Port *transport* an die Knotenkomponente. Innerhalb eines Knotens ist *trading* mit *user* verbunden.

Mit *Lanes* und *Flooding* stehen zwei alternative Vermittlungsprotokolle zur Komposition in einem Knoten zur Verfügung. *Lanes* (Klein u. a. 2003) ist ein Protokoll zur Vermittlung von Diensten auf Basis einer logischen Netzstruktur. In der logischen Struktur werden Knoten in Gruppen, genannt *lanes*, aufgeteilt, in denen jeder Knoten über vollständiges Wissen bezüglich der Dienstangebote

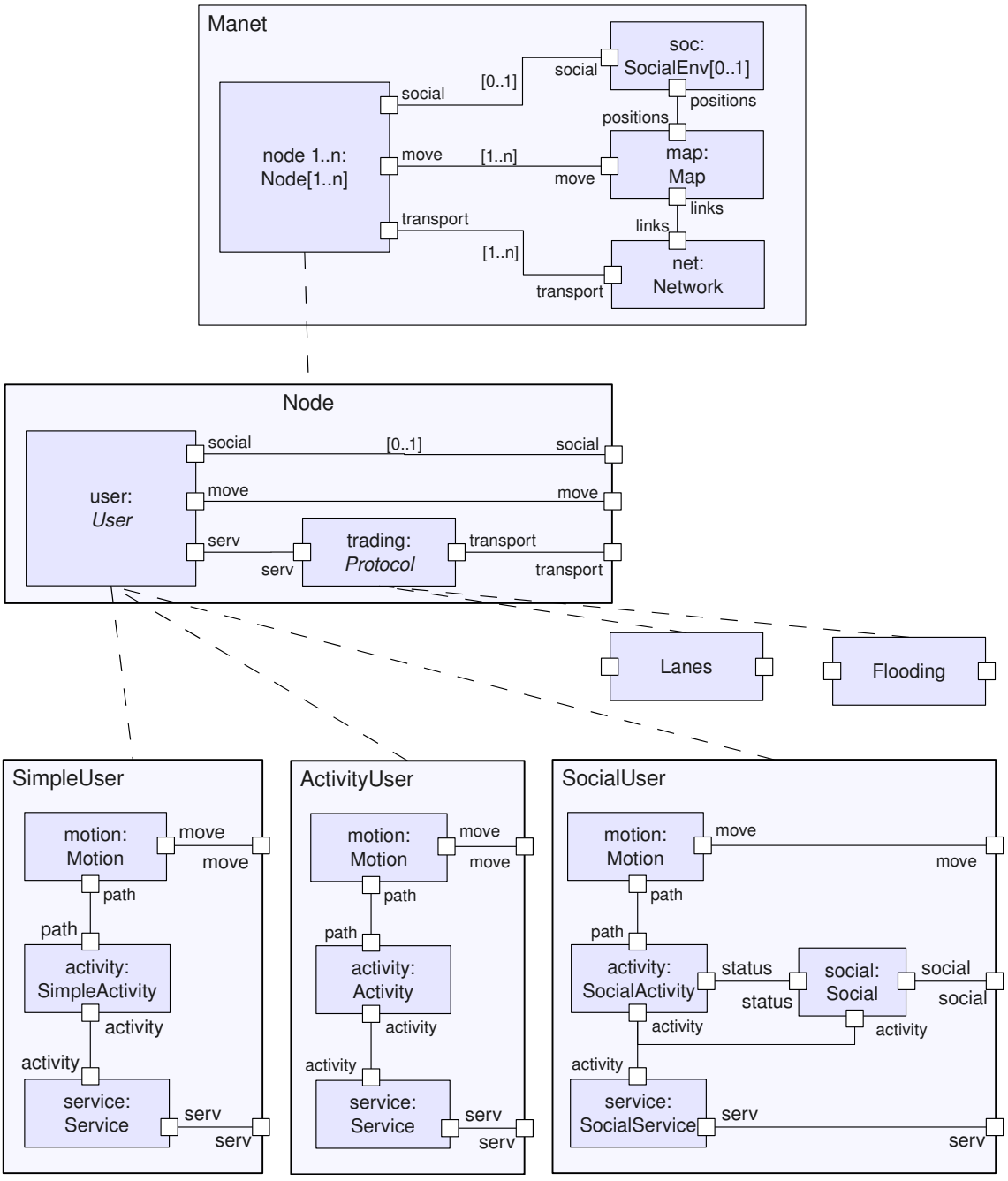

Abbildung 6.6: Kompositionstrukturen für MANETs

aller Gruppenmitglieder verfügt. Jeder Knoten kennt seinen Vorgänger- und Nachfolgerknoten innerhalb einer *lane* und die Gruppenadressen benachbarter *lanes*. Im Falle einer Dienstsuche kann ein Knoten das Dienstangebot innerhalb der eigenen *lane* direkt auswerten. Zusätzlich werden Dienstsuchen mittels *anycast* an die benachbarten *lanes* gesendet. Im Gegensatz zum *Lanes*-Protokoll baut *Flooding* keine logischen Netzstrukturen auf und bedarf keiner Wartungsoperationen. Im Falle einer Dienstsuche wird das gesamte Netz mittels *Broadcast*-Nachrichten geflutet.

Unabhängig von dem genutzten Vermittlungsprotokoll können drei alternative Nutzerkomponenten eingesetzt werden. Alle drei Nutzer sind als zusammengesetzte Komponenten realisiert. *SimpleUser* beinhaltet eine Komponente *activity*, die vorgibt, in welchem Zeitfenster ein Nutzer aktiv ist. Die Komponente *activity* aktiviert ein Nutzermodell gleichverteilt über einen Zeitraum von einer Stunde. In der aktiven Phase publiziert und sucht die Komponente *service* Dienste in Zeitabständen, die durch gleichverteilte Zufallszahlen bestimmt werden. Standardmäßig beträgt die Zeit zwischen zwei Dienstsuchen 360 Sekunden $\pm$120 Sekunden.

Des Weiteren wählt *activity* einen Zielpunkt aus und berechnet eine Route zu ihm. Routen werden zu *motion* propagiert, wo entsprechende, stückweise Bewegungen generiert werden. Ist der Zielpunkt erreicht, benachrichtigt *motion* die Komponente *activity*, so dass eine neue Route berechnet wird. Im einfachen Nutzermodell laufen Dienstverhalten und Bewegungsaktivitäten unabhängig voneinander ab.

Der aktivitätsbasierte Nutzer erweitert die Funktionalität des einfachen Nutzers, indem Dienstverhalten und Bewegungsaktivitäten auf Grundlage eines Tagesablaufes konsistent zueinander generiert werden. Dafür wurde in *ActivityUser* die Komponente *activity* ersetzt. Die Komponente *activity* erstellt einen Tagesablauf, der zeitlich vorgegebene Aktivitäten, wie bspw. das Besuchen einer Lehrveranstaltung, sowie zeitlich flexible Aktivitäten, wie bspw. Lernen, beinhaltet. Den unterschiedlichen Aktivitäten sind jeweils spezifische Intensitäten für die Nutzung von Diensten sowie spezielle Orte zugeordnet.

Die dritte Nutzervariante erweitert den aktivitätsbasierten Nutzer um soziale Verhaltensmuster (Hirschmeier 2006). Die Unterkomponente *social* gibt geplante Aktivitäten dem sozialen Umgebungsmodell bekannt und erhält von diesem Vorschläge zur Gruppenbildung. Nutzergruppen können sich gemeinsam bewegen und Dienste, neben der netzwerktechnischen Vermittlung, auch mündlich bekanntgeben.

Das Bewegungsmodell wird von allen drei Nutzerkomponenten wiederverwendet. Die Komponente *Service* wird vom einfachen und vom aktivitätsbasierten Nutzer verwendet.

6.4 Experimente

Die vorgestellten Komponenten werden nun zum Experimentieren genutzt. So sollen die beiden Protokolle *Lanes* und *Flooding* im Kontext ansteigender Netzwerkgröße verglichen werden (Röhl u. a. 2007a). In vorangegangenen Experimenten (Klein u. a. 2004) wurden die beiden Protokolle bereits für kleinere Knotenzahlen (≤ 64) verglichen[2]. Die Zielstellung der folgenden Experimente liegt darauf

- die beiden Protokolle *Lanes* und *Flooding* für höhere Knotenzahlen zu vergleichen und
- den Einfluss unterschiedlicher Nutzermodelle auf die Performance des Lanes-Protokolls zu untersuchen.

Um die erste Fragestellung zu beantworten, müssen Simulationsmodelle mit einer unterschiedlichen Anzahl von Knoten, jeweils mit *Lanes* oder *Flooding* erzeugt werden. Genau dies leistet die Komponente *Manet*. Abbildung 6.7 veranschaulicht eine beispielhafte Simulationskonfiguration für die Komponente *Manet*. CoMo generiert für diese Konfiguration ein ausführbares Simulationsmodell mit 100 Knoten, dem Vermittlungsprotokoll *Lanes* und dem einfachen Nutzermodell.

[2]Mit einem speziellen Simulator, der im Rahmen des DIANE-Projekts entwickelt wurde (Klein 2003).

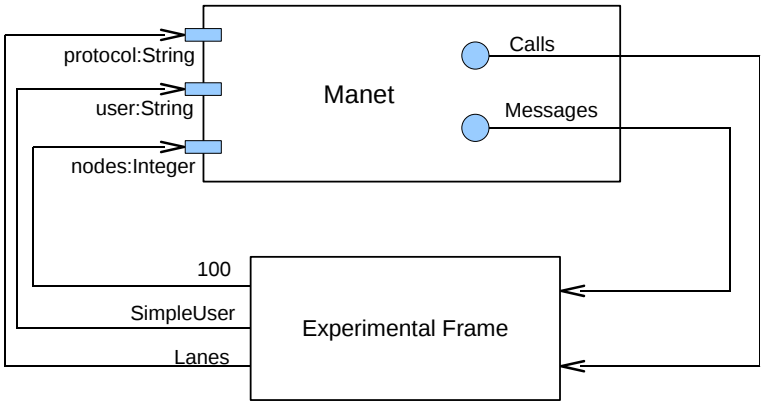

Abbildung 6.7: Experimenteller Rahmen mit Parametern und Beobachtungsmöglichkeiten

Abbildung 6.8 zeigt für die Konfiguration das abgeleitete Simulationsmodell. Das Simulationsmodell ist selbst nicht mehr parametrisierbar, sondern spiegelt die ausgewerteten Komponentenparameter in seiner Struktur. Zu tätigende Beobachtungen erfordern hingegen die Präsenz von Beobachtungsinstrumenten zur Laufzeit. So wird einmal der Ausgabeport *call* von Nutzern instrumentiert, um Dienstsuchen zu registrieren. Zum anderen wird der Zustand des Netzwerkmodells beobachtet, um Netzwerkkommunikation zu protokollieren[3].

Ein Simulationslauf mit obigem Modell liefert für eine vorgegebene Anzahl von Knoten und einem vorgegebenem Protokoll- sowie Nutzertyp Ausgabedaten über stattgefundene Dienstsuchen und das Nachrichtenaufkommen im Netzwerk. Die Ergebnisse des Experiments fasst Abbildung 6.9 zusammen. Das Diagramm zeigt die durchschnittliche Anzahl an Netzwerknachrichten, die für eine Dienstsuche benötigt werden. Entsprechend dieses Experiments eignet sich Fluten für Umgebungen mit relativ geringen Suchanfragen in kleineren Netzen. Mit steigender Größe des Netzwerkes erhöhen sich die Kosten pro Dienstsuche mittels Fluten. Für das simulierte Szenario stellen Netzwerke mit über 50 Knoten den Punkt dar, ab dem sich das Erstellen und die Wartung einer logischen Netzstruktur mittels *Lanes* im Vergleich zu Fluten auszahlt. Die Simulationläufe reproduzieren für kleine Knotenzahlen die Ergebnisse vorangegangener Experimente (Klein u. a. 2004) und legten damit die Voraussetzung für eine differenziertere Evaluierung des *Lanes*-Protokolls.

Die zweite zentrale Zielstellung für Experiment war, das *Lanes*-Protokoll im Kontext unterschiedlicher Nutzermodelle zu evaluieren. Abbildung 6.10 enthält exemplarische Simulationsläufe mit jeweils 400 Netzwerkknoten für die drei erstellten Nutzermodellen. Für Unterabbildung 6.10 a) kam das einfache Nutzermodell zum Einsatz. Die Trajektorien in Unterabbildung b) wurden mit dem aktivitätsbasierten Nutzer generiert und in c) wurde das soziale Nutzermodell eingesetzt.

Um die Netzwerklast zu evaluieren, wird zwischen drei Arten von Nachrichten unterschieden. Alle Nachrichten, die notwendig sind, um die *Lanes*-Struktur aufzubauen sind unter *login* zusammengefasst. Nachrichten zur Wartung von *Lanes*-Struktur sind mit *intra lanes* bezeichnet. Das Nachrichtenaufkommen für diese beiden Typen ist nahezu unabhängig von den Dienstanfragen des Nutzers. Unter *inter lane* sind diejenigen Nachrichten zusammengefasst, die notwendig sind, um Dienstanfragen von Nutzern zu beantworten. Für den einfachen Nutzer ergibt sich eine relativ gleichmäßige Verteilung der Netzwerklast über den gesamten Zeitraum, für den alle Nutzer im Netzwerk angemeldet sind. Mit dem aktivitätsbasierten Nutzermodell können hingegen unterschiedliche Perioden mit höherem und niederem Aufkommen an Dienstsuchen identifiziert werden. Diese Perioden reflektieren die unterschiedlichen Intensitäten der Dienstnutzung in verschiedenen Aktivitätsphasen. Im Simulationlauf mit dem sozialen Nutzermodell ist die Synchronisation weni-

[3] Eine mögliche Implementierung des Konfigurators für Beobachtungsinstrumente wurde in Abbildung 5.15 skizziert.

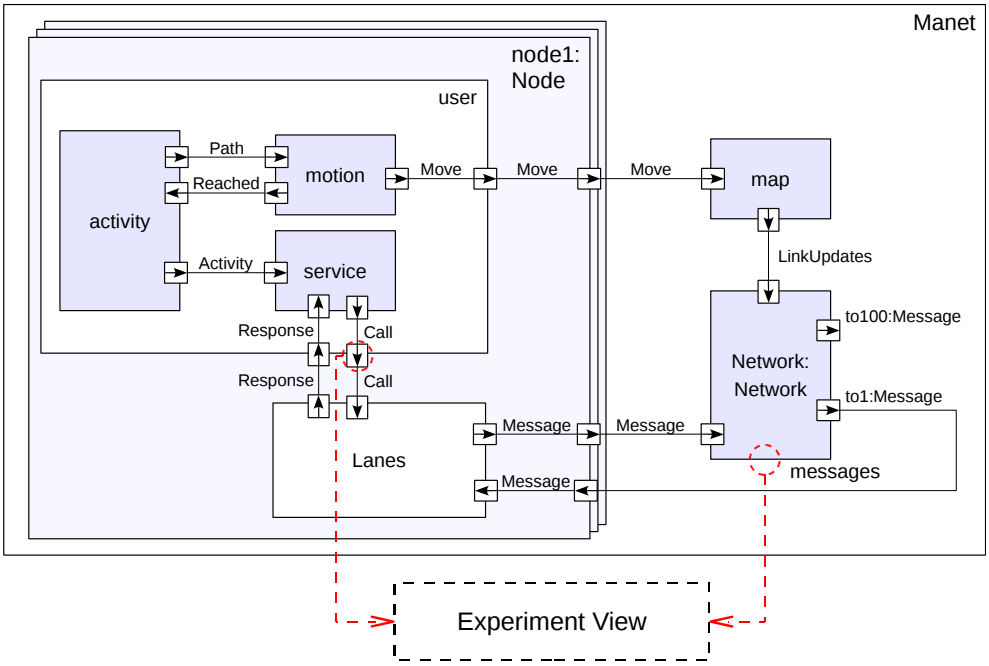

Abbildung 6.8: Struktur eines Simulationsmodells mit konfigurierten Beobachtungsinstrumenten

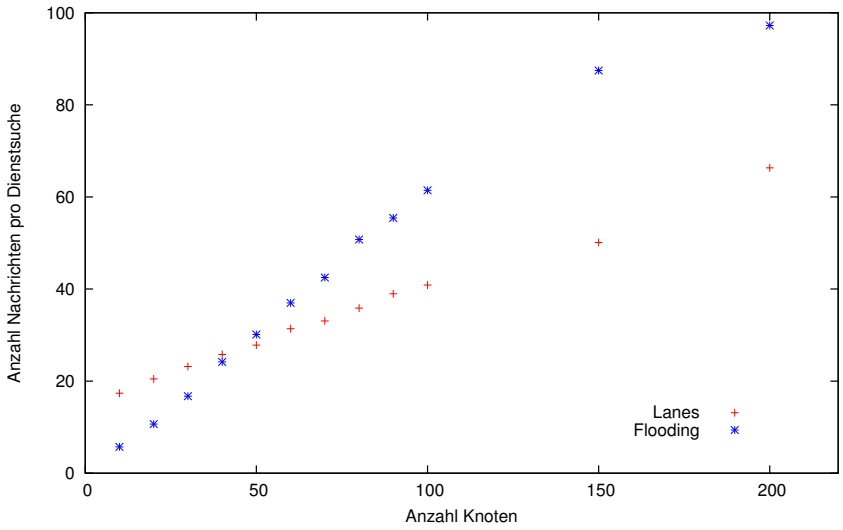

Abbildung 6.9: Durchschnittlicher Aufwand pro Dienstsuche mit Lanes und Fluten

ger ausgeprägt.

