Skalierbare Programmier-Paradigmen und Architekturen im Bereich Distributed Artificial Intelligence

Sigurd Sippel

09. März 2014

1 Einleitung

Rechenleistung steigert sich bei stagnierenden Taktraten der Prozessoren nur noch mit Verteilung der Rechenlast auf mehrere Kerne oder vernetzter Rechner [MD10, S. 239][ZD10, S. 388]. Damit eine Applikation verteilt werden kann, benötigt diese eine Kommunikationsmodell, dass die Verteilung zulässt. Am weitest verbreitet ist Shared State Concurrency. Dabei wird eine gemeinsame Resource von mehreren Threads geteilt und durch Locking geschützt.

Durch Locking wird das an sich asynchrone Verhalten zweier Threads synchronisiert. Die Synchronisierung ist fehleranfällig [NM92, S. 75f], da bei falscher Anwendung Race Conditions auftreten, die zu einem nicht-determistischen Verhalten führen. Fehler, wie Deadlocks oder fehlerhafte Werten im Shared State, sind schwer zu finden und die Komplexität der Synchronisierung nimmt mit steigender Komplexität der Applikation zu [BAD+09, S. 97]. Synchronisierung bedeutet, dass nur ein Thread den Shared State zur gleichen Zeit verwenden kann, dadurch bleiben Rechenkapazitäten durch Wartzeiten ungenutzt, sodass das Shared State Concurrency nur schlecht skaliert.

Im Bereich Distributed Artificial Intelligence werden skalierbare, verteilbare Systeme benötigt [Huh09, S. 46f] um viele Daten auswerten zu können und beteiligte Systeme echtzeitfähig einbinden zu können. Der Fokus dieser Arbeit liegt daher auf Skalierbarkeit und lose gekoppelte Architekturen, die Anwendung beispielsweise in Echzeitstrategiespielen wie 0AD [0AD14] finden.

Diese Hausarbeit gliedert sich in folgende sechs Abschnitte: Zunächst werden Objekt-orientierte und funktionale Welt kurz gegenübergestellt (Abschnitt 2) und danach auf die funktionale Welt genauer eingegangen. Das funktionale Aktormodell wird vorgestellt und von Agenten abgegrenzt (Abschnitt 3). Aufbauend auf Agenten und Aktoren werden der Kommmunikationsmechnismus Publish und Subscribe sowie die Blackboard Architektur gegenübergestellt (Abschnitt 4). Es wird das für Agenten und Aktoren ausgelegte Architekturmuster FIPA (Abschnitt 5) und das Resourcen Descriptionframework RDF zum Design von Nachrichten vorgestellt. Aktuelle Konferenzen und die Arbeit der drei Keynotespeaker im Bezug auf Spiele werden kurz vorgestellt (Abschnitt 6). Zum Abschluss wird eine konkrete Problemstellung dargelegt, die mit den vorgestellten Konzepten lösbar ist und ein Fazit gezogen (Abschnitt 7).

2 Zwei Welten: Objekt-orientiert vs. funktional

Es gibt zwei wichtige Programmierparadigmen: Die Objekorientierung und die funktionale Programmierung [FF06, S. 145]. Die objektorienterten Sprachen spalten sich nochmal in klassenbasierte und prototypbasierte Sprachen auf. Klassenbasierte Sprachen, wie Java, haben eine globale Vererbungshierachie von Klassen, die zum Erzeugen von Objekten dienen. Die Prototypbasierten Sprachen, wie Javascript, besitzen keine Klassen. Die Objekte werden zur Laufzeit durch Klonen von existierenden Objekten erzeugt. Beim Klonen wird kopiert und nicht referenziert.

Funktionale Sprachen basieren auf unveränderlichen Werten. Es gibt kein veränderlichen Zustand wie bei klassenbasierten Sprachen, daher wird auch keine Unterscheidung zwischen Feldern und Funktionen. Der Zugriff auf Felder kann durch nullstellige Funktionen erfolgen. Auf Basis von algebraischen Datenstrukturen kann Pattern Matching angwendet werden. In klassenbasierten Sprachen ist dies nicht möglich, denn es ist nur möglich auf die Klasse zu prüfen, jedoch nicht auf das innere einen Typs. Funktionale und prototypbasierte Sprachen haben keinen veränderliche Zustand und sind daher einfacher vermischbar als funktionale und objektorientierte Sprachen [FF06, S. 146]. Ein wesentliches Ausdrucksmittel der funktionalen Sprachen ist Pattern Matching auf algebraischen Datenstrukturen.

Objektorientierte Sprachen beinhalten folgende Eigenschaften [FF06, S. 146]: Sie unterstützen die Kapselung von Daten und Code, sodass der Zugriff über eine einzige Schnittstelle erfolgt. Objektdefinitionen können vererbt werden und Objekte nehmen durch Polymorphie mehrere Typen an. Dagegen stehen Seiteneffekte durch innere Zustände, die Finden von Fehlern stark erschweren können.

Im Bezug auf Nebenläufigkeit teilen sich die beiden Welten wie folgt auf [Nie88, S. 174]: Wird in der Objekt-orientierte Welt der Zustand des Objekts geteilt, wird dieser durch Locking geschützt (Seperation). Dann kann der Zustand verändert (Manipulation) und freigegeben werden. Die Verwendung des Shared States erfolgt blockierend. In der funktionalen Welt werden die unveränderliche Werte durch Message Passing verteilt. Durch Queues werden Messages asychron verarbeitet, denn das Versenden und das Verarbeiten geschieht getrennt von einander und ist daher nicht blockierend.

2.1 Algebraische Datenstrukturen

Die funktionale Programmierung blickt aus der Perspektive der Mathematik auf Algorithmen und wendet deren Gesetze an. D.h. es werden Datenstrukturen und darauf basierende Funktionen definiert [FF06, S. 146].

