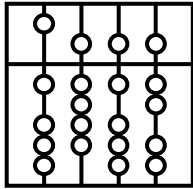
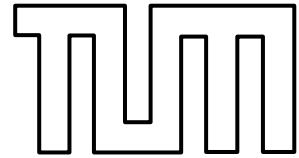


Kontextsensitive Informationsfilter

Christof Söhngen



FAKULTÄT FÜR INFORMATIK
TECHNISCHE UNIVERSITÄT MÜNCHEN



Diplomarbeit

Kontextsensitive Informationsfilter

Bearbeiter:	Christof Söhngen
Aufgabensteller:	Prof. Gudrun Klinker, Ph. D. Prof. Dr. Donald Kossmann
Betreuer:	Prof. Gudrun Klinker, Ph. D. Prof. Dr. Donald Kossmann
Abgabedatum:	15. Februar 2003

Ich versichere, dass ich diese Diplomarbeit
selbständig verfasst und nur die angegebenen
Quellen und Hilfsmittel verwendet habe.

München, den 15. Februar 2003

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Aufgabenstellung	1
2	Modellierung	3
2.1	Anforderungsanalyse	3
2.2	Modell des Aktiven Campus	5
2.2.1	Infrastruktur	5
2.2.2	Teilnehmer	5
2.2.3	Kontext	6
2.2.4	Nachrichtensystem	6
2.3	Modell der Infrastruktur	7
2.4	Modell der Teilnehmer	8
2.5	Modell des Nachrichtensystems	9
2.6	Zentrale Abläufe im System	10
2.6.1	Nachrichtenerstellung	10
2.6.2	Verbleib der Nachricht im System	10
2.6.3	Sendung der Nachricht	11
2.6.4	Löschen einer Nachricht	13
2.7	Modell des Kontextes	14
2.7.1	Funktionen	14
2.7.2	Attribute	15
2.7.3	Hierarchie	15
2.7.4	Grundstruktur des Kontexts	16
2.7.5	Weitere Aufgaben und Funktionen des Kontextes	19
2.8	Modell einer Nachricht	20
2.9	Modell eines Filters	23
2.10	Szenarien (siehe Anhang)	27

3	Implementierung	28
3.1	Anforderungen	28
3.2	Heuristik für die Implementierung der Auswertung	30
3.2.1	Sensormeldung	32
3.2.2	Data Cooking	32
3.2.3	Change Detection	33
3.2.4	Matching	33
3.2.5	Nachrichtenversand	35
3.3	Algorithmus für die Auswertung im Nachrichtensystem	37
3.3.1	Trivialer Ansatz	37
3.3.2	Grundlegende Optimierungen	37
3.3.3	Beispiel für den Auswertungsalgorithmus	40
3.3.4	Pseudo-Code für den Auswertungsalgorithmus	42
3.3.4.1	Methode <code>initialisierung()</code>	42
3.3.4.2	Methode <code>update()</code>	45
3.3.4.3	Methode <code>auswerten()</code>	46
3.3.5	Einschätzung des Algorithmus	46
3.3.6	Weiterführende Optimierungen	48
3.4	Ansatz zur Verwaltung des Kontextes	50
4	Zusammenfassung	51
4.1	Rückblick	51
4.2	Verbesserungsmöglichkeiten	51
4.3	Ausblick	53
5	Literaturverzeichnis	54
6	Anhang A: Szenarien	55
7	Anhang B: Index	75

Abbildungsverzeichnis

Abb. 1:	Modell des Aktiven Campus (UML Klassendiagramm)	5
Abb. 2:	Modell der Teilnehmerstruktur (UML Klassendiagramm)	8
Abb. 3:	Modell der Aktionen im Nachrichtensystem (UML Klassendiagramm)	9
Abb. 4:	Modell der Vorlagen (UML Klassendiagramm)	11
Abb. 5:	Ablauf von Nachrichtenerstellung, -sendung und -löschung (UML Sequenzdiagramm)	12
Abb. 6:	Modell der Kontextstruktur (UML Klassendiagramm)	14
Abb. 7:	Kontext-Beispiel (stark vereinfacht)	19
Abb. 8:	Implementierung des Aktiven Campus-Modells	28
Abb. 9:	Kern-Funktionalität des Nachrichtensystems (UML Statechart Diagramm)	30
Abb. 10:	Beispiel einer Interval Skip List	35
Abb. 11:	Beispiel für den Auswertungsalgorithmus	41

1. Einleitung

1.1 Motivation

Der Begriff „Aktiver Campus“ beschreibt das Modell einer Universität, die ihren Teilnehmern aus dem Lehr- und Forschungsbetrieb eine möglichst große Zahl von Informationen zur Verfügung stellt. Sowohl die Infrastruktur, die zur Verfügung stehende Hard- und Software als auch die bereitgestellten Dienste bilden die Grundlage für diese aktive Wissensbank. Die Teilnehmer können mit eigenen und öffentlichen Systemen am Aktiven Campus teilnehmen und miteinander bzw. mit den Daten-Verarbeitenden Systemen kommunizieren.

Um eine möglichst umfassende Information zu gewährleisten, werden alle Aspekte des Universitätsgeschehens in den Aktiven Campus einbezogen, d.h. sie bilden den so genannten Kontext. Diese Informationen können aus drei verschiedenen Quellen stammen:

- andere Teilnehmer (z.B. Kommunikation zwischen Kommilitonen)
- Anbieter von Diensten (z.B. Betreiber der Mensa, Studentenfachschaften, etc.)
- automatisch vom System erfasste Informationen (z.B. Anwesenheit anderer Personen, Wetter, etc.)

Typische Szenarien beinhalten hauptsächlich die kontextabhängige Benachrichtigung von Benutzern. Beispiele sind:

- Alle Teilnehmer einer Vorlesung werden über deren Ausfall informiert
- Ein Benutzer wird über die Anwesenheit seiner Freunde in der Mensa benachrichtigt
- Allen möglicherweise Interessierten wird eine Einladung zu einer Rede eines Gastprofessors zugestellt.

1.2 Aufgabenstellung

Um mit dieser Menge an Daten umzugehen, werden Techniken benötigt, die auf der einen Seite die Daten im System verarbeiten, aber auch die Interaktion und das Verarbeiten durch den Teilnehmer ermöglichen. Der Datenschutz, also der einschränkbare Zugriff auf nicht-öffentliche Informationen muss ebenfalls gewährleistet werden.

Die Grundidee, um den Teilnehmer die Arbeit mit all diesen Informationen zu ermöglichen besteht aus einem automatischen Filter, der die für den Teilnehmer relevanten Daten aussortiert. Diese Relevanz ergibt sich einerseits aus Sicht des Teilnehmers, andererseits aus Sicht der Dienste, die neue, möglicherweise für den Teilnehmer unbekannte Informationen anbieten.

Die Automatik sorgt dabei allerdings nur für die Auswertung, die Definition der Relevanz muss von Teilnehmern des Systems vorgenommen werden.

Im Folgenden soll ein Modell entwickelt werden, das Lösungen für die oben angesprochenen Probleme bereitstellt. Des Weiteren soll der Grundstein für eine Implementierung gelegt werden, in dem Ansätze für eine Umsetzung in ein reales System erarbeitet werden.

2. Modellierung

2.1 Anforderungsanalyse

Ziel der Arbeit ist der Entwurf eines Systems, das die oben genannte Arbeitsweise des Aktiven Campus ermöglicht. Es muss folgenden Anforderungen genügen:

- Das System unterstützt alle Teilnehmer des Universitätsbetriebes sowie deren Gäste. (Bsp.: Professoren, wissenschaftliche Mitarbeiter, Studenten, Verwaltung, ...)
- Jede (digital) verfügbare Information kann im System gespeichert werden. (Bsp.: Informationen über Personen, deren Aktivitäten, den Lehr- und Forschungsbetrieb, physikalische Informationen wie Wetter, etc.)
- Jeder Teilnehmer kann überall, unabhängig vom verwendeten Gerät am Aktiven Campus teilnehmen (Idee des *Ubiquitous Computing*) (Bsp.: Computer in der Rechnerhalle, Laptop, PDA, ...)
- Informationen und Dienstleistungen werden dem Teilnehmer mit Hilfe von Nachrichten übermittelt. Diese Nachrichten können zwischengespeichert werden, falls zum Zeitpunkt der Sendung kein Empfänger (Gerät) verfügbar ist.
- Die Relevanz einer Nachricht kann bzw. muss vom Ersteller der Nachricht (der auch der Empfänger selbst sein kann) manuell definiert werden. Das System wertet alle Nachrichten aus und stellt sie zu, wenn alle Bedingungen der Nachricht erfüllt sind. (Bsp.: Ein Mensa-Mitarbeiter entscheidet, unter welchen Umständen der aktuelle Speiseplan an einen Teilnehmer geschickt werden soll. Als Alternative kann sich auch jeder Teilnehmer selbst mit einer Nachricht informieren lassen, wenn er über den Speiseplan informiert werden will.)
- Jeder Teilnehmer hat die umfassende Möglichkeit, durch Filter zu entscheiden, welche Nachrichten er empfangen will und welche nicht. (Bsp.: Ein Teilnehmer kann alle fremden Nachrichten, die mit der Mensa zu tun haben, ablehnen.)
- Das System, mit dessen Hilfe die Nachrichten verwaltet werden kann moderiert werden, um Missbrauch zu verhindern und einfache Bedienung zu gewährleisten.

(Bsp.: Ein Angestellter der Universität kann als Moderator gewisse Nachrichten löschen, die nicht mit den Richtlinien der Universität konform sind. Um Benutzern den Umgang mit dem System zu erleichtern, kann er Vorlagen entwerfen, die von den Teilnehmern für Nachrichten oder Filter benutzt werden können.)

- Jeder Teilnehmer kann die Informationen, die im System gespeichert werden (Kontext) selbst beeinflussen und zur eigenen Datenspeicherung bzw. Nachrichtenauslösung benutzen.

(Bsp.: Ein Student speichert seinen privaten Stundenplan und lässt sich an Veranstaltungen erinnern.)

- Die Struktur der Daten, die im System gespeichert werden und zur Ermittlung der Relevanz von Informationen führen (Kontext) ist variabel und kann jederzeit (ggf. automatisch) an die tatsächlichen Bedürfnisse auf dem Aktiven Campus angepasst werden.

(Bsp.: Falls neue Sensoren zur Verfügung stehen, die bisher nicht verfügbare Daten liefern können, können deren Daten in den Kontext aufgenommen und zur Relevanzbewertung eingesetzt werden.)

- Der Sender einer Nachricht hat die Möglichkeit, Quittungen der erfolgten Sendungen zu erhalten.

(Bsp.: Ein Student möchte mitgeteilt bekommen, welche seiner Kommilitonen seine Nachricht über eine Terminänderung erreicht hat.)

- Der Nachrichten-Inhalt selbst kann dynamisch sein, und auf Informationen auf verteilten Quellen verweisen.

(Bsp.: Eine Nachricht, die unter bestimmten Bedingungen an Teilnehmer verschickt wird, um den aktuellen Mensa-Speiseplan vorzustellen, muss nicht jeden Tag neu erstellt werden, sondern trägt einen dynamischen Inhalt, sodass der jeweils aktuelle Plan eingefügt wird.)

2.2 Modell des Aktiven Campus

2.2.1 Infrastruktur

Die Basis für die gewünschte Funktionalität des Aktive Campus bildet die *Infrastruktur*. Sie verbindet das zentrale Nachrichtensystem mit den Geräten der Teilnehmer und mit dem Kontext. Am Beispiel der Technischen Universität München in Garching ist der benötigte Ausbau bereits zu großen Teilen vorhanden. Mit Hilfe des internen Netzwerkes (sowohl drahtgebunden als auch über Funk) ist die Anbindung von Teilnehmern, Servern und Sensoren sichergestellt.

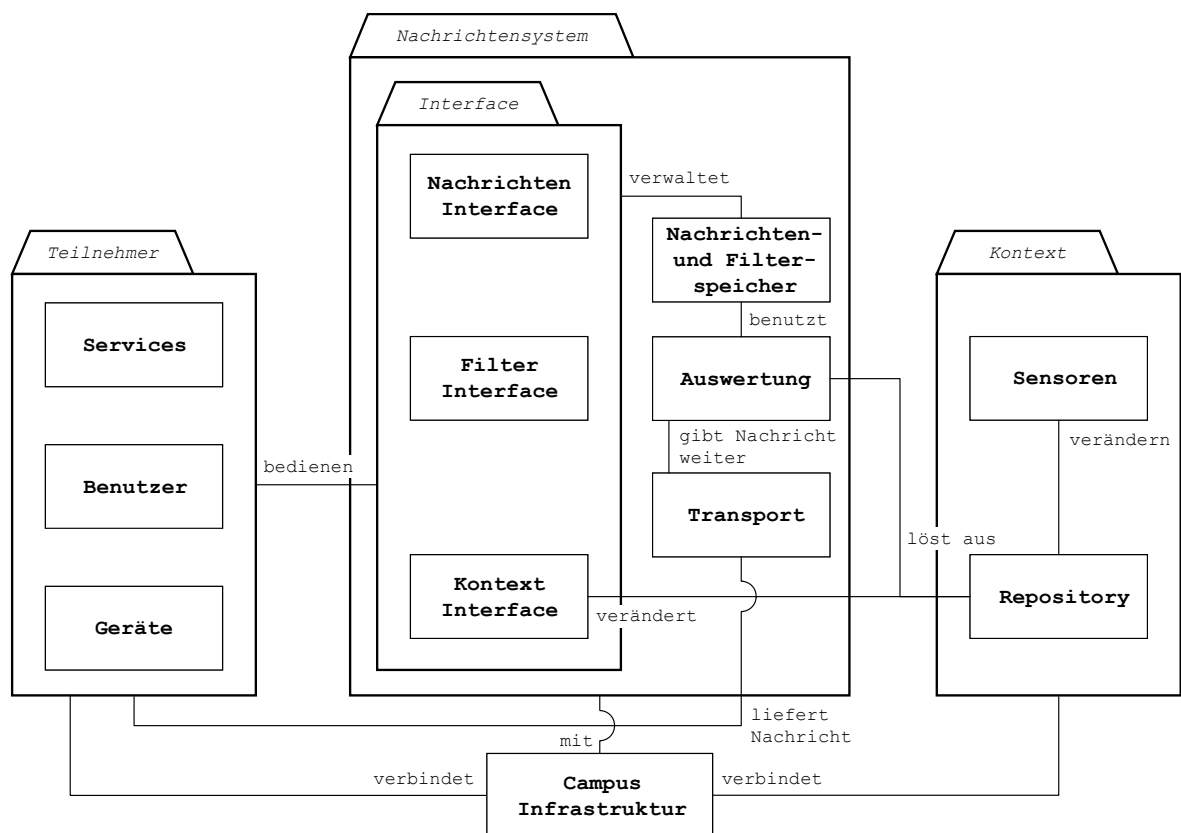


Abbildung 1: Modell des Aktiven Campus (UML Klassendiagramm)

2.2.2 Teilnehmer

Die Teilnehmer des Aktiven Campus, müssen in der Lage sein, eine Verbindung über die vorhandene Infrastruktur zum Nachrichtensystem aufzubauen. Dazu sind, neben der bereits vorhandenen stationären EDV-Ausrüstung, vor allem mobile Geräte nötig, die in der Lage

sind, mit der Infrastruktur zu kommunizieren. Dies könnten zum Beispiel Laptops mit WaveLan Adaptern oder PDAs mit Bluetooth Modul sein.

Die Teilnehmer sollten in der Lage sein, neben Komponenten, die vom Betreiber des aktiven Campus zur Verfügung gestellt werden auch eigene Geräte zu verwenden. Neben den ständigen Teilnehmern wie Professoren, wissenschaftlichen Mitarbeitern, Angestellten und Studenten sollen auch Gäste in der Lage sein, den Aktiven Campus zu nutzen. Zu den Teilnehmern gehören auch Dienste, also Teilnehmer, die zum großen Anteil nur Informationen anbieten.

Neben der Ausstattung der Teilnehmer mit Hardware muss weiterhin sichergestellt sein, dass Software-Lösungen bzw. Protokolle vorhanden sind, sodass mit möglichst vielen - wahrscheinlich in der Ausstattung sehr stark variierenden - Geräten auf die Angebote des aktiven Campus zugegriffen werden kann.

2.2.3 Kontext

Eine weitere Voraussetzung ist das Vorhandensein von *Sensoren*, derart, dass die gewünschten Kontextinformationen bereitgestellt werden können. Hierzu zählt vor allem ein System, das es erlaubt, die Position der Benutzer zu erkennen. Die Daten der Sensoren müssen im *Kontext-Repository* persistent gespeichert werden können, um z.B. später auch Rückschlüsse aus vergangenen Daten zu erlauben.

2.2.4 Nachrichtensystem

Das Nachrichtensystem verbindet die Teilnehmer untereinander und gestattet Zugriff auf die vorhandenen Kontextinformationen. Über ein Interface kann ein Teilnehmer Nachrichten, Filter und den Kontext erstellen, einsehen und verändern. Die Daten der Benutzer werden im System gespeichert.

Eine Nachricht stellt eine Anweisung dar, unter welchen Umständen Information an Teilnehmer geschickt werden soll. Ein Filter symbolisiert die Möglichkeit eines Teilnehmers, auf Nachrichten anderer Teilnehmer, die an ihn gerichtet sind oder sich auf Elemente seines (privaten) Kontextes beziehen, einzuwirken.

Ein Beispiel: Ein Teilnehmer wünscht, benachrichtigt zu werden, falls er den Aktiven Campus betritt und bereits ein Kommilitone anwesend ist. Ein Kommilitone hat einen Filter eingerichtet, der seine Position dem Teilnehmer nicht zur Verfügung stellt, um seine Privatsphäre zu wahren.

Die Relevanz einer Nachricht für einen Teilnehmer ergibt sich aus der Kombination der Anweisungen in Nachrichten und Filtern, jeweils angewendet auf die aktuellen Kontextinformationen.

Eine Auswertung erzeugt aus den veränderten Informationen, die ständig vom Kontext geliefert werden, ggf. diese Nachrichten, in denen die Information zu den Teilnehmern transportiert werden kann, für die sie relevant ist.

2.3 Modell der Infrastruktur

Da die Infrastruktur an der Technischen Universität München bereits zu großen Teilen implementiert ist, wird in dieser Arbeit auf eine Modellierung verzichtet.

