UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR MULTIMEDIALE
UND INTERAKTIVE SYSTEME

Direktor: Prof. Dr. rer. nat. Michael Herczeg

Erweiterung der Reasoning-Fähigkeiten des Context-Awareness-Frameworks CAKE

Extension of the Reasoning-Features of the Context-Awareness-Framework CAKE

Bachelorarbeit

im Rahmen des Studiengangs

Informatik

der Universität zu Lübeck

vorgelegt von

Kevin Böckler

ausgegeben und betreut von

Prof. Dr. rer. nat. Michael Herczeg

mit Unterstützung von:

Felix Schmitt, M.A.

Lübeck, 19. November 2013

Kurzfassung

Der Einsatz von intelligenten Mensch-Computer-Systemen gewinnt an immer mehr Bedeutung. Dabei erhöht sich stetig der Anspruch nach intelligenter Unterstützung bei Aufgaben, Vorhersage von Ereignissen oder das Einschätzen von Mitmenschen in der unmittelbaren Umgebung. Zu diesem Zweck entstand am Institut für multimediale und interaktive Systeme (IMIS) der Universtität zu Lübeck das sogenannte Context Aware Knowledge- and sensorbased Environment (CAKE). Dieses in Java entwickelte Framework bietet eine Plattform zur Verwaltung von Benutzerdaten und verwendbarer Sensorik und Aktuatorik, um mit Hilfe von logischen Schlüssen den Benutzer ambient in seinen Aufgaben zu unterstützen. Zu diesem Zweck wird das Framework um zusätzliche Reasoning-Fähigkeiten erweitert. Diese sollen eine höhere Aussagekraft liefern und zudem beliebige Anwendungsdomänen schnell an CAKE adaptierbar machen. Neben einigen Ansätzen zum Reasoning beschreibt diese Arbeit das Maschinelle Lernen (ML), welches heutzutage überwiegend in medizinischen oder biologischen Bereichen Anwendung findet, aber dennoch bereits größere Bedeutung in verschiedenen Computersystemen erfährt. Durch den aktiven Lernprozess bietet es eine gute Verwendung für CAKE im Hinblick auf große Sensor-Datenmengen. In dieser Arbeit wird das Weka-Framework zur Umsetzung von ML-Verfahren verwendet und in die CAKE-Architektur eingebettet und adaptiert. In dem Zusammenhang finden Beschreibungen von ML-Verfahren statt, wie Naïve Bayes, Nearest-Neighbour-Suchen oder künstliche neuronale Netze. Weka implementiert diese Algorithmen und wird im Verlauf des Entwicklungsprozesses den Spezifikationen von CAKE entsprechend verwendet. Diese Arbeit beschreibt Aufgaben und Eigenschaften, die in der Verwendung eines Reasoners mit CAKE relevant sind und konzipiert einfache Ansätze zur Modellierung von Sensoren und Aktuatoren in ambienten Domänen. Das umgesetzte Maschinelle Lernen in CAKE wird abschließend mit Beispielszenarien evaluiert und technische Eigenschaften erhoben. Dabei werden Geschwindigkeiten in Reasoner-Erzeugung und Reasoning mit Sensorwerten ermittelt und miteinander verglichen.

Schlüsselwörter

Ambiente Systeme, Context Awareness, Benutzererfassung, Künstliche Intelligenz, Maschinelles Lernen, Sensoren, Aktuatoren, Reasoning

Abstract

There has been much interest in intelligent human-computer-systems for the last years. At the same time more and more people rely on intelligent assistance for managing everyday tasks, forecasting events or calculating the position of fellow men in the immediate vicinity. On this purpose the Institut für multimediale und interaktive Systeme (IMIS) at the Universität zu Lübeck developed the Context Aware Knowledge- and sensorbased Environment (CAKE). This framework is written in Java and provides a platform for holding user accounts and processing data from sensors to actuators in order to support the user in an ambient way based on logical conclusions. To achieve this goal additional reasoning-features will be developed throughout this thesis. These features shall deliver more reasonable conclusions and new domains of application being adapted to CAKE in less time and without complex operations. This paper describes some approaches to reasoning followed by the main approach of Machine Learning (ML), which nowadays is predominantly used in biological and medical applications while there is increasing interest in several computational domains. Because of the ability to directly learn from input data this approach fits well into the context of CAKE. This thesis describes the adaptation of the Weka-Framework to the CAKE architecture to implement different ML-techniques. In addition to that some of these techniques are explained such as Naïve Bayes, Nearest-Neighbour-Searches or artifical neural networks, which have been implemented by Weka. In the process of development these implementations are adapted to the interfaces of CAKE and therefore will be used for reasoning. Furthermore the user's tasks and properties are described, which define the interaction with a specific reasoner with CAKE. Also there are concepts of modelling the interrelation between sensors and actuators in ambient domains. After implementing the ML-reasoner in CAKE there will be an evaluation using samples of data to determine technical features. Additionally the speed of creating a reasoner instance or using it for reasoning is tested and compared to each other.

Keywords

Ambient Systems, Context Awareness, User Recognition, Artificial Intelligence, Machine Learning, Sensors, Actuators, Reasoning

Inhaltsverzeichnis

1 Einleitung

Die Unterstützung von Computersystemen im Alltag gewinnt an immer mehr Bedeutung. Den Menschen bei Tätigkeiten zu unterstützen, ihm Hinweise und Vorschläge zu Entscheidungen zu geben oder Personen im Arbeitskontext untereinander Kontakt zu ermöglichen - all diese Aufgaben übernehmen mehr und mehr ambiente Computersysteme, die den Menschen sowohl im öffentlichen als auch im privaten Bereich begleiten.

Im Folgenden wird dieses Thema genauer beschrieben. Anschließend werden die Ziele der Arbeit angeführt, gefolgt von dem aktuellen Stand der Technik zum Thema Reasoning und zur derzeitigen Ausstattung des Context Awareness Frameworks. Zuletzt schließt ein kurzer Überblick über die einzelnen Kapitel dieser Arbeit an, der die Strukturierung und Arbeitsweise darlegt.

1.1 Thema der Arbeit

Die Anwendung von CAKE findet im Bereich von Context Awareness statt. Heutzutage wird permanent daran gearbeitet, Arbeitsprozesse, die zunehmend komplexer werden, zu strukturieren, in Teilaufgaben aufzuspalten und für den jeweiligen Bearbeiter so angenehm und effektiv wie möglich zu gestalten. Damit dies möglich ist, wird viel Zeit in die Analyse von Systemen gesteckt - und in die Analse der Benutzer. Benutzerzentrierte Systementwicklungsprozesse haben das Ziel, gebrauchstaugliche Software zu fertigen - das sogenannte Usability Engineering.

Neben dem Ziel der Arbeitsprozessoptimierung finden sich auf dem privaten Anwendungssektor viele Neuerungen. Multimediale Geräte wie Smartphones oder Tablets werden zu stetigen Begleitern im Alltag. Es entstehen dabei immer mehr Anforderungen an die Software solcher Produkte, denn die Verwendung dieser Geräte wächst zunehmend an ambienter und tangibler Mensch-Computer-Interaktion. Nutzer haben den Anspruch, dass Geräte ihre Sprache erkennen und interpretieren, Musiktitel wiedererkennen, Bilder zuordnen und viele weitere automatisierte Vorgänge mehr.

Es finden sich auch Anwendungen der Context Awareness wieder, wenn es darum geht, Menschen zu beobachten und zu überwachen. Hierbei dient die Überwachung aber primär der Hilfestellung, Fehler im Alltag zu vermeiden, bessere Lösungen zu finden oder geeignete alarmierende Prozesse einzuleiten, sollten Menschen sich mit Schwierigkeiten konfrontiert sehen, die sie selbst nicht richtig

erkennen oder lösen können.

Das CAKE[1]-Framework zielt darauf ab, Context Awareness in verschiedenen Domänen zu realisieren und genau solche zuvor beschriebene Situationen zu optimieren. In dem gesamten Projekt, zu dem diese Arbeit modulweise gehört, geht es darum, reale Kontexte von Benutzern zu erfassen. Hierzu dienen sowohl echte Sensoren wie Kameras, Mikrophone, tangible Schalter oder Lichtschranken, als auch versteckte Sensormodule wie GPS-/Ortstracker, Interaktions-Analysetools für die Benutzung von interaktiven Mensch-Computersystemen oder auch RFID-Scanner. Zusammen mit diesen Sensoren ist es die Aufgabe von CAKE, zuvor festgelegte Schlüsse über die erfassten Benutzer zu bilden und diese für andere Systeme (Aktuatoren) bereitzustellen, die dann letztendliche sichtbare Schritte einleiten (wie z.B. Auslösen von Alarm, Anbieten von Hilfe, Abspielen von Signalen).

Diese Arbeit befasst sich mit einem Teilmodul des gesamten Projekts: dem Reasoner. Die Logikschicht des CAKE-Systems besteht aus idealerweise mehreren Reasonern, die sich ausschließlich damit beschäftigen, empfangene Sensorwerte zu präzisen Aussagen oder Schlüssen zu verarbeiten. Diese Berechnungen, Reasoning genannt, können auf verschiedenste Arten realisiert sein. Dazu gibt es kontextabhängig verschiedene Ansätze, wie ein Problem gelöst werden kann. Analog dazu wird das Wissen über die Welt modelliert und für das Reasoning geeignet vorgehalten. Das Zusammenspiel von modellierter Welt, Eingang von Sensorwerten und Zielsetzung der anfragenden Aktuatoren machen die Komplexität des Reasoners aus.

1.2 Ziele der Arbeit

In dieser Arbeit wird das Context Awareness Framework CAKE durch einen zusätzlichen Reasoner erweitert. Um später eine Featureliste ermitteln zu können, werden folgende grundsätzliche Ziele an diese Arbeit gestellt.

Neue Anwendungsdomänen. Der bisherige Reasoner wurde direkt für die Büro-Domäne entwickelt bzw. implementiert. In dieser Arbeit soll ein Reasoner entstehen, der verschiedene und insbesondere neue Domänen abdeckt. Das Hauptaugenmerk liegt darauf, Domänen leicht neu beschreiben oder modellieren und somit den neuen Reasoner effektiv für mehrere Domänen einsetzen zu können.

Mächtigkeit des Reasonings. Der Leistungsumfang des neuen Reasoners soll steigen. Mit einer größeren Auswahl an Domänen kommen auch mehrere Reasoningziele einher, sodass z.B. nicht nur einfache Entscheidungen getroffen, sondern eventuell beliebige Zahlenwerte vorausbestimmt werden können. Die aus der Arbeit resultierende Funktionalität wird ermöglichen, Sensorwerte als vielfältige und verschiedenste Datentypen (u.a. diskrete Arrays, Integer, Fließkommazahlen) zu verwenden und dem Anwendungskontext entsprechend komplexe Resultate zu berechnen. Um das CAKE Framework effektiver zu gestalten, kann es hilfreich sein, für ein gegebenes Entscheidungsproblem mehrere (technisch unterschiedliche) Reasoner zu befragen. Auch können Reasoner sich untereinan-

[1] Context Aware Knowledge- and sensorbased Environment

der referenzieren und als weitere Sensorwerte betrachten. So könnte der bereits von Wilken (2012) entwickelte Activity-Reasoner wertvolle zusätzliche Eingabewerte (Zustände der aktuellen Aktivität) für eine andere Entscheidungsfrage beinhalten.

Einfache Adaptierbarkeit. Der neue Reasoner soll für verschiedene Kontexte eingesetzt werden. Es muss damit gewährleistet sein, dass diese Komponente angemessen schnell konfiguriert und an eine Anwendungswelt angepasst werden kann. Modellierung und Zielsetzung des Reasoners müssen somit problemlos in CAKE realisierbar sein. Der Reasoner wird diesen Anforderungen zur Adaptierbarkeit gerecht und ermöglicht diese ohne aufwändiges Re-Engineering oder Re-Factoring am Programmcode.

Lernen zur Laufzeit. Sehr wichtig für die Arbeit mit Systemen im Rahmen von Context Awareness ist es, dass diese stets dazulernen und ihre Aussagefähigkeit erweitern und Fehleranfälligkeit reduzieren. Insofern liegt ein bedeutsames Ziel darin, dass der neue Reasoner in der Lage ist, mit neuen Inhalten und Konfigurationen in seiner Modellierung umzugehen und mit steigender Laufzeit ein größeres Wissen über die modellierte Welt zu besitzen. Abgrenzend bleibt zu erwähnen, dass der Reasoner durch den Lernprozess keine neuen Fragen beantwortet, sondern nur die Wahrscheinlichkeit einer richtigen Antwort erhöht, also weniger Falschaussagen produziert als ein Reasoner, der zur Laufzeit kein Wissen ansammelt.

1.3 Stand der Technik

Für die Arbeit sind vorhergegangene Arbeiten und Erkenntnisse sowie Methoden zur Problemlösungen von großer Bedeutung. Der Stand der Technik ist besonders relevant im Bereich des Reasonings, welches zunächst beschreiben wird. Der anschließende Abschnitt führt an, welche Module und Ansätze bereits im direkten Umfeld dieser Arbeit entwickelt wurden, also welche Systeme von CAKE bereits einsatzbereit vorliegen.

1.3.1 Reasoning

In diesem Abschnitt führe ich einige Arten von Reasoningstrategien auf und führe Modellierungen an, die verschiedene Kontexte auf geeignete Art und Weise für die Wahl bestimmter Reasoner beschreiben.

Ontologiebasiertes Reasoning. Mit dem Reasoning der Context Awareness in verschiedenen Domänen beschäftigen sich Hu et al. (2012) mittels kombinierter Ontologien. Demnach sei Context Awareness denkbar im Bereich von Kommunikation, Verkehr oder auch Agrarwirtschaft. Das Konzept basiert darauf, zwei Level der Abstraktion zu modellieren, wobei eine Meta-Ebene sehr abstrakt gehaltene Artefakt oder Personentypen, so genannte Entitäten, darstellt. Eine grobe Visualisierung bietet Abbildung 1. Diese Ebene lässt sich nun an eine domänenspezifische Ebene der Ontologie an-

knüpfen. Diese konkretere Ontologie wird für jede Anwendungsdomäne entwickelt und verfeinert die Entitäten um tiefere ontologische Verzweigungen. Somit können Aktivitäten und Beziehungen zwischen Entitäten erst in der Domain-spezifischen Ebene beschrieben werden. Die Ontologien werden für das Reasoning in eine Beschreibungssprache, hier SWRL[2] - vergleiche auch Parsia et al. (2005) - übersetzt und mit Hilfe von Abfragesprachen (wie hier benutzt: SPARQL[3]) zu Ergebnissen der jeweiligen Reasoninganfrage verarbeitet.

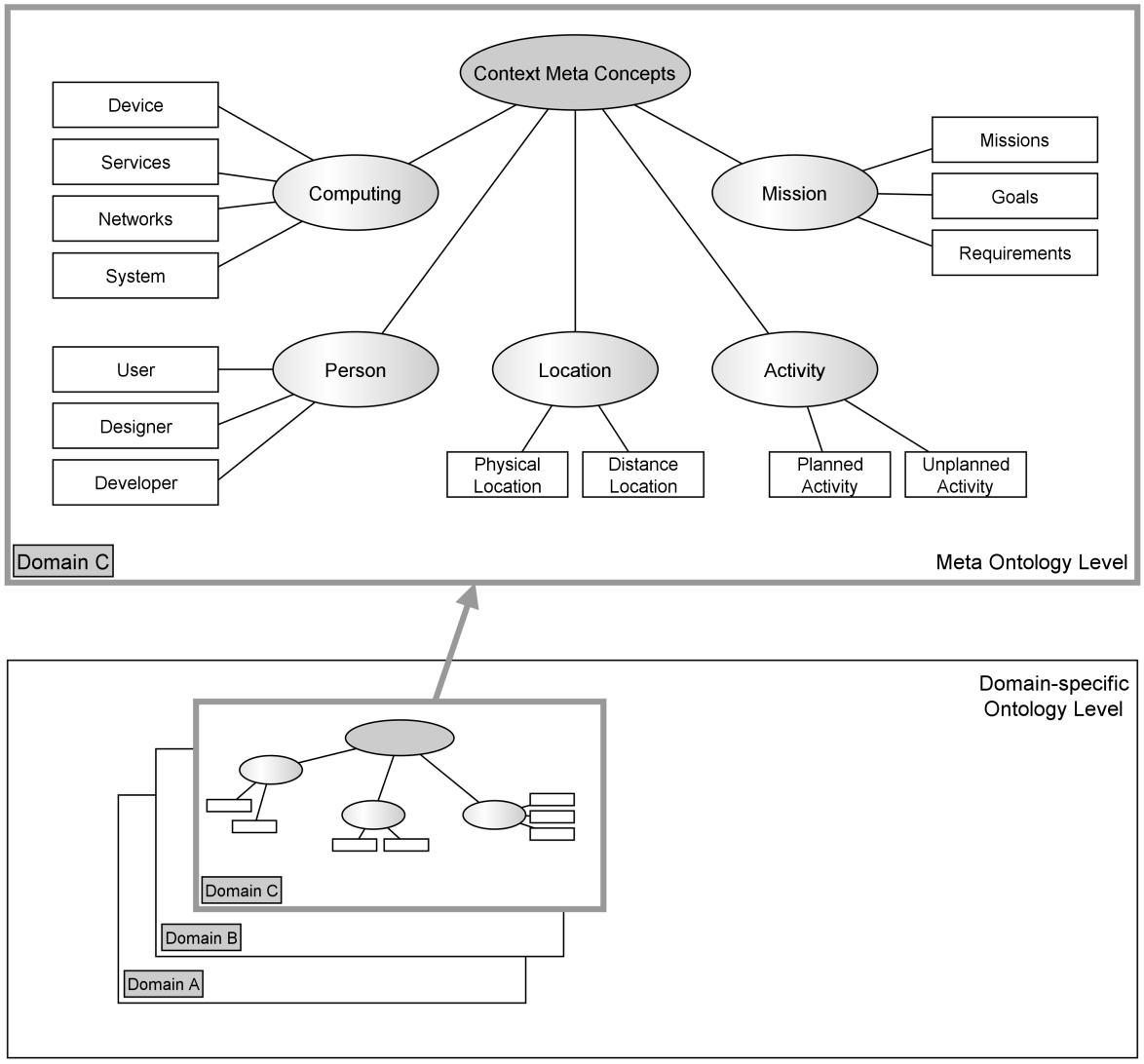

Abbildung 1: Ontologiebasiertes Reasoning - Abstraktionsebenen nach Hu et al. (2012)

Machine Learning. Den Bereich des Machine Learning beschreiben Schmitt et al. (2008) auf technischer Ebene. Neben Grundlagen für Algorithmen werden Online Learning Ansätze angeführt. In solchen Verfahren werden Datensätze durch Sensoren erfasst und in das temporäre Model des Reasoners gespeichert. Aufgrund diese Datensätze legt das System mathematisch stochastische Berechnungen

[2] Semantic Web Ontology Langage
[3] SPARQL Protocol and RDF Query Language

zugrunde, um Entscheidungen zu abgefragten Datensätzen zu treffen. Diese Art Reasoning bedient sich einem großen eingehenden Fluss an Eingangswerten, die für stochastische Berechnungen des Models auswertbar sind. Mangelt es an einem steten Datenstrom, so lassen sich die Sensorwerte auch in einem Model langfristig speichern. Hierbei spricht man von Offline Learning. Um trotzdem einen Lernprozess zu involvieren, wird das mathematische Model immer wieder neu erzeugt - auf Grundlage aller gespeicherter Datensätze in einer Datenbank. Der mögliche Lebenszyklus des Reasoningmoduls im Hinblick auf Revalidierung zeigt Abbildung 2. Hierbei können selektive Teilbereiche der Datenbank (z.B. immer nur die neuesten Wertepaare) bis hin zu allen Datensätzen für das Generieren des Models verwendet werden.

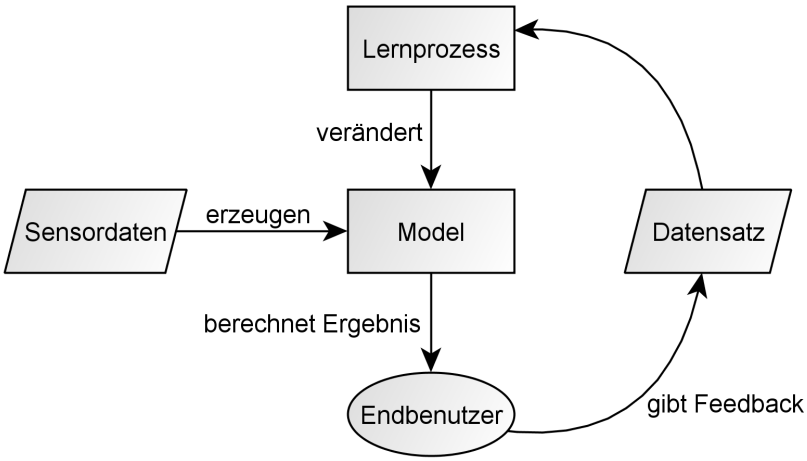

Abbildung 2: Machine Learning - Ablauf des Lernprozesses nach Schmitt et al. (2008)

Case Based Reasoning. Einer der bisher im vorangegangenen MATe[4]-System betriebenen Reasoner ist der von Wilken (2012) eingeführte und beschriebene CBR[5]-Reasoner. In einem iterativen Lernprozess wird versucht, Sensorwerte in Form eines Tupels in der bestehenden Datenbank von gelösten Problemen des CBR-Reasoners zu suchen. Nach bestimmten vordefinierten Regeln wird hier der am besten zum aktuellen Problem passende Fall herausgesucht, der als gelöst gilt. Danach wartet das System auf eine Rückmeldung, ob die Entscheidung korrekt war, schlägt bei Misserfolg eine Alternativlösung vor oder verwirft das Problem, sollte keine Lösung gefunden oder durch den Benutzer manuell zurückgegeben werden. Bei Erfolg wird das Problem mit den gegebenen Wertepaaren als gelöst eingestuft und steht künftigen Reasoning-Anfragen zur Verfügung.

Rule Based Reasoning. Ein etwas anderer Ansatz als das zuvor betrachtete CBR stellt das RBR[6] dar. Im CAKE-System liegt aktuell als einziger lauffähiger Reasoner der Rule Based Reasoner vor. Dieser wird anhand einer Menge von Regeln betrieben, die mit Hilfe einer bestimmten Beschreibungssprache definiert sind und dazu dienen, aufgrund von bestimmten Eingangsvariablen vorher definierte Ausgangsvariablen zu bestimmen. Dabei werden diese Variablen mit grundlegenden Abfragen geprüft,

[4]MATe for Awareness in Teams
[5]Case Based Reasoning
[6]Rule Based Reasoning

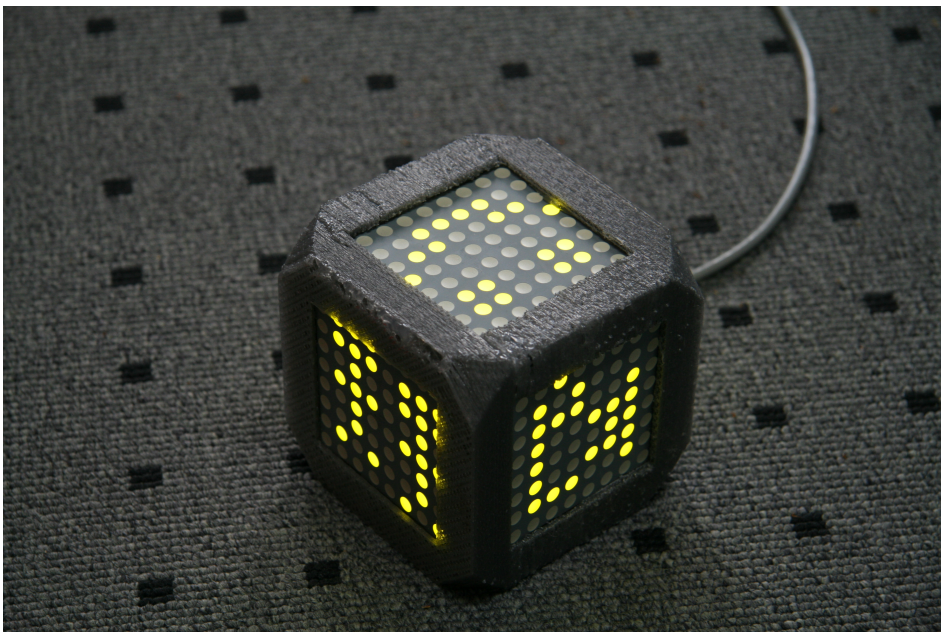

Abbildung 3: Der Cube zeigt den aktuellen Status auf der oberen Seite an. Bildquelle: MATe

wie z.B. mit `IF/ELSE`-Bedingungen und einfachen mathematischen Vergleichen. Einen Reasoningansatz, der CBR und RBR miteinander verknüpft nutzt, stellen Ranganathan & Campbell (2003) vor.

1.3.2 CAKE Bestand

Im CAKE Framework stehen bereits echte, funktionsfähige Sensoren zur Verfügung, mit denen Daten zur Kontexterfassung erhoben und in der Logikebene von CAKE verarbeitet werden können. Kindereit et al. (2011) stellen den Cubus vor, welcher gleichzeitig als Sensor und Aktuator des von Wilken (2012) entwickelten Activity-Reasoners agiert - siehe Abbildung 3. Hierbei werden sechs verschiedene Aktivitäten dargestellt: `break_long`, `break_short`, `reading`, `writing`, `meeting` oder bei Unkenntnis der Aktivität `unknown`. Die Interaktion geschieht hierbei, indem der Benutzer seinen aktuellen Beschäftigungsstatus, der auf einer der sechs Würfelseiten des Cubus angezeigt wird, so ausrichtet, dass dieser auf dem Würfel nach oben zeigt. Der Benutzer dreht den Würfel also von Hand, um eine Interaktion zu vollziehen. Ermittelt CAKE den Status korrekt, so erscheint der Aktivitätsstand auf der nach oben ausgerichteten Würfelseite ohne Zutun des Benutzers.

Neu hinzugekommen ist der von Auwetter & Thomsen (2013) im Rahmen eines Praktikum entstandenen Non-cubic Cube, welcher ebenso als Sensor und Aktuator dient. Das Artefakt ist hierbei eine würfelförmige Laterne, welche ihre Werte abstrakt durch Darstellung verschiedener Farben zum Ausdruck bringt. Diese Farben lassen sich im RGB-Farbraum zusammenstellen und beschränken sich nicht auf diskrete Werte. Interaktionen lassen sich mit der Hand ausüben, indem man in den Innenraum der Laterne greift und dort Handbewegungen und -drehungen ausführt, die von einer Kamera

analysiert werden.

Zusätzlich beschreiben Kindereit et al. (2011) verschiedene MATe-Sensoren in einer Zusammenfassung. Darunter ist der DAA[7], welcher die aktive Anwendung des Computernutzers (Brower, Schreibprogramm) ermittelt und ebenso Echtzeit-Statistiken erfassen kann wie die Tippgeschwindigkeit (Tasten pro Minute) oder Inaktivität in der Nutzung der offenen Anwendungen.

Um Personen im Raum zu erkennen, wird das entwickelte *MikeRophone* angeführt. Dieses besteht aus Mikrofon und Erkennungssoftware und dient dazu, die CAKE-Nutzer über ihre Stimmen im Raum zu identifizieren. So kann die Anwesenheit von Benutzer ermittelt und des Weiteren allgemein eine Anzahl an anwesenden Personen vermutet werden.

Im bisherigen Bürokontext war es von grundsätzlicher Bedeutung, ob eine Bürotür geöffnet oder geschlossen ist. Hierzu wurde ein Sensor entwickelt, der den Öffnungszustand der Tür in den beiden Varianten anzeigte: `geöffnet` oder `geschlossen`.

Neben den Sensoren entwickelte Wilken (2012) den bereits erwähnten Activity-Reasoner, der die weiter oben beschriebenen sechs Zustände der aktuellen Aktivität eines Benutzers ermittelt. Dieser Reasoner basiert auf der Grundlage des CBR und liegt derzeit nur konzeptuell in der Logikschicht von CAKE vor. Aktuell befindet sich der RBR-Reasoner von Bouck-Standen et al. (2013) in Verwendung, der die logische Verarbeitung in CAKE mittels regelbasiertem Reasoning übernimmt.

Zusätzlich zeigt Ruge (2010) eine Möglichkeit, den Unterbrechbarkeits-Zustand im Bürokontext zu messen. Mit einem ontologiebasierten Reasoner ergeben sich messbare Formen der Unterbrechbarkeit, durch die Tendenzen (steigende oder fallende Unterbrechbarkeit) ermittelt werden.

1.4 Vorgehensweise

Die Entwicklung des Reasoners durchläuft mehrere Phasen. Zunächst wird auf Grundlage der Analyse eine Anforderungsliste an Funktionalitäten sowie Architektur festgelegt. Anschließend wird in einem iterativ inkrementellem Entwicklungsprozess dieses System umgesetzt. Abbildung 4 zeigt diesen Prozess. Dieses Prototype-ähnliche Entwickeln ermöglicht das stetige Validieren der Umsetzung von Features und Anforderungen, die in einem stetigen Prozess der Formativen Evaluation angepasst werden können. Die resultierenden Änderungen in der Umsetzung sind so leicht vorzunehmen.

Analysiert werden in Kapitel 2 zwei wesentliche Punkte: zum einen die Zielsetzung des Reasoners, zum anderen die Zielarchitektur von CAKE und die Einbindung in den Systemkontext. Die genaue Modellierung und Umsetzung von Schnittstellen bezüglich CAKE und des Reasoners werden in Kapitel 3 beschrieben. Zusätzlich wird die Konzeption des Reasoner-Systems im Hinblick auf Arbeitsweise und technische Funktionsweise beleuchtet. Die tatsächliche Umsetzung bzw. die abstrakte und konkrete Realisierung im Quellcode von CAKE wird in Kapitel 4 beschrieben. Das Ziel dieses Ka-

[7] Desktop Activity Analyser

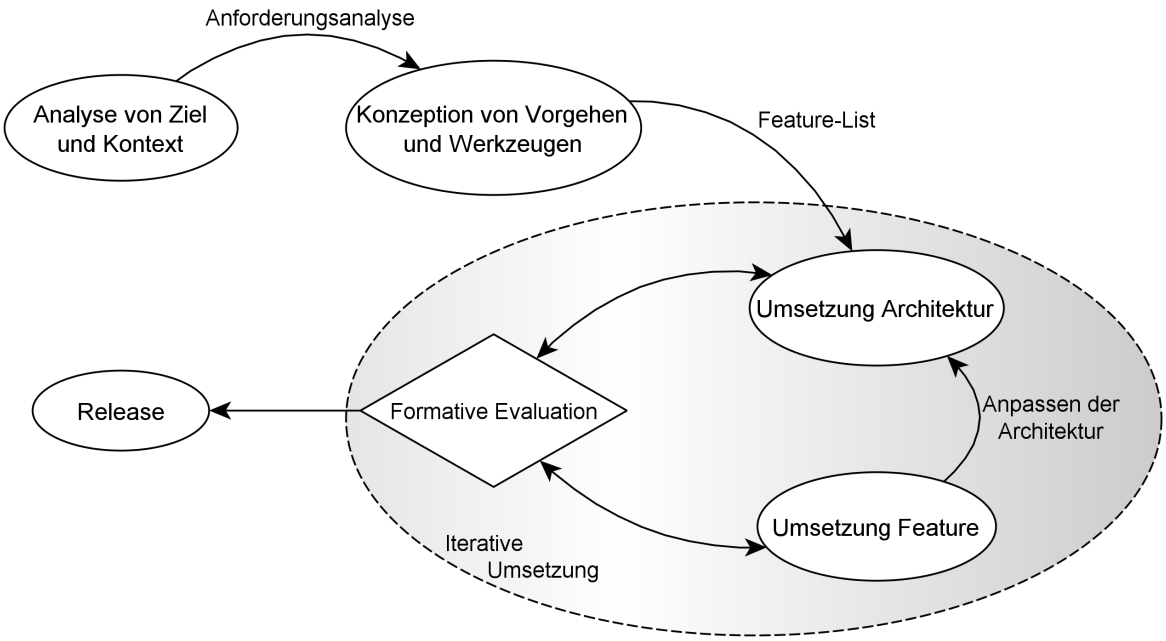

Abbildung 4: Iterativ inkrementeller Entwicklungsprozess

pitels besteht hauptsächlich darin, die finale Architektur des Reasoners zu verstehen und einen konkreten Einblick in die Realisierung der zugehörigen Komponenten zu erhalten. Entscheidungen, die während des Realisierung getroffen wurden, behandelt dieses Kapitel mit dem Teil der formativen Evaluation (Abschnitt 4.5). Um das nun gegebene System auf die erreichten Ziele der Arbeit zu prüfen, wird in Kapitel 5 eine Summative Evaluation des Systems vorgenommen. Dieses führt an, welche Verfahren der Evaluation durchgeführt wurden und welche Ergebnisse daraus gezogen werden können. Das letzte Kapitel 6 schließt die Arbeit ab, womit mögliche offene Punkte oder Erweiterbarkeiten des Systems und Vor- und Nachteile der neuen Funktionen, die diese Arbeit ergibt, zusammenfassend beschrieben werden.