Auf Basis von elementaren Typen werden eigene Typen definiert [GWK12, S. 124]: Komposition aus mehreren Typen könen durch ein Kreuzprodukt definiert werden, genannt Produkttyp (Formel 1). Summentypen (Formel 2) vereinigt mehrere Typen [GWK12, S. 126]. Der Aufzählungstyp (Formel 3), der aus Java als Enumeration bekannt ist, stellt eine definierte Menge von Elementen darstellt, die zu einem Typ zusammengefasst werden.

$$type\ Interval = Euro \times Euro \tag{1}$$

$$type\ Currency = Euro \mid Dollar \mid Yen \tag{2}$$

$$type\ CurrencyType = \{\ Euro,\ Dollar,\ Yen\ \} \tag{3}$$

Sind die Bestandteile des Kreuzproduktes Interval mit einem Namen versehen, gilt folgendes: Die inneren Werte des Kreuzproduktes können beispielsweise mit den Selektoren $start(i)$ oder $end(i)$ abgerufen werde. Die objektbezogene Schreibweise und aus vielen Sprachen bekannte Schreibweise $i.start$ ist dabei äquivalent zu $start(i)$ zu verstehen.

Die Typen können auch rekursiv verwendet werden [GWK12, S. 124] um z.B. einen verkette Liste zu erzeugen:

$$type\ CurrencySeq = \{\} \mid (head = Currency :: tail = CurrencySeq) \tag{4}$$

Die Funktionen haben keine Seiteneffekte, genannt pure functions, sodass das Ergebnis eine Funktion bei gleichen Parametern immer gleich bleibt. Es werden Parameter und Rückgabewert definiert (Formel 5). Der Definitionsbereich entspricht dem Parametern und Wertebereich dem Rückgabewert. Eine Konstante ist eine nullstellige Funktion (Formel 6).

$$fun\ f:\ (A_1\ \times\ \dots\ \times\ A_n) \to B \tag{5}$$

$$val\ c:\ B \tag{6}$$

2.2 Anonyme Funktionen durch Lambdakalkül

Vergleichend zu der allgemeine Definition einer Funktion ist es auch möglich eine Funktion mittels der Lambdanotation (Formel 7) zu beschreiben [PH06, S. 14]

$$f:\ \lambda x_1, \ldots, \lambda x_n \bullet e \tag{7}$$

Dabei werden in diesem λ Ausdruck die Variablen an den Ausdruck e gebunden. Diese Schreibweise bietet die Möglichkeit Funktionen ohne Namen als anonyme Funktionen zu definieren. Anonyme Funktionen können als Parameter verwendet werden. Beispielhaft ist eine Filterfunktion denkbar, die über eine Sequenz von Currency iteriert und dabei eine Liste erzeugt, die der übergebenen anonymen Filterfunktion $Integer \rightarrow Boolean$ entspricht:

$$[filter(\lambda x \bullet x.value > 2)\langle Euro(1), Euro(2), Euro(3), Euro(4)\rangle \tag{8}$$

In diesem Beispiel (Formel 8) ergibt sich als Ergebnis die Sequenz $\langle Euro(3), Euro(4)\rangle$. Funktionen, die Funktionen beherbergen, nennen sich Funktionen höherer Ordnung.

2.3 Patternmatching

Die Basis für Patternmatching (Musterabgleich) ist die partielle Funktion. Existiert für eine Funktion $fun\ f:\ (A_1\ \times\ \ldots\ \times\ A_n) \rightarrow B$ ein konkretes Element, für das die Funktion nicht definiert ist, ist die Funktion partiell. Es wird mit $\nrightarrow$ gekennzeichnet. Das aus switch-case Konstrukt, welches aus Sprachen wie C und Java bekannt ist, ermöglicht für wenige primitive Typen eine Verarbeitung, die im Grund einer partiellen Funktion entspricht ohne dabei die Funktion als Typ abzubilden. Mit Pattern Matching werden passende Typen oder Werte gefunden [PH06, S. 26]. Dafür wird ein Pattern, welches einen komplexen Ausdruck abbilden kann, auf einen Block abgebildet (Codebeispiel 1). Wird ein passendes Muster zur zu prüfenden Expression gefunden, wird der zugehörige Block ausgeführt. Ein Match beendet das Matching.

Codebeispiel 1: Pattern Matching

```
expression match {
        case pattern1 => block1
        ...
        case patternn => blockn
}
```

Prüft das Pattern auf ein Produkttyp, kann ebenso ein Pattern, dass den inneren Aufbau beschreibt, verwendet werden. Beispielsweise eine verkette Liste mit mindestens einem Element kann durch Pattern Matching getroffen werden ($CurrencySeq(x|_)$. Zu beachten ist dass die Expression der gemeinsame Obertyp aller Pattern sein muss, welches vom Compiler zu prüfen ist. Die Auswertung erfolgt dagegen zur Laufzeit.

2.4 Vereinigung beider Welten

Die funktionale Programmierung geht dabei einen konträren Weg zu der Objektorientierung. Trotzdem können beide Welten vereinigt werden. In der objektorientierten Sprache Java kann eine funktionale Datenstruktur per Konvention definiert werden: Es wird eine Klasse als Typdefiniton verwendet, in der alle Instanzvariablen als unveränderlich (final) deklariert werden und alle Methoden öffentlich sind [GWK12, S. 126]. Allerdings fehlen Java Ausdrucksmöglichkeiten wie Pattern Matching oder Lambdas. Scala wurde entwickelt um beide Welten zu verbinden. Daher enthält sie sowohl die funktionalen als auch die objektorientierten Ausdrucksmöglichkeiten und Schnittstellen. Beispielsweise gibt es Mutable Collections und Immutable Collections, die voneinander vollständig getrennt sind, jedoch bidirektional konvertiert werden können. Diese Schnittstellen sind nötig um Programmierparadigma und dessen Ausdrucksmöglichkeiten vollständig nutzen zu können.

3 Aktorenmodell

Das Aktorenmodell ist ein Architekturmuster [HBS73, S. 235], dass auf Basis von Message Passing eine verteilte Applikation ermöglicht, ohne dabei einen Shared State zu benötigen [DKVCD12, S. 11]. Das Aktorenmodell ist vielfach implementiert, teils in funktionalen Sprachen wie Erlang [Erl14], teils als Frameworks wie Akka [Akk14] oder libcppa [lib14]. Funktionale Sprachen garantieren das keine Race Conditions auftreten und das keine Wert veränderlich sind, dafür sind sie eingeschränkt [DKVCD12, S. 11]. In Sprachen, die funktionale und objektierte Aspekte verbinden, wie Scala, ist dies nur per Konvention möglich.