2.4 Modell der Teilnehmer

Das Nachrichtensystem vermittelt Nachrichten zwischen den Teilnehmern, die sowohl Benutzer, Dienste und Moderatoren als auch Geräte sein können.

- Ein Benutzer ist eine einzelne Person, die auf dem aktiven Campus arbeitet, studiert oder in einer sonstigen Weise daran beteiligt ist. Beispiel: Studenten, Professoren, Wissenschaftliche Mitarbeiter, ...
- Ein Dienst ist ein Oberbegriff für (beispielsweise offizielle) Stellen, die allen Teilnehmern (oder Teilmengen daraus) Informationen und Dienstleistungen zur Verfügung stellen. Beispiel: Mensa, Bibliothek, Fachschaft, ...
- Ein Moderator ist eine offiziell beauftragte Person, die berechtigt ist, das Nachrichtensystem zu verändern, d.h. Nachrichten zu moderieren, Funktionen einzustellen, etc. Beispiel: Systemadministrator

Jeder Teilnehmer hat Zugriff auf verschiedene (nicht notwendigerweise mobile) Geräte, die zur Nachrichtensendung / -empfang benutzt werden. Um den Nachrichtenversand soweit wie möglich zu abstrahieren, kann ein Teilnehmer auch ein Gerät sein (Beispiel: Ein Drucker kann sowohl als Adressat einer Nachricht auftreten als auch als Ausgabegerät für die Nachricht, die ein Teilnehmer empfangen hat).

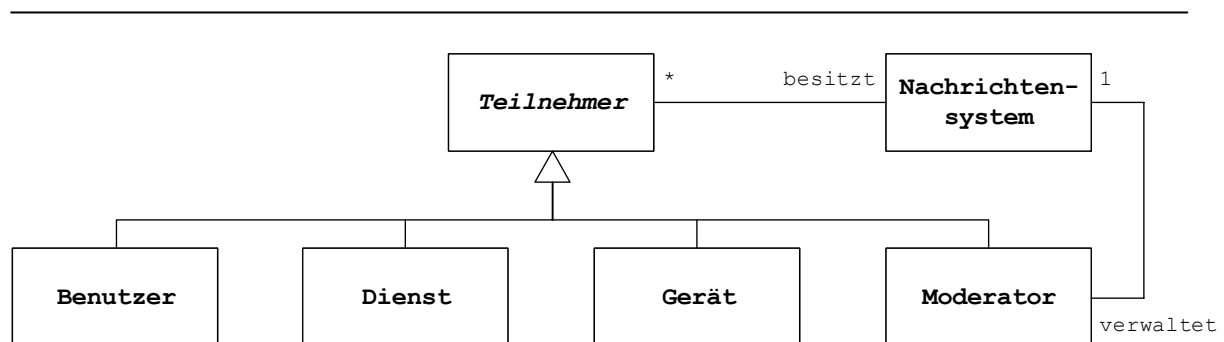


Abbildung 2: Modell der Teilnehmerstruktur (UML Klassendiagramm)

Von einer Analyse und Modellierung der für die Teilnehmer nötigen Hardware beziehungsweise Software und wird in dieser Arbeit abgesehen.

2.5 Modell des Nachrichtensystems

Zentraler Mittelpunkt des Aktiven Campus ist das Nachrichtensystem. Der Informationsaustausch (d.h. Informationsangebot und -abruf) innerhalb des Aktiven Campus findet komplett über dieses Nachrichtensystem statt. Das System unterstützt sowohl traditionelle, asynchrone Kommunikation (ähnlich der E-Mail), als auch Ereignisgesteuerte Benachrichtigung beim Eintritt von bestimmten Bedingungen. Das Relevant-Werden von Informationen löst eine Nachricht, die diese Informationen enthält, an einen oder mehrere Teilnehmer aus.

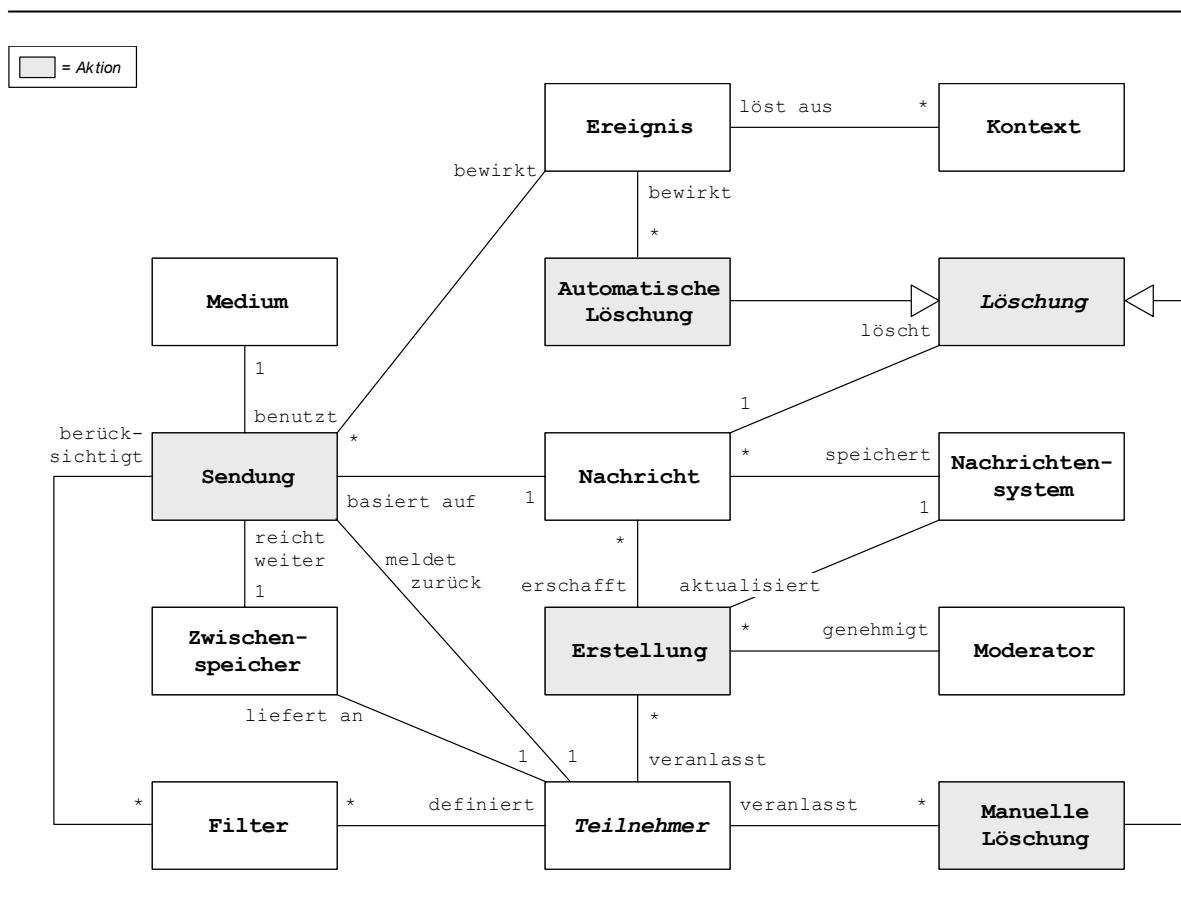


Abbildung 3: Modell der Aktionen im Nachrichtensystem (UML Klassendiagramm)

Vergleicht man das Nachrichtensystem des Aktiven Campus mit dem klassischen Ansatz für Expertensysteme, so stellen die im System gespeicherten Nachrichten und Filter die Regeln einer logischen Programmiersprache dar, das Nachrichtensystem selbst tritt als Interpreter auf.

2.6 Zentrale Abläufe im System

Die zentralen Aktionen im System sind die Erstellung und der Versand von Nachrichten, wie in Abbildung 3 beschrieben.

2.6.1 Nachrichtenerstellung

Eine Nachricht (Siehe auch 2.8 für eine detailliert Erläuterung) ist die Beschreibung eines Kommunikations- bzw. Informationswunsches eines Teilnehmers. Dieser Wunsch kann sowohl passiver (Teilnehmer möchte informiert werden) als auch aktiver Natur (Teilnehmer möchte andere Teilnehmer informieren) sein. Die Beschreibung wird im System hinterlegt, und von diesem automatisch ausgewertet. Sobald die beschriebenen Bedingungen eintreffen, wird die in der Nachricht enthaltene Information an alle Teilnehmer verschickt, für die die Information relevant ist. Die Relevanz ergibt sich dabei aus der Definition der Nachricht.

Jeder Teilnehmer kann Nachrichten erstellen. Diese Nachrichten werden im System gespeichert, falls die Erstellung von einem Moderator genehmigt wird. Dies ist der erste (triviale) Schritt zur Verhinderung unerwünschter Nachrichten im System (Siehe Abbildung 5 - Abschnitt a).

Neue Nachrichten können auf einer Vorlage basieren (sofern diese von einem Moderator genehmigt wurde). Durch diese Vorlagen soll gewährleistet werden, dass auch unerfahrene Teilnehmer die volle Funktionalität des Systems nutzen können (Siehe Abbildung 5).

2.6.2 Verbleib der Nachricht im System

Die Nachricht, genauer deren Beschreibung oder Prototyp, verbleibt solange ohne Aktion im System, bis ein Ereignis eintritt, sodass eine Bedingung erfüllt wird, die der Ersteller vorher in der Nachricht angegeben hat. Diese Ereignisse werden durch einen Kontextwechsel, also eine Veränderung des Systemzustands ausgelöst. Eine Veränderung kann sowohl lokal (Beispiel: Teilnehmer bewegt sich) als auch global (Beispiel: Temperatur steigt) auftreten (Siehe Abbildung 5 - Abschnitt b).

Der Informations-Inhalt der Nachrichten muss nicht zwangsläufig statisch sein, so dass es möglich ist, dass eine Nachricht - trotz wiederholter Sendung - lange Zeit im System

verbleibt, ohne dass sich an Ihrer Beschreibung etwas ändert. Ein Beispiel wäre das Informationsangebot der Mensa, die auf den aktuellen Speiseplan hinweist, falls ein Teilnehmer sich der Mensa nähert. Die Beschreibung der Nachricht, also hauptsächlich wann bzw. für wen sie ausgelöst werden soll, kann unverändert im System verbleiben, wohingegen die Information lediglich als Verweis gespeichert wird, sodass beispielsweise immer der tagesaktuellen Speiseplan in die Nachricht eingebettet wird, falls sie versandt wird.

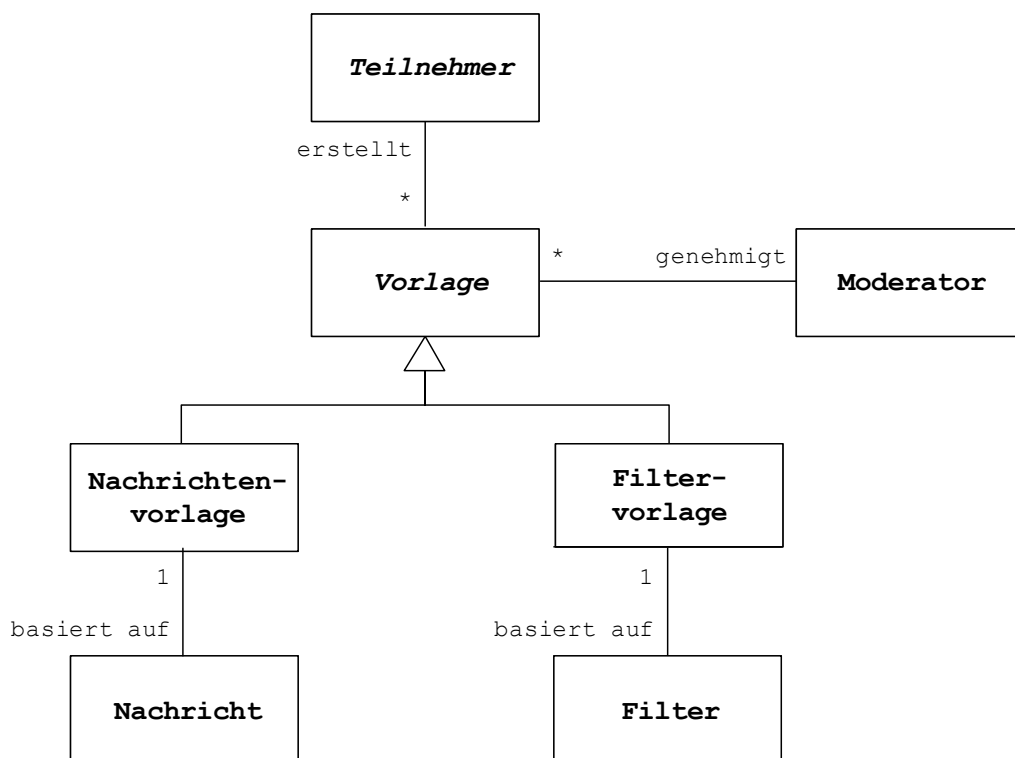


Abbildung 4: Modell der Vorlagen (UML Klassendiagramm)

2.6.3 Sendung der Nachricht

Durch ein Ereignis ausgelöst, wird eine Instanz der Nachricht auf das Senden an den Adressaten (entweder der Teilnehmer, der am Auslösen des Ereignisses beteiligt war, der Ersteller der Nachricht oder ein anderer gewünschter Teilnehmer) vorbereitet. Bevor die Nachricht über ein passendes Medium zum Empfänger gelangt, werden auf die Nachricht (evtl. bei eingefrorenem Kontext, um eine Null-Zeit Verarbeitung anzunähern) alle passenden Filter (Siehe 2.9 für eine detaillierte Erläuterung) angewandt, die der Adressat

vorher eventuell definiert hat (Siehe Abbildung 5 - Abschnitt b). Dies stellt die zweite Maßnahme zur Verhinderung eines Missbrauchs des Systems dar. Der Filter überprüft einerseits, ob die Nachricht gesendet werden kann bzw. soll, und ermöglicht andererseits auch eine direkte Beeinflussung der Attribute einer Nachricht, also z.B. im Bezug auf Aufdringlichkeit (Akustisches oder optisches Signal, ...) oder Übermittlung der Nachricht (Wahl des Mediums oder Empfänger-Gerätes). Wenn der Filter es erlaubt, wird die Nachricht dann – möglicherweise entsprechend abgeändert – über das gewählte Medium an den Teilnehmer gesendet (Siehe Abbildung 5 - Abschnitt c). Falls dieser Teilnehmer ein Gerät ist, so kann die Nachricht direkt an ihn gesendet werden, handelt es sich beim Teilnehmer um einen Benutzer, Dienst oder Moderator, so muss, z.B. in einem Filter definiert werden, an welches Gerät die Nachricht gesendet (in Kopie oder als Weiterleitung) werden soll, damit eine Person sie wahrnehmen kann. Wenn vom Ersteller gewünscht, wird anschließend eine Rückmeldung an ihn gesendet (Siehe Abbildung 5 - Abschnitt d).

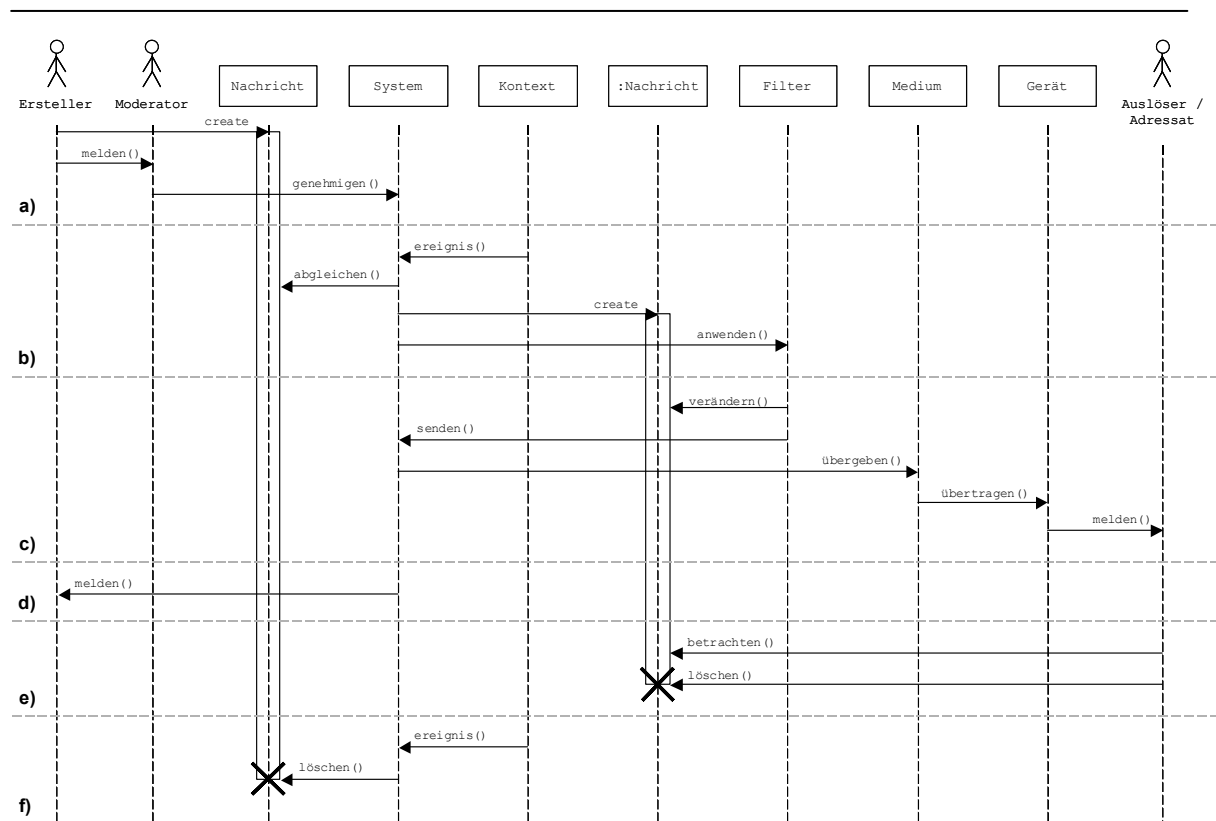


Abbildung 5: Ablauf von Nachrichtenerstellung, -sendung und -löschung (UML Sequenzdiagramm)

Im dem Fall, dass die Nachrichten-Instanz nicht sofort über das gewählte Medium gesendet werden kann, wird sie im System zwischengespeichert. Ob bzw. wann die Instanz der Nachricht aus dem Zwischenspeicher gelöscht wird, hängt vom Vorgehen des Adressaten ab (Siehe Abbildung 5 - Abschnitt e).

