Computerspiele mit Head-up Display

Manuel Rodriguez

14.Oktober 2017

Unpublished manuscript

Inhaltsverzeichnis

1 Einleitung

1.1 Teaching Games

Ein Head-up Display ist ein Unterrichtsmittel. Genauer gesagt wird Künstliche Intelligenz verwendet um ein Computerspiel einem Menschen zu erklären. Der Pilot blickt auf den künstlichen Horizont und lernt dadurch sein Flugzeug zu steuern. Der Autofahrer sieht die Fahrspur auf dem Display und weiß dadurch, wann er gegenlenken muss. Künstliche Intelligenz wird nicht etwa als erstrebenswertes Ziel bezeichnet mit dem man autonome Roboter steuert, sondern Künstliche Intelligenz wird als Unterrichtsmethode verstanden um Menschen leistungsfähiger zu machen. Genauer gesagt wird deren kognitve Load in einem Spiel reduziert, und es werden ihnen Informationen gegeben um bessere Entscheidungen zu treffen. Ein gutes Beispiel hierfür ist die Night-Vision Funktionalität in modernen Driver Assistence Systemen wo der Fahrer trotz Dunkelheit sehr genau die Straßenschilder erkennt.

Leider hat das Konzept des "Teaching Games" einen gewichtigen Nachteil: es verlässt die traditionelle Informatik und Mathematik und erweitert das Problem zu einem Psychologischen. Genauer gesagt geht es um Behaviorismus, Lerntheorie und Didaktik. Üblicherweise ist die Mathematik und die Informatik gut darin, von Menschen und deren Gefühlen zu abstrahieren, also sie nur als Zahlen in einer Gleichung zu betrachten und sich stattdessen mit synthetischen Modellen zu beschäftigen. Wenn man jedoch Künstliche Intelligenz einsetzt um darüber Menschen zu unterrichten und sie durch Spiele zu führen, dann holt man den unberechenbaren Homo Faber zurück in die Gleichung. Man passt das System dem Menschen an.

Generell gilt es zu unterscheiden zwischen Human Teaching und "Machine Learning". Beim Machine Learning geht es um Meta-Algorithmen. Also um neuronale Netze und Q-Learning mit HIlfe derer die Programmierung von Computern erleichtert wird. Die Idee lautet, dass man Computer nicht programmiert, sondern ihnen etwas beibringt. Leider ohne Erfolg, so funktioniert es nicht, und das Thema ist ohne Bedeutung. Interessanter sind hingegen formale Modelle um Menschen etwas beizubringen, also human teaching zu betreiben. Hier ist die Zielperson nicht einfach nur ein Neuronales Netz in das man Werte hineinschreibt, sondern die Zielperson ist sehr viel leistungsfähiger. Genauer gesagt entspricht ihre kognitive Leistung der einer Human-Level-AI, sie kann Sprache verstehen, logische Zusammenhänge erkennen und ihren eigenen Vorteil optimieren.

Die einfachste Methode um Human Teaching zu betreiben ist die Verwendung eines Human-Robot-Interface. Also eines grafischen Modells was von Mensch und Maschine geteilt wird. Der Mensch schaut sich die lustigen Grafiken an, die Maschine nutzt die API um Textwerte auszulesen.

1.2 Augmented Assembly

Der Begriff "head-up Display" beschreibt ein technisches Device, was als Frontscheibe in Flugzeugen eingesetzt wird. Der allgemeinere Begriff lautet "Augmented Reality". Das umfasst HuD genauso wie spezielle Brillen und Smartphones bei denen Realität und Grafik übereinandergelegt sind. Augemented Reality ist nicht so eine konkrete Technologie als vielmehr eine Perspektive die Dinge zu betrachten. EIn Beispiel: vor einiger Zeit hat die Firma Moley Robotics einen Kitchenroboter vorgestellt. Wie er funktioniert ist nirgendwo dokumentiert. Wenn man versucht menr über Roboter in Erfahrung zu bringen und sich mit künstlicher Intelligenz zu beschäftigen wird man das Geheimnis nicht entschlüsseln, weil der Moley Roboter nicht als Roboter sondern als Augmented Kitchen entwickelt wurde. Das System wurde als Hiilfe für einen Menschen programmiert. Das heißt, der Koch sieht auf einem Display wo der Topf ist, vergleichbar mit einem Head-up Display in einem Flugzeug oder vergleichbar mit vielen anderen "Augmented Assembly Projekten". Dieses System wurde dann stückweise zu einem autonomen System erweitert. Der UR5 Roboterarm und die Shadowhand wurde erst ganz am Schluss hinzugefügt, es ist das unwichtigste Element von allen.

Der konkrete Sourcecode des Moley Roboters ist zwar auch in [15] nicht enthalten, aber dafür eine knappe Beschreibung wie Augmented Cooking grundsätzlich funktioniert.

> "Tabletop display can provide rich and intelligent interaction through analysis of the touch input. Hand gesture analyses have three main steps that are hand gesture, command and event. "

Die Idee ist, dass Menschen den French Toast manuell zubereiten, und das System Zusatzinformationen auf dem Bildschirm anzeigt, so ähnlich als wenn man etwas zusammenbaut. Damit die Bildschirmdarstellung funktioniert benötigt man ein Domain-Modell. Das Programmieren von Augmented Reality Systemen ist gleichbedeutend mit dem Erstellen eines Domain-Modells. Dieses wiederum wird verwendet, um einen Roboterarm zu steuern.

Schaut man sich im Gegensatz einmal an, wie normalerweise Robotik-Anwendungen erstellt werden, so beginnt man ohne Domain-Modell und wundert sich dann warum der Problemraum so groß ist und wieso der Roboterarm in Zustände hineinläuft, die keiner vorhergesehen hat. Dann wird noch hastig versucht, on-the-fly ein Domain-Modell zu erstellen (Reinforcement Learning) was ebenfalls nicht funktioniert und als letzte Hoffnung wird dann der Kollege Zufall bemüht (probalastische Robotik) was überhaupt nicht praxistauglich ist. Und so sind klassische Robotik-Projekte zum Scheitern verurteilt.

Die Augmented Reality Fraktion hingegen lässt es sehr viel ruhiger angehen. Man beginnt mit dem Domain-Modell was man für Menschen entwickelt, erweitert das um grafische Elemente und erhält stückweise ein formale Vorstellung was Kochen ist und welche Motion Primitive darin vorkommen. Erst ganz zum Schluss wird ein Roboterarm eingefügt, der dann 1:1 den Menschen ersetzt.

[8] berichtet noch ausführlicher wie solche System im Detail programmiert werden. Als Basis dient die Unity3D Gameengine, für das Tracking der Marker wird ARToolKit verwendet.

Effizienzsteigerung Aktuell wird Augmented Assembly noch als Industrie 4.0 vermarktet. Das Versprechen lautet, dass sich darüber eine Effizienzsteigerung von bis zu 30% erzielen ließe. Der Arbeiter der Zukunft setzt einen Helm mit Kamera auf, nutzt Smartphones und Touchpads und taucht so ein in die virtuelle Realität. Positiv ist natürlich, dass die Industrie das Thema für sich entdeckt hat, aber die 30% mögichen Effizienzsteigerungen dürften weit untertrieben sein. Zur Erinnerung: die eigentliche Starke entfaltet das Konzept, wenn man den Menschen aus der Loop komplett herausnimmt. Also das Domainenmodell nutzt um damit Roboter zu steuern. Genau das wollen oder können die aktuellen Studien jedoch nicht sehen. Sie denken, Augmented Assembly wäre eine Technologie um menschliche Arbeit produktiver zu machen. So nach dem Motto, dass man damit die kognitive Last des Arbeiters reduziert und er pro Stunde nicht 2 PCs sondern 2 1/2 PCs zusammenbaut.

In der Tat, die Effizienzsteigerung ist unbestritten da. Meine eigenen Experimente mit Head-up Displays für Computerspiele haben ergeben, dass man mit smart Interfaces tatsächlich schneller wird in der Bedienung des Systems. Der eigentliche Vorteil liegt jedoch woanders. Das man schneller wird in der manuellen Bedienung ist nur ein Zeichen dafür, dass die gewählte Systemodellierung stimmig ist, also das Head-up Display Sinn macht und den Arbeitsprozess gut abbildet. Das jedoch ist der Startschuss um die eigentliche Automatisierung zu

starten. Zur Erinnerung: es geht nicht darum, die Produktität um läppische 30% zu erhöhen, sondern es geht darum, die Lohnkosten auf 0 zu senken und gleichzeitig die Produktivität um 1000% zu erhöhen. Oder anders gesagt, die Menschenleere Fabrik ist das Ziel. Augmented Assembly ist nur ein Entwicklungsschritt dahin. Nicht etwa Arbeiter setzen sich Helmkameras auf und ziehen sich Datenhandschuhe an, sondern es sind überwiegend Wissenschaftler und Programmierer. Die machen das nicht, um Autos zusammenzubauen, sondern weil sie Software für Roboter programmieren.

Wenn in 10 Jahren noch immer menschliche Arbeiter benötigt werden, um Autos zu montieren ist bei den Augmented Assembly Projekten irgendwas schief gelaufen. Es ist nicht gelungen, das gewonnene Systemmodell zur Robotersteuerung aufzubereiten.

Aber ich will nicht übertriebben kritisch sein: Augmented Assembly Projekte zu starten (egal mit welcher Zielstellung) ist genau der richtige Ansatz. Darüber gelangt man an formale Arbeitsplatzbeschreibungen. Kann also definieren welche Handgriffe benötigt werden, welche Bauteile ausliegen und wie die Task-Ontologien aussehen.

Trotz angestrengter Suche konnte ich keine Literatur finden zum Thema wie man Augmented Assembly Modelle zur Robotersteuerung verwendet. Es gibt lediglich wage Andeutungen über Shared Workplace bei dem Mensch und Roboter sich eine virtuelle Realität teilen und dort Arbeiten ausführen. Dabei wäre die Awendung von erstellten Domain-Modellen zur Robotersteuerung die eigentliche Killerapplikation. Dazu mehr in einem anderen Kapitel.