Ein Aktor ist ein leichtgewichtiger und autonomer Prozess [HBS73, S. 235]. Dem Aktor können Nachrichten geschickt werden, für die er ein eigenes Verhalten bereit hält, welches sich zur Laufzeit nicht verändert. Das Aktormodell ist nicht an spezielle Datenstrukturen gebunden, sodass Aktoren lose gekoppelt untereinander kommunizieren können. Leichtgewichtig ist ein Aktor, da er kein eigenen Prozess besitzt, sondern zum Verarbeiten einer Nachrichten einen Prozess zugewiesen bekommt.

3.0.1 Aktorenframework Akka

Das Aktorenframework Akka implemntiert das Aktorenmodell von Carl Hewitt in Scala (1): Jeder Aktor besitzt eine Mailbox. Nachrichten die an den Aktor geschickt werden, werden zunächst in der Mailbox zwischengespeichert. Der Dispatcher hält einen Pool von Threads vor, von denen ein Thread vom Dispatcher ausgewählt wird. Zusammen mit einer Nachricht aus der Mailbox kann der Aktor die Nachricht verarbeiten [Typ13, S. 11].

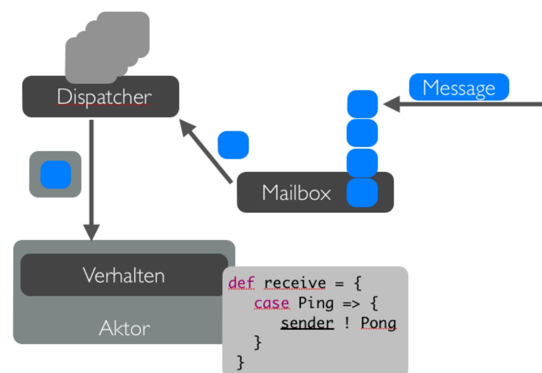

Abbildung 1: Aktormodell in Akka

Eine Mailbox ist eine Queue, die nicht-deterministisch ist. Es gibt keine zeitliche Ordnung bei der Verarbeitung der Nachrichten. Dadurch kann keine Aussage darüber gemacht werden, wann eine Nachricht verarbeitet wird [Typ13, S. 14].

In einem Aktorensystem existieren eine beliebige Anzahl von Aktoren, die jeweils kleinschrittig Aufgaben lösen können. Lösen bedeutet auch Teilen und Delegieren. Eine große Aufgabe wird aufgeteilt und dessen Teilaufgaben an die dafür zuständigen Aktoren delegiert.

3.1 Kommunikation der Aktoren

Ein Aktor wird definiert, indem eine beliebige Klasse von dem Trait Actor erbt. Dazu muss die partiellen Funktion receive $Any \nrightarrow Unit$ implementiert werden. Any ist in Scala der oberste Typ aller Typen, damit kann jede Instanz einer Klasse eine Nachricht sein. Unit ist vergleichbar mit void aus Java. Nachdem der Aktor in seinem Lebenszyklus in den Zustand started übergeht, können die Nachrichten in der Mailbox verarbeitet werden.

Durch Patternmatching werden einzelne Nachrichten zugeordnet einer dafür vorgesehenen Aktion zugeordnet. Im folgenden Beispiel (Codebeispiel 2) wird die Nachricht des Typs Int ausgegeben, und bei allen anderen Nachrichten wird ein Standardtext ausgegeben. Durch einen Unterstrich wird die

zugeordnet Aktion für alle Nachrichten ausgeführt. Dier hier dargestellte receive Implementierung ist total, denn es gibt für alle Nachrichten einen passenden Fall. Da die receive Funktion, konzeptionell gesehen, partiell ist, muss nicht für jede Nachricht ein passenden Fall bereitgestellt werden. Wenn kein Fall getroffen wird, erfolgt eine Exception.

Codebeispiel 2: Beispiel eines Aktors

```
class ExampleActor extends Actor {
    def receive = {
        case i: Int => log.debug("int")
        case _      => log.debug("something else")
    }
}
```

Ein Actor wird durch einen Bang oder Tell genannten Operator (kurz: !) eine Nachricht gesendet (Formel 9). Es wird nicht auf eine Antwort gewartet. Die Nachricht wird in die Queue des Zielaktors gelegt.

$$actor \; ! \; message \tag{9}$$

Durch das Absenden der Nachricht gibt der Absender seine Identität preis, die über die Variable sender (Formel 10) verwendet werden kann und darüber auch Nachrichten an den Absender zurückgeschickt werden können [Typ13, S. 214].

$$sender \; ! \; message \tag{10}$$

3.2 Umgang mit nicht-funktionalen Funktionen

Zum Kapseln von blockierenden oder nicht funktionalen Funktionen (z.B. Lesen vom Dateisystem) bietet Akka eine eigene Future Implementierung an. Die Funktion ask (Formel 11) sendet beispielsweise an einen Aktor eine Message vom Typ Read. Dieser liest die Datei und sendet den Inhalt zurück. Dafür ist ein Timeout vorgegeben. Sendet der Actor nicht innerhalb von dem Timeout die Antwort, wird eine Exception geworfen.

$$\begin{aligned} valfuture &= fileactor.ask(message)(duration) \\ valfuture &= fileactor.ask(Read(''my/path''))(5 \; seconds) \end{aligned} \tag{11}$$

Ist ein Aktor durch Verwendung objekt-orientierter Bibliotheken nicht thread-safe, kann diesem ein spezieller PinnedDispatcher gegeben werden, sodass der Aktor einen eigenen Thread bekommt [Typ13, S. 257]. Ein PinnedDispatcher ist weniger performant, jedoch wird dadurch eine Bereich geschaffen, der von den anderen Aktoren losgelöst ist und ein Shared State gefahrlos existieren kann [DKVCD12, S. 13]. Es wird keine inperformante Sicherstellung der Konsistenz (durch z.B. Replizieren) benötigt. Es können beide Welten parallel existieren ohne sich negativ zu berühren. Da die Nachrichten sequentiell abgearbeitet werden, kann auch nicht parallel auf den Shared State zugegriffen werden.