2 Analyse

Der Konzeption und letztendlich der Entwicklung eines neuen Reasoners geht eine Analyse des Benutzungs- und Anwendungskontextes voran. Die Analyse stellt einen Teil der Softwarequalität sicher, indem sie Schwerpunkte setzt und Abgrenzungen formuliert. So wird im folgenden Kapitel geklärt, welche Benutzer mit dem Reasoner arbeiten, auf welche Art sie mit diesem interagieren und welche Probleme oder Besonderheiten auftreten und zu berücksichtigen sind. Letztlich wird versucht eine abstrakte Darstellung der Anforderungen und Features des Reasoners zu geben.

2.1 Aufgabenanalyse

Der Zweck der Entwicklung eines neuen Reasoners für CAKE besteht in der Erschließung neuer Anwendungsdomänen sowie deren Validierung, um die Kontexte der Benutzer korrekt zu erfassen. Im Folgenden wird der Entwickler bei seiner Arbeit zur Umsetzung einer neuen Domäne an CAKE und den anfallenden Reasoningfunktionen betrachtet. Dazu lassen sich grob zwei Arbeitsabläufe beschreiben: das initiale Entwickeln einer Domäne mit Context Awareness, welches vom Erzeugen eines Reasoners handelt, und die Interaktion mit dem Reasoner zur Laufzeit.

2.1.1 Entwicklung einer Domäne mit Context Awareness

Das Erschließen einer neuen Domäne im Hinblick auf die Umsetzung im CAKE-System umfasst im Groben drei Schritte. Zunächst wird die Domäne analysiert, hierbei müssen sowohl alle handelnden und beteiligten Personen berücksichtigt als auch die zu betrachtenden Artefakte analysiert werden. Für die Erfassung von Kontext-relevanten Daten benötigt der Entwickler als nächstes Sensoren und zur Wiedergabe auch Aktuatoren, die sich direkt aus einer Kontextanalyse ergeben können. Letztendlich wird der Reasoner selbst entwickelt.

Zu diesem Arbeitsablauf zeigt Abbildung 5 eine HTA[1] Vertiefung in der Entwicklung des Reasoners (relevante Pfade grau gefärbt). Um den zu entwickelnden Reasoner effizient und übersichtlich mehrere Male für verschiedene Umgebungen zu nutzen, wird es notwendig sein, mehrere voneinander unabhängige Instanzen zu erzeugen. Jede Instanz stellt eine Konkretisierung des zu entwerfenden

[1] Hierarchical Task Analysis

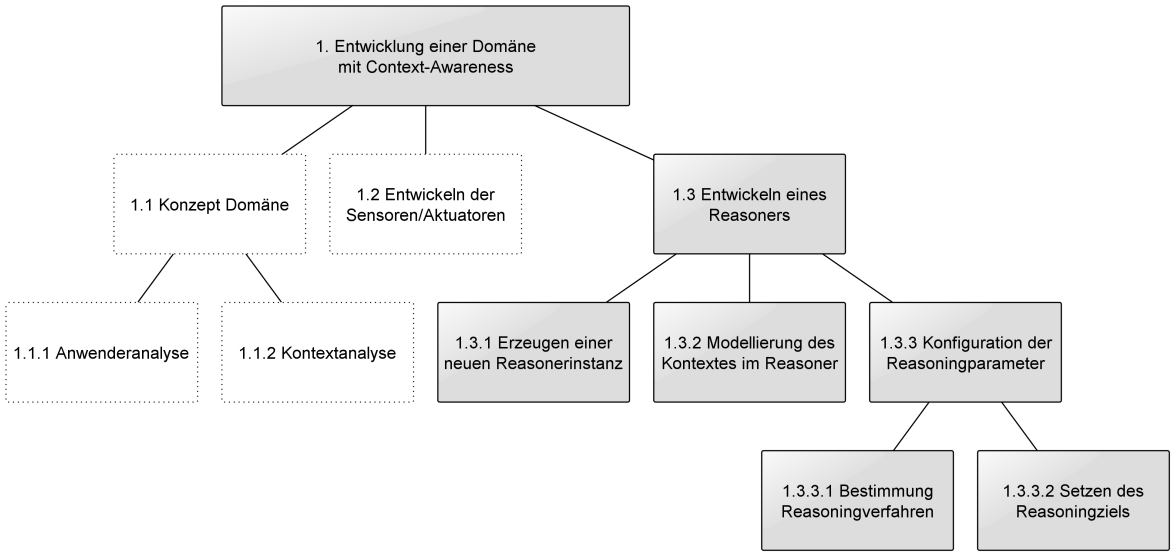

Abbildung 5: Aufgabe der Erschließung einer neuen Domäne, der graue Pfad beschreibt die mit dem Reasoner in Interaktion stehenden Schritte

Reasoning-Frameworks dar und führt ihre Aufgaben eigenständig aus. Dabei können die Abläufe bzw. Verfahren in jeder Reasonerinstanz unterschiedlich sein. Anzunehmen ist, dass vor der Entwicklung des Reasoners die Domäne und der Anwendungskontext genau analysiert, Daten(-typen) zusammengetragen und Zusammenhänge erstmals hergestellt wurden. Die Sensoren und Aktuatoren wurden zudem im Schritt *1.2 Entwickeln der Sensoren/Aktuatoren* entwickelt oder zumindest deren genaue Funktionsweise konzipiert.

Dieses Wissen über die Anwendungswelt muss nun modelliert werden, sodass dieses zudem im Reasoningkontext beschreibbar und anschließend benutzbar ist. Daher wird eine Modellierung auf wesentliche Aspekte reduziert und abstrahiert. Es bietet sich hierbei z.B. an, alle Daten, die erfassbar sind, als Variablen zu definieren und in einem großen Variablenvektor darzustellen. Ebenso können Variablen von verschiedenen Datentypen sein, von binären Aussagen bis hin zu numerischen Werten. Diese liegen nun definiert vor und können möglicherweise verschieden stark gewichtet werden, sodass einige Messungen des Kontextes mehr Aussagekraft erlauben. Ebenso bleibt zu entscheiden, welche Variablen notwendig angegeben werden müssen und welche nur optional dem Wissen in dem Anwendungskontext zur Verfügung stehen.

Letztlich muss der Reasoner selbst in seiner Funktionsweise konfiguriert werden. Je nach vorheriger Modellierung muss das Ziel des Reasonings konkretisiert werden, d.h. der Ausgabewert muss semantisch und syntaktisch festgelegt werden. Es ist also denkbar, als Reasoningausgabe binäre Entscheidungen zu modellieren. Darüber hinaus können aber auch numerische Werte oder Entscheidungen aus einer diskreten Menge an Möglichkeiten Ergebnisse darstellen. Da jede Modellierung andere Parameter und Variablen voraussetzt, können verschiedene Reasoningansätze und -verfahren unterschiedlich effektiv arbeiten. Der Entwickler sollte an dieser Stelle die Möglichkeit haben, technische Parameter zum Verfahren des Reasonings nach Bedarf festzulegen.

2.1.2 Interaktion mit dem Reasoner zur Laufzeit

Der Entwickler will auf verschiedene Arten mit dem Reasoner kommunizieren und interagieren. Dabei verfolgt er zum einen das Ziel, mit Aktuatoren Reasoningergebnisse zu erfragen, zum anderen aber will er den Reasoner in seiner Aussagekraft und Modellierung verifizieren und validieren. Insbesondere will er später Veränderungen vornehmen können.

Zunächst soll der Reasoner nach Ergebnissen befragt werden. Wie in der HTA aus Abbildung 6 zu diesem Arbeitsablauf zu erkennen ist, gliedert sich diese Teilaufgabe in zwei Aspekte auf: dem Abonnieren des jeweiligen Reasoners über einen Aktuator, der den Reasoner beauftragt, seine Ergebnisse an ihn über das CAKE-Whiteboard zu senden, und dem Senden von Daten an den Reasoner. Damit dieser die logischen Aussagen berechnen kann, benötigt er aktuelle Sensorwerte, bzw. allgemein die Eingabedaten, denen die Modellierung des Kontextes zugrunde liegt. Dazu werden die Sensoren derartig konfiguriert, dass sie ihre Daten über das Whiteboard an den soeben registrierten Reasoner senden. Der Reasoner muss insbesondere diese vereinzelt ankommenden Sensor- und Eingabewerte erhalten und als einen Datensatz zusammenfassen können, um anschließend Ergebnisse zu schlussfolgern.

Eine für den Lernprozess wichtige und für den Validierungsprozess notwendige Funktionalität ist es, das Reasoningmodel flexibel zu gestalten, d.h. es können Modellierungen, Techniken und Daten verändert werden. Diese Flexibilität erleichtert dem Entwickler die Aufgabe der formativen Evaluation seines geschaffenen Reasoners, da er zur Lauf- oder Verwendungszeit bestimmte Module rekonfigurieren kann und somit diese Präferenzen gegeneinander testen und vergleichen kann. Zunächst soll der Reasoner in der Lage sein, neue Datensätze einzuspeichern. Zu Beginn der Inbetriebnahme besitze der Reasoner eine gewisse Anzahl an Datensätzen zur Entscheidungsfindung von eingehenden Reasoninganfragen. Dann möchte der Entwickler zur Laufzeit diesen Bestand an Datensätzen erweitern, um die Berechnung auf mehrere vorliegende Daten zu stützen und damit den Lernprozess zu fördern. Während durch die Bereicherung der Datenbank das Model wächst, will der Entwickler im

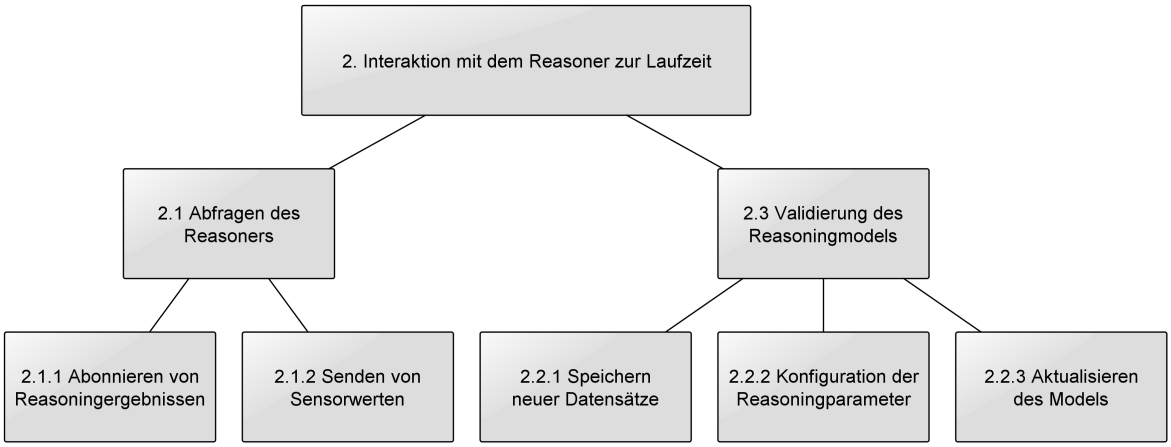

Abbildung 6: Aufgabe der Interaktion mit dem Reasoner zur Laufzeit

Prozess der Validierung auch Änderungen an den Konfigurationsparametern vornehmen. Wie schon im vorherigen Abschnitt (zur Entwicklung eines Reasoners) beschrieben, kann der Entwickler die initial gesetzen Parameter zur Konfiguration von Reasoning-Verfahren komplett umstellen, um möglicherweise seinen Anwendungskontext mit einer andersartigen Herangehensweise zum Reasoning auszustatten.

2.2 Benutzeranalyse

Der Reasoner unterstützt Entwickler für CAKE, neue Domänen an CAKE anzubinden bzw. bestehende Systeme in ihrer Mächtigkeit oder Reasoningfähigkeit zu erweitern. Die Gruppe der Benutzer lässt sich also auf erfahrene Systembenutzer eingrenzen. Dazu lassen sich die zwei wichtigsten Rollen im Anwendungskontext des Reasoners beschreiben mit Hilfe von Benutzerklassen genauer definieren.

Wie bereits beschrieben sind die Benutzer grundlegend erfahren und lassen sich somit ableiten von einer gemeinsamen Basisklasse. Wie von Herczeg (2009) näher beschrieben, kann grundlegend zwischen unerfahrenen Benutzern, Routinebenutzern und Experten unterschieden werden. Bei den letzten beiden Basisklassen unterscheidet sich der Erfahrungsgrad weitestgehend vom Wissen über das Anwendungssystem. Da CAKE am IMIS in der Forschung eingesetzt wird und voraussichtlich von wechselnden Studenten und studentischen Hilfskräften betrieben und weiterentwickelt wird, liegt nur ein grobes Wissen über das Anwendungssystem vor. Dadurch entstehen im Folgenden die zu betrachtenden Benutzerklassen als Konkretisierungen der Routinebenutzer-Basisklasse.

2.2.1 Benutzer-Basisklasse: Routinebenutzer

> *"Benutzer, die durch intensive und regelmäßige Arbeit mit einem bestimmten System Erfahrung und Routine besitzen."* - nach Herczeg (2009)

Routinebenutzer haben ein grundlegendes Verständnis in der Benutzung von Computersystemen und verwenden darüber hinaus im Anwendungsfall von CAKE Frameworks, die ihnen das Arbeiten an speziellen Aufgaben erleichtern.

Sie können grundlegend Architekturen eines Anwendungssystems verstehen und Programmcode in den wichtigsten Punkten lesen und zum Teil auch verändern. Zudem fällt die Einarbeitung in neue Systeme und Techniken selten schwer. Neue Umgebungen und Frameworks lassen sich leicht erlernen. Routinebenutzer für CAKE verstehen Analyse- und Konzeptionsverfahren und sind in der Lage, Probleme abstrakt und formal zu beschreiben, ebenso diese in konkrete Problemstellungen zu überführen.

Des Weiteren erwartet ein Routinebenutzer im Rahmen einer Arbeit am IMIS, dass Dokumentationen über das Anwendungssystem vorliegen und Programmcode und -prozeduren ausführlich kommentiert und strukturiert vorliegen. Sie gehen davon aus, dass sie sich an bestehende Arbeitsweisen anpassen

und führen Projekte in gleicher Weise fort, wie sie sie vorfinden.

Die Routinebenutzer haben meist an ähnlich strukturierten Systemen gearbeitet oder besitzen genauere Kenntnis über deren Beschaffenheit. Ebenso liegen Erfahrungen in der Planung und Umsetzung größerer Softwareprojekte vor. Die Programmierung von Modulen oder Funktionen in einem Softwareprojekt fallen dem Benutzer nicht schwer.

Die Arbeitsweise eines solchen Routinebenutzers basiert häufig auf dem Einsatz von digitalen Systemen. So werden Textverarbeitung, Bildbearbeitung und auch Programmierung üblicherweise am Computer vorgenommen. Die Benutzer verfügen in der Benutzung dieser Programme fortgeschrittene Erfahrung, arbeiten zügig und lösen Problemstellungen mit diesen selbstständig.

2.2.2 Benutzerklasse: Analytiker

Benutzerklasse: Analytiker einer Anwendungsdomäne

Übergeordnete Klasse: Routinebenutzer

Zugeordnete Softwareprodukte:

- Grafikprogramm zur Modellierung
- Textverarbeitung

Allgemeine Charakterisierung:

- Analyse von realen Kontexten (im Hinblick auf Computerwelten)
- effektiv im Extrahieren von Eigenschaften von Artefakten
- hohes Wissen an kontext-spezifischen Zusammenhängen von Attributen
- moderates Wissen über die Endandwendungswelt
- sind kommunikativ zur Erweiterung ihres Wissens über die Anwendungswelt

Besondere Kenntnisse und Probleme in der Anwendung:

- nur oberflächige Kenntnisse über einen realen Anwendungskontext
- Flexibilität im Bezug auf Akteure und Artefakte im Anwendungskontext

Besondere Erfahrungen und Probleme mit Computern:

- ausreichendes Wissen über Modellierungsprogramme
- hohes Wissen an Beziehungen von realen Attributen zur Computerwelt
- Hintergrundwissen für Übersetzung in realisierungsfertige Modelle vorhanden
- Effizienz von Systemen sind unbekannt oder werden nicht berücksichtigt

- tendiert dazu, zu hohe Ziele von der Anwendung (Reasoner) zu erwarten

2.2.3 Benutzerklasse: Entwickler

Benutzerklasse: Entwickler

Übergeordnete Klasse: Routinebenutzer

Zugeordnete Softwareprodukte:

- Entwicklungsumgebung
- API-Browser
- Simulationstools
- Compiler

Allgemeine Charakterisierung:

- Umsetzung von modellhaften Beschreibungen
- verstehen allgemeine Visualisierungen (Graphen, nicht zwangsläufig UML)
- haben ein hohes Wissen an Programmierung
- arbeiten sich selbstständig in neue Programmierumgebungen ein
- besitzen minimales Wissen über die Endanwendungswelt

Besondere Kenntnisse und Probleme in der Anwendung:

- Fachwissen in der Implementierung und Anbindung
- keine Nähe zur Endanwendungswelt
- benötigt klare Strukturen und Vorgaben

Besondere Erfahrungen und Probleme mit Computern:

- umfassendes Wissen im Umgang mit dem Computer
- versteht sich, multimediale Systeme zu nutzen
- große Computererfahrungen lassen ihn Anforderungen unterschätzen

2.3 Kontextanalyse

Da diese Arbeit kontextuell im Backend der bestehenden CAKE Architektur stattfindet, soll der folgende Abschnitt den Nutzungskontext des Reasoners als daraus resultierende Endanwendung be-

schreiben. Es befinden sich heutzutage einige wenige Systeme im Einsatz, die den Kontext um den Benutzer herum erfassen können und als Folge die Interaktion mit diesem derart verändern, dass Arbeitsschritte eingespart oder optimiert werden. Während wir heute diesen Systemen am ehesten im WWW[2] begegnen (nämlich in Chatprogrammen, sozialen Netzwerken, Suchmaschinen), gibt es viele Anwendungsgebiete, die reale Kontexte erfassen. Die folgenden Abschnitte führen Szenarien an, die zum einen die Umsetzung oder die Modellierung konkreter beschreiben, zum anderen aber auch sekundäre Szenarien, die abstrakt ein mögliches Anwendungsbeispiel repräsentieren sollen.

2.3.1 Primäres Szenario: Museum

Für ein neu eröffnendes ägyptisches Museum wird ein multimediales, interaktives Konzept geplant. Demnach sollen alle Exponate mit einem eigenen Darstellungssystem (Bildschirm mit möglicher Interaktion) ausgestattet werden. Um dem Benutzer dieses Museums seinen Besuch so angenehm und zielstrebig wie möglich zu gestalten, wird Martin Lange, ein Entwickler, beauftragt, dem Besucher geeignete Informationen zu einzelnen Exponaten anzuzeigen – je nach Interesse des Besuchers fallen diese verschieden aus. Dem Besucher können auf dem Bildschirm angezeigt werden: nur Bilder, kleine zusätzliche Überschriften oder Stichpunkte, große detaillierte Texte. Zusätzlich kann der Benutzer diese Ansichten umschalten, sollte er mehr Interesse bekunden, als vom System suggeriert.

Martin zieht einen Zusammenhang zwischen der Darstellungsart auf dem Bildschirm und der Zeit, die ein Besucher vor dem Bildschirm steht, in Betracht. Steht der Besucher bis zu zehn Sekunden davor, reichen einfache Bildanzeigen, bis zu dreißig Sekunden können diese mit Überschriften angereichert werden und darüber werden dem Besucher große Texte präsentiert.

Nun soll für die Schätzung des Interesses das CAKE-System benutzt werden und Martin will über einen Reasoner berechnen lassen, wie lange der Besucher vor dem jeweiligen Exponat stehen wird, um so den geeigneten Inhalt anzuzeigen. Er speichert so viele Datensätze wie möglich, um daraus Verhaltens- und Interessemuster zu erkennen. Jeder Datensatz beschreibt eine Kombination aus Exponat und Besucher. Wie in dem ER[3]-Modell in Abbildung 7 zu sehen ist, wird ein Exponat beschrieben durch seine Kennnummer, den Ausstellungstyp (wie Bauwerk, Wirtschaft, Politik, Schmuck, Schätze, Sonstiges), die Größe des Exponats und die betreffende geschichtliche Jahreszahl. Der Besucher wird mit Hilfe von Sensoren zu jedem Zeitpunkt in seinen Eigenschaften beschrieben: seine durchschnittliche Schrittgeschwindigkeit, den zeitlichen Anteil, den er sich nicht bewegt (steht), seine Lautstärke in Unterhaltungen durch Mikrophone (ruhig bis zu maximal laut), die Bewegung des Körpers beim Durchqueren des Raums mittels einer Kamera (starres Verhalten bis hin zu häufigen Umschauen) und letztlich wird gemessen, wie lange der Besucher nun vor dem Bildschirm stand, um den Reasoner für künftige Schlüsse weiterzubilden (Live-Learning).

Ein Besucher betritt zu Beginn das Museum und erhält an der Kasse einen RFID-Chip, um im System

[2] World Wide Web
[3] Entity Relationship

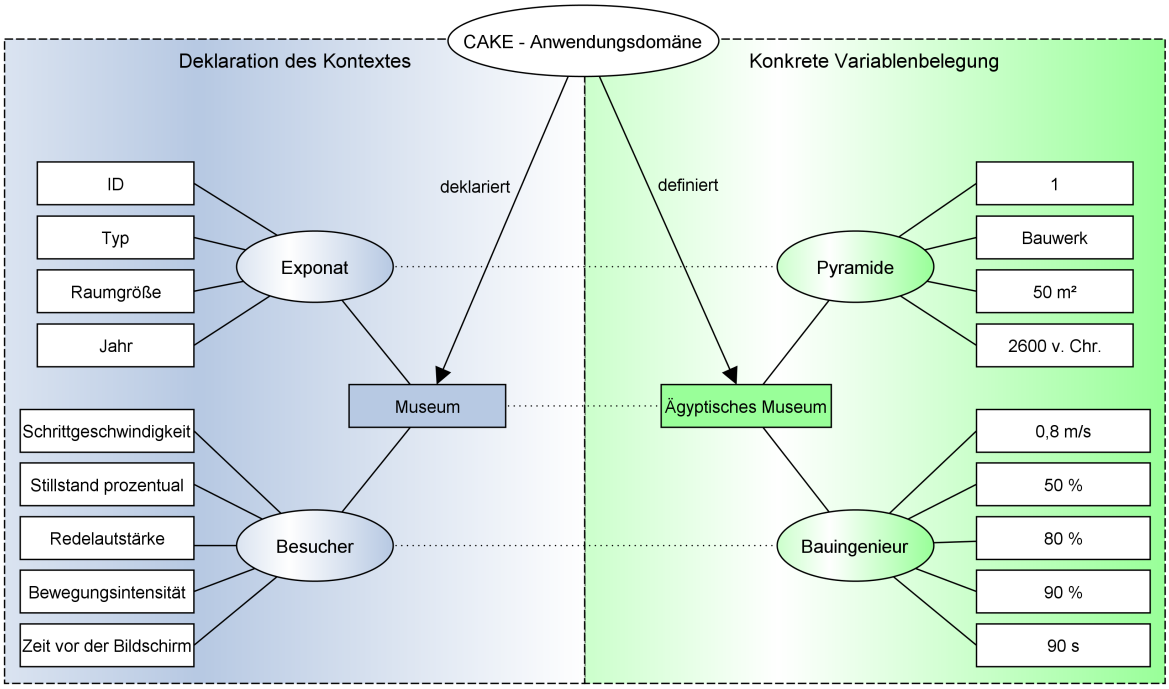

Abbildung 7: Modell einer Museumsumgebung: Die blaue Seite repräsentiert die Variablenmodellierung, die grüne Seite ein Beispiel eines zugehörigen Datensatzes

wiedererkannt und geortet zu werden. Alle weiteren Informationen über den Besucher ergeben sich im Verlauf des Museumsbesuches. Die Funktionstauglichkeit des neuen Context Awareness Systems prüft Martin mit formativer Evaluation. So wird das System in Zusammenarbeit mit vielen Besuchern getestet und während des Prozesses immer wieder mit den Erfahrungen der Besucher abgeglichen. Dabei kann Martin einfach nur abwarten, damit das Model des Reasoners wächst oder aber auch selbst die Modellierung oder Beschaffenheit der Sensoren anpassen.

Technisch setzt Martin diese Modellierung um, indem er den Reasoner auf seinen Kontext konfiguriert. Dazu gibt er alle Attribute in eine Maske ein und markiert entsprechend das Attribut, welches die Zeit vor dem Bildschirm beschreibt. Im Rahmen der formativen Evaluation kann Martin das semantische Reasoningverfahren bestimmten. Er entscheidet sich zunächst für einen Wahrscheinlichkeitsbasierenden Ansatz und wählt als Machine-Learning-Verfahren Naïve Bayes aus. Anschließend wird dieser Reasoner im CAKE-System in Betrieb genommen und erhält dann über das Kommunikationsframework von CAKE, das sogenannte *Whiteboard* (siehe Frahm, 2011) die Sensorwerte. Es wird im Folgenden für die anfragenden Exponate die Werte berechnet und auf dem Whiteboard zurückgegeben. Die Informationen kann Martin von entsprechenden Aktuatoren lesen lassen und dann zielgerecht auf dem Bildschirm wiedergegeben.

Abbildung 7 zeigt neben der Modellierung einen Beispieldatensatz. In diesem Fall wird ein ägyptische Museum von einem Bauingenieur besucht. Sein Interesse gilt überwiegend Bauwerken, welche während des Besuches mit höheren Aufenthaltszeiten vor Exponat und Bildschirm erfasst werden.

Das Beispielexponat besucht dieser nun also interessiert, indem er sich nur 50% der Zeit bewegt, sich viel unterhält und zudem vor dem $50m^2$ großen Exponat häufig seinen Kopf oder Körper bewegt. Der Bauingenieur verbringt zudem 90 Sekunden vor dem Bildschirm und liest Informationen.

Mit diesen Informationen erkennt das System schnell, dass wiederum andere Bauwerke sehr interessant sind, bzw. vergleicht das Interesse für Bauwerke mit anderen Exponaten, für die Bauingenieure aber auch Interesse haben könnten (aufgrund von Erfahrungen mit Besuchern, die früher auch überwiegend gerne Bauwerke betrachtet haben).

2.3.2 Sekundäres Szenario: Ambient Assisted Living

In einer Wohnung sollen für einen körperlich eingeschränkten Bewohner (z.B. Rollstuhlfahrer) diverse Haushaltsgeräte interaktiv bedienbar gemacht und überwacht werden, um den Menschen in der Ausübung von Arbeiten im Haushalt maximal zu unterstützen. Alle Geräte, zu denen neben Küchengeräten auch Licht-, Heiz- und Lüftungsanlagen zählen, werden in einem Netzwerk zusammengefasst und zentral über CAKE überwacht und gesteuert. Um die Geräte zu bedienen, kann der Bewohner eine konservative Tastatureingabe machen, aber möglicherweise auch über Gesten- und Sprachsteuerung interagieren.

Geräte werden teilweise selbst ein- und ausgeschaltet. Ebenso werden dem Bewohner über ambiente Aktuatoren je nach Wichtigkeit Vorschläge zu solchen automatisieren Vorgängen vermittelt. Des Weiteren kann überwacht werden, ob besondere Vorfälle im Haushalt vorliegen (Bewohner in Not, benötigt Hilfe, unfähig, selbst Hilfe zu rufen). Die Abläufe im Haushalt und die Individualität des Bewohners bestimmen, wie selbstständig das CAKE-System auf verschiedene Abläufe reagiert und handelt. Die Wohnung kann dazu mit Lichtschranken, Luftfeuchtigkeits- und –temperatursensoren, Kameras und Mikrophonen ausgestattet werden, die wesentliche Informationen über den Haushalt und den Bewohner erfassen.

Ein Entwickler, der eine solche Wohnung an das CAKE-System anbindet, entwickelt entsprechende Sensoren und stellt diese auf dem CAKE-Whiteboard dar. Als Antwort erhält er von meinem Reasoner Interaktionsinterpretationen von Gesten und Spracheingaben, die als Anweisungen dann im System ausgeführt werden können. Je nach Anfragenbeschreibung an das Whiteboard ermittelt der Reasoner diese Anweisungen oder auch Meldungen über Wohnungszustand oder gegebenenfalls auftretende Probleme.

2.3.3 Sekundäres Szenario: Fehlerprognostizierung

Ein völlig neuartiges Überwachungssystem in Automobilen soll dafür sorgen, dass Fahrzeugführer in verschiedenen Kraftfahrzeugen die Kontrolle über das Fahrzeug behalten. Im Gegensatz zu Fahrhilfen soll das Überwachungssystem nun den Fahrzeugführer dahingehend analysieren, wie sein Konzentrationszustand beschaffen ist. Befindet er sich in einem kritischen Zustand, weil der Fahrer abgelenkt

oder gar zu sehr ermüdet ist, so soll das System geeignet Alarm schlagen (ohne eine Schrecksituation hervorzurufen) und muss dem Fahrer signalisieren, dass erhöhte Aufmerksamkeit bzw. eine Fahrpause gefordert ist.

Ein Entwickler kann dieses System am CAKE-System umsetzen, indem er den zu ermittelnden Konzentrationszustand entsprechend einer Skala als Zahlenwert messbar modelliert. Anschließend lässt sich für die Berechnung dieses Wertes die CAKE-Logik- abfragen. Der Entwickler modelliert den Fahrzeugkontext und bringt entsprechend geeignete Sensoren an. Dazu zählen ein Pulsmessgerät, Kamera, Motioncapturing-Geräte an Lenkrad sowie Geschwindigkeitstaster, die ständig ermitteln, wie regelmäßig und konsistent der Fahrer sein Fahrzeug führt. So ließen seltene, aber ruckartige Lenkbewegungen auf einen Sekundenschlaf hindeuten.

Die Sensorwerte werden im CAKE-Whiteboard zusammengetragen und der Reasoner ermittelt daraus einen Konzentrationswert, welcher an den Aktuator im Fahrzeug zurückgegeben wird. Je nach Dringlichkeit reagiert das System nun mit Geschwindigkeitsdrosslung, Warnhinweisen visueller oder auditiver oder sogar haptischer Art. Das System gibt einen Teil der Kontrolle erst zurück, wenn ein sicherer Zustand wiederhergestellt ist.

2.4 Systemanalyse

Im Folgenden soll das CAKE-System grob umrissen werden. Dazu stellt der Abschnitt ein abstraktes Bild der Zusammensetzung der Module von CAKE dar und trennt die eigentliche CAKE Umgebung in die drei großen Schichten auf. Am Ende soll das System in Grundzügen bekannt und der Einsatz des Systems vorstellbar sein.

Abbildung 8 verzichtet auf zusätzliche Elemente und zeigt die elementarsten Module und Schichten des CAKE-Systems. Diese sehr abstrakte Sicht auf das System wird im Kapitel 3 detaillierter betrachtet und konzipiert. Zunächst folgt eine Betrachtung der abgebildeten Systemteile bzw. -schichten.

CAKE-Environment. Die CAKE Umgebung stellt das Basis- und Servermodul von CAKE dar. Hier werden Berechnungen gemacht und alle beteiligten Systeme miteinander verbunden. Eine CAKE Umgebung kann sich mit anderen verbinden, siehe dazu mehr unter *XMPP-Server*. Die Submodule der CAKE Umgebung werde ich gesondert betrachten.

XMPP-Server. Das XMP[4]-Protokoll (XMPP) dient der Kommunikation zwischen verschiedenen Systemen. In dem Fall von CAKE dient ein solcher XMPP-Server dazu, verschiedene CAKE Umgebungen miteinander zu verbinden. Es können hierbei beliebig viele Umgebungen über den Server zusammengeschlossen werden. Mit Hilfe dieser Kommunikationsschnittstelle lässt sich auch geographisch weiter entfernte Sensorik und Logik für ein System lokal nutzen.