1.3 Task ontology

Die Schwierigkeit bei der Implementierung eines Augmented Assembly System ist die Task Ontology. Gerade wenn die Aufgabe umfangreicher ist (dexterous Manipulation) benötigt man eine Anzahl von Motion Primitiven. Diese werden traditionell als Task ontology oder als Grammar bezeichnet. Es handelt sich um eine wiederkehrende Anzahl von Griffen die der menschliche Operator ausführt. Beispielsweise: Schraubenzieher benutzen um eine Schraube eindrehen, einen Gegenstand greifen, etwas ablegen, Handschuhe anziehen, etwas wegwischen usw.

An dieser Stelle ein kleiner Exkurs zum Moley Kitchen Robot. Laut der offiziellen Darstellung hat sich der Chefkoch Tim Anderson einen Datenhandschuh angezogen und damit wurde dann der Roboter trainiert. Richtig? So wird es doch verbreitet. Nun in Wahrheit erfolgte die Erstellung des Systems anders. Und zwar hat Moley Robotics ein Tutoring-System entwickelt. Das heißt, man hat dem Chefkoch Tim Anderson der wahrlich genug Ahnung hat, erzählt dass er jetzt die Herdplatte herunterregeln soll, jetzt den Inhatl in den Topf schüttet und jetzt umrührt. Was Tim Anderson in einer virtuellen Kochschule soll ist unklar, weil normalerweise ja das was für Leute ist die noch nicht wissen wie es geht. Aber sei es drum, so funktioniert nunmal die Erstellung eines Domänemodells. Das man also Software entwickelt, mit der Menschen lernen können wie man kocht. Es war also nicht so, dass die Hand des Kochexperten analysiert wurde um daraus ein Modell zu erstellen, sondern es war genau andersherum. Der Propand führt die Hand so, wie es das System möchte.

1.4 Aimbot Konstruktion für einen Cooking Simulator

Laut Youtube gibt es eine stattliche Anzahl an Cooking Games, also Simutationsspielen in denen man Hamburger braten kann oder Nudeln aufsetzen. Einige von ihnen sind sogar in 3D gehalten. Wie sie programmiert wurden ist simpel: als Basis wird eine Game-Engine genutzt, ontop noch eine Physik-Engine damit der Topf herunterfällt wenn man ihn loslässt und dann noch einige grafische Extras, so dass man die Tomaten zerschneiden kann. Soweit so gut, doch lassen sich diese Simulatoren als Basis für einen Aimbot verwenden? Und ob, ein Cooking Game ist im Grunde eine fertig modellierte Domäne. Es enthält eine maschinenlesbare Beschreibung wie der Kochvorgang abläuft, welche Zutaten benötigen werden und welche Aktionen sinnvoll sind. Man kann sich einen Cooking Simulator vorstellen wie eine PDDL Datei. Sie ordnet den Handlungsraum in abstrakte Zustände. Dazu ein kleiner Exkurs:

Normalerweise besitzt ein Roboter unendlich viele Optionen was er als nächstes tut. Er kann seinen linken Arm vorstrecken, bis er damit gegen den Kochlöffel stößt, er kann aber auch etwas ganz anderes machen. Will man alle Möglichkeiten durchprobieren würde das Millionen von Jahre dauern. Der Handlungsraum ist viel zu groß. Schon in den 1960'er wurde im Shakey Projekt das STRIPS Plannungssystem entwickelt. Danach besteht die Welt nicht einfach aus xyz Koordinanten sondern die Welt besteht aus Tasks, Gegenständen und Folgezustände. Genau dieser abstrakte Raum wird in einem cooking Simulator bereitgestellt. Die Software kennt bereits DInge wie Tomaten, Schneidbretter, Kochlöffel, und sie kennt Aktionen wie schäle Tomaten, oder fülle Wasser in den Topf. Der Handlungsraum den man in diesem Spiel ausführen kann besitzt also Sinn. Und an dieser Stelle kommen Aimbots ins Spiel Ein Aimbot hat die Aufgabe Dinge zu automatiseren. Es geht darum, dass man nicht selber kochen will, sondern dass man einen Zielzustand automatisiert ansteuern möchte. Der Zielzustand wäre beispielsweise dass der Tisch gedeckt ist, und sich dort erlesene Speisen befinden, die alle frisch zubereitet wurden. Für genau diese Aufgaben wurde STRIPS Planner entwickelt, sie probieren die möglichen Aktionen der Reihenfolge nach durch und bringen das System in den gewünschten Zustand. In einem Cooking Simulator ist die Anzahl der möglichen Aktionen überschaubar. Selbst in komplexen Simulationen hat die Küche nicht mehr als 100 Gegenstände und 30 Tasks die man mit diesen ausführen kann. Das lässt zwar viel Spielraum ist aber von einem Solver in weniger als 1 Sekunde als Graph aufgelistet und durchgesucht. Und genau darüber kann man einen Aimbot programmieren. Man nutzt das vorhandene Domänemodell um darüber einen PDDL artigen Solver zu steuern.

Man muss nicht zwingend einen Planner verwenden, will man das System vollautonom steuern. Man kann die Aktionsreihenfolge auch fest als Behavior Tree spezifizieren. solche Systeme sind weniger flexibel aber sie sind ebenfalls in der Lage zu kochen. Man kann sich das wie ein Makro vorstellen was abgearbeitet wird. Entweder spielt man das Spiel so, dass man manuell zuerst das Wasser heißmacht und dann die Nudel reingibt, oder man lässt einfach ein Script laufen, wo das selbe im Batchbetrieb durchgeführt wird. Der entscheidene Punkt ist, dass man das vorhandene Domänenmodell nutzt. Also an das Cooking Spiel einen Querry sendet um zu erfahren wo der Topf ist, wo das Wasser, und wie man einen Gegenstand aufnimmt. Das ist dort bereits implementiert, weil sonst das Spiel im manuellen Betrieb nicht funktionieren würde.

Für ein echtes Roboter-System muss man die Cooking Spiele lediglich um eine Augmented Reality Komponente erweitern, wo man also Gegenstände in der echten Welt verwendet, die getrackt werden mittels QR-Code-Marker. Und voila, man kann jetzt den menschlichen User durch einen Roboterarm ersetzen. Der Rest ist dann nur noch typisches Sfotware-engineering, also BugFixing bis die Software läuft. Und um das Anpassen an neue Aufgaben zu erleichtern.

1.5 Domain-Modelle auf einem Roboter ausführen

Im vorherigen Kapitel klang es bereits an die Zielstellung: es geht darum, Augmented Assembly Domain-Modelle von einem Roboter ausführen zu lassen. Zunächst ist zu klären was genau ein Domain-

Modell eigentlich ist. In der PDDL Community wird darunter eine PDDL Datei verstanden. Etwas allgemeiner handelt es sich um eine API mit der man auf vorhandene Methoden und Variablen zugreifen kann. Vielleicht ein Beispiel: Angenommen man hat ein Augmented Assembly Projekt durchgeführt bei dem Bauteile aus einem Regal nimmt um damit ein Lego Flugzeug zusammenzubauen. Die programmierte Augmented Reality Software kann in Echtzeit anzeigen, wo das jeweilige Bauteil gerade ist. Es gibt also in der Software eine Variable "objectpos" wo die Koordinaten hinterlegt sind. Diese Werte werden normalerweise verwendet um auf dem User-Display das Bauteil einzukreisen, man kann die Werte aber auch als Zahlenwerte ausgeben. Ferner wurde im Augmented Reality Projekt ein Task definiert, wonach "nimmBauteil" bedeutet, dass man mit der Hand ins Regal greift. Hinter dieser Methode verbirgt sich eine Trajektorie, die dynamisch berechnet wird.

Mit diesen Werte lässt sich nun ein Roboter steuern. Man sendet an den Roboterarm den Befehl "nimmBauteil(objectpos)" und schon führt der Roboter und nicht der Mensch den Task aus. Ok ganz so simpel ist es nicht, man muss noch einige Modfikationen vornehmen, aber im Prinzip wird es so gemacht. Je genauer das Domänenmodell bereits ist, desto leichter ist es für den Roboterprogrammierer daraus Steueranweisungen zu erzeugen. Im Optimalfall kann man das Augmented Assembly Modell 1:1 auf die ROS Middleware mappen und schon hat man ein vollautonomes System. ROS bringt dabei grundlegende Funktionen bereits mit, die auf High-Level-Ebene durch das Domänenmodell vervollständigt werden.

Zur Realisierung ist es ausreichend, wenn man das Head-up Display um eine textuelle API erweitert. Also nicht die Informationen nur grafisch ausgibt, sondern noch eine Programmierschnittstelle schafft um direkt auf die Informationen zuzugreifen. Man nutzt das Head-up Display als eine Art von KI-Engine. Diese Engine stellt Variablen und Methoden bereit und aktualisiert diese in Echtzeit. Vielleicht noch ein anderes Beispiel.

In einem Kochspiel ist in aller Regel definiert, wo sich der Herd befindet. Es gibt irgendwo ein Objekt herd, was eine Position besitzt. Diese Wert wird normalerweise verwendet um das Spiel auf dem Bildschirm anzuzeigen. Genausogut kann man den Wert aber auch nutzen um darauf aufbauend ein Script zu schreiben. Also einen Behavior Tree. Dieser könnte beispielsweise den Topf auf den Herd stellen. Normalerweise ist das eine Aufgabe, die extrem schwer zu programmieren ist. Selbst Robotersprachen wie VAL sind nicht im Stande diesen Ausdruck auswerten. Mit der oben genannte KI-Engine Kochstudio geht das jedoch spielend. Die Koordinaten sind bekannt, vielleicht gibt es sogar einen Trajektoriegenerator für die Animation und das nutzt man einfach in einem Script. Anders formuliert, für ein vorhandenes Head-up Display ein Script zu programmieren was sinnvolle Aktionen ausführt ist sehr leicht. Man nutzt das vorhandene Domänenwissen und programmiert den Aimbot gegen die API des Kochspiels.