3.3 Aufbau und Referenzierung

Aktoren sind Teil eines Aktorensystems (Codebeispiel 3), dass für die Aktoren eine lebenswichtige Einheit darstellt. Es liefert im Hintegrund Dispatcher- und Mailbox Implementation, außerdem realisiert es das Message Passing [Typ13, S. 11].

Codebeispiel 3: Instanziierung eines Aktorensystems

```
object MyApplication extends App {
        val system = ActorSystem("FileReaderSystem")
        val fileactor = system.actorOf(Props[FileActor])
}
```

Die Referenzierung der Aktoren (Formel 12) erfolgt auf Basis einer URI [Typ13, S. 18f], eine direkte Referenzierung ist nicht möglich. Jeder Aktor hat einen Parent Aktor, der oberste Aktor ist der Root Aktor. Dem Root ist vom System vorgegeben der User Actor unterstellt. Alle nicht vom Akka vorgegbenen Aktoren sind der User unterstellt. Dadurch entsteht ein Baum.

$$akka://actorsystem/user/parent/child \tag{12}$$

Die die URI repräsentiert den Pfad im Baum. Das Aktorensystem kann die URI auflösen, indem es entscheidet, mit welchem Aktorensystem es kommunizieren muss. Ist es das eigene, kann es den eigenen Baum verwenden. Bei einem fremden Aktorensystem muss dazu die Anfrage dorthin weitergeleitet werden.

3.4 Supervising

Parent Aktoren erschaffen nicht nur Child Aktoren, sie wachen auch über ihre Child Aktoren. Kommt es bei einem Child zu einer Exception, entscheidet der Parent welche Strategie [Typ13, S. 15] zum Abfangen des Fehler verwendet wird (Codebeispiel 4). Zunächst wird entschieden, ob ein Fehler bei einem Child eine Reaktion bei allen (AllForOneStrategy) oder nur bei dem einen Child durchgeführt wird (OneForOneStrategy). Danach wird eine Abbildung $Exception \rightarrow Reaction$ durch Pattern Matching realisiert. Es gibt vier Möglichkeiten: Die Nachricht kann erneut verarbeitet werden (Resume). Der Aktor (inkl. aller seiner Childs) wird neugestartet (Restart). Der Aktor wird gestoppt (Stop) oder der Fehler wird an den Parent des Parents weitergereicht (Escalate).

Codebeispiel 4: Strategie für Fehlerbehandlung

```
override val supervisorStrategy =
  OneForOneStrategy {
    case _: ArithmeticException      => Resume
    case _: NullPointerException     => Restart
    case _: IllegalArgumentException => Stop
    case _: Exception                => Escalate
  }
```

3.5 Nachteile des Aktormodells

Nach außen bieten Aktoren Nachrichten-gesteuerte Services an, die interne Umsetzung ist davon jedoch zunächst unberührt. Anders als im originalen Aktorenmodell sind in Akka Zustände möglich, die die Nebenläufigkeit beeinflussen können. Die Verarbeitung der Nachrichten verläuft sequentiell, ein Aktor bleibt solange aktiv, bis die Verarbeitung der aktuellen Nachricht abgeschlossen ist [RY13, S. 1:1]. Blockiert die Verarbeitung der Nachricht, blockiert der Aktor. Nachfolgende Nachrichten können nicht weiter verarbeitet werden. Das Aktormodell liefert außerdem weder ein Möglichkeit eine bestimmte Reihenfolge in der Verarbeitung einzuhalten, noch einer Sicherstellung der Zustellung von Nachrichten. Das nötige Umdenken um Aktoren zu entwickeln, behindert Entwickler mit objekt-orientiertem Hintergrund.

3.6 Abgrenzung zu Agenten

Agenten sind gleichbar zu Aktoren. Sie kommunizieren über Nachrichten, sind autonom und bieten einen Service an. Aktoren sind im Gegensatz zu Agenten reaktiv, d.h. sie reagieren nur auf Aufforderung, wie z.B. eine Nachricht [RY13, S. 1:3]. Agenten dagegen sind proaktiv, denn sie können durch einen Sensor ihre Umwelt ertasten und durch ein Aktuator auf die Umwelt Einfluss nehmen [Woo02, S. 5]. Dies kann durch Nachrichten abgebildet werden. Eines der wesentlichen Ziele ist das Lernen, wofür sie auch einen internen Zustand benötigen, indem sie ihre Erfahrung speichern.

4 Vergleich: Publish and Subscribe vs. Blackboard

Publish and Subscribe (Pub/Sub) ist ein Kommunikationsmchanismus, der einen lose gekoppelten, skalierbaren Nachrichtenaustausch ermöglicht [BESP08, S. 23]. Denn Empfänger der Nachrichten müssen für den Versender nicht bekannt sein, außerdem erfolgt die Zustellung der Nachrichten zum Empfänger unabhängig vom Zustand des Empfängers.

Pub/Sub funktioniert grundlegend folgendermaßen: Ein Interessent bekundet sein Interesse bei einem Message Broker. Der Sender (Publisher) sendet seine Nachrichten an den Message Broker, dieser leitet die Nachrichten an die Empfänger (Subscribers) weiter [BESP08, S. 24]. Es muss mindestens ein Message Broker vorhanden sein, es können jedoch beliebig viele sein.

Es gibt zwei Typen von Pub/Sub [CJ10, S. 9:5]: Beim Topic-based gibt es mehrere Topics, zu den eine Subscription möglich ist, ein Publisher sendet Message als Tupel (Topic,Value), welches vom Broker an die Subscriber gesendet wird, die für das Topic Interesse bekundet haben. Beim Content-based gibt es keine Topics sondern Attribute, auf die Interesse bekundet werden kann. Ein Publisher hat die Möglichkeit eine Menge von Attributen anstatt eines Topics dem Value beizufügen.

Bei mehreren Message Brokern gilt: Die Netzwerktopologie kann entweder statisch oder dynamisch sein. Bei einer dynamischen Topolgie kann beispielsweise ein Overlay Netzwerk auf Basis eine Distributed Hash Tables (DHT) verwendet werden [BESP08, S. 24][FCFB05, S. 38]. Die einzelnen Knoten auf dem Ring können sich dabei dynamisch auf Performanceunterschiede im Netzwerk einstellen, indem sie ihre Routingtabelle anpassen. Der Lookup (um z.B. ein Subscribe(key1,key2,..) durchzuführen) erfolgt baumartig durch das DHT, sodass der Kommunikationsoverhead mit einem BigO(log n) möglichst klein bleibt [SMK+01, S. 149]. Alternativ dazu gibt es Load Balancer [CJ10, S. 9:11], bei dem nicht direkt auf die Message Broker, sondern zunächst auf einzelne Load Balancer zugegriffen wird, die auf die Message Broker weiterleiten.