1. Schicht: Kommunikation. Die Kommunikationsschicht dient dem Datentransport von dieser CAKE

[4]Extensible Messaging and Presence

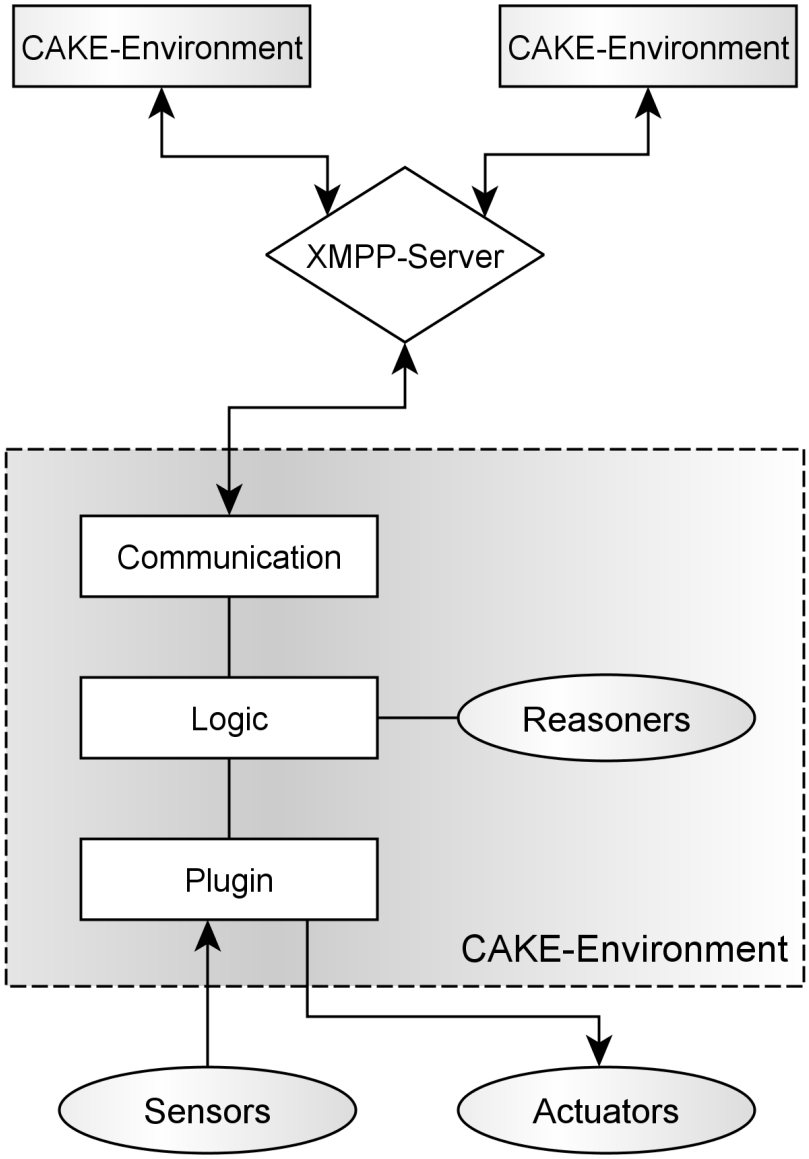

Abbildung 8: Eine vereinfachte, abstrahierte Sicht auf die Schichten und Module von CAKE

Umgebung in andere CAKE Umgebungen. Da CAKE mit Benutzergruppen und -berechtigungen arbeitet (vergleiche Bouck-Standen et al., 2013, Kapitel 4), können hiermit Anfragen für Sensorwerte über das XMP-Protokoll gestellt werden und damit für die eigene CAKE Umgebung genutzt werden. Durch die Kommunikationsanbindung lassen sich unkompliziert mehrere Systeme zusammenfassen. Diese können durch die Aufteilung in Schichten miteinander kommunizieren, wobei die Logik- oder Pluginschicht einer CAKE Umgebung selbst keinen Unterschied für die empfangenen Daten registriert.

2. Schicht: Logik. Die Logikschicht setzt sich aus der inneren Kommunikation in CAKE sowie dem Reasoning zusammen. Das CAKE-Whiteboard stellt hierbei die zentrale Rolle dar: vergleichbar mit

einem Verteiler übernimmt das Whiteboard die Aufgaben der Kommunikation und Weiterleitung von Daten zwischen allen Plugins und den dafür benötigten Reasonern. Benutzer und Plugins müssen sich ebenso wie Reasoner an dem Whiteboard registrieren. Das Whiteboard hält zum einen alle angemeldeten Benutzer- und Plugininstanzen in einer dafür angepassten, recht abstrakten Ontologie vor. Zum anderen verfügt es über einen Nachrichtenstrom, dessen Anschlussstellen zu allen Plugins und Reasonern führen. Durch ein Datenabonnementverfahren leitet das Whiteboard entsprechende Datentypen an die jeweiligen Plugins weiter.

3. Schicht: Plugin. Die Pluginschicht dient der Registrierung von Sensoren und Aktuatoren am System. Ein Pluginmanager fasst alle Instanzen zusammen und stellt entsprechend den Datenverkehr mit dem Whiteboard her. Es können Sensoren weiter unterschieden werden: Remote-Sensoren sind diejenige Sensoren, die in einer anderen CAKE Umgebung vorliegen. Diese werden über die Kommunikationsschicht und XMPP-Verbindung abonniert und liegen anschließend zur weiteren Verarbeitung bereit. Weiterhin hält das CAKE-System virtuelle Sensoren bereit, die in der Logikschicht des Systems vorliegen und virtuell betrieben werden. Das System kann diese Sensoren trotzdem wahrnehmen und verwenden, als wären es lokale Sensoren. Für die Pluginentwicklung können des Weiteren Treiberpakete als Plugin registriert werden, die ebenso über die Pluginschicht eingebunden werden. Mehr dazu ist der Dokumentation von Bouck-Standen et al. (2013) zu entnehmen.

2.5 Machine Learning

In Abschnitt 1.3 wurden kurz drei mögliche Ansätze zum Reasoning eingeführt. Der kommende Teil konkretisiert das Konzept des Machine Learnings und beleuchtet grundlegende Mechanismen und Funktionsweisen. Machine Learning gewinnt in wissenschaftlichen Anwendungen an Beliebtheit und bietet durch das Lernverhalten und die Vielfältigkeit der Reasoningverfahren vielseitige Einsatzgebiete.

2.5.1 Funktionsweise

Machine Learning lässt sich einfach beschreiben als ein maschineller Ansatz zum Erlernen und Wiedererkennen bestimmt Muster. Der Lernvorgang setzt sich daraus zusammen, dass dem Machine Learning Model bereits bekannte Muster beigebracht werden, d.h. das System erhält solche und speichert zu diesen die zugehörigen Attribute zur Einordnung (die wiederum dem Reasoningziel gleichkommt). Fortlaufend kann ein Machine Learner durch korrekt zugeordnete Datensätze erweitert werden. Das System lernt weitere Muster dazu. Gleichzeitig werden bereits gelernte und damit zugeordnete Muster als Grundlage verwendet, um neue und unbekannte Datenstränge und -muster einzuordnen.

Grundsätzlich lassen sich Lernprozesse in verschiedene temporale Formen unterteilen: dem *Online-* und dem *Offlinelearning*. Der Unterschied hierbei ist die Art und Weise, wie bereits zugeordnete Muster dem Machine Learner beigebracht bzw. von diesem aufgenommen und gelernt werden. Das

Onlinelearning bedient sich einem beständigen Eingangstrom an Datensätzen, die im Machine Learning Model zwischengespeichert (also temporär eingelagert) und direkt zu Ergebnissen für unbekannte Muster weiterverarbeitet werden. Durch den hohen Echtzeit-Zugang an erlernbaren Mustern speichert dieses Verfahren keine Datensätze langfristig und benötigt diese auch nicht.

Im Zusammenhang mit den angedachten Anwendungsbeispielen für CAKE, die einen eher geringen Datendurchsatz oder Eingangsstrom besitzen, bietet sich für diese Arbeit viel mehr das *Offlinelearning* an. Hierbei werden die erlernbaren Muster dem Machine Learner zugeführt und von diesem in einer Datenbank gespeichert. Aus diesen Daten können nun selektiv einzelne oder auch alle zugeordneten Muster ausgelesen und in einem Verarbeitungsprozess zu einem Machine Learning Model berechnet werden. Dieses Model ist im Anschluss für das Wiederfinden von ungekannten Mustern zuständig. Aufgrund der Möglichkeit, bekannte Muster zu speichern, lassen sich Modelumfang sowie auch -typ beliebig im Nachhinein verändern, indem ein neues Model generiert wird.

2.5.2 Algorithmen

Dieser Abschnitt führt kurz die Kernbereiche der Methoden an, mit denen ein Machine Learning Model aus einer gegebenen Menge an bekannten Mustern berechnet werden kann. Jedem Bereich unterliegt eine Vielzahl an unterschiedlich komplex implementierten Algorithmen, die sich in Laufzeit und Aussagekraft jeweils unterscheiden. Diese wurden von Pitakrat et al. (2013) im Rahmen einer Analyse zur Fehlererfassung von Festplatten miteinander vergleichen und dargestellt. Im Folgenden werden einzelne Bereiche aus dieser Darstellung rekapituliert, sowie jeweils ein Beispielalgorithmus aus dem jeweiligen Bereich genannt.

Wahrscheinlichkeitsbasierte Algorithmen. Basierend auf der Wahrscheinlichkeitsrechnung werden die Attribute von erlernbaren Mustern klassifiziert und mit Hilfe von Wahrscheinlichkeitsverteilung beschrieben. Soll ein unbekanntes Muster wiedergefunden werden, so werden die Eingabeattribute zu der wahrscheinlichsten Klasse zugeordnet. Nach dem Bayes'schen Theorem der bedingten Wahrscheinlichkeit gilt der Naïve Bayes Algorithmus als sehr bekannt.

Entscheidungsbäume. Bei der Erstellung von Entscheidungsbäumen werden die zu erlernenden Muster nach Entscheidungsregeln der Attribute zerlegt. Der C4.5 Algorithmus stellt hierbei einen Top-Down Entscheidungsbaum dar, welcher soweit aufgezweigt wird, bis ein Blattknoten nur noch zugeordnete Muster einer bestimmten Klasse aufzeigt.

Regelbasierte Algorithmen. Bei den Regelbasierten Algorithmen handelt es sich primär darum, Regeln aus einer gegebenen Menge an bekannten Mustern abzuleiten. Dabei gilt es, Besonderheiten und unterscheidbare Eigenschaften zu finden und somit zur Wiedererkennung als Regeln darzustellen. Der OneR Algorithmus erstellt hierbei Regeln für jedes einzelne Attribut und errechnet beim Klassifizieren die Fehlerrate jeder möglichen Zuordnung ausgehend von diesen Regeln. Die niedrigste Fehlerrate bildet hierbei das gesuchte Ergebnis ab.

Hyperplanare Aufteilungen. Bei dem Ansatz der hyperplanaren Aufteilung werden die Muster nach ihren Attributen in einen mehrdimensionalen Raum abgebildet und mittels Hyperplanes derart aufgeteilt, dass ein Teilraum nur noch Instanzen solcher Muster beinhaltet, die der gleichen zugeordneten Klasse angehören. Der Support Vector Machine Algorithmus teilt hierbei die Räume binär auf, indem er solange einen Raum in zwei eindeutige Klassen mittels Hyperplane zerlegt, bis nur noch eine Klasse im Raum vorliegt. Mit Hilfe dieser Zerlegungstechnik werden unbekannte Muster analog dazu aufgeteilt und somit dem korrekten Raum der bekannten Klassen zugeordnet.

Funktionsapproximierung. Um simple Funktionen mit Eingabe eines Eingabevektors und Ausgabe einer zuzuordnenden Klassenvariable zu erschaffen, arbeiten Algorithmen mit Approximationsverfahren, die diese Funktionen abschätzen und mit deren Hilfe nun unbekannte Muster einer Klasse zuordnen. Der Multilayer Perceptron Algorithmus kann hierbei als künstliches neuronales Netz verstanden werden, bei dem jedes Neuron eine Funktion darstellt, die ein bestimmtes Attribut auf eine Klasse abbildet.

Instanzbasierte Algorithmen. Das instanzbasierte Lernen speichert alle erlernten Muster vorerst ab. Erst zu dem Zeitpunkt der Klassifizierung beginnt der Algorithmus, die gelernten Muster zur Zuordnung in Berechnungen einzubeziehen. Der Nearest-Neighbour Algorithmus stellt hierbei einen klassischen Ansatz für das instanzbasierte Lernen dar. Hierbei werden (meist euklidische) Entfernungen der zu bestimmenden Musterinstanz zu den bereits erlernten Mustern berechnet und die Klasse der Instanz, die am nächsten liegt, wird ausgegeben.

2.6 Fazit der Analyse

Abschließend zur Analyse fasst dieser Abschnitt zusammen, was die Ergebnisse der Analyse sind und worauf sich das Projekt in Entscheidungsfragen festlegt. Zunächst wird die Benutzergruppe als Zielgruppe reflektiert, anschließend die Reasoning-Technik aufgegriffen und zuletzt eine vom System ungefähr zu erwartende Feature-Liste aufgezeigt.

2.6.1 Aufgabenübernahme von Benutzergruppen

In Abschnitt 2.1 werden die verschiedenen Aufgaben zur Erschließung einer neuen Domäne für das CAKE-System beschrieben. Abbildung 5 zeigte dazu eine Übersicht. Abschnitt 2.2 führt zwei grundlegende Benutzergruppen ein, die für die Domänentwicklung benötigt werden. Die folgende Aufstellung führt die Aufgaben und die Benutzergruppen zusammen, um den Schwerpunkt der Entwicklung bezüglich einer Benutzergruppe zu erläutern.

Analytiker. Die Analyse- und Konzeptionsphase für die Erschließung einer neuen Domäne unterliegen dem Aufgabenbereich des Analytikers. Hier werden Informationen zum Kontext, zum Kunden und Zielgruppen erworben und konsolidiert. Wie Abbildung 5 bereits farblich suggerierte, fallen

diese Bereiche von dem eigentlichen Anwendungsgebiet des neuen Reasonersystems heraus. Ferner könnte der Analytiker in das Entwicklungsgeschehen des Reasoners eingebunden werden, wenn Punkt *1.3.2.1 Variablendefinition* und Punkt *1.3.3.2 Setzen des Reasoningziels/-variable(n)* umzusetzen sind. Hierbei fließen gleichermaßen Implementierungsanforderungen, aber auch Kontext- und Analyseanforderungen in die Arbeit ein.

Entwickler. Der Entwickler übernimmt die restlichen Aufgaben, die grundsätzlich die gesamte Entwicklung, Implementierung, Konfiguration und Wartung des Reasoners betrifft. Aus diesem Grund müssen Handhabbarkeit und Schnittstellen, sowie die Dokumentation vorrangig an die Benutzergruppe des Entwicklers angepasst werden. Die Arbeit am Reasonersystem selbst setzt also primär Kenntnisse über Programmierung und Algorithmen voraus. Des Weiteren muss der Entwickler mit dem CAKE-System sowie der Verwendung des Reasoner-Basissystems vertraut sein. Die folgenden Kapitel der Arbeit betrachten den Entwickler für CAKE als zentrale Benutzergruppe.

2.6.2 Machine Learning als Reasoning-Technik

Aufgrund ausführlicher Recherche und durch die Analyse der Methoden verschiedener Reasoning-Ansätze (vergleiche Abschnitt 1.3) ist die weitere Entwicklung der Arbeit auf die Methodik des Machine Learning festgelegt, wie schon durch Abschnitt 2.5 suggeriert wurde. Das Machine Learning erlaubt einen sehr grundlegenden Lernprozess, der sich aufgrund einer großen Bandbreite an Algorithmen und Techniken in beliebige Bereiche spezialisieren lässt. Da insbesondere die unterschiedlichen Reasoningverfahren bzw. -algorithmen verschiedene Arten von Eingabevariablen erwarten oder auch Tradeoffs wie Effizienz gegen Effektivität beeinflussen können, bildet ein Machine Learning Reasoner die ideale Basis für den für diese Arbeit angestrebten flexiblen und beliebig anwend- und erweiterbaren Context Awareness Reasoner.

Das folgende Kapitel 3 geht näher auf die Umsetzungen und Verwendung von Machine Learning im Umfeld von CAKE ein und stellt geeignete Methoden und Bibliotheken für die Verwendung mit CAKE vor. Des Weiteren wird die Sicht auf das CAKE-System mit Fokus auf die Logikschicht konkretisiert werden und die im folgenden Bereich aufgeführte Feature-Liste erneut aufgegriffen und re-konzipiert.

2.6.3 Feature-Liste

Die Anforderungen des Systems ergeben sich zu einem großen Teil aus der Aufgaben- und Kontextanalyse. Die folgende Liste soll eine ungefähre Übersicht über die geforderten Ziele des Reasoners geben. Erweiterungen und andere Aspekte werden gegebenenfalls bei Bedarf beschrieben. Das Kapitel 3 wird diese Liste für die Entwicklung der Architektur nutzen und diese Liste eventuell verändern oder konkretisieren.

1. Mehrfach-Instanziierung des ML-Reasoners

   Der Reasoner soll mehrfach in seiner Umgebung gestartet werden können und dabei verschiedene Formen bezüglich Reasonings und Modellierung annehmen können

2. Flexible Modellierung von Variablen und Ausgaben

   Entsprechend Punkt 1 muss jede Reasoninginstanz einfach zu modellieren und zu individualisieren sein

3. Konfiguration Reasoning-Verfahren

   Gesondert soll es möglich sein, ganz spezielle Reasoningansätze zu wählen und mit wenig Aufwand in Betrieb zu nehmen

4. Erweiterung des Models durch gegebene Datensätze

   Zur Laufzeit sollen Daten gesammelt/gespeichert werden und das Model muss aktualisierbar sein (möglicherweise manuell zu bestimmten Zeiten)

5. Konfiguration und Adaptierbarkeit zur Laufzeit

   Während der Benutzung des Reasoners kann die Konfiguration angepasst werden, um eine Umstellung aufgrund von Evaluation oder Anforderungsänderungen zu realisieren

6. Erweiterbarkeit Schnittstellen

   Der Reasoner soll architektonisch abstrahiert und gut erweiterbar vorliegen, um auch tiefer gehende Änderungen, die nicht durch äußere Konfiguration abgedeckt sind, am Code schnell verwirklichen zu können

3 Konzeption

Mit der Entwicklung eines Reasoners entsteht ein Teilmodul des Gesamtsystems. Genauer siedelt sich dieses in der Logikschicht von CAKE an. Aus diesem Grund ist es notwendig, die bereits bestehende Architektur rund um einen CAKE-Reasoner zu verstehen, damit der neue Reasoner an das Grundsystem angepasst und adaptiert werden kann. Im Folgenden wird zunächst die Logikschicht detailliert betrachtet, wobei auf Nähe zur wirklichen Implementierung am Programm geachtet wird, gefolgt von einer zielgerichteten Beschreibung der Reasonerklasse in CAKE.

3.1 Bestehende Architektur

In Abschnitt 2.4 wurde das Grundsystem von CAKE analysiert und die drei Schichten (Kommunikation, Logik, Plugin) eingeführt. Der folgende Abschnitt zeigt nun einen tieferen Einblick in die Logikschicht. In der Abbildung 9 werden die wichtigsten Klassen in der Logikschicht deutlich, die nun anschließend beleuchtet werden sollen.

3.1.1 Logikschicht von CAKE

Whiteboard. Das Whiteboard stellt in CAKE eine zentrale Einheit dar. Sämtliche Kommunikation der Plugins (wie Sensoren, Aktuatoren) mit entsprechenden Reasonern oder anderen CAKE-Umgebungen läuft primär über das Whiteboard. Das Whiteboard speichert die registrierten Instanzen, zu denen Benutzer und Sensoren gehören, auf zwei Arten: zunächst liegen alle Objekte in Listen oder Maps vor, die für einen einfachen Datentransport ausreichend sind. Sollten komplexere Zugriffe oder Informationsabfragen nötig werden, so besitzt das Whiteboard eine Ontologie, in der die CAKE-Objekte untereinander in Beziehung gesetzt werden. Zu diesem Zweck gibt es einen `OntologyManager`, der in der Lage ist, die intern betriebene Ontologie abzufragen.

Relevante Funktionen für die Interaktion mit dem Reasoner sind die folgenden:

```
+ updateSensorValue(String, Map<String, Object>, PluginType) : void
+ getReasonerManager() : ReasonerManager
```

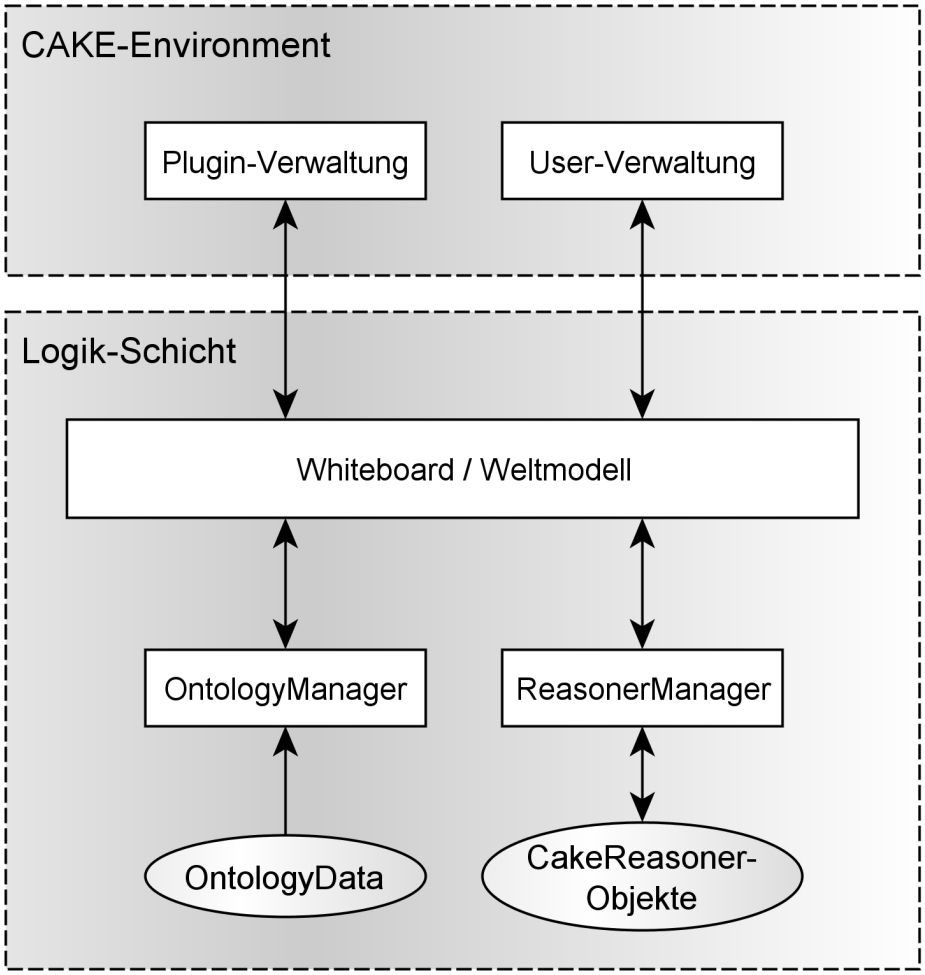

Abbildung 9: Zusammensetzung der Logikschicht nach Bouck-Standen et al. (2013)

Die erste Methode wird von Sensoren aufgerufen und erhält deren Wertepaare. Diese werden vom Whiteboard an den `ReasonerManager` weitergeleitet. Zur Registrierung von Reasonern wird derselbige benötigt. Auf dessen Funktion wird im Folgenden eingegangen.

`ReasonerManager`. Neben Sensoren werden auch beliebig viele Reasoner unterstützt. Zur besseren Unterteilung der Logikschicht finden diese Anschluss im `ReasonerManager`. Dieser hält alle Reasoner vor und verknüpft diese mit den von ihnen benötigten Eingabe-Datentypen. Der Datenverkehr zwischen dem CAKE-System und einem Reasoner findet über das Whiteboard und den `ReasonerManager` statt. Wenn das Whiteboard Daten an den diesen sendet, wird folgende Methode aktiv:

```
+ postInputUpdateToReasoners(InputUpdate) : void
```

Diese Methode verteilt die Eingabewerte an daran interessierte Reasoner. Neben dieser Methode gibt es noch weitere, die für die Aktualisierung der Sensoren oder für das Hinzufügen oder Entfernen von

Reasonern selbst zuständig sind.

In CAKE werden Sensoren an das Whiteboard angeschlossen. An diesem treffen Sensorwerte ein und können nun weiter verteilt werden. Dabei gelangt der gesamte eingehende Datenverkehr an das Whiteboard. Ebenso kann dieses die Daten, die von Reasonern ermittelt wurden, wieder an entsprechende Aktuatoren weitergeben, die ebenfalls in der Whiteboard-Klasse als Referenzen gespeichert sind. Sowohl die Aktuatoren als auch die Reasoner werden in Maps gespeichert, die als Schlüssel jeweils den Wertetyp angeben. Somit können spezielle Informationen schnell an den nachfragenden Reasoner oder Aktuator weitergeleitet werden. Der `ReasonerManager` stellt hierfür Methoden zur Verfügung, um sogenannte *Subscriptions* an die Geräte bzw. Sensoren zu stellen. Mit dieser Technik weiß der Manager, welche eingehenden Datensätze an welchen Reasoner zu verteilen sind.

3.1.2 Einbindung von Reasonern in der Logikschicht

Wie bereits im vorangegangenen Abschnitt erwähnt, verbindet der `ReasonerManager` das Whiteboard mit den Reasonern. So bedeutet dies also, dass alle Reasoner im `ReasonerManager` registriert und nur von dort referenzierbar sind. Trotzdem muss das CAKE-Programm im Startprozess die benötigten Reasoner instanziieren und laden. Dies geschieht zur Zeit im Initialisierungsprozess des Whiteboards.

Wie in Quelltext 1 zu sehen ist, durchläuft der Initialisierungsprozess einige Schritte vor der Erzeugung des Whiteboards. Der `ManagerService` stellt hierbei die Schnittstelle für die Benutzerverwaltung dar. Da Plugins verschieden Treibervoraussetzungen mitbringen können, werden diese anschließend direkt geladen. Zuletzt stellt CAKE dann die XMPP-Verbindung zum Server her, um die Kommunikation zu anderen Umgebungen zu ermöglichen. Nun beginnt der Startprozess des Whiteboards, an den sich im Anschluss sofort der Reasoner erzeugen lässt.

Angedeutet in der letzten Zeile von Quelltext 1, benötigt der Reasoner eine Objektreferenz auf den `ReasonerManager`, der wiederum von der soeben erzeugten Whiteboardinstanz abzufragen ist. Der in CAKE implementierte `RuleBasedReasoner` erzeugt sich, indem zunächst eigene Komponenten initialisiert und anschließend zwei Interaktionen mit dem `ReasonerManager` möglich

```
main(String[] args)
  new CakeStarter().start()
    new ManagerService()
    initDriverRepository()
    loginXMPP()
    initWhiteboard()
      new Whiteboard()
      new RuleBasedReasoner(ReasonerManager)
```

Quelltext 1: Stacktrace an Methodenaufrufen bis zum Reasoneraufruf

gemacht werden: das Abonnieren von Meldungen, sollten sich neue Geräte an CAKE anmelden, sowie das eigentliche Registrieren an dem `ReasonerManager`, was zur Folge hat, dass der Reasoner berechtigt ist, Sensoren über diesen zu abonnieren.

Die Reasonerklasse, im Beispiel der `RuleBasedReasoner`, stellt den obersten Zugriff auf den Reasoner dar. Abbildung 10 zeigt den Paketaufbau der Logikschicht in CAKE. Betrachtet wird zunächst der **`core`**: Die Teilmodule lassen sich aufteilen in `board`, welches das Whiteboard mit der gleichnamigen Klasse enthält, `ontology`, das die zugehörige Ontologie aufbaut und bereitstellt, und `reasoner`, der für die Schnittstellen eines Reasoners benötigt wird. In diesem Paket liegen der `ReasonerManager` und einige Ein- und Ausgabeklassen bzw. -Threads vor, die für die Grundfunktionalität von implementierten Reasonern nötig sind. Insbesondere beinhaltet dieses Paket das Java-Interface `CakeReasoner`, das in Abbildung 11 abgebildet ist. Dieses zentrale Interface muss von jedem Reasoner implementiert werden. Die Implementierung und weitere Verfeinerung eines Reasoners geschieht in einem eigenen Reasonerpaket, welches nicht mehr im Kern der Logikschicht liegt, sondern im **`reasoner`**-Teil, der im unterem Teil in Abbildung 10 abgedruckt ist.

Die im Interface von Abbildung 11 gezeigten Methoden dienen dem Zugriff auf diesen Reasoner seitens der `ReasonerManager`-Klasse und sind nötig für eine korrekte Interaktion. Dieser Abschnitt

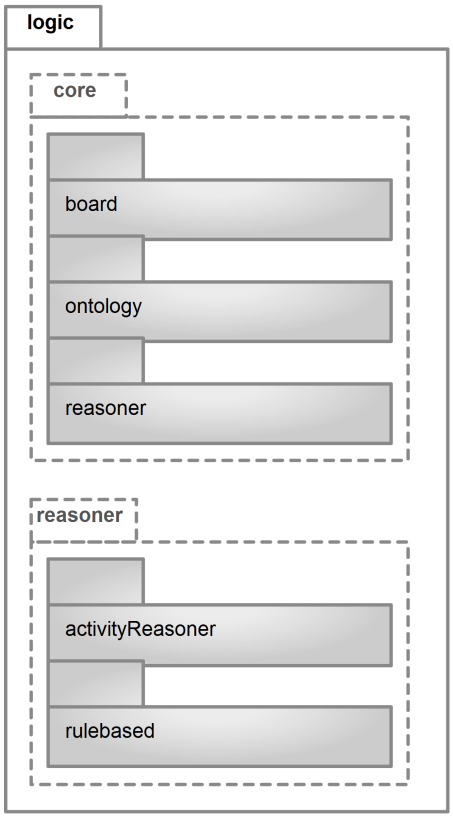

Abbildung 10: Pakete in der Logikschicht

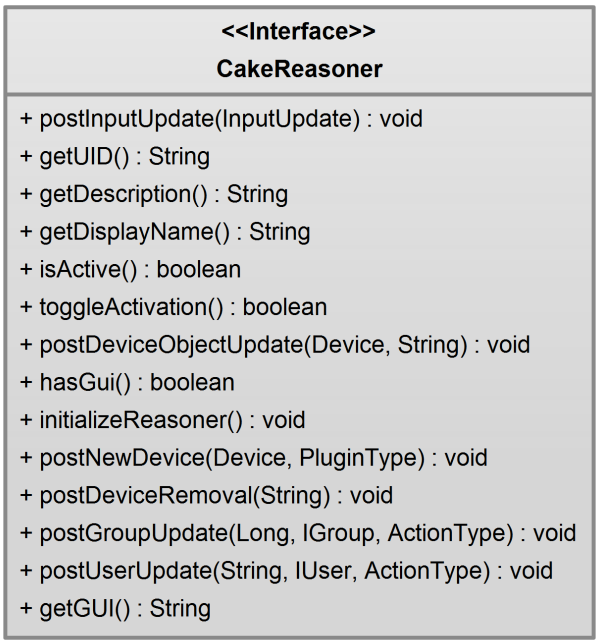

Abbildung 11: Das CakeReasoner-Interface, welches jeder neue Reasoner implementieren muss

vertieft keine Details der einzelnen Methoden. Grundlegend dient die eine Hälfte der Methoden dem Abruf von Metadaten des Reasoners, die anderen Methoden entsprechen *Observer*-Methoden und registrieren Veränderungen in der Logik und werden bei neu eingehenden Meldungen und Sensoreingaben aufgerufen (Abonnement und Registrierung des Reasoners vorausgesetzt).