Abbildung 6.11 zeigt für die drei Simulationsläufe die Anzahl an Hops, die durchschnittlich benötigt wurden, um eine Netzwerknachricht zuzustellen. *Lanes*-Struktur werden auf Basis physikalischer Nähe gebildet. Knoten melden sich in einer *lane* an, die möglichst wenige Hops entfernt ist. So ist der Hopaufwand zu frühen Simulationszeitpunkten relativ gering. Aufgrund der Bewegung von Nutzern entfernen sich Knoten jedoch voneinander. Dadurch „degenerieren“ *Lanes*-Struktur.

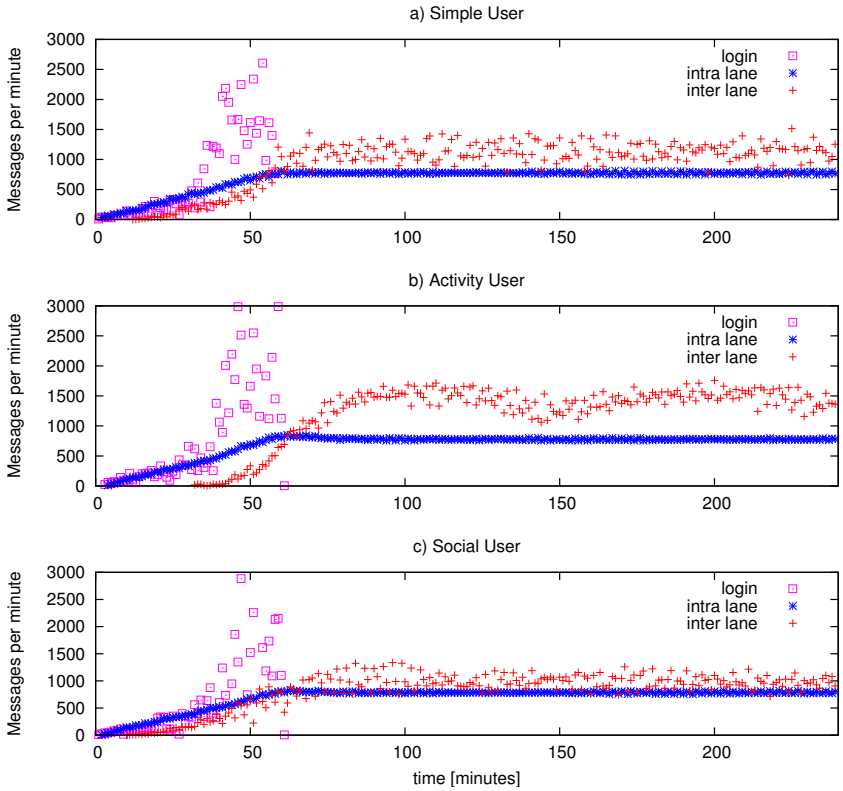

Abbildung 6.10: Absolute Anzahl von Netzwerknachrichten pro Minute

Der Aufwand, um innerhalb einer *lane* zu kommunizieren, steigt mit zunehmender Simulationszeit. Dieser Effekt ist für das aktivitätsbasierte und das soziale Nutzermodell stärker ausgeprägt, als für den einfachen Nutzer. Aus diesen Experimenten ist der Wunsch entstanden, dass *Lanes*-Strukturen sich an verändernde Netzwerktopologien anpassen können sollen (Röhl u. a. 2007a).

Im Zuge der weiteren Integration der Kompositionsplattform mit dem Simulationswerzeug James II, um bspw. Parameterkombinationen automatisch und wiederholt zu simulieren, wurde die Laufzeit des DIANE-Modells evaluiert (Himmelspach u. Röhl 2008). Abbildung 6.12 veranschaulicht die Ausführungszeiten mit dem einfachen und dem sozialen Nutzermodell für Netzwerke mit einer Größe von 50 bis 300 Knoten. Für jede Parameterkombination wurden zehn Replikationen durchgeführt. Der Plot zeigt die Durchschnittswerte und Standardabweichungen der Ausführungszeiten auf einer logarithmischen Skala. Die Zeiten demonstrieren, dass sich auch große MANET-Modelle (ca. 250 Knoten) noch in Echtzeit (ca. zwei Stunden Ausführungszeit für zwei Stunden simulierte Zeit) simulieren lassen.

Im Rahmen einer noch laufenden Masterarbeit (Borchardt 2008) werden die entwickelten Nutzer- und Protokollkomponenten genutzt, um Experimente mit anderen Zielstellungen durchzuführen. Der Fokus liegt dabei auf der Evaluierung von Vertrauensbildung in MANETs auf der Grundlage von Kontextinformationen. Durch das Mithören von Netzwerknachrichten auf der zweiten Schicht des OSI-Stacks (Sicherungsschicht, engl. *data link layer*) sollen Nutzer als kooperativ bzw. unkooperativ erkannt werden. Da von dieser Schicht im zentralen Netzwerkmodell abstrahiert wurde, ist *diese* Komponente nicht für die neuen Experimente nutzbar. Stattdessen erhält jeder Netzwerknoten eine eigene Komponente zur Repräsentation der Sicherungsschicht.

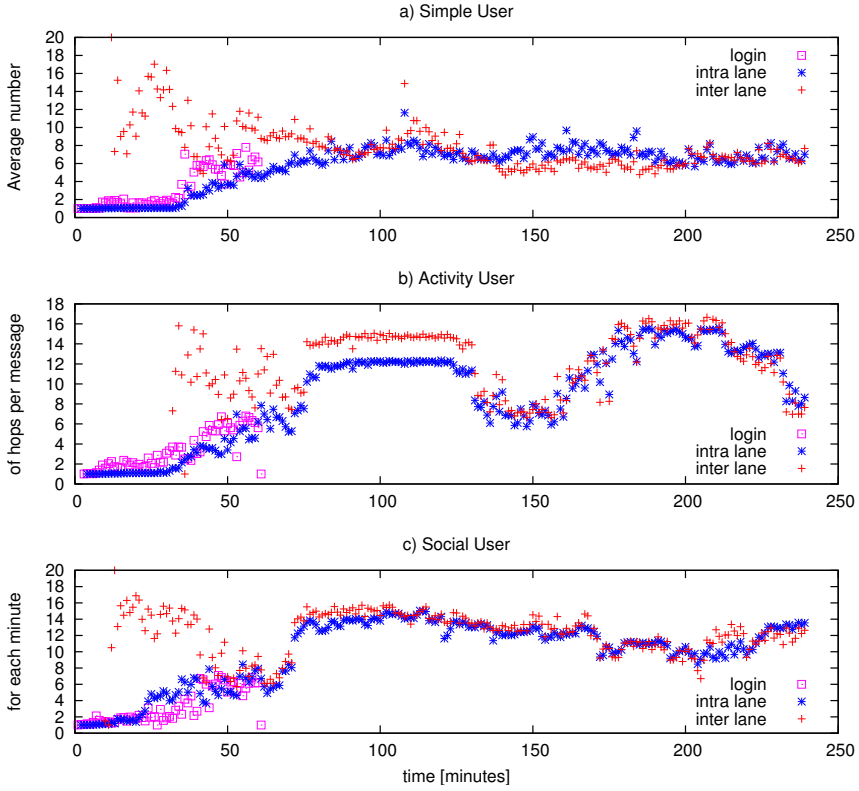

Abbildung 6.11: Durchschnittliche Anzahl von Hops für ein Nachricht

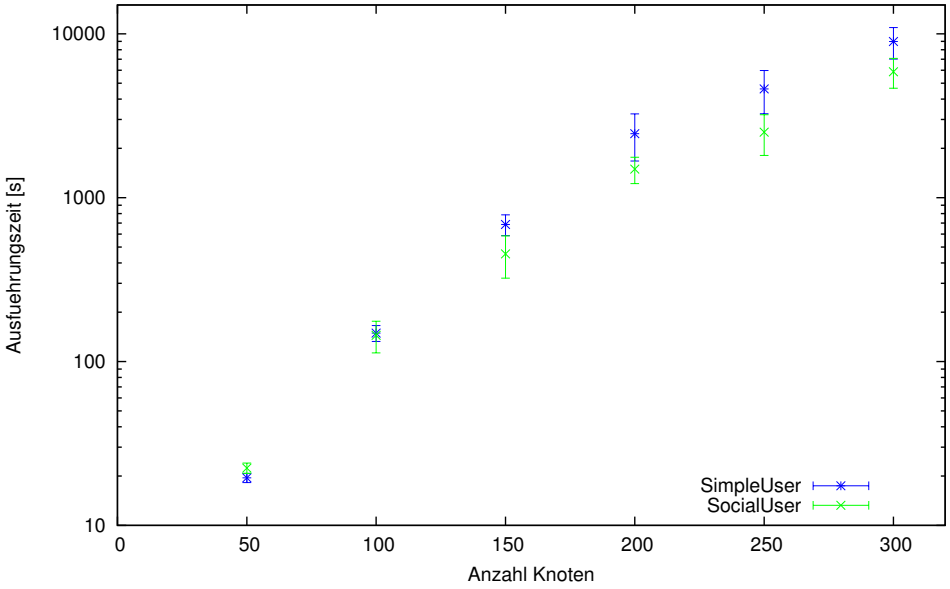

Abbildung 6.12: Ausführungszeiten für unterschiedliche Anzahl von Knoten

7 Zusammenfassung und Ausblick

Es wurde eine Komponentenplattform konzipiert und realisiert, auf der Simulationsmodelle modellbasiert komponiert werden können. Die Plattform definiert explizite Beschreibungsmittel für Schnittstellen, Komponenten und Kompositionen. Schnittstellendefinitionen basieren auf Rollenbeschreibungen, die ihrerseits auf Typdefinitionen verweisen. Als separate XML-Dokumente können öffentliche Beschreibungen in Datenbanken hinterlegt werden, um Kompositionskandidaten aufzufinden, zu analysieren und zu selektieren. Mit den Beschreibungsmitteln der Plattform definierte Komponenten lassen sich in Kompositionen einsetzen und in ausführbare Simulationsmodelle abbilden. Komponenten können für eine konkrete Fragestellung mit einer Menge von Parametern konfiguriert und im Rahmen einer konkreten Einsatzplattform mit Beobachtungsmöglichkeiten versehen werden. Die entwickelte Komponentenplattform bietet Entwicklern und Anwendern von Modellkomponenten folgende Vorteile:

- Die Ausdrucksmittel von Schnittstellen- und Komponentendefinitionen komplementieren existierende Modellierungsformalismen.
- Schnittstellendefinitionen lassen sich selbst unterteilen, mittels URIs transparent verteilen und als eigenständige Definitionseinheiten wiederverwenden.
- Ein Metamodell definiert die syntaktischen Mittel und die Semantik von Komponenten und Kompositionen formal.
- Aus korrekten Komponenten lassen sich wohlgeformte operationale Modelle ableiten.
- Die Definitionseinheiten der Plattform sind in XML-Dokumenten repräsentierbar, deren Struktur mittels des W3C-Standards *XML Schema Definitions* definiert wurde.
- Das Kompositionswerkzeug CoMo realisiert die Kompositionsplattform. Kompositionen lassen sich mit CoMo instantiieren und werden im Zuge der Abbildung auf ein Simulationsmodell auf Kompatibilität, Vollständigkeit und Verfeinerung geprüft.
- Für CoMo existiert eine Anbindung an das Simulationssystem James II, mit dem Simulationsmodelle ausgeführt werden können.
- Ein reales Anwendungsbeispiel dokumentiert die Verwendungsmöglichkeiten der Plattform.

Diese Arbeit beansprucht Schnittstellenbasiertheit, Formalität und Praxisbezogenheit erstmals in einem Kompositionsansatz für Modellkomponenten zu vereinen. Tabelle 7.1 schlüsselt die Eigenschaften der entwickelten Komponentenplattform in Hinblick auf die Anforderungen an Kompositionsansätze auf.

Es lassen sich eine Reihe nützlicher Erweiterungen und Ergänzungen für die Kompositionsplattform identifizieren.

Plattformunabhängige Definition von Beobachtungsinstrumenten Für die entwickelte Komponentenplattform existiert noch keine ausgereifte Lösung, um Beobachtungen flexibel und plattformunabhängig zu konfigurieren. Die Instrumentierung mit Observern muss bisher als spezifische Lösung für eine konkrete Zielplattform implementiert werden. In der Modellierung und Simulation beginnen sich Ansätze zu entwickeln, um Simulationsmodelle systematisch zu instrumentieren

Kriterium	*Grad*	*Bemerkungen*
Spezifiz., Publiz. & Auffinden		
Austauschformat	+	XML-basiertes Format, definiert im W3C-Standard XSD
Separate Schnittstelle	+	Schnittstellen, Rollen und Typen lassen sich unabhängig von Modellierungsformalismen in jeweils separaten Dokumenten definieren.
Implementieren & Verifizieren		
Modellierung	+	Modellierungsformalismen können innerhalb von Komponenten frei gewählt werden, sofern sie gerichtete Ereignisports unterstützen. Komponenten lassen sich modular-hierarchisch zusammensetzen.
Parametrisierung	+	Schnittstellen deklarieren Parameter. Eine Komponente beinhaltet eine Funktion zur Auswertung von Parametern.
Verfeinerung	○	Formale Verfeinerungsrelationen definieren, wann eine Implementierungen eine Schnittstelle verfeinert. Ereignistypen von Modellimplementierungen lassen sich aus XSD-Dokumenten mittels JAXB generieren und sind damit automatisch konsistent zu publizierten Typdefinitionen.
Komponieren & Analysieren		
Technische Ebene	○	Von der technischen Dimension der Kompatibilität wird weitestgehend abstrahiert. Die Integration von Modellen ist theoretisch sowohl mittels Modelltransformation als auch durch die Kopplung heterogener Modelle möglich. Dies muss jedoch von einer Zielplattform geleistet werden. Für das Simulationsystem James II sind PDEVS und SCXML mit CoMo komponierbar.
Syntax	+	Ereignistypen werden vollständig mit XSD definiert. In CoMo sind, unabhängig voneinander entwickelte XSD-Typen automatisch vergleichbar.
Semantik	○	Typdefinitionen in XSD lassen sich mit semantischen Informationen ähnlich zu SAWSDL bereichern.
Pragmatik	○	Natürlichsprachliche Metadaten; manuell auszuwerten
Dynamik	○	Rollenbeschreibungen geben die sichtbaren Aktionen für dynamische Beschreibungen vor.
Konzepte	○	Natürlichsprachliche Metadaten in Schnittstellen.
Validität	○	Natürlichsprachliche Metadaten. Modellkomponenten können für konkrete Fragestellungen konfiguriert, für eine Zielplattform in eine ausführbare Form gebracht und mit Beobachtungsinstrumenten versehen werden.
Experimentieren	+	Aus korrekten Komponenten lassen sich wohlgeformte Simulationsmodelle ableiten. CoMo generiert ein operationales Modell für das Simulationssystem James II.