Der eigentliche Prototyp der Nachricht bleibt auch nach der Sendung im System erhalten und wartet auf das erneute Eintreten eines auslösenden Ereignisses.

2.6.4 Löschen einer Nachricht

Eine Nachricht kann entweder manuell durch einen Teilnehmer oder automatisch durch ein Ereignis gelöscht werden. Dieses Ereignis kann vorher vom Ersteller in der Nachricht definiert werden. Nach dem Löschen einer Nachricht werden keine Instanzen mehr an irgendwelche Teilnehmer gesendet (Siehe Abbildung 5 - Abschnitt f)

2.7 Modell des Kontextes

Der Kontext stellt den Gesamt-Zustand der Informationen des Aktiven Campus dar. Er vereinigt die lokalen Kontextinformationen der Teilnehmer und den globalen Kontext des gesamten Campussystems. Der Begriff Kontext kann sich somit auf die Informationen über ein gerade betrachtetes Objekt beziehen oder aber auf die Gesamtheit an Informationen im System. Der Kontext ist hierarchisch aufgebaut und bietet als Schnittstelle den Zugriff auf Funktionen.

2.7.1 Funktionen

Funktionen können entweder direkt auf Attribute (Name/Wert-Paare) verweisen, als auch Berechnungen auf diesen Attributen und/oder anderen Funktionen ausführen und anbieten. Die Funktionen können von Moderatoren zur Verfügung gestellt und gepflegt werden.

Die Daten selbst, die im Kontext gespeichert werden, sind die Attribute.

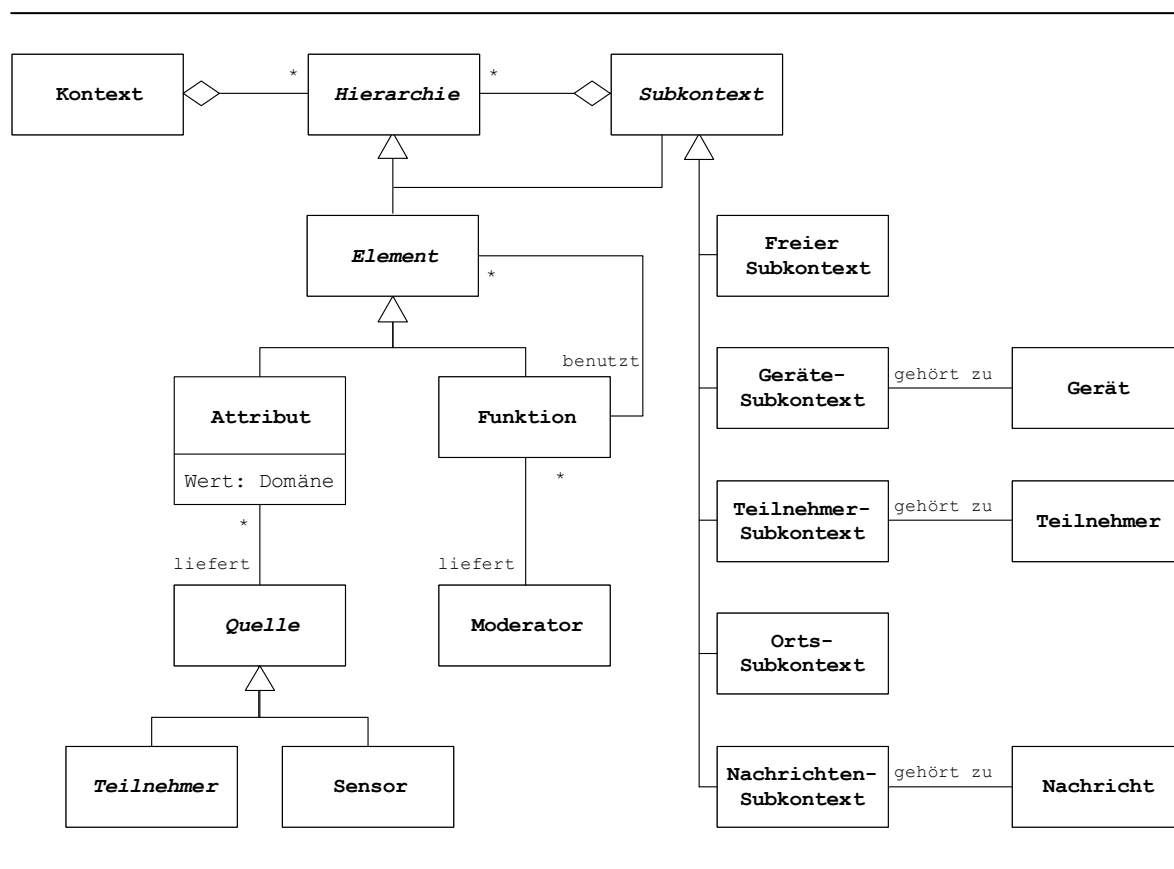


Abbildung 6: Modell der Kontextstruktur (UML Klassendiagramm)

2.7.2 Attribute

Attribute stellen Daten dar, die von Sensoren stammen oder von direkt von den Teilnehmern geliefert werden. Sensoren sind Geräte, die automatisch oder manuell eine Teilmenge des Kontextes aktualisieren können (Bsp.: Kamera, die die Länge einer Warteschlange misst, Zentrale Uhr des Campus, GPS-Sensor eines Gerätes, ...). Jedes Attribut speichert einen Wert vom Typ der Domäne des Attributes, z.B. Integer, Währung, Text, Zeit, Ort, Physikalische Daten, Mengen, ...). Von Moderatoren können Abstrakte Datentypen (ADTs) benutzt werden, um die Mächtigkeit des Kontextes den Gegebenheiten des Aktiven Campus anzupassen. Die Domäne eines Attributes kann sowohl kontinuierlich als auch diskret sein.

2.7.3 Hierarchie

Der Kontext besteht rekursiv aus Subkontexten, bei einer Wurzel beginnend. Jeder Subkontext kann weitere Subkontexte und Funktionen bzw. Attribute besitzen. Es ist auch eine Implementierung denkbar, die statt der Hierarchie ein Netz mit n:m Beziehungen benutzt, so dass ein Subkontext mehrere übergeordnete Kontexte besitzt, um die Informationen besser zu strukturieren.

Beispiel: Eine Person kann sowohl unter `context.persons.xyz` als auch unter `context.tum.staff.xyz` in den Kontext eingestellt werden. Um Redundanz zu vermeiden, handelt es sich um ein und denselben Subkontext.

Auf die hierarchisch angeordneten Subkontexte und/oder deren Attribute kann folgendermaßen zugegriffen werden: Es kann durch Ausdrücke ein bestimmter Subkontext (*Kontext.Subkontext*), alle seine direkten Subkontexte / Attribute (*Kontext.CHILDREN*), oder alle Subkontexte / Attribute in der Hierarchie unter ihm (*Kontext.**) ausgewählt werden. Auf Subkontexte kann sowohl absolut über die Wurzel als auch relativ durch den inhaltlich am nächsten liegenden Subkontext zugegriffen werden (Beispiel: In einer Nachricht kann auf ihre Attribute absolut über `context.message.ATTRIBUTE` oder direkt über `thiscontext.ATTRIBUTE` zugegriffen werden).

Da in einem stark dynamischen System wie dem Aktiven Campus ein Kontextwechsel sehr häufig und vor allem automatisch (ohne Zutun des Teilnehmers) passiert, muss die

Sichtbarkeit einzelner Elemente erweitert werden. Dadurch lassen sich die Situationen, in denen ein Ausdruck über Elementen zu `NULL` ausgewertet werden muss (Wert einer Funktion lässt sich nicht ermitteln, da entweder einer oder mehrere Typen nicht zusammenpassen oder sich ein Attribut nicht auffinden lässt), reduzieren, in dem bei nicht vorhandenen Elementen eines Subkontextes auf die entsprechenden Elemente der Subkontexte, die in der Hierarchie näher an der Wurzel liegen, zugegriffen wird. Ein Subkontext kann so als die Vereinigung von sich selbst mit allen übergeordneten Kontexten gesehen werden.

Beispiel: Ein Zugriff auf `context.tum.staff.person_xyz` greift auch auf mögliche Informationen in `context.tum.staff`, `context.tum` und natürlich `context` zu.

Die Problemfälle, die sich im Bezug auf die Semantik ergeben könnten, müssen von der Implementierung behandelt werden. Ein Beispiel: Der Subkontext einer Person wird durch Eintritt in eine Arbeitsgruppe in deren Subkontext kopiert. Da im Subkontext der Person das Element `Alter` fehlt, wird bei einer entsprechenden Anfrage auf das Element im Subkontext der Arbeitsgruppe gesucht. Falls diese ein Element `Alter` besitzt, so wird – semantisch inkorrekt – auf dieses Element zugegriffen.

2.7.4 Grundstruktur des Kontexts

Der Kontext enthält sowohl Subkontexte und Attribute, die systemweit wohldefiniert bzw. obligatorisch sind, als auch solche, die von jedem Teilnehmer frei definiert werden können. Die Bedingungen, die beispielsweise im System benutzt werden, um die Sendung einer Nachrichteninstanz zu veranlassen, müssen berücksichtigen, dass diese proprietären Attribute möglicherweise nicht in allen Subkontexten vorhanden sind.

Beispiele für obligatorische Subkontexte (von Implementierung abhängig):

- `context.participants`
Verzeichnis aller Teilnehmer des Systems
- `context.locations`
Übersicht aller Orte, die im System eine besondere Bedeutung haben (z.B. die Mensa, die Bibliothek, ein Hörsaal)

- `context.devices`
Geräte, sowohl die der Nutzer als auch öffentliche
- `context.functions`
Funktionen, die der Kontext zur Auswertung zur Verfügung stellt.
- `context.message`
Nachricht, deren Bedingung ausgewertet wird, oder die die Auswertung eines Filters anstößt.
- `context.messages`
Gesamtheit aller Nachrichten im System
- `context.events`
Veranstaltungen wie Vorlesungen, Vorträge, Kinofilme, etc.
- `context.groups`
Öffentliche / private Gruppen, realisiert durch Mengen von Subkontexten / Attributen.

Die Subkontexte in der Kontext-Hierarchie sind typisiert, sodass sie verschiedene semantische Objekte repräsentieren können, die jeweils obligatorische Subkontexte / Attribute besitzen:

- **FreierKontext**: Beliebiger Subkontext, der frei verwendbar ist und beliebig in die Hierarchie eingefügt werden kann. Beispiel: Lieblingsspeisen eines Teilnehmers
- **GeräteSubkontext**: Repräsentiert ein öffentliches oder privates Gerät.
Obligatorische Attribute:
 - `max_impact`: Maximale Wirkung einer Nachricht, die ausgeführt werden kann (Bild, Ton) (Siehe auch Kapitel 2.8)
 - `user`: Besitzer des Geräts (Student X)
 - `name`: Name des Geräts (HP Laserjet, Palm m500, Compaq iPaq)
 - `type`: Typ des Geräts (Laptop, Drucker)
 - ...
- **TeilnehmerSubkontext**: Repräsentiert einen Teilnehmer.
Obligatorische Attribute (beispielhaft):
 - `name`: Name (`context.participants.student_x`)

- `devices`: Geräte (`context.laptop_y`, `context.handheld_z`)
 - `coordinates`: Aufenthaltsort (`context.locations.mensa`, `x345:y678:z901`)
 - ...
- `OrtsSubkontext`: Repräsentiert eine Menge Koordinaten mit besonderer semantischer Bedeutung im Campus. Beispiel: Mensa, Hörsaal X
 Obligatorische Attribute (beispielhaft):
 - `name`: Name (Mensa)
 - `coordinates`: Koordinaten (`x345..x350:y678..680:z901..902`)
- `NachrichtenSubkontext`: Repräsentiert eine Nachricht im System.
 Obligatorische Attribute:
 - `id`: Eindeutige ID der Nachricht
 - `last_send`: Zeitpunkt, zu dem die Nachricht das letzte Mal an irgendeinen Teilnehmer gesendet wurde.
 - `last_received`: Zeitpunkt, zu dem die Nachricht das letzte Mal an den Auslöser des aktuellen Ereignisses gesendet wurde.
 - `actuator`: Teilnehmer, der das Ereignis ausgelöst hat, das die Sendung der Nachricht zur Folge hat.
 - `originator`: Teilnehmer, der die Nachricht erstellt hat.
 - `content_format`: Format des Inhalts der Nachricht
 - `content_size`: Format des Inhalts der Nachricht
 - `condition_predicates`: Alle Elemente des Kontexts, die in der Bedingung der Nachricht vorkommen (sinnvoll für Filter)
 - `content_context_elements`: Alle Elemente des Kontexts, die im Inhalt der Nachricht vorkommen (sinnvoll für Filter)
 - `content_message_attributes`: Alle Attribute der Nachrichte, die im Inhalt der Nachricht vorkommen (sinnvoll für Filter)

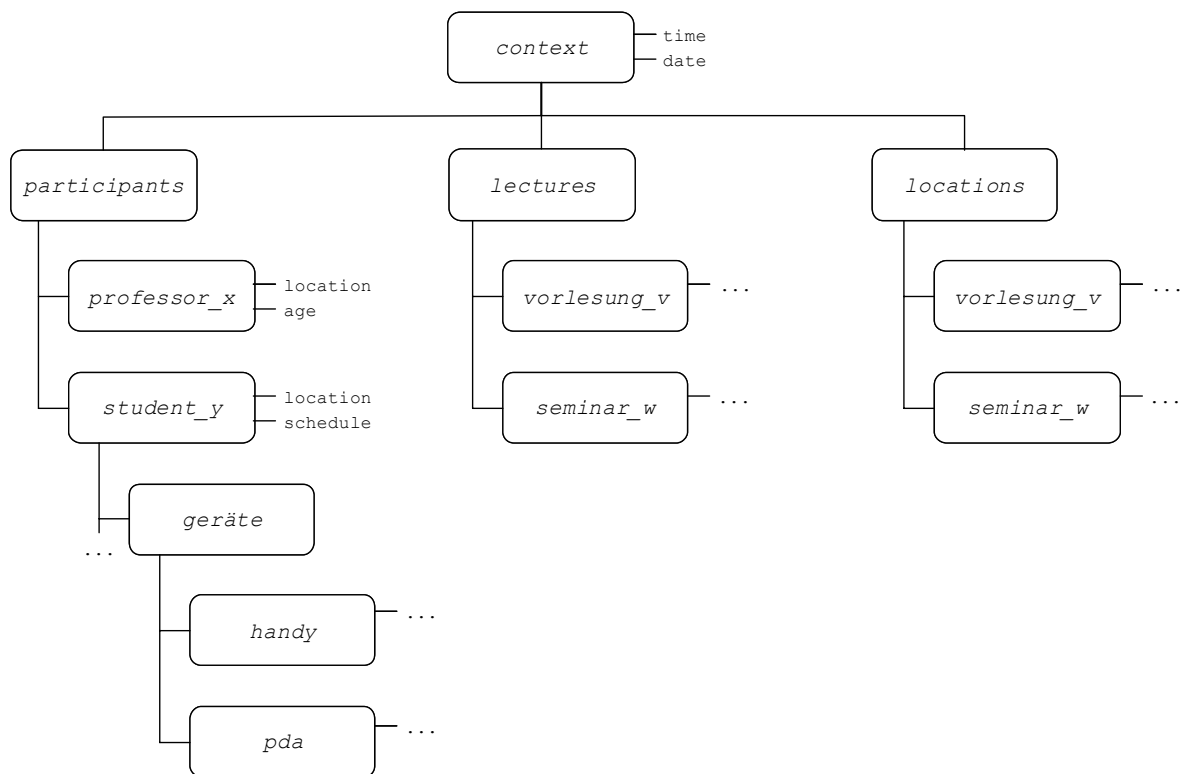


Abbildung 7: Kontext-Beispiel (stark vereinfacht)

2.7.5 Weitere Aufgaben und Funktionen des Kontextes

Der Kontext besitzt sowohl ein Gedächtnis als auch eine Berechnungslogik und kann somit vergangene Kontextinformationen erneut aufrufen und zukünftige Informationen interpolieren.

Der Zugriff auf den Kontext muss durch ein Rechtssystem gesichert werden, um persönliche Daten zu schützen. Ähnlich der Rechtevergabe eines relationalen Datenbanksystems werden Rechte auf jeder Ebene des Kontexts bzw. in jeder Granularität vergeben, also für jede mögliche Aktion gesondert (Beispiel: Lesen, Schreiben, Verändern von Attributen, Verändern von Metadaten, ...).

2.8 Modell einer Nachricht

Eine Nachricht dient als Vorlage für das Verhalten des Systems bei einem bestimmten Ereignis. Das System reagiert nach einem *Event-Condition-Action* (ECA)-Muster, dessen Bedingung (Condition) und daraus resultierende Reaktion (Action) in der Nachricht beschrieben werden.

- `<message> ::=`
 - `<condition>`
 - `<lifetime>`
 - `<iteration>`
 - `{ <content> <impact> }+`
 - `<addressee>`
 - `<feedback>`
 - `<originator>`

Nachricht: Prototyp einer Nachricht deren Instanzen an die Teilnehmer versendet werden können.

- `<condition>`: Bedingung als prädikatenlogischer Ausdruck, muss sich zu `true` auswerten lassen, damit eine Instanz der Nachricht gesendet wird. Es wird eine dreiwertige Logik unterstützt, die neben `true`, `false` auch `null` für unbekannt kennt. Dieser Fall tritt ein, wenn ein Attribut des Kontextes nicht vorhanden oder anderweitig nicht auswertbar ist (z.B. keine Interpolation möglich, etc.). Für den Ausdruck stehen die Funktionen und Attribute des Kontexts zur Verfügung, die insbesondere z.B. auch die aktuelle Nachricht oder den Auslöser der Bedingungs-Auswertung umfassen.

- `<lifetime> ::= <condition>`

Lebenszeit: Sobald dieser Ausdruck zu `false` ausgewertet werden kann, wird die Nachricht automatisch aus dem System gelöscht.

- `<iteration> ::= <condition>`

Wiederholung: Es wird solange keine neue Instanz der Nachricht versendet, bis der Ausdruck `true` ist. Kann vom Teilnehmer durch einen Filter beeinflusst werden.

- `<content> ::= { format file } * <document>`

Inhalt: (Möglicherweise dynamische) Nutzdaten, die durch die Nachricht übermittelt werden. Kann vom Teilnehmer durch die Erstellung eines Filters beeinflusst werden.