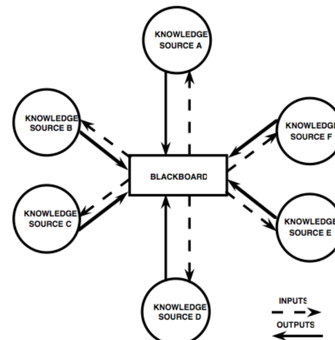

Abbildung 2: Verbindungsgraph zwischen Blackboard und Knwoledge Sourcen [McM03, S. 4]

Das Bloackboard Pattern ist mit Pub/Sub vergleichbar. Das Blackboard ist eine zentraler Ablage (Repository) für Daten [OAF08, S. 22:1]. Es gibt Wissensquellen (Kownlegde Sources), die in der Lage sind vom Balckboard zu lesen, die Daten zu verarbeiten und zu aktualisieren (Abbildung 2). Eine Kontrolleinheit steuert die Synchronisierung, dazu aktiviert diese die einzelnen Wissensquellen. Diese holen sich die Daten aus dem Blackboard und speichern dor das Ergebnis ihrer Verarbeitung. Durch das Blackboard muss keine Knowledge Source bekannt sein, wenn eine neue Aufgabe auf das Blackboard geschrieben wird. Im Aktorenmodell muss der jeweilige Aktor, der den gewünschten Service anbietet, bekannt sein. Beim Pub/Sub muss der Publisher bekannt sein, damit eine Subscription möglich ist. Das Blackboard Pattern gibt nicht vor wie die Kommunikation auf unterster Ebene abläuft und ist daher eine Architektur, während Pub/Sub ein Kommunikationsmechanismus ist.

Als Wissensquellen eignen sich Aktoren oder Agenten [GGS11, S. 163], die jeweils Services bereitstellen und durch das Blackboard mit Aufgaben versorgt werden. Das Blackboard Pattern erfordert, dass sich alle Beteiligten geegenseitig vetrauen, da alle Informationen sich an einem zentralen Ort befinden und von jedem abgerufen werden können. Die Kontrolleinheit muss entscheiden, welches Aufgabe als nächsten zu behandeln ist.

5 Architekturmuster für Agenten und Actors: FIPA

Die Foundation of Intelligent Physical Agents [IPA14], kurz FIPA, spezifiierte ab 1997 ein Archtekturmuster für Actors, Agents und Webservices [GLMS07, S. 1413]. Der Fokus liegt dabei auf der einheitlichen Kommunikation und einer Middleware. Die Middleware übernimmt die Übertragung der Nachrichten und ermöglicht das Verteilen der Actors, Agents und Webservices. Eine abstrakte Sprache, die Agent Communication Language, dient zur Kommunikation.

FIPA ist zunächst sprach- und implementationsunabhängig (Abbildung 3), jedoch sind Implementationen in Java und auf Basis der CORBA Architektur explizit vorgesehen [FIP02b, S. 7 1]. Da die Schnittstellen ausschließlich die ACL ist, können die verschiedenen Implementationen von FIPA miteinander kommunizieren [FIP02b, S. 4 1].

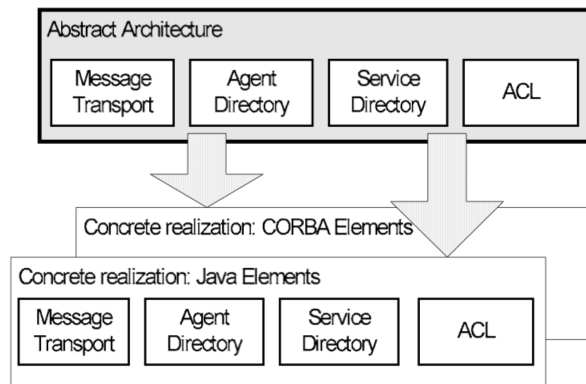

Abbildung 3: FIPA Architektur [FIP02b, S. 6 Figure 1]

Eine ACL Nachricht enthält sowohl Metainformationen wie Sender und Receiver (global eindeutige Ids), als auch den Content, den es zu Übertragen gilt (Abbildung 3). Der Content ist in der ACL nicht näher beschrieben und kann beliebig gewählt werden. Zum Serialisieren und Deserialisieren des Contents wird ein Encoding (z.B. XML) benötigt [FIP02a, S. 2].

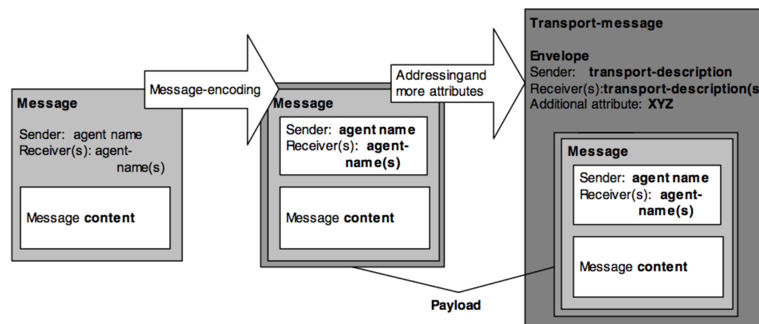

Abbildung 4: FIPA ACL [FIP02b, S. 14 Figure 7]

Agents und Services sind in Directiories registriert, dort wird der global eindeutige Name auf einen technische Transportadresse und ein Transporttyp (z.B. HTTP) abgebildet [FIP02b, S. 11]. Nachdem die ACL Message serialisiert wurde, wird sie mit einem technischen Sender und Empfänger in einer Transportmessage gekapselt.