2 Programmierung eines Head-up Displays

2.1 SIch langsam steigern

Der Ablauf um ein Head-up Display zu programmieren sollte unterteilt werden. Am besten fängt man mit dem eigentlichen Spiel an. Das kann ein Autorennspiel sein, was man in Python programmiert. Oberhalb von diesem Spiel werden nun mehrere Lerneinheiten programmiert. Im Tutorial 1 soll der Spieler lernen einem Pfad zu folgen. In Tutorial 2 Hindernissen ausweichen, bei Schritt 3 die Ampel zu respektieren und bei Schritt 4 Vorfahrt gewähren. Jede dieser Lektionen implementiert man in Python und erweitert so stückweise das Head-up Display. Am Ende kombiniert man die Elemente zu einer großen Übung. Der Spieler steuert das Auto und fährt über die Map, und live dazu trackt das Head-up Display den Spielfortschritt und gibt Feedback wie ein Fahrlehrer. Wenn das alles sauber funktioniert kommt die eigentliche KI an die Reihe. Jetzt nutzt man das vorhandene Head-up Display um als weiteren Layer einen Behavior Tree zu programmieren, also eine Software die autonom agiert. Fertig, das ist eigentlich der komplette Ablauf um ein selbstfahrendes Auto zu programmieren. Klingt gar nicht so schwer, oder?

Der Rest ist nur eine Fleißsache, also wieviel Lines of Code man täglich dazuprogrammiert, wie gut man das Spiel, die Tutorials und den Behavior Tree testet. Die Frage welche noch offen ist lautet was man nutzen kann um sich die Aufgabe zu erleichtern. Python als Programmiersprache wurde schon genannt, selbstverständlich sind Game-Engines und Physik-Engines empfehlenswert. Leider gibt es derzeit keine Head-up Display Engine oder eine Serios Game Engine, wo also Lektionen die man im Spiel unterrichten möchte schon fertig vorprogrammiert sind. Und es ist auch unwahrscheinlich, dass jemand sowas programmieren wird, weil das stark vom jeweiligen Spiel abhängig ist. In einer Autosimulation besteht das Lernziehl darin, einer Trajektorie zu folgen, während bei einer Kochsimulation man häufig Pick&Place Aufgaben wahrnimmt. Die Baseline auf die man zurückgeworfen wird, ist eine Hochsprache wie Python und vielleicht die Wikihow-Anleitungen wo für Menschen erläutert wird, wie man bestimmte Aufgaben löst. Grob kann man sich daran orientieren, wenn man tutorials erstellt. Beispielsweise gibt es hier [1] eine Anleitung wie man ein GPS nutzt, sowas könnte man einbauen in ein Serios Game.

2.2 Teleoperation

Nehmen wir mal an, ein Roboterarm soll mit einem Head-up Display erweitert werden, wie geht man bei der Programmierung konkret vor? Zunächst einmal gilt es die gängigen Ansätze der Künstlichen Intelligenz Robotik über Bord zu werfen und den Roboterarm wie ein ferngesteuertes Auto zu betrachten. Das heißt, man benötigt einen Joystick, einen Datenhandschuh, eine Maus oder was auch immer. Das wird für die nächste Zeit das primäre Steuerungsinstrument sein, man bewegt den Joystick und Parallel dazu bewegt sich der Roboter. Sehr simpel wie effektiv gleichermaßen.

Das ganze ist aus Sicht der Robotik natürlich unbefriedigend, weil es ja nicht angeht, dass man einen Human-Operator benötigt nur damit der Roboterarm Obst vom Tisch aufnimmt. Deshalb besteht der nächste Schritt darin ein Head-up Display zu programmieren. Das besteht im einfachsten Fall aus einer Objekterkennung, das heißt, es wird neben dem Objekt und neben dem Robotergreifer jeweils ein kleines Schild auf dem Display eingeblendet, als Zeichen dass die Position korrekt getrackt wurde. Wenn man jetzt den Roboter bewegt, verschiebt sich auch der Marker. Im nächsten Schritt baut man eine Pick&Place Funktion ein. Keine richtige zur automatischen Steuerung sondern ebenfalls nur angedeutet. Damit ist gemeint, dass ausgehend von der Ist-Position des Apfels zur Sollposition auf dem Tisch ein kleiner Pfeil eingezeichnet wird. Der Bediener sieht so anhand des Bildschirms was zu tun ist. Der sieht die Zielposition und steuert daraufhin den Roboter so, dass die Aufgabe erfüllt wird.

Diese Funktionalität erweitert man Stück für Stück. Im nächsten Schritt beispielsweise durch eine Kollisionserkennung, dass also auf dem Bildschirm auch Hindernisse erkannt werden und eine Ausweichroute festgelegt wird. Die hätte der Human-Operator zwar ohnehin gewählt, weil er während der Steuerung mitdenkt, wenn sie aber bereits im HuD erwähnt wird kann es nicht schaden. Noch eine Stufe weiter geht das Spiel, wenn das HuD sogar die Greifpunkte des Objektes einzeichnet. Also nicht nur sagt: nimmt den Apfel und leg ihn dort ab, sondern "nimmt den Apfel an dieser Griffpunkten und zwar

[1] http://www.wikihow.com/Use-a-GPS

mit Daumen und Zeigefinger". Der Human-Operator leistet dem natürlich folge und bedient den Datenhandschuh exakt so, wie es auf dem Dsiplay zu sehen ist. Aller Vorssicht nach wird das HuD Schwächen besitzen. Das heißt, die Griffpunkte werden nur in 80% der Fälle korrekt angezeigt in den anderen Fällen muss der Bediener einen anderen Greifpunkt wählen um das Objekt nicht zu verlieren. Diese Fälle zeigen auf, wo das Modell noch schwächen hat. Das heißt, im HuD sind Bugs enthalten.

Trotz dieser Probleme bleibt das System einsatzfähig. Nach wie vor wird die Steuerung des Roboterarmes manuell ausgeführt, das System bleibt also nicht stehen, nur weil der Greifpunkt falsch berechnet wurde. Sondern der Übergang zwischen manueller und automatischer Steuerung ist ein kontinuierlicher Prozess, das Ziel besteht darin, die kognitive Load für den Nutzer zu senken.

Nehmen wir mal an, das Problem mit den fehlerhaften Greifpunkten lässt sich nicht beheben, was dann? Nun, dann muss das Domänenmodell erweitert werden. Und zwar dahingehend, dass schwierige Fölle erkannt werden und eine Gegenstrategie erarbeitet wird, so dass der User beispielsweise aufgefordert wird, das Objekt zunächst freizustellen weil erst dann der optimale Griffpunkt berechnet werden kann. Dadurch steigt die Komplexität. Das HuD wird der Wirklichkeit besser gerechet und emuliert stärker wie Menschen die Aufgabe lösen würden. Technisch gesehen führt das dazu, dass die Anzahl an Codezeilen des Head-up Displays ansteigt. Es werden mehr Variablen und mehr Klassen benötigt. Das System kennt nicht nur, move und pickup sondern viele weitere Motion Primitive wie pregrasp, roll und tilt.

Die Mensch-Maschine-Schnittstelle bleibt dabei erstaunlicherweise konstant. Der Human-Operator hat den Datenhandschuh auf und folgt den Answeisungen des Lernprogramms. Wenn die Anweisungen des HuD mißverständlich oder gar falsch sind, sagt er dies dem Programmierer damit er das Domain-Modell anpassen kann.

2.3 Augmented telerobotics

Obwohl Head-up Display und Augmented Assembly schonmal in die richtige Richtung gehen wird damit noch nicht exakt beschrieben um was es geht. Weil ja das Ziel nicht darin besteht, die Produktivität von Menschen zu erhöheh, sondern Head-up Displays sind nur ein Mittel zum Zweck um autonome Systeme zu entwickeln. Der Begriff worunter das in der Literatur subsummiert wird lautet "Augmented Telerobotics".[31] Ein Konzept bei dem ein Mensch einen Roboter mittels Geomagic Phantom haptic device steuert und dabei von einem Head-up Display unterstützt wird. Man kann sich das vorstellen wie eine Fernsteuerung, nur dass man nebem dem Videosignal noch weitere Informationen erhält, ähnlich wie beim Head-up Display in einem Auto.

Heuristiken Ein Head-up Display (HuD) lässt sich für nahezu jede Aufgabe einsetzen. Die Verallgemeinerung lautet, dass ein HuD die Programmierung einer domänenspezifischen Heurstik erleichtert. Man muss dazu wissen, dass bei jeder Domäne wie Sokoban, Soccer usw. jeweils unterschiedliche Heuristiken benötigt werden. Ein Heuristik ist dabei als Computersourcecode formuliert. Einmal wird darüber ein Pfad zum Ziel geplant, ein anderes Mal das Passspiel unterstützt. Manchmal sind zur Implementierung mathematische Gleichungen nötig, manchmal arbeiten Heuristiken datenbasiert. Bis heute wurde übrigens keine praktisch einsetzbare Metaheuristik gefunden. Im Bereich des Maschine-Learning forscht man daran zwar schon seit Jahrzehnten, ohne Erfolg übrigens. Benötigt man einen Pathplanner, ist noch immer Handarbeit angesagt. Die einzige Vereinfachung besteht darin, bereits erstellte Python Libraries zu nutzen.

Domäne	Heuristik
Sokoban	Pathplanning
Inverted Pedulum	Differential equation
Moorhuhnjagd	Aiming
Soccer	Teamplaying
biped Walking	Zero moment point

Tabelle 1: Head-up Display Heuristik

Generell lässt sich ein HuD als Ablaufumgebungen für Heuristiken definieren. In den 600x400 Pixeln wird das Resultat einer Faustregel angezeigt, das kann eine Trajektorie sein, oder auch die Freund/Feind-Erkennung. Die Funktionsweise dieser Heurstiken werden in Informatik-Lehrbüchern erläutert. Wirklich kompliziert ist davon nichts, allerdings gilt es mehrere Heurstiken zu kombinieren bis das HuD einsatzbereit ist.