- `format ::=` Element aus der Hierarchie von möglichen Formaten (inklusive Ordnung, damit Filter z.B. ein Minimum bilden können), festgelegt durch Moderatoren
 - `file`: Nutzdatei in beliebigem Format
 - `<document> ::= format { <documentElement> | text }+`
 - `<documentElement> ::=`
`contextElement |`
`INCLUDE |`
`...`
 - `contextElement`: Kontext-Attribut oder Subkontext
 - `INCLUDE`: Bindet dynamische Quellen in den Inhalt ein
 - `text`: Text in festgelegtem Format (z.B. HTML oder XML)
- `<impact> ::= {`
`BEEP |`
`DISPLAY |`
`STORE |`
`<proprietaryImpact>`
`}+`

Wirkung: Vom Absender gewünschte Wirkung des Inhalts beim Empfänger. Kann eine oder mehrere, auf allen Geräten wirksame oder aber auch geräteabhängige Aktion sein, die als Reaktion auf die Nachricht (möglicherweise auch abhängig vom Inhalt), ausgeführt wird. Kann vom Teilnehmer durch einen Filter beeinflusst werden. Auf den Wirkungen muss eine Ordnung bestehen, damit per Filter eine maximale/minimale Wirkung bestimmt werden kann.

- `BEEP`: Macht den Teilnehmer akustisch auf die Nachricht aufmerksam
- `DISPLAY`: Zeigt die Nachricht auf dem Ausgabeteil des Gerätes an

- STORE: Speichert die Nachricht zwischen, bis sie abgerufen wird
- <proprietaryImpact> = PRINT | MAPGUIDE | REMIND | ...
Wirkungen, die bestimmte Geräten benötigen und auf anderen Geräten keine Wirkung zeigen
- PRINT: Druckt den Inhalt aus
- MAPGUIDE: Ruft ein Navigationsmodul auf, das den Teilnehmer zum Ort, der im Inhalt angegeben ist, leitet
- REMIND: In das Terminkalender-Modul wird ein Termin aus den Daten des Inhalts eingerichtet

- <addressee> ::=
[context.message.actuator]
[context.message.originator]
{ participant | device }*

Adressat: Teilnehmer und/oder Gerät, der die Nachricht empfangen soll. Die Teilnehmer können auch als Gruppe angesprochen werden

- participant: Teilnehmer-Subkontext
- device: GeräteSubkontext

- <feedback> ::=
NONE | COUNT | COPY | messageAttribute | contextElement

Rückmeldung: Beschreibt Daten, die nach einer Sendung einer Nachrichteninstanz an den Ersteller der Nachricht gesendet wird.

- NONE: Es wird keine Benachrichtigung gesendet
- COUNT: Es wird die Gesamtzahl der bisherigen Sendungen gemeldet
- COPY: Es wird eine Kopie des Inhalts der Nachrichteninstanz gesendet
- messageAttribute: Beliebiges Attribut von context

- <originator> ::= participant

Ersteller: Teilnehmer, der die Nachricht ins System eingestellt hat.

2.9 Modell eines Filters

Im Gegensatz zur Bedingung einer Nachricht dient ein Filter dazu, die Zahl der *false positives*, also der fälschlich als relevant zugestellten Nachrichten, zu minimieren.

Eine zentrale Annahme des beschriebenen Nachrichtensystems ist die, dass es sich beim Aktiven Campus um ein geschlossenes System handelt. Das bedeutet im Kern, dass - anders als bei offenen Systemen wie dem Internet - eine zentrale Kontrollinstanz vorhanden ist. Durch den Einsatz von Filtern ist es trotz alledem möglich, eine Unterdrückung von unerwünschten Inhalten zu erzielen. Die praktische Benutzbarkeit würde aber wahrscheinlich in einem vergleichbaren offenen System (z.B. Internet) zu sehr leiden (vgl. Problematik des E-Mail-Spams).

Ein Filter kann sowohl präventiv eingerichtet werden, um von vorn herein alle möglichen unerwünschten Nachrichten abzublocken, als auch nachträglich, um auf bestimmte unerwünschte Nachrichten zu reagieren.

Weiterhin können auch erwünschte Nachrichten beeinflusst werden. Ein Beispiel ist die Weiterleitung bestimmter Nachrichten oder die Hervorhebung besonderer Nachrichten, in dem z.B. die Wirkung angehoben wird. Eine wichtige Nachricht würde dann eventuell nicht nur angezeigt, sondern auch durch ein akustisches Signal gemeldet.

- `<filter> ::= order name <decay> <condition>`
`<modifications>`

Filter: Bietet dem Teilnehmer die Möglichkeit, Nachrichten entweder abzublocken oder zu verändern

- `order`: Reihenfolge, in der der Filter anzuwenden ist. Dadurch wird der kombinierte Einsatz mehrerer Filter gewährleistet.
- `name`: Name des Filters
- `<decay> ::= <condition>`

Verfall: Wenn diese Bedingung, die vor und nach Auswerten des Filters evaluiert wird, erfüllt ist, so wird der Filter gelöscht.

- `<condition>`: Bedingung wie bei einer Nachricht. Wenn sich die Bedingung zu `true` auswerten lässt, werden die Modifikationen des Filters ausgelöst.

- `<modifications> ::= { <modification> }+`

Modifikationen: Aktionen, die die Nachricht, durch die der Filter ausgelöst wurde, verändern kann.

- `<modification> ::= <weight> (`
`<modifyDelivery> |`
`<modifyAddressee> |`
`<modifyFeedback> |`
`<modifyImpact> |`
`<modifyIteration> |`
`<modifyContent>)`

Modifikation: Mit einem Gewicht versehene Anweisung, auf ein bestimmtes Attribut einer Nachrichten-Instanz Einfluss zu nehmen. Nicht alle Attribute können verändert werden.

- `<weight> ::= WEAK | STRONG`

Gewicht der Modifikation bestimmt Abarbeitungsreihenfolge, falls mehrere Modifikationen (von eventuell mehreren Filtern) auf das gleiche Attribut wirken sollen.

- WEAK: wird von STRONG überschrieben

- `<modifyDelivery> ::= ALLOW | DENY`

Zustellungsstatus der Nachricht: Mit dieser Modifikation kann die Zustellung der Nachricht unterbunden oder erlaubt werden. Somit kann z.B. mit einem Filter, der in der Reihenfolge am Anfang steht, ein generelles Verhalten für Nachrichten festgelegt werden, auf die kein weiterer Filter zutrifft.

- ALLOW: Erlaubt die Zustellung der Nachricht
- DENY: Verweigert die Zustellung der Nachricht

- `<modifyAddressee> ::=`
`[FORWARD_TO <group>]`
`[COPY_TO <group>]`

Erlaubt die Weiterleitung oder Senden einer Kopie an eine Gruppe von Teilnehmern (möglicherweise inklusive Angabe der Geräte). Dadurch wird eine erneute Auswertung der Filter der neuen Adressaten angestoßen.

- FORWARD_TO: Leitet die Nachricht weiter zu einer Gruppe von Teilnehmern (möglicherweise inklusive Angabe der Geräte)
- COPY_TO: Sendet eine Kopie der Nachrichten-Instanz weiter zu einer Gruppe von Teilnehmern (möglicherweise inklusive Angabe der Geräte)

- `<modifyFeedback> ::= context.message.feedback = <feedback>`
Erlaubt es, die Art und Weise der Rückmeldung zu beeinflussen, die nach Sendung der Nachricht auf Wunsch des Erstellers veranlasst wird.

- `<modifyImpact> ::= context.message.impact =
<impact> |
<selectionFunction> (
 { <impact> | context.message.impact }+
)`

Die Wirkung der Nachricht kann angepasst werden, entweder durch eine Konstante oder eine Funktion bzw. ein Attribut des Kontextes.

- `<selectionFunction> ::= MIN | MAX`

Auswahlfunktionen für Werte (auf der Domäne muss dazu eine Ordnung existieren)

- `<modifyIteration> ::=
context.message.iteration = <condition>`

Die Wiederholung der Nachricht (im Bezug auf den Filter-Ersteller) kann verändert (d.h. neu gesetzt) werden.

- `<modifyContent> ::=
MESSAGE_ATTRIBUTE_ALLOW <messageAttribute> |
MESSAGE_ATTRIBUTE_DENY <messageAttribute> |
CONTEXT_ELEMENT_ALLOW contextElement |
CONTEXT_ELEMENT_DENY contextElement |
FORMAT (
 format |`

```

    <selectionFunction> { <format> |
                                context.message.content.format }+
) |
SIZE (
    constant |
    <selectionFunction> { <constant> |
                                context.message.content.size }+
) |
...

```

Der Inhalt kann in seinem Format und seiner Größe eingeschränkt werden. Zusätzlich können Nachrichten-Attribute oder Kontext-Elemente im Inhalt verboten bzw. erlaubt werden. Dies betrifft natürlich nur Kontextinformationen, die im privaten Kontext des Filtererstellers liegen bzw. Kontextinformationen in Nachrichten, die an den Filterersteller adressiert sind.

2.10 Szenarien (siehe Anhang)

Um den benötigten Funktionsumfang des Systems zu ermitteln und die Modellierung zu ermöglichen wurden zahlreiche Szenarien entworfen, die den Einsatz des Systems so gut wie möglich beschreiben sollen. Diese Szenarien sind gut geeignet, die Funktionsweise des Systems zu verdeutlichen und zu verstehen.

Ein Szenario wird im Wesentlichen durch folgende Informationen beschrieben:

- **Akteure:** Personen und Systeme, die am Szenario teilnehmen (aktiv oder passiv)
- **Kontext:** Art der Kontextinformationen, die benötigt werden
- **Flow of events:** Detaillierter Ablauf des Szenarios in einzelnen Schritten
- **Nachricht:** Eine oder mehrere Nachrichten, die die gewünschten Funktionalität ermöglichen
- **Filter:** Einer oder mehrere Filter, die die gewünschten Funktionalität ermöglichen

Für die genaue Beschreibung der Nachrichten und Filter wird die in Kapitel 2.8 und 2.9 bereits eingeführte Syntax verwendet.

3. Implementierung

Als Vorarbeit zu einer vollständigen, lauffähigen Implementierung des gesamten Modells des Aktiven Campus sollen hier zentrale Fragen bezüglich der Machbarkeit erörtert werden.

3.1 Anforderungen

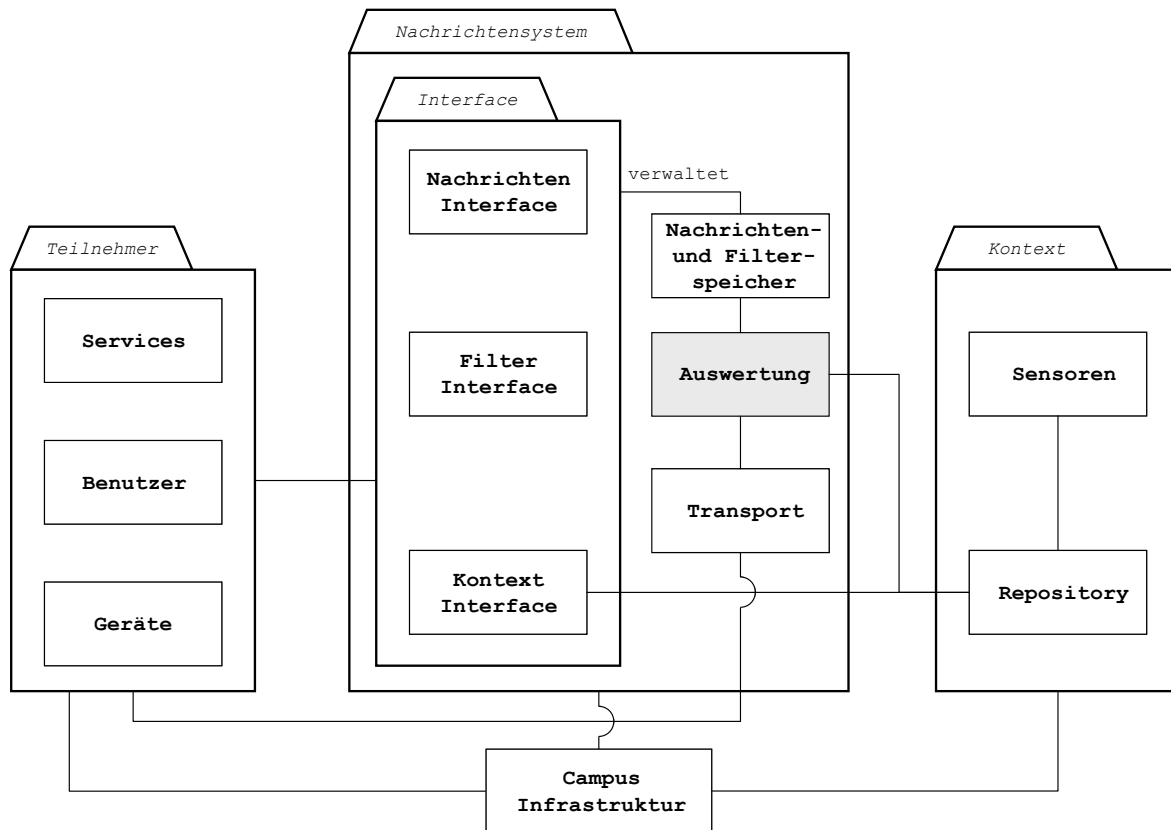


Abbildung 8: Implementierung des Aktiven Campus-Modells

Eine vollständige Implementierung sollte die folgenden Funktionen beinhalten:

- Verwaltung des *Kontextes* (inklusive Behandlung von Sensordaten, Veränderung des Kontextes durch Teilnehmer, Gewährleistung der Sicherheitsaspekte des Kontexts, Registrierung und Auswertung von Kontextänderungen)
- Verwaltung der *Nachrichten* (inklusive Erstellung, Löschung und automatischer Verarbeitung, Möglichkeiten zur Verwendung von Vorlagen und Moderation)

- Verwaltung der *Filter* (inklusive Erstellung, Löschung, Anwendung auf Nachrichten, Möglichkeiten zur Verwendung von Vorlagen und Moderation)
- *Übertragung* der Nachrichten (inklusive Wahl des Übertragungsmediums und Ausgabegerätes, Bereitstellung eines Zwischenspeichers)
- Bereitstellung eines Rahmens, um eigene Dienste einzubinden, die automatisch neue Nachrichten erstellen können

Weiterhin ist die genaue Definition, Modellierung und Implementierung der Infrastruktur (Server, Netzwerke, ...) und der Protokolle nötig, durch die auf den Aktiven Campus zugegriffen wird (mobile und stationäre Geräte der Teilnehmer und Betreiber des Aktiven Campus).

Im Folgenden werden Überlegungen und Vorgehensweisen zu einer Implementierung präsentiert, die die Kern-Funktionalität des Nachrichtensystems, die *Auswertung*, gewährleistet (siehe Abbildung 8). Diese Kern-Funktionalität besteht aus folgenden Schritten (siehe Abbildung 7):

- Sensoren melden neue Werte an das System
- Aus den Sensorendaten wird ein Wert für die korrespondierenden Kontext-Attribute gebildet (sog. *Data Cooking*, d.h. aus den rohen Sensordaten werden abstraktere Daten errechnet. Bsp.: Errechnung der Position des Benutzers aus Wave-Lan-Empfangsstärken)
- Das System prüft den Kontextwechsel auf Relevanz bezüglich der vorhandenen Nachrichten. (*Change Detection*)
- Falls eine Relevanz vorliegt, werden alle Nachrichten ermittelt, deren Bedingungen durch den Kontextwechsel erfüllt werden (*Matching*)
- Die relevanten Nachrichten werden gesendet

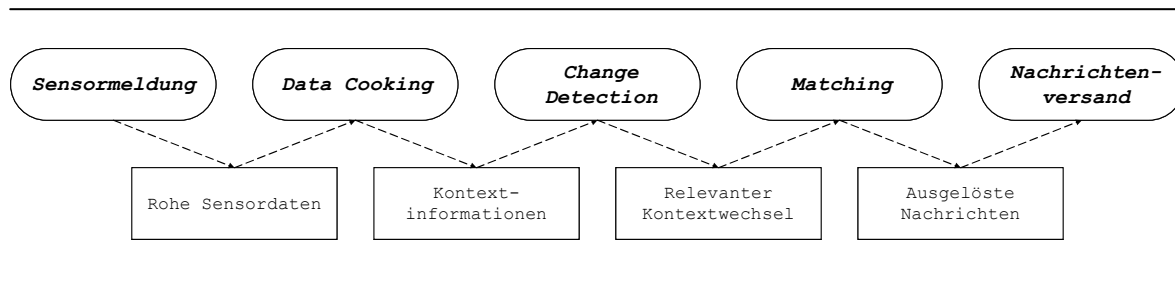


Abbildung 9: Kern-Funktionalität des Nachrichtensystems (UML Statechart Diagramm)

Im Hinblick auf diesen Umfang der Implementierung erscheinen die folgenden Eigenschaften des Systems als wünschenswert:

- Effizienzeigenschaften:
 - Anzahl der unterstützten Kontextwechsel pro Zeit
 - Anzahl der unterstützten Dimensionen des Kontexts
 - Anzahl der unterstützten gleichzeitigen Nachrichten im System
 - Maximale Komplexität der Nachrichten und Filter
- Qualitätseigenschaften:
 - Tatsächliche Entdeckung relevanter Kontextwechsel (*recall*, d.h. Anzahl der gemeldeten relevanten Kontextwechsel / tatsächlich eingetretene relevante Kontextwechsel)
 - Reaktionszeit innerhalb von Grenzen, die sinnvollen Einsatz der Nachrichten ermöglichen

3.2 Heuristik für die Implementierung der Auswertung

Für die nachfolgend erläuterte Heuristik werden daher folgende Einschränkungen, Annahmen und Zusicherungen gemacht:

- Jede ausgelöste Nachricht wird gesendet. Da die Auswertung jedoch nicht in Nullzeit erfolgen kann, kann sich bis zum Versenden/Erhalt der Nachricht ein neuer Kontextwechsel ereignet haben, der die Nachricht theoretisch obsolet werden ließe. Um eine bessere Handhabung dieser Fälle zu erreichen, sollte ein *time-to-live* Wert eingeführt werden. Überschreitet die Abarbeitung einer Nachricht diese Grenze, so

wird die Nachrichteninstanz fallengelassen und nicht weiter bearbeitet oder gesendet.