3.2 Weka

Die Analyse ergab die Festlegung auf das Machine Learning als Reasoningverfahren. Zudem wurde zu diesem Zeitpunkt bereits ein interessantes und für den Einsatz in CAKE gut verwendbares Framework analysiert: Weka[1]. Weka ist eine von der namensgebenden University of Waikato in Java geschriebene Bibliothek, die frei zur Verwendung zur Verfügung steht. Ein umfassendes Handbuch beschreibt dazu alle Funktionalitäten dieses mächtigen Frameworks (siehe Bouckaert et al., 2013). In dieser Sammlung an Funktionen sind diverse Algorithmen des Machine Learning implementiert. Weka bietet zur Verwendung eine grafische Oberfläche - dessen unterschiedliche Anwendungsmöglichkeiten durch Hall et al. (2009) beschrieben werden - aber ebenso auch die Interaktion auf der Befehlszeile oder direkt in Java-Code an (vergleiche hierzu Witten et al., 2000). Die Umsetzung eines einfachen Klassifizierungs- oder Reasoningbeispiels gestaltet sich zu Testzwecken recht unkompliziert. Obwohl diese Arbeit nur einen Teilbereich von Weka verwendet und betrachtet, verfügt das Framework über ein mächtiges Repertoire an Möglichkeiten, Daten zu verarbeiten, zu analysieren

[1] Waikato Environment for Knowledge Analysis

und zu verwenden. Für die Entwicklung eines CAKE Reasoners bietet sich Weka zudem an, da es eine gute Trennung zwischen Daten- und Verarbeitungsschicht ermöglicht und außerdem verschiedene Schnittstellen für die Speicherung von Daten, aber auch für die Ausführung von Berechnungen, anbietet.

Die folgenden Abschnitte umfassen die verschiedenen Schritte zum Bauen eines Models. Dazu zeigt Abbildung 12 eine Übersicht über die drei wesentlichen Teilaspekte. Anschließend wird der Einsatz eines solchen Models kurz aufgezeigt.

Abbildung 12: Weka Pipeline, die Datensätze zu einem Model verarbeitet

3.2.1 Datenhaltung

Weka ist ein Offline Learner, das bedeutet, die Daten, die Weka zum Klassifizieren nutzt, liegen bereits vor und wurden zuvor gespeichert. Weka entwickelt sein Model aus einem gegebenem Satz aus Daten. Genauer werden diese Datensätze *Instanzen* genannt. Eine Instanz beschreibt genau ein Eingabetupel, zudem auch das zugehörige Klassenattribut gehört. Daher ist eine Instanz mit einer Zeile in einer Datenbank vergleichbar. Die Spalten sind hierbei die Attribute.

Ein Attribut ist ein Wert eines bestimmten Datentyps, z.B. treten hier `boolean`, `string`, `integer` oder `float` auf. Diese können beliebig definiert oder durch gegebene diskrete Mengen beschränkt sein. So könnte eine Wetterprognose durch einen String beschrieben werden, der aber fest zu wählen ist aus der Menge von Attributwerten wie {*"Sonnig","Bewölkt","Regnerisch","Schnee"*}. Ein Beispiel für eine kleine Mengen an Instanzen bringt Tabelle 1 mit einer Auswahl von Beispieldatensätzen. Von den fünf Attributen ist das letzte Attribut *play* das Klassenattribut, welches dasjenige Attribut ist, das in Zukunft aus den anderen erschlossen werden soll.

No.	outlook	temperature	humidity	windy	**play**
1	sunny	85	85	`FALSE`	no
2	sunny	80	90	`TRUE`	no
3	overcast	83	86	`FALSE`	yes
4	rainy	70	96	`FALSE`	yes
5	rainy	68	80	`FALSE`	yes

Tabelle 1: Beispieldaten von Weka für die Entscheidung, ob bei aktuellem Wetter draußen gespielt werden kann

Für den eigentlich Prozess des Ladens und Parsens der Datensätze bietet Weka verschiede Schnittstellen der Speicherung an. Es gibt zwei Dateiformate, die Weka lesen und in den aktuellen Programmspeicher übernehmen kann. Darüber hinaus besteht die Möglichkeit, die Datensätze auch in

einer Datenbank zu speichern, wie es je nach Aufgabenbereich unterschiedliche Vor- und Nachteile bietet. In diesem Abschnitt beschränkt sich die Darstellung auf die Dateiformate und grenzt sich somit von der Datenbankspeicherung ab. Die Dateiformate ähneln sich semantisch stark und werden in Grundsätzen diskutiert.

Die gängige Form von gespeicherten Instanzen sind die *.arff*-Dateien. Dies sind einfache Textdokumente, in denen nach einem gewissen Schema die Datensätze gespeichert vorliegen. Es wird unterschieden zwischen dem Headerbereich und dem Bodybereich. Der Header beschreibt die aktuelle Relation und deklariert diese für die später folgende Verwendung der Bodydaten. Hierbei müssen alle verwendeten Attribute Reihenfolge-erhaltend deklariert werden. Im Anschluss werden im Bodybereich zeilenweise alle Instanzen abgelegt, indem die Werte für die Attribute mit Kommata getrennt aneinander gereiht werden. Folgender Quelltext 2 zeigt das obige Beispiel aus Tabelle 1:

```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature real
@attribute humidity real
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,85,85,FALSE,no
sunny,80,90,TRUE,no
overcast,83,86,FALSE,yes
rainy,70,96,FALSE,yes
rainy,68,80,FALSE,yes
```

Quelltext 2: Ausschnitt aus einer *.arff*-Datei von Weka

Eine andere Darstellungsform der *.arff*-Datei ist die *.xrff*-Datei, die sich nur dadurch unterscheidet, dass ihre innere Syntax dem XML[2]-Schema entspricht. Es liegt eine DTD[3] vor, die den Aufbau festsetzt und somit die Datei syntaktisch auf Korrektheit prüfen kann. Semantisch betrachtet besitzen aber beide Dateiformate die gleiche Aussagekraft. Bei der Verwendung einer Datenbank lägen die Instanzen ebenso zeilenweise vor und wären durch die Spaltendeklaration äquivalent zum Headerbereich der Dateien beschrieben.

Im Weka Java Framework lassen sich *.arff*-Dateien problemlos über eine Klasse einlesen und in den Programmspeicher laden. Aus diesem Zweck bieten diese Dateien einen schnellen Zugang und eine einfache Verwendung durch den bereits implementierten Parser. Für die Entwicklung eines Reasoners für CAKE könnte allerdings auch die Datenanbindung mittels einer Datenbank von Interesse sein, die aber zunächst aus Komplexitätsgründen der Einrichtung nicht betrachtet wird.

[2] Extensible Markup Language
[3] Document Type Definition

3.2.2 Preprozessierung der Daten

Sind die Instanzen geladen, so können diese Daten noch weiter verarbeitet werden. Die *Preprozessierung* bietet verschiedene sogenannte Filter an, mit denen die gegebene Datenmenge nun verändert werden kann. Dabei ist es denkbar, die vorhandenen Instanzen zu manipulieren, zu verringern oder auch zu vergrößern.

Datensätze können invertiert werden. In diesem Fall würden alle Attribute, die einen umkehrbaren Wert besitzen (wie boolsche Werte - `TRUE` oder `FALSE`), direkt invertiert werden. Dies kann nützlich sein, um gewisse Features der Instanzen, d.h. deren Besonderheiten oder Eigenschaften, zum Vorschein zu bringen, die zuvor unscheinbar geblieben sind, etwa, weil sie in zu geringen Zahlen im Datenbestand vorlagen.

Genauso können Minderheiten an Datensätzen interessant sein, gehen aber aufgrund ihrer geringen Stückzahl im Datenbestand verloren, wenn zu viele dominante Instanzen anderer Art vorliegen. In diesem Fall kann beim Preprozessieren eine Art Minderheit von Instanzen künstlich vermehrt werden, sodass die numerische Mächtigkeit dieser Gruppe steigt.

In der Verarbeitung der Daten stehen Möglichkeiten zur Verfügung, die Datenmengen mit einem gewissen Fehler zu betrachten, indem z.B. Rauschen eingefügt wird. Dadurch lassen sich möglicherweise Fehleranfälligkeiten herausfinden. Es ist auch möglich, die Instanzen einem Mittelwert näher anzupassen, um weniger Abweichungen zu erhalten.

Für die Verwendung einiger Klassifizierungsalgorithmen dürfen nur Instanzen mit bestimmten Attributen vorliegen. So kann es vorkommen, dass Werte in Fließkommazahlen auftreten. Sind allerdings nur Ganzzahlen erlaubt, können Preprozessierungsfilter alle Fließkommazahlen zuvor auf Ganzzahlen runden. Analog dazu wandelt der `NominalToBinaryFilter` für den Einsatz eines ML-Verfahrens alle Attribute in binäre und numerische Datentypen um, damit diskrete mehrwertige Attributtypen herausfallen. Dieser Filter wird zusammen mit anderen von Witten et al. (2000) beschrieben.

3.2.3 Bauen des Klassifizierers

Die letzte Phase, die in Abbildung 12 ersichtlich wird, ist das eigentliche Erzeugen oder Bauen des Klassifizierers von Weka (oder das Model). Dazu werden die nun vorliegenden Instanzen nach einem gegebenem Verfahren des Machine Learning zu dem Klassifizieren kompiliert. Je nach Einsatz und Beschaffenheit der Daten kommen hier verschiedene Algorithmen in Frage. Weka implementiert eine große Bandbreite dieser Algorithmen, die in Java direkt verwendet werden können. Im Abschnitt 2.5 wurde dazu bereits allgemein eine Auflistung der verschiedenen Klassen von Machine Learning Algorithmen vorgenommen. Diese Algorithmen und Klassen liegen hierbei ebenso in Weka vor. Für einen Einblick in die genauere Funktionsweise kann dieser Abschnitt also nochmals betrachtet werden. Die Verwendung einiger der dort aufgeführten Klassifizierer werden ebenfalls von Othman & Yau (2007) im Einsatz mit Weka beschrieben und bei der technischen Effektivität von Krebserken-

nung untersucht.

Die Ausgabe der Klassifizierung ist dann im Anschluss ein Model, auf welches unbekannte, neue Instanzen angewandt werden können, um das Klassifizierungsattribut (also das unbekannte Attribut) zu berechnen. Dazu skizziert Abbildung 13 den Ablauf. Dem Model wird eine Instanz übergeben, in der das Klassifizierungsattribut nicht gesetzt ist. Das Model wird nun nach seinen Vorgaben eine Berechnung durchführen und gibt eine Instanz mit einer Belegung der zuvor unbekannten Variablen zurück. Dabei wird immer ein Wert berechnet, der ebenso falsch ermittelt werden kann. Die Aufgabe hierbei ist es, den effektivsten Machine Learning Algorithmus zum Bauen des Models zu finden.

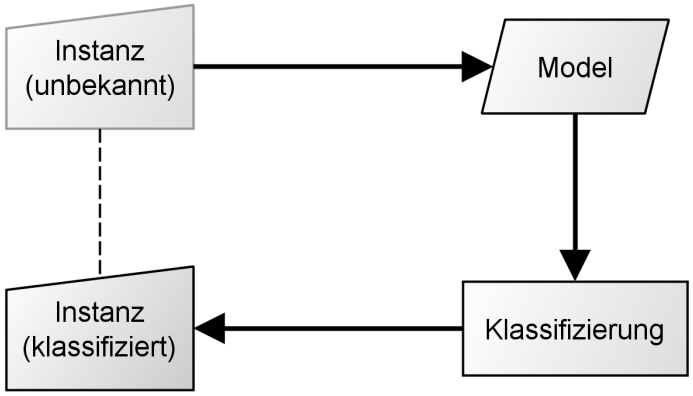

Abbildung 13: Eine Klassifizierung mittels des erzeugten Models

3.3 Umsetzung Reasoner

Die vorangegangenen Abschnitte beschrieben die Architektur von CAKE und ebenso Weka als Framework für das Machine Learning. Für die Umsetzung des Reasoners werden diese Konzepte zusammengeführt und der Reasoner als neues Modul integriert. Die Schnittstellenarchitektur in der Logikschicht von CAKE stellt die Verwendbarkeit des Reasoners nach der Aufgabenanalyse sicher, sodass die Reasonerklassen nur die benötigten Schnittstellen implementieren müssen. Alles weitere geschieht losgelöst von der bestehenden Architektur und wird im Folgenden konzipiert.

Aus der Architekturerhebung von Abschnitt 3.1 wird die Position in der CAKE-Logischicht deutlich. Abbildung 14 zeigt hierzu nochmals die zu betrachtende Stelle. Für die Basis und Logik des Reasoner entsteht ein neues Java-Paket im `reasoner`-Unterpaket der Logikschicht. Dieses teilt sich in drei grobe Bereiche innerhalb des Pakets auf: das Model, die Datenhaltung und das Framework Weka. Mit der fertigen Modellierung müssen Datensätze für das Offline Machine Learning gesammelt und auch gespeichert werden. Dazu ist in CAKE eine Speicherform sowie Interaktion mit dem Machine Reasoner vorzusehen. Eine Anbindung und Implementierung der Schnittstellen, die durch den Reasoner und den `ReasonerManager` deklariert werden, ist dazu ebenso notwendig, um Daten zu erhalten. Das Weka Framework lässt sich problemlos in das Java-Projekt einbinden. Dennoch kön-

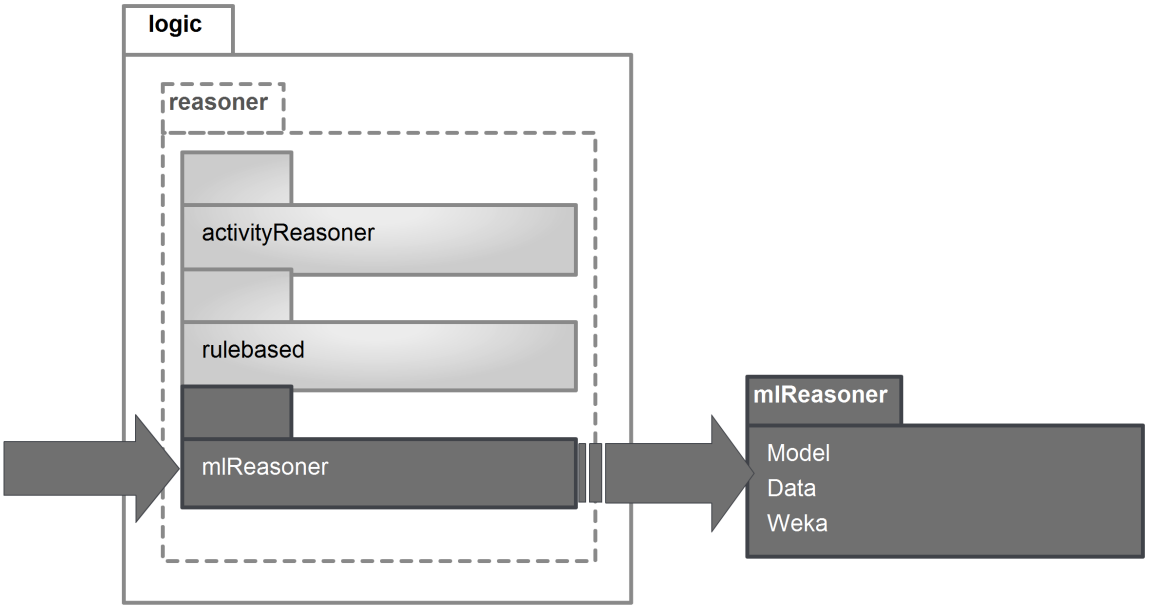

Abbildung 14: Die Ansiedlung des Machine Learning Reasoners

nen die Klassen nicht direkt genutzt werden, ohne sie auf die Bedürfnisse der Interaktion mit CAKE anzupassen. Dazu soll es Wrapper-Klassen für die Weka-Klassen und auch gewisse abstrahierende Klassen geben, um Aufgaben und Logik innerhalb des Reasoners sinnvoll und später erweiterbar umzusetzen.

3.3.1 Modellierung

Ein Reasoner in CAKE hält neben den zu implementierenden Schnittstellen einige Metadaten vor, auf die seitens der Logikschicht ein lesender Zugriff ermöglicht werden muss. Da diese Daten spezifisch auf eine Instanz eines Reasoners angepasst sind, müssen bei der Verwendung eines abstrakten Reasoners diese Metadaten ebenso von den konkretisierten Reasonerklassen definiert werden. Diese Daten lassen sich zusammen mit der eigentlichen Verwendung des Reasoners modellieren. Die Verwendung von XML-Dokumenten bietet sich hierbei an. Ein solches Dokument könnte als komplette Beschreibung einer konkreten Reasonerinstanz gelten und damit den Entwicklungsaufwand gering halten. Die Benutzergruppe der Entwickler, die in der Benutzeranalyse aus Kapitel 2 als zentrale Anwendergruppe betrachtet wird, versteht den Umgang mit XML-Dokumenten und kann mit Hilfe einer kleinen Dokumentation zur Verwendung dieses Dokumentes auch ohne größere Einarbeitungszeit gewünschte Ziele umsetzen.

Das in Quelltext 3 gezeigte XML-Dokument stellt eine beispielhafte Modellierung einer Reasonerinstanz dar.

`<declaration>`. In dem Deklarationsbereich können Metadaten festegelegt werden, dazu zäh-

```
<reasoner>

 <declaration>
  <name>Temperaturregler</name>
  <description>Regelung der Heizung</description>
  <sensors>
   <name>sensor_uuid01</name>
   <name>sensor_uuid02</name>
  </sensors>
  <actuators>
   <name>actuator_uuid01</name>
  </actuators>
 </declaration>

 <model>
  <field>
   <label>temperature</label>
   <type>numeric</type>
   <class>false</class>
  </field>
  <field>
   <label>heating</label>
   <type>nominal</type>
   <nominals>
    <option>yes</option>
    <option>no</option>
   </nominals>
   <class>true</class>
  </field>
 </model>

 <reasoning>
  <default_reasoner>NaiveBayes</default_reasoner>
 </reasoning>

</reasoner>
```

Quelltext 3: XML-Beispieldokument für die Modellierung einer Reasonerinstanz

len der Reasonername oder die Beschreibung (optional). Ebenso gehört zur Deklaration, dass dem Reasoner mitgeteilt wird, welche Sensoren am CAKE-System abonniert werden und welche Aktuatoren die Reasoningergebnisse dieser Reasonerinstanz benötigen.

<model>. In der Modellierung werden die Eingangswerte, die der Reasoner speichern, laden und zur Klassifizierung verwenden kann, bestimmt. Dieser Bereich macht den größten Teil der Definition einer Reasonerinstanz aus und geht aus den Benutzerszenarien aus Abschnitt 2.3 hervor. Die Hauptaufgabe bei der Erschließung einer Anwendungsdomäne ist hierbei die Zusammenführung von Sensorwerten, wie sie in verschiedenen Datenformaten vorliegen können. Basierend auf dieser Modellierung wird später ein Weka-Klassifizier erzeugt, der eingehende Datensätze zuordnen und vorausberechnen kann. Jedes `<field>` ist dabei ein Attribut, welches in Abschnitt 3.2 mit Weka eingeführt wurde. Angenommen werden können Daten von den Typen

1. `numeric` - Zahlenwerte, erlaubt sind Integer oder Real-Zahlen
2. `string` - Zeichenketten, die insbesondere bei Texterkennungsalgorithmen interessant sind
3. `date` - Datumsobjekt in Weka, das wiederum selbst als Zeichenkette abgelegt wird
4. `nominal` - Nominale Werte sind Arrays, von denen jeweils ein Wert einer gegebenen diskreten Menge angenommen werden kann

<reasoning>. Im letzten Bereich der XML-Datei wird die Art des Reasonings bestimmt. Dieses Vorgehen beschreibt das primäre Szenario, in welchem der Entwickler für CAKE seine zu entwickelnde Reasonerinstanz formativ evaluiert und dabei möglicherweise mehrmals das Verfahren des Reasonings ändert. Ziel dieser Arbeit ist es, verschiedene Machine Learning Verfahren als so zu bezeichnende "Default_Reasoner" bereitzustellen. Alternativ soll dieser Bereich vorsehen, erfahrene Entwickler mit Weka ihre eigenen ML-Verfahren in eingebettetem XML zu realisieren. Mit Weka ist es möglich, nach der in Quelltext 4 gezeigten DTD einen XML-Klassifizierer zu definieren und diesen über Java einzubinden. Im Kapitel 4 wird die Verwendung dieser Schnittstelle beispielhaft gezeigt.

```
<!DOCTYPE object
[
   <!ELEMENT object (#PCDATA | object)*>
   <!ATTLIST object name      CDATA #REQUIRED>
   <!ATTLIST object class     CDATA #REQUIRED>
   <!ATTLIST object primitive CDATA "yes">
   <!ATTLIST object array     CDATA "no">
]
>
```

Quelltext 4: DTD für XML-Klassifizierer in Weka

Durch die Modellierung des Reasoners in einem XML-Dokument, welches Metadaten, Ein- und Ausgabedeklarationen und Weka-Klassifizierungsverfahren beschreibt, erfüllt der Reasoner bereits einen

Großteil der Interaktionsaufgaben zwischen einem Entwickler für CAKE und dem Reasoner. Schon durch diese Konzeption scheinen einige in Abschnitt 2.6 aufgezeigten Features erfüllt zu werden: Mehrfach-Instanziierung des ML-Reasoners, Flexible Modellierung von Variablen und Ausgaben, Konfiguration Reasoning-Verfahren.

3.3.2 Datenhaltung

Das Model von Weka, also der Klassifizierer, setzt sich aus den jeweiligen Instanzen und dem ML-Verfahren zusammen. Wie bereits beschrieben handelt es sich bei diesem Reasoningansatz um einen Offline Learner, sodass alle Datensätze lokal gespeichert werden müssen. Für eine einfache Einrichtung und Zugriff auf die Datensätze bietet sich bei der eingehenden Datenmenge einiger Sensoren eine Textdatei als Datenspeicherformat an.

Die beiden Quelltexte 5 und 6 zeigen die gleiche Instanz, wie sie in den beiden von Weka unterstützten Textdateiformaten darzustellen sind. Obgleich die XML-basierte Variante einen strukturierteren Eindruck erweckt, beinhaltet diese Struktur einen Nachteil. Die Schachtelung einer XML-Datei unterstützt nicht das Einfügen eines neuen Datensatzes ohne auf die Struktur zu achten und innerhalb der bestehenden Zeilen neue Werte einzugliedern. Im Falle der einfachen *.arff*-Datei lässt sich eine weitere Zeile an das Dokument anhängen. Betrachten wir den Speicheraufwand bei einer großen Menge anstehender Daten, so produziert die *.xrff*-Datei viel größeren redundanten Text durch die XML-Tags. Nachteil des *.arff*-Formats ist dafür, dass die Syntax nicht geprüft werden kann. Da nur der Reasoner Schreibrechte auf die Datei bekommen sollte, kann aber von korrekter Dateiführung ausgegangen werden.

```
@data
5.1,3.5,1.4,0.2,yes
```

Quelltext 5: Eine Instanz als *.arff*-Datei

```
<body>
 <instances>
  <instance>
   <value>5.1</value>
   <value>3.5</value>
   <value>1.4</value>
   <value>0.2</value>
   <value>yes</value>
  </instance>
 </instances>
</body>
```

Quelltext 6: Eine Instanz als *.xrff*-Datei

Aufgrund dessen wird der Reasoner auf das Speicherformat *.arff* zurückgreifen. Der Speicherort dieser Datei bestimmt sich daraus, wo der Reasoner möglichst exklusive Schreib- und Leserecht inne hat. Analog zum Speicherort des `RuleBasedReasoners` findet sich dieser im `res`-Verzeichnis von CAKE. Der Reasoner legt seine Regeldatei in folgendem Pfad ab:

```
res/reasoner/rulebased/[uuid_des_Reasoners]/
```

Da jeder Reasoner in CAKE eine eigene `UUID` erhält, lässt sich diese Aufteilung der Ordnerstruktur auch auf den ML-Reasoner anwenden. Damit kann jede Instanz einen für sich seperaten Datensatz anlegen, der auf die jeweils eigene Datenmodellierung passt. Da jede Instanz neben der Datenspeicherung auch die Modellierung (siehe Quelltext 3) speichern sollte, kann diese in dem gleichen Verzeichnis abgelegt werden. Bei der Umsetzung des Reasoners in Kapitel 4 muss eine Versionierung vorgesehen sein, damit Änderungen an dieser Modellierungsdatei möglich sind. Die Instanz-spezifische Dateistruktur wäre folgende:

```
res/reasoner/mlReasoner/[uuid_des_Reasoners]/instances/reasoner.xml
res/reasoner/mlReasoner/[uuid_des_Reasoners]/data/reasoner.arff
```

3.3.3 Architektur des ML-Reasoners

Abschließend wird die konzipierte Architektur beschrieben, die die Funktionsweise, -umfang und deren Zusammenhang erklären soll. Nach der Architekturanalyse aus Abschnitt 3.1 existiert das `reasoner`-Paket in der Logikschicht. Hier entsteht nun mit dem `mlReasoner`-Paket das neue Modul.

```
logic.reasoner.mlReasoner.MachineLearningReasoner
logic.reasoner.mlReasoner.ModelInstanceManager
logic.reasoner.mlReasoner.Model
logic.reasoner.mlReasoner.io.XMLParser
logic.reasoner.mlReasoner.builders.Builder
```

Diese Hauptklassen fassen den Umfang des neuen Reasoners im Wesentlichen zusammen und sollen dem Konzept und der Grundlage für die in Kapitel 4 folgende Realisierung des Reasoners dienen. Eine visuelle Darstellung in Form eines UML-Klassendiagramms findet sich in Abbildung 15. Im Folgenden findet eine kurze Beschreibung der Klassen in der angedachten Funktionsweise statt, die im Anschluss die resultierende Funktionalität im Gesamtbild darstellt.

`MachineLearningReasoner`. Die Hauptklasse für den Reasoner. Diese implementiert das allgemeine `CakeReasoner`-Interface und stellt damit sämtliche Verbindungen zu der restlichen CAKE-Logikschicht her. Diese Klasse erzeugt die benötigte Instazt des im folgenden Absatz beschriebenen

ModelInstanceManager und leitet von diesem Zeitpunkt an sämtliche eingehenden Sensorwerte an diesen weiter. Außerdem wird der MachineLearningReasoner benötigt, wenn Aktuatorwerte zum Versand bereit stehen.

ModelInstanceManager. Um alle Reasonerinstanzen zu initialisieren, vorzuhalten und Sensorwerte zielgerichtet an diese zu verteilen, entsteht der ModelInstanceManager. Er erzeugt zu Beginn Model-Objekte aus XML-Dateien und speichert diese in seiner internen Datenstruktur. Die Aufgabe des Managers ist es nach erfolgreicher Initialisierung aller Reasonerinstanzen, die eingehenden Sensorwerte an entsprechende Model-Instanzen weiterzureichen und von diesen Reasoningergebnisse abzufragen, die dann wiederum zum MachineLearningReasoner geleitet werden, um Aktuatoren diese mitzuteilen.

Model. Das Model soll zum einen alle Metadaten vorhalten, die in Unterabschnitt 3.3.1 in Form des XML-Dokuments beschrieben wurden. Zudem entwickelt das Model beim Aufruf der Methode buildClassifier() den Weka-Klassifizierer und kann ab diesem Zeitpunkt eine beliebige eingehende Instanz klassifizieren.

XMLParser. Um das Model mit den Metadaten zu füllen, müssen die bereits beschriebenen XML-Dokumente validiert und eingelesen werden. Aus Abstraktionsgründen liegt daher die XMLParser-Klasse vor, deren Implementierung der Parsing-Funktionen immer dem aktuellen Stand des XSD[4]-Dokuments entsprechen müssen. Das XSD-Dokument (zu finden im Anhang unter Quelltext 16, 17) beschreibt hierbei die Beschaffenheit der einzulesenden XML-Datei und stellt damit eine Validitätsprüfung zur Verfügung.

Builder. Ein eigenes Unterpaket stellt das builders-Paket dar. Hier liegt die abstrakte Builder-Klasse. Diese dient dazu, aus einem übegebenen Model einen Weka-Klassifizierer zu erzeugen. Das Model selbst soll standardmäßig diesen Prozess einleiten. Bei der Erzeugung eines Klassifizierers durchläuft die Methode build() drei Schritte: Laden der Instanzdatenbank, Preprozessieren der Instanzdaten, Bauen des Klassifizierers mit gegebenem Machine Learning Algorithmus in Weka. Diese drei Schritte bilden abstrakte Methoden in Java, die von zu entwickelnden Klassen konkretisiert werden. Somit ist es möglich, die in XML zu bestimmenden *default_reasoner* hier zu implementieren und vom restlichen Reasoningprozess getrennt in einem Java-Paket zu entwickeln. Näheres hierzu wird in der Realisierung (Kapitel 4) deutlich.

Die hier aufgezeigte grobe Architektur bietet bereits eine gewisse Trennung von Arbeitsschritten und Interaktion mit CAKE und dem Weka Framework. Daher bietet sich die Architektur gut für Erweiterungen oder Code-Reengineering an. Dies erfüllt das Feature: Erweiterbarkeit Schnittstellen aus der Analysephase (Kapitel 2, Abschnitt 2.6). Ebenso sorgt diese Art der Model-Instanziierung dafür, dass der Reasoner zu jeder Zeit rekonfiguriert werden kann. Handelt es sich dabei um eine Änderung von Metadaten oder der Wahl des Machine Learning Algorithmus, ebenso wie ein anders-artig gewähltes Preprozessieren der Daten, so kann der Datensatz in Form der *.arff*-Datei komplett bestehen

[4] XML Schema Definition

bleiben. Wird die Modellierung der Attribute geändert, muss diese Datei angepasst oder komplett neu erstellt (und somit möglicherweise geleert) werden. Die hier beschriebenen Änderungen werden wirksam, wenn der Machine Learning Reasoner im Java-Programm neu instanziiert wird. Somit folgt die Anforderung an den Reasoner, Konfiguration und Adaptierbarkeit zur Laufzeit vorzunehmen. Eine Neuinstanziierung der Reasonerinstanz ist zur Programmlaufzeit von CAKE denkbar und kann somit umgesetzt werden.

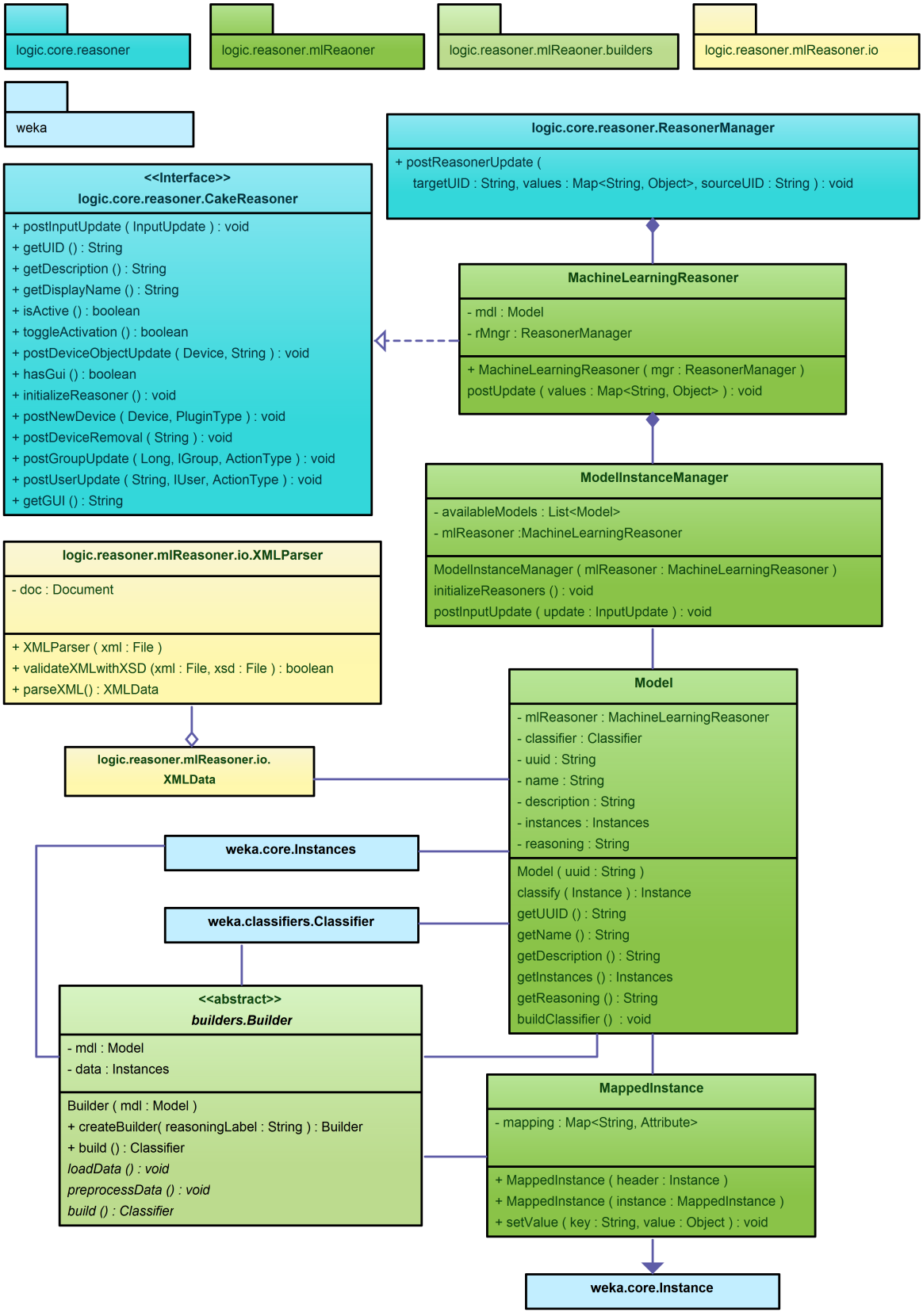

Abbildung 15: Architektur des Machine Learning Reasoner Packages

4 Realisierung

Um die Realisierung des Reasonersystems nachvollziehbar darzustellen, bildet dieses Kapitel einen großen Teil der Umsetzung aus der Sicht von Programm-internen Prozessen ab. Diese Prozesse beschreiben, wie die einzelnen Schritte eines bestimmten Ablaufs im Machine Learning Reasoner aussehen, der durch CAKE mittels einer Benutzerinteraktion ausgelöst wird. Dieses Kapitel verdeutlicht Strukturen und Implementierungsdetails, ebenso werden Begründungen für getroffene Entscheidungen in der Realisierung angeführt. Im Anschluss an diese Prozesse findet sich eine Erklärung des `builders`-Pakets, welches im Wesentlichen verantwortlich ist für die Weiterentwicklung von Reasoningverfahren im Machine Learning Reasoner. Während der Umsetzung dieser Komponenten validiert die formative Evaluation in Abschnitt 4.5 die Funktionalität und beschreibt mögliche Änderungen in der Architektur.