Tabelle 7.1: Abdeckung der Anforderungen durch die entwickelte Komponentenplattform

(Dalle u. Mrabet 2007). Eine Möglichkeit ist, dass Komponenten die potentiell beobachtbaren Ereignisse in ihren Schnittstellen deklarieren (Röhl u. a. 2007b). Analog zu Interaktionspunkten für die Kommunikation zwischen Modellen, sollten Schnittstellen Beobachtungsmöglichkeiten mittels Typbeschreibungen in XSD deklarieren, so dass plattformspezifische Beobachtungsinstrumente generiert werden können.

Formale Beschreibungsmittel für den dynamischen Aspekt von Schnittstellen Formale Analysetechniken auf Basis von Prozessalgebren und LTS sind prinzipiell in ihrer Verwendung nicht auf Software begrenzt, sondern lassen sich auch für Simulationsmodelle nutzen. Schwache Bisimulation auf der Basis von LTS erscheint als besonders geeigneter Kandidat um dynamische Beschreibungen formal zu analysieren. Schwache Bisimulation abstrahiert von nichtsichtbaren Ereignissen und korrespondiert mit dem Zusammenfassen von Kommunikationsfähigkeiten bzgl. eines bestimmten Aspekts in Rollen. Darüber hinaus bieten LTS Anknüpfungspunkte zur Theorie der Semantischen Komponierbarkeit. Zu identifizieren, welche konkrete Algebra zur Beschreibung in Schnittstellen für Simulationsmodelle sinnvoll ist, stellt jedoch eine beträchtliche theoretische Herausforderung dar.

Deklaration struktureller Variabilität Schnittstellen deklarieren momentan rein statische Strukturen. Bisher sind variable Strukturen nur innerhalb einer atomaren Komponente erlaubt. Eine Komposition wird vollständig vor der Ausführungsphase analysiert und ein Teil der Implementierung auf Basis der Kompatibilitätsprüfung generiert. Zur Laufzeit ist kein Überprüfen mehr möglich, da sämtliche Analyseschritte aus Effizienzgründen vor die Ausführungsphase verlegt wurden. Um dynamische Strukturveränderungen über Komponentengrenzen hinaus analysieren zu können, müssen sämtliche strukturellen Möglichkeiten vor der Simulationsausführung auf Konsistenz geprüft werden. Ein aussichtsreicher Kandidat ist der π-Kalkül (Milner 1999).

Integration vollständig XML-basierter Modellierungsformalismen in CoMo Um Modellkomponenten unabhängig von Zielplattformen entwickeln und flexibel einsetzen zu können, sind XML-basierte Austauschformate sinnvoll. Es ist wünschenswert, weitere Modellierungsformalismen in Verbindung mit entsprechenden Transformatoren nach PDEVS in CoMo zu integrieren. Analog zur prototypischen Integration von Statecharts bieten sich UML Zustandsmaschinen als standardisierte und vollständig in XML repräsentierbare Notation an.

Vergleich von semantischen Beschreibungen Rollendefinitionen und Kompatibilitätsprüfung wurden auf syntaktischer Ebene vollständig definiert und in CoMo realisiert. Analog zur syntaktischen Ebene sollten semantische Beschreibungen automatisch ausgewertet werden können, um semantische Variationen, bspw. *is a*-Beziehungen, je nach Bedarf, als kompatibel bzw. inkompatibel zu erkennen. Für die Nutzung der semantischen Beschreibungen bedarf es jedoch anwendungsspezifischer Ontologien.

Graphische Modellierung und Implementierung der formaler Analysemethoden Rollen, Schnittstellen, Komponenten, Modelle und Simulationsläufe lassen sich bisher nur manuell in XML-Dokumenten definieren, automatisch gegen ihre Schemata (aus Anhang B) validieren und mittels Adaptoren prüfen. Kompositionale Eigenschaften werden bisher nur im Zuge der Abbildung auf ein Simulationsmodell geprüft. Die entwickelten Beschreibungsformen ließen sich über eine graphische Benutzungsschnittstelle wesentlich einfacher nutzen. Syntaktische Anforderungen und kompositionale Eigenschaften, wie Verfeinerung, Kompatibilität und Vollständigkeit, sollten dann direkt durch das Werkzeug, zum Zeitpunkt der Spezifikation prüfbar sein.

Ausführung von Kompositionen mit simulationsbasierten Ansätzen Von den zwei in Abschnitt 4.1.7 dargestellten Alternativen zur Abbildung einer Komposition auf ein Simulationsmodell wurde in der vorliegenden Arbeit nur die Transformationsvariante über einen Superformalismus realisiert. Alternativ zu Formalismentransformationen ist die Abbildung von Kompositionsstrukturen auf Beschreibungsmittel sinnvoll, die es ermöglichen, die Heterogenität der Modellimplementierungen auch für die Ausführung zu bewahren. Bei heterogenen Formalismen wäre dies mit Ptolemy möglich. Für heterogene Simulationswerkzeuge bietet sich die HLA an.

Komponentenorientierte Werkzeugarchitektur Der Fokus lag in dieser Arbeit darauf, die prinzipiellen Funktionen der Komponentenplattform möglichst konsistent zu ihrer formalen Definition zu realisieren. Für die softwaretechnische Umsetzung bieten die in Kapitel 3 vorgestellten Ansätze aus CBSE und SOA die Möglichkeit, CoMo in wiederverwendbare Softwarebausteine zu gliedern.

In diesem Rahmen gilt es bspw. auch zu lösen, dass Modellimplementierungen, die am Einsatzort zusammengefügt werden, vom Einsetzenden nicht vollständig einsehbar sind. Modellimplementierungen in Austauschformaten bereitzustellen, ist nicht immer erwünscht. Der Schutz privater Implementierungen ließe sich technisch im Rahmen der Kommunikationsinfrastruktur zwischen Komponentenhersteller und -nutzer sicherstellen. Implementierungen könnten auf Seiten des Komponentenherstellers mittels Diensten bereitgestellt werden, die Modelle für Zielplattformen generieren. Alternativ ist auch denkbar, dass Modellimplementierungen nur von signierten Werkzeugimplementierungen und ausschließlich in der Kompositionsphase eingesehen werden dürfen.

A Beweise zur Plattformdefinition

A.1 Beweis von Lemma 1

Wiederholung von Lemma 1: Gegeben seien eine Kompositionsverbindung $c = (start, end)$ und zwei Schnittstellenreferenzen s, s'. Ist c wohlgeformt, konstruiert EC wohlgeformte atomare Verbindungen:

$$\begin{aligned} wellformed(c, s, s') \quad \Rightarrow \forall ec \in EC(c, s, s'). \quad & start_{ec} = start_c \wedge wellformed(ec, s, s') \\ & \vee start_{ec} = end_c \wedge wellformed(ec, s', s) \end{aligned}$$

Beweis. Aus $wellformed(c, s, s')$ folgt, dass $\mathscr{R} = role(\mathscr{I}, start)$ und $\mathscr{R}' = role(\mathscr{I}', end)$ mit $\mathscr{I} = drefI(name_s)$ und $\mathscr{I}' = drefI(name_{s'})$ existieren. Nach Definition 4.10 gilt für eine Kompositionsverbindung bereits, dass $if_{start} \neq if_{end}$. Im Folgenden wird gezeigt, dass jedes $ec \in EC(c, s, s')$ existente Ereignisports verbindet und die Bedingungen für die Wertebereiche der Ports einhält.

1. Falls $if_{start} =$ „this“, folgt aus $wellformed(c, s, s')$, dass $\overline{\mathscr{R}} \sim \mathscr{R}'$. Damit existiert $M = matches(\mathscr{R}, \overline{\mathscr{R}'})$ und für alle $m \in M$ mit $m = (from, to)$ sind zwei Fälle möglich.
   a) Existiert ein $x \in EP_{\mathscr{R}}$ mit $name_x = from$ und inp_x, ist $ec = (start, from, end, to)$. Entsprechend der Definition von $\overline{\mathscr{R}}$ existiert ein $y \in EP_{\overline{\mathscr{R}}}$ mit $name_y = from$, $\neg inp_y$ und $tid_y = tid_x$. Nach der Definition von $matches(\overline{\mathscr{R}}, \mathscr{R}')$ existiert ein Eingabeport $x' \in EP_{\mathscr{R}'}$ mit $name_{x'} = to$ und $inp_{x'}$, für den $drefT(tid_y) \sqsubseteq drefT(tid_{x'})$ und somit $drefT(tid_x) \sqsubseteq drefT(tid_{x'})$. Damit sind alle Bedingungen für $wellformed(ec, s, s')$ erfüllt.
   b) Existiert ein Port $y \in EP_{\mathscr{R}}$ mit $name_y = from \wedge \neg inp_y$, ist $ec = (end, to, start, from)$. Nach der Definition von $matches(\overline{\mathscr{R}}, \mathscr{R}')$ existiert dann ein Ausgabeport $y' \in EP_{\mathscr{R}'}$ mit $name_{y'} = to \wedge \neg inp_{y'}$, für den $drefT(tid_{y'}) \sqsubseteq drefT(tid_y)$. Damit sind alle Bedingungen für $wellformed(ec, s', s)$ erfüllt.

2. Falls $if_{end} =$ „this“ folgt aus $wellformed(c, s, s')$, dass $\mathscr{R} \sim \overline{\mathscr{R}'}$. Damit existiert $M = matches(\mathscr{R}, \overline{\mathscr{R}'})$ und für alle $m \in M$ mit $m = (from, to)$ sind zwei Fälle möglich.
   a) Existiert ein $x \in EP_{\mathscr{R}}$ mit $name_x = from \wedge inp_x$, ist $ec = (end, to, start, from)$. Nach der Definition von $matches(\mathscr{R}, \overline{\mathscr{R}'})$ existiert ein Eingabeport $x' \in EP_{\mathscr{R}'}$ mit $name_{x'} = to$ und $drefT(tid_{x'}) \sqsubseteq drefT(tid_x)$. Damit gilt $wellformed(ec, s', s)$.
   b) Existiert ein $y \in EP_{\mathscr{R}}$ mit $name_y = from \wedge \neg inp_y$, ist $ec = (start, from, end, to)$. Nach der Definition von $matches(\mathscr{R}, \overline{\mathscr{R}'})$ existiert ein Ausgabeport $y' \in EP_{\mathscr{R}'}$ mit $name_{y'} = to$ und $\neg inp_{y'}$, für den $drefT(tid_y) \sqsubseteq drefT(tid_{y'})$. Damit gilt $wellformed(ec, s, s')$.

3. Falls $if_{start} \neq$ „this“ $\wedge\ if_{end} \neq$ „this“ folgt aus $wellformed(c, s, s')$, dass $\mathscr{R} \sim \mathscr{R}'$. Dementsprechend existiert $M = matches(\mathscr{R}, \mathscr{R}')$, so dass für alle $m \in M$ mit $m = (from, to)$ zwei Fälle möglich sind.
   a) Existiert ein $y \in EP_{\mathscr{R}}$ mit $name_y = from$ und $\neg inp_y$, ist $ec = (start, from, end, to)$. Nach der Definition von $matches(\mathscr{R}, \mathscr{R}')$ existiert ein $x' \in EP_{\mathscr{R}'}$ mit $name_{x'} = to$ und $inp_{x'}$, für den $drefT(tid_y) \sqsubseteq drefT(tid_{x'})$. Damit gilt $wellformed(ec, s, s')$.
   b) Existiert ein Port $x \in EP_{\mathscr{R}}$ mit $name_x = from$ und inp_x, ist $ec = (end, to, start, from)$. Nach der Definition von $matches(\mathscr{R}, \mathscr{R}')$ existiert ein $y' \in EP_{\mathscr{R}'}$ mit $name_{y'} = to$ und $\neg inp_{y'}$, für den $drefT(tid_x) \sqsubseteq drefT(tid_{y'})$. Dann gilt $wellformed(ec, s', s)$.

□

A.2 Beweis von Lemma 2

Wiederholung von Lemma 2: Gegeben seien eine Kompositionsverbindung c, zwei Schnittstellenreferenzen s, s', mit $\mathscr{I} = drefI(s)$ und $\mathscr{I}' = drefI(s')$, zwei Bindungen $\mathscr{B}, \mathscr{B}'$ und zwei PDEVS-Modelle $\mathscr{M}, \mathscr{M}'$. Werden die beiden Schnittstellen durch ihre Modelle und Bindungen bewahrend verfeinert und durch eine wohlgeformte Kompositionsverbindung verbunden, dann sind alle durch MC konstruierten Modellkopplungen wohlgeformt:

$$\mathscr{I} \succ_p (\mathscr{M}, \mathscr{B}) \wedge \mathscr{I}' \succ_p (\mathscr{M}', \mathscr{B}') \wedge wellformed(c, s, s') \Rightarrow \forall mc \in MC(c, s, s').wellformed(mc)$$

Beweis. Die Funktion MC nutzt für jede atomare Verbindung $ec \in EC(c, s, s')$ die Funktion mc, die eine atomare Verbindung für zwei Bindungen $\mathscr{B}, \mathscr{B}'$ auf eine Modellkopplung abbildet. Aus Lemma 1 folgt, dass für eine wohlgeformte Kompositionsverbindung c, für alle $ec =\in EC(c, s, s')$ entweder (ec, s, s') oder (ec, s, s') wohlgeformt ist und damit, dass $mc(ec, s, s', \mathscr{B}, \mathscr{B}') \neq \bot$. Seien im Folgenden $c = (start, end)$, $\mathscr{R} = role(\mathscr{I}, start)$ und $\mathscr{R}' = role(\mathscr{I}', end)$. Nach Lemma 1 sind für jedes $ec =\in EC(c, s, s')$, mit $ec = (k, n, k', n')$, in der Funktion mc zwei Fälle zu unterscheiden.