Zur Anbindung von Fremdsystemen, bietet die Java-Implementierung Jade eine Kompatiblitätsschicht für Webservices und SOAs. Dies ist möglich durch eine beidseitige Konvertierung der ACL zur Webservice Description Language (WSDL) und zur XML-Nachricht für Service-orientend Architecture Protocol (SOAP). Der Service selbst wird in einem universellen Serviceverzeichnis (UDDI repository) beschrieben (Abbildung 5). In JADE kann jeder Agent sein Service im Directory Facilitator (DF) anmelden (ähnlich zu Blackboard Pattern), um auffindbar für andere Agents zu sein. Der Jade Gateway

stellt sicher, dass bei Verwendung des Services von außen die Agenten gestartet und die Nachrichten zugestellt werden.

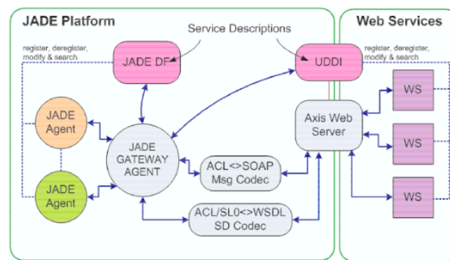

Abbildung 5: Jade Kompatibilität zu SOA und WS [GLMS07, S. 1414 Figure 1]

5.1 RDF

Das Resource Description Framework (RDF) [RDF14] ist eine ontologische Beschreibungsprache auf Basis von XML [CLS01, S. 7]. RDF wurde vom W3C ursprünglich zum Beschreiben von Inhalten im Web entwickelt. RDF definiert Syntax und Semantik ohne dabei die Domain vorzugeben.

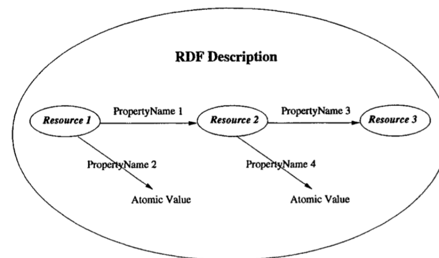

Abbildung 6: RDF Model [CLS01, S. 9 Figure 1]

Das RDF Model ist der Kern und besteht aus Resourcen, Properties und Atomic Values (Abbildung 6). Resourcen, die beispielsweise durch URIs angegeben werden, können durch Properties (z.B. Author) einen Atomic Value (z.B. "Carl Hewitt") zugewiesen bekommen. Anstatt eines Atomic Values könen auch weitere Resourcen zugewiesen werden. Daraus ergibt sich ein Ontologiebaum, der eine Resource beschreibt, genannt Statement [CLS01, S. 10]. Dieser Ontologien sind auch für das Austauschen von Informationen zwischen Agenten (wie z.B. in FIPA) verwendbar [CLS01, S. 9]. Das Encoding der ACL Message ist in diesem Fall RDF.

Es gibt diverse Implementierung wie Redland oder Jena, die neben der Implementierung des RDF Modells auch eine SQL-artige Anfragesprache bereitstellen [CLS01, S. 15].

6 State of the Art

Aktoren und Agenten sind keine neuen Konzepte, dennoch sind sie auf Grund ihrer Skalierbarkeit auf aktuellen Konferenzen vertreten. Insbesondere werden von den Keynotespeakern die Spiele im Bereich Distributed Artificial Intelligence in Verbindung mit Aktoren und Agenten gebracht:

Für den Bereich Distributed Artificial Intelligence ist die ACM Special Interest Group on Artificial Intelligence (SIGAI)[ACM14] wichtig, denn sie vereint Konferenzen über Autonomous Agents and Multiagent Systems (AAMAS), Human Robot Interaction (HRI) und Intelligent User Interfaces (IUICONF). Keynote Speaker der AAMAS'12 Jeff Rosenschein ist spezialisiert auf Agenten und Spiele an der Hebrew University of Jerusalem. Agenten lernen empirisch und machen Fehler. Jeff Rosenschein beschäftigt sich mit Mechanismen um Fehler zu erkennen [MR11]. Insbesondere beim Zusammenspiel

mehrerer Agenten ist das Finden eines Lügners, ob bewusst oder unbewusst, wichtig, um Entscheidungen stabiler zu gestalten.

Die IEEE Conference on Computational Intelligence and Games (CIG) ist eine Konferenz spezialisiert auf Spiele. Auf der CIG'12 [IEE12] ist Jeff Orkin Keynotespeaker. Er ist Spieleentwickler und spezialisiert auf Collective Artificial Intelligence am MIT Media Lab [OSB12].

Für Jeff Orkin stand die Planung von sozialen Interaktionen und menschlicher Kommunikation im Mittelpunkt [OR11]: Mitschnitte menschlicher Dialoge in Onlinestrategiespielen sind die Basis für eine formale Klassifikation, dies es Agenten erlaubt die Inhalte zu verarbeiten und zu teilen. Mit Hilfe des Hidden Markov Model wird eine statistische Analyse erstellt, sodass eine Abbildung von einer menschlichen Aussage möglich ist: Das Ergebnis (Formel 13) ist ein Tripel aus einem Spech Act (z.B. Expressive), Content (z.B. Thank) und Referent (z.B. Money).

$$thank\ you\ for\ the\ tip \rightarrow \{Expressive, Thank, Money\} \tag{13}$$

Die International Conference of Agents and Artificial Intelligence (ICCART) ist nicht auf Spiele spezialisiert sondern allgemein auf Agenten und AI. Der Keynote Speaker Pieter Spronck auf der ICAART'14 [ICA14] ist dennoch spezialisiert auf AI und Profiling in Games an der Tilburg University.