Es gibt drei Prozesse, die zu beschreiben sind. Der *Initialisierungsprozess* behandelt das Starten des Machine Learning Reasoners bis zur vollen Einsatzbereitschaft für das Reasoning. Im *Reasoningprozess* selbst geht es darum, eingehende Sensorwerte mittels Reasoning in Aktuatorwerte umzuwandeln. Abschließend können Aktuatoren Feedback an den Reasoner senden, ob das Reasoning korrekt oder fehlerbehaftet war. Den daraus ableitbaren Lernprozess für den Reasoner stellt der *Feedbackprozess* dar, der auch auf eine Veränderung in der Kommunikationsschicht von CAKE eingeht.

4.1 Initialisierungsprozess

Die Instanziierung des Machine Learning Reasoners geschieht wie in Abschnitt 3.1 ausgehend von der `Whiteboard`-Klasse. Die Hauptklasse `MachineLearningReasoner` registriert sich hierbei am `ReasoningManager` und wird daraufhin initialisiert. Die Initialisierung bewirkt Erzeugung der verwendeten internen Datenstrukturen und führt genauer folgende Schritte durch: Registrierung des `MachineLearningReasoner` als `MessageQueueListener`, Erzeugung der Listen für die Vorhaltung aller aktiver Sensoren und Aktuatoren, Abonnieren sämtlicher Sensoren sowie Erzeugung und Initialisierung des `ModelInstanceManager`.

Dieser Prozess wird in Abbildung 16 gezeigt. Der `ModelInstanceManager` liest entsprechend der Konzeption sämtliche zur Verfügung stehende XML-Dokumente als Instanzen der `Model`-Klasse ein, sofern möglich. Als solche Models deklarierten Dateien gelten alle diejenigen, die im Pfad

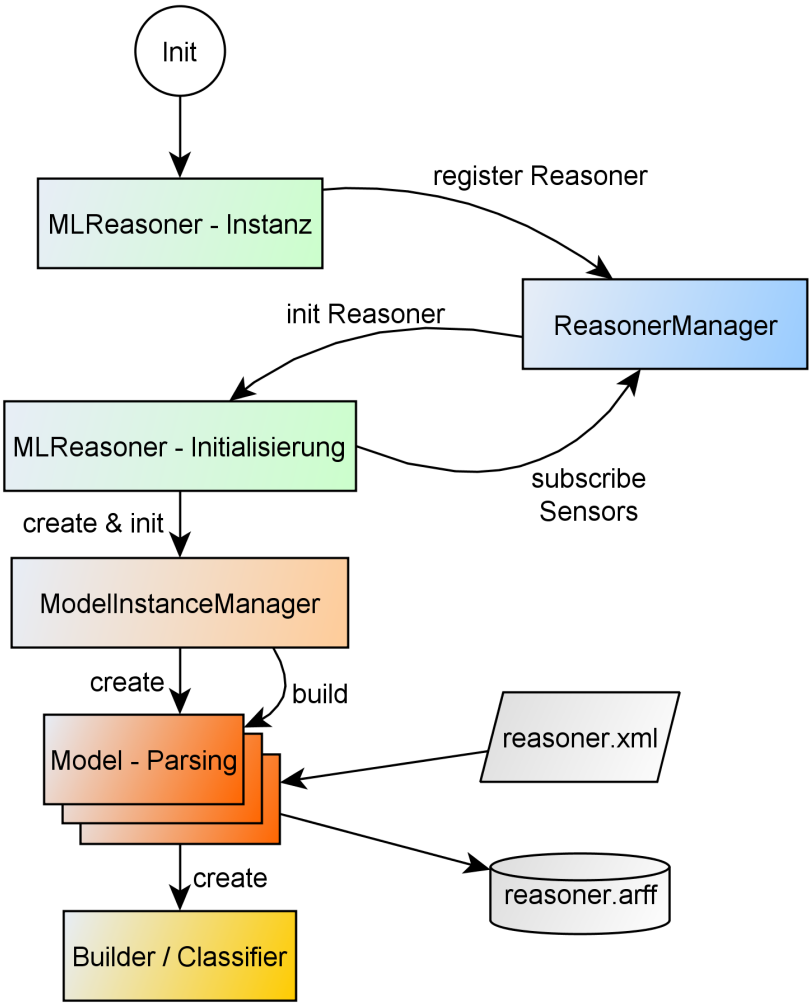

Abbildung 16: Initialisierungsprozess des Machine Learning Reasoners

`res/reasoner/mlReasoner/[uuid_des_Reasoners]/instances/`

liegen. Dabei erfasst der Manager alle Modellierungsdateien und validiert diese gemäß dem vorliegenden XSD-Dokument[1]. Bei Abweichung der erlaubten Syntax wird das Model ignoriert und eine Mitteilung über den Logger ausgegeben. Ist das Dokument valide, so übergibt dieser das Dokument der `Model`-Klasse. Hierbei wird die Datei über den `XMLParser` eingelesen und der extrahierte Datensatz im `Model` gesetzt. Anschließend folgt der Aufruf der `buildClassifier`-Methode durch den `ModelInstanceManager`, die nun den entscheidenden Entwicklungsprozess des Klassifizierers einleitet. Das `Model` verfügt außerdem über die Funktionalität, die gegebene Dateistruktur auf eine vorhandene *.arff*-Datei zu prüfen. Liegt keine vor, legt das `Model` sie an. Weicht die vorhandene Datei vom aktuell eingelesenen Model ab, so wird zwecks Datensicherung keine Änderung vorgenommen, sondern das aktuelle `Model` übersprungen und nicht am System zum Reasoning bereit stehen.

[1] Speicherort des XSD-Dokuments: `res/reasoner/mlReasoner/[uuid_des_Reasoners]/reasoner.xsd`

Der letzte zu erkennende Schritt stellt das eigentliche Bauen des Klassifizierers dar. Hierbei wird der in der entsprechenden XML-Datei angegebene Reasoner instanziiert, wobei jedes im XML-Tag `<default_reasoner>` spezifizierte ML-Verfahren eine eigene Klasse - abgeleitet von der abstrakten Klasse `Builder` - implementiert. Bei der Verwendung einer externen XML-Spezifikation zum Reasoning gemäß der DTD aus Kapitel 3, Quelltext 4 wird der Pfad zu dieser Datei im XML-Tag `<path_to_reasoner>` erwartet und durch die Klasse `XMLBuilder` eingelesen. Unterstützt wird der Pfad zum einen als absoluter Pfad im Dateisystem oder aber als relativer Pfad ausgehend von:

```
res/reasoner/mlReasoner/[uuid_des_Reasoners]/models/
```

Die Verwendung des `builders`-Pakets und der damit verbundenen Klassen werden seperat im Abschnitt 4.4 beschrieben.

4.2 Reasoningprozess

Wann immer ein Sensor Werte an das CAKE Whiteboard sendet, leitet dieses die Werte direkt an den `ReasonerManager` weiter, der im Weiteren jeden entsprechend abonnierten Reasoner diese übergibt. Der `MachineLearningReasoner` erfragt auf diese Weise alle relevanten Sensoren und wird bei jedem Sensorwert, der gesendet wird, über den Aufruf der Methode `postInputUpdate` mit Übergabe eines Datenobjekts darüber informiert. Dieses Objekt `InputUpdate` beinhaltet neben der Quelle wiederum eine Map, bestehend aus Wertepaaren mit Prefix als Schlüssel und einem Sensorwert als Wert. Im Folgenden tritt das Reasoning in Kraft, welches die so erhaltenen Sensorwerte schrittweise zu einem `ActuatorUpdate`-Objekt umwandelt, das im Anschluss als Reasoningergebnis für den Aktuator dient.

Dieses Vorgehen verdeutlicht Abbildung 17. Sobald der `MachineLearningReasoner` ein solches `InputUpdate` erhält, leitet dieser das Datenobjekt an den `ModelInstanceManager` weiter, der das Update an alle diejenigen vorgehaltenen `Model`-Instanzen übergibt, die den entsprechenden Sensor abonniert haben, von dem das Datenobjekt stammt (Abonnement der einzelnen Models vergleiche Konzept der Modellierung in Abschnitt 3.3 - dargestellt in Quelltext 3). Damit nun das Model das Reasoning übernehmen kann, ist es notwendig, die Daten in ein `Instances`-Objekt aus dem Weka-Framework umzuwandeln. Dieses Objekt enthält im Wesentlichen die gleiche Datenstruktur im Hinblick auf Schlüssel-/Wertepaare wie das `InputUpdate`, zusätzlich aber enthält es genauere Informationen über die Attribute und einen Header, der sich aus der Modellierung der Instanz (Quelltext 3, Bereich `<model>`) ergibt. Ein solches `Instances`-Objekt kann mittels Weka klassifiziert werden.

Die Klassifizierung selbst geschieht durch die in Weka gegebenen `Classifier`. Abbildung 18 beschreibt die sehr einfache Einbindung des Weka-Klassifizierers, der zuvor in Abschnitt 4.1 initialisiert

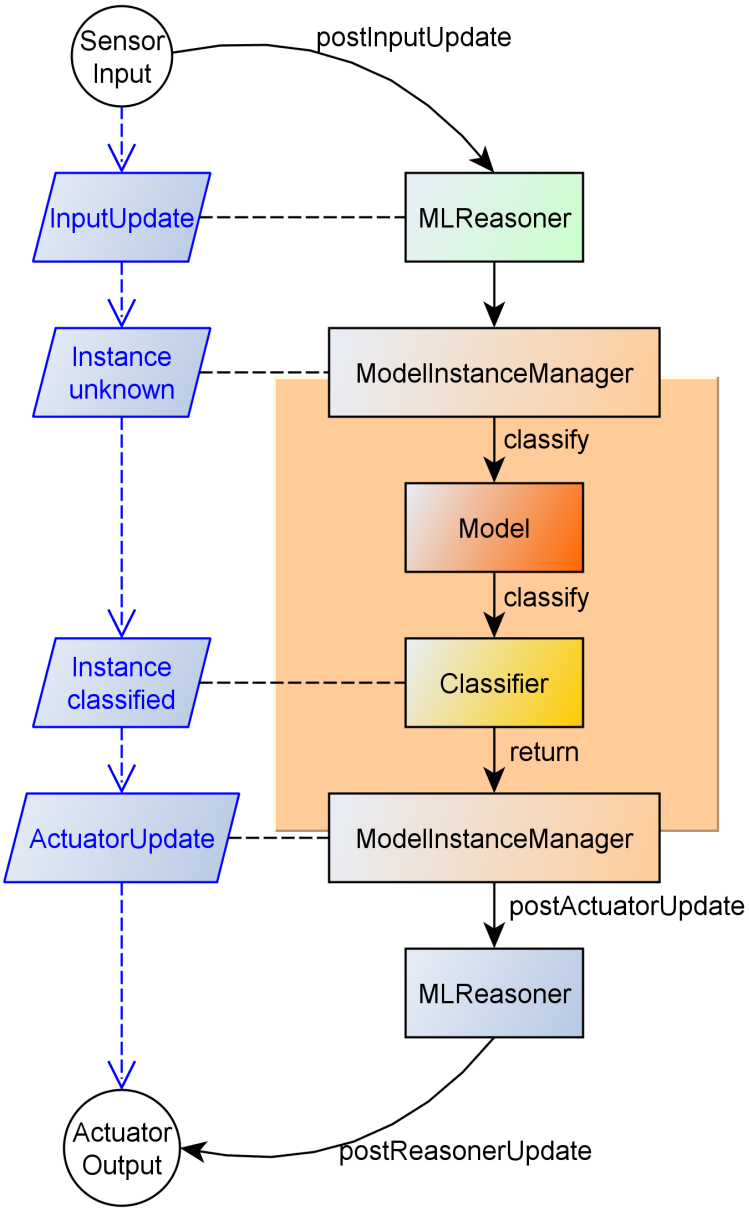

Abbildung 17: Reasoningprozess des Machine Learning Reasoners

wurde. Die Rückgabe an das `Model` ist eine Gleitkommazahl. Diese stellt bei numerischem Reasoning das Ergebnis direkt dar. Handelt es sich um ein nominales Reasoning, d.h. aus einem Array an Werten soll eines ausgewählt werden, so drückt die Zahl den Index des gewählten Objekts innerhalb dieses Arrays dar. Das `Model` setzt diese Zahl als Klassenattribut in dem zugehörigen `Instances`-Objekt und gibt dieses an den `ModelInstanceManager` zurück.

Das so erhaltene und klassifizierte `Instances`-Objekt enthält nun insbesondere das Reasoningergebnis als Klassenattribut. Zuletzt verarbeitet der `ModelInstanceManager` diesen Wert, sodass er ein mögliches nominales Ergebnis aus dem Array extrahiert und somit den echten Wert (also kein

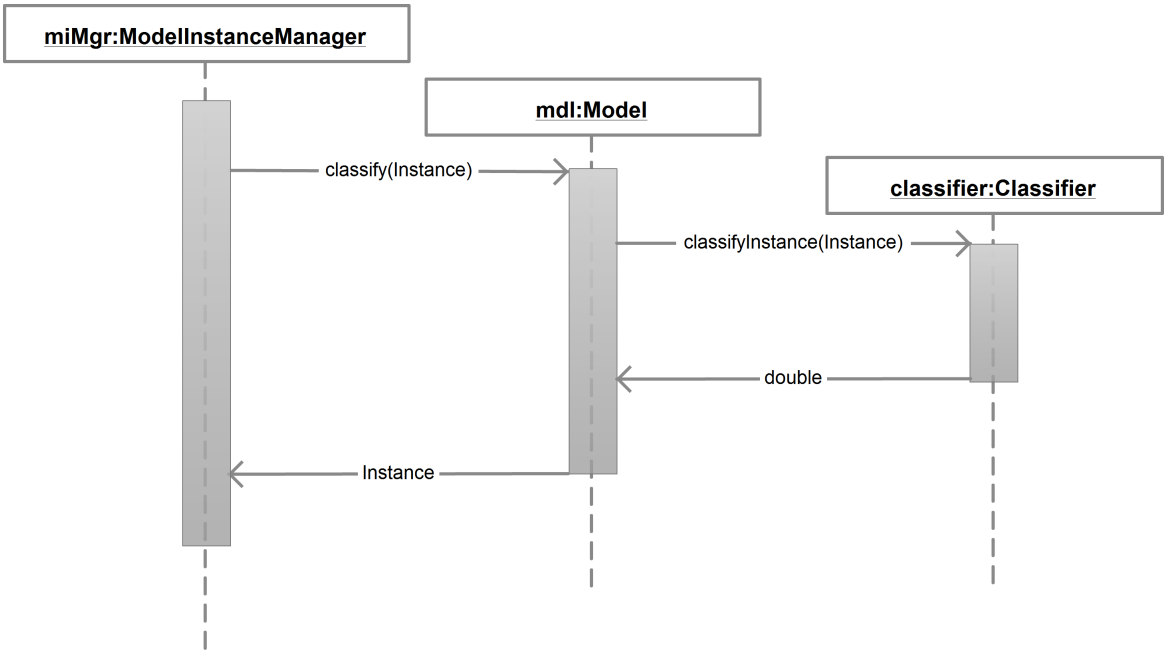

Abbildung 18: Klassifizierung im Model mittels des Weka-Klassifizierers

Index für ein nominales Array, sondern dessen Eintrag) in Form eines `ActuatorUpdate`-Objekts an die Hauptklasse weiterleitet. Dieses Datenobjekt enthält wiederum eine Map, die das klassifizierte Attribut unter gegebenem Schlüssel enthält. Zudem wird das gesamte `Instances`-Objekt in eine Zeichenkette serialisiert mitgereicht. Dessen Verwendung folgt im nächsten Abschnitt 4.3. Tabelle 2 zeigt eine mögliche Instanz einer solchen Map:

Prefix	**Value**
"textform"	"augmented"
"__complete_data__"	"MachineLearningReasoner#Model#test_reasoner_museum #agriculture,20,100,0.6,1,augmented"

Tabelle 2: Beispielbelegung eines `ActuatorUpdate`-Objekts

Dieser Datensatz wird nun so von dem `MachineLearningReasoner` an alle deklarierten Aktuatoren übermittelt, indem er das Objekt über den `ReasonerManager` an die entsprechenden Aktuatoren sendet. Der Aktuator selbst verwendet zur eigenen Verarbeitung dieses Objekts lediglich das Feld, das klassifiziert wurde, in diesem Beispiel *"textform"*.

4.3 Feedbackprozess

Der Machine Learning Reasoner stellt nach der Konzeption den Anspruch, durch Erweiterung seiner Wissensdatenbank stetig weiter zu lernen. Dieser Vorgang kann zu einem gewissen Teil dadurch vorangetrieben werden, dass direktes Feedback vom Aktuator über Reasoningergebnisse an den Reasoner zurückgesendet werden. Durch eine minimale Erweiterung in der Kommunikationsschicht von CAKE ist es nun möglich, im Aktuator eine Nachricht zu schicken, die den neuen Nachrichtentyp `REASONER_FEEDBACK` enthält. Diese Nachricht kann jede Implementierung des `MessageQueueListener` über statischen Zugriff von überall des CAKE-Systems empfangen und verarbeiten.

Die neu hinzugekommene Methode `sendMessageToReasoner` des Kommunikations-Interfaces `ICakeCommunication` erwartet dazu zwei Parameter: die Nachricht als Zeichenkette, wie die bereits in diesem Interface vorliegenden Methoden auch erforderten, und als Zusatz eine Instanz der Klasse `Object`, d.h. der zweite Parameter stellt ein beliebiges Datenobjekt als Anhang an diese Nachricht dar. Der Hintergrundgedanke ist es, das Feedback an den Reasoner auf zwei Arten zu ermöglichen: sendet der Aktuator diese Nachricht an den Reasoner, indem der zweite Parameter auf `null` gesetzt wird, so interpretiert der Machine Learning Reasoner das Feedback als korrekt eingestufte Instanz des Reasonings. Wird hingegen ein Wert eingetragen, wie eine Zeichenkette oder eine Zahl, so gibt der Aktuator damit bekannt, dass die gegebene Sensorbelegung diesen spezifierten Wert als Ergebnis erwartet hätte. Beide Fälle dienen der Vergrößerung der Datenbank im Reasoner, da dieser die korrekte Instanz zusätzlich speichert.

Den übersichtliche Prozess der Feedbacksendung verdeutlich Abbildung 19. Nachdem eine durch die soeben beschriebene Methode gesendete Nachricht in der `MessageQueue` des CAKE-Systems gelandet ist, empfängt der `MachineLearningReasoner` diese durch die Implementierung des oben genannten `MessageQueueHandler`-Interfaces. Nach einer Prüfung, ob die gegebene Nachricht dem Nachrichtentyp `REASONER_FEEDBACK` entspricht, leitet der Reasoner die Nachricht im unveränderten Zustand an den `ModelInstanceManager` weiter. Dieser liest die Nachricht ein, filtert die übergebenen Werte und verpackt diese in einem für Weka geeigneten `Instance`-Objekt. Dieses kann nun im `Model` angenommen und in die zugehörige *.arff*-Datei geschrieben werden. Der Lernprozess ist damit abgeschlossen und der neue Datensatz steht beim nächsten Erzeugen des Machine Learning Reasoners bereit.

Der Wert des *"__complete_data__"*-Felds sieht im Beispiel wie folgt aus und wird im Folgenden weiter beschrieben:

```
MachineLearningReasoner#Model#test_reasoner_museum#
agriculture,20,100,0.6,1,augmented
```

Diese Zeichenkette, die als Beispielbelegung aus Tabelle 2 stammt, wird dem Aktuator bei einem Re-

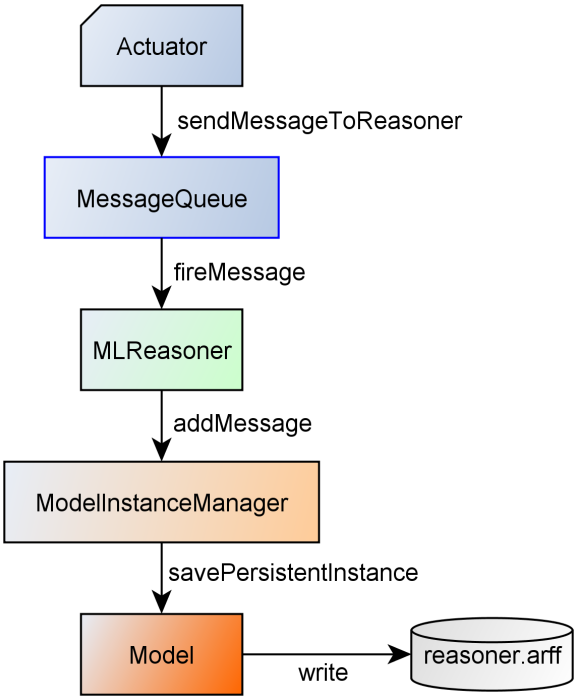

Abbildung 19: Feedbackprozess durch den Aktuator an den Reasoner

asoningergebniss mitgesendet. Genau diese wird nun für das Feedback als Nachricht verwendet. Die Zeichenkette soll als erster Parameter in der Methode `sendMessageToReasoner` übergeben werden und unterliegt keiner Veränderung durch den Aktuator. Die Zusammensetzung der Zeichenkette besteht aus mehreren Informationen, die mittels eines Hashzeichens voneinander getrennt werden. Dazu zeigt Tabelle 3 die einzelnen Felder der zerlegbaren Zeichenkette.

Feldname	**Beschreibung**
Quelle	Die Klasse, die die Nachricht erzeugt hat
Modelinstanz	Die Klasse, die `Model` heißen muss und die ID der Reasoninginstanz
Variablenbeleung	Die Serialisierung der `Instance`

Tabelle 3: Zerlegung der zusätzlich übermittelten Zeichenkette in Unterfeldern

In Bezug auf die oben genannte Beispielnachricht entspricht diese Zerlegung einer bestimmten Belegung dieser drei Felder. Diese werden anhand des Beispiels in Tabelle 4 aufgelistet. Der Reasoner verwendet genau diese Belegung, um das Feedback entsprechend zuzuordnen.

Die Verwendung dieser Feedbackfunktion setzt voraus, dass der Aktuator das Feedback-Datenfeld mit dem Prefix *"__complete_data__"* aus dem ankommenden `SensorData`-Objekt ausliest und auf entsprechende Art und Weise verarbeitet. Ebenso ist es notwendig, dass der Aktuator für dieses Feld das `ValidRange`-Objekt definiert, da dieses der Validierung der Sensorwerte dient. Der Quelltext 7 zeigt die Implementierung für den Empfang dieses Datenfeldes. Die Implementierung

Feldname	**Beispielbelegung**
Quelle	MachineLearningReasoner
Modelinstanz	Model#test_reasoner_museum
Variablenbeleung	agriculture,20,100,0.6,1,augmented

Tabelle 4: Beispielbelegung der zusätzlich übermittelten Zeichenkette

dieser Empfangsmethode wird von der CAKE-Architektur für Plugins vorausgesetzt. Näheres wird in der zugehörigen Dokumentation durch Bouck-Standen et al. (2013) beschrieben.

```
@Override
public ValidRange getInputType(String prefix) {
  if (prefix.equals("__complete_data__")){
    return new ValidRange(String.class);
  }
  // weitere prefixes ...
}
```

Quelltext 7: Implementierung der Validierung des Feedback-Datenfeldes

4.4 Implementierung von Klassifizierern

Eine Grundfunktionalität des Machine Learning Reasoners stellt die flexible Verwendung von unterschiedlichen Reasoningverfahren dar. Konkret sollen verschiedene Machine Learning Algorithmen angewandt oder aber auch implementiert werden können. Zu diesem Zweck stellt der Reasoner ein eigenes Paket bereit: das `builders`-Paket. Dieses beinhaltet die Basisklasse `Builder`, die die Funktion übernimmt, einen Weka-Klassifizierer aus entsprechenden Daten für das Reasoning zu erzeugen.

4.4.1 Das `builders`-Paket

Die abstrakte Basisklasse wird über das XML-Tag `<reasoning>` spezifiziert. Je nach Angabe wird versucht, eine direkte konkrete Unterklasse von `Builder` zu instanziieren oder ansonsten diese Zeichenkette als Pfad zu einer XML-Datei zu interpretieren, die anschließend in der Unterklasse `XMLBuilder` eingelesen und somit als externe Reasoning-Spezifikation interpretiert wird.

Nachdem ein solcher `Builder` für das angegebene Reasoning erzeugt wurde, ruft das `Model` die zugehörige `buildClassifier`-Methode auf, die den Weka Klassifizierer berechnen und zurückgeben soll. Wie in Quellcode 8 angegeben, bewirkt diese Methode genau drei weitere Aufrufe. Diese drei Schritte - Laden des Datensatzes, Preprozessieren des Datensatzes, Bauen des Klassifizieres aus

dem Datensatz -, die auch schon in Kapitel 3 beschrieben wurden, bilden Methoden ab, die von erweiternden Klassen der `Builder`-Klasse überschrieben werden können. Damit ist es möglich, mit einer ableitenden Klasse direkten Einfluss in alle drei Schritte zu nehmen.

```
public Classifier buildClassifier() {
  try {
    if (loadData() && preprocessData()) {
      return build();
    }
  } catch (Exception e) {
    lastException = e;
  }
  return null;
}
```

Quelltext 8: Abstrakte Definition des Weka Klassifizierer-Bau-Prozesses

Während der erste Schritt im Allgemeinen bei jeder Implementierung gleich verläuft und daher auch schon implementiert in der `loadData`-Methode zur Verfügung steht, kann die `preprocessData`-Methode erweitert werden, wenn dieser Vorgang gewünscht ist. Der Datensatz, der in allen drei Schritten verwendet wird, ist in der Variablen `dataset` vom Weka-Typ `Instances` abgelegt und sollte immer dieser Variable zugewiesen werden. Im Gegensatz zu den ersten beiden Schritten ist die Methode `build` abstrakt deklariert. Das bedeutet, dass jede ableitende Klasse selbst dafür sorgen muss, dass ein Klassifizierer gebaut und zurückgegeben wird. Diese Umsetzung folgt daraus, dass jede Implementierung eines `Builder` einen anderen Weka Klassifizierer zurückgeben wird, der sich im Allgemeinen von anderen ML-Verfahren unterscheidet.

Alle drei Schritte beinhalten intern eine Fehlerbehandlung. Die beiden ersten Methoden müssen jeweils zurückgeben, ob der Prozess erfolgreich absolviert wurde (Rückgabe: `true`). Andernfalls bricht der Bauprozess des Klassifizierers an dieser Stelle ab. Tritt ein Fehler im letzten Schritt oder insgesamt auf, so gibt der `Builder` als Objekt `null` zurück. Parallel wird die zuletzt von Java geworfene Ausnahme in der Variablen `lastException` hinterlegt. Wann immer das `Model` bei der Abfrage nach dem Klassifizierer `null` erhält, kann dieses diese zuletzt geworfene Ausnahmemeldung über Aufruf von `getLastException` auslesen und dem Nutzer entsprechend Rückmeldung geben.

Die soeben beschriebenen Objekte und Vorgänge basieren auf zwei Basisklassen, die von Weka benutzt und zur Verfügung gestellt werden: dem `weka.classifiers.Classifier` und dem `weka.filters.Filter`. Jeder Reasoning-Algorithmus wird in einem `Classifier` umgesetzt. Die später verwendeten Klassifizierer leiten von `Classifier` ab und stellen die beiden von CAKE verwendeten Methoden `buildClassifier` und `classifyInstance` zur Verfügung.

Der Vorgang des Preprozessierens soll den Datensatz verändern. Diese Veränderung an Bestand und Belegung der vorliegenden Daten sind über die `Filter`-Klassen von Weka möglich. Jeder Filter

kann andere Parameter erwarten, um die Filterung durchzuführen. Die von CAKE verwendeten Methoden sind hierbei `input` und `output`, mit denen die ursprünglichen Datensätze stückweise eingelesen und ebenso wieder gefiltert ausgelesen werden können. Im Anhang findet sich dazu ein Beispiel im Quelltext 13 bei der Benutzung des `SMOTE`-Filters, der in Abschnitt 4.4.3 eingeführt und erläutert wird.

4.4.2 Vergleich Umsetzung in XML und Java

Die Realisierung eines Reasonerverfahrens in XML beschränkt sich auf die Spezifikation des Klassifizierers. Mittels der externen Spezifikation eines Reasoners in XML kann keine Preprozessierung ermöglicht werden. Dazu benötigt der Entwickler für CAKE eine Implementierung in Java (wie im vorherigen Abschnitt 4.4 beschrieben). Allerdings können die Weka Klassifizierer selbst komplett als externes XML-Dokument angefügt werden und haben so den Vorteil, unabhängig vom Programmcode erweitert und verändert zu werden.

```
<?xml version="1.0" encoding="utf-8"?>

<object name="__root__" class="weka.classifiers.trees.J48"
    primitive="no" array="no">
   <object name="options" class="String" primitive="no" array="yes">
      <object name="0" class="String" primitive="no" array="no">
         -C
      </object>
      <object name="1" class="String" primitive="no" array="no">
         0.25
      </object>
      <object name="2" class="String" primitive="no" array="no">
         -M
      </object>
      <object name="3" class="String" primitive="no" array="no">
         2
      </object>
   </object>
</object>
```

Quelltext 9: J48Tree Klassifizierer in XML-Spezifikation

Die in Quelltext 9 gezeigte Spezifikation eines einfachen J48Tree Klassifizierers stellt eine Instanz der Klasse `weka.classifiers.trees.J48` dar. Die DTD zu dieser Spezifikation war in Quelltext 4 zu sehen. Nach dem Aufruf dieses Klassifizierers wird ein Array an Optionen übergeben, die den Kommandozeilenbefehlen für die Benutzung von Weka entspricht. Für diesen Fall muss der Entwickler mit der Verwendung der Weka Befehle vertraut sein. In dem Beispiel stehen die Parameter für den sogenannten *Confidence Level* und die minimale Anzahl an Instanzen, die dieser Klassifizierer benötigt.

Die Formulierung über XML ist äquivalent zu der Befehlseingabe in der Kommandozeile, die wie

folgt lautet:

```
weka.classifiers.trees.J48 -C 0.25 -M 2
```

Analog zu dieser Spezifikation zeigt Quelltext 10 die Umsetzung der `build`-Methode in Java, die effektiv dieselbe Funktionalität abbildet.

```
@Override
protected Classifier build() throws Exception {
  // weka.classifiers.trees.J48
  J48 cls = new J48();
  // -C 0.25
  cls.setConfidenceFactor(0.25f);
  // -M 2
  cls.setMinNumObj(2);

  cls.buildClassifier(dataset);
  return cls;
}
```

Quelltext 10: J48Tree Klassifizierer in Java-Spezifikation

Da die Umsetzung in XML fehleranfälliger und unübersichtlicher erscheint, ist es empfehlenswert, im Allgemeinen eine Java-Implementierung vorzuziehen. Durch diese ist die Fehlerbehandlung durch das Abfangen von Ausnahmen informativer (durch Ausgabe an den Benutzer) und Konfigurationen können detaillierter debugged werden. Außerdem verfügt die Konkretisierung der `Builder`-Klasse zusätzlich über die Möglichkeit, die Datensätze andersartig einzulesen (z.B. um manuell einige Instanzen auszulassen) und diese anschließend mit speziellen Filtern zu preprozessieren. Die Verwendung von externen Reasoning-Spezifikationen kann eingesetzt werden, wenn eine einfache Funktionalität erwünscht ist oder keine Möglichkeit für das Refactoring mit Kompilierung des CAKE-Systems gegeben ist.