1. Falls $wellformed(ec, s, s')$, ist $if_k = name_s$ und $mc = (name_s, fp, name_{s'}, tp)$, mit $fp = impl(port_k, n, pos_k, \mathscr{B})$ und $tp = impl(port_{k'}, n', pos_{k'}, \mathscr{B}')$. Aus der Wolgeformtheit von ec folgt, dass ein $e \in EP_{\mathscr{R}}$ sowie ein $e' \in EP_{\mathscr{R}'}$ existieren, mit $name_e = n$, $name_{e'} = n'$ sowie $\mathscr{T} \sqsubseteq \mathscr{T}'$, mit $\mathscr{T} = drefT(type_e)$ sowie $\mathscr{T}' = drefT(type_{e'})$. Für fp und tp sind drei Fälle zu unterscheiden.
   a) Falls $if_k =$ „this“, folgt aus $wellformed(ec, s, s')$, dass inp_e und $inp_{e'}$. Aus $\mathscr{I} \succ_p (\mathscr{M}, \mathscr{B})$ und $\mathscr{I}' \succ_p (\mathscr{M}', \mathscr{B}')$ folgt, dass $fp \in InPorts_{\mathscr{M}}$ sowie $tp \in InPorts_{\mathscr{M}'}$, für die $X_{fp,\mathscr{M}} = car_{\mathscr{T}}$ bzw. $X_{tp,\mathscr{M}'} = car_{\mathscr{T}'}$. In Kombination mit $\mathscr{T} \sqsubseteq \mathscr{T}'$ gilt dann, dass $X_{fp,\mathscr{M}} \subseteq X_{tp,\mathscr{M}'}$.
   b) Falls $if_k =$ „this“, folgt aus $wellformed(ec, s, s')$, dass $\neg inp_e$ und $\neg inp_{e'}$. Aus $\mathscr{I} \succ_p (\mathscr{M}, \mathscr{B})$ und $\mathscr{I}' \succ_p (\mathscr{M}', \mathscr{B}')$ folgt, dass $fp \in OutPorts_{\mathscr{M}}$ und $tp \in OutPorts_{\mathscr{M}'}$, für die $Y_{fp,\mathscr{M}} = car_{\mathscr{T}}$ bzw. $Y_{tp,\mathscr{M}'} = car_{\mathscr{T}'}$. Aus $\mathscr{T} \sqsubseteq \mathscr{T}'$ folgt dann $Y_{fp,\mathscr{M}} \subseteq Y_{tp,\mathscr{M}'}$.
   c) Falls $if_k \neq$ „this“ und $if_{k'} \neq$ „this“, folgt aus $wellformed(ec, s, s')$, dass $\neg inp_e$ und $inp_{e'}$. Aus $\mathscr{I} \succ_p (\mathscr{M}, \mathscr{B})$ und $\mathscr{I}' \succ_p (\mathscr{M}', \mathscr{B}')$ folgt, dass $fp \in OutPorts_{\mathscr{M}}$ und $tp \in InPorts_{\mathscr{M}'}$, für die $Y_{fp,\mathscr{M}} = car_{\mathscr{T}}$ bzw. $X_{tp,\mathscr{M}'} = car_{\mathscr{T}'}$. Da $\mathscr{T} \sqsubseteq \mathscr{T}'$ gilt $Y_{fp,\mathscr{M}} \subseteq X_{tp,\mathscr{M}'}$.

2. Falls $wellformed(ec, s', s)$, ist $if_k = name_{s'}$ und $mc = (name_s, fp, name_{s'}, tp)$, mit $fp = impl(port_{k'}, n', pos_{k'}, \mathscr{B}')$ und $tp = impl(port_k, n, pos_k, \mathscr{B})$. Aus der Wolgeformtheit von (ec, s', s) folgt, dass ein $e' \in EP_{\mathscr{R}'}$ sowie ein $e \in EP_{\mathscr{R}}$ existieren, mit $name_{e'} = n'$, $name_e = n$ sowie $\mathscr{T}' \sqsubseteq \mathscr{T}$, wobei $\mathscr{T} = drefT(type_e)$ sowie $\mathscr{T}' = drefT(type_{e'})$. Für fp und tp sind wiederum drei Fälle zu unterscheiden.
   a) Falls $if_k =$ „this“, folgt aus $wellformed(ec, s', s)$, dass $inp_{e'}$ und inp_e. Aus $\mathscr{I}' \succ_p (\mathscr{M}', \mathscr{B}')$ und $\mathscr{I} \succ_p (\mathscr{M}, \mathscr{B})$ folgt, dass $fp \in InPorts_{\mathscr{M}'}$ sowie $tp \in InPorts_{\mathscr{M}}$, für die $X_{fp,\mathscr{M}'} = car_{\mathscr{T}'}$ bzw. $X_{tp,\mathscr{M}} = car_{\mathscr{T}}$. In Kombination mit $\mathscr{T}' \sqsubseteq \mathscr{T}$ gilt dann, dass $X_{fp,\mathscr{M}'} \subseteq X_{tp,\mathscr{M}}$.
   b) Falls $if_{k'} =$ „this“, folgt aus $wellformed(ec, s', s)$, dass $\neg inp_{e'}$ und $\neg inp_e$. Aus $\mathscr{I}' \succ_p (\mathscr{M}', \mathscr{B}')$ und $\mathscr{I} \succ_p (\mathscr{M}, \mathscr{B})$ folgt, dass $fp \in OutPorts_{\mathscr{M}'}$ und $tp \in OutPorts_{\mathscr{M}}$, für die $Y_{fp,\mathscr{M}'} = car_{\mathscr{T}'}$ bzw. $Y_{tp,\mathscr{M}} = car_{\mathscr{T}}$. Aus $\mathscr{T}' \sqsubseteq \mathscr{T}$ folgt dann $Y_{fp,\mathscr{M}'} \subseteq Y_{tp,\mathscr{M}}$.
   c) Falls $if_k \neq$ „this“ und $if_{k'} \neq$ „this“, folgt aus $wellformed(ec, s', s)$, dass $\neg inp_{e'}$ und inp_e. Aus $\mathscr{I}' \succ_p (\mathscr{M}', \mathscr{B}')$ und $\mathscr{I} \succ_p (\mathscr{M}, \mathscr{B})$ folgt, dass $fp \in OutPorts_{\mathscr{M}'}$ und $tp \in$

$InPorts_{\mathscr{M}}$, für die $Y_{fp,\mathscr{M}'} = car_{\mathscr{T}'}$ bzw. $X_{tp,\mathscr{M}} = car_{\mathscr{T}}$. Da $\mathscr{T}' \sqsubseteq \mathscr{T}$ gilt $Y_{fp,\mathscr{M}'} \subseteq X_{tp,\mathscr{M}}$.

□

A.3 Beweis von Lemma 3

Wiederholung von Lemma 3: Gegeben sei eine Kompostion (Sub, Con), eine Menge von Modellen $\{\mathscr{M}_i\}$, eine Menge von Bindungen $\{\mathscr{B}_i\}$ und eine Menge von Schnittstellendefinitionen $\{\mathscr{I}_i = drefI(s_i) | s_i \in Sub\}$. Des Weiteren sei $\mathscr{I}_j \in \{\mathscr{I}_i\}$, $\mathscr{M}_j \in \{\mathscr{M}_i\}$ sowie $\mathscr{B}_j \in \{\mathscr{B}_i\}$.

$$complete(Sub, Con) \wedge \mathscr{I}_j \succ_r (\mathscr{M}_j, \mathscr{B}_j) \wedge \forall \mathscr{I}_i . \mathscr{I}_i \succ_p (\mathscr{M}_i, \mathscr{B}_i) \Rightarrow$$
$$\forall ip \in required(\mathscr{M}_j) . \exists (fm, fp, tm, tp) \in Couplings(Con, Sub). \quad fm = name_{\mathscr{M}_j} \wedge fp = ip$$
$$\vee tm = name_{\mathscr{M}_j} \wedge tp = ip$$

Beweis. Die reflektierende Verfeinerung $\mathscr{I}_j \succ_r (\mathscr{M}_j, \mathscr{B}_j)$ garantiert, dass alle benötigten Modellports $ip \in required(\mathscr{M}_j)$ als zu verbinden deklariert sind, d.h. ein Port $p \in Ports\mathscr{I}$, ein $e \in EP_{\mathscr{R}}$, mit $\mathscr{R} = drefR(rid_p)$, und ein pos_p existiert, so dass $ip = impl(p, e, pos_p, \mathscr{B}_j)$. Durch $complete(Con, Sub)$ ist sichergestellt, das für alle $c \in Con$ zwei Schnittstellenreferenzen $s, s' \in Sub$ existieren, für die c wohlgeformt ist. Damit ist das Ergebnis aller $Coup(c, Sub) \neq \bot$ und es muss ein Port $p' \in Ports_{\mathscr{I}'}$ existieren, dessen Rolle $\mathscr{R}' = drefR(rid_{p'})$ kompatibel zu $\mathscr{R}$ bzw. $\overline{\mathscr{R}}$ ist, so dass durch *matches* jedes e ein Gegenstück $e' \in EP_{\mathscr{R}'}$ zugeordnet bekommt. Aus $\mathscr{I}' \succ_p (\mathscr{M}', \mathscr{B}')$ folgt, dass ein Modellport *peer* existiert dessen Name $peer = impl(p', e', pos_{p'}, \mathscr{B}')$ ist. Da alle Schnittstellen bewahrend verfeinert werden folgt aus Lemma 2, dass MC für wohlgeformte Kompositionsverbindungen ausschließlich wohlgeformte Modellkopplungen produziert. Ob $fp = peer$ oder $tp = peer$ ergibt sich aus der Art der Kompositionsverbindung. □

B XML Schema Definitionen

B.1 Rollen

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified"
 attributeFormDefault="unqualified"
 targetNamespace="http://www.informatik.uni-rostock.de/cosa/role"
 xmlns:xmli="http://www.informatik.uni-rostock.de/cosa/role">

  <xs:element name="role" type="xmli:Role"/>

  <xs:complexType name="Role">
    <xs:sequence>
      <xs:element name="id" type="xs:QName"/>
      <xs:element ref="xmli:types"/>
      <xs:element ref="xmli:description"/>
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"
          processContents="lax"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="types" type="xmli:Types"/>

  <xs:complexType name="Types" mixed="false">
    <xs:sequence>
      <xs:element name="import" type="xmli:Import" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Import">
    <xs:attribute name="namespace" type="xs:anyURI" use="required"/>
    <xs:attribute name="schemaLocation" type="xs:anyURI"/>
  </xs:complexType>

  <xs:element name="description" type="xmli:Description">
    <xs:key name="portKey">
      <xs:selector xpath="xmli:eventPort"/>
      <xs:field xpath="@name"/>
    </xs:key>

    <xs:unique name="dir-type">
      <xs:selector xpath="xmli:eventPort"/>
      <xs:field xpath="@isInput"/>
      <xs:field xpath="xmli:type/@element"/>
    </xs:unique>
  </xs:element>

  <xs:complexType name="Description">
    <xs:sequence>
      <xs:element name="eventPort" type="xmli:PortDeclaration" minOccurs="1"
          maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
```

```
  <xs:complexType name="PortDeclaration">
    <xs:attribute name="element" type="xs:QName" use="required"/>
    <xs:attribute name="name" type="xs:NCName" use="required"/>
    <xs:attribute name="isInput" type="xs:boolean" use="required"/>
  </xs:complexType>

</xs:schema>
```

B.2 Schnittstellen

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.informatik.uni-rostock.de/cosa/publici"
 xmlns="http://www.informatik.uni-rostock.de/cosa/publici"
 elementFormDefault="qualified">

  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      This Schema defines public visible part of components.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="interface" type="Interface"/>

  <xsd:complexType name="Interface">
    <xsd:sequence>
      <!-- Unique identifier -->
      <xsd:element name="id" type="xsd:QName"/>
      <!-- Free form meta data -->
      <xsd:element name="profile" type="Profile" minOccurs="0"/>
      <!-- A set of parameters that can be set for customization -->
      <xsd:element name="param" type="Parameter" minOccurs="0"
                   maxOccurs="unbounded"/>
      <!-- A set of abstract interaction points -->
      <xsd:element name="port" type="Port"
                   minOccurs="0" maxOccurs="unbounded"/>
      <!-- Reference to the implementation -->
      <xsd:element name="impl" type="xsd:QName"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Parameter">
        <xsd:attribute name="name" type="xsd:NCName" use="required"/>
        <xsd:attribute name="type" type="xsd:string" use="required"/>
        <xsd:attribute name="value" type="xsd:string" use="required"/>
    <!-- Help/Documentation for the user. -->
        <xsd:attribute name="description" type="xsd:string"/>
  </xsd:complexType>


  <!-- A port denotes a point of interaction typed by an interface
       definition that is referred to by an unique identifier. A
       required port has to be connected at the time of composition. -->
  <xsd:complexType name="Port">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:NCName"/>
      <xsd:element name="type" type="xsd:QName"/>
    </xsd:sequence>
    <xsd:attribute name="minMultiplicity" type="xsd:unsignedInt"/>
    <xsd:attribute name="maxMultiplicity" type="xsd:nonNegativeInteger"/>
  </xsd:complexType>
```

```
  <xsd:complexType name="Profile">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="application_domain" type="xsd:string"/>
      <xsd:element name="description" type="xsd:string"/>
      <xsd:element name="objective" type="xsd:string"/>
      <xsd:element name="key_abstractions" type="xsd:string"/>
        </xsd:sequence>
  </xsd:complexType>
 </xsd:schema>
```

B.3 Komponenten

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.informatik.uni-rostock.de/cosa/component"
 xmlns="http://www.informatik.uni-rostock.de/cosa/component"
 xmlns:public="http://www.informatik.uni-rostock.de/cosa/publici"
 elementFormDefault="qualified">

  <!-- import parameter type -->
  <xsd:import
   namespace="http://www.informatik.uni-rostock.de/cosa/publici"
   schemaLocation="publici.xsd"/>

  <xsd:element name="component" type="Component"/>

  <xsd:complexType name="Component">
    <xsd:sequence>
      <xsd:element name="id" type="xsd:QName"/>

      <!-- reference to the interface this component implements -->
      <xsd:element name="implements" type="xsd:QName"/>

      <!-- Does configuration according to set parameters -->
      <xsd:element name="mapper" type="xsd:QName" minOccurs="0"/>

      <!-- Reference to a model definition -->
      <xsd:element name="model" type="xsd:QName"/>

      <!-- Bind public visible event ports to ports of the
           implementing model -->
      <xsd:element name="binding" type="Binding"
                   minOccurs="0" maxOccurs="unbounded"/>

      <!-- Set of sub components -->
      <xsd:element name="composition" type="Composition"
                   minOccurs="0" maxOccurs="unbounded"/>

      <!-- Connections between subcomponents (including this component) -->
      <xsd:element name="connection" type="Connection"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Composition">
    <xsd:sequence>
      <xsd:element name="type" type="xsd:QName"/>
      <xsd:element name="name" type="xsd:NCName"/>
      <xsd:element name="parameter" type="public:Parameter"
```

```
                    minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Connection">
    <xsd:sequence>
      <xsd:element name="from" type="Connector"/>
      <xsd:element name="to" type="Connector"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Connector">
    <xsd:attribute name="component" type="xsd:NCName" use="required"/>
    <xsd:attribute name="port" type="xsd:string" use="required"/>
    <xsd:attribute name="position" type="xsd:integer"/>
  </xsd:complexType>

  <!-- Declared ports need to be associated with real model ports -->
  <xsd:complexType name="Binding">
    <xsd:sequence>
      <xsd:element name="bind" type="PortBinding"
                   maxOccurs="unbounded"/>
    </xsd:sequence>
    <!-- name of the component port to be configured -->
    <xsd:attribute name="name" type="xsd:NCName" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="PortBinding">
    <!-- Name of the event port declaration in the interface -->
    <xsd:attribute name="declName" type="xsd:NCName" use="required"/>

    <!-- name of the model port that provides an according type and
         the declaration should be bound to -->
    <xsd:attribute name="implName" type="xsd:NCName" use="required"/>