Echtzeitstrategiespiele bestehen aus Spielfiguren, die in einer Welt sich bewegen. Der Spielablauf wird durch Regeln bestimmt. Pieter Spronck beschäftigt sich mit einem Design Pattern für Echtzeitstrategiespiele, dem Resource Entity Action Pattern - kurz REA Pattern [ADGO+13]. Ähnlich zu RDF werden Resourcen (z.B. Sträke, Verteidigung oder Geschwinidigkeit) und Entities (z.B. Kobolde und Waldläufer) definiert. Darauf basiert ein Actionsgraph, der einem Entscheidungsbaum entspricht. Jede Entity hat einen Resource Vector, der angibt, welche Resourcen inkl. Werte zur Verfügung stehen. Können Kobolde und Waldläufer gegeneinander kämpfen, besteht eine Kante zwischen den Knoten Kobold und Waldläufer im Actiongraph. Eine Kante entspricht einer vorhanden Transformationsmatrix (gebunden an ihre Resourcen). In folgenden Beispiel greift der Kobold mit den Resourcen Angriffsschaden und Lebenspunkte den Waldläufer an. Der Waldläufer hat die gleichen Resourcen mit anderen Werten. A ist die Transformationsmatrix (Formel 14). Es werden die Resourcen das Waldläufers nach dem Angriff berechnet (Formel 15): Der Waldläufer verliert 15 Lebenspunkte.

$$A = \begin{pmatrix} 0 & -1 \\ 0 & 0 \end{pmatrix} \tag{14}$$

$$r'_w = r_w + r_k * A \tag{15}$$

$$r'_w = \begin{pmatrix} 20 \\ 500 \end{pmatrix} + \begin{pmatrix} 15 \\ 1000 \end{pmatrix} * A = \begin{pmatrix} 20 \\ 485 \end{pmatrix}$$

Durch das REA Pattern lässt sich ein Spieldesign erstellen, ohne das explizit Programmierkenntnisse vorhanden sein müssen, denn die Logik basiert ausschließlich auf der Definition von Action, Entities und der Transformationsmatrizen.

7 Problemstellung und Fazit

Echzeitstrategiespiele, wie das Open Source Spiel 0AD, benötigen hohe Performanz, um ihrem Echzeitanspruch gerecht zu werden. Außerdem benötigen sie ein gute Skalierbarkeit, um gleichzeitig eine hohe Anzahl an Spielfiguren auf dem Feld interagieren zu lassen. Diese werden sowohl autonom als auch manuell durch den Spieler gesteuert. Die Kombination aus funktionalem und objektorientiertem Programmierparadigma und einer Architektur, die sowohl Aktoren, Agenten und Fremdsysteme verbindet, ist in der Lage eine konkrete Umgebung, wie Echzeitstrategiespiele, skalierbar zu gestalten. Die Integration in vorhandene Spiele ist dabei eine große Herausforderung und gleichzeitig ein Risiko, da deren Architektur aufgebrochen und der schon vorhandene Shared State integriert werden muss. An Hand dieser Integration in einer konkreten Umgebung lässt sich die Kohärenz der Programmierstile und der Werkzeuge der Sprachen, wie Schnittstellen und Ausdrucksmöglichkeiten, analysieren, um das Zusammenspiel der Programmierparadigmen bewerten zu können.

Literatur

[0AD14] *0AD - A free, open-source game of ancient warfare.* Online Abruf (02.03.2014) play0ad.com : Website, 2014

[ACM14] ACM: *Special Interest Group on Artificial Intelligence.* Online Abruf (02.03.2014) sigai.acm.org : Website, 2014

[ADGO+13] Abbadi, Mohamed ; Di Giacomo, Francesco ; Orsini, Renzo ; Plaat, Aske ; Spronck, Pieter ; Maggiore, Giuseppe: Resource Entity Action: A Generalized Design Pattern for RTS games. (2013)

[Akk14] *Akka.* Website, 2014. – Online Abruf (02.03.2014) www.akka.io

[BAD+09] Bocchino, Robert L. Jr. ; Adve, Vikram S. ; Dig, Danny ; Adve, Sarita V. ; Heumann, Stephen ; Komuravelli, Rakesh ; Overbey, Jeffrey ; Simmons, Patrick ; Sung, Hyojin ; Vakilian, Mohsen: A Type and Effect System for Deterministic Parallel Java. In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications.* New York, NY, USA : ACM, 2009 (OOPSLA '09). – ISBN 978–1–60558–766–0, 97–116

[BESP08] Bacon, Jean ; Eyers, David M. ; Singh, Jatinder ; Pietzuch, Peter R.: Access Control in Publish/Subscribe Systems. In: *Proceedings of the Second International Conference on Distributed Event-based Systems.* New York, NY, USA : ACM, 2008 (DEBS '08). – ISBN 978–1–60558–090–6, 23–34

[CJ10] Cheung, Alex King Y. ; Jacobsen, Hans-Arno: Load Balancing Content-Based Publish/Subscribe Systems. In: *ACM Trans. Comput. Syst.* 28 (2010), Dezember, Nr. 4, 9:1–9:55. `http://dx.doi.org/10.1145/1880018.1880020`. – DOI 10.1145/1880018.1880020. – ISSN 0734–2071

[CLS01] Candan, K. S. ; Liu, Huan ; Suvarna, Reshma: Resource Description Framework: Metadata and Its Applications. In: *SIGKDD Explor. Newsl.* 3 (2001), Juli, Nr. 1, 6–19. `http://dx.doi.org/10.1145/507533.507536`. – DOI 10.1145/507533.507536. – ISSN 1931–0145

[DKVCD12] De Koster, Joeri ; Van Cutsem, Tom ; D'Hondt, Theo: Domains: Safe Sharing Among Actors. In: *Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions.* New York, NY, USA : ACM, 2012 (AGERE! '12). – ISBN 978–1–4503–1630–9, 11–22

[Erl14] *Erlang.* Website, 2014. – Online Abruf (02.03.2014) www.erlang.org

[FCFB05] Fiorentino, C. ; Cilia, M. ; Fiege, L. ; Buchmann, A.: Building a Configurable Publish/Subscribe Notification Service. In: *Proceedings of the 5th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems.* Berlin, Heidelberg : Springer-Verlag, 2005 (DAIS'05). – ISBN 3–540–26262–8, 978–3–540–26262–6, 136–147

[FF06] Fernández, Maribel ; Fleutot, Fabien: A Historic Functional and Object-oriented Calculus. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming.* New York, NY, USA : ACM, 2006 (PPDP '06). – ISBN 1–59593–388–3, 145–156

[FIP02a] FIPA: *ACL Message Structure Specification*, 2002. `www.fipa.org/specs/fipa00061/SC00061G.pdf`. – Online; Abruf am 17.11.2013

[FIP02b] FIPA: *FIPA Abstract Architecture Specification*, 2002. `www.fipa.org/specs/fipa00001/SC00001L.pdf`. – Online; Abruf am 17.11.2013