4.4.3 `Builder` des Machine Learning Reasoners

Der aktuelle Stand des Machine Learning Reasoners umfasst eine Vielzahl an implementierten Reasoningverfahren, die gewählt werden können. Unter anderem bieten diese auch unterschiedliche Ansätze für das Preprozessieren von vorhandenen Datensätzen. Es wurden zwei solcher Filter implementiert und für einige Reasoningverfahren umgesetzt.

SMOTE. SMOTE[2] ist ein Filter, der Datensätze betrifft, die aufgrund ihrer Attributbelegung im gesamten Datenbestand nur sehr gering auftreten. Diese erfahren im Preprozessierungsvorgang mit einem bestimmten zu spezifizierenden Faktor das sogenannte *Over-Sampling*, welches die betreffende

[2] Synthetic Minority Oversampling TEchnique

Minderheit durch Erzeugung neuer Instanzen dieser zugehörigen Klasse vergrößert. Das SMOTE-Verfahren kennt zudem mehrere Ansätze, wobei das Over-Sampling nur eine Methode ist. Diese wird von Chawla & Bowyer (2011) beschrieben und gegen andere abgewogen. Im Allgemeinen bietet sich dieses Verfahren also an, wenn Reasoningalgorithmen dazu tendieren, Datensätze mit geringer Auftrittsrate nur marginal zu beachten, obwohl diese trotz ihrer Minderheit sehr aussagekräftig sind.

Resample & RemovePercentage. Eine Kombination aus Resampling und Wiederentfernens stellt der zweite Filter dar, der zwei innere Filter kombiniert. Resampling kann je nach Konfiguration eine beliebige und zufällig ausgewählte Datenmenge duplizieren und damit den Datenbestand künstlich vergrößern oder durch Ersetzung bereits vorkommender Datensätze vereinfachen. Nach der Verzehnfachung des Datenbestands führt nun der zweite Filter eine zufällige Auswahl von lediglich 10% des neuen Datenbestands durch. Die Datenmenge ist numerisch gleich groß geblieben, besitzt nun aber einen zufällig vereinfachten Datenbestand, der sich dadurch auszeichnet, dass einige Datensätze mehrfach vorliegen und andere dafür entfernt wurden.

Daraus resultieren folgende Implementierungen, die so in der Modellierung der jeweiligen Reasonerinstanz als XML-Tag `<default_reasoning>` angegeben und verwendet werden können.

- `NaiveBayes` Der klassische Ansatz für ML-Reasoning nach dem Bayes'schen Theorem für bedingte Wahrscheinlichkeiten
- `SmotedNaiveBayes` - Erweiterung mit SMOTE-Filter
- `ResampledNaiveBayes` - Erweiterung mit Resampling-Filter
- `MultilayerPerceptron` Eine Implementierung von künstlichen neuronalen Netzen zur Darstellung der Datensätze
- `SmotedMultilayerPerceptron` - Erweiterung mit SMOTE-Filter
- `ResampledMultilayerPerceptron` - Erweiterung mit Resampling-Filter
- `KStar` Eine einfache Implementierung eines Graphen nach dem *Nearest-Neighbor-Verfahren*
- `SmotedKStar` - Erweiterung mit SMOTE-Filter
- `ResampledKStar` - Erweiterung mit Resampling-Filter
- `ZeroR` Sehr einfaches Verfahren, welches die Mehrheiten von vorkommenden Werten betrachtet
- `RandomForest` Verwendung mehrerer Bäume zur Kategorisierung von Attributen
- `OneR` Berechnung von Fehlerraten für jede Klassifizierung und Auswahl des geringsten Fehlers

4.5 Formative Evaluation

Die formative Evaluation beinhaltet einen wesentlichen Bereich der iterativen Umsetzung, wie dies in Abschnitt 1.4 für die Vorgehensweise beschrieben wurden. Durch die Evaluierung der schrittweise umgesetzten Komponenten lassen sich tatsächliche technische Funktionalitäten erfassen und ebenso die Interaktion mit der Benutzergruppe - den Entwicklern für CAKE - analysieren. Ziel der formativen Evaluation ist, Mängel und Fehler während der Umsetzung festzustellen und während des Realisierungsprozesses zu beseitigen. Im Folgenden werden die im Rahmen der formativen Evaluation aufgetretenen Entscheidungen im Kontext der Umsetzung erläutert.

Bei der Implementierung des `CakeReasoner`-Interfaces sind einige Spezifikationen bezüglich des Reasoners in CAKE zu treffen. Nach der ersten Umsetzung dieser Spezifikation ergab sich, dass die Angabe eines eigenen Reasonernamens und einer eindeutigen, zu generierenden Kennung (`UUID`) ungeeignet für die Entwicklung einzelner Reasonerinstanzen innerhalb des Machine Learning Reasoners ist. Wie in der Abbildung 20 zu den Ergebnissen der formativen Evaluation zu sehen, ergab sich aus dieser Problematik der Instanziierung das geänderte Konzept des `ModelInstanceManager`, der nun eingebettet ist innerhalb eines einzigen Reasoner-Objekts in CAKE. Mit dieser Architektur entfallen unnötige Angaben wie Bezeichnung oder Kennung einer individuellen Reasonerinstanz.

Mit der Implementierung des `MachineLearningReasoner` ergab sich die Notwendigkeit, Sensoren direkt beim Initialisierungsprozess zu abonnieren und zur Laufzeit jederzeit abzufragen, um neu ankommende Sensoren am CAKE-System zu registrieren. Aus dieser Umsetzung ergab sich die

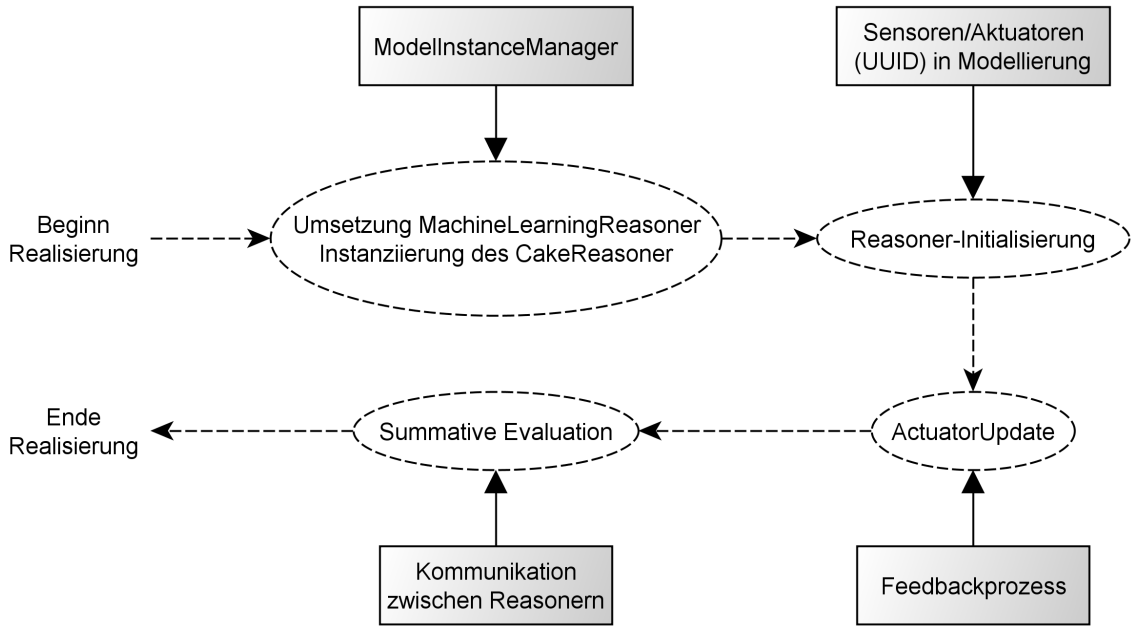

Abbildung 20: Ergebnisse (grau) der formativen Evaluation nach jedem Entwicklungsschritt (weiß)

zusätzliche Anforderung, die zu abonnierenden Sensoren und auch die anzusprechenden Aktuatoren direkt in die Modellierung einer einzelnen Reasonerinstanz zu übernehmen. Diese werden daher im Initialisierungsprozess des Reasoner eingelesen.

Eine wesentliche Funktion bildet der Feedbackprozess aus Abschnitt 4.3. Mit der Entwicklung der Ausgabe an die Aktuatoren fiel im Rahmen der formativen Evaluation auf, dass nach der Kontextanalyse die Funktionalität für direktes Feedback fehlt. Im primären Szenario wurde dazu ein Museumsbesucher definiert, der möglicherweise während seines Besuchs Änderung am Model des Machine Learning Reasoners auslösen könnte. Daher entstand das Konzept des unmittelbaren Feedbacks, welches direkt in die Datenbank des Reasoners gespeichert und beim nächsten Klassifzierer-Bau zur Verfügung steht.

Zuletzt wurde durch eine abschließende Evaluation festgestellt, dass die verschiedenen Reasoner fähig sein sollten, untereinander zu kommunizieren. Dies wurde ebenfalls in der Kontextanalyse skizziert. Daraus ergibt sich die Funktionalität, dass Reasonerinstanzen beim Senden von Ergebnissen diese nicht nur an Aktuatoren, sondern auch an andere beliebige `CakeReasoner` senden können. Diese Reasoner erhalten die Ergebnisse wie Sensorwerte und können sie auf dem selben Wege wie Sensoren abonnieren.

5 Summative Evaluation

Diese in den vorangegangenen Kapiteln beschriebene und entwickelte Arbeit soll abschließend bewertet und ihr effektiver Nutzen eingestuft werden. Damit einhergehend ist zu prüfen, ob das abstrakte Ziel, die Erweiterung der Reasoning-Fähigkeiten des Context-Awareness-Frameworks CAKE, erreicht und ein Mehrwert geschaffen wurde. In diesem Zusammenhang wird die Evaluation verschiedene Aspekte betrachten, auf die in den folgenden Abschnitten näher eingegangen wird.

5.1 Ziel der Evaluation

Die Evaluation beantwortet die Frage nach der Nutzbarkeit des neu entwickelten Reasoners. Dazu muss der Reasoner zunächst funktionsfähig und in seiner aktuellen Konfiguration verwendbar sein, sodass ein Reasoningprozess in CAKE über diesen Reasoner möglich ist. Zudem wird die Frage beantwortet, wie zielführend und effektiv der Reasoner arbeitet und welche Eigenschaften er im Hinblick auf Performance bietet. Auf diese Fragestellungen bezieht sich die technische Evaluation, die im folgenden Abschnitt 5.2 ausführlich beschrieben und durchgeführt wird. Es werden hierbei Zeiten und korrekte Aussageraten der Reasoningergebnisse ermittelt.

Neben der reinen Funktionsweise soll das Ziel der Arbeit erreicht werden. Dieses wurde im Kapitel 2 in Funktionen und Anforderungen an den Reasoner formuliert. Der Teil der funktionalen Evaluation im kommenden Abschnitt 5.3 ermittelt die Erfüllung dieser angestrebten Ziele mittels eines deskriptiven Walkthroughs durch die Funktionalitäten und diskutiert den Grad der Erfüllung. Dieser Abschnitt gibt Aufschluss darüber, ob die gesetzten Ziele hinsichtlich der Bedienbarkeit und Verwendbarkeit des Machine Learning Reasoners erfüllt und umgesetzt wurden.

Abschließend führt der Abschnitt 5.4 Ergebnisse dieser beiden Evaluationsprozesse zusammen und ermittelt daher abschließend die Frage, ob der Reasoner im Gesamten die Anforderungen an die gestellte Arbeit erfüllt.

5.2 Technische Evaluation

Die technische Evaluation betrachtet verschiedene Messungen im Betrieb des Machine Learning Reasoners. Hierzu werden Testmodellierungen erzeugt und auf das umgesetzte CAKE-System angewandt. Das Ergebnis soll Vergleichswerte auflisten, die verschiedene Verwendungsmöglichkeiten des neuen Moduls in CAKE gegeneinander abwägt und zudem die Effektivität im Reasoning darlegt.

Im Folgenden wurden drei Szenarien künstlich entwickelt. Der Beschaffung und Umsetzung von Modellierung sowie Datensätzen liegt keine nähere Analyse oder Erfassung durch Benutzer zugrunde, sondern stützt sich auf die Übersichtlichkeit der verwendeten Attribute, um die Testumgebung der technischen Evaluation mit angemessen geringem Entwicklungsaufwand zu betreiben. Die kommenden Abschnitte umfassen die Beschreibung der drei entwickelten und modellierten Szenarien sowie die Testprozeduren in Verwendung mit dem Machine Learning Reasoner.

5.2.1 Test Reasoner Museum

Eine vereinfachte Form der Museumsmodellierung, die Besucher und Exponate durch einige Attribute modelliert. Diese Modellierung und Testumgebung soll den Anwendungskontext des primären Szenarios umreißen und die technische Verwendung mit einem realen Bezug prüfen. Hierbei stellt *type* die Kategorie des Ausstellungsstücks dar, *size* die Größe in Quadratmetern und *year* das Jahr, zu dem das Exponat gehört. Zudem sagt *idle* im Bereich zwischen 0 und 1 prozentual aus, wie viel sich der Mensch bewegt (interessiert den Kopf/Körper drehend) und *speed* gibt dazu die Bewegungsgeschwindigkeit (Meter pro Sekunde) an. Lediglich die letzten beiden Werte müssen von Sensoren erfasst werden und dienen der Kategorisierung des Besuchers, um die *textform* für ihn ermitteln zu können. Je nach Interesse erhält dieser unterschiedlich große Mengen an Daten für das Exponat angezeigt. Tabelle 5 zeigt hierfür eine Beispielbelegung an Datensätzen, die für den Test verwendet werden können.

5.2.2 Test Reasoner Activity

Mit Hilfe einiger Sensorwerte soll der Aktivitätszustand einer Person ermittelt werden. Die Aktivität soll beschreiben, welcher Tätigkeit am Tag die Person nachgeht, d.h. ob sie zu Hause ist, auf der Arbeit, unterwegs beschäftigt oder einer Freizeitaktivität nachgeht. Dazu ermittelt ein Smartphone-Sensor, ob die Person sitzt (*sitting*), wie viel der vergangenen Stunde die Person das Smartphone verwendet hat (*smartphone_percentage* von 0 bis 1 prozentual) und ebenso, wie viel die Person in den vergangenen Minuten aktiv geredet hat (*talking_percentage*). Dazu kommt das Feld *date* mit der aktuellen Tageszeit und die Bewegungsgeschwindigkeit der Person (*speed* in Metern pro Sekunde). Ermittelt wird nun also der Aufenthaltsort bzw. die Aktivität bezüglich dieser Modellierung, der die Datensätze aus Tabelle 6 zugrunde liegen.

type	size	year	idle	speed	textform
agriculture	20	100	0.8	1.5	huge
agriculture	20	100	0.95	1	huge
agriculture	20	100	0	2	small
agriculture	20	100	0.5	1.5	augmented
agriculture	12	20	0.95	0.5	huge
agriculture	12	20	0.6	1	augmented
agriculture	12	20	0.4	1.5	small
politics	14	280	0.1	2	small
politics	14	280	0.3	1	augmented
politics	14	280	0.2	0.5	augmented
politics	1	10	0.95	0.4	huge
politics	1	10	0.9	0.7	huge
politics	1	10	0	1.5	small
architecture	40	150	0.2	0.2	augmented
architecture	40	150	0.8	0.4	huge
architecture	40	150	0.1	2	small
architecture	40	150	0.2	0.5	small
architecture	20	40	0.6	1	augmented
architecture	20	40	0.1	1.5	small
architecture	20	40	0.8	0.35	huge

Tabelle 5: Datensätze für die Museum-Testumgebung

sitting	smartphone_percentage	talking_percentage	date	speed	result
yes	0.3	0	20:00:00	0	AtHome
no	0	0.8	18:30:00	0.5	AtHome
no	0	0	06:30:00	0.5	AtHome
yes	0	0.5	11:00:00	0	AtWork
no	0.7	0.7	16:00:00	1	AtWork
no	0.3	0.1	17:45:00	1.5	Busy
yes	0	0	18:15:00	55	Busy
yes	0.5	0.9	21:30:00	0	SpareTime
no	0	0.9	23:30:00	0.8	SpareTime
yes	0.8	0.4	22:15:00	0	SpareTime
no	0	0.9	23:30:00	1	SpareTime

Tabelle 6: Datensätze für die Activity-Testumgebung

5.2.3 Test Reasoner Heating

Als dritte Testumgebung kommt eine sehr einfache Fragestellung zum Einsatz. Es soll für ein ambientes Heim ermittelt werden, ob die Heizung ein- oder ausgeschaltet werden muss. Diese Fragestellung soll allerdings nicht anhand der Temperatur alleine entschieden werden, sondern wägt zusätzlich ab, ob die Heizung gebraucht wird aufgrund der Benutzeraktivität und der Außentemperatur. Diese Werte

werden repräsentiert durch *activity*, *innerTemperature* und *outerTemperature*, die jeweils in Grad Celcius die Temperatur angeben. Zur Kontextabhängikeit kommt dazu die Anzahl der in der Wohnung befindlichen Personen mittels *presentPeople* und deren allgemeiner Bewegungsgrad, d.h. die Frage, ob Personen eher sitzen oder sich viel im Raum bewegen, in Prozent von 0 bis 1. Daraus ergibt sich im Fazit eine Aussage über die Heizungsfrage. Tabelle 7 hält dazu die Testdatensätze vor.

activity	**innerTemp**	**outerTemp**	**presentPeople**	**movement_percentage**	**result**
SpareTime	18.5	16	0	0	no
SpareTime	22	24	0	0	no
SpareTime	14	10	0	0	yes
AtWork	16.5	16	0	0	no
AtWork	14	12	0	0	yes
Busy	15.5	13.5	0	0	yes
Busy	20	18.5	0	0	no
Busy	17	4.5	0	0	yes
AtHome	16.5	15	1	0.1	yes
AtHome	18.5	19	1	0.3	no
AtHome	17.5	16	3	0.8	no
AtHome	22	23.5	1	0.1	no
AtHome	15.5	7.5	2	0.5	yes
AtHome	15.5	-10	1	0.8	yes
AtHome	10	16	1	1	yes

Tabelle 7: Datensätze für die Heating-Testumgebung

5.2.4 Technische Tests und Ergebnisse

Die gegebenen Szenarien besitzen jeweils annähernd ähnlich viele Datensätze, wobei die Museumsumgebung mit zwanzig Instanzen die meisten aufweist. Da einzelne Reasonerinstanzen besonders im Lernprozess einen enormen Zuwachs an Datensätzen erfahren sollen, werden die folgenden Testprozeduren ebenso auf eine künstlich vergrößerte Datenmenge im Museumskontext angewandt, um numerisch höher skalierte Datenbestände auf ihr Verhalten zu evaluieren. Diese drei Szenarien mit einem zusätzlichen vergrößertem Museumsszenario (im folgenden als MUSEUM bezeichnet) werden mit Hilfe von drei Reasoningalgorithmen und zwei unterschiedlichen Preprozessierungsansätzen getestet. Diese Verfahren sind in Abschnitt 4.4 aufgeführt. Darunter finden sich die drei klassischen Techniken `NaiveBayes`, `MultilayerPerceptron` und `KStar`, die jeweils auch eine Implementierung mit vorgeschaltetem SMOTE und Resampling-Filter besitzen. Diese insgesamt neun verschiedenen Techniken werden im Folgenden untersucht im Hinblick auf folgende Kriterien:

- Bauzeit des Klassifizierers
- Korrektheit der Aussagen
- Dauer der Klassifizierung / des Reasonings

In den Tabellen zur Ergebnispräsentation der Testprozeduren werden die einzelnen Reasoningkonfigurationen aus Platzgründen abgekürzt und durchnummeriert dargestellt. Dabei entsprechen die Nummern den jeweiligen Konfigurationen nach folgender Auflistung:

(1)	`NaiveBayes`
(2)	`SmotedNaiveBayes`
(3)	`ResampledNaiveBayes`
(4)	`MultilayerPerceptron`
(5)	`SmotedMultilayerPerceptron`
(6)	`ResampledMultilayerPerceptron`
(7)	`KStar`
(8)	`SmotedKStar`
(9)	`ResampledKStar`

Dabei ist anzumerken, dass der `NaiveBayes`-Ansatz im Test der Activity-Umgebung ausgelassen wird, da dieser Bayes'sche Algorithmus den Datums-Datentyp nicht verarbeiten kann und daher keine Klassifizierung durchgeführt werden kann. In den folgenden Tabellen erscheinen diese Felder daher leer.

Die nun im folgenden durchgeführten Testprozeduren wurden alle auf der gleichen Testmaschine[1] bei gleicher Hardware- und Softwarekonfiguration durchgeführt. Ziel dieser Testverfahren sind u.a. Ermittlungen von benötigten Zeiten. Diese werden überwiegend zu Vergleichszwecken zwischen den einzelnen Reasoningverfahren und -domänen erhoben und stellen ebenso ein realistisches Beispiel für eine CAKE-Umgebung dar. Die hier gezeigten Zeiten können sich durch andere Computersysteme unterscheiden.

Bauzeit des Klassifizierers

Die Bauzeit des Klassifiziers beinhaltet das Einlesen der vorhandenen Datensätze, das Preprozessieren der Daten nach angegebenem Filter und das Erzeugen des Weka-Models. Für jedes Szenario wird nun jede der zuvor vorgestellten Reasoningkonfigurationen angewandt und miteinander verglichen.

Alle Ergebnisse werden in Tabelle 8 in Millisekunden angegeben und beschreiben einen Mittelwert aus den gemessenen Bauzeiten. Abweichungen vom Mittelwert liegen in einem kleinen Rahmen von wenigen Millisekunden (im Bereich von `NaiveBayes` und `KStar`) bis hin zu hundert Millisekunden (im Bereich von `MultilayerPerceptron`). Die größeren Abweichungen liegen hierbei auch bei einer größeren Ausgangsdatenmenge vor, wie im künstlich erweiterten Datensatz der Museum-

[1] Testsystem-Konfiguration:
Intel Core2Quad 2.83GHz
Arbeitsspeicher DDR2 4,00 GB
Windows 8 Pro 64-Bit

sumgebung (MUSEUM), die mit 200 Instanzen um den Faktor 20 größer ist als ihre Ausgangsmodellierung.

	(1)	(2)	(3)	(4)	(5)	(6))
Museum	2,231	4,596	14,771	29,873	131,209	517,521
Activity	-	-	-	16,264	66,886	567,559
Heating	1,640	5,447	14,205	22,411	123,741	534,414
MUSEUM	23,037	106,484	59,974	1063,045	2337,002	1457,776

	(7)	(8)	(9)
Museum	1,346	2,993	16,997
Activity	2,045	7,610	26,130
Heating	2,101	3,386	16,264
MUSEUM	15,995	80,510	51,811

Tabelle 8: Bauzeiten der Klassifizierer in den verschiedenen Testumgebungen. Die Zeiten werden in Millisekunden angegeben

Auffällig ist es, dass das Preprozessieren der Datensätze bei geringen Datenbeständen bei Verwendung des SMOTE-Filters schneller erfolgt als der zufällige Resampling-Prozess. Dieser Sachverhalt scheint sich aber umzukehren, wenn die Anzahl an Datensätzen zunimmt, da offenbar eine Filterung an Minderheiten eine höhere Rechenzeit mit sich bringt. Insgesamt sticht hervor, dass die Konstruktion eines künstlichen neuronalen Netzes die meiste Zeit in Anspruch nimmt.

Korrektheit der Aussagen

In diesem Abschnitt zeigen die Tests die Aussagekraft der erzeugten Reasoner. Dabei werden zwei Reihen von Klassifizierungstests durchgeführt: Zunächst werden alle Datensätze, die zum Erzeugen des Reasoners verwendet wurden, vom fertig erstellten Model geprüft. Bei diesem Test kann ermittelt werden, wie gut die Logik des Reasoners die Eingangsdatensätze verarbeiten und wiedererkennen kann, ohne möglicherweise einige abweichende Datensätze fehl zu interpretieren. Die zweite Testreihe wendet zuvor noch nicht verwendete und neue Datensätze auf den Reasoner an. Dieser Test entspricht viel mehr der realen Verwendung des Reasoners und soll zeigen, inwieweit die Reasonerlogik neue Probleme korrekt lösen kann.

Aus Tabelle 9 lässt sich schließen, dass die Wiedererkennung von den zu lernenden Datensätzen im Bayes'schen Verfahren sowie in künstlich neuronalen Netzen mit hoher Wahrscheinlichkeit korrekt ist. Der `KStar`-Algorithmus scheint im Verbund mit der Activity-Testumgebung einige Fehler aufzuweisen, was daran liegen könnte, dass die Ähnlichkeit der Datensätze zu falschen Heuristiken in der *Nearest-Neighbor-Suche* führt.

Für die Problemlösung von noch nicht bekannten Instanzen erhält jedes Szenario drei Beispielinstanzen, die die Reasoner zu lösen haben. Tabelle 10 beinhaltet dabei die genaue Belegung der Testdaten,

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
Museum	0,9000	0,9750	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
Activity	-	-	-	1,0000	0,9677	1,0000	0,2727	0,9677	0,5455
Heating	0,9333	0,9882	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000

Tabelle 9: Anteil der korrekt wiedererkannten Instanzen aus dem Eingangsdatensatz. Der Anteil reicht von 0 (0%) bis 1 (100%) als prozentuale Angabe

die für eine spätere Verwendung mit kleinen Buchstaben abgekürzt werden.

Diese neun Instanzen werden im Folgenden von den entsprechenden Reasoningkonfigurationen untersucht und klassifiziert. Der erzielte Vergleich lässt einen möglichen Schluss darüber zu, wie gut welche Konfigurationen gegebene Probleme lösen kann und unter welchen Arten der Anwendungskontext-Modellierung Probleme auftreten.

		type	**size**	**year**	**idle**	**speed**	**textform**
Museum	(a)	politics	40	150	0.75	0.2	huge
	(b)	agriculture	65	50	0.75	0.2	huge
	(c)	agriculture	65	50	0.2	1	augmented

		sitting	**smartphone_%**	**talking_%**	**date**	**speed**	**result**
Activity	(d)	yes	0	0	19:00:00	30	Busy
	(e)	yes	0	1	22:30:00	0	SpareTime
	(f)	no	0.5	0.9	02:00:00	1	SpareTime

		activity	**innerTemp**	**outerTemp**	**presentPpl**	**move_%**	**result**
Heating	(g)	SpareTime	15.5	18	0	0	no
	(h)	Busy	19	15	0	0	no
	(i)	AtHome	17	8.5	3	0.5	yes

Tabelle 10: Test-Instanzen für die Klassifizierungstests

Die Klassifizierungstests aus Tabelle 11 zeigen verschiedene Aspekte. Zum einen fällt auf, dass die Reasoningergebnisse der verschiedenen Verfahren und Preprozessierungsvarianten häufig sehr ähnlich ausfallen im Bezug auf eine bestimmte Eingabe. Der sehr einfache Fall der Heating-Umgebung zeigt, dass mathematische unkomplizierte Zusammenhänge fast vollständig ohne Fehler gelöst werden können. Im Fall der schwierigen Activity-Umgebung kommen größere Klassifierungsabweichungen und -fehler zum Vorschein. Semantisch liegen die Ergebnisse meistens eine Stufe von der Lösung entfernt. Erklärung: *AtHome* liegt *SpareTime* nahe, da die Aktivitäten möglicherweise viele Gemeinsamkeiten haben, während *AtWork* logisch weiter entfernt liegt von *SpareTime* (keine gemeinsamen Eigenschaften des Alltags). Dazu analog könnte *Busy* bedeuten, dass direkt im Anschluss einer kurzen Beschäftigung jede andere Aktivität folgt, d.h. jede Klassifizierung, die von *Busy* abweicht, wäre nur eine logische Stufe von der korrekten Lösung entfernt.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
(a)	aug.	huge	huge	huge	huge	huge	huge	huge	huge
(b)	huge	huge	huge	huge	aug.	huge	huge	huge	huge
(c)	small	small	small	small	aug.	aug.	small	aug.	aug.
(d)	-	-	-	Busy	Busy	Busy	AtHome	AtHome	AtHome
(e)	-	-	-	SpareT.	SpareT.	SpareT.	AtHome	AtHome	AtHome
(f)	-	-	-	AtWork	AtWork	AtHome	AtHome	AtHome	AtHome
(g)	no	no	no	no	no	no	no	no	no
(h)	no	no	no	no	no	no	no	no	no
(i)	yes	yes	yes	yes	yes	yes	yes	no	yes

Farblegende: Ziel korrekt klassifiziert - Ziel leicht verfehlt - Ziel maximal verfehlt

Tabelle 11: Reasoningergebnisse zur Problemlösung von Testdaten, die Feldfarben symbolisieren die logische Genauigkeit

Dauer der Klassifizierung

Für den Betrieb von CAKE und den entwickelten und einsatzbereiten Reasonern gilt zu ermitteln, wie lange eine Klassifizierung dauert und von welchen Faktoren diese abhängt. Dazu wurden beliebige Sensordaten simuliert übertragen und von dem Reasoner klassifiziert. Für diese Testreihe wurden die drei Reasoningverfahren ohne Preprozessierung betrachtet, da die Klassifizierungszeit nur vom fertig entwickelten Weka-Model abhängt.

Die Klassifizierung von Eingaben im Bereich der Museumsumgebung zeigt Tabelle 12 in Bezug auf die benötigte Zeit in Millisekunden. Während der `NaiveBayes`-Ansatz scheinbar keine Beeinflussung durch die Mächtigkeit der vorangegangen Datenmenge zeigt, fallen deutliche Steigerungen bei den anderen beiden Verfahren auf. Es liegt nahe, dass sich bei diesen Machine Learning Verfahren die Anzahl an zu lernenden Instanzen im Klassifizierungsprozess widerspiegeln. Das künstliche neuronale Netz scheint hierbei bei der numerisch mächtigeren Museums-Umgebung die doppelt vorkommenden Instanzen effizienter zu verarbeiten und abzurufen als der `KStar`-Algorithmus, der im Vergleich den doppelten Zuwachs an Rechenzeit erfährt.

	`NaiveBayes`	`MultilayerPerceptron`	`KStar`
Museum	0,135	0,123	0,622
MUSEUM	0,108	0,660	5,457

Tabelle 12: Klassifizierungszeiten von Testdaten der Museumsumgebung. Die Werte sind in Millisekunden angegeben

5.3 Funktionale Evaluation

Im Hinblick auf die Analyse aus Kapitel 2 stellt die funktionale Evaluation eine deskriptive Bewertung über die erreichten Ziele dar. In diesem Abschnitt werden Zielsetzungen und genaue Funktionen

aus der Anforderungsanalyse mit dem entwickelten System abgeglichen und eine Aussage darüber gemacht, wie sehr die Umsetzung die Anforderungen erfüllt hat. Dabei werden die Ziele zusammen mit der Benutzergruppe betrachtet, die für die Ausübung der Interaktionen bestimmt ist.

5.3.1 Hierarchical Task Analysis - Walkthrough

In diesem Teil der Evaluation werden die aus der Analyse erhobenen Schritte einer Reasonerentwicklung und -interaktion mit der umgesetzten Version abgeglichen. Dazu stehen im Folgenden Teile der HTA-Diagramme (Abbildung 5 und Abbildung 6 aus Kapitel 2) in Betrachtung.

Abbildung 21 zeigt nochmals den Ablauf einer Reasonerentwicklung laut der Anforderungsanalyse (siehe Abschnitt 2.1). Wie die Grafik verdeutlicht, enthält die umgesetzte Form der Modellierung einer Reasonerinstanz alle Unterpunkte der gezeigten HTA. Aus der Analyse ergab sich, dass die Benutzerklasse der Entwickler zuständig ist für die Entwicklung einer neuen Reasonerinstanz (siehe Kapitel 2 Abschnitt 2.6). Dieser Entwickler für CAKE schreibt nun für die Entwicklung einer neuen Instanz die Modellierung. Diese im Folgenden als *reasoner.xml* abgekürzte XML-Datei liegt im Pfad

```
res/reasoner/mlReasoner/[uuid_des_Reasoners]/"NAME".xml
```

und dient der Erzeugung einer neuen Reasonerinstanz, die der Machine Learning Reasoner bereits einliest und versucht zu einem Reasonermodel zu verarbeiten.