    <!-- indicates the position within the multiplicity range -->
    <xsd:attribute name="position" type="xsd:integer"/>
  </xsd:complexType>
</xsd:schema>
```

B.4 Simulationsläufe

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.informatik.uni-rostock.de/cosa/experiment"
 xmlns="http://www.informatik.uni-rostock.de/cosa/experiment"
 xmlns:pub="http://www.informatik.uni-rostock.de/cosa/publici"
 elementFormDefault="qualified">

  <!-- import basic types, like ID and Parameter -->
  <xsd:import
   namespace="http://www.informatik.uni-rostock.de/cosa/publici"
   schemaLocation="publici.xsd" />

  <xsd:element name="experiment" type="Experiment"/>

  <!-- A Simulation Experiment -->
  <xsd:complexType name="Experiment">
    <xsd:sequence>
      <xsd:element name="id" type="xsd:QName"/>

      <!-- Reference to the root model (component) -->
```

```
      <xsd:element name="model" type="xsd:QName"/>

      <!-- Parameters the component should be instantiated with -->
      <xsd:element name="mparams" type="ModelParameters" minOccurs="0"/>

      <!-- Namespace of the target simulation formalism -->
      <xsd:element name="targetNs" type="xsd:anyURI"/>

      <!-- responsible for initializing observers for the simulation run -->
      <xsd:element name="observercfg" type="JavaClass" minOccurs="0"/>

      <xsd:element name="repetitions" type="xsd:positiveInteger"/>

      <!-- Simulation parameters -->
      <xsd:any namespace="##other" minOccurs="0" maxOccurs="unbounded"
               processContents="lax"/>
    </xsd:sequence>
  </xsd:complexType>

  <!-- A fully qualified Java class -->
  <xsd:simpleType name="JavaClass">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[a-z][0-9a-zA-Z.]*"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="ModelParameters">
    <xsd:sequence>
      <xsd:element name="param" type="pub:Parameter"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

B.5 Parallel DEVS

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.informatik.uni-rostock.de/cosa/model/pdevs"
 xmlns="http://www.informatik.uni-rostock.de/cosa/model/pdevs"
 elementFormDefault="qualified">

  <!-- Define root element via substitution group, such that it can
       either be an atomic or a coupled model -->
  <xsd:element name="model" abstract="true" type="BasicDEVS"/>
  <xsd:element name="atomic" substitutionGroup="model" type="AtomicPDEVS"/>
  <xsd:element name="coupled" substitutionGroup="model" type="CoupledPDEVS"/>

  <xsd:complexType name="Port">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="type" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:complexType name="Coupling">
    <xsd:attribute name="fromModel" type="xsd:NCName" use="required"/>
    <xsd:attribute name="fromPort" type="xsd:NCName" use="required"/>
    <xsd:attribute name="toModel" type="xsd:NCName" use="required"/>
    <xsd:attribute name="toPort" type="xsd:NCName" use="required"/>
  </xsd:complexType>
```

```
<!-- Base of all DEVS models. Declares input and output ports. -->
<xsd:complexType name="BasicDEVS" abstract="true">
  <xsd:sequence>
    <xsd:element name="inport" type="Port"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="outport" type="Port"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<!-- Base of all atomic models -->
<xsd:complexType name="BasicAtomicDEVS" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="BasicDEVS">
      <xsd:sequence>
        <xsd:element name="state" type="State" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- Base of all coupled models. -->
<xsd:complexType name="BasicCoupledDEVS" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="BasicDEVS">
      <xsd:sequence>
        <xsd:element name="eic" type="Coupling"
                     minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="ic" type="Coupling"
                     minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="eoc" type="Coupling"
                     minOccurs="0" maxOccurs="unbounded"/>

        <xsd:element name="model" type="SubModel"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- Atomic PDEVS Model -->
<xsd:complexType name="AtomicPDEVS">
  <xsd:complexContent>
    <xsd:extension base="BasicAtomicDEVS">
      <xsd:sequence>
        <!-- The class is supposed to extend
             james.core.model.devs.parallel.AtomicModel -->
        <xsd:element name="impl" type="JavaClass"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- Coupled PDEVS Model -->
<xsd:complexType name="CoupledPDEVS">
  <xsd:complexContent>
    <xsd:extension base="BasicCoupledDEVS">
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="State">
  <xsd:sequence>
```

```
      <!-- A state is structured by a set of substates -->
      <xsd:element name="substate" type="SubState"  maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="SubState">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="type" type="xsd:string" use="required"/>
        <xsd:attribute name="init" type="xsd:string" use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <!-- A named reference to a submodel -->
  <xsd:complexType name="SubModel">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:NCName"/>
      <xsd:element name="model" type="BasicDEVS"/>
    </xsd:sequence>
  </xsd:complexType>

  <!-- A fully qualified Java class -->
  <xsd:simpleType name="JavaClass">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[a-z][0-9a-zA-Z.]*"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Literaturverzeichnis

[Aceto u. a. 2005] Aceto, Luca ; Fokkink, Wan ; Ingolfsdottir, Anna ; Luttik, Bas: Finite Equational Bases in Process Algebra: Results and Open Questions. In: *Klop Festschrift*. 2005

[Agarwal u. a. 2005] Agarwal, Vikas ; Dasgupta, Koustuv ; Karnik, Neeran ; Kumar, Arun ; Kundu, Ashish ; Mittal, Sumit ; Srivastava, Biplav: A service creation environment based on end to end composition of Web services. In: *WWW '05: Proceedings of the 14th international conference on World Wide Web*. New York, NY, USA : ACM Press, 2005. – ISBN 1–59593–046–9, S. 128–137

[Agha u. a. 1997] Agha, Gul A. ; Mason, Ian A. ; Smith, Scott F. ; Talcott, Carolyn L.: A foundation for actor computation. In: *J. Funct. Program.* 7 (1997), Nr. 1, S. 1–72. `http://dx.doi.org/http://dx.doi.org/10.1017/S095679689700261X`. – DOI http://dx.doi.org/10.1017/S095679689700261X. – ISSN 0956–7968

[Alesso 2004] Alesso, Peter: *Preparing for Semantic Web Services*. `http://www.sitepoint.com/article/semantic-web-services` [zugegriffen am 18. Januar 2008], Mai 2004

[de Alfaro u. Henzinger 2001] Alfaro, Luca de ; Henzinger, Thomas A.: Interface automata. In: *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, ACM Press, 2001, S. 109–120

[de Alfaro u. Henzinger 2005] Alfaro, Luca de ; Henzinger, Thomas A.: Interface-based Design. In: *Engineering Theories of Software-intensive Systems* Bd. 195 Springer, M. Broy, J. Gruenbauer, D. Harel, and C.A.R. Hoare, 2005 (NATO Science Series: Mathematics, Physics, and Chemistry), S. 83–104

[Andel u. Yasinac 2006] Andel, Todd R. ; Yasinac, Alec: On the Credibility of Manet Simulations. In: *Computer* 39 (2006), Nr. 7, S. 48–54. `http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/MC.2006.242`. – DOI http://doi.ieeecomputersociety.org/10.1109/MC.2006.242. – ISSN 0018–9162

[Anft u. Friese 2003] Anft, Stephan ; Friese, Peter: Der Prokurist: Praktische Anwendungsszenarien des Dynamic Proxy API. In: *Javamagazin* (2003), Oktober, S. 15–20

[Aumueller u. a. 2005] Aumueller, David ; Do, Hong-Hai ; Massmann, Sabine ; Rahm, Erhard: Schema and ontology matching with COMA++. In: *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. New York, NY, USA : ACM Press, 2005. – ISBN 1–59593–060–4, S. 906–908

[Baeten 2005] Baeten, J. C. M.: A brief history of process algebra. In: *Theor. Comput. Sci.* 335 (2005), Nr. 2-3, S. 131–146. `http://dx.doi.org/http://dx.doi.org/10.1016/j.tcs.2004.07.036`. – DOI http://dx.doi.org/10.1016/j.tcs.2004.07.036. – ISSN 0304–3975

[Balci 1997] Balci, Osman: Verification validation and accreditation of simulation models. In: *WSC '97: Proceedings of the 29th conference on Winter simulation*. Washington, DC, USA : IEEE Computer Society, 1997. – ISBN 0–7803–4278–X, S. 135–141

[Balci u. a. 2002] Balci, Osman ; Nance, Richard E. ; Arthur, James D. ; Ormsby, William F.: Improving the model development process: expanding our horizons in verification, validation, and accreditation research and practice. In: *WSC '02: Proceedings of the 34th conference on Winter simulation*, Winter Simulation Conference, 2002. – ISBN 0–7803–7615–3, S. 653–663

[Bartholet u. a. 2004] Bartholet, Robert G. ; Brogan, David C. ; Reynolds, Paul F. ; Carnahan, Joseph C.: In Search of the Philosopher's Stone: Simulation Composability Versus Component-Based Software Design. In: *Proceedings of the 2004 Spring Simulation Interoperability Workshop*, 2004

[Becker u. a. 2005] Becker, Joerg ; Niehaves, Bjoern ; Klose, Karsten: A Framework for Epistemological Perspectives on Simulation. In: *Journal of Artificial Societies and Social Simulation* 8 (2005), Nr. 4, S. 1

[ter Beek u. a. 2006] Beek, Maurice ter ; Bucchiarone, Antonio ; Gnesi, Stefania: A Survey on Service Composition Approaches: From Industrial Standards to Formal Methods / Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche. 2006 (2006-TR-15). – Forschungsbericht

[Bergmans u. a. 2003] Bergmans, Lodewijk ; Tekinerdogan, Bedir ; Nagy, Istvan ; Aksit, Mehmet: An Analysis of Composability and Composition Anomalies / University of Twente. 2003. – Forschungsbericht. – `http://purl.org/utwente/55812`

[Bernauer u. a. 2004] Bernauer, Martin ; Kappel, Gerti ; Kramler, Gerhard: Representing XML Schema in UML – A Comparison of Approaches. In: *Web Engineering*. Springer, 2004, S. 440–444

[Berners-Lee u. a. 2001] Berners-Lee, Tim ; Hendler, James ; Lassila, Ora: The Semantic Web. In: *Scientific American* (2001), Mai, S. 34–43

[Beugnard u. a. 1999] Beugnard, Antoine ; Jézéquel, Jean-Marc ; Plouzeau, Noël ; Watkins, Damien: Making Components Contract Aware. In: *Computer* 32 (1999), Nr. 7, S. 38–45. `http://dx.doi.org/http://dx.doi.org/10.1109/2.774917`. – DOI http://dx.doi.org/10.1109/2.774917. – ISSN 0018–9162

[Blais u. a. 2005] Blais, Curtis ; Brutzman, Don ; Drake, David ; Moen, Dennis ; Morse, Katherine ; Pullen, Mark ; Tolk, Andreas: Extensible Modeling and Simulation Framework (XMSF) 2004 Project Summary Report / Naval Postgraduate School. Monterey, California, Februar 2005. – Forschungsbericht

[Boer u. a. 2006a] Boer, Csaba A. ; Bruin, Arie de ; Verbraeck, Alexander: Distributed simulation in industry – a survey: part 1 – the COTS vendors. In: *WSC '06: Proceedings of the 38th conference on Winter simulation*, Winter Simulation Conference, 2006. – ISBN 1–4244–0501–7, S. 1053–1060

[Boer u. a. 2006b] Boer, Csaba A. ; Bruin, Arie de ; Verbraeck, Alexander: Distributed simulation in industry – a survey: part 2 – experts on distributed simulation. In: *WSC '06: Proceedings of the 38th conference on Winter simulation*, Winter Simulation Conference, 2006. – ISBN 1–4244–0501–7, S. 1061–1068

[Borchardt 2008] Borchardt, Ulrike: *Evaluation of trust building from contextual information in MANETs*, Universität Rostock, Diplomarbeit, 2008. – to appear

[Bordeaux u. Salaün 2005] Bordeaux, Lucas ; Salaün, Gwen: Using Process Algebra for Web Services: Early Results and Perspektives. In: *Technologies for E-Services (TES 2004)* Bd. 3324. Springer-Verlag, 2005, S. 54–68

[Bordeaux u. a. 2005] Bordeaux, Lucas ; Salaün, Gwen ; Berardi, Daniela ; Mecella, Massimo: When are Two Web Services Compatible? In: *Technologies for E-Services (TES 2004)* Bd. 3324. Springer-Verlag, 2005, S. 15–28

[Borland u. Vangheluwe 2003] Borland, Spencer ; Vangheluwe, Hans: Transforming Statecharts to DEVS. In: Bruzzone, A. (Hrsg.) ; Itmi, Mhamed (Hrsg.): *Summer Computer Simulation Conference. Student Workshop.* Montréal, Canada : Society for Computer Simulation International (SCS), Juli 2003, S. 154–159

[Bourret 2005] Bourret, Ronald: *XML Data Binding Resources.* `http://www.rpbourret.com/xml/XMLDataBinding.htm`, [zugegriffen am 11. Juli 2005], April 2005

[Brodkin 2001] Brodkin, Sam: *Use XML data binding to do your laundry: Explore JAXB and Castor from the ground up.* `http://www.javaworld.com/javaworld/jw-12-2001/jw-1228-jaxb.html`, [zugegriffen am 11. Juli 2005], December 2001

[Brogi u. a. 2004] Brogi, Antonio ; Canal, Carlos ; Pimentel, Ernesto ; Vallecillo, Antonio: Formalizing Web Service Choreographies. In: *Electronic Notes in Theoretical Computer Science* 105 (2004), S. 73–94

[Brooks u. a. 2007] Brooks, Christopher ; Lee, Edward A. ; Liu, Xiaojun ; Neuendorffer, Stephen ; Zhao, Yang ; Zheng, Haiyang: Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II) / EECS Department, University of California, Berkeley. Version: Jan 2007. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-7.html`. 2007 (UCB/EECS-2007-7). – Forschungsbericht

[Broy u. a. 2006] Broy, Manfred ; Crane, Michelle L. ; Dingel, Juergen ; Hartman, Alan ; Rumpe, Bernhard ; Selic, Bran: 2nd UML 2 Semantics Symposium: Formal Semantics for UML. In: Kühne, T. (Hrsg.): *MoDELS 2006 Workshops* Bd. 4364, Springer, 2006 (LNCS), S. 318–323

[Cámara u. a. 2005] Cámara, Javier ; Canal, Carlos ; Cubo, Javier: Issues in the formalization of Web Service Orchestrations. In: *Second International Workshop on Coordination and Application Techniques for Software Entitiees (WCAT'05)*, 2005

[Cámara u. a. 2006] Cámara, Javier ; Canal, Carlos ; Cubo, Javier ; Vallecillo, Antonio: Formalizing WSBPEL Business Processes Using Process Algebra. In: *Electr. Notes Theor. Comput. Sci.* 154 (2006), Nr. 1, S. 159–173

[Canal u. a. 2003] Canal, Carlos ; Fuentes, Lidia ; Pimentel, Ernesto ; Troya, José M. ; Vallecillo, Antonio: Adding Roles to CORBA Objects. In: *IEEE Trans. Softw. Eng.* 29 (2003), Nr. 3, S. 242–260. `http://dx.doi.org/http://dx.doi.org/10.1109/TSE.2003.1183935`. – DOI http://dx.doi.org/10.1109/TSE.2003.1183935. – ISSN 0098–5589