- Falls von einer Dimension bekannt ist, dass sie kontinuierlich ist, so kann zugesichert werden, dass ein Sprung über eine bestimmte Grenze erkannt wird, auch wenn dieser Grenze nicht genau erreicht wird.
- Die Dimension der Zeit wird mit einer speziellen Semantik versehen, da alle anderen Dimensionen von ihr abhängig sind. Dadurch lassen sich Vergleiche in gewissen Dimensionen auf Vergleiche in der Zeit-Dimension übertragen. Durch diese Interpolationen opfert das System Genauigkeit, um Performanz zu gewinnen.
- Zu jeder Dimension kann eine Angabe zur feinsten Granularität gemacht werden, in der Änderungen sinnvoll erkannt werden (Bsp.: cm, sec, ...). Diese Granularität lässt sich auch aus dem aktuellen Nachrichtenbestand ableiten.
- Es wird die Annahme gemacht, dass sich die Nachrichtenschemata nur selten im Vergleich zu den Attributswerten ändern. Die Schemata verändern sich im Allgemeinen nur dann, wenn eine Nachricht hinzugefügt oder gelöscht wird.
- Die Heuristik geht davon aus, dass nicht alle Änderungen der Sensordaten eine Kontextänderung herbeiführen und nur wenige dieser Kontextänderungen für das Auslösen einer Nachricht relevant sind.

Die hier beschriebene Heuristik versucht, den Abgleich zwischen Kontextwechseln und Nachrichtenbedingungen so zu optimieren, dass er effizient implementiert werden kann.

Dazu werden folgende Schritte benutzt:

- Übergang von Sensormeldungen zu Kontextinformationen
- Interpretation von neuen Kontextinformationen als relevante Kontextwechsel bezüglich der existierenden Nachrichten
- Beziehungen der Nachrichten zueinander und zu verschiedenen Dimensionen des Kontextes
- Techniken wie Caching und Interpolation

Das System, das die Nachrichtenvermittlung übernimmt, arbeitet nach dem Prinzip, dass jede eingehende Sensormeldung verarbeitet wird (*passives Warten*) und in der Sendung aller Nachrichten endet, die durch die Sensormeldung ausgelöst werden.

3.2.1 Sensormeldung

Sobald sich der Wert eines Sensors (WLAN-Karte, Kamera, Mikrofon, ...) ändert, wird dieser neue Sensorwert an das System übertragen.

Der Sensorwert muss unter Umständen erst umgewandelt werden, bevor er in den Kontext eingetragen werden kann. So könnte beispielsweise aus der Empfangsstärke einer WLAN-Karte und Informationen zu den Sendern ihre Position errechnet werden. Im Kontext wird dann nicht die Feldstärke sondern die Position des Gerätes gespeichert. Dieser Vorgang wird *Data Cooking* genannt.

3.2.2 Data Cooking

Bereits in dieser Phase kann das System optimiert werden, da nicht jede Veränderung des Sensors eine Kontextänderung bewirken muss. Es liegt also nahe, die Sensordaten auf möglichst abstrakte Daten abzubilden, damit die Bearbeitung vieler Sensormeldungen bereits hier abgebrochen werden kann. Die Granularität der Dimension, die zu den Daten eines oder mehrerer Sensoren gehört, kann auch von den vorliegenden Nachrichten beeinflusst werden. So macht beispielsweise eine Nachricht, die auf einen Stundenwechsel wartet, die Weiterverarbeitung von Sekunden unnötig. Hierbei muss natürlich beachtet werden, dass eine solche Herabsetzung der Granularität abhängig von den jeweiligen Nachrichten rechtzeitig wieder heraufgesetzt werden muss. Dazu kann es nötig sein, vorher eine Interpolation durchzuführen, um zu erfahren, wie lange eine niedrige Granularität möglich oder sinnvoll ist.

Ergebnis des *Data Cookings* ist eine Kontextinformation, von der bekannt ist, dass das Attribut, zu dem sie gehört, sich gerade als letztes verändert hat. Das *Data Cooking* lässt sich mit dem Vorgehen bei Kantenerkennungs-Verfahren aus der Bildverarbeitung vergleichen.

Weitere Informationen zum Einsatz von Sensoren und dem Vorgang des Data Cooking können aus Abhandlungen über Wearable Computing (Forschungsbereich Augmented Reality) entnommen werden (Siehe z.B. [GoLe99]).

3.2.3 Change Detection

Die Grundidee der *Change Detection* ist die Erkenntnis, dass viele Kontextwechsel für die vorhandenen Nachrichten irrelevant sind. Die zwei Hauptgründe sind:

1. Falls sich ein Attribut ändert, auf das sich keine Bedingung einer Nachricht bezieht, so muss ein Wechsel des Attributes nicht beachtet werden, bis sich die Nachrichtenschemata ändern (trivial).
2. Die Bedingungen der Nachrichten enthalten logische AND-Verknüpfungen. Falls sich ein Attribut des Kontextes ändert, aber die Nachrichtenbedingung durch keinen Wert, den das Attribut annehmen könnte ausgelöst werden kann, da ein anderes Prädikat das verhindert (da es zu `false` ausgewertet wird), so müssen Veränderungen dieses Attributs solange nicht beachtet werden, bis das andere Prädikat sich zu `true` auswerten lässt.

Die *Change Detection* leitet nur Kontextwechsel weiter, die für die Datenbasis der Nachrichten relevant sind. Nur diese Kontextwechsel müssen daraufhin untersucht werden, welche Nachrichten sie auslösen. Die *Change Detection* lässt sich als Filter auffassen, der so viele Kontextwechsel wie möglich als irrelevant ablehnt, um das nachfolgende *Matching* so wenig wie möglich in Anspruch zu nehmen.

3.2.4 Matching

Die Funktion des *Matchings* liefert zu jedem Kontextwechsel eine Menge von Nachrichten, die durch diesen Wechsel ausgelöst werden, d.h. deren Bedingungen sich mit den aktuellen Werten des Kontextes zu `true` auswerten lassen. Dabei ist zu beachten, dass auch die Wiederholungsanweisungen einer Nachricht (*iteration*) eine Bedingung darstellen. Da es sich bei beiden Bedingungen um prädikatenlogische Ausdrücke handelt, können sie leicht kombiniert werden. In den folgenden Ausführungen wird immer von einer Bedingung pro Nachricht ausgegangen, in der bereits alle Informationen einer Nachricht eingebunden sind. In diesem Schritt müssen auch die Attribute des Kontexts, die eine spezielle Semantik tragen, also z.B. `actuator`, `originator` und `message`, umgewandelt werden. So bedeutet beispielsweise ein Vergleich auf `originator.location` und eine Bedingung auf `originator.name = „Professor X“`, dass `professor_x.location` ausgewertet werden muss.

Die Hauptarbeit des *Matchings* liefert ein Index über jede einzelne Dimension, in dem die Intervalle der Bedingungsprädikate aller Nachrichten gespeichert sind, die das Attribut verwenden. Da die Bedingungen aber für Vergleichen nicht nur Konstanten, sondern auch andere Attribute verwenden können, ist es nötig, zusammengesetzte, *virtuelle Attribute* des Kontexts zu bilden, sodass nur noch von Vergleichen mit Konstanten ausgegangen werden muss.

Ein geeigneter Index ist die *Interval Skip List*, die für die Prädikatenindexierung eingesetzt werden kann. Sie kann Intervalle und Datenpunkte indexieren, ermöglicht somit *Stabbing Queries* und benötigt für die Suche $O(\log n)$, für das Einfügen und Löschen von Intervallen $O(\log^2 n)$. Ein Nachteil ist der Platzbedarf von $O(n \log n)$.

Die Interval Skip List nutzt einen wahrscheinlichkeitstheoretischen Ansatz. Sie ähnelt einer verketteten Liste, bis auf die Tatsache, dass jedes Element mehrere vorwärts zeigende Verweise enthält. Die Anzahl von Verweisen eines Listeneintrages nennt man *Level*, ein neuer Eintrag erhält eine zufällige Anzahl von Levels. Die Liste beginnt mit einem leeren Listenkopf (dem *Header*) und endet mit leeren Verweisen.

Die Level der Verweise sind nach aufsteigenden Werten geordnet, ein Verweis auf Level x zeigt also auf ein höheres Element als ein Verweis auf Level $x-1$. Die Verweise werden mit den Intervallen beschriftet, die sie abdecken. Falls ein Verweis auf einem Level x mit einem Intervall beschriftet ist, so werden Verweise, die das Intervall ebenfalls enthalten, aber auf einem kleineren Level als x liegen, nicht beschriftet, da sie im Intervall auf dem höheren Level enthalten sind. Ein Listenelement wird mit einem Intervall beschriftet, falls er einen Endpunkt des Intervalls bildet.

Abbildung 10 zeigt das Beispiel einer Interval Skip List, in der folgende fünf Intervalle gespeichert sind (eine geschlossene untere Intervallgrenze wird durch $[$ gekennzeichnet, eine offene durch $]$, und umgekehrt):

- a: $[2, 17]$
- b: $] 17, 20]$
- c: $[8, 12]$
- d: $[7, 7]$
- e: $[-\infty, 17 [$

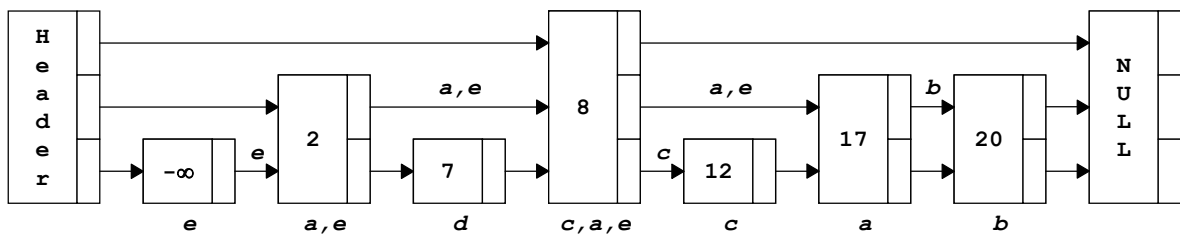


Abbildung 10: Beispiel einer Interval Skip List

Um bei einer Suche alle Intervalle zu ermitteln, die einen Eintrag k überdecken, wird beim höchsten Level des Listenkopfes begonnen. Falls dieser Verweis auf einen Eintrag $\geq k$ zeigt, wird das aktuell betrachtete Level reduziert. Ansonsten wird die Untersuchung bei dem Listeneintrag fortgesetzt, auf den verwiesen wird. Falls das aktuelle Level auf das niedrigste Level gesenkt wird, ist der gesuchte Eintrag entweder direkt hinter dem aktuellen Listeneintrag zu finden oder nicht in der Liste vorhanden.

Eine Menge S speichert während dieser Suche die Intervalle, die den gesuchten Eintrag enthalten. Immer wenn auf ein niedrigeres Level gesprungen wird, wird die Markierung des oberen Verweises zu S hinzugefügt. Wenn der gesuchte Eintrag K in der Liste vorhanden ist, werden anschließend alle seine Markierung zu S hinzugefügt, ansonsten die Markierungen auf dem niedrigsten Level des aktuellen Eintrags.

Im oberen Beispiel liefert die Suche nach allen Intervallen, die den Punkt 6 überdecken, folgendes Ergebnis: Nachdem beim Eintrag 2 vom zweiten Level auf das erste reduziert wird, werden die Markierungen a und e zu S hinzugefügt. Da der Eintrag 6 nicht vorhanden ist und auf dem Verweis von 2 zu 7 kein Intervall steht, bilden diese beiden Markierungen das endgültige Ergebnis.

Weitere Details zur Interval Skip List und Ihrem Einsatz als Prädikatenindex können [HaJo94] entnommen werden.

3.2.5 Nachrichtenversand

Auf alle Nachrichten, die durch geeignete Kontextwechsel ausgelöst wurden, werden abschließend für sie relevante Filter der jeweiligen Teilnehmer angewendet, da diese Filter noch über alle Modalitäten einer Sendung entscheiden können.

Falls die Filter es zulassen, werden die Nachrichten dann an die entsprechenden Adressaten versandt.

In der Anwendung der Filter liegt eine weitere Optimierungsmöglichkeit, auf die hier aber nicht näher eingegangen wird: Falls Filter Nachrichten verändern oder gar ganz unterbinden, kann es in einigen Fällen auch sinnvoll sein, aus diesen Informationen über die Filter Modifikationsmöglichkeiten im *Data Cooking* bzw. in der *Change Detection* zu gewinnen.

Beispiel: Ein Benutzer unterbindet per Filter alle Nachrichten einer bestimmten Quelle, daher können alle Nachrichten und somit auch die Wechsel der in der Bedingung enthaltenen Attribute ignoriert werden, die diesen Benutzer als direkten Adressaten tragen bzw. die Vergleiche von Attributen ignoriert werden, die aus seinem Kontext stammen.

3.3 Algorithmus für die Auswertung im Nachrichtensystem

Im Folgenden wird der oben beschriebene Prozess abstrahiert. Es wird davon ausgegangen, dass die Verarbeitung beginnt, sobald sich ein Attribut des Kontextes verändert.

3.3.1 Trivialer Ansatz

Die einfachste Möglichkeit der Implementierung besteht darin, nach jeder Änderung des Kontextes alle Nachrichten daraufhin zu überprüfen, ob sie ausgelöst werden. Bei der erwarteten Vielfalt des Kontextes ist mit einer sehr ineffizienten Auswertung zu rechnen, die kaum geeignet scheint, eine Nachricht in ausreichender Zeit (d.h. Zustand des Kontexts beim Auslösen der Nachrichten entspricht weitgehend dem Zustand beim Eintreffen der Nachricht) zu überprüfen, zu verarbeiten und zu versenden.

3.3.2 Grundlegende Optimierungen

Gegeben ist

- eine Menge von k *Nachrichten* $M_0 \dots M_{k-1}$, die im System gespeichert ist
- eine Menge von m *Attributen* des Kontextes, $A_0 \dots A_{m-1}$

Jede Nachricht M_i hat eine *Bedingung* (die, wie weiter oben erläutert, um die Bedingungen aus Nachrichtenattribut Löschung bzw. Wiederholung erweitert werden kann). Diese prädikatenlogische Bedingung kann in die disjunktive Normalform gebracht werden, sodass sie sich aus mehreren Teilformeln, die mit \vee verknüpft sind, zusammensetzt. Jede Teilformel wird als *Auslöser* (Trigger) bezeichnet. Jedem Auslöser ist mindestens eine Bedingung zugeordnet, deren Teil er ist. Die Auslöser aller Bedingungen ergeben die Menge an Auslösern $T_0 \dots T_{n-1}$. Jeder Auslöser besteht aus einem oder mehreren *Prädikaten*, die mit \wedge verknüpft sind und hat die Form:

$$T_i = P_1 \wedge P_2 \wedge \dots \wedge P_i.$$

Jedem Prädikat können ein oder mehrere Auslöser zugeordnet werden, von denen es ein Teil ist. Die Menge der Prädikate aller Auslöser ist $P_0 \dots P_{q-1}$.

In den Prädikaten selbst kommen nur Vergleiche mit Konstanten vor: $P_i = b_1 <_1 A_j <_2 b_2$ wobei A_j ein Attribut des Kontextes, b_1 und b_2 zwei beliebige, aber konstante Grenzen und

$<_1$ und $<_2$ zwei Vergleichsoperatoren aus der Menge $\{ <, \leq \}$ darstellen. Die Grenzen dürfen auch die Werte $-\infty$ und $+\infty$ annehmen.

Falls in einer Bedingung ein Vergleich eines Attributs mit einem anderen Attribut vorkommt, so kann dieser Vergleich durch einen oder mehrere Vergleiche eines zusammengesetzten Attributs ersetzt werden:

- Ein Vergleich mit mehr als einem Attribut wird durch eine Grenze von $-\infty$ und $+\infty$ in zwei Vergleiche mit jeweils nur einem Attribut umgewandelt: Aus $A_1 < A_2 < A_3$ wird $A_1 < A_2 < \infty$ und $-\infty < A_2 < A_3$
- Ein Vergleich zwischen zwei Attributen wird mittels mathematischen Umformen zu einem Vergleich eines virtuellen (zusammengesetzten) Attributs mit zwei Konstanten umgeformt: Aus $k_1 A_1 < k_2 A_2 < \infty$ wird $k_1 / k_2 < A_3 < \infty$ wobei A_3 ein virtuelles Attribut mit $A_3 = A_2 / A_1$ ist.

Gleiche Prädikate können zu einem neuen Prädikat verschmolzen werden, das auf die Auslöser der ursprünglichen Prädikate verweist.

Gleiche Auslöser können zu einem neuen Auslöser verschmolzen werden, der auf die Bedingungen der ursprünglichen Auslöser verweist.

Gleiche Bedingungen können zu einer neuen Bedingung verschmolzen werden, die auf die Nachrichten der ursprünglichen Bedingung verweist.

Die Menge an Prädikaten, die zu einem Auslöser gehören, werden nach der Häufigkeit geordnet, mit der sich die Attribute, auf die in dem jeweiligen Prädikat zugegriffen wird, verändern. Diese so genannte *Update Ratio* kann entweder manuell pro Attribut eingegeben werden oder, besser, automatisch ermittelt werden. Die Sortierung beginnt mit den Prädikaten, deren Attribute sich am langsamsten ändern.

Die Heuristik, nach der diese Sortierung stattfindet, bestimmt zu einem großen Teil die Effizienz der Implementierung, da Attribute, die sich häufig ändern, viele Vergleiche fordern. Es ist sinnvoll, in diese Ordnung Expertenwissen einfließen zu lassen, z.B. um Prädikate zu ermitteln, deren Wert sich (in absehbarer Zeit) nicht mehr von *false* auf *true* ändern kann. (Vergleiche zu dieser Ordnung die Gruppierungs- und Vorrangsschemata in der Regelbasierten Programmiersprache OPS5, die in den 70er und 80er Jahren für Expertensysteme und KI eingesetzt wurde [Fo77])

Die jeweils letzte Auswertung eines Prädikates P_i wird gecacht, bis sich der Wert bei einer späteren Auswertung ändert.

Für jeden Auslöser wird ein Zeiger gespeichert, der auf das (in der Sortierung nach Update Ratio) erste Prädikat zeigt, dessen gecachter Wert `false` ist. Alle Prädikate, die in der Sortierung nach diesem Prädikat stehen, müssen bei einem Kontextwechsel nicht überprüft werden, da der Auslöser durch die \wedge -Verknüpfung und das Prädikat, auf das der Zeiger zeigt, auf jeden Fall zu `false` ausgewertet wird. Diese Prädikate werden als *inaktiv* bezeichnet, sofern nicht in einem anderen Auslöser dasselbe Prädikat an einer Stelle vor dem Zeiger dieses Auslösers steht.