Die Modellierung selbst ist in dieser Modellierungsdatei enthalten, indem alle Sensor- und Reasoningwerte mittels Datentyp und -bezeichung deklariert werden. Der Entwickler, der in einem Analyseprozess vor der Entwicklung des Reasoners selbst eine Erhebung aller Sensoren durchgeführt hat,

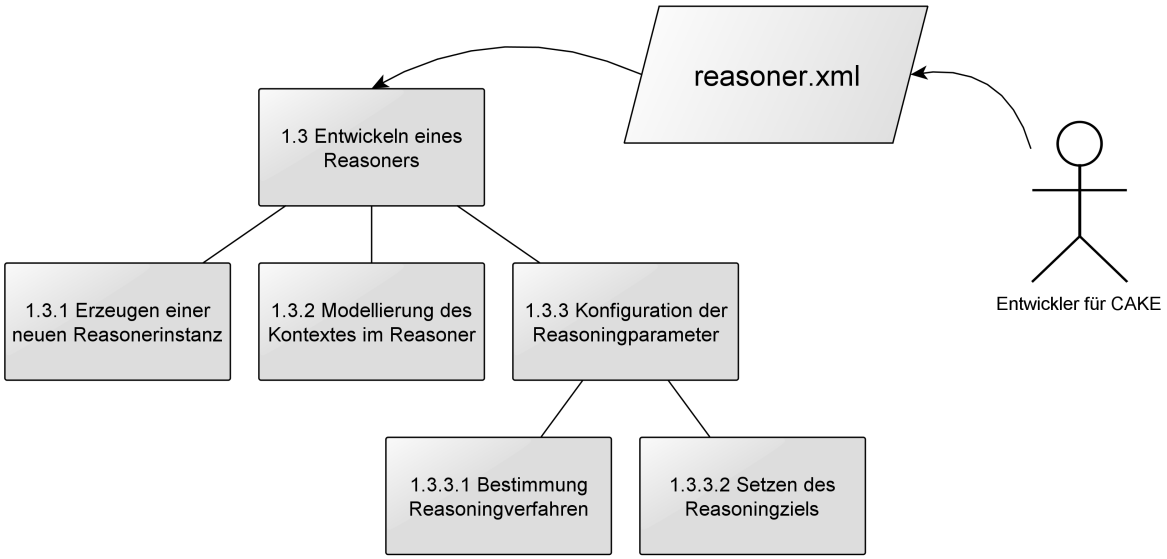

Abbildung 21: Abdeckung der *reasoner.xml* für alle Schritte der Reasonerentwicklung

kennt alle eingehenden Sensorwerte und deklariert diese in der XML-Datei mit Hilfe der vorgegebenen Spezifikation (vergleiche dazu Abschnitt 3.3). Ebenso fügt der Entwickler alle Pluginkennungen (`UUID`) der Sensoren und Aktuatoren ein, mit denen der Reasoner kommunizieren soll.

Zusätzlich zu der Auflistung aller Aktuatoren lässt sich mit dem innerhalb eines Objekts vorkommenden und notwendigen XML-Elements `<class>true/false</class>` bestimmen, ob der angegeben Wert der Modellierung das Reasoningergebnis bzw. -ziel darstellt. Das Reasoningverfahren wird letztlich am Ende der *reasoner.xml* angegeben und ist, wie in Kapitel 4 beschrieben, aus einer Menge gegebener Verfahren auswählbar oder auch selbst mittels zusätzlicher Definition (mittels XML, siehe Abschnitt 3.3.1) erweiterbar.

Während die Modellierung mittels XML bereits den gesamten Initialisierungsprozess entsprechend der Analysephase umsetzt, bietet die Interaktion mit dem Reasoner zur Laufzeit (Abbildung 6) mehrere Funktionalitäten, die unterschiedlich arbeiten. Abbildung 22 zeigt einen Ausschnitt der HTA, die die jeweiligen Aufgaben den Schnittstellen des Machine Learning Reasoners zuordnet. Der Entwickler einer neuen Reasonerinstanz benötigt grundlegende Möglichkeiten, mit dem Reasoner zu kommunizieren, um dessen Ergebnisse des Reasonings zu erfragen oder Sensorwerte an diesen zu schicken. Zum Abfragen der Reasoningergebnisse stellt der Entwickler sicher, in der zuvor angesprochenen XML-Modellierung den nachfragenden Sensor mit seiner eindeutigen Pluginkennung (`UUID`) einzutragen. Anschließend wird der Empfang des Reasoningergebnisses über die CAKE-API eines Sensors sichergestellt, indem die zugehörige Methode `ActuatorPlugin::fireStateChanged` implementiert und somit eingehende Daten empfangen werden können. Dabei kann der Entwickler auf einfache Weise den sendenden Reasoner ermitteln und zuordnen, ob die Daten vom gewünschten Typ sind.

Gleichermaßen ergibt sich für den Entwickler das Senden neuer Sensorwerte an den Reasoner, in-

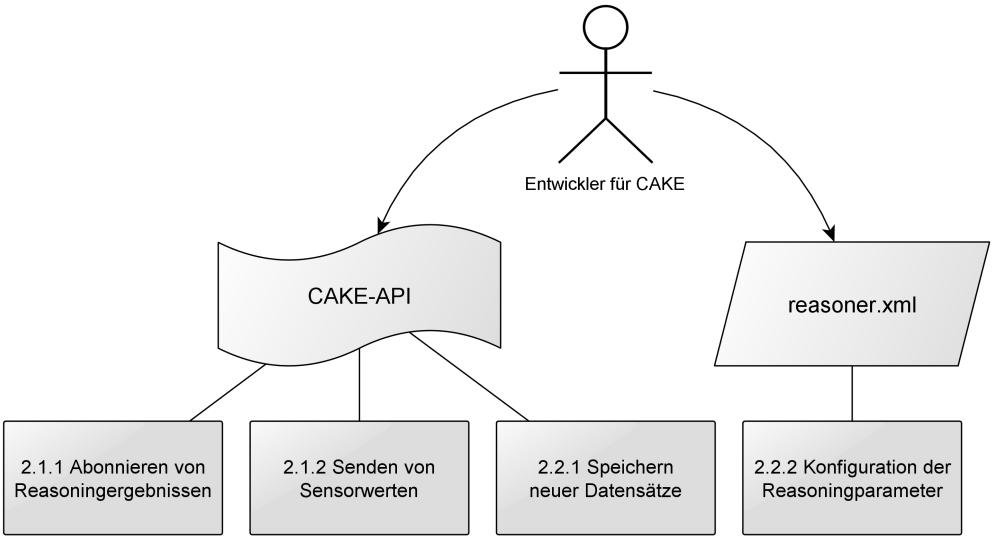

Abbildung 22: Aufgabenabdeckung für die Interaktion mit dem Reasoner zur Laufzeit

dem wiederum eine Methode aus der CAKE-API für Aktuatoren verwendet wird. Die bereitgestellte Kommunikationsmethode `ISensorEmitter::onValue` sendet somit Werte an den Reasoner, die allerdings zeitgleich validiert werden müssen. Dazu geht der Entwickler gemäß der Dokumentation zur Entwicklung eines Sensors vor (vergleiche hierzu Bouck-Standen et al. (2013)).

Auch der Arbeitsvorgang *2.3 Validierung des Reasoningmodels* aus der HTA gemäß Abbildung 6 lässt sich in der Reasonerumsetzung wiederfinden. So sieht der Reasoner vor, neue Datensätze zum Zwecke des Lernprozesses zu empfangen und an das eigene Model zu schicken. Dies geschieht zur Laufzeit und ist als *Feedbackprozess* im Abschnitt 4.3 beschrieben. Für dieses Vorgehen kann der Entwickler für CAKE eine weitere CAKE-API Methode benutzen. Wie bereits in der entsprechenden Dokumentation erwähnt, sorgt die Methode `ICakeCommuniation::sendMessageToReasoner` für den Datentransport eines solchen Feedbacks mit einer zu lernenden Instanz, die im Anschluss in der eigenen Reasonerdatenbank abgelegt wird.

Die in der HTA weiter betrachteten Punkte sehen vor, dass der Entwickler zur Laufzeit des Reasoner die Konfiguration des Reasoningverfahrens verändern kann. Zudem soll in dem gleichen Moment das zugehörige Model dieser Reasonerinstanz aktualisiert und erneuert werden, ohne dass Datensätze bzw. der vorangegangene Lerneffekt verloren geht. Diese Funktionalität findet sich im Machine Learning Reasoner wieder, indem die XML-Modellierungsdatei angepasst wird. Die Konfiguration beinhaltet hierbei nur Reasoningverfahren sowie verwendete Sensoren und Aktuatoren. Eine Übernahme dieser Anpassung lässt sich im Anschluss über einen Neustart von CAKE, bzw. des Machine Learning Reasoners in CAKE, verwirklichen. Hierbei werden auch ausstehende Lernfortschritte übernommen. Der Entwickler kann mit diesem Verfahren die Reasonerinstanz anpassen und die Datenbestände verlustfrei beibehalten. Einschränkend ist hierbei, dass die Modellierung grundsätzlich in ihrer Variablendeklaration nicht geändert werden kann (siehe dazu auch Kapitel 4). Außerdem stellt das Neustarten von CAKE eine im Hinblick auf die Bedingung *"Zur Laufzeit"* noch eine eingeschränkte Funktionalität dar. Die aktuellen Schnittstellen lassen es nur zu, diesen Arbeitsschritt mit einem Schließen des CAKE-Servers und einer erneuten Initialisierung umzusetzen (dazu mehr im kommenden Kapitel 6).

5.3.2 Funktionsabdeckung

Analog zum vorherigen Abschnitt werden nun Funktionen, die aus der Analyse als Zielsetzung hervorgegangen sind, auf ihre Umsetzung geprüft. Der folgende Teil behandelt in Kürze die in Abschnitt 2.6 zusammengestellte Feature-Liste und diskutiert, ob die in ihr aufgestellten Ziele erfüllt werden konnten.

Tabelle 13 listet die Anforderungen an das System auf und nennt dabei bereits einen Grad der Erfüllung. Diese ersten vier Features wurden im HTA-Walkthrough detailliert beschrieben. Die letzten beiden Funktionen sind nur mit Einschränkung erfüllt. Wie bereits beschrieben ist die Konfiguration des Reasoners zwar zur Laufzeit möglich, aber Änderungen werden erst nach einem kompletten

CAKE-Neustart übernommen. Dieses Verhalten erscheint möglicherweise nicht unmittelbar verständlich bzw. erlaubt dem Entwickler keine Interaktion, eine veränderte XML-Modellierung nachträglich in das Reasoner-System zu laden und zu verwenden. Ein Neustart ist in jedem Fall notwendig. Der daraus resultierende Aufwand, wenn eine Konfiguration im Detail überprüft und mehrere Male in einer kurzen Zeitperiode geladen werden muss, ließe sich mit einer besseren Umsetzung in diesem Teilbereich vermindern.

Die Erweiterbarkeit der Schnittstellen in der Architektur des Reasoners ist überwiegend vorhanden. Die einzige Einschränkung besteht darin, dass die Anbindung der Datenschicht nicht direkt austauschbar vorliegt. So werden derzeit *.arff*-Dateien als Speicherformat für die Datensätze verwendet. Die daraus entstehenden Klassifizierer erwarten jeweils immer das gleiche Speicherformat, weshalb es nicht möglich wäre, neben *.arff*-Dateien noch andere Formate gleichzeitig für ein ML-Verfahren zu unterstützen. Hier würde der Umstieg auf ein anderes Speicherformat (wie etwa den *.xrff*-Dateien oder aber einer lokalen Datenbank) mit größerem Aufwand in der Anpassung der Implementierung der Modellierung erfolgen.

Die Umsetzung der Funktionen sollte laut der Analyse von der Benutzergruppe der Entwickler erfolgen. Diese besitzen laut Abschnitt 2.2 Erfahrungen und Fähigkeiten im Umgang mit der Programmierung und können insbesondere Programmier- und Beschreibungssprachen selbstständig erlernen. Für die Verwendung des Machine Learning Reasoners in seinen vorgesehenen und konzipierten Aufgaben beschränkt sich die Interaktion auf zwei Techniken: Verwendung von XML und Benutzen von Methoden der CAKE-API. Letztere liegt in der Programmiersprache Java vor, die der Entwickler aber

1. Mehrfach-Instanziierung des ML-Reasoners	✓
1.1 Mehrfach startbar in der CAKE-Umgebung	✓
1.2 Parallel mehre Modellierungen starten	✓
2. Flexible Modellierung von Variablen und Ausgaben	✓
3. Konfiguration Reasoning-Verfahren	✓
3.1 Auswahl an vielen Reasoningalgorithmen	✓
3.2 Einfache Inbetriebnahme einzelner Algorithmen	✓
4. Erweiterbarkeit des Models durch gegebene Datensätze	✓
4.1 Datenaufnahme- und Speicherung zur Laufzeit	✓
4.2 Aktualisierbarkeit des Models	✓
5. Konfiguration und Adaptierbarkeit zur Laufzeit	◐
5.1 Anpassung der Konfiguration	✓
5.2 Konfigurierbarkeit zur Laufzeit	◐
6. Erweiterbarkeit Schnittstellen	◐
6.1 Architektur abstrakt gestalten	◐
6.2 Änderung an Funktionalitäten im Java-Code sinnvoll umsetzbar	◐

Tabelle 13: Erfüllung der Feature-Liste aus der Analyse, Haken symbolisieren eine vollständige, Halbkreise eine unvollständige Umsetzung

bereits beherrscht, da die gesamte Entwicklung von CAKE - insbesondere das Entwickeln von Sensoren und Aktuatoren - in dieser Sprache vorliegt. Es wird ebenso davon ausgegangen (vergleiche dazu die HTA aus Abbildung 5 zum Ablauf der Erschließung einer neuen Domäne), dass vor Verwendung des Reasoners die Sensoren vom Entwickler implementiert wurden, daher ist vorauszusetzen, dass dieser in der Lage ist, die bereits beschriebenen Kommunikationsmethoden in der CAKE-API aus Sicht eines Sensors oder Aktuators zu nutzen.

Da XML als Sprache weit verbreitet ist, kann davon ausgegangen werden, dass ein Entwickler zumindest in Grundzügen Kenntnisse besitzt. Die zu schreibende Modellierungsdatei beschreibt eine sehr einfache Syntax, die wenig komplizierte Schachtelung in der Modellierungssprache benutzt und mit Hilfe einer XSD-Datei oder anhand von Beispielmodellierungsdateien schnell zu verstehen ist. Der Entwickler kann diese Vorgaben nutzen, ohne sich tiefgehend mit anderen Sprachen zu beschäftigen und letztlich auch, ohne Weka genauer zu kennen, sofern ein bereits bestehendes Reasoningverfahren in Betracht gezogen wird.

5.4 Ergebnisse

Nun werden die aus der Einleitung eingangs beschriebenen Ziele aus Abschnitt 1.2 rekapituliert und mit den Ergebnissen aus der Evaluation abgeglichen. Dieser Abschnitt diskutiert zunächst, inwieweit die Ziele der Arbeit im Allgemeinen erfüllt wurden und zieht anschließend weitere Schlüsse aus der technischen Evaluation.

Die funktionale Evaluation zeigte bereits auf, welche in der Analyse erhobenen Funktionen in welcher Weise umgesetzt wurden. Im Folgenden soll eine Auflistung an Eigenschaften des umgesetzten Reasoning-Systems verdeutlichen, wie diese die Erfüllung der Ziele beeinflussen. Diese Eigenschaften ergeben sich aus den Funktionalitäten, die die funktionale Evaluation hervorbrachte. Die technische Evaluation zeigte unter anderem, dass die grobe Funktionsweise des Systems sichergestellt ist.

Die der Arbeit zugeordneten Ziele (beschrieben in der Einleitung in Abschnitt 1.2) wurden durch die genannten Eigenschaften und Funktionen des Systems verfolgt und bearbeitet. Die Ziele im Hinblick auf das Thema der Arbeit, der Erweiterung der Reasoningfähigkeit von CAKE, können hierdurch als erfüllt angesehen werden. Mögliche offene Punkte oder Schwächen der Umsetzung werden dazu im kommenden Kapitel 6 dargelegt. Die Funktionsweise des neuen Reasoners findet sich in den gezeigten Eigenschaften wieder und wurde durch eine technische Evaluation auf effektive Funktionsweise getestet. Die hier aufgelisteten Funktionen sind demnach verwendbar und tragen zur Zielführung bei. Ebenso hat die technische Evaluation Effizienzfragen zum Reasoning beantwortet, wodurch die Brauchbarkeit des Reasoners im realen Einsatz realistisch erscheint.

Neue Anwendungsdomänen:

+ Freie Deklaration von Sensorvariablen

+ Reasoningverfahren nur bedingt abhängig von Sensortypen, ansonsten freie Gestaltung

Mächtigkeit des Reasonings:

+ Reasoning von beliebigen diskreten Zustandsmengen
+ Reasoning-Berechnung von diskreten und auch numerischen Werten möglich (unter Berücksichtigung der jeweiligen ML-Verfahren)
+ Senden und Empfangen von ML-Reasoningergebnissen unter Reasonern in CAKE

Einfache Adaptierbarkeit:

+ Modellierung in eigener Modellierungsdatei in XML
+ Anpassung der Modellierung und des Reasoningalgorithmus über genannte XML-Datei
+ Direkte Wahl verschiedener Reasoningverfahren über XML-Tag
+ Erweitertes Reasoningverfahren mit Weka durch Einbindung in XML möglich

Lernen zur Laufzeit:

+ CAKE-Neustart erzeugt automatisch neues Model aus allen gespeicherten Daten
+ Erweiterung des Datensatzes zur Laufzeit über Aktuator möglich (Feedback)
+ Datensätze können in zugehöriger *.arff*-Datei ständig angepasst / erweitert werden

Abschnitt 5.2 führte drei Testumgebungen ein, auf die verschiedene Reasoningverfahren getestet und evaluiert wurden. Die ermittelten Zeiten und Aussagen der jeweiligen Testprozeduren liefern einen Eindruck darüber, wie hoch skaliert der Machine Learning Reasoner in Bezug auf Datenmengen und paralleles Reasoning verwendet werden kann und welche ML-Verfahren etwa Vorteile in bestimmten Anwendungsdomänen aufzeigen.

Effizienz. Im Test ergaben sich Initialisierungszeiten für einzelne Reasonerinstanzen von 1,64ms bis zu 1,457s. Diese Unterschiede in der Dauer lassen sich auf die Anzahl der vorliegenden Datenmengen, aber auch auf das gewählte Reasoningverfahren zurückführen. Interessant ist hierbei der Aspekt des Lernens zur Laufzeit. Grundsätzlich geht das Aktualisieren des Models einher mit dem Bau des Klassifizierers, welcher in Tabelle 8 zeitlich analysiert wurde. Möchte eine CAKE-Umgebung also sehr häufig den Lernprozess aktiv umsetzen und das Model oft aktualisieren, so ergibt sich eine obere Schranke durch die Bauzeit der Reasonerinstanz. Bei der im Test aufgetretenen Obergrenze von 1,457s kann der Aktualisierungsprozess also nur 0,686-mal die Sekunde ausgeführt werden. Werden dazu parallel mehrere Reasoner geführt und die Datenmengen vergrößern sich, sinkt diese Frequenz weiter, sodass der Aktualisierungsvorgang des Lernprozesses nur in zeitlich geplanten Abständen umgesetzt werden kann.

Analog dazu wurden die Klassifizierungszeiten und somit die Reasoningprozedur getestet. Die in Tabelle 12 abgebildeten Zeiten zeigen im Mittel eine Dauer von 1,84ms. Überwiegend lag die Klassi-

fizierungszeit unter einer Millisekunde. Je nach Reasoningverfahren ändern sich die Zeiten aufgrund der Datenmenge. Hier wird deutlich, dass eine Anwendungsdomäne oder mehrere an CAKE angeschlossene Domänen nur ein begrenztes Volumen an Reasoninganfragen pro Zeiteinheit besitzen. Betrachtet man Klassifizerungszeiten des `NaiveBayes`-Ansatzes, dessen Reasoningzeiten mit größeren Datenmengen nicht skalieren, dann ergeben sich ca. 0,12ms pro klassifizierte Instanz. Hieraus folgt eine Frequzen an Reasoninganfragen an diesen Reasoner von 8333 Anfragen pro Sekunde. Verwendet man nun mehrere Reasoner parallel, so wird diese Anzahl auf die laufenden Instanzen aufgeteilt. Das Volumen an Klassifizierungen pro Zeit schränkt sich schnell ein, wenn die Anzahl der Anwendungsdomänen steigt oder komplexere Reasoningverfahren gewählt werden.

Effektivität. Die Aussagekraft des Machine Learning Reasoners wurde mit den drei Testumgebungen evaluiert. Wie bereits Tabelle 11 zeigte, wurden Reasoningergebnisse ungenauer, je ungenauer oder komplexer die Testumgebung modelliert und mit Datenmengen fundiert war. So konnte das mathematisch recht übersichtliche Beispiel der Heating-Testumgebung in fast allen Konfigurationen die richtigen Resultate liefern, während die Activity-Umgebung in deutlich mehr Testfällen falsche Ergebnisse hervorbrachte. Es wurde ersichtlich, dass die zugrunde liegende Datenmenge einen großen Einfluss auf die Genauigkeit des Reasoners nimmt, sofern die Modellierung an Komplexität zunimmt. Die besten Ergebnisse im Bereich dieses Reasoningtests erzielte der `MutlilayerPerceptron`-Reasoner, also ein künstliches neuronales Netz, während die Graphsuche des `KStar`-Algorithmus am häufigsten falsch lag. Die Wahl des Reasoningverfahrens ist also je nach Anwendungsdomäne und Ansprüche zu wählen, denn das in diesem Fall effektivste Verfahren zeigte auf der anderen Seite große Initialisierungs- und damit Lernzeiten.

6 Zusammenfassung und Ausblick

In diesem Abschluss der Arbeit werden die umgesetzten Komponenten und damit die erreichte Ziele und Eigenschaften reflektiert. Dazu gibt Abschnitt 6.1 eine Übersicht über das entstandene Produkt. Des Weiteren beschäftigen sich Abschnitt 6.2 und Abschnitt 6.3 mit offenen Punkten in der Entwicklung und darüber hinaus Weiterentwicklungsmöglichkeiten, die den Reasoner betreffen, der aus dieser Arbeit hervorgeht.

6.1 Zusammenfassung

Dieser Abschnitt stellt eine Zusammenfassung der umgesetzten Komponenten des Reasoners dar und beschreibt die Funktionen, die mit dieser Arbeit umgesetzt wurden.

Mit dem Machine Learning Reasoner entstand im CAKE-Framework ein eigenständiger Reasoner, der neben dem bestehenden Rulebased Reasoner den Reasoningprozess in der Logikschicht übernimmt. Dabei arbeitet der Reasoner komplett parallel und steht damit nicht im Konflikt mit dem zuvor bestehenden System. Der Machine Learning Reasoner bietet eine zusätzliche Verarbeitung von Sensorwerten an.

Die Verarbeitung der Eingangsdaten im Hinblick auf semantische Schlussfolgerungen wurden nach dem technischen Verfahren des Machine Learnings umgesetzt. Diese Technik ist mit Hilfe des Weka-Frameworks an das CAKE-System angebunden, sodass die innere Logik des Reasonings bzw. der ML-Algorithmen zunächst durch die Implementierung von Weka vorliegt.

Die Anbindung von Sensoren und deren Verknüpfung zu einer Anwendungsdomäne, bestehend aus einer Vielzahl an Werten von mehreren Sensoren, lässt sich einfach in der XML-Sprache beschreiben und für beliebige Domänen parallel entwickeln. Sensoren und Aktuatoren können mehrfach in der Modellierung von Reasonerinstanzen vorkommen.

Zusätzlich bietet die Modellierung in XML die Möglichkeit, im Java-Code umgesetzte und an CAKE angebundene Weka-Reasoner einfach für einen Anwendungskontext auszuwählen, indem die zugehörige Bezeichnung des Reasoners als XML-Tag in der Modellierungsdatei eingetragen wird.

Zu der bisherigen Anbindung von Weka-Reasonern an CAKE können darüber hinaus weitere er-

zeugt werden. Diese lassen sich im Java-Code entlang einer objektorientierten Architektur schnell hinzufügen. Ebenso können ML-Algorithmen, die nicht in Weka vorliegen, im Rahmen der Weka-Schnittstellen implementiert und im Anschluss an CAKE angebunden werden. Für Verwendung von nicht angebundenen Reasonern ohne Anpassung des Java-Codes bieten externe XML-Dateien die Möglichkeit, von der Reasonerinstanz eingelesen und verwendet zu werden.

Über die Kommunikationsschicht von CAKE lassen sich nun Nachrichten seitens des Aktuators an den Reasoner schicken. Mit Hilfe dieses Feedbacks kann dem Reasoner sofort oder zu einem beliebigen Zeitpunkt mitgeteilt werden, wie eine bestimmte Belegung von Sensoren zu deuten ist. Dieser Vorgang fördert das aktive Lernen des Machine Learning Reasoners.

Die Datensätze für den Reasoner liegen komplett im dazugehörigen Unterverzeichnis des `res/`-Ordners. Die dazu verwendete Datenhaltung ist derzeit durch die mit Weka gelieferten *.arff*-Dateien umgesetzt. Neben den Datensätzen finden sich auch die Modellierungsdateien der einzelnen Reasonerinstanzen in diesem Unterbaum der Ordnerstruktur wieder.

6.2 Offene Punkte

Im Abschnitt 5.3 wurde mit Tabelle 13 aufgeführt, dass zwei Teilziele der Feature-Liste aus der Analysephase nur bedingt umgesetzt wurden. Im Folgenden sollen die bisherigen nicht umgesetzten Teilfunktionen des Reasoners betrachtet werden. Dabei bisher entstandene Schwierigkeiten und mögliche spätere Lösungsansätze beschreibt der folgende Abschnitt.

Der Machine Learning Reasoner lässt sich komplett über die jeweiligen XML-Modellierungsdateien konfigurieren und steuern. Ebenso erlangt der Entwickler für CAKE Zugriff auf die vorhandenen Daten, indem er zugehörige *.arff*-Datei öffnet und gegebenenfalls bearbeitet. Diese Zugriffe auf die Dateien im vorliegenden Dateisystem müssen derzeit lokal am Serversystem ausgeführt werden. Ein offener Ansatz besteht darin, die Zugriffe auf diese Dateien über ein Webinterface in der von CAKE bereitgestellte Web-GUI[1] zu realisieren. Dieser Zugriff benötigt eine Umsetzung des Datenaustauschs zwischen Web-GUI und dem Reasoner über die Kommunikationsschicht von CAKE. Des Weiteren führt dies zu einer Implementierung von entsprechenden Methoden im Machine Learning Reasoner, die den Datenzugriff durch die Kommunikationsschicht lesend und schreibend ermöglichen.

Die Evaluation zeigte, dass die Konfiguration und das Lernen zur Laufzeit von CAKE nur bedingt möglich ist, da für dieses Vorhaben das gesamte CAKE-System neu gestartet werden muss. Damit einhergehend werden alle Module, lokale Sensoren und Reasoner neu initialisiert. Ein Neuladen einer einzelnen Reasonerinstanz oder zumindest des Machine Learning Reasoners würde den Gesamtaufwand reduzieren und Arbeit bei der Entwicklung und Wartung neuer Anwendungskontexte erleichtern. Diese Funktion ließe sich in die im vorherigen Unterpunkt angesprochene Web-GUI von CAKE integrieren. Die Umsetzung von derartigen Methoden im Machine Learning Reasoner sind in der

[1] Graphical User Interface

aktuellen Architektur über entsprechende Neuinstanziierungen der jeweiligen Java-Klassenobjekte (`Builder` oder `Model`) zu realisieren.

Die Evaluation ergab zudem, dass die Datenschicht nur eingeschränkt frei erweiterbar und verwendbar ist im Zusammenhang mit der Architektur des Machine Learning Reasoners. Die aktuelle Umsetzung erfordert, dass *.arff*-Dateien als Datenbestand vorliegen. Ebenso erwartet ein Reasoningverfahren, dass alle verwendbaren Anwedungskontexte die gleiche Art der Datenhaltung verwenden (d.h. die Datensätze werden abhängig vom implementierten `Builder` vom gleichen Dateiformat gelesen). Eine Abstrahierung der Datenschicht über zusätzliche I/O[2]-Klassen würde die Nutzbarkeit verbessern, indem jedes Model seine eigene Datenhaltung realisiert.

6.3 Ausblick

Der Machine Learning Reasoner erfüllt die Grundfunktionalitäten. Im Hinblick auf das verwendete Framework Weka und den späteren Einsatz von CAKE erscheinen Erweiterungen am Reasoner sinnvoll. Dieser Abschnitt beschreibt weitere mögliche Arbeiten am bestehenden System.

In der Analyse wurde erwägt, Sensorwerte unterschiedlich zu gewichten. Diese Eigenschaft der Modellierung des Anwendungskontextes lässt sich in der Logik von Weka umsetzen. Eine Weiterentwicklung würde also beinhalten, Gewichte in die XML-basierte Modellierung aufzunehmen und sie in der Verwendung mit Weka zu übernehmen.

Neben der normalen Weiterentwicklung durch Anbindung mehrerer Weka-Reasoner können zudem neue Reasoner implementiert werden. Diese dann an CAKE anzubindenen Klassifizierer erben von der Weka-Klasse `Classifier`. In der Konkretisierung dieser Klassen können Logik-interne Methoden und Machine Learning Algorithmen frei implementiert werden, falls Weka diese nicht zur Verfügung stellt.

Die im Abschnitt 6.2 beschriebene Verwendung der Web-GUI stellt eine für den späteren Betrieb brauchbare Funktionalität dar, die eine Weiterentwicklung an CAKE sinnvoll macht. Die Umsetzung bedarf wie bereits beschrieben eine Erweiterung der CAKE-Kommunikationsschicht sowie der entsprechenden Endpunkte: die Web-GUI selbst und der Machine Learning Reasoner.

[2] Input/Output

Abbildungen

Tabellen

Quelltexte

Quellen

Literatur

Auwetter, S. & Thomsen, K. (2013). *Non-cubic Cubus*. Projektarbeit, Universität zu Lübeck.

Bouck-Standen, D., Dubbels, T., Eberhardt, B., Jehle, E., Kock, S., Schulze, A. & Wilken, D. (2013). *Dokumentation zur Fallstudie Context Aware Knowledge- and sensor-based Environment*. Fallstudie, Universität zu Lübeck.

Bouckaert, R. R., Frank, E., Hall, M., Kirkby, R., Reutemann, P., Seewald, A. & Scuse, D. (2013). *WEKA Manual for Version 3-6-10*. Handbuch, University of Waikato.

Chawla, N. & Bowyer, K. (2011). Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16, 321–357. URL http://arxiv.org/abs/1106.1813.

Frahm, O. (2011). *MATe-Kommunikationsframework zur Einbindung multipler Reasoner*. Masterarbeit, Universität zu Lübeck.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. & Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1), 10–18. URL http://doi.acm.org/10.1145/1656274.1656278.

Herczeg, M. (2009). *Software-Ergonomie: Theorien, Modelle und Kriterien für gebrauchstaugliche interaktive Computersysteme*. München: Oldenbourg Wissenschaftsverlag GmbH. URL http://www.oldenbourg-link.com/doi/book/10.1524/9783486595406.

Hu, B., Wang, Z. & Dong, Q. (2012). A Modeling and Reasoning Approach Using Description Logic for Context-Aware Pervasive Computing. In Lei, J., Wang, F., Deng, H. & Miao, D. (Hrsg.), *Emerging Research in Artificial Intelligence and Computational Intelligence*, Communications in Computer and Information Science. Springer Berlin Heidelberg, S. 155–165. URL http://dx.doi.org/10.1007/978-3-642-34240-0_21.

Kindereit, M., Momsen, S., Röhr, K. & Weiss, T. (2011). *MATe : Sensoren und Aktuatoren Kurzfassung*. Projektarbeit, Universität zu Lübeck.

Othman, M. & Yau, T. (2007). Comparison of Different Classification Techniques Using WEKA for Breast Cancer. In Ibrahim, F., Osman, N., Usman, J. & Kadri, N. (Hrsg.), *3rd Kuala Lumpur International Conference on Biomedical Engineering 2006*, Band 15 von *IFMBE Proceedings*. Springer Berlin Heidelberg, S. 520–523. URL http://dx.doi.org/10.1007/978-3-540-68017-8_131.

Parsia, B., Sirin, E., Cuenca, B., Ruckhaus, E. & Hewlett, D. (2005). Cautiously Approaching SWRL. Preprint submitted to Elsevier Science. URL https://cs.uwaterloo.ca/~gweddell/cs848/SWRL_Parsia_et_al.pdf.

Pitakrat, T., van Hoorn, A. & Grunske, L. (2013). A comparison of machine learning algorithms for proactive hard disk drive failure detection. In *Proceedings of the 4th international ACM Sigsoft symposium on Architecting critical systems*, ISARCS '13. New York, NY, USA: ACM, S. 1–10. URL http://doi.acm.org/10.1145/2465470.2465473.

Ranganathan, A. & Campbell, R. (2003). An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing*, 7(6), 353–364. URL http://dx.doi.org/10.1007/s00779-003-0251-x.

Ruge, L. (2010). *Koordinierung von Inferenz- und Lernprozessen für Pervasive-Computing-Anwendungen*. Diplomarbeit, Universität zu Lübeck.

Schmitt, J., Hollick, M., Roos, C. & Steinmetz, R. (2008). Adapting the User Context in Real-time: Tailoring Online Machine Learning Algorithms to Ambient Computing. *Mobile Networks and Applications*, 13(6), 583–598. URL http://www.springerlink.com/index/10.1007/s11036-008-0095-8.

Wilken, D. (2012). *Identifikation der Benutzeraktivität im MATe-System*. Bachelorarbeit, Universität zu Lübeck.

Witten, I. H., Frank, E., Trigg, L., Hall, M., Holmes, G. & Cunningham, S. J. (2000). *Weka: Practical Machine Learning Tools and Techniques with Java Implementations*. Computer Science Working Papers. Hamilton, New Zealand: Morgan Kaufmann Publishers Inc.

Software

Eclipse IDE for Java EE Developers, Version: Juno Service Release 1,
http://www.eclipse.org/downloads/

Creately,
http://creately.com/

Java Development Kit 1.7.0_10,
http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html

Weka 3.6.9,
http://www.cs.waikato.ac.nz/ml/weka/index.html

yED,
http://www.yworks.com/en/products_yed_about.html

Abkürzungen

CAKE	-	Context Aware Knowledge- and sensorbased Environment
CBR	-	Case Based Reasoning
DAA	-	Desktop Activity Analyser
DTD	-	Document Type Definition
ER	-	Entity Relationship
GUI	-	Graphical User Interface
HTA	-	Hierarchical Task Analysis
I/O	-	Input/Output
MATe	-	MATe for Awareness in Teams
RBR	-	Rule Based Reasoning
SPARQL	-	SPARQL Protocol and RDF Query Language
SMOTE	-	Synthetic Minority Oversampling TEchnique
SWRL	-	Semantic Web Ontology Langage
Weka	-	Waikato Environment for Knowledge Analysis
WWW	-	World Wide Web
XML	-	Extensible Markup Language
XMP	-	Extensible Messaging and Presence
XSD	-	XML Schema Definition

Glossar

CAKE-API	-	Java Programmschnittstellen von CAKE
Context Awareness	-	Bewusstsein und Erkennung von Zuständen in der Umgebung
Kardinalität	-	Anzahl an Elementen einer bestimmten Menge
Map	-	Datenstruktur, die eine Abbildung von Schlüssel-/Wertepaaren ermöglicht
Model	-	Reasoning: hält Datensätze vor und kann Schlussfolgerungen berechnen
Ontologie	-	Meist ein Graph, der Rollen und Beziehungen zwischen Objekten ausdrückt
Relation	-	In Weka: Eine Sammlung von Instanzen, die aus einer bestimmten Menge von Attributen definiert sind
Resampling	-	Verfahren, welches gegebene Daten zufällig kopiert und/oder ersetzt
Walkthrough	-	Iteratives Durchlaufen einer Liste an Komponenten oder Objekten
(CAKE-)Whiteboard	-	Zentrales Modul in der Logikschicht von CAKE: zuständig für die Verwaltung aller angeschlossenen Plugins und Reasoner und dessen Kommunikation

Anhänge

In diesem Abschnitt wird Programmcode aufgeführt, der aus dem Hauptteil der Arbeit hervorgeht. Ebenso wird die konkrete Benutzung des Machine Learning Reasoners zur Erzeugung einer neuen Reasonerinstanz beschrieben.

A Programmcode

Dieser Abschnitt beschreibt einige Kernmethoden und -klassen im Java-Projekt.

A.1 Java-Code

Hier erscheinen die wichtigsten Auszüge aus dem Java-Quellcode für Grundfunktionalitäten oder Abläufe.

Der initiale Prozess in der Erzeugung des Machine Learning Reasoner ist die Initialisierungsmethode, die in Quelltext 11 gezeigt wird. Hier werden Datenstrukturen erzeugt, Sensoren abonniert und ein Listener für Nachrichten aus dem Feedbackprozess registriert. Abschließend erzeugt die Methode den `ModelInstanceManager`, der dann alle eingehenden Daten enthält. Dieser Prozess wird im darauffolgenden Quelltext 12 abgebildet. Zunächst ermittelt die Methode eine Liste an Models, die an den eingehenden Daten interessiert sind. Diese werden im Folgenden mit den neuen Werten beschrieben und im Falle einer Änderung von Attributbelegungen führt die Methode eine Klassifizierung aus, indem die Daten in das Weka-konforme `Instance`-Objekt überführt wird. Abschließend geht das Ergebnis der Klassifizierung als `ActuatorUpdate` an alle spezifizierten Aktuatoren.

Ein Beispiel für eine Implementierung der abstrakten `Builder`-Klasse bietet Quelltext 13. Hierbei wird die Preprozessierung mittels der Weka-Werkzeuge mit einem `SMOTE`-Filter durchgeführt und danach der Klassifizierer selbst gebaut, was ebenfalls durch das Weka-Framework geschieht.

```
@Override
public void initializeReasoner() {
  // init all the device lists
  this.availableSensors = new ArrayList<>(
      this.manager.getRegisteredSensors());
  this.availableActuators = new ArrayList<>(
      this.manager.getRegisteredActuators());
  this.virtualDevices = new ArrayList<>(
      this.manager.getRegisteredVirtualIOElements());
  this.availableReasoners = new ArrayList<CakeReasoner>(
      this.manager.getRegisteredReasoners());
  this.isActive = true;

  // easily subscribe to all sensors
  for (Sensor s : availableSensors) {
    manager.subscribeReasonerToDevice(this, s.getID());
  }
  // and to all reasoners
  for (CakeReasoner reas : availableReasoners) {
    manager.subscribeReasonerToDevice(this, reas.getUID());
  }

  // Register its listener to message queue
  MessageQueue.addMessageQueueListener(this, this);

  // init the modelinstancemanager
  miMgr = new ModelInstanceManager(this);
  miMgr.initializeReasoners();
}
```

Quelltext 11: MachineLearningReasoner::initializeReasoner

```
void postInputUpdate(InputUpdate update) {
  List<Model> models = assignedModels.get(extractUID(update
      .getSourceUID()));
  if (models == null)
    return;
  for (Model mdl : models) {
    if (mdl.getState() != Model.STATE_READY) {
      // Skip this model if not ready for reasoning
      logger.debug("Skipped one reasoning model. Status: Not Ready");
      continue;
    }
    // create an instance and give it to the model
    boolean changed = false;
    MappedInstance inst = new MappedInstance(mdl.getLastInstance());
    for (Entry<String, Object> entry : update.getUpdateValue()
        .entrySet()) {
      if (inst.setValue(entry.getKey(), entry.getValue()) != null) {
        changed = true;
      }
    }
    if (changed) {
      // If a value has changed, is might be necessary to start
      // reasoning
      Object result = null;
      Instance classInst = mdl.classify(inst);
      if (classInst.classAttribute().isString()
          || classInst.classAttribute().isDate()) {
        result = classInst.classAttribute().value(0);
      } else if (classInst.classAttribute().isNominal()) {
        result = classInst.classAttribute().value(
            (int) classInst.classValue());
      } else {
        result = classInst.classValue();
      }
      logger.debug("Sensorupdate changed values. Reasoning-Result is "
          + result);
      // Fetch all output-actuators and finally send the result
      for (String act : mdl.getDeliveredActuators()) {
        Map<String, Object> values = new HashMap<String, Object>();
        values.put("__complete_data__", mlReasoner.toString()
            .concat("#").concat(mdl.toString()).concat("#")
            .concat(classInst.toString()));
        values.put(classInst.classAttribute().name(), result);
        mlReasoner.postActuatorUpdate(act, values);
      }
    } else {
      logger.debug("Sensorupdate delivered no changes ...");
    }
  }
}
```

Quelltext 12: ModelInstanceManager::postInputUpdate

```
public class SmotedNaiveBayes extends Builder {

  @Override
  protected boolean preprocessData() throws Exception {
    SMOTE f = new SMOTE();
    f.setPercentage(1000);
    f.setInputFormat(dataset);
    for (int i = 0; i < dataset.numInstances(); i++) {
      f.input(dataset.instance(i));
    }
    f.batchFinished();
    Instances filtered = new Instances(dataset);
    Instance out = f.output();
    while (out != null) {
      filtered.add(out);
      out = f.output();
    }
    dataset = filtered;
    return true;
  }

  @Override
  protected Classifier build() throws Exception {
    Classifier cls = new NaiveBayes();
    cls.buildClassifier(dataset);
    return cls;
  }

}
```

Quelltext 13: SmotedNaiveBayes.java

A.2 Weitere Klassen/Dokumente

Neben dem Java-Quellcode liegen in diesem Abschnitt zusätzliche Dateien im Zusammenhang mit dem Machine Learning Reasoner, wie sie unter anderem auch in der Konzeption und Realisierung erwähnt und beschrieben wurden.

Quelltext 14 zeigt die in der Evaluation verwendete Museumsumgebung als vollständige *.arff*-Datei. Diese liegt so im aktuellen CAKE-Repository vor und wird durch den Reasoningprozess eingelesen und verwendet. Die Modellierung einer Reasonerinstanz ist durch eine *reasoner.xml*-Datei vorgesehen, die analog zum Datensatz aus Quelltext 14 die Einbindung in CAKE beschreibt. Dieses XML-Dokument findet sich in Quelltext 15. Die dazugehörige XSD-Datei legt die Modellierung auf die konzipierte Art fest. Diese findet sich aufgeteilt in Quelltext 16 und 17 und dient der Validierung einer jeden neuen Modellierung einer Reasonerinstanz.

```
@relation 'Museum Einfacher Reasoner'

@attribute type {agriculture,politics,architecture}
@attribute size numeric
@attribute year numeric
@attribute idle numeric
@attribute speed numeric
@attribute textform {huge,augmented,small}

@data
agriculture,20,100,0.8,1.5,huge
agriculture,20,100,0.95,1,huge
agriculture,20,100,0,2,small
agriculture,20,100,0.5,1.5,augmented
agriculture,12,20,0.95,0.5,huge
agriculture,12,20,0.6,1,augmented
agriculture,12,20,0.4,1.5,small
politics,14,280,0.1,2,small
politics,14,280,0.3,1,augmented
politics,14,280,0.2,0.5,augmented
politics,1,10,0.95,0.4,huge
politics,1,10,0.9,0.7,huge
politics,1,10,0,1.5,small
architecture,40,150,0.2,0.2,augmented
architecture,40,150,0.8,0.4,huge
architecture,40,150,0.1,2,small
architecture,40,150,0.2,0.5,small
architecture,20,40,0.6,1,augmented
architecture,20,40,0.1,1.5,small
architecture,20,40,0.8,0.35,huge
```

Quelltext 14: test_reasoner_museum.arff

```
<reasoner>

  <declaration>
    <name>Museum Einfacher Reasoner</name>
    <description>Dieser Reasoner gibt die Entscheidung an, welche Textform angezeigt werden
        soll</description>
    <sensors>
      <name>bbd19fe8-5d8d-a2ac-e537-9ca56c75000d</name>
    </sensors>
    <actuators>
      <name>924ec1f2-61d5-dd23-e8e5-11a78485000c</name>
    </actuators>
  </declaration>

  <model>
    <field>
      <label>type</label>
      <type>nominal</type>
      <class>false</class>
      <nominals>
        <option>agriculture</option>
        <option>politics</option>
        <option>architecture</option>
      </nominals>
    </field>
    <field>
      <label>size</label>
      <type>numeric</type>
      <class>false</class>
    </field>
    <field>
      <label>year</label>
      <type>numeric</type>
      <class>false</class>
    </field>
    <field>
      <label>idle</label>
      <type>numeric</type>
      <class>false</class>
    </field>
    <field>
      <label>speed</label>
      <type>numeric</type>
      <class>false</class>
    </field>
    <field>
      <label>textform</label>
      <type>nominal</type>
      <nominals>
        <option>huge</option>
        <option>augmented</option>
        <option>small</option>
      </nominals>
      <class>true</class>
    </field>
  </model>

  <reasoning>
    <default_reasoner>KStar</default_reasoner>
  </reasoning>

</reasoner>
```

Quelltext 15: test_reasoner_museum.xml

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="reasoner">
    <xs:complexType>
      <xs:all>

        <xs:element name="declaration">
          <xs:complexType>
            <xs:all>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="description" type="xs:string" minOccurs="0" maxOccurs="1"/>
              <xs:element name="sensors">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="name" type="xs:string" minOccurs="0" maxOccurs="
                        unbounded" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="actuators">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="name" type="xs:string" minOccurs="0" maxOccurs="
                        unbounded" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:all>
          </xs:complexType>
        </xs:element>

        <xs:element name="model">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="field" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:all>
                    <xs:element name="label" type="xs:string"/>
                    <xs:element name="type">
                      <xs:simpleType>
                        <xs:restriction base="xs:string">
                          <xs:enumeration value="numeric"/>
                          <xs:enumeration value="string"/>
                          <xs:enumeration value="date"/>
                          <xs:enumeration value="nominal"/>
                        </xs:restriction>
                      </xs:simpleType>
                    </xs:element>
                    <xs:element name="nominals" minOccurs="0">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="option" maxOccurs="unbounded" type="xs:string"/
                              >
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="date_specification" minOccurs="0"/>
                    <xs:element name="class" type="xs:boolean"/>
                  </xs:all>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
```

Quelltext 16: reasoner.xsd - Teil 1

```
        <xs:element name="reasoning">
          <xs:complexType>
            <xs:choice>
              <xs:element name="path_to_reasoner" type="xs:string" maxOccurs="1"/>
              <xs:element name="default_reasoner" maxOccurs="1">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:enumeration value="NaiveBayes"/>
                    <xs:enumeration value="SmotedNaiveBayes"/>
                    <xs:enumeration value="ResampledNaiveBayes"/>
                    <xs:enumeration value="MultilayerPerceptron"/>
                    <xs:enumeration value="SmotedMultilayerPerceptron"/>
                    <xs:enumeration value="ResampledMultilayerPerceptron"/>
                    <xs:enumeration value="KStar"/>
                    <xs:enumeration value="SmotedKStar"/>
                    <xs:enumeration value="ResampledKStar"/>
                    <xs:enumeration value="ZeroR"/>
                    <xs:enumeration value="RandomForest"/>
                    <xs:enumeration value="OneR"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
            </xs:choice>
          </xs:complexType>
        </xs:element>

      </xs:all>
    </xs:complexType>
  </xs:element>


</xs:schema>
```

Quelltext 17: reasoner.xsd - Teil 2

B Installationsanweisung

B.1 Merging CAKE

Die neuen Funktionen in CAKE sind überwiegend in dem `mlReasoner`-Paket untergebracht. Dieses findet sich zum einen im Quellcode im `logic`-Paket und zum anderen im `res`-Ordner von CAKE. Neben dem Einbinden der Weka-Library wurden aber Änderung an Plugin- und Nachrichtenschicht vorgenommen worden zwecks Feedbackprozess. Alle hinzugefügten und auch geänderten Klassen sind in Abbildung 24 dargestellt. Die ebenfalls einzufügenden Ressourcen zeigt Abbildung 23. Dabei sind beim Merging die geänderten Klassen zu betrachten, die aber meist nur minimale Ergänzungen zum bestehenden Code darstellen.

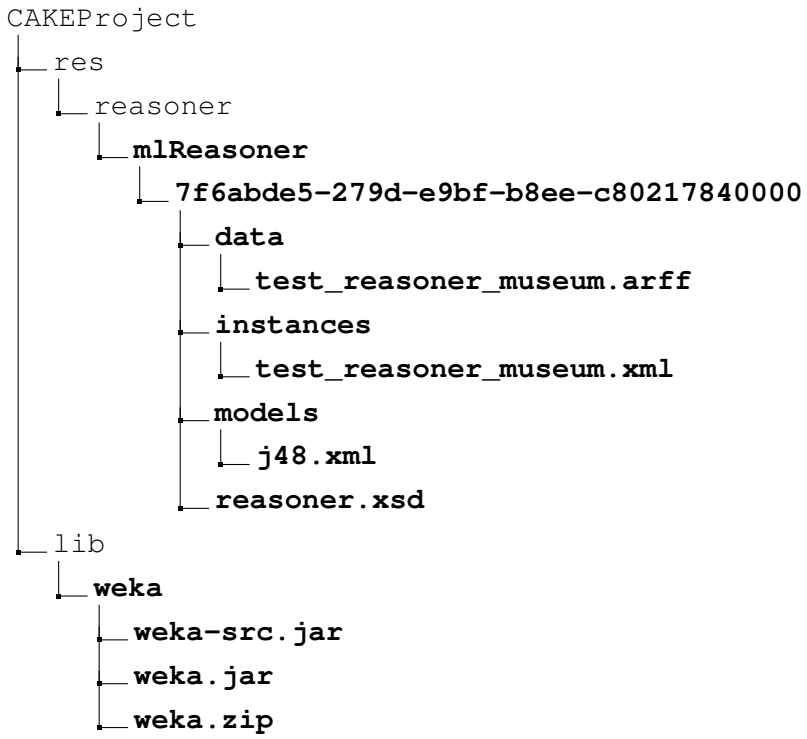

Abbildung 23: Die Ordnerstruktur, die die zusätzlichen Ressourcen und Bibliotheken enthält, **neue Dateien** sind hervorgehoben

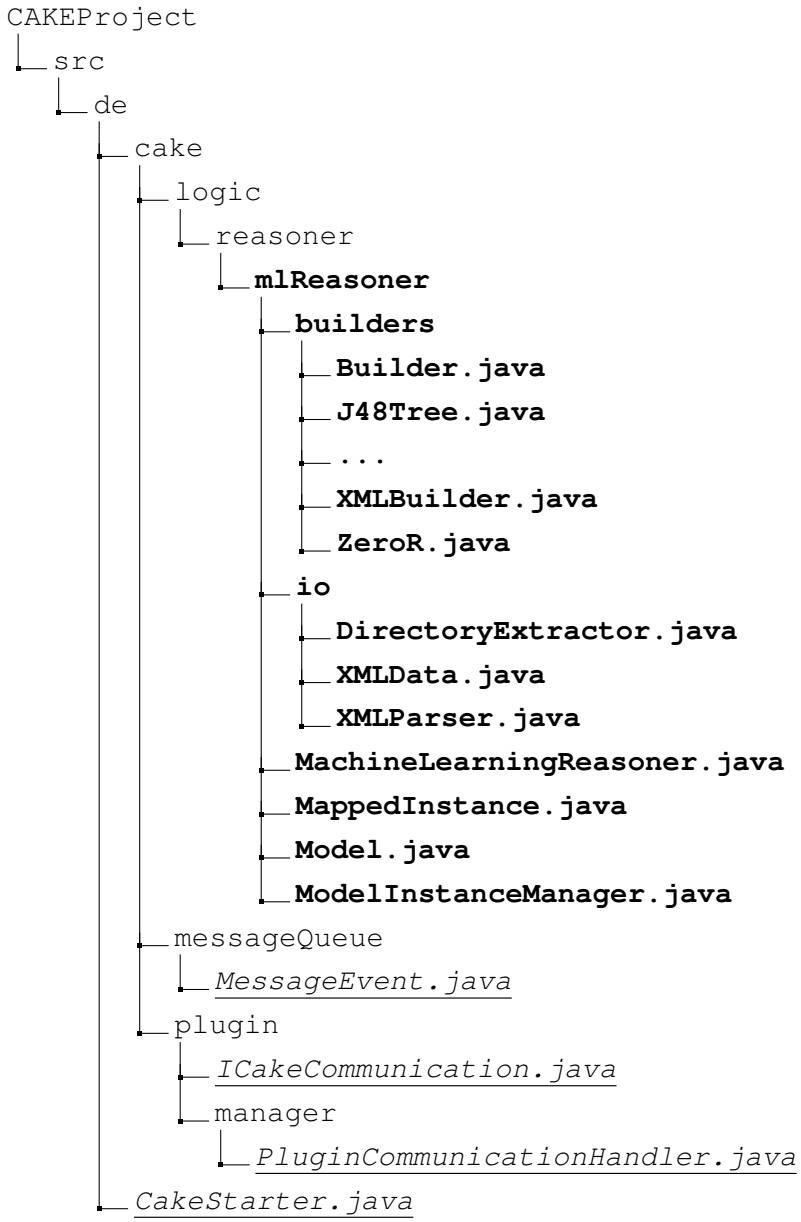

Abbildung 24: Die Ordnerstruktur, die die hinzugefügten und geänderten Dateien im Programmcode enthält, hervorgehoben sind *geänderte* und **hinzugefügte** Dateien

B.2 Entwickeln einer Anwendungsdomäne - Schritt 1: Modellierung

Für die Entwicklung einer neuen Anwendungsdomäne sind zwei Schritte zu tätigen: das Erstellen der *reasoner.xml*-Datei ist der erste, wobei der Name dieser Datei eindeutig sein muss. Diese Datei muss beim CAKE-Start im folgenden Verzeichnis liegen:

`res/reasoner/mlReasoner/[uuid_des_Reasoners]/instances/`

<declaration>. Die Angabe eines eindeutigen Relationsnamens (`name`) und eine optionale Angabe einer Beschreibung (`description`) werden hier erwartet. Zusätzlich werden alle `UUID`s von für diese Anwendungsdomäne relevanten Sensoren hier eingetragen (`sensors`), wie Quelltext 15 im Beispiel der Museumsumgebung im vorherigen Abschnitt zeigte. Aktuatoren werden analog angelegt.

<model>. Hier wird jeder Sensorwert beschrieben, der von abonnierten Sensoren aufgenommen werden soll. Jedes `field` benötigt die Angabe eines eindeutigen Namens (`label`), der zudem äquivalent zu dem Prefix eines Sensorwerts sein muss, einem Datentyp (`type`) und der bool'schen Angabe, ob es sich bei diesem Feld um das Klassenattribut handelt (`class`). Es kann von den Attributen nur ein Klassenattribut gesetzt werden. Je nach Datentyp, die in Abschnitt 3.3 zu Weka aufgeführt wurden und auch in der zugehörigen XSD-Datei aufgezählt sind (vergleiche Quelltext 16), werden weitere Definitionen des Sensorwerts benötigt:

`nominal` Handelt es sich um einen nominalen Wert, also um ein Array an möglichen Werten, dann muss im Anschluss dieses Array in XML spezifiziert werden (`nominals`). In der Museumsumgebung (Quelltext 15) wird beispielhaft die Ausgabe nominal beschrieben (das Feld "textform").

`date` Handelt es sich um ein Datum, so wird entweder das Standardformat als Zeichenkette erwartet ("yyyy-MM-dd HH:mm:ss") oder man gibt dieses Format selbst an, etwa, wenn nur ein Tag ohne Uhrzeit relevant ist. Hierzu wird das Format als Zeichenkette definiert (`date_speficiation`). Quelltext 18 zeigt eine Datumsdefinition eines Felds, in dem nur Stunden, Minuten und Sekunden als Datum erwartet werden im Format "HH:mm:ss".

```
<field>
  <label>activity_time</label>
  <type>date</type>
  <date_specification>HH:mm:ss</date_specification>
  <class>false</class>
</field>
```

Quelltext 18: Beispiel eines Felds in der XML-Modellierung, in der das Datumsformat überschrieben wird

<reasoning>. Für die Anwendungsdomäne kann der Enwtickler nun die gewünschte Art des Reasonings wählen. Dies beinhaltet Laden der Dateien, Preprozessierung und Klassifizieren nach bestimmten ML-Algorithmus. Diese drei Schritte sind standardmäßig in einem Reasoningverfahren zu-

sammengeführt implementiert (siehe Abschnitt 4.4). Die dort beschrieben und verfügbaren Reasoner sind ebenfalls in dem aktuellen XSD-Dokument (*reasoner.xsd*) aufgeführt und in diesem Abschnitt der Modellierung wählbar (`default_reasoner`).

Alternativ zur Angabe dieses XML-Tags kann ein eigener Reasoningalgorithmus gewählt werden, indem der Pfad zu einer XML-Datei angegeben wird (`path_to_reasoner`). Hierbei wird standardmäßig die *.arff*-Datei als Datensatz ausgelesen und keine Preprozessierung durchgeführt. Dafür darf die zu spezifizierende XML-Datei ein von Weka bereitgestelltes Reasoning definieren. Dies basiert auf dem Verfahren der Weka-Kommandozeile. Für diese Verwendung wird weiteres Wissen über Weka benötigt. Ein Beispiel dieser Spezifikation war bereits in Quelltext 9 zu sehen.

B.3 Entwickeln einer Anwendungsdomäne - Schritt 2: Datensätze

Die aktuelle Implementierung des Machine Learning Reasoners basiert auf *.arff*-Dateien als Datenhaltung. Um eine zuvor modellierte Anwendungsdomäne zu benutzen, benötigt der Reasoner in jedem Fall Datensätze, aus denen er ein Model erzeugen kann, um später unbekannte Situationen in der Domäne zu klassifizieren. Nach der Erstellung der *reasoner.xml* sollte CAKE zunächst einmal gestartet werden (und kann anschließend sofort beendet werden). CAKE legt die *reasoner.arff*-Datei an, sofern diese nicht vorhanden ist (was beim ersten Starten der Fall ist). In dieser Datei werden dabei automatisch die nötigen Header-Angaben ausgefüllt, sodass der Entwickler direkt Datensätze in diese Datei eingeben kann.

Das Einfügen neuer Datensätze geschieht über zeilenweises Einfügen in den `@data`-Abschnitt der *.arff*-Datei. Wie in Quelltext 14 zu sehen ist, muss die Reihenfolge der deklarierten Sensorwerte bzw. Attribute persistent bleiben. CAKE erhält die Reihenfolge der XML-Modellierung und fügt die Attribute so in die *.arff*-Datei ein. In jeder Zeile finden sich die Werte zu den Attributen in gleicher Sequenz wieder und werden jeweils mit einem Komma (`,`) voneinander getrennt. Ist ein Wert nicht bekannt, wird dieser mittels eines Fragezeichens aufgenommen (`?`), wie Quelltext 19 veranschaulicht. Die meisten Reasoningalgorithmen unterstützen Datensätze mit unbekannten Attributen.

```
@data
agriculture,?,100,0.8,1.5,huge
?,20,100,?,1.5,augmented
```

Quelltext 19: Ausschnitt einer *.arff*-Datei mit unbekannten Attributen

Der Datenbestand *.arff*-Datei kann jederzeit manuell verändert oder erweitert werden, jedoch nicht die Deklaration (Änderung von Attributtypen oder -reihenfolge). Mit einem Neustart des CAKE-Projekts werden alle ML-Models anhand der *.arff*-Dateien neu erzeugt, d.h. auch hinzugefügte Datensätze verwendet.

B.4 Implementierung neuer `Builder`

Um einen neuen `Builder` verwenden zu können, muss dieser zunächst implementiert werden. Dazu muss die neue Klasse im `builders/`-Paket liegen und von der Basisklasse `Builder` erben. Wie schon in der Realisierung der Arbeit detailliert beschrieben, können nun einzelne Schritte individuell implementiert werden. Quelltext 20 zeigt und kommentiert die einzelnen Schritte.

`loadData()` lädt dabei standardmäßig die Datensätze aus der korrespondierenden *.arff*-Datei und ist in der Regel nicht zu überschreiben.

`preprocessData()` führt standardmäßig keine Änderung aus und kann wie in Quelltext 13 zu sehen für einen Filterungsprozess überschrieben werden.

`build()` wird abstrakt deklariert und verlangt hier nun die Erzeugung eines Weka Klassifizierers (dessen Basisklasse ist der Rückgabetyp `Classifier`). Dieser sollte immer mit dem Datensatz in der konventionell beizubehaltenden Variable `dataset` gebaut und anschließend erst zurückgegeben werden.

```
public class MeinBuilder extends Builder {

  @Override
  protected boolean loadData() {
    // Schritt 1 [OPTIONAL] : Lade Daten in Objekt dataset
    // dataset = new Instances();
    // ...
  }

  @Override
  protected boolean preprocessData() throws Exception {
    // Schritt 2 [OPTIONAL] : Modifizieren der Preprozessierung
    // dataset = filter(dataset);
    // ...
  }

  @Override
  protected Classifier build() throws Exception {
    // Schritt 3 [ZWINGEND] : Erzeugen des Weka-Klassifizierers
    // Hinweis: Der Workflow entspricht immer in etwa dem folgenden:
    Classifier cls = new NaiveBayes();
    cls.buildClassifier(dataset);
    return cls;
  }

}
```

Quelltext 20: Spezifizieren der abstrakten `Builder`-Klasse

Um die neu erzeugte Klasse (im Beispiel `MeinBuilder`) verwenden zu könne, sind zwei Schritte nötig: der Dateiname muss für die Instanziierbarkeit im `Builder` und für die Verwendung in

der XML-Modellierung in der zugehörigen *reasoner.xsd* eingetragen werden. Wie in Quelltext 21 zu sehen ist, wird die Referenz auf die Klasse der neu erzeugten `Builder`-Implementierung hinzugefügt. Dies geschieht in der Klasse `Builder` in der Variablen `buildersByClass`. CAKE kann daraufhin eine Instanz dieser Klasse erzeugen, indem der Methode `createBuilder` lediglich die Bezeichnung der Klasse als Zeichenkette übergeben wird.

```
/**
   * The buildersByClass variable is used to hold all implemented
   * Builder-classes. Once added to this array, the class can be called
      by its
   * simple name specified in the &ltreaoning&gt-tag in the model.xml
   */
  @SuppressWarnings("rawtypes")
  private static final Class[] buildersByClass = { NaiveBayes.class,
      SmotedNaiveBayes.class, ResampledNaiveBayes.class,
      MultilayerPerceptron.class, SmotedMultilayerPerceptron.class,
      ResampledMultilayerPerceptron.class, KStar.class,
      SmotedKStar.class, ResampledKStar.class, ZeroR.class,
      RandomForest.class, OneR.class, MeinBuilder.class };
```

Quelltext 21: `buildersByClass`-Variable zur Definition, welche Implementierungen der `Builder`-Klasse vorliegen

Anschließend erhält das XSD-Dokument den neuen Klassennamen für die Auswahl eines im XML-Tag `<default_reasoner>` zu spezifierenden ML-Verfahrens. Quelltext 22 zeigt analog den relevanten Teil der *reasoner.xsd*-Deklaration. Hier wird ein neuer Eintrag für den Klassennamen des Builders gemacht. Diese neue Implementierung kann nun verwendet werden.

```
<xs:element name="default_reasoner" maxOccurs="1">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="NaiveBayes"/>
      <xs:enumeration value="SmotedNaiveBayes"/>
      <xs:enumeration value="ResampledNaiveBayes"/>
      <xs:enumeration value="MultilayerPerceptron"/>
      <xs:enumeration value="SmotedMultilayerPerceptron"/>
      <xs:enumeration value="ResampledMultilayerPerceptron"/>
      <xs:enumeration value="KStar"/>
      <xs:enumeration value="SmotedKStar"/>
      <xs:enumeration value="ResampledKStar"/>
      <xs:enumeration value="ZeroR"/>
      <xs:enumeration value="RandomForest"/>
      <xs:enumeration value="OneR"/>
      <xs:enumeration value="MeinBuilder"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Quelltext 22: Deklaration in der *reasoner.xsd* aller verwendbarer Reasoningverfahren in der XML-Modellierung

Erklärung

Ich versichere, die vorliegende Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Kevin Böckler

Lübeck, den 19. November 2013