[Canal u. a. 2001] Canal, Carlos ; Pimentel, Ernesto ; Troya, José M.: Compatibility and inheritance in software architectures. In: *Sci. Comput. Program.* 41 (2001), Nr. 2, S. 105–138. `http://dx.doi.org/http://dx.doi.org/10.1016/S0167-6423(01)00002-8`. – DOI http://dx.doi.org/10.1016/S0167–6423(01)00002–8. – ISSN 0167–6423

[Cavilla u. a. 2004] Cavilla, Andr'es L. ; Baron, Gerard ; Hart, Thomas E. ; Litty, Lionel ; Lara, Eyal de: Simplified Simulation Models for Indoor MANET Evaluation Are Not Robust. In: *Proc. SECON'04*, IEEE, 2004, S. 610–620

[Chen u. Szymanski 2002] Chen, Gilbert ; Szymanski, Boleslaw K.: Cost: A Component-Oriented Discrete Event Simulator. In: *Proceedings of the 2002 Winter Simulation Conference*, 2002, 776–782

[Cholkar u. Koopman 1999] Cholkar, Arjun ; Koopman, Philip: A widely deployable Web-based network simulation framework using CORBA IDL-based APIs. In: *WSC '99: Proceedings of the 31st conference on Winter simulation*. New York, NY, USA : ACM Press, 1999. – ISBN 0–7803–5780–9, S. 1587–1594

[Chow u. Zeigler 1994] Chow, Alex Chung H. ; Zeigler, Bernard P.: Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In: *WSC '94: Proceedings of the 26th conference on Winter simulation*. San Diego, CA, USA : Society for Computer Simulation International, 1994. – ISBN 0–7803–2109–X, S. 716–722

[Corson u. Macker 1999] Corson, S. ; Macker, J.: *Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations*. `http://www.ietf.org/rfc/rfc2501.txt`, Januar 1999. – Network Working Group Memo (Request for Comments: 2501)

[Dalle u. Mrabet 2007] Dalle, Olivier ; Mrabet, Cyrine: An Instrumentation Framework for component-based simulations based on the Separation of Concerns paradigm. In: *Proc. of 6th EUROSIM Congress (EUROSIM'2007)*. Ljubljana, Slovenia, September 9-13 2007

[Daskalova u. Atanasova 2005] Daskalova, Hristina ; Atanasova, Tatiana: Semantic Web Services – Where We are and Where We are Going. In: *Proceedings of the International Conference "Generic issues for knowledge technology"*. Budapest, Hungary, 2005

[Davis u. Anderson 2004] Davis, Paul K. ; Anderson, Robert H.: Improving the Composability of DoD Models and Simulations. In: *JDMS* 1 (2004), April, Nr. 1, S. 5–17

[Davis u. Tolk 2007] Davis, Paul K. ; Tolk, Andreas: Observations on new developments in composability and multi-resolution modeling. In: *Proceedings of the 2007 Winter Simulation Conference*, 2007, S. 859–870

[DEVS 2007] *Proceedings of the 2007 DEVS Integrative M& S Symposium (DEVS'07, part of SpringSim '07)*. Norfolk, Virginia, USA : SCS, März 2007 . – General Chair: James Nutaro, Program Chair: Xiaolin Hu

[Diane 2007] Diane: *DIANE-Projekt: Services in ad hoc Networks (Dienste in Ad-Hoc-Netzen)*. `http://www.ipd.uni-karlsruhe.de/DIANE` (zugegriffen am 19. September 2007), 2007

[Do u. a. 2003] Do, Hong H. ; Melnik, Sergey ; Rahm, Erhard: Comparison of Schema Matching Evaluations. In: *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*. London, UK : Springer-Verlag, 2003. – ISBN 3–540–00745–8, S. 221–237

[Do u. Rahm 2002] Do, Hong H. ; Rahm, Erhard: COMA - A System for Flexible Combination of Schema Matching Approaches. In: *VLDB 2002*, 2002, S. 610–621

[DoD 2003] DoD: *DoD Modeling and Simulation (M& S) Verification, Validation, and Accreditation (VV& A)*. Instruction 5000.61, 13 May 2003

[Eker u. a. 2003] Eker, Johan ; Janneck, Jörn W. ; Lee, Edward A. ; Liu, Jie ; Liu, Xiaojun ; Ludvig, Josef ; Neuendorffer, Stephen ; Sachs, Sonia ; Xiong, Yuhong: Taming heterogeneity — The Ptolemy approach. In: *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software* 91 (2003), January, Nr. 1, S. 127–144

[Elmqvist u. a. 2001] Elmqvist, Hilding ; Mattsson, Sven E. ; Otter, Martin: Object-Oriented and Hybrid Modeling in Modelica. In: *Journal Européen des systèmes automatisés* 35 (2001), Nr. 1, S. 1–10

[Erl 2005] Erl, Thomas: *Service-Oriented Architecture. Concepts, Technology, and Design.* Prentice Hall PTR, 2005

[Ewald u. a. 2006] Ewald, Roland ; Himmelspach, Jan ; Uhrmacher, Adelinde M.: A Non-Fragmenting Partitioning Algorithm for Hierarchical Models. In: *Proc. of the 2006 Winter Simulation Conference*, 2006, S. 848–855

[Favre 2004] Favre, Jean-Marie: Towards a basic theory to model, model driven engineering. In: *Workshop on Software Model Engineering (WISME).* Lisboa, Portugal, 2004

[Favre 2005] Favre, Jean-Marie: Foundations of Meta-Pyramids: Languages vs. Metamodels – Episode II: Story of Thotus the Baboon1. In: Bezivin, Jean (Hrsg.) ; Heckel, Reiko (Hrsg.): *Language Engineering for Model-Driven Software Development*, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005 (Dagstuhl Seminar Proceedings 04101). – ISSN 1862–4405. – <http://drops.dagstuhl.de/opus/volltexte/2005/21> [date of citation: 2005-01-01]

[Favre u. Nguyen 2005] Favre, Jean-Marie ; Nguyen, Tam: Towards a Megamodel to Model Software Evolution Through Transformations. In: *Electr. Notes Theor. Comput. Sci.* 127 (2005), Nr. 3, S. 59–74

[Fishwick u. Miller 2004] Fishwick, Paul A. ; Miller, John A.: Ontologies for modeling and simulation: issues and approaches. In: *WSC '04: Proceedings of the 36th conference on Winter simulation*, Winter Simulation Conference, 2004. – ISBN 0–7803–8786–4, S. 259–264

[Fröberg 2002] Fröberg, Joakim: Software Components and COTS in Software System Development. In: Crnkovic, I. (Hrsg.) ; Larsson, M. (Hrsg.): *Building Reliable Component-Based Systems.* Artech House, Juli 2002, S. 59–67

[Gamma u. a. 1995] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995

[van Glabbeek 2005] Glabbeek, Rob J.: A Characterisation of Weak Bisimulation Congruence. In: *Klop Festschrift* Bd. 3838. 2005, S. 26–39

[Gustavson u. Chase 2004] Gustavson, Paul ; Chase, Tram: Using XML and BOMs to rapidly compose simulations and simulation environments. In: *Proceedings of the 2004 Winter Simulation Conference*, 2004, S. 1467–1475

[van der Ham u. a. 2006] Ham, Jeroen van d. ; Dijkstra, Freek ; Travostino, Franco ; Andree, Hubertus ; Laat, Cees de: Using RDF to Describe Networks. In: *Future Generation Computer Systems, Feature topic iGrid 2005* (2006). `http://staff.science.uva.nl/~vdham/research/publications/0510-NetworkDescriptionLanguage.pdf`

[Hamadi u. Benatallah 2003] Hamadi, Rachid ; Benatallah, Boualem: A Petri net-based model for web service composition. In: *ADC '03: Proceedings of the 14th Australasian database conference.* Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2003. – ISBN 0–909–92595–X, S. 191–200

[Hameurlain 2005] Hameurlain, Nabil: On Compatibility and Behavioural Substitutability of Component Protocols. In: *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05.* Los Alamitos, CA, USA : IEEE Computer Society, 2005, S. 394–403

[Harel 1987] Harel, David: Statecharts: A visual formalism for complex systems. In: *Sci. Comput. Program.* 8 (1987), Nr. 3, S. 231–274. `http://dx.doi.org/http://dx.doi.org/10.1016/0167-6423(87)90035-9`. – DOI http://dx.doi.org/10.1016/0167–6423(87)90035–9. – ISSN 0167–6423

[Harel u. Rumpe 2004] Harel, David ; Rumpe, Bernhard: Meaningful Modeling: What's the Semantics of Semantics? In: *Computer* 37 (2004), Nr. 10, S. 64–72

[Harold 2002] Harold, Elliotte R.: *Processing XML with Java.* Pearson Education, 2002

[Heisel u. a. 2004] Heisel, M ; Uhrmacher, A M. ; Luethi, J ; Valentin, E: A Description Structure for Simulation Model Components. In: *Proc. of the Summer Computer Simulation*, 2004

[Hewitt 1977] Hewitt, Carl: Viewing Control Structures as Patterns of Passing Messages. In: *Artif. Intell.* 8 (1977), Nr. 3, S. 323–364

[Himmelspach u. Röhl 2008] Himmelspach, Jan ; Röhl, Mathias: JAMES II – Experiences and Interpretations. In: Adelinde M. Uhrmacher, Danny W. (Hrsg.): *Agents, Simulation and Applications.* Taylor and Francis, 2008, Kapitel 20. – to appear

[Himmelspach u. Uhrmacher 2004] Himmelspach, Jan ; Uhrmacher, Adelinde M.: A Component-based Simulation Layer for James. In: *PADS '04: Proceedings of the eighteenth workshop on Parallel and distributed simulation.* Kufstein, Austria : IEEE Computer Society Press, 2004, S. 115–122

[Himmelspach u. Uhrmacher 2007] Himmelspach, Jan ; Uhrmacher, Adelinde M.: Plug'n simulate. In: *Proceedings of the 40th Annual Simulation Symposium*, IEEE Computer Society, 2007, 137–143

[Hirschmeier 2006] Hirschmeier, Sebastian: *Modellierung sozialer MANET-Nutzer.* Studienarbeit, Universität Rostock, November 2006

[Hofmann 2004] Hofmann, Marko A.: Challenges of Model Interoperation in Military Simulations. In: *Simulation* 80 (2004), Dezember, Nr. 12, S. 659–667

[Hogg 2004] Hogg, John: UML 2.0 Automates Code Generation from Architecture. In: *COTS Journal* (2004), Mai

[Hucka u. a. 2004] Hucka, M. ; Finney, A. ; Bornstein, B.J. ; Keating, S.M. ; Shapiro, B.E. ; Matthews, J. ; Kovitz, B.L. ; Schilstra, M.J. ; Funahashi, A. ; Doyle, J.C. ; Kitano, H.: Evolving a Lingua Franca and Associated Software Infrastructure for Computational Systems Biology: The Systems Biology Markup Language (SBML) Project. In: *Systems Biology* 1 (2004), Juni, Nr. 1, S. 41–53

[IEEE 2000] IEEE: *Standard for Modeling and Simulation (M& S) High Level Architecture (HLA) — Framework and Rules.* Document 1516-2000, September 2000

[IEEE 2003] IEEE: *Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP).* Document 1516.3, März 2003

[Janoušek u. a. 2006] Janoušek, Vladimír ; Polášek, Petr ; Slavíček, Pavel: Towards DEVS Meta Language. In: *ISC 2006 Proceedings*, 2006. – ISBN 90–77381–26–0, 69–73

[Janssen 1997] Janssen, Theo M. V.: Compositionality (with an appendix by B. Partee). In: Benthem, J. van (Hrsg.) ; Meulen, A. ter (Hrsg.): *Handbook of Logic and Language.* Amsterdam : Elsevier, 1997, S. 417–473

[Kasputis u. Ng 2000] Kasputis, Stephen ; Ng, Henry C.: Composable Simulations. In: *Proceedings of the 2000 Winter Simulation Conference*, 2000, S. 1577–1584

[Kelton u. a. 1998] Kelton, W. D. ; Sadowski, Randall P. ; Sadowski, Deborah A.: *Simulation with Arena*. McGraw-Hill, 1998

[Kim u. Kang 2004] Kim, Ki-Hyung ; Kang, Won-Seok: CORBA-Based, Multi-threded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One. In: *ICCSA 2004* Bd. 3046. Berlin Heidelberg : Springer-Verlag, 2004, S. 167–176

[Kim u. Kang 2005] Kim, Ki-Hyung ; Kang, Won-Seok: A Web Service-Based Distributed Simulation Architecture for Hierarchical DEVS Models. In: *AIS 2004* Bd. 3397. Berlin Heidelberg : Springer-Verlag, 2005, S. 370–379

[Klein 2003] Klein, Michael: DIANEmu – A Java-Based Generic Simulation Environment for Distributed Protocols / Universität Karlsruhe, Faculty of Informatics. 2003 (2003/7). – Forschungsbericht

[Klein u. a. 2004] Klein, Michael ; Hoffman, Markus ; Matheis, Daniel ; Müssig, Michael: Comparison of Overlay Mechanisms for Service Trading in Ad hoc Networks / University of Karlsruhe. 2004 (TR 2004-2). – Forschungsbericht. – ISSN 1432-7864

[Klein u. a. 2003] Klein, Michael ; König-Ries, Birgitta ; Obreiter, Philipp: Lanes – A Lightweight Overlay for Service Discovery in Mobile Ad Hoc Network. In: *3rd Workshop on Applications and Services in Wireless Networks (ASWN2003). Berne, Swiss*, 2003

[König-Ries u. a. 2006] König-Ries, Birgitta ; Klein, Michael ; Breyer, Tobias: Activity-based user modeling in wireless networks. In: *Mob. Netw. Appl.* 11 (2006), Nr. 2, S. 267–277. `http://dx.doi.org/http://dx.doi.org/10.1007/s11036-006-4478-4`. – DOI http://dx.doi.org/10.1007/s11036–006–4478–4. – ISSN 1383–469X

[Kurkowski u. a. 2005] Kurkowski, S. ; Camp, T. ; Colagrosso, M.: MANET simulation studies: The Incredibles. In: *ACM's Mobile Computing and Communications Review* 9 (2005), Nr. 4, S. 50–61

[Küster u. König-Ries 2006] Küster, Ulrich ; König-Ries, Birgitta: Dynamic Binding for BPEL Processes - A Lightweight Approach to Integrate Semantics into Web Services. In: *Second International Workshop on Engineering Service-Oriented Applications: Design and Composition (WESOA06) at 4th International Conference on Service Oriented Computing (ICSOC06)*. Chicago, Illinois, USA, December 2006

[Küster u. a. 2007] Küster, Ulrich ; König-Ries, Birgitta ; Stern, Mirco ; Klein, Michael: DIANE: an integrated approach to automated service discovery, matchmaking and composition. In: *WWW '07: Proceedings of the 16th international conference on World Wide Web*. New York, NY, USA : ACM Press, 2007. – ISBN 978–1–59593–654–7, S. 1033–1042

[Larsson 2006] Larsson, Jonas: A Framework for Implementation-Independent Simulation Models. In: *Simulation* 82 (2006), September, Nr. 9, S. 563–579

[Lau u. Wang 2006] Lau, Kung-Kiu ; Wang, Zheng: *A Survey of Software Component Models (second edition)*. Preprint CSPP-38, School of Computer Science, The University of Manchester, Mai 2006

[Law u. Kelton 2000] Law, A.M. ; Kelton, W.D.: *Simulation, Modeling, and Analysis*. 3rd. New York : MCGraw Hill International Editions, 2000

[Le Novère u. a. 2005] Le Novère, N. ; Finney, A. ; Hucka, M. ; Bhalla, U. S. ; Campagne, F. ; Vides, Collado J. ; Crampin, E. J. ; Halstead, M. ; Klipp, E. ; Mendes, P. ; Nielsen, P. ; Sauro, H. ; Shapiro, B. ; Snoep, J. L. ; Spence, H. D. ; Wanner, Barry L.: Minimum

information requested in the annotation of biochemical models (MIRIAM). In: *Nat Biotechnol* 23 (2005), Dezember, Nr. 12, S. 1509–1515

[Lee u. Neuendorffer 2004a] Lee, Edward A. ; Neuendorffer, Stephen: Actor-oriented models for codesign: balancing re-use and performance. (2004), S. 33–56. ISBN 1–4020–8051–4

[Lee u. Neuendorffer 2004b] Lee, Edward A. ; Neuendorffer, Stephen: Classes and subclasses in actor-oriented design. In: *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*, IEEE, 2004, S. 161–168

[Lee u. Xiong 2004] Lee, Edward A. ; Xiong, Yuhong: A behavioral type system and its application in Ptolemy II. In: *Form. Asp. Comput.* 16 (2004), Nr. 3, S. 210–237. http://dx.doi.org/http://dx.doi.org/10.1007/s00165-004-0043-8. – DOI http://dx.doi.org/10.1007/s00165–004–0043–8. – ISSN 0934–5043

[Liu u. a. 2005] Liu, Fangfang ; Zhang, Liang ; Shi, Yuliang ; Lin, Lili ; Shi, Baile: Formal Analysis of Compatibility of Web Services via CCS. In: *International Conference on Next Generation Web Services Practices (NWeSP'05)*. Los Alamitos, CA, USA : IEEE Computer Society, 2005, S. 143–148

[MacSween u. Wainer 2004] MacSween, Peter ; Wainer, Gabriel: On the Construction of Complex Models Using Reusable Components. In: *Proceedings of SISO Spring Interoperability Workshop*. Arlington, VA,. USA, 2004

[Martín u. a. 2007] Martín, Jose Luis R. ; Mittal, Saurabh ; na, Miguel Angel López P. ; Cruz, Jesus M. l.: A W3C XML Schema for DEVS Scenarios. In: *SpringSim '07, DEVS Integrative M&S Symposium DEVS' 07*, 2007, S. 279–286

[Martin 2003] Martin, Robert C.: *Agile Software Development: Principes, Patterns, and Practices*. Prentice Hall, 2003

[McLaughlin 2002] McLaughlin, Brett: *Why Data Binding Matters*. http://www.onjava.com/pub/a/onjava/2002/05/15/databind.html, [zugegriffen am 11. Juli 2005], May 2002

[Medjahed u. Bouguettaya 2005] Medjahed, Brahim ; Bouguettaya, Senior A.: A Multilevel Composability Model for Semantic Web Services. In: *IEEE Transactions on Knowledge and Data Engineering* 17 (2005), Nr. 7, S. 954–968. http://dx.doi.org/http://dx.doi.org/10.1109/TKDE.2005.101. – DOI http://dx.doi.org/10.1109/TKDE.2005.101. – ISSN 1041–4347

[Medvidovic u. Taylor 2000] Medvidovic, Nenad ; Taylor, Richard N.: A Classification and Comparison Framework for Software Architecture Description Languages. In: *IEEE Transactions on Software Engineering* 26 (2000), Nr. 1, S. 70–93. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/32.825767. – DOI http://doi.ieeecomputersociety.org/10.1109/32.825767. – ISSN 0098–5589

[Megginson 2008] Megginson, David: *Simple API for XML*. http://www.megginson.com/downloads/SAX/, zugegriffen am 23. Januar 2008, 2008

[Mellor u. a. 2004] Mellor, Stephen J. ; Scott, Kendall ; Uhl, Axel ; Weise, Dirk: *MDA Distilled*. Addison-Wesley, 2004

[Miller u. Baramidze 2005] Miller, John A. ; Baramidze, Gregory: Simulation and the semantic web. In: *WSC '05: Proceedings of the 37th conference on Winter simulation*, Winter Simulation Conference, 2005. – ISBN 0–7803–9519–0, S. 2371–2377

[Milner 1990] Milner, Robin: Operational and Algebraic Semantics of Concurrent Processes. In: Leeuwen, J. van (Hrsg.): *Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics*. Amsterdam : Elsevier, 1990, S. 1201–1242

[Milner 1999] Milner, Robin: *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, 1999

[Minsky 1965] Minsky, Marvin: Matter, mind and models. In: *Proc. International Federation of Information Processing Congress 1965* Bd. 1. Washington, D.C. : Spartan Books, 1965, S. 45–49. – nach `http://groups.csail.mit.edu/medg/people/doyle/gallery/minsky/mmm.html` von 1995

[Mittal u. a. 2007] Mittal, Saurabh ; Luis, José ; Martín, Risco: DEVSML: Automating DEVS Execution over SOA Towards Transparent Simulators. In: *SpringSim '07, DEVS Integrative M&S Symposium DEVS' 07*, 2007, S. 287–295

[Modelica 2006] *Proceedings of the 5th International Modelica Conference*. Vienna, Austria : The Modelica Association and arsenal research, September 2006 . – Conference Chair: Dr. Christian Kral, Program Chair: Anton Haumer

[Möller u. Dahlin 2006] Möller, Björn ; Dahlin, Clarence: A First Look at the HLA Evolved Web Service API. In: *Proceedings of 2006 Euro Simulation Interoperability Workshop* Simulation Interoperability Standards Organization, 2006. – 06E-SIW-061

[Möller u. a. 2007a] Möller, Björn ; Dahlin, Clarence ; Karlsson, Mikael: Developing Web Centric Federates and Federations using the HLA Evolved Web Services API. In: *Proceedings of 2007 Spring Simulation Interoperability Workshop*, Simulation Interoperability Standards Organization, März 2007. – 07S-SIW-107

[Möller u. a. 2007b] Möller, Björn ; Gustavson, Paul ; Lutz, Bob ; Löfstrand, Björn: Making your BOMs and FOM modules play together. In: *Proc. of 2007 Fall Simulation Interoperability Workshop*, 2007. – SISO paper no. 07F-SIW-069

[Möller u. Löfstrand 2007] Möller, Björn ; Löfstrand, Björn: Use Cases for the HLA Evolved Modular FOMs. In: *Proceedings of 2007 Euro Simulation Interoperability Workshop*, Simulation Interoperability Standards Organization, Juni 2007. – 07E-SIW-040

[Möller u. a. 2007c] Möller, Björn ; Löfstrand, Björn ; Karlsson, Mikael: An Overview of the HLA Evolved Modular FOMs. In: *Proceedings of 2007 Spring Simulation Interoperability Workshop*, Simulation Interoperability Standards Organization, März 2007. – 07S-SIW-108

[Moller u. a. 2004] Moller, Faron ; Smolka, Scott ; Srba, Jiři: On the computational complexity of bisimulation, redux. In: *Inf. Comput.* 194 (2004), Nr. 2, S. 129–143. `http://dx.doi.org/http://dx.doi.org/10.1016/j.ic.2004.06.003`. – DOI http://dx.doi.org/10.1016/j.ic.2004.06.003. – ISSN 0890–5401

[Moradi u. a. 2006] Moradi, Farshad ; Nordvaller, Peder ; Ayani, Rassul: Simulation Model Composition using BOMs. In: *Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'06)*. Los Alamitos, CA, USA : IEEE Computer Society, 2006, S. 242–252

[Morgenstern 2007] Morgenstern, Stefan: *Plattformunabhängige Schnittstellenbeschreibung von Modellkomponenten*, Universität Rostock, Diplomarbeit, März 2007

[Morse u. a. 2006] Morse, Katherine L. ; Lightner, Mike ; Little, Reed ; Lutz, Bob ; Scrudder, Roy: Enabling Simulation Interoperability. In: *Computer* 39 (2006), Nr. 1, S. 115–117. – ISSN 0018–9162

[Muguira u. Tolk 2006] Muguira, James ; Tolk, Andreas: Applying a Methodology to identify Structural Variances in Interoperations. In: *Journal for Defense Modeling and Simulation* 3 (2006), April, Nr. 2, S. 77–93

[Neubauer u. a. 2004] Neubauer, Bertram ; Ritter, Tom ; Stoinski, Frank: *CORBA Komponenten*. Springer-Verlag, 2004

[OASIS 2007] OASIS: *Web Services Business Process Execution Language Version 2.0*. `http://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm`, zugegriffen am 22. Januar 2008, Mai 2007

[Oestereich 2006] Oestereich, Bernd: *Analyse und Design mit UML 2.1*. 8. Oldenbourg Verlag, 2006

[O'Keefe 2006] O'Keefe, Greg: Improving the Definition of UML. In: Nierstrasz, Oscar (Hrsg.) ; Whittle, Jon (Hrsg.) ; Harel, David (Hrsg.) ; Reggio, Gianna (Hrsg.): *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006* Bd. 4199, Springer, 2006 (Lecture Notes in Computer Science), S. 42–56

[Oliver u. Luukkala 2006] Oliver, Ian ; Luukkala, Vesa: On UML's Composite Structure Diagram. In: *SAM'06*. Kaiserslautern, Germany, May 31 – June 2 2006

[OMG 2002] OMG: *CORBA Components Version 3.0 (Document formal/02-06-65)*. `http://www.omg.org/cgi-bin/doc?formal/02-06-65`, [zugegriffen am 13. Juli 2006, June 2002

[OMG 2005] OMG: *UML Superstructure Specification Version 2.0 (Document formal/05-07-04)*. `http://www.omg.org/cgi-bin/doc?formal/05-07-04`, Juli 2005

[OMG 2006] OMG: *Unified Modeling Language: Superstructure Version 2.1 (Document ptc/2006-04-02)*. `http://www.omg.org/cgi-bin/doc?ptc/2006-04-02`, April 2006

[OMG 2007a] OMG: *Systems Modeling Language (OMG SysML^{TM}) 1.0 Specification*. Proposed Available Specification ptc/2007-02-04, März 2007

[OMG 2007b] OMG: *Unified Modeling Language: Superstructure Version 2.1.1 (Document formal/2007-02-05)*. `http://www.omg.org/cgi-bin/doc?formal/2007-02-05`, April 2007

[Ort u. Mehta 2003] Ort, Ed ; Mehta, Bhakti: *Java Architecture for XML Binding (JAXB)*. `http://java.sun.com/developer/technicalArticles/WebServices/jaxb`, [zugegriffen am 11. Juli 2005], March 2003

[Overstreet u. a. 2002] Overstreet, C. M. ; Nance, Richard E. ; Balci, Osman: Issues in Enhancing Model Reuse. In: *International Conference on Grand Challenges for Modeling and Simulation, Jan. 27-31*. San Antonio, Texas, USA, 2002

[Page u. Kreutzer 2005] Page, Bernd ; Kreutzer, Wolfgang: *The Java Simulation Handbook – Simulating Discrete Event Systems with UML and Java*. Aachen : Shaker Verlag, 2005

[Parr u. Keith-Magee 2004] Parr, Shawn ; Keith-Magee, Russel: How To Apply MDA To Simulation. In: *SimTecT 2004 Simulation Conference*, 2004

[Petty u. Weisel 2003a] Petty, Mikel D. ; Weisel, Eric W.: A Composability Lexicon. In: *Proceedings of the Spring 2003 Simulation Interoperability Workshop*. Orlando FL, 2003, S. 181–187

[Petty u. Weisel 2003b] Petty, Mikel D. ; Weisel, Eric W.: A formal basis for a theory of semantic composability. In: *Proc. of the Spring 2003 Simulation Interoperability Workshop*, 2003, S. 416–423

[Petty u. a. 2005] Petty, Mikel D. ; Weisel, Eric W. ; Mielke, Roland R.: Composability Theory Overview and Update. In: *Proceedings of the Spring 2005 Simulation Interoperability Workshop*, 2005, S. 431–437

[Pidd u. Carvalho 2006] Pidd, M. ; Carvalho, A.: Simulation software: not the same yesterday, today or forever. In: *Journal of Simulation* 1 (2006), S. 7–20

[Pierce 2002] Pierce, Benjamin C.: *Types and programming languages*. Cambridge, MA, USA : MIT Press, 2002. – ISBN 0–262–16209–1

[Posse 2004] Posse, Ernesto: *DEVSlang and DEVS Operational Semantics*. Presentation at MSDL 2004 (`http://moncs.cs.mcgill.ca/MSDL/presentations/04.08.27.MSDLsummer/schedule.dtml`, [zugegriffen am 17. September 2007], August 2004

[Prenninger u. Pretschner 2004] Prenninger, Wolfgang ; Pretschner, Alexander: Abstractions for Model-Based Testing. In: *Proc. Test and Analysis of Component-based Systems (TACoS'04)*. Barcelona, March 2004

[Pullen u. a. 2005] Pullen, J. M. ; Brunton, Ryan ; Brutzman, Don ; Drake, David ; Hieb, Michael ; Morse, Katherine L. ; Tolk, Andreas: Using Web services to integrate heterogeneous simulations in a grid environment. In: *Future Generation Computer Systems* 21 (2005), S. 97–106

[QM-Lexikon 2007a] QM-Lexikon: *Validierung*. `http://www.quality.de/lexikon/validierung.htm` (zugegriffen am 18. September 2007), 2007

[QM-Lexikon 2007b] QM-Lexikon: *Verifizierung*. `http://www.quality.de/lexikon/verifizierung.htm` (zugegriffen am 18. September 2007), 2007

[Röhl 2006] Röhl, Mathias: Platform Independent Specification of Simulation Model Components. In: *ECMS 2006*. Bonn, Sankt Augustin, Germany, May 28th-31th 2006, S. 220–225

[Röhl u. a. 2007a] Röhl, Mathias ; König-Ries, Birgitta ; Uhrmacher, Adelinde M.: An Experimental Frame for Evaluating Service Trading in Mobile ad-hoc Networks. In: *Mobilität und Mobile Informationssysteme (MMS 2007)* Bd. 104, 2007 (Lect. Notes Inform.), S. 37–48

[Röhl u. a. 2007b] Röhl, Mathias ; Marquardt, Florian ; Uhrmacher, Adelinde M.: Exploiting Web Service Techniques for Composing Simulation Models. In: *Proceedings of the 2007 Winter Simulation Conference*, 2007, S. 833–841

[Röhl u. Morgenstern 2007] Röhl, Mathias ; Morgenstern, Stefan: Composing Simulation Models Using Interface Definitions based on Web Service Descriptions. In: *Proceedings of the 2007 Winter Simulation Conference*, 2007, S. 815–822

[Röhl u. Uhrmacher 2005] Röhl, Mathias ; Uhrmacher, Adelinde M.: Controlled Experimentation with Agents – Models and Implementations. In: Gleizes, Marie-Pierre (Hrsg.) ; Omicini, Andrea (Hrsg.) ; Zambonelli, Franco (Hrsg.): *Post-Proc. of the 5th Workshop on Engineering Societies in the Agents World* Bd. 3451, Springer Verlag, 2005 (LNAI), S. 292–304

[Röhl u. Uhrmacher 2005] Röhl, Mathias ; Uhrmacher, Adelinde M.: Flexible Integration of XML Into Modeling and Simulation Systems. In: *Proceedings of the 2005 Winter Simulation Conference*, 2005, S. 1813–1820

[Röhl u. Uhrmacher 2006] Röhl, Mathias ; Uhrmacher, Adelinde M.: Composing Simulations from XML-Specified Model Components. In: *Proceedings of the Winter Simulation Conference*, ACM, 2006, S. 1083–1090

[Salaün u. a. 2004] Salaün, Gwen ; Bordeaux, Lucas ; Schaerf, Marco: Describing and Reasoning on Web Services using Process Algebra. In: *Proceedings of the IEEE International Conference on Web Services (ICWS '04)*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7695–2167–3, S. 43

[Sargent 2005] Sargent, Robert G.: Verification and validation of simulation models. In: *WSC '05: Proceedings of the 37th Winter simulation conference*, Winter Simulation Conference, 2005. – ISBN 0–7803–9519–0, S. 130–143

[Sarjoughian 2006] Sarjoughian, Hessam S.: Model composability. In: *WSC '06: Proceedings of the 38th Winter simulation conference*, Winter Simulation Conference, 2006. – ISBN 1–4244–0501–7, S. 149–158

[Schmoelzer u. a. 2004] Schmoelzer, Gernot ; Teiniker, Egon ; Mitterdorfer, Stefan ; Kreiner, Christian ; Kovacs, Zsolt ; Weiss, Reinhold: Model-Driven Development of Recursive CORBA Component Assemblies. In: *euromicro* 00 (2004), S. 170–175. `http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/EURMIC.2004.1333369`. – DOI http://doi.ieeecomputersociety.org/10.1109/EURMIC.2004.1333369. – ISSN 1089–6503

[Schröter 2004] Schröter, Gunnar: *Characterization and Comparison of Formal Refinement and Development Relations for Software Modeling Techniques*, Technical University of Berlin, School of Electrical Engineering and Computer Sciences, Diss., 2004

[Schulz u. a. 2000] Schulz, Stephan ; Ewing, T.C. ; Rozenblit, Jerzy W.: Discrete Event System Specification (DEVS) and StateMate StateCharts Equivalence for Embedded Systems Modeling. In: *7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. Edinburgh, Scotland, April 2000, S. 308

[SCM 2006] SCM: *Simulation Conceptual Modeling (SCM) Study Group, Final Report*. SISO-REF-017-2006, 30 June 2006

[Selic 2004a] Selic, Bran: UML 2.0: Exploiting Abstration and Automation. In: *SD Times* (2004), Februar

[Selic 2004b] Selic, Bran V.: On the Semantic Foundations of Standard UML 2.0. In: *Formal Methods for the Design of Real-Time Systems* Bd. 3185. Springer, 2004, S. 181–199

[Shadbolt u. a. 2006] Shadbolt, Nigel ; Berners-Lee, Tim ; Hall, Wendy: The Semantic Web Revisited. In: *IEEE Intelligent Systems* 21 (2006), Nr. 3, S. 96–101

[Shen 1996] Shen, Chien-Chung: A CORBA facility for network simulation. In: *WSC '96: Proceedings of the 28th conference on Winter simulation*, 1996. – ISBN 0–7803–3383–7, S. 613–620

[Siegel 2001] Siegel, Jon: *CORBA 3: Fundamentals and Programming*. 2. John Wiley & Sons, 2001

[SimVentions 2008] SimVentions: *BOMworks*. `http://www.simventions.com/bomworks`, zugegriffen am 30. Januar 2008, 2008

[SISO 2006a] SISO: *Base Object Model (BOM) Template Specification.* März 2006. – SISO-STD-003-2006

[SISO 2006b] SISO: *Guide for Base Object Model (BOM): Use and Implementation.* März 2006. – SISO-STD-003.1-2006

[SNE 2006] SNE: *NDL: Network Description Language.* `http://www.science.uva.nl/research/sne/ndl`, 2006. – Research Group System and Network Engineering (SNE), Universiteit van Amsterdam, Faculty of Science

[Srivastava u. Koehler 2003] Srivastava, B. ; Koehler, J.: Web Service Composition - Current Solutions and Open Problems. In: *ICAPS 2003 Workshop on Planning for Web Services*, 2003

[Sun 2003] Sun: *Enterprise JavaBeans Specification, Version 2.1.* `http://java.sun.com/products/ejb/docs.html`, [zugegriffen am 13. Juli 2006], November 2003

[Sun 2005] Sun: *Java Architecture for XML Binding (JAXB).* `http://java.sun.com/xml/jaxb`, [zugegriffen am 11. Juli 2005], 2005

[Sun 2008] Sun: *Java SE 6.* `http://java.sun.com/javase/6`, [zugegriffen am 18. Januar 2008], 2008

[Szyperski 2002] Szyperski, Clemens: *Component software: beyond object-oriented programming.* 2nd. ACM Press/Addison-Wesley Publishing Co., 2002

[Szyperski 2003] Szyperski, Clemens: Component technology: what, where, and how? In: *Proceedings of the 25th International Conference on Software Engineering (ICSE '03).* Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 0–7695–1877–X, S. 684–693

[Tan u. a. 2002] Tan, Desney S. ; Zhou, Shuheng ; Ho, Jiann-Min ; Mehta, Janak S. ; Tanabe, Hideaki: Design and Evaluation of an Individually Simulated Mobility Model in Wireless Ad Hoc Networks. In: *Communication Networks and Distributed Systems Modeling and Simulation Conference 2002.* San Antonio, TX, 2002

[Tanenbaum u. van Steen 2002] Tanenbaum, Andrew S. ; Steen, Maarten van: *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2002

[Tolk 2002] Tolk, Andreas: Avoiding Another Green Elephant — A Proposal for the Next Generation HLA based on the Model Driven Architecture. In: *Simulation Interoperabilty Workshop.* Orlando, September 2002

[Tolk 2004] Tolk, Andreas: Metamodels and Mappings — Ending the Interoperability War. In: *Simulation Interoperability Workshop (SISO).* Orlando, Florida, September 2004, S. 748–761

[Tolk 2006] Tolk, Andreas: What Comes After the Semantic Web – PADS Implications for the Dynamic Web. In: *PADS '06: Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation.* Washington, DC, USA : IEEE Computer Society, 2006. – ISBN 0–7695–2587–3, S. 55

[Tolk u. Muguira 2003] Tolk, Andreas ; Muguira, James: *The Level of Conceptual Interoperability Model.* Fall Simulation Interoperability Workshop (SISO), Orlando, September 2003

[Tolk u. a. 2006] Tolk, Andreas ; Turnitsa, Charles D. ; Diallo, Saikou Y.: Composable M& S Web Services for Net-Centric Applications. In: *JDMS* 3 (2006), Januar, Nr. 1, S. 27–44

[Turnitsa 2005] Turnitsa, C.D.: Extending the Levels of Conceptual Interoperability Model. In: *IEEE Summer Computer Simulation Conference*, IEEE CS Press, 2005

[Uhrmacher 2001] Uhrmacher, Adelinde M.: Dynamic Structures in Modeling and Simulation - A Reflective Approach. In: *ACM Transactions on Modeling and Simulation* 11 (2001), Nr. 2, S. 206–232

[Uhrmacher u. a. 2005] Uhrmacher, Adelinde M. ; Degenring, Daniela ; Lemcke, Jens ; Krahmer, Mario: Towards Re-Using Model Components in Systems Biology. In: *CMSB 2004* Bd. 3082, Springer, 2005 (LNBI), S. 192–206

[Uhrmacher u. a. 2002] Uhrmacher, Adelinde M. ; Röhl, M. ; Kullick, B: The Role of Reflection in Simulating and Testing Agents: An Exploration Based on the Simulation System James. In: *Applied Artificial Intelligence* 16 (2002), October-December, Nr. 9-10, S. 795–811

[Valentin u. a. 2003a] Valentin, Edwin C. ; Verbraeck, Alexander ; Sol, Henk G.: Advantages and Disadvantages of Building Blocks in Simulation Studies. In: *Simulation in Industry 15th European Simulation Symposium*, SCS-European Publishing House, 2003, S. 141–148

[Valentin u. a. 2003b] Valentin, Edwin C. ; Verbraeck, Alexander ; Sol, Henk G.: Effect of Simulation Building Blocks on Simulation Model Development. In: Ibarra, A. (Hrsg.): *Proceedings of International Conference of Technology, Policy and Innovation*, 2003, S. 54–61

[Vangheluwe 2000] Vangheluwe, Hans: DEVS As a Common Denominator for Multi-formalism Hybrid System Modeling. In: *Proceedings of the IEEE International Symposium on Computer Aided Control System Design*. Anchorage, Alaska, September 2000, S. 129–134

[Vangheluwe u. de Lara 2002] Vangheluwe, Hans ; Lara, Juan de: XML-based modeling and simulation: meta-models are models too. In: *WSC '02: Proceedings of the 34th conference on Winter simulation*, Winter Simulation Conference, 2002. – ISBN 0–7803–7615–3, S. 597–605

[Verbraeck 2004] Verbraeck, Alexander: Component-based distributed simulations: the way forward? In: *PADS '04: Proceedings of the eighteenth workshop on Parallel and distributed simulation*. New York, NY, USA : ACM Press, 2004. – ISBN 0–7695–2111–8, S. 141–148

[W3C 1999] W3C: *XSL Transformations (XSLT) Version 1.0*. `http://www.w3.org/TR/xslt`, [zugegriffen am 11. Juli 2005], November 1999

[W3C 2004a] W3C: *Document Object Model (DOM) Level 3 Core Specification*. `http://www.w3.org/TR/DOM-Level-3-Core`, zugegriffen am 23. Januar 2008, April 2004. – Version 1.0, W3C Recommendation 07 April 2004

[W3C 2004b] W3C: *OWL-S: Semantic Markup for Web Services*. `http://www.w3.org/Submission/OWL-S`, November 2004. – Member Submission 22 November 2004

[W3C 2004c] W3C: *RDF Primer*. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`, 2004. – W3C Recommendation 10 February 2004

[W3C 2004d] W3C: *Web Services Architecture*. `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`, 2004. – W3C Working Group Note 11 February 2004

[W3C 2004e] W3C: *Web Services Glossary*. `http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/`, 2004. – W3C Working Group Note 11 February 2004

[W3C 2004f] W3C: *XML Schema Part 0: Primer Second Edition*. `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`, Oktober 2004. – W3C Recommendation 28 October 2004

[W3C 2005] W3C: *Web Services Choreography Description Language Version 1.0*. `http://www.w3.org/TR/ws-cdl-10`, zugegriffen am 22. Januar 2008, November 2005. – W3C Candidate Recommendation 9 November 2005

[W3C 2006a] W3C: *Semantic Annotations for WSDL*. `http://www.w3.org/TR/2006/WD-sawsdl-20060928/`, 2006. – W3C Working Draft 28 September 2006

[W3C 2006b] W3C: *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. `http://www.w3.org/TR/2006/CR-wsdl20-primer-20060327`, 2006. – W3C Candidate Recommendation 27 March 2006

[W3C 2006c] W3C: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. `http://www.w3.org/TR/2006/CR-wsdl20-20060327`, 2006. – W3C Candidate Recommendation 27 March 2006

[W3C 2007] W3C: *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. `http://www.w3.org/TR/2007/WD-scxml-20070221`, 2007. – W3C Working Draft 21 February 2007

[Walsh 2004] Walsh, Norman: *Infoset Equality*. `http://norman.walsh.name/2004/05/19/infoset-equal`, Mai 2004. – From the Technical Plenary, a URI that got lost: a quick "off-the-cuff" definition for XML chunk equality based on the Infoset.

[Wang u. a. 2001] Wang, Nanbor ; Schmidt, Douglas C. ; O'Ryan, Carlos: Overview of the CORBA component model. (2001), S. 557–571. ISBN 0–201–70485–4

[Ware 2005] Ware, Tech K.: *LMX: W3C XML Schema to C/C++ Data Binding Tool*. `http://www.tech-know-ware.com/lmx/`, [zugegriffen am 11. Juli 2005], March 2005

[Weisel u. a. 2003] Weisel, Eric W. ; Petty, Mikel D. ; Mielke, Roland R.: Validity of Models and Classes of Models in Semantic Composability. In: *Proceedings of the Fall 2003 Simulation Interoperability Workshop*, 2003, S. 526–536

[Weisel u. a. 2004] Weisel, Eric W. ; Petty, Mikel D. ; Mielke, Roland R.: A Survey of Engineering Approaches to Composability. In: *Fall Simulation Interoperability Workshop*. Washington, D.C., April 2004

[Yilmaz 2004a] Yilmaz, Levent: On the need for contextualized introspective models to improve reuse and composability of defense simulations. In: *JDMS* 1 (2004), August, Nr. 3, S. 141–151

[Yilmaz 2004b] Yilmaz, Levent: Verifying Collaborative Behavior in Component-Based DEVS Models. In: *Simulation* 80 (2004), July–August, Nr. 7–8, S. 399–415

[Yilmaz u. Ören 2006] Yilmaz, Levent ; Ören, Tuncer I.: Prospective Issues in Simulation Model Composability: Basic Concepts to Advance Theory, Methodology, and Technology. In: *The MSIAC's M& S Journal Online* 6 (2006), September, Nr. 2

[Zeigler 1976] Zeigler, Bernard: *Theory of Modelling and Simulation*. John Wiley & Sons, 1976

[Zeigler 1984] Zeigler, Bernard P.: *Multifacetted modelling and discrete event simulation*. Academic Press, 1984

[Zeigler 1990] Zeigler, Bernard P.: *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*. San Diego : Academic Press, 1990

[Zeigler u. a. 2000] Zeigler, Bernard P. ; Praehofer, Herbert ; Kim, Tag G.: *Theory of Modeling and Simulation*. 2nd. London : Academic Press, 2000

[Zeigler u. Sarjoughian 1999] Zeigler, Bernard P. ; Sarjoughian, Hessam S.: Support for Hierarchical Modular Component-based Model Construction in DEVS/HLA. In: *Simulation Interoperability Workshop*. Orlando, FL., März 1999

[Zeigler u. Sarjoughian 2002] Zeigler, Bernard P. ; Sarjoughian, Hessam S.: Implications of M& S Foundations for the V& V of Large Scale Complex Simulation Models. In: *Proceedings of the Foundations for V&V in the 21st Century Workshop*. Laurel, MD, 2002

[Zeigler u. Sarjoughian 2005] Zeigler, Bernard P. ; Sarjoughian, Hessam S.: *Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models*. Arizona Center for Integrative Modleing and Simulation, University of Arizona and Arizona State University, Tucson, Arizona, USA, Januar 2005

[Zhan u. a. 2004] Zhan, Xuede ; Miao, Huaikou ; Liu, Ling: Formalizing the Semantics of UML Statecharts with Z*. In: *CIT '04: Proceedings of the The Fourth International Conference on Computer and Information Technology (CIT'04)*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7695–2216–5, S. 1116–1121

[Zimmerman u. a. 2002] Zimmerman, Philomena ; Heusmann, Jim ; Sciarretta, Al: *Executing the DoD Modeling and Simulation Stragegy - Making Simulation Interoperability and Reuse a Reality*. SISO Document 02F-SIW-052, 2002

[Zinoviev 2005] Zinoviev, Dmitry: Mapping DEVS Models onto UML Models. In: *Proc. of the 2005 DEVS Integrative M&S Symposium*. San Diego, CA, April 2005, S. 101–106

Anlagen

Mathias Röhl

Thesen

1. Existierende Kompositionsansätze bieten nur limitierte Möglichkeiten, um Modelle dezentral zu entwickeln und sie als Komponenten in Experimenten zu verwenden. Während modular-hierarchische Formalismen nicht explizit zwischen Schnittstelle und Implementierung unterscheiden, bieten simulationsbasierte Ansätze wenig Unterstützung beim Modellieren und eine beschränkte Flexibilität beim Experimentieren.

2. Schnittstellen für Softwarekomponenten und Dienste geben typischerweise abstrakte Kommunikationsendpunkte bekannt, über die Funktionalität angezeigt bzw. gefordert wird. Schnittstellenbeschreibungen werden üblicherweise in Rollen aufgeteilt, die jeweils unterschiedliche Aspekte des gesamten Kommunikationspotentials einer Implementierung beschreiben.

3. Im Kontext von Web Services ist mit WSDL eine standardisierte und allgemein akzeptierte Sprache zur Beschreibung von Schnittstellen entstanden. WSDL integriert sich gut mit einer Reihe weiterer XML-basierter Standards, mit denen sich der semantische und dynamische Aspekt der Kompatibilität abdecken lässt.

4. Standards wie URIs, XSD und SAWSDL eignen sich direkt auch für die Modellierung und Simulation, um Komponenten flexibel und entkoppelt voneinander entwickeln zu können.

5. Mit den Beschreibungsmitteln der entwickelten Kompositionsplattform können Modelle räumlich und zeitlich verteilt entwickelt, als XML-Dokumente in Datenbanken verwaltet sowie modular-hierarchisch komponiert werden.

6. Die entwickelten Beschreibungsmittel erlauben, separate Schnittstellenbeschreibungen mit existierenden Modellierungsformalismen zu kombinieren.

7. Im Gegensatz zu den Beschreibungsmitteln der UML und der SysML sind die vorgestellten Kompositionsstrukturen auf die ereignisorientierte Modellierung zugeschnitten und wurden mit einer formalen Semantik ausgestattet.

8. Auf Grundlage einer expliziten Semantikdefinition für Komponenten konnten Eigenschaften wie die Kompatibilität verbundener Rollen, die Verfeinerung von Schnittstellen in Modellimplementierungen und die Korrektheit von Komponenten formal definiert werden.

9. Die Definition von bewahrender und reflektierender Verfeinerung ermöglicht, sowohl die Anforderungen an Komponenten als auch die Kontextanforderungen von Komponenten zu prüfen. Sind Komponentendefinitionen korrekt, lassen sich aus ihnen wohlgeformte Simulationsmodelle ableiten.

10. Eine konkrete XML-basierte Syntax, definiert im W3C-Standard XSD, erleichtert den Austausch von Definitionseinheiten der Plattform.

11. Das Kompositionswerkzeug CoMo realisiert die Kompositionsplattform in Java als erweiterbares Rahmenwerk, mit dem Kompositionen instantiiert, analysiert und operationale Modell generiert werden können.

12. James II fungiert für CoMo als Zielplattform, mit der sich komponierte Simulationsmodelle effizient ausführen lassen.

13. Die Praxistauglichkeit der entwickelten Kompositionsplattform wird durch ihre Anwendung im Rahmen der Simulation von Dienstvermittlung in mobilen Ad-Hoc-Netzen demonstriert.