Ein Attribut wird als *inaktiv* bezeichnet, falls es nur in inaktiven Prädikaten oder in gar keinem Prädikat vorkommt. Die Kontextwechsel eines inaktiven Attributs werden nicht beachtet. Dadurch kann in einem ersten Schritt die Menge an Attributen, deren Wert überhaupt beachtet bzw. mit Grenzen verglichen werden muss, reduziert werden.

Im nächsten Schritt müssen die Werte von aktiven Attributen, die sich verändert haben, mit den Grenzen der vorhandenen Prädikaten verglichen werden, um die Prädikate zu ermitteln, die sich zu `true` auswerten lassen.

Dazu wird zu jedem Attribut der zuletzt gemeldete Wert gespeichert. Weiterhin wird zu jedem Attribut ein Index erstellt, der die Intervalle aus den Grenzen der Prädikate enthält, die das jeweilige Attribut benutzen. Die Intervall-Grenzen b_{lower} und b_{upper} , die dem zuletzt gemeldeten Wert als nächstes stehen, werden noch einmal gesondert gespeichert (kann z.B. auch durch Zeiger innerhalb der Indexstruktur geschehen).

Sobald ein neuer Wert eines (aktiven!) Attributs gemeldet wird, wird dieser Wert mit den beiden Grenzen b_{lower} und b_{upper} verglichen. Falls der neue Wert keine dieser Grenzen überschritten hat, ist eine weitere Bearbeitung unnötig. Die gecachten Daten der Prädikate, die dieses Attribut benutzen, müssen nicht geändert werden, somit kann auch keine Nachricht ausgelöst werden.

Falls der neue Wert eine Grenze überschreitet, oder auf der Dimension des Attributs keine Ordnung vorliegt (z.B. auf der Haarfarbe), muss geprüft werden, welche Prädikate betroffen sind und ob sie durch den Kontextwechsel ihren (logischen) Wert ändern. Falls auf der Dimension des Attributs eine Ordnung vorliegt, so kann mit Hilfe des Indexes - insbesondere bei der Interval Skip List in $O(\log n)$ - ermittelt werden, welche Grenzen

überschritten wurden. Dazu ist noch eine vorherige Prüfung, ob es sich um eine einfache Veränderung des Wertes (z.B. Benutzer bewegt sich) oder um einen Sprung (z.B. Benutzer legt Batterien in sein GPS-Gerät ein) handelt. Bei einem Sprung muss ebenso vorgegangen werden, wie in dem Fall, dass keine Ordnung vorliegt: Es müssen alle Prädikate überprüft werden, die das Attribut benutzen.

Handelt es sich jedoch um eine einfache Veränderung, müssen nur die Prädikate überprüft werden, deren Grenzen überschritten wurden. Auch die Errechnung des Ergebnisses eines Prädikats kann in diesem Fall effizient erledigt werden. Beim Abschreiten des Index kann der Wert eines Prädikates immer dann invertiert werden, wenn eine Grenze überschritten wird (Das Verhalten beim Auftritt einer doppelten Grenzüberschreitung ist von der Implementierung abhängig, da es bedeutet, dass sich ein Wert so schnell verändert hat, dass die Auswertung und Versendung der Nachricht auf jeden Fall verzögert stattfindet. Eventuell könnte der Eintrittszeitpunkt interpoliert werden und die *time-to-live* der Nachricht überprüft werden).

Falls sich der Wert eines Prädikates nach dieser Überprüfung verändert hat, muss eventuell der Zeiger jedes Auslösers, in dem das Prädikat vorkommt, neu angepasst werden.

Es wird davon ausgegangen, dass der Fall der einfachen Veränderung bei einer bestehenden Ordnung der vorherrschende ist, somit lässt sich durch die oben beschriebene Vorgehensweise die Change Detection effizienter durchführen.

Durch das oben beschriebene Vorgehen ist es möglich, auch Kontextwechsel, die durch fehlerhaftes Data Cooking verpasst wurden, nachzuholen, da der Wert, den das Attribut bei korrektem Data Cooking angenommen hätte, auf dem Index überschritten wird, und so die Nachricht trotzdem noch ausgeführt werden kann. Falls sichergestellt ist, dass auch bei einem Sprung eine Grenze überschritten wurde (z.B. bei der Zeit, falls ein Gerät einige Zeit ausgeschaltet war), können so ebenfalls alle Nachrichten ermittelt werden, die evtl. nachträglich gesendet werden sollten.

3.3.3 Beispiel für den Auswertungsalgorithmus

Angenommen, im Nachrichtensystem seien zwei Nachrichten eingetragen, deren Bedingungen wie in Abbildung 10 dargestellt formuliert wurden. Der Kontext enthält neben dem Zeitattribut noch die (kontinuierlichen) Attribute A_0 bis A_3 . Die Bedingung von Nachricht M_0 wurde aufgespaltet in zwei Auslöser, da die Bedingung eine OR-Verknüpfung

enthält. Die Prädikate der Auslöser seien nach Update Ratio geordnet. Zum Zeitpunkt t sind nur die Prädikate P_2 und P_5 erfüllt. Die Menge der aktiven Prädikate enthält neben diesen beiden nur ihre Nachfolger in der Ordnung nach Update Ratio (das sind P_1 und P_4). Daraus ergibt sich, dass die Attribute A_0 und A_3 nicht aktive sind, da keine Prädikate, die A_0 oder A_3 enthalten, aktiv sind.

Nachrichten (bzw. deren Bedingungen):

$$M_0: [P_0 \wedge P_1 \wedge P_2] \vee P_3$$

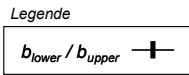
$$M_1: P_4 \wedge P_5$$

Auslöser:

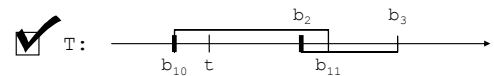
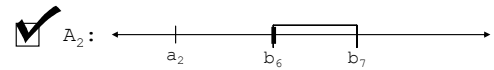
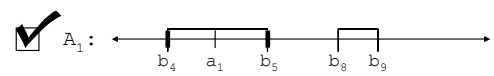
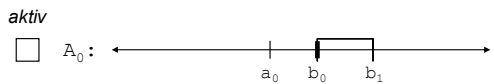
$$T_0: P_0 \wedge P_1 \wedge P_2$$

$$T_1: P_3$$

$$T_2: P_4 \wedge P_5$$



Attribute:



Auslöser, mit geordneten Prädikaten:

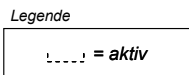
niedrig *Update Ratio* *hoch*

→

$$T_0: \dots P_2 \wedge P_1 \wedge P_0$$

$$T_1: \dots P_3$$

$$T_2: \dots P_5 \wedge P_4$$



Prädikate und zugehörige Attribute:

$$P_0: b_0 < A_0 < b_1 \quad P_3: b_6 < A_2 \leq b_7$$

$$P_1: b_2 \leq T < b_3 \quad P_4: b_8 \leq A_1 \leq b_9$$

$$P_2: b_4 < A_1 \leq b_5 \quad P_5: b_{10} < T < b_{11}$$

Werte der Prädikate zum Zeitpunkt t :

$$P_0: \text{false} \quad P_1: \text{false} \quad P_2: \text{true}$$

$$P_3: \text{false} \quad P_4: \text{false} \quad P_5: \text{true}$$

Abbildung 11: Beispiel für den Auswertungsalgorithmus: Zustand zum Zeitpunkt t

Wird vom Kontext-Repository nun ein Kontextwechsel in einem Attribut gemeldet, so muss nur dann reagiert werden, wenn dieses Attribut in der Menge $\{A_1, A_2, T\}$ liegt, da A_0 und A_3 inaktiv sind.

Falls reagiert werden muss, so müssen maximal zwei, im Fall des Attributes A_2 sogar nur eine Grenze überprüft werden.

3.3.4 Pseudo-Code für den Auswertungsalgorithmus

Nachfolgend wird der oben beschriebene Algorithmus in Pseudo-Code Notation angegeben. Die Methode `initialisierung()` erstellt die benötigten Strukturen auf Basis der vorhandenen Nachrichten, die Methode `update(A, value(A))` wird aufgerufen, sobald sich ein Attribut des Kontextes verändert hat. Die Methode `auswerten(changed[])` dient dazu, entweder nachdem alle veränderten Attribute erkannt wurden oder bei zweifacher Überschreitung einer Grenze alle auszulösenden Nachrichten zu ermitteln.

3.3.4.1 method void initialisierung begin

```
// Erster Schritt: Variablen leer initialisieren

Ms          // Menge aller Nachrichten im System
Bs          // Menge aller Bedingungen
Ts          // Menge aller Auslöser
Ps          // Menge aller Prädikate
As          // Menge aller Attribute

B(M)        // Bedingung einer Nachrichten
Ts(B)       // Menge der Auslöser einer Bedingung
Ps(T)       // Prädikate eines Auslösers

P           // Prädikat
blower(P)  // Untere Grenze des Intervalls
bupper(P)  // Obere Grenze des Intervalls
clower(P)  // Unterer Vergleichsoperator
cupper(P)  // Oberer Vergleichsoperator
Ps(b)       // Menge der Prädikate, zu denen eine Grenze
             gehört
A(P)        // Attribut, das in P verwendet wird

M(B)        // Nachricht, zu der eine Bedingung gehört
Bs(T)       // Menge der Bedingungen, zu denen ein Auslöser
             gehört
Ts(P)       // Menge der Auslöser, in denen ein Prädikat
             vorkommt
Ps(A)       // Menge der Prädikate, die ein Attribut
             benutzen

active(P)   // true, wenn Prädikat P aktiv ist
active(A)   // true, wenn Attribut A aktiv ist
value(A)    // zuletzt gemeldeter Wert von Attribut A
value(P)    // zuletzt errechneter Wert von Prädikat P
```

```

blower(A)    // größte Grenze kleiner als value(A)
bupper(A)    // kleinste Grenze größer als value(A)

// Zweiter Schritt: Strukturen erstellen

Ms = Nachrichten im System
for each M ∈ Ms do
    B = Bedingung von M
    B(M) = B          // Bedingung der Nachricht zuordnen

    Bs = Bs ∪ B
    M(B) = M         // Nachricht zu Bedingung speichern
    Ts(B) = Teile der disjunktiven Normalform von B
                  // entspricht Auslösern
    for each T ∈ Ts(B) do
        Ts = Ts ∪ T
        Bs(T) = Bs(T) ∪ B

        Ps(T) = Prädikate des Auslösers T
        for each P ∈ Ps(T) do
            // Prädikat eventuell umformen:
            P in P1, P2, P3 so umformen, dass Pi = b1 <1 Ai <2 b2
            P = P1; Ps(T) = Ps(T) ∪ {P2, P3}
            A = A1; As = As ∪ {A2, A3}

            Ps = Ps ∪ P
            As = As ∪ A

            Ts(P) = Ts(P) ∪ T
            A(P) = A
            Ps(A) = Ps(A) ∪ P

            blower(P) = Untere Grenze des Intervalls von P
            bupper(P) = Obere Grenze des Intervalls von P
            clower(P) = Unterer Vergleichsoperator von P
            cupper(P) = Oberer Vergleichsoperator von P
            Ps(bupper) = Ps(bupper) ∪ P
            Ps(blower) = Ps(blower) ∪ P
        od
    od

    Ps(T) Ordnen nach Update Ratio von A(P) | P ∈ Ps(T)
od

// Dritter Schritt: Verschmelzen
Gleiche Auslöser verschmelzen
(Ts, Ts(B) und Bs(T) anpassen)

```

Gleiche Bedingungen verschmelzen
(Bs, Bs(M) und Ms(B) anpassen)

// Vierter Schritt: Werte / Grenzen der Attribute
ermitteln

```
for each A ∈ As do
  value(A) = letzter Wert des Attributs A
  blower(A) = NULL oder max( -∞ ∪
    {blower(P) ≤ value(A) | P ∈ Ps(A)} ∪
    {bupper(P) ≤ value(A) | P ∈ Ps(A)} )
  bupper(A) = NULL oder min( ∞ ∪
    {blower(P) ≥ value(A) | P ∈ Ps(A)} ∪
    {bupper(P) ≥ value(A) | P ∈ Ps(A)} )
od
```

Ps Ordnen nach Ordnung der Ps in jeweiligen Ts
// In der Auswertung werden die Ps von links nach rechts
bearbeitet, so kann ein Auslöser nur dann true werden,
wenn alle Prädikate true sind

```
for each P ∈ Ps do
  value(P) = Wert des Prädikats P bei value(A)
od
```

// Fünfter Schritt: Aktivierungen ermitteln

```
for each T ∈ Ts do
  active = true
  for each P ∈ Ps(T) do // geordnet
    active(P) = active ∨ active(P)
    // Prädikat kann durch anderen Auslöser bereits
    // aktiviert worden sein!
    active = active ∨ value(P)
  od
od
for each A ∈ As do
  active(A) = logisches oder von {value(P) | P ∈ Ps(A)}
od
end
```

```

3.3.4.2 method void update(A, value(A)) begin
  if (active(A)) do
    if (value(A) < blower(A) ∨ value(A) > bupper(A)) do
      // Optimierte Auswertung überschrittener Grenzen
      ordnung = Ist auf A eine Ordnung gegeben?
      sprung = Ist die Attributs-Veränderung ein Sprung?
      changed(P | P ∈ Ps) = false
      // Hat sich Prädikat durch die Auswertung verändert?
      if (ordnung ∧ !sprung) do
        // Veränderte Ps seit letztem Update ermitteln
        Gs = Überschrittene aktive Grenzen
        for each G ∈ Gs do
          value(P) = !value(P)
          if changed(P) then do
            // Falls beide Grenzen von P überschritten
            wurden
            time-to-live = Auswertung übersprungener
                          Grenze sinnvoll?
            if (time-to-live) do
              auswerten(changed[])
              changed(P) = false
            od
          else
            changes(P) = true
          od
        od
      od
    else
      // Neu-Berechnung aller Prädikate, die A verwenden
      for each P ∈ Ps do
        oldvalue = value(P)
        value(P) = Wert des Prädikats P bei value(A)
        changed(P) = (oldvalue == value(P))
      od
    od
    call auswerten(changed[])
  od
end

```

```

3.3.4.3 method void auswerten(changed[]) begin
    // Veränderungen auswerten
    for each P ∈ { P | changed(P) ∧ active(P) } do
        for each T ∈ Ts(P) do
            Pn = Liste der Nachfolger von P im Auslöser T
            if (|Pn| == 0) do
                if (value(P))
                    Nachrichten M(B | B ∈ Bs(T)) auslösen
                    // Das letzte Prädikat ist true geworden
                else
                    if (value(P)) do
                        // Prädikat ist true geworden
                        active(Pn[0]) = true
                        // Direkten Nachfolger aktiv setzen
                    else
                        // Prädikat ist false geworden
                        active(Pn) = false
                        // Alle Nachfolger inaktiv setzen
                    od
                    Eventuell aktivierte oder inaktivierte As
                    ermitteln und evtl. deren bupper(A) und blower(A)
                    anpassen
                od
            od
        od
    od
end

```

3.3.5 Einschätzung des Algorithmus

Der Ablauf des Algorithmus ist stark von der Art des zugrunde liegenden Kontextes und seiner Änderungen bzw. der Menge der Nachrichten im System abhängig.

Der *Worst Case* tritt in folgender Konstellation ein:

- a) Alle Attribute sind aktiv. Das kann zum Beispiel der Fall sein, wenn genug triviale Nachrichten, die auf die unterschiedlichsten Attribute des Kontextes zugreifen, im System gespeichert sind.

- b) Es sind ständig fast alle Prädikate aktiv, d.h. es können keine (oder nur ganz wenige) Prädikate ausgeschlossen werden. Dieser Fall könnte eintreten, wenn zum Auslösen nur noch ein Prädikat benötigt wird, dessen Attribut seine Werte ständig (d.h. schneller als die meisten anderen Attribute) ändert, aber nie einen Wert annimmt, der das Prädikat auslöst.
- c) Zusätzlich finden in den Attributen häufig Sprünge statt, bzw. ist auf den Attributen keine Ordnung gegeben, so dass bei jeder Änderung statt b_{lower} und b_{upper} immer alle Grenzen der Prädikate ausgewertet werden müssen.

Aus b) lässt sich leicht ableiten, dass die Zeit als spezielles Attribut behandelt werden muss, da sie sonst in allen Nachrichten, die von einem Zeitintervall abhängen, eine schlechte Performanz erzeugt.

Der *Best Case* wird durch folgende Bedingungen begünstigt:

- a) Viele Nachrichten im System sind komplex, derart, dass in Ihren Bedingungen viele Prädikate mit \wedge verknüpft sind. Dadurch können die meisten Attribute ignoriert werden, nur die aussagekräftigsten Prädikate (d.h. die mit der geringsten Update Ratio) müssen den Großteil der Zeit über geprüft werden.
- b) Die Werte vieler Attribute, die häufig einem Kontextwechsel unterliegen, lassen sich interpolieren. So lassen sich viele weitere Attribute ignorieren, zum Preis der Unsicherheit, nicht alle Kontextwechsel zu erkennen, die natürlich gegen den Gewinn an Performanz abgewogen werden muss.

Die oben angesprochenen Bedingungen hängen stark von der Art der Nachrichten ab. Je spezifischer die Nachrichten, desto effizienter lassen sie sich auswerten.

Betrachtet man die für den aktiven Campus entwickelten Szenarien (siehe Anhang), so fällt auf, dass fast sämtliche Nachrichten Attribute mit dem Schlüsselwort `actuator` benutzen. Das bedeutet, dass nicht nur ein bestimmtes Attribut (z.B. `location`) überwacht werden muss, sondern eine ganze Reihe von Attributen des gleichen Typs. Um auch für diesen Fall eine effiziente Auswertung zu ermöglichen, sollten weitere Optimierungen in Betracht gezogen werden.

3.3.6 Weiterführende Optimierungen

Zusätzlich zu den oben angesprochenen Optimierungen können noch weitere Vereinfachungen angewendet werden, die aber eventuell zu einem indeterministischen Verhalten führen, da sie mit Interpolation arbeiten.