[GGS11] Gaha, Mohamed ; Gagnon, Michel ; Sirois, Frederic: A Modern Blackboard System. In: *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 03*. Washington, DC, USA : IEEE Computer Society, 2011 (WI-IAT '11). – ISBN 978–0–7695–4513–4, 163–166

[GLMS07] Greenwood, Dominic ; Lyell, Margaret ; Mallya, Ashok ; Suguri, Hiroki: The IEEE FIPA Approach to Integrating Software Agents and Web Services. In: *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*. New York, NY, USA : ACM, 2007 (AAMAS '07). – ISBN 978–81–904262–7–5, 276:1–276:7

[GWK12] Gehlot, Vijay ; Way, Thomas ; Klassner, Frank: Coexistence of Functional and Object-oriented Paradigms. In: *J. Comput. Sci. Coll.* 27 (2012), Januar, Nr. 3, 122–129. http://dl.acm.org/citation.cfm?id=2038772.2038798. – ISSN 1937–4771

[HBS73] Hewitt, Carl ; Bishop, Peter ; Steiger, Richard: A universal modular ACTOR formalism for artificial intelligence. In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1973 (IJCAI'73), 235–245

[Huh09] Huhns, Michael N.: From DPS to MAS to ...: Continuing the Trends. In: *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*. Richland, SC : International Foundation for Autonomous Agents and Multiagent Systems, 2009 (AAMAS '09). – ISBN 978–0–9817381–6–1, 43–48

[ICA14] *International Conference of Agents and Artificial Intelligence*. Online Abruf (02.03.2014) www.icaart.org : Website, 2014

[IEE12] IEEE: *Conference on Computational Intelligence and Games*. Online Abruf (02.03.2014) geneura.ugr.es/cig2012 : Website, 2012

[IPA14] Intelligent Physical Agents, Foundation of: *FIPA*. Online Abruf (02.03.2014) www.fipa.org : Website, 2014

[lib14] *libcppa*. Website, 2014. – Online Abruf (02.03.2014) libcppa.blogspot.com

[McM03] McManus, John W.: Design and Analysis Tools for Concurrent Blackboard Systems. NASA Langley Technical Report Server, 2003. – Forschungsbericht

[MD10] Marr, Stefan ; D'Hondt, Theo: Many-core Virtual Machines: Decoupling Abstract from Concrete Concurrency. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. New York, NY, USA : ACM, 2010 (SPLASH '10). – ISBN 978–1–4503–0240–1, 239–240

[MR11] Meir, Reshef ; Rosenschein, Jeffrey S.: Strategyproof Classification. In: *SIGecom Exch.* 10 (2011), Dezember, Nr. 3, 21–25. http://dx.doi.org/10.1145/2325702.2325708. – DOI 10.1145/2325702.2325708. – ISSN 1551–9031

[Nie88] Nierstrasz, O. M.: Two Models of Concurrent Objects. In: *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-based Concurrent Programming*. New York, NY, USA : ACM, 1988 (OOPSLA/ECOOP '88). – ISBN 0–89791–304–3, 174–176

[NM92] Netzer, Robert H. B. ; Miller, Barton P.: What Are Race Conditions?: Some Issues and Formalizations. In: *ACM Lett. Program. Lang. Syst.* 1 (1992), März, Nr. 1, 74–88. http://dx.doi.org/10.1145/130616.130623. – DOI 10.1145/130616.130623. – ISSN 1057–4514

[OAF08] Ortega-Arjona, Jorge L. ; Fernandez, Eduardo B.: The Secure Blackboard Pattern. In: *Proceedings of the 15th Conference on Pattern Languages of Programs*. New York, NY, USA : ACM, 2008 (PLoP '08). – ISBN 978–1–60558–151–4, 22:1–22:5

[OR11] Orkin, Jeff ; Roy, Deb: Agents for Games and Simulations II. Version: 2011. `http://dl.acm.org/citation.cfm?id=1985721.1985735`. Berlin, Heidelberg : Springer-Verlag, 2011. – ISBN 978–3–642–18180–1, Kapitel Semi-automated Dialogue Act Classification for Situated Social Agents in Games, 148–162

[OSB12] Orkin, Jeff ; Smith, Gillian ; Bowling, Michael: Keynotes [abstracts of three keynote presentations]. In: *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 2012, S. E–1–E–1

[PH06] Pepper, Peter ; Hofstedt, Petra: *Funktionale Programmierung: Sprachdesign und Programmiertechnik (German Edition)*. Springer, 2006. – ISBN 978–3540209591. – ISBN 978-3-54-0209591

[RDF14] *RDF - Resource Description Framework*. Online Abruf (02.03.2014) www.w3.org/RDF : Website, 2014

[RY13] Ricci, Alessandro ; Yonezawa, Akinori: Away from the Sequential Paradigm Tarpit: Modelling and Programming with Actors, Concurrent Objects and Agents. In: *Proceedings of the Second International Workshop on Combined Object-Oriented Modelling and Programming Languages*. New York, NY, USA : ACM, 2013 (ECOOP'13). – ISBN 978–1–4503–2039–9, 1:1–1:6

[SMK+01] Stoica, Ion ; Morris, Robert ; Karger, David ; Kaashoek, M. F. ; Balakrishnan, Hari: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. New York, NY, USA : ACM, 2001 (SIGCOMM '01). – ISBN 1–58113–411–8, 149–160

[Typ13] TypeSafe: *Akka Documentation 2.1.1*. Online Abruf (04.03.2013) doc.akka.io/docs/akka/2.1.1/Akka.pdf, 2013

[Woo02] Wooldridge, Michael: Intelligent Agents: The Key Concepts. In: *Proceedings of the 9th ECCAI-ACAI/EASSS 2001, AEMAS 2001, HoloMAS 2001 on Multi-Agent-Systems and Applications II-Selected Revised Papers*. London, UK, UK : Springer-Verlag, 2002. – ISBN 3–540–43377–5, 3–43

[ZD10] Zhou, Jin ; Demsky, Brian: Bamboo: A Data-centric, Object-oriented Approach to Many-core Software. In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2010 (PLDI '10). – ISBN 978–1–4503–0019–3, 388–399