Das Programmieren eines HuD bedeutet, domänenspezifische Heuristiken zu identifzieren und in das HuD einzuprogrammieren. So dass sich in Echtzeit ausgeführt werden und der Human-Operator einen Vorteil erzielt. Bei einem Laufroboter der durch ein HuD aufgewertet wird, kommt eine gänzlich andere Heurstik zum Einsatz, als bei einem Echtzeitstrategiespiel. Die Gemeinsamkeit besteht lediglich darin, dass in beiden Fällen das HuD als Overlay über das ursprüngliche Bild drübergelegt wird, um die kognitive Load des Bedieners zu senken.

Bei sehr einfachen Domäne wie Moorhuhnjagd ist die Heuristik geradezu trivial. Es reicht aus, das Fadenkreuz auf das Zielobjekt zu bringen. Die Heuristik besteht darin, dass man den Winkel zwischen Ist und Soll-Koordinaten ermittelt und dann das Fadenkreuz dahinbewegt. Um diese Heuristik zu implementieren reichen 5 Lines of Code aus. Bei anderen Spielen sind die Heuristiken komplexer. Das macht die Programmierung eines HuD anspruchsvoller.

Metaalgorithmen Wer sich optimistisch auf die Suche begibt nach Metaalgorithmen wird enttäuscht feststellen, dass es sowas nicht gibt. In diesem Zusammenhang werden zwar häufig machine Learning und Ant colonoy optimization genannt, doch von einem Meta-Algorithmus sind derartige Verfahren weit entfernt, Wenn überhaupt funktionieren sie nur auf unendlich schnellen Quantencomputern. Was hingegen der Idee eines Meta-Algorithmus sehr viel näher kommt sind Algorithm libraries, gemeint sind Softwarebibliotheken wie numpy, glibc oder OpenCV. Diese Bibliotheken lassen sich nutzen um auf die schnelle ein Problem zu lösen. Auch Chess-Engines kann man in diese Kategorie einordnen, weil man deren Hilfe man in weniger als 10 Lines of Code ein Schachprogramm konstruieren kann.

Der Trick um die Programmierung von Head-up Displays zu erleichtern besteht darin, solche fertigen Librarys ausfindig zu machen und miteinander zu kombinieren. Und wo es noch keine git, müssen neue geschrieben werden.

2.4 Komplexe Software-Systeme erstellen

Im akademischen Software-Engineering hat sich eine sehr merkwürdige Haltung verfestigt. Üblichweise sieht man sich der Anforderung ausgesetzt, komplexe Software zu entwickeln, üblicherweise im Bereich Robotik und Künstliche Intelligenz. Als Antwort darauf hat man sich zwei Dinge überlegt: modellgetriebene Softwareentwicklung und Meta-Algorithmen. Letztere werden nochmal unterteilt in Machine Learning, Neuronale Netze und Genetische Algorithmen. Darüber hofft man gut gerüstet für die Zukunft zu sein.

Schaut man sich diese Technologie etwas genauer an, wird man feststellen, dass sie nicht funktionieren. Zuerst glaubt man, es liege an

einem selber, also dass man noch nicht richtig verstanden hätte was Reinforcement Learning bedeutet. Doch objektiv betrachtet stimmt etwas mit diesen Software-Entwicklungsparadigmen nicht. Was das ist kann man einem Beispiel erläutern.

Nehmen wir mal keine Robotersoftware sondern etwas einfacheres: ein Betriebssystem wie Fedora. Wenn man damit einen PC startet ist nach ca. 20 Sekunden die GUI zu sehen, wo man mittels Maus weitere Anwendungen starten kann. Von außen betrachtet ist also Fedora etwas simples: nachdem man den Computer mit Strom versorgt hat, lädt er das Programm in den Arbeitsspeicher und zeichnet dann auf den Bildschirm einige Rechtecke. Eigentlich ein sehr simple Sache. So ein Programm kann man vermutlich mit Machine Learning und ein wenig herumprobieren in 1 Tag entwickeln. Weil, eigentlich macht Fedora ja nichts weiter als die Mausposition abzufragen, und wenn man irgendwo draufklickt passiert etwas. Das lässt sich wunderbar als selbst-lernendes Metamodell implementieren. Leider geht es so nicht. Schaut man sich etwas genauer an, was Fedora ist, wird man feststellen, dass sich dahinter nicht nur Gigabyteweise an Sourcecode verbergen, sondern dass es eine Entwicklungsgeschichte gibt, die 50 Jahre zurückreicht und angefangen hat mit UNIX und dann weitere Iterationen in der kommerziellen Softwareentwicklung und den UNIX Wars hatte, später kam dann noch Linus Torvalds hinzu. Anders formuliert, Fedora ist nicht nur ein Metaalgorithmus, sondern es ist vor allem ein soziologischer Prozess, der aus handelnden Personen, Firmenaufkäufen, Medialer Aufmerksamkeit und manueller Programmierung besteht.

Nach diesem kleinen Exkurs zurück zur Robotik. Irgendwann wird es sie sie geben: eine Software um Roboter zu steuern. Aber sie wird nicht mit Modellierungstools oder Machine Learning entwickelt, sondern sie wird ähnlich entstehen wie auch heutige Betriebssysteme. Als soziologischer Prozess der eine Geschichte besitzt, handelnde Personen und ökonomische Auswirkungen.

Zoomt man aus Computersoftware nur weit genug heraus, findet sich irgendwann Programmierer, Hacker, Genies und Firmen die rund um Softwareentwicklung aktiv sind. Der Überbau um Software zu erstellen ist nicht etwas objektorientierte Programmierung und auch keine OWL Ontologien, sondern es handelt sich um soziale Interaktionsformen. Sie wird ausgedrückt in Computermessen, Fachkonferenzen, Wettbewerben und investierter Manpower. Genauer formuliert wird jedes Stück Software, egal ob es ein Betriebssystem oder ein Robot-Control-System von Menschen programmiert. Software fällt nicht vom Himmel und wird auch nicht durch andere Software generiert, sondern Software ist Ergebnis eines kreativen Schaffensprozess der aus der Gesellschaft heraus initiiert wird.

Die Erstellung von Software lässt sich als Technik-Geschichte analyisieren. Es gibt dort eine dezidierte Geschichte des Programmierens. Also eine Zeitleiste, handelnde Personen und wichtige Ereignisse welche zur heute verfügbaren Software geführt haben. Das besondere an der Geschichte des Programmierens ist, dass es einen kurzen Zeitraum umfasst. Die Hochzeit war von 1970-1990, das war die Zeit wo sehr viel Anstrengungen unternommen wurden, und die großen Konzepte wie Programmiersprachen, Compiler und Betriebssysteme entwickelte wurden. Seit den 1990'er war Programmieren als eigenständige Displizin anerkannt und die Verfahren wurden erweitert. In der Geschichte ist es also nur eine sehr kurze Epoche. Viele Jahrhunderte lang, haben Menschen überhaupt nicht programmiert.

Eine Besonderheit der 1970'er Jahre war, dass man verstärkt auf Training Languages wie Pascal gesetzt hat. Aus technischer Sicht macht diese Sprache keinen Sinn, es bremst den Computer aus. Sondern Pascal hat die Aufgabe, möglichst viele Menschen an das Programmieren heranzuführen. Also den Kreis derjenigen Personen zu erhöhen, die Coden können. Etwas ähnliches war ab den 2000'er mit Lego Mindstorms zu beobachten. Auch diese Technologie ist nicht so sehr State-of-the-art Robotics, sondern es geht primär um Education. Der Masterplan um Robotik-Software zu entwickeln besteht nicht darin, Maschinen schlauer zu machen sondern Menschen.

Fahrerassistenzsysteme Schauen wir nun konkret, wie eine komplexe Software das Autofahren unterstützt. Zu sogenannten Fahrerassistenzsystem hat Wikipedia einen gut gemachten Artikel [2] Auffällig daran, dass es nicht das eine Assistenzsystem gibt, was sich mit Hilfe einer mathematischen Theorie implementieren lässt, sondern vielmehr gilt es die Thematik geisteswissenschaftlich aufzubereiten. Das heißt, die Submodule eines solches Systems sind ausgedacht und historisch gewachen. Wikipedia nennt beipspielsweise:

- Ampelassistent
- ABS
- Bergfahrhilfe
- Einparkhilfe
- Geschwindigkeitsassistent

Die Liste ist noch sehr viel länger, Stolze 41 Einträge sind in der Tabelle enthalten, Tendenz steigend. All diese Module behandeln ein spezielles Thema, wenn man sie ordnen wollte, dann nach semantischen Aspekten, vielleicht so wie man Literatur oder Computerspiele untergliedert. Ich will damit sagen, dass der Komplex Fahrassistent sich nicht mathematisch oder naturwissenschaftlich fassen lässt, sondern unter technikhistorischen Dimensionen zu betrachten ist.

Warum ich darauf hinweise hat den simplen Grund dass in der Mathematik und Robotik bis heute das Vorurteil herscht, dass obiges Softwaresystem etwas mit PID Control zu tun hätte, wo man also Kostenfunktion aufstellt die von einem Regler minimiert wird. Genau damit wird man dem Problem nicht gerecht. Es ist trivial einen Bremsassistenten als PID Regler zu bezeichnen, genauso könnte man sagen dass es eine Künstliche Intelligenz ist. Das ist zwar richtig, hilft aber nicht bei der Entwicklung eines derartigen Systems.

Der Überbau um solche komplexen Systeme zu beschreiben und weiterzuentwickeln orientiert sich vielmehr an der Arbeitsweise in den Geistes- und Sozialwissenschaften. Das heißt, es wird mit Quellen gearbeitet. Einen Bremsassistenten lässt sich am besten dadurch definieren, indem man ihn geschichtlich einordnet und auf Veröffentlichungen verweist, die damit in Beziehung stehen. Es handelt sich also nicht um mathematisch/analystisches Vorgehen, sondern es geht um Textarbeit.

Das man die Erstellung eines Fahrassistenzsystems derart aufwendig vorantreibt und weitaus mehr bedarf, als eine hastig hingeschriebene Optimierungsgleichung mag überraschen. Weil naiv betrachtet, das Autofahren sehr übersichtlich ist. Im Grunde muss der Fahrer nur zwei Parameter einstellen: Geschwindigkeit und Richtung. Demzufolge lautet die Gleichung zum Autofahren wie folgt:

$$Steuerung = Geschwindigkeit + Richtung$$

Die obige Formel ist richtig wie nutzlos gleichermaßen, weil genau so die Steuerung eines Autos eben nicht erfolgt. Selbst wenn man noch ein Differentialzeichen mit aufnimmt wird man damit die Aufgabe nicht beschreiben können.

[2] https://de.wikipedia.org/wiki/Fahrerassistenzsystem

2.5 Verallgemeinerung möglich?

Oberste Pflicht eines jeden Programmierers ist es, möglichst effizient ein Problem zu lösen. Im Bereich Head-up Display ist das jedoch schwierig. Fragen wir dochmal github was bereits für Code aus dieser Richtung geschrieben wurde. Es gibt eine Menge Projekte, die Head-up Displays implementieren. Angefangen von einer Android App, die mit dem Auto verbunden die Geschwindigkeit anzeigt über ein Addon für World of Warcraft was als HuD ein besseres User-Interface verspricht bis hin zu kleineren Projekten gibt es jede Menge Code. Das Problem ist nur, dass man nichts davon weiternutzen kann. Es steht zwar zur Einsicht im Internet aber kompatibel mit den eigenen Bedürfnissen ist es nicht.

Das Grundproblem mit Head-up Displays liegt darin, dass sie sehr speziell auf die jeweilige Domäne hinprogrammiert wurden. Vergleichbar mit einem Behavior Tree. Die meisten Head-up Displays wurden in Quick&Dirty in einer Scriptsprache programmiert und lassen sich genau 1x verwenden. Einen Vorwurf kann man den Programmierern nicht machen, sondern es gibt offenbar keine Alternative dazu.

Ein wenig besser sind da schon wissenschaftliche Paper die ein Head-up Display für eine Anwendung beschrieben. Solche Paper lassen sich schon eher auf die eigene Domäne übertragen. Man kann daraus ablesen, welche Funktionen grundsätzlich benötigt werden, und wie man grob das Programmieren erledigt. Wenn dann noch Screenshots mit in dem Paper abgedruckt sind umso besser. Solche Arbeiten lassen sich relativ leicht reproduzieren.

Ich glaube, dass die beste Methode wie man ein Head-up Display kopieren kann besteht darin, eines in Aktion zu sehen. Wenn man möglichst das Echtzeitverhalten sieht, wie ein Abstandswarner funktioniert, kiann man diese Funktionalität in eigenen Sourcecode übertragen. Die Schwierigkeit besteht derzeit darin, dass es nur wenig konkrete Beispiele gibt, und noch seltener finden sich Screenshots.

Lines of Code Softwarekomplexität wird von der Industrie in Lines of Code gemessen. Ein simpler aber aussagekräftiger Maßstab. Auf mehreren Folien war zu lesen, dass im Bereich Embedded Car Software und Driver Assistive System mit einer Codegröße von 100 Million Lines of Code zu rechnen ist. Diese Größe könnte ungefähr hinkommen. Leider ist bis heute nicht klar, wie man solche umfangreichen Projekte zu geringen Kosten erstellt. Es werden zwar Konzepte diskutiert wie objektorientierte Programmierung, Codegeneratoren und Versionsverwaltung aber vermutlich wird der wichtigste Treiber immernoch der menschliche Fleiß sein.

Auf [3] gibt es eine Übersicht über große Softwareprojekte und deren Codeumfang. Dort ist beispielswelse in Lines of Code aufgeführt der Umfang von Debian 5.0, Mac OS X, Facebook, Mozilla, MySQL und viele andere Projekte. Schaut man sichmal die absolute Zahl an, so ist es schon erstaunlich über welche Dimensionen wir hier reden. Offenbar gibt es seit ungefähr den 1980'er Jahren (älter sind die meisten Softwareprojekete nicht) große 'Anstrengungen die zumindest nominell zu umfangreichen Codebasen geführt haben. Es ist wahrscheinlich, dass dies in der kommenden Robotik-Revolution nach demselben Schema weitergehen wird. Wenn der Roboter nicht intelligent genug ist, wird einfach der Codeumfang verdoppelt.

Das ist deshalb erwähnenswert, weil in der Standardliteratur über selbstfahrende Autos und Künstliche Intelligenz das Thema "Lines of Code" praktisch nie zur Sprache kommt. Stattdessen werden Konzepte wie DeepLearning, philosophische Betrachtungen und PDDL Solver diskutiert. Man kann daraus ableiten, dass vermutlich 90% aller Paper die über Robotik und Fahrerassistenzsysteme veröffentlicht werden, irreführend oder sogar bewusst falsch sind. Wir haben es also mit Autofirmen zu tun, die in der Praxis umfangreiche Codebasen manuell erstellen dann aber in ihren Papern von DeepLearning schwadronieren, also von sich selbst programmierenden Systemen.

Zum Vergleich: Starcraft AI Bots die in Wettbewerben antreten haben rund 12000 Lines of Code. Vermutlich gilt hier: je mehr Lines of Code, desto spielstärker. Der Flaschenhals dürfte darin bestehen, eine derartige Codemenge zu erzeugen und zu debuggen, die nichts weiter macht als Starcraft zu spielen. Wenn man ein HuD für ein Auto benötigt, was eine Einparkhilfe, einen Bremsassistenten und einen Fahrspur-Assistenten beinhaltet, dürfte die benötigte Lines of Code Menge stark ansteigen. Gleichzeitig besitzt die Informatik bis heute jedoch keine Technologie um Code automatisch zu erzeugen, so dass es darauf hinausläuft, dass wie im Mittelalter Mönche damit beauftragt werden, Code manuell zu übertragen. Anders gesagt, man muss sich die Ist-Situaation so vorstellen, dass überall auf der Welt in Klosterähnlichen Programmierstuben gute wie schlechte Programmierer tag und nacht nichts anderes zu tun als Computercode für selbstfahrende Autos, Roboter, Dronen oder was auch immer zu erstellen. Wieviel genau kann man nur ahnen, aber es dürfte eine Menge Zeit investiert werden. Der meiste erstellte Code wird niemals veröffentlicht sondern verbleibt im Unternehmen und wird dort gehütet wie ein Schatz. Siehe das Beispiel Google von dem bekannt ist, dass sehr viel Code existiert aber nicht bekannt ist, was das für Code ist.

Anhand der bereits realisierten Softwareprojekte kann man ableiten, dass es für Firmen relativ leicht ist, in kurzer Zeit große Codebasen anzulegen. Wenn man 100 Programmierer, die jeder 10000 Lines of Code zu einem Projekt beisteuern, kommt man bereits auf die stattliche Anzahl von 1 Mio Lines of Code. Selbst wenn man den erstellten Code komplett löscht und from scratch beginnt, hat man in kurzer Zeit wieder einen ähnlichen Umfang erzielt. Und hier findet sich auch die Antwort, warum die Softwareprojekte so groß geworden sind und was das Geheimnis ist um in kurzer Zeit viel Code zu schreiben. Die Anzahl der Programmierer ist am steigen. Allein in den USA gibt es viele tausend Programmierern, in Ländern wie China sind es noch mehr. Die letzte halbwegs seriöse Schätzung beläuft sich auf 21 Millionen Programmierer weltweit. Davon allein 4 Millionen in den USA. Tendenz steigend. Und diese Anzahl ist das Geheimnis hinter den vielen Lines of Code. Wenn die Programmierer nur halbwegs im Team arbeiten kann man in kurzer Zeit große Mengen an Software erzeugen. Nicht über Generatoren oder ähnliche Zukunftstechnologie sondern über simple Manpower.

Github ist nur die Spitze des Eisberges. Es ist die Codemenge die die Leute aus Langeweile und nebenbei erzeugen und wofür sie keine andere Verwendung haben, als es ins Netz zu stellen. In den Firmen selber wird noch sehr viel mehr programmiert. Anders als bei Büchern, gibt es noch nichtmal einen Katalog der auflistet, welche Firma wieviel Sourcecode erstellt hat.

Machen wir eine kleine Rechnung: wenn jeder Programmierer pro Jahr nur 1000 Lines of Code erstellt (was nicht besonders viel ist), würde das bedeuten, dass die 21 Mio weltweiten Programmierer jedes Jahr 21 Milliarden Lines of Code neu erstellen. Das heißt, selbst wenn man nochmal alles neu schreiben würde, was bisher erstellt wurde, hätte man nach kurzer Zeit wieder die Ausgangsmenge erreicht. Wenn also Apple über Nacht den Sourcecode seines kompletten Mac OS X Systems verlieren würde, wäre es im Bereich des machenbaren den Code from Scratch in 1-2 Jahren neu zu schreiben. Genau dies ist die eigentliche Technologie welche dafür sorgen wird, dass die Robotik-Revolution eintritt. Nicht neue mathematische Theorien oder KI-Algorithmen werden der Antrieb sein, sondern es wird die Anzahl der weltweiten Programmierer multipliziert mit ihrem jährliches Output an Codezeilen sein, der dafür sorgt, dass Skynet Wirklichkeit wird.

Dass ein einzelner durchgeknallter Wissenschaftler Skynet in Be-

[3] http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

trieb nimmt ist ausgeschlossen, selbst wenn er Tag und Nacht durcharbeitet ist sein jährlicher Output an Sourcecode begrenzt. Wenn jedoch eine ganze Generation von machtversessenenen Programmierer zusammenarbeitet könnte es zu Human-Level-AI kommen. Also in Software codiertes Wissen, was alles kann.

2.6 Roboterwettbewerbe anstatt Domain-Modelling Languages

Will man ein komplexes Softwaresystem programmieren benötigt man eine Domain-Modelling-Language. Das sind Sprachen wie UML wo man auf einem hohen Abstraktionsgrad arbeitet. Leider hat UML einen großen Nachteil: es besitzt gegenüber einer Sprache wie Python keinen Vorteil. Man kann zwar in UML wunderbar ein Objektmodell formulieren, das selbe lässt sich aber auch mit Python, Java oder C# tun. Die bessere Alternative um den Abstraktionsgrad zu erhöhen sind Techniken die außerhalb der Kerninformatik liegen. Genauer gesagt ist es nicht möglich, eine Computersprache zu entwickeln mit der man abstrakt Probleme lösen kann, sondern was man tun muss ist es, Programmierer in einen sozialen Kontext zu bringen. Der Robocup Wettbewerb ist dafür ein gutes Beispiel. Dort lernen Programmierer das Modellieren von komplexer Software.

Bei Robotikwettbewerben wird normalerweise ein formales Regelsystem vorgegeben, innerhalb derer man Punkte sammeln kann. Es sind die Spielregeln, was ein Team gewinnt und wo definiert ist, wie groß der Roboter sein darf und auf welcher API die KI-Engine aufsetzen soll. Solche Regelsysteme sind es, gegen die man programmieren kann. Sie bilden das eigentliche Metamodell.

2.7 Synthetic Vision

Wer die Geschichte der Computeranimation beleuchtet wird erkennen, dass ab den 1980'er Craig Reynolds und andere ein Konzept vorgestellt haben, was als Behavior Animation bekannt ist und was mittels objektorientierter Programmierung realisiert wurde. Anfangs wurde selbstverständlich objektorientiertes LISP verwendet, weil das die Mainstream-Programmiersprache in den 1980'er Jahren war (Stichwort Lisp-Machines), genausogut kann man Schwärme aber auch in Python realisieren. Die Frage ist: was kommt danach? Eine Antwort darauf gibt [26] und stellt das Konzept des Synthetic Vision vor:

> "The originality of our approach is the use of a synthetic vision as a main information channel between the environment and the digital actor."

Synthetic Vision wird also als Entwicklungsmethode verstanden um auf sehr allgemeine Weise eine Schnittstelle zwischen einem Actor und seiner Umgebung zu schaffen. Der Begriff Actor deutet bereits darauf hin, dass es anders als noch bei Reynolds nicht nur um Fische oder Vögel geht, sondern humanoide Avatare gemeint sind, die es zu animieren gilt. Leider wird der Text im weiteren Verlauf etwas unscharf. Vermutlich soll das momente Synthetic Vision Abbild in einem Kurzzeitgedächtnis gespeichert werden (Visual Memory). Die Frage ist jedoch, ob es Sinn macht, Begriffe aus der Psychologie für ein Programmierproblem zweckzuentfremden. Meiner Meinung nach liegt die Schwäche darin, dass nicht herausgearbeitet wurde was ein teilautonomes bzw. ein vollautonomes System ist. Der Vorteil von Head-up Displays und Synthetic Vision besteht darin, dass selbst wenn das Programm nicht funktioniert, man damit trotzdem sinnvolle DInge machen kann, weil der menschliche Bediener das erzeugte Signal interpretiert. Die Frage lautet demzufolge, wie man eine möglichst aussagekräftige Synthetic Vision Umgebung programmiert, damit ein Human-Operator darauf aufbauend richtige Entscheidungen treffen kann.

Immerhin geht der Ansatz in die richtige Richtung. In einem anderen Paper wird erläutert wie man Synthetic Actors konstruiert. [27] Konkret geht es darum, dass zuerst ein Synthetic Vision Display erzeugt wird (ein Head-up Display) und darauf aufbauend dann ein autonomer Charakter der Entscheidungen trifft um Tennis zu spielen. In der Arbeit werden "behavioral L-system" erwähnt, was das ist weiß ich nicht. Meiner Meinung nach ist es wie der ebenfalls erwähnte "Behavior Stack" ein Hilfsmittel des objektorientierten Designs. Wie also der fertige Sourcecode entwickelt wird. Die Diagramme deuten daraufhin, dass objektorientierte Programmierung verwendet wurde, um das System zu entwerfen. Laut Wikipedia handelt es sich bei L-Systemen um eine mathematische Theorie aus den 1960'er die sich mit Rekursion und Fraktalen beschäftigt. Also nichts, was mächtiger wäre als eine Programmiersprache wie C++ sondern vermutlich nur dem Autor vertraut war und er sich davon hat inspirieren lassen bei der Programmierung seines Synthetic Actors.

Viel entscheidener dürfte das Konzept des Synthetic Actors sein. Es handelt sich um eine Mischung aus Head-up Display mit der man die Umgebung wahrnimmt plus einer Behavior Engine um die Bewegungen zu steuern. Es ist nicht so sehr ein konkreter Algorithmus, sondern vielmehr ein Ziel, wie die fertige Software einmal aussehen soll. Das heißt, wenn man in C++ einen Synthetic Actor programmiert erstellt man im Schritt 1 die Synthetic Vision Komponente und erstellt dann weitere Klassen um die Motion Primitive des Actors zu definieren, mit dem Ziel am Ende einen Non-Player-Charakter also eine Künstliche Intelligenz zu erzeugen.

Synthetic Actor Was das besondere an einem Synthetic Actor ist erläutert [36] wobei die Paper allesamt vom selben Autor stammen (Thalmann, Daniel). Auf Seite 2 findet sich eine Tabelle mit den wichtigsten Merkmalen:

- surface modelling
- clothing
- locomotion
- grasping
- facial animation
- hair
- shadow
- skin

Es sind also sehr viel Dinge zu berücksichtigen als noch bei den Flocking Animation von Craig Reynolds. Meiner Ansicht nach gilt es jedoch zu unterscheiden, zwischen dem Rendering der Actors und der eigentlichen Animation. Das Rendern von Haaren, Kleidung usw. wird durch Standard-Game-Engines heute bereits bewältigt. In Unity3D und anderen gibt es dafür fertige Module. Viel spannender sind hingegen die Animationen als solche, also Dinge wie grasping und locomotion. Es macht also Sinn, sich nur mit den Drahtgittermodellen zu beschäftigen. Worum es eigentlich geht ist die Domain-Modellierung. Die wiederum wird in der Synthetic Vision Komponente dargestellt.

In der Fachliteratur gibt es zwar Ansätze, Synthetic Vision um weitere Komponenten zu ergänzen (Synthetic Memory und Synthetic Learning) [18] doch mir persönlich scheint das Konzept doch arg konstruiert zu sein. Offenbar wird um jeden Preis versucht, Begriffe der Kognitionswissenschaft bei der Programmierung von Softwaresystemen wiederzuverwenden. Schaut man sich jedoch einmal

den konkreten Sourcecode an, was dort konkret “Synthetic Memory” sein soll, so handelt es sich um eine C++ Klasse die Attribute abspeichert. Das eigentliche Ordnungsprinzip ist nicht so sehr Memory sondern worum es geht ist objektorientiertes Domain-Modelling. Das heißt, man überträgt die Umgebung in eine objektorientierte Syntax. Erhält also Klassen wie Tennisschläger, Pathplanner und Ball und generiert darüber dann Aktionen des Actors.

Die eigentliche Messgröße auf die es ankommt dürfte Lines of Code sein. Also wieviel Sourcecode man schreiben muss bis der virtuelle Actor auf zwei Beinen geht und einen Gegenstand greifen kann.

Der Grund warum in den wissenschaftlichen Veröffentlichungen durchgehend von Memory und Learning gesprochen wird hat etwas damit zu tun, dass die Autoren über die eigentlich interessanten Details ihrer Arbeit, insbesondere den Sourcecode und wieviel Aufwand dort hineingeflossen ist, nicht reden wollen. Bemerkenswert ist auch, dass der Code eben nicht bei github liegt sondern so getan wird, als gäbe es ihn nicht, oder als ob er unwichtig wäre. Das hat jedoch weniger etwas mit Synthetic Actors zu tun, sondern vielmehr mit dem sozialen Rahmen in dem universitäre und private Forschung betrieben wird.

2.8 Synthetic Actors

Laut [1] benötigt man zur Erstellung eines Synthetic Actors ein Gedächtnis. Dies wird realisiert als Semantic Network, genauer gesagt als Frames (siehe Marvin Minsky). Frames wiederum bedeutet, objektorientierte Programmierung einzusetzen. Die Domäne wird folglich in Klassen modelliert, die Attribute und Methoden besitzen.

Wenn man sich den oben zitierten Text genauer anschaut, so geht es wohl nicht nur um C++ sondern worum es geht ist ein Gedächtnis, also Klassen, die sich von alleine erweitern können. Folglich muss das Klassenmodell dynamisch im Speicher verändert werden. Das heißt, zur Laufzeit müssen neue Attribute hinzugefügt werden. Wie das gehen soll ist unklar, aber schön ist zumindest zu wissen, dass Frames bzw. OOP eine geeiegnete Methode darstellt Wissen zu speichern. Auf Seite 5 des oben zitierten Papers ist das in einem Algorithmus nochmal explizit dargelegt. Dort gibt es einen Unterpunkt namens “create a temporary Scene Frame using all short term memory elements”. Das hört sich nicht danach an, als ob ein Programmierer eine neue Klasse definiert sondern scheinbar erstellt das Programm seine eigenen Klassen.

In einem neueren Aufsatz ist jedoch ernüchternd festgehalten, dass Frames normalerweise manuell erstellt werden und erst in jüngerer Zeit “induction” Techniken erprobt werden:

“Until recently, frames and related representations have been manually constructed [...] Recently, there has been increasing interest in automatically inducing frames from text..” [6]

3 Lines of Code

3.1 Einführung

Der Ablauf um ein Robot-Control-System zu entwickeln ist simpel:

1. Spiel erstellen, beispielsweise eine Autorennsimulation
2. Head-up Display erstellen, was den menschlichen Fahrer unterstützt
3. Autonome Artifical Intelligenz erstellen, was die Daten des Head-Up Displays auswertet und als Behavior Tree eigene Aktionen ausführt

Als weitere Tools benötigt man noch eine effiziente Programmiersprache wie Python, eine Entwicklungsumgebung, und ein Versionscontrol-System. Doch die wichtigste Zutat fehlt noch. Wie die Überschrift dieses Kapitels vermuten lässt hat sie etwas mit dem Quellcode als solchen zu tun: Eine Autosimulation mit 200 Lines of Code sieht nicht besonders anspruchsvoll aus, eine mit 2000 Lines of Code hingegen kann sich bereits sehen lassen. Das gleiche gilt auch für Head-up Displays und Fahrerassistenzsysteme.

Lines of Code sind wohl das am meisten unterstützte Technologiemerkmal um den Reifegrad von Künstlicher Intelligenz zu beschreiben. Vermutlich deshalb, weil sich dahinter kein Algorithmus sondern Männerschweiß verbirgt. Anders kann man es nicht ausdrücken, will man beschreiben wie diese ominöse Größe zustandekommt.

[16] gibt einen kurzen Rückblick. Erst ab ca. 1960 wurde der Begriff “Lines of Code” das erste Mal verwendet. Und zwar von IBM Mitarbeitern welche sich über ihr kommendes Projekt unterhalten haben. Seit damals ist die Anzahl der Programmierer gestiegen und ihre Produktivität ebenfalls. Aktuell hat die Weseite OpenHub [12] stolze 21 Milliarden Lines of Code für OpenSource Projekte indiziert [4] Der meiste der Codebasis ist noch relativ jung. Um die Dimensionen etwas näher zu betrachten lohnt es, auf die 1980’er Jahre zurückzugehen. Eine Zeit von der relativ viele Aufzeichnungen digital verfügbar sind und wo dank der GNU Bewegung Software in die breite Öffentlichkeit getragen wurde. Der bekannte Emacs Editor besteht aus rund 1 Mio Lines of Code in der Sprache LISP. Obwohl Emacs heute niemand mehr nutzt, der halbwegs modern aufgestellt ist, dürfte das Programm zum Kanon gehören. Die Leistungsfähigkeit dieser Software definierte sich weniger durch die Programmiersprache LISP und auch nicht durch seinen Erfinder Richard Stallman sondern das eigentliche Geheimnis dürfte in der statttlichen Anzahl von 1 Mio LoC zu suchen sein. Ein Editor der aus weniger Codezeilen besteht kann vermutlich nicht das selbe wie Emacs. Zum Vergleich sei erwähnt, dass Eclipse aus 9 Mio Lines of Code besteht (Quelle: OpenHub) und der als leichtgewichtige bekannte Editor Geany bringt es immernoch auf stolze 223000 Lines of Code.

Jetzt stellt sich die Frage: wer schreibt das alles? Die Antwort ist in der steigenden Anzahl von Programmierern zu suchen. Aktuell gibt es weltweit 21 Mio, und in einigen Jahren dürften es noch mehr sein. Selbst wenn diese Programmierer nicht besonders guten Code schreiben, nur ihre eigenen Projekte voranbringen und überhaupt die Mehrzahl der Projekte scheitert bleibt immernoch genügend viel Code übrig, dass die globale Menge an Software ständig ansteigt.

Das ganze ist nicht so sehr ein wissenschaftliches Thema oder hat etwas mit neuer Technologie zu tun, sondern es ist schlichtweg der Fleiß multipliziert mit der Anzahl der Programmierer die dafür sorgt, dass es eine Flug an immer neuen Spielen, Anwenderprogrammen und in jüngerer Zeit auch Robotik-Software gibt.

Laut der Star Trek Kanon wurde Data mit einem positronischen Gehirn ausgestattet, also einer Künstliche Intelligenz. Was das sein soll ist nicht so ganz klar. Viel wahrscheinlicher ist hingegen, dass die vielen Kolonisten mit dessen Erfahrung Data gefüttert wurde, in Wahrheit Programmierer waren die zusammen die benötigten Lines of Code erstellt haben, und das wiederum hat die Maschine in einen Menschen verwandelt. Eigentlich ist die Sache sehr simpel: für einen Androiden bedarf es einer bestimmten Anzahl von Codezeilen die wiederum von Programmierern erstellt werden.

3.2 Frameworks erstellen

In den Anleitungen wie man richtig programmiert findet sich häufiger der Hinweis, man möge bitte Frameworks und Librarys erstellen

[4] http://blog.openhub.net/about/

anstatt isolierten Code zu schreiben. Schaut man sich einmal die Realität auf OpenHub an wird man das genaue Gegenteil davon sehen. Sehr kleine Projekte mit weniger als 1000 lines of Code nennen sich großspurig Framework und behaupten, dass man damit komplexere Systeme erstellen können, während größere Robotik Programme wie die PaparazziUAV-Software mit der man Dronen steuern sich sehr viel bescheidener nur als Programm bezeichnen, in Wahrheit handelt es sich jedoch um eine ausgewachsene Biblitohek. Das Unterscheidungsmerkmal ist weniger die Art der Programmierung (also ob man externe Module programmiert) sondern es ist abhängig von der Lines of Code Anzahl. Je umfangreicher der Sourcecode, desto eher lässt er sich als Library verwenden.

3.3 Objektorientierte Programmierung

Generell ist die Leistungsfähigkeit von Software abhängig von der Lines of Code. Und zwar in dem Sinne, dass Masse statt Klasse gilt. Mag ein superkurzes Programm in Forth noch so elegant wirken, in der Praxis wollen die Anwender lieber ausufernd lange Programme in Java oder C++ haben, die oberhalb von 1 Mio Lines of Code bei OpenHub taxiert sind. Obwohl viele Lines of Code immer auch Ort für mögliche Fehler sind, spielt das in der Praxis offenbar keine Rolle. Wenn man selber die Möglichkeit hat, entweder ein Programm zu verwenden was 100 kb benötigt oder eines was 100 MB verbraucht, dann ist letzteres vermutlich sehr viel leistungsfähiger und sollte bevorzugt werden.

Machen wir es etwas konkreter: Angenommen man möchte ein Computerspiel spielen. Dann kann man dieses nicht gut in wenigen Kilobytes unterbringen, sondern allein die Gameengine verbraucht viele Megabyte an Sourcecode. Und je komplexer die Programme sind, desto mehr fühlen sich davon auch Gelegenheitsnutzer angezogen, die dann mit Lua ähnlichen Sprachen Erweiterungen programmieren, was das Spiel noch attraktiver macht. Die wichtigste Frage im Bereich des Softwareengineering ist nicht so sehr: was ist die beste Programmiersprache, sondern womit kann man die umfangreichsten Programme erstellen. Und genau hier kommt objektorientierte Programmierung ins Spiel. Man mag über C#, PHP, C++ oder Java denken was man will, aber die damit durchgeführten Projekte sind vor allem sehr umfangreich. Offenbar machen es OOP-Sprachen sehr leicht, viele Lines of Code zu erstellen.

Auf den ersten Blick wirken Klassen und Objekte eher albern. Sie haben gegenüber der strukturierten Programmierung und erst recht gegen einer eleganten Sprache wie Forth fast nur Nachteile. Und wirklich verstanden haben Java auch nur die wenigsten, was auch der Grund ist warum es so viele Schlechte Spiele für Android gibt. Aber offenbar ist das kein Hinderniss mit dem Programmieren zu beginnen. Mit objektorientierten Sprachen kommt man relativ weit, selbst wenn man keinen guten Code programmieren kann. Bei hardwarenahen Sprachen wie Assembly oder forth hingegen merkt man relativ schnell, dass Programmieren doch nicht das richtige ist, weil nach ca. 5 Minuten bereits die erste unverständliche Fehlermeldung kommt und damit das Projekt als solches gestorben ist. Die Programme die tatsächlich in Forth geschrieben werden, sind in aller Regel von absoluten Profis erstellt worden. Nicht so bei Java: dort kann man erstmal 5 Monate lang viele Codezeilen tippen ohne dass sich daran jemand stören würde. Das Programm macht irgendwas, und wenn es Sicherheitsprobleme gibt, merkt das keiner.Meiner Meinung nach ist objektorientierte Programmierung die beste Methode wenn man umfangreiche Programme erstellen möchte. Also bei denen die Qualität nicht so wichtig ist und es vor allem darauf ankommt dass viele Zeilen Code geschrieben werden.

Wenn sehr gute Programmierer neu in ein Projekt dazukommen beginnen sie nicht gleich mit dem Programmieren sondern arbeiten sich zunächst in die Aufgabe ein. Sie sprechen mit den Anwendern, analysieren den vorhandenen Code und fangen frühestens nach 1 Monat an die erste Zeile Code zu tippen. Mittelmäßige Programmierer hingegen stürzen sich gleich an Tag 1 ins Getümmel. Sie erzeugen erstmal einen neuen Git branch, schreiben dort eine neue Java Klasse hinein und haben so gezeigt, dass sie schonmal die ersten 100 Zeilen Code schreiben können, ohne überhaupt sich mit dem Projekt näher beschäftigt zu haben. Leider gibt es sehr viele von mittelmäßigen Programmierern so dass die Codebasis konstant ansteigt. Das Problem mit diesem Arbeitsstil ist, dass man ihn nicht grundsätzlich als falsch bezeichnen kann, weil sich ja rein theoreitsch aus der neu erstellten Klasse etwas sinnvolles ergeben kann.

Geschichte Objektorientierte Programmierung ist sehr alt. Sie wurde erfunden in den 1970'er am Xerox Park. Zuerst wurde mit OOP eine grafische Oberfläche entwickelt um darauf aufbauend dann einen Texteditor zu programmieren und Ende der 1980'er folgte dann ein objektorientes Animationssystem. Nicht ein konkretes, sondern es gibt mehrere davon. Die entsprechende Paper sind Anfang der 1990'er erschienen und vielfach wird Pixar als Referenz genannt. Im Kern geht es darum, OOP zu verwenden um komplexe Systeme zu programmieren und diese in einer hochentwickelten Scriptsprache zu steuern.

Auch die Animationssequenzen von Craig Reynolds lassen sich am leichtesten Objektorientiert realisieren. Dabei ist es egal, ob man Smalltalk, Lisp oder eine andere Sprache verwendet, in jedem Fall helfen Konzepte wie Objekte, Abstraktion und Vererbung dabei die Komplexität zu beherschen. Das bekannteste objektorientierte Animationssystem heißt Confucius [21] [22] und ist vergleichbar mit Improv. Die Autoren bezeichnen es als:

> "general-purpose human character animation system included in an intelligent multimedia storytelling system"

Es handelt sich dabei weniger um einen konkreten Algorithmus, als vielmehr um die geschickte Kombination sehr vieler Verfahren. Es sind 3D Rendering, Sprachverarbeitung und Prozedurale Animation kombiniert worden. Möglich machte dies objektorientierte Programmierung. Offenbar war nur diese Programmiertechnik in der Lage, die Komplexität zu reduzieren. Confucius kann man als Vorläufer einer Software betrachten, mit der später Pixar ihre Filme erzeugt haben. Also ein System, was auf Knopfdruck einen abendfüllenden Spielfilm erzeugt, inkl. Storyboard, Rigging und Animation.

Leider ist es etwas schwer, Beispielvideos auf Youtube zu finden, die mittels Confucius erstellt wurden. Möglicherweise wurde das System verwendet um die folgende Shadowplay Animation zu rendern [5] Das ist jedoch nicht sicher. Auch sonstige Screenshots die das System in Aktion zeigen waren nicht auffindbar. Aber zumindest was in den Papern steht macht das ganze einen durchdachten und leistungsfähigen Eindruck.

Das Vorgängersystem ASAS (Actor/Scripting Animation System) wurde von Craig Reynolds entwickelt, der ab den 1990'er durch computeranimierte Schwärme in Erscheinung getreten ist. Die ASAS Software selber ist schon etwas älter. [33, 32] Sie wurde 1975 am MIT entwickelt und basiert auf einem objektorientierten LISP worüber man Animationen erstellen kann. Das entscheidene war offenbar nicht so sehr LISP und dessen gewöhnungsbedürfte Syntax sondern die Leistungsfähigkeit kam durch die Anwendung von OOP zum Tragen. Mit ASAS wurden die Animationen in Tron erstellt und das selbe Prinzip des objektorientierten Animationssystem kam später bei den ersten Pixar Filmen zum Einsatz. Keineswegs geht es nur um das Rendern der Animation an sich, also das Berechnen von Licht und Schatten, sondern worum es eigentlich ist eine Animationssprache mit der

[5] https://www.youtube.com/watch?v=IDHRHBxONIo

man Drehbücher erstellen kann. Also um animierte Drahtgittermodelle zu erzeugen. Derartige Animationssysteme sind universell einsetzbar von der Computeranimation, über Computerspiele, Augmented Reality bis hin zu Robotics.

3.4 Kleine Geschichte von C++

Die objektorientierte Programmierung ist untrennbar mit C++ verbunden. Ab 1985 war dies die erste Sprache, welche von der Industrie breit unterstützt wurde und objektorientierte Features beinhaltete. Schaut man sich die parallel dazu verlaufende Entwicklung des Personalcomputers, der Computerspiele und der Fernster-Betriebssysteme an, so war es wohl kein Zufall dass ab 1985 sowohl C++ einen Boom erlebte während Gleichzeitig die ersten umfangreichen Betriebssysteme und Anwenderprogramme entstanden. C++ wurde explizit dafür entwickelt um Große Projekte zu realisieren. Nach wie vor gilt die Sprache als schwer zu erlernen aber offenbar ist es mit C++ immernoch leichter komplexe Applikationen zu schreiben als mit jeder anderen verfügbaren Sprache, insbesondere mit C.

Historisch betrachtet macht C++ Ende der 1980'er Jahre durchaus Sinn. Zum einen hatten die damaligen Computer nur wenig Leistung, gleichzeitig wollte man jedoch umfangreiche Spiele und Programme realisieren. C++ ist die ideale Verbindung aus leistungsfähiger Programmiersprache plus objektorientierter Features. Die C++ Compiler erzeugen sehr effizienten Code und durch die Unterteilung in Klassen bleibt er dennoch leicht wartbar.

Aus heutiger Sicht ist C++ nicht länger die beste Wahl. Es hat allenfalls historische Gründe, warum sich Leute damit noch auseinandersetzen. Viele der großen Codebasen aus dem Bereich MS-Windows sind nach wie vor in C++ gehalten. Neuere Programmiersprachen wie C#, Java und Python lassen sich leichter verwenden als das altmodische C++. Dennoch sollte erwähnt sein, dass zum objektorientierten Paradigma bis heute keine Alternative gefunden wurde. Im Grunde ist ein Buch aus dem Jahr 1985 was C++ thematisiert nach wie vor relevant wenn man größere Softwareprojekte umsetzen muss.

Obwohl es dazu keine belastbaren Quellen gibt wage ich mal die These, dass C++ dafür gesorgt hat, dass die maximale Programmgröße sprunghaft angestiegen ist. Und zwar jene Größe, welche von einem Programmierteam noch geschrieben und gebugfixed werden kann. Zwar gab es vor Erfindung von C++ auch schon Fensterbetriebssysteme und Computerspiele. Aber im Regelfall waren die Programme kompakt. Bestanden also aus nur wenigen Codezeilen und waren minimalistisch gestaltet. Erinnert sei hier an die ersten Texteditoren, die buchstäblich mit 4 kb Speicher auskamen und wo es noch keine Menüleiste gab, sondern man einfach die ESC Taste drückte um ins Hauptmenü zu gelangen. Ebenfalls spartanisch muten die ersten Actionspiele an. Bei Pong gibt es außer einem Schläger und einem Ball nichts was den Spieler groß ablenken würde. Wichtigstes Merkmal auch dieser Programme war der geringe Codeumfang. Zwar kann eine CPU auch sehr umfangreiche Assembler Programme verarbeiten, nur irgendwie gelingt es Menschen nicht derartige Programme zu schreiben.

3.5 Komplexität von objektorientierter Programmierung

Anfänglich sieht OOP unübersichtlich aus. Es gibt in der Regel 20 oder noch mehr Klassen, die untereinander netzartig verbunden sind, zusätzlich ist noch der Zugriff auf fremde Klassen nicht oder nur mit Umwegen möglich. Warum das Konzept OOP dennoch sinnvoll ist ergibt sich wenn man versucht für eine einzelne Klasse zu definieren was ihre Aufgabe ist. Nimmt man sich nur eine Klasse aus einem UML Diagramm heraus gibt es nur wenige Schnittstellen. Entweder erbt sie Eigentschaften von anderen Klassen, oder ist irgendwo als Instanz eingebunden. In beiden Fällen gibt es dafür Linien-Verbindungen. Mehr Schnittstellen als diese Linien gibt es nicht und so lässt sich ziemlich gut sagen, was die Aufgabe ist.

Ein weiterer Punkt bei OOP besteht darin, dass es zwar sehr viele Klassen in einem Programm gibt, man diese aber zu Gruppen zusammenfasst. Diese werden als Layer, Packages oder Module bezeichnet. Man kann darüber aus einem UML Diagramm herauszommen, sieht also nicht mehr hunderte Einzelklassen sondern nur 3-4 Packages die sehr weitreichende Aufgaben ausführen. Diese Gliederung ist keineswegs nachträglich im Rahmen einer Dokumentation erstellt worden, sondern ergibt sich unmittelbar aus dem Sourcecode. Jedenfalls bei OOP Projekten.

3.6 UML Diagramme für echte Systeme

Wie man ein selbstfahrendes Auto programmiert, ist schwer zu sagen. Was man aber sagen kann ist wie eine fertige Software ungefähr aussehen dürfte. Die Idee, Machine Learning einzusetzen, damit sich die Software von allein verbessert ist eine Utopie. Sie funktioniert in der Realität nicht. Was stattdessen gemacht wird, ist objektorientierte Programimerung zu verwenden um die Domäne zu modellieren. Ein Beispiel UML Diagramm für ein driverless besteht folglich aus Klassen wie Vehicle, Road, Trafficlight, Trajectory, Sensor, Lidar. Diese Klassen sind über Verbung verbunden und sie sind in weitere Unterklassen aufgeteilt. Wenn man unterstellt, dass man rund 10 Mio Lines of Code benötigt und eine Klasse maximal 200 Lines of Code lang sein darf (guter Programmierstil), dann kann man sich leicht ausrechnen, dass das audgedruckte UML Diagramm riesige Außmaße annimmt und allenfalls vom Schaltplan eines Intel-Mikrorprozessors getoppt wird.

Softwareengineering lässt sich nicht gut automatisieren. Es gibt zwar sogennante CASE Tools die versprechen aus UML automatisch Sourcecode zu erzeugen, und es gibt auch Forschungsansätze welche mit der Sprache KLONE nicht nur Wissen darstellen sondern dieses Wissen auch automatisch aus Volltexten ableiten, nur leider sind diese Dinge nicht einsatzfähig für echte Projekte. Im wesentlichen läuft es darauf hinaus, dass echte Programmierer echte Teams bilden und darin dann Software nach dem OOP Paradigma entwickeln, um so Auto zu fahren. Das klingt zunächst wie Overengeneering, weil ein Auto zu steuern simpel ist, wenn man sich mit PID Reglern auskennt. Im Grunde gibt es nur zwei Variablen: Lenkereinstellung und Geschwindigkeit. Mehr muss der Autopilot nicht regeln. Nur, um diese simple Entscheidung zu treffen braucht man jede Menge Background Knowledge und das wiederum wird in objektorientierten Programmiersprachen kodifiziert.

3.7 UML Malprogramme

Rein Anzahlmäßig gibt es ungefähr seit dem Jahr 2000 sehr viele UML Tools. Das hat damit zu tun, dass ab dieser Zeit die UML Spezifikation verabschiedet wurde und damit die Boochs Notation aus Mitte der 1990'er abgelöst wurde. Man mag es kaum glauben, aber von 1985-2000 hatte man keine einheitliche Notation um Klassen zu zeichnen.

Seit den Anfängen wurde UML häufig kritisiert. Im wesentlichen handelt es sich dabei