Die folgende Optimierung basiert auf der Annahme, dass sich die letzten Änderungen eines Attributs A_i ermitteln lassen. Aus diesen Änderungen kann man für die Attribute, auf denen eine Ordnung besteht ein geeignetes statistisches Mittel bilden.

Da weiterhin die beiden nächsten Grenzen b_{lower} und b_{upper} bekannt sind, kann man aus dem Abstand zur näheren Grenze und der mittleren Änderungsrate eine Zeit t berechnen, die auf den aktuellen Zeitpunkt addiert als virtueller Punkt für das Attribut Zeit eingetragen wird. Realisiert wird das durch ein neues Prädikat, das das Zeitattribut verwendet. Alle Prädikate, die das Attribut A_i verwenden, werden durch das neue Prädikat ersetzt. Attribut A_i kann nun deaktiviert werden, sodass Änderungen nicht mehr beachtet werden müssen. Bei Auslösung des neuen virtuellen Attributes wird es gelöscht und alle Prädikate, die A_i verwenden, werden wieder eingesetzt bzw. eine erneute Interpolation für A_i wird errechnet.

Bei dieser Vorgehensweise muss aber auch die Standardabweichung vom Mittelwert der Änderungsrate berücksichtigt werden, da eine zu große Abweichung eventuell zu einer Änderung des Prädikatswertes führen könnte, die wegen der Interpolation nicht oder erst viel zu spät ausgelöst wird.

Je nachdem, wie viel Logik im Kontext-Repository integriert ist, ist es auch möglich, diese Interpolation auszulagern und vom Repository den nächsten Zeitpunkt anzufordern, zu dem das Attribut den gesuchten Wert annimmt (möglichst unter Berücksichtigung eines Konfidenzintervalls).

Durch dieses Vorgehen lassen sich all jene Attribute sinnvoll ausblenden, die sich in regelmäßiger Art und Weise ändern. Besonders effizient ist der Einsatz bei häufigem Kontextwechsel in einem Attribut, der aber nicht zu einer Änderung der Prädikatswerte führt.

In dem oben genannten Beispiel (siehe Abbildung 11) könnte man so das Attribut A_2 ebenfalls inaktiv werden lassen, indem man die Distanz vom aktuellen Wert a_2 zur Grenze b_2 in eine Grenze auf der Zeitachse umrechnet.

Beispiele aus der Praxis:

- Falls ein Prädikat eine gewisse (hohe) Außentemperatur voraussetzt, so kann die Temperatur über den Winter hinweg interpoliert werden.
- Ein langer Fußweg eines Studenten benötigt keine genaue Beobachtung jeder Veränderung, falls der nächste relevante Grenzwert (mit größerer Wahrscheinlichkeit) um Dimensionen entfernt ist.

Eine weitere Optimierung betrifft Prädikate auf Attributen mehrerer ähnlicher Subkontexte, z.B. die `location` aller `persons` die bei Verwendung von `actuator.location` verglichen werden müssen. Falls in der Bedingung ein weiteres Prädikat `P` mit dem Schlüsselwort `actuator` verwendet wird, dessen Update Ratio unter der von `actuator.location` liegt, so kann, falls Prädikat `B` bei irgendeiner `person` nicht zu `true` ausgewertet werden kann, eine virtuelle Grenze in diesem Prädikat `B` eingefügt werden. Solange diese Grenze nicht erreicht wird, wird das Prädikat mit `actuator.location` für diese `person` nicht geprüft.

Ein Beispiel kann dem Szenario *baustelle* (siehe Anhang A) entnommen werden, in dem die `location` der Personen überprüft werden muss, deren `job = bauarbeiter` ist.

Diese *virtuellen Grenzen* lassen sich auf beliebige andere Fälle übertragen, verallgemeinert lassen sich dadurch Attribute deaktivieren, die sich häufiger als andere ändern. Ähnlich wie bei den Auslösern wird der Umstand genutzt, dass mehrere Attribute logisch von einander abhängig sind. Ziel ist es, das Attribut eines Objektes mit der kleinsten Update Ratio zu finden, das die anderen Objekte repräsentiert.

Vergleiche hierzu den Schlüssel einer Relation bzw. die Normalformen bei Datenbanken.

3.4 Ansatz zur Verwaltung des Kontextes

Im Zusammenhang mit dem Kontext des Aktiven Campus müssen hauptsächlich für zwei Problembereiche Implementierungen erarbeitet werden: Die Gewinnung der Kontextinformationen und die Speicherung und Auswertung der Sensordaten.

Ein Ansatz für ein System, das geeignet scheint, die Kontextinformationen zu speichern und Auswertungen zu generieren wird in [Ha02] beschrieben.

Das Hauptaugenmerk liegt dabei auf der Handhabung dynamischer Daten. Im System können Objekte samt zugehöriger Daten (hier als Aspekte und Facetten bezeichnet) im gesamten temporalen Verlauf gespeichert werden. Ein Aspekt kennzeichnet eine Eigenschaft eines Objektes, die sich einerseits durch mehrere Facetten, also messbare Größen, ausdrücken lässt, andererseits auch komplexe Prädikate (Funktionen) zulässt.

Ein Beispiel: Das Objekt `student` hat die Aspekte `alter` und `ort`. Der Aspekt `ort` selbst fasst die Facetten `position` und `geschwindigkeit` zusammen. Als Prädikate für `age` kommen die Vergleiche `olderThan` oder `youngerThan` in Frage.

Auf die gespeicherten Objekte kann über eine Anfragesprache (Y.A.Q.L., Yet Another Query Language) zugegriffen werden. Die Syntax ähnelt dabei der intuitiven Beschreibung des Ergebniswunsches der Anfragesprache S.Q.L. (Structured Query Language). Das Ergebnis kann durch Vergleiche eingeschränkt werden. Gleichzeitig lassen sich auf den gespeicherten Objekten Berechnungen durchführen. So wird der Zugriff auf die historischen Daten bzw. eine Interpolation ermöglicht. Eine Rückmeldung, die bei Kontextwechsel neue bzw. veränderte Daten meldet, ist im Modell vorgesehen, jedoch noch nicht implementiert worden.

Die Implementierung des Modells erlaubt die Anpassung an Daten jeder beliebigen Domäne, indem Methoden hinzugefügt werden können, die zur Analyse (Vergleich, Interpolation, etc.) dieser Daten nötig sind.

Um den Anforderungen für ein Kontext-Repository des Aktiven Campus zu genügen, müsste – neben der Benachrichtigung über Kontextwechsel und dem Sicherheitsaspekt – hauptsächlich der hierarchische Aufbau ergänzt werden.

4. Zusammenfassung

4.1 Rückblick

In dieser Arbeit wurde versucht, einen Grundstein zu legen, um die Idee des Aktiven Campus zu verwirklichen. Für dieses Projekt sind Ansätze aus vielen klassischen Bereichen der Informatik nötig, unter Anderem Augmented Reality, Datenbanken, Expertensysteme (KI), Wissensbasen, Rechnernetze und Rechnerkommunikation, etc.

Um den Aktiven Campus vollständig zu verwirklichen, ist es nötig, eine Zusammenarbeit zwischen all diesen Disziplinen zu erreichen.

Da der Aktive Campus in Hinsicht auf die möglichen bzw. nötigen Forschungsarbeiten ein sehr weites Feld darstellt, bieten sich aber an fast allen Stellen weitere Möglichkeiten zur Vertiefung, die in dieser Arbeit nicht behandelt werden konnten (siehe Kapitel 3.1 und Abbildung 8)

Bisher wurden folgende Ergebnisse erzielt:

- Die Anforderungen und Einsatzmöglichkeiten des Aktiven Campus wurden untersucht
- Es wurde ein Modell entwickelt, das alle Anforderungen abdeckt
- Ein Algorithmus, der die Kernfunktionalität des Systems gewährleistet, wurde erarbeitet

Diese Ergebnisse stellen grundlegende Ansätze dar, die zahlreiche vertiefende Arbeiten ermöglichen.

4.2 Verbesserungsmöglichkeiten

Folgende Verbesserungsmöglichkeiten liegen nahe:

- Empirische Ermittlungen könnten detailliertere Auskünfte (und somit mehr Möglichkeiten zur Optimierung) bringen über
 - die im Detail zu erwartenden Szenarien
 - Arten und Ausprägungen von Kontextinformationen

- Technische Geräte, die für die Verbindung mit dem Aktiven Campus genutzt werden könnten
- Anwendungen und Protokolle, die einen möglichst breiten Einsatz in der IT-Landschaft der Universität ermöglichen
- Die Heuristik für die Sortierung der Prädikate (in Auslösern und mehreren Objekten) kann erweitert werden. Aktuell findet eine fest verdrahtete Sortierung nach Update-Ratio statt, die aber ständig an die tatsächlichen Verhältnisse angepasst werden muss. Sehr zu empfehlen wäre eine Heuristik mit dynamischer Anpassung.
- Die Auswertung des Nachrichtensystems berücksichtigt eventuell vorhandene Filter noch nicht. Dies ist zum einen nötig, um die im Modell definierte Funktionalität zu Erfüllen, zum anderen könnten sich auch Optimierungsmöglichkeiten bieten, falls restriktive Filter das Überprüfen gewisser Attribute überflüssig machen (siehe Kapitel 3.2.5).
- Ein weiterer Punkt, der bei der Auswertung berücksichtigt werden muss, sind die Subkontexte, die mit einer spezieller Semantik belegt sind. Das sind zum Beispiel die Schlüsselworte in Nachrichten `actuator` und `message`. Obwohl oder gerade weil sie die Erstellung der Nachrichten sehr vereinfachen und mächtig machen, sind sie sehr schwer zu implementieren, da sie große Auswirkungen nach sich ziehen.
- Die Interval Skip List, die für den Auswertungsalgorithmus empfohlen wird, ist in der Standardausführung gerichtet. Da viele Dimensionen des Kontextes sich nicht monoton verhalten, wäre eine doppelt verkettete Interval Skip List sinnvoll, die in *einer* Struktur beide Richtungen unterstützt.
- Die zur Verwaltung des Kontexts geeignete *Context Presentation Engine* in [Ha02] muss wie in Kapitel 3.4 beschrieben an das hier präsentierte Modell angepasst und erweitert werden.
- Der Transport der Nachrichten muss gewährleistet werden. Es müssen die vorhandenen Medien ermittelt und kombinierbare Mechanismen für den Transport entwickelt werden.

- Ein wichtiger Bereich ist das Interface, durch das auf die vorhandenen Nachrichten und Filter zugegriffen werden kann. Es müssen Wege gefunden werden, dem Teilnehmer den umfassenden Überblick über alle Nachrichten und Filter sowie deren Einstellungen zu ermöglichen, um seinen Schutz vor unerwünschtem Nachrichtenversand zu gewährleisten.

4.3 Ausblick

Zusätzlich zu den hier bereits angesprochenen Problemen, Fragestellungen und Vertiefungen gibt es zahlreiche Erweiterungen, die für den Aktiven Campus interessant erscheinen. Beispiele sind:

- Eine automatische Nachrichtenerstellung könnte vorhandene Nachrichten bzw. den Kontext analysieren und selbständig das Verhalten des Benutzers voraussagen. Vergleichbar zum *Data Mining* aus dem Bereich Datenbanken.
Beispiel: Aus den Bewegungs-Routen eines Benutzers könnten seine Interessen abgeleitet werden.
- Durch automatisches *Clustering* (Bereich Datenbanken) könnten Gemeinsamkeiten im Kontext entdeckt werden.
Beispiel: Studierende, die ähnliche Vorlesungen belegen und in räumlicher Nähe lernen, könnten zu einer Lerngemeinschaft animiert werden.
- *Outlier Detection* (Bereich Datenbanken) ermöglicht es, den Teilnehmern automatisiert Alternativen aufzuzeigen.
Beispiel: Einer Person, die den Speiseplan der Mensa abrufen, könnte vom System auf ein Zeitfenster aufmerksam gemacht werden, in dem die Mensa von wenig Studenten besucht wird.

5. Literaturverzeichnis

- [Fo77] Charles Forgy, John P. McDermott. International Joint Conference on Artificial Intelligence (IJCAI): OPS, A Domain-Independent Production System Language, 1977
- [GoLe99] Andrew R. Golding und Neal Lesh. 3rd International Symposium on Wearable Computing: Indoor Navigation using a set of cheap, wearable sensors, 1999
- [Ha02] Florian Haftmann. Bachelor Thesis: Management Of Dynamic User Context, Juli 2002
- [HaJo94] Eric N. Hanson und Theodore Johnson. Selection Predicate Indexing for Active Databases Using Interval Skip Lists, April 1994

6. Anhang A: Szenarien

Auf den folgenden Seiten werden die Szenarien beschrieben, die der Anforderungsanalyse und Modellierungsphase zu Grunde lagen.

Ein Szenario wird im Wesentlichen durch folgende Informationen beschrieben:

- **Akteure:** Personen und Systeme, die am Szenario teilnehmen (aktiv oder passiv)
- **Kontext:** Art der Kontextinformationen, die benötigt werden
- **Flow of events:** Detaillierter Ablauf des Szenarios in einzelnen Schritten
- **Nachricht:** Eine oder mehrere Nachrichten, die die gewünschten Funktionalität ermöglichen
- **Filter:** Einer oder mehrere Filter, die die gewünschten Funktionalität ermöglichen

Für die Beschreibung der Nachrichten und Filter wird die in Kapitel 2.8 und 2.9 bereits eingeführte Syntax verwendet.

Szenario: baustelle
Beschreibung: Passanten werden vor Baustelle gewarnt.
Akteure: Passanten
Service
Kontext: Ort (Entfernung zur Baustelle)
Persönliche Daten (Ist Person ein Bauarbeiter oder nicht)

Flow of events:

1. Für eine Woche wird ein wichtiger Knotenpunkt im Gebäude wegen Bauarbeiten gesperrt.
2. Alle Personen, die nicht Bauarbeiter sind und sich diesem Knotenpunkt nähern, werden vom Service mit einer Nachricht daraufhingewiesen.

Nachricht:

Bedingung: (context.message.actuator.is_at_location(context.locations.baustelle_x)) AND
(NOT (context.message.actuator.is_at_location(context.locations.hoersaal_4) OR
context.message.actuator.is_at_location(context.locations.hoersaal_5))) AND
(NOT context.message.actuator.job == context.jobs.bauarbeiter)

Adressat: context.message.actuator

Inhalt: Achtung: Bauarbeiten im Durchgang zwischen Hörsaal 4 und 5! Bitte benutzen Sie einen anderen Durchgang!

Wirkung: BEEP, DISPLAY

Lebensdauer: context.functions.distance(context.message.creation, context.time) LE 1week

Wiederholung: context.functions.distance(context.message.send_to(context.message.actuator), context.time) GE 14h

Rückmeldung: NONE

Anmerkungen: Hörsaal 4 und Hörsaal 5 müssen vorher als Gebiet definiert werden (per Hand oder automatisch)

Szenario: **benachrichtigungAdressat**

Beschreibung: Ein Professor lässt sich per Nachricht über Ankunft eines Studenten informieren

Akteure: Professor
Student

Kontext: Zeit
Ort (Nähe zum Büro des Professors)
Persönliche Daten (Name des Studenten)
Physikalische Daten (Bewegungsumkreis der letzten 2 Minuten)

Flow of events:

1. Ein Professor X hat mir einem Studenten Y einen Termin um 14:00 ausgemacht.
2. Da er kurz vor dem Termin weg muss, richtet er eine Nachricht ein, die ihm signalisiert, wenn der Student länger als 2 Min vor seinem Büro wartet.

Nachricht:

Bedingung: (context.functions.distance(14:00, context.time) GE -15min) AND
(context.message.actuator.distance(context.persons.professor_x.office) LE 20m) AND
(context.message.actuator == context.persons.student_y) AND
(context.message.actuator.movement_radius(context.time, context.time - 2min) LE 10m)

Adressat: context.message.originator

Inhalt: Student Y wartet seit [context.time] vor Ihrem Büro!

Wirkung: BEEP, STORE

Lebensdauer: context.functions.distance(14:00, context.time) LE 30min

Wiederholung: false

Rückmeldung: NONE

Szenario: **benachrichtigungAdressat (fortgesetzt)**

Nachricht:

Bedingung: (context.functions.distance(14:00, context.time) GE -15min) AND
(context.message.actuator.distance(context.persons.professor_x.office) LE 20m) AND
(context.message.actuator == context.persons.student_y) AND
(context.message.actuator.movement_radius(context.time, context.time - 2min) LE 10m)

Adressat: context.message.actuator

Inhalt: Professor X hat das Büro für wenige Minuten verlassen. Bitte haben Sie ein wenig Geduld, vielen Dank!

Wirkung: BEEP, STORE

Lebensdauer: context.functions.distance(14:00, context.time) LE 30min

Wiederholung: false

Rückmeldung: [context.time]

Anmerkungen: In der ersten Nachricht wird nur der Professor aktiv informiert, in der zweiten Nachricht der Student aktiv, der Professor passiv durch die Rückmeldung.

Szenario: **benachrichtigungGruppe**

Beschreibung: Professor händigt Studenten Unterlagen aus.

Akteure: Professor
 Studenten

Kontext: Ort (Nähe zum Hörsaal)

Flow of events:

1. Ein Professor hält eine Vorlesung vor Studenten und möchte einige Dokumente elektronisch zur Verfügung stellen.
2. Alle anwesenden Studenten, auch die, die die Vorlesung nicht regelmäßig besuchen, bekommen das Dokument mittels einer Nachricht zugestellt.

Nachricht:

Bedingung: context.message.actuator.is_at_location(context.locations.hoersaal_x)

Adressat: context.message.actuator

Inhalt: Aktueller Foliensatz: [FILE <http://www.bayer.in.tum.de/fohlen.pdf>]

Wirkung: STORE

Lebensdauer: context.functions.distance(context.events.lecture_x.endtime(context.date), context.time) LE 15min

Wiederholung: false

Rückmeldung: COUNTER

Anmerkungen: **Keine**

Szenario: **benachrichtigungOrt**

Beschreibung: Vorlesungsteilnehmer werden über den Ausfall einer Vorlesung informiert

Akteure: Professor
Studenten

Kontext: Ort (Nähe zum Hörsaal)
Stundenplan (Hörer der Vorlesung)
Historie (Zur Vorlesungszeit im Hörsaal)

Flow of events:

1. Ein Professor ist kurzfristig verhindert.
2. Die Studenten, die in seiner Vorlesung sind (Vorlesung im Stundenplan oder mindestens 3 Mal besucht), werden benachrichtigt, dass die Vorlesung ausfällt.
3. Die Personen, die sich im Hörsaal aufhalten, werden auch benachrichtigt.

Nachricht:

Bedingung: (context.message.actuator.is_at_location(context.locations.hoersaal_y)) OR
(context.message.actuator.schedule.contains(context.events.lecture_y)) OR
(context.message.actuator.number_of_visits(context.events.lecture_y) GE 3)

Adressat: context.message.actuator

Inhalt: Vorlesung fällt aus, da Professor X krank ist.

Wirkung: DISPLAY

Lebensdauer: context.functions.distance(context.events.lecture_x.endtime(context.date), context.time) LE 5min

Wiederholung: false

Rückmeldung: COUNTER

Anmerkungen: Die besonderen Funktionen müssen vorher im Kontext definiert werden.

Szenario: **benachrichtigungRückmeldung**

Beschreibung: Einem Gastdozent wird zurückgemeldet, wer seine Nachricht erhalten hat.

Akteure: Studenten und wissenschaftliche Mitarbeiter
Gastdozent

Kontext: Persönliche Daten (Student oder wissenschaftlicher Mitarbeiter)

Flow of events:

1. Ein Gastdozent hält einen Vortrag in der Uni. Studenten und wissenschaftliche Mitarbeiter werden per Nachricht davon informiert.
2. Dem Gastdozent wird vom Service gemeldet, wieviele Personen die Nachricht in welchem Modus erreicht hat (sofortige Zustellung/Verzögerung).

Nachricht:

Bedingung: (context.message.actuator.job = context.jobs.student) OR
(context.message.actuator.job = context.jobs.wissenschaftlicher_mitarbeiter)

Adressat: context.message.actuator

Inhalt: Um 16:00 findet in 01.07.014 ein Gastvortrag von Prof. Goos über KI statt.

Wirkung: STORE

Lebensdauer: context.functions.distance(context.events.speech_x.starttime, context.time) LE -15min

Wiederholung: false

Rückmeldung: COUNTER

Anmerkungen: Keine

Szenario: benachrichtigungUmkreis

Beschreibung: Für das Studenten kino werden noch Besucher gesucht

Akteure: Studenten
Filmvorführer

Kontext: Zeit, Ort (Nähe zum Kino), Persönliche Daten (Person ist Student)

Flow of events:

1. Eine viertel Stunde vor Beginn der Vorstellung des Studentenkinos ist der Saal noch fast leer.
2. Der Vorführer läßt über einen Service eine Nachricht an alle Studenten schicken, die sich im näheren Umkreis aufhalten.
3. Ein Student besucht oft eine Übung, die in der Nähe des Saals zur gleichen Uhrzeit. Da er normalerweise nicht gestört werden will, wenn er eine Studienveranstaltung besucht, möchte er die Nachricht erst nach Ende der Veranstaltung bekommen.
4. Ein weiterer Student möchte überhaupt keine Nachrichten erhalten, die mit dem Kino zu tun haben.

Nachricht:

Bedingung: (context.message.actuator.job = context.jobs.student) AND
(context.message.actuator.distance(context.locations.cinema) LE 500m)

Adressat: context.message.actuator

Inhalt: Um 20:00 läuft in 01.07.014 der Film "Gladiator".

Wirkung: BEEP, DISPLAY

Lebensdauer: context.functions.distance(context.events.movie_x.starttime, context.time) LE -5min

Wiederholung: false

Rückmeldung: NONE

Filter:

Name: Kein Interesse am Kino

Verfall: false

Bedingung: context.message.originator.in_group(context.groups.public.cinemas)

Modifikationen: DENY

Szenario: **benachrichtigungUmkreis (fortgesetzt)**

Filter:

Name: Keine Störung während Vorlesung

Verfall: false

Bedingung: context.message.actuator.is_at_location(context.groups.public.locations.hoersaeale)

Modifikationen: context.message.impact = MIN(context.message.impact, STORE)

Anmerkungen: Um wirksam und einfach zu Filtern, sind Hierarchien und/oder Gruppen/Mengen für Absender, Adressat, Orte, ... nötig.

Szenario: **eigenerFilter**

Beschreibung: Student definiert Filter

Akteure: Student
Service

Kontext: Ort (Nähe zum Hörsaal)

Flow of events:

1. Ein Student betritt einen Hörsaal.
2. Sein mobiles Gerät zeigt ab sofort die Nachrichten nur noch mit einem grafischen Symbol, nicht mehr akustisch.

Filter:

Name: Keine Störung während Vorlesung

Verfall: false

Bedingung: `context.message.actuator.is_at_location(context.groups.public.locations.hoersaele)`

Modifikationen: `context.message.impact = MIN(context.message.impact, STORE)`

Anmerkungen: Hörsaal muss eine Sammlung von Orten sein, die vorher definiert wurde.

Szenario: fehlendesAttribut

Beschreibung: Bedingung kann nicht ausgewertet werden, da das falsche Gerät ausgewählt wurde

Akteure: Student
Laptop

Kontext: Proprietär (Gerätetyp, Druckgeschwindigkeit)

Flow of events:

1. Ein Student möchte über alle Drucker informiert werden, die in seiner Nähe sind.
2. Er erstellt eine Nachricht, die die Druckgeschwindigkeit aller nahen Drucker abfragen soll, wählt als Gerätetyp aber fälschlicherweise Laptop statt Drucker
3. Die Nachricht kann nicht ausgelöst werden. Da ein Laptop das Attribut „Druckgeschwindigkeit“ nicht unterstützt wird die Bedingung zwangsläufig zu NULL ausgewertet.

Nachricht:

Bedingung: (context.message.actuator.type_of_device == laptop) AND
(context.message.actuator.distance(context.participants.student_x) LE 500m) AND
(context.message.actuator.pages_per_minute GE 4ppm)

Adressat: context.message.originator

Inhalt: Der Drucker [context.message.actuator.name] unterstützt [context.message.actuator.pages_per_minute] Seiten pro Minute. Der Drucker befindet sich bei folgenden Koordinaten: [context.message.actuator.location]

Wirkung: DISPLAY

Lebensdauer: (context.function.distance(context.message.creation, context.time.now) LE 5mins)

Wiederholung: NOT (context.message.send_to(context.message.actuator))

Rückmeldung: NONE

Anmerkungen: Keine

Szenario: mensaBelegung
Beschreibung: Student wird von Mensa zur Cafeteria umgeleitet
Akteure: Student, Mensa-Service
Kontext: Ort (Nähe zur Mensa), Proprietär (Warteschlangenlänge Mensa)

Flow of events:

1. Ein Student verlässt den Hörsaal und begibt sich zur Mensa.
2. Die Mensa ist voll belegt und die Länge der Warteschlangen beträgt mindestens 20 Meter.
3. Der Student nähert sich der Mensa und erhält vom Mensa-Service eine Nachricht, die ihm empfiehlt, auf die Cafeteria auszuweichen.
4. Da der Student generell ungeduldig ist, folgt er dem Hinweis.

Nachricht:

Bedingung: (context.message.actuator.distance(context.locations.mensa) LE 500m) AND
(context.locations.mensa.occupancy GE 95%) AND
(context.locations.mensa.length_of_line GE 20m) AND
(context.locations.cafeteria.occupancy LT 95%) AND
(context.locations.cafeteria.length_of_line LT 20m)

Adressat: context.message.actuator

Inhalt: Die Mensa ist zu [context.locations.mensa.occupancy] belegt und die Warteschlangen sind mindestens [context.locations.mensa.length_of_line] Meter lang. Weichen sie wenn möglich auf die Cafeteria aus.

Wirkung: BEEP, DISPLAY

Lebensdauer: true

Wiederholung: context.functions.distance(context.message.send_to(context.message.actuator), context.time) GE 14h

Rückmeldung: NONE

Anmerkungen: Die Belegung/Warteschlangenlänge der Mensa kann entweder direkt per Sensor ermittelt oder per Funktion berechnet werden.

Szenario: mensaEmpfehlungen
Beschreibung: Student bekommt Essensempfehlung von der Mensa
Akteure: Student
Mensa-Kasse, Mensa-Service
Kontext: Ort (Nähe zu Speisen mit Nussanteil)
Persönliche Daten (Allergie auf Nüsse)
Historie (Bewegung des Studenten)

Flow of events:

1. Ein Student sucht sich in der Mensa ein Hauptgericht und mehrere Beilagen aus.
2. Da der Student allergisch auf Nüsse ist, erhält er - als er gerade längere Zeit vor der Beilage mit Nussanteil steht - eine Nachricht, mit dem Hinweis, dass die Beilage für ihn nicht geeignet ist.

Nachricht:

Bedingung: (context.message.actuator.distance(context.locations.mensa.critical_food(context.message.actuator)) LE 0.5m) AND
(context.message.actuator.movement_radius(context.time, context.time - 0.5min) LE 0.5m)

Adressat: context.message.actuator

Inhalt: Folgende Speisen sind für Allergiker ungeeignet: [context.mensa.kritische_speisen].

Wirkung: BEEP, DISPLAY

Lebensdauer: true

Wiederholung: context.functions.distance(context.message.send_to(context.message.actuator), context.time) GE 14h

Rückmeldung: NONE

Anmerkungen: Hier erfolgt keine direkte Abfrage auf die Nuss-Allergie, sondern eine generelle Abfrage auf kritische Speisen. Der Vorteil: Die kritischen Speisen des Tages können im Kontext dynamisch verwaltet werden. Falls keine Allergie vorhanden ist, wird die Bedingung wegen der dreiwertigen Logik (distance(NULL) = NULL) nicht erfüllt.

Szenario: mensaSpeiseplan
Beschreibung: Student erhält Speiseplan der Mensa
Akteure: Student
Mensa-Service
Kontext: Ort (Nähe zur Mensa)
Historie (Bewegung des Studenten)

Flow of events:

1. Ein Student verlässt den Hörsaal und begibt sich zur Mensa.
2. Auf dem Weg erhält er eine Nachricht vom Mensa-Service, die den aktuellen Speiseplan enthält.

Nachricht:

Bedingung: context.message.actuator.movement_target(context.time, context.time + 30min) = context.locations.mensa

Adressat: context.message.actuator

Inhalt: [FORMAT=HTML] Die heutige Speisekarte: [INCLUDE [http://mensa.in.tum.de/speisekarte.php?day=\[context.day\]](http://mensa.in.tum.de/speisekarte.php?day=[context.day])]

Wirkung: DISPLAY

Lebensdauer: true

Wiederholung: context.functions.distance(context.message.send_to(context.message.actuator), context.time) GE 14h

Rückmeldung: NONE

Anmerkungen: Die Art und Weise, wie ermittelt wird, ob sich ein Device der Mensa nähert, hängt von der Implementierung ab. Falls keine Vorhersage möglich ist, müssen eventuell Entfernung und/oder Geschwindigkeit benutzt werden. In diesem Fall benutzt die Funktion movement_target(time, time) eine Zeitspanne, um zu ermitteln, ob die Person in diesem Intervall an dem bezeichneten Ort eintreffen könnte.

Szenario: **nachrichtenPersistenz**
Beschreibung: Ein Gastdozent will ein Dokument ausdrucken.
Akteure: Gastdozent
Öffentlicher Drucker
Drucker-Service
Kontext: Ort (Nähe zum Drucker)
Persönliche Daten (Druckbefehl, Druckdaten)

Flow of events:

1. Ein Gastdozent hat sich aus dem WWW ein Dokument heruntergeladen und möchte es ausdrucken.
2. Er befindet sich in Reichweite (50 m) des öffentlichen Druckers 15 und des Personal-Druckers 13.
3. Das Dokument wird automatisch auf dem öffentlichen Drucker gedruckt.
4. Nach Beendigung des Auftrags wird der Dozent dorthin geleitet.

Nachricht:

Bedingung: `context.message.actuator.distance(context.participants.guest_referent_x) LE
MIN(context.participants.guest_referent_x .distance(context.tum.fmi.printer))`

Adressat: `context.message.actuator`

Inhalt: `[FILE /home/dokument.pdf]`

Wirkung: `PRINT`

Lebensdauer: `NOT context.message.sent`

Wiederholung: `false`

Rückmeldung: `context.message.actuator`

Szenario: **nachrichtenPersistenz (fortgesetzt)**

Nachricht:

Bedingung: context.message.actuator.distance(context.tum.fmi.printer.printer_x) GT 3m

Adressat: context.participants.guest_referent_x

Inhalt: [context.message.originator.location]

Wirkung: MAP_GUIDE

Lebensdauer: NOT context.message.sent

Wiederholung: false

Rückmeldung: context.message.actuator

Anmerkungen: Die zweite Nachricht wird als Reaktion des Druckers erstellt. Durch die Angabe der Lebenszeit wird die Nachricht nach dem ersten Senden gelöscht.

Szenario: navigationFreunde
Beschreibung: Student wird über die Anwesenheit seiner Freunde informiert.
Akteure: Student
Freunde
Service
Kontext: Ort
Physikalische Daten (Bewegung)
Persönliche Daten (Freunde des Studenten)

Flow of events:

1. Ein Student betritt die Uni. In der Haupthalle befinden sich einige seiner Freunde.
2. Der Student bekommt eine Nachricht, dass Bekannte anwesend sind und wo (Haupthalle, Vorlesung) sie sich aufhalten.

Nachricht:

Bedingung: (context.message.originator.distance(context.locations.university == INTERVAL(600m +/- 5m)) AND
(context.message.actuator.movement_radius(context.time, context.time - 5min) GT 15m) AND
(context.message.actuator.distance(context.locations.university) LE 600 Meter)

Adressat: context.message.originator

Inhalt: [context.message.actuator.name] befindet sich in Deiner Nähe, er ist [context.message.actuator.location].

Wirkung: BEEP, STORE

Lebensdauer: true

Wiederholung: context.functions.distance(context.message.send_to(context.message.originator), context.time) GE 5min

Rückmeldung: none

Anmerkungen: Sehr wichtig: Schutz vor "passiven" Nachrichten, die einen anderen Adressaten haben aber doch private Daten lesen/schicken!

Szenario: navigationTreffen

Beschreibung: Professor wird über die Verspätung eines Studenten, mit dem ein Treffen anberaumt war, informiert.

Akteure: Professor
Student
Service

Kontext: Ort
Physikalische Daten (Bewegung/Geschwindigkeit)

Flow of events:

1. Ein Professor hat ein Treffen mit einem Studenten um 15:00 vereinbart.
2. Der Student fährt mit dem Auto in die Uni und steht 5 Minuten vor dem Termin im Stau.
3. Sein Terminplaner erinnert ihn an den Termin (keine Gefahr, da keine Bewegung) und bietet ihm an, den Professor zu benachrichtigen.

Nachricht:

Bedingung: (context.participants.student_x.distance(context.locations.university) GT 1000m) AND
(context.time = INTERVAL(15:00h +-5min))

Adressat: context.participants.student_x

Inhalt: context.participants.student_x.schedule.meeting_z

Wirkung: REMIND

Lebensdauer: context.functions.distance(15:00, context.time) LE 5min

Wiederholung: false

Rückmeldung: NONE

Anmerkungen: Die Wirkung hängt vom Gerät ab, das die Nachricht empfängt. Es muss ggf. entschieden werden, welches von mehreren Geräten benutzt wird.

Szenario: univverwaltungFristen

Beschreibung: Ein Student, der gerade sich im 8 Semester befindet, wird über bevorstehende Frist der Anmeldung zur DVP informiert.

Akteure: Studenten
Universitätsverwaltungs-Service

Kontext: Persönliche Daten (Studienplan)

Flow of events:

1. Die Frist zur Anmeldung zur DHP läuft in einer Woche aus.
2. Jeder Student, der im 8.ten Semester (oder größer) ist, bekommt eine Meldung, in der er über die Frist informiert wird.
3. Ein Student läßt sich den Weg zum Prüfungsamt zeigen, um sich anzumelden, ein anderer lässt sich zu Informationen weiterleiten, welche Anforderungen er erfüllen muss.

Nachricht:

Bedingung: (context.functions.distance(context.fmi.dhp.due_date, context.time) LE 1week) AND
(context.message.actuator.semester GE 8)

Adressat: context.message.actuator

Inhalt: Frist zur DHP-Anmeldung läuft aus

Wirkung: STORE

Inhalt: [FILE=http://in.tum.de/dhpinfo.html]

Wirkung: STORE

Inhalt: [context.tum.administration.examination-office.location]

Wirkung: MAP_GUIDE

Lebensdauer: true

Wiederholung: context.functions.distance(context.fmi.dhp.due_date, context.time) LE 1week

Rückmeldung: none

Anmerkungen: Es ist möglich, eine Nachricht mit mehreren Wirkungen zu definieren.

Szenario: weiterleitungNachricht

Beschreibung: Ein Student läßt sich eine Nachricht eines Kommilitonen weiterleiten

Akteure: Studenten
Vorlesungs-Service

Kontext: Keiner

Flow of events:

1. Ein Student hat sich eine Nachricht x eingerichtet, die ihn informieren soll, wenn eine bestimmte Veranstaltung im Vorlesungsverzeichnis angekündigt wird.
2. Ein Kommilitone interessiert sich ebenfalls für diese Veranstaltung und läßt sich diese Nachricht weiterleiten, sobald sie eintritt.

Filter:

Name: Weiterleitung an Freund y

Verfall: false

Bedingung: context.message.id = x

Modifikationen: ADDRESSEE COPY TO context.participants.student_y

Anmerkungen: Keine

7. Anhang B: Index

aktiv (Attribut, Prädikat): 39

Attribut: 15

virtuelles: 34, 38

Auslöser: 37

Benutzer: 8

Best Case: 47

Dienst: 8

Event-Condition-Action: 20

Filter: 11, 23

Bedingung (condition): 23

Modifikation (modification): 23

Verfall (decay): 23

Grenzen: 37, 39

virtuelle: 49

Interpolation: 48, 50

Interval Skip List: 34

Kontext: 1, 4, 6, 14

-Repository: 6, 48, 50

-wechsel: 10

Moderator: 8

Nachricht: 10, 20

Adressat (addressee): 22

Auslöser (actuator): 18

Bedingung (condition): 20

Ersteller (originator): 22

Inhalt (content): 20

Lebenszeit (lifetime): 20

Rückmeldung (feedback): 22

Wiederholung (iteration): 20

Wirkung (impact): 21

Prädikat: 37

Rechtesystem: 19

Subkontext: 15

Szenarien: 27, 47, 49, 51

Time-to-live: 30, 40

Update Ratio: 38, 39, 40

Worst Case: 46

Y.A.Q.L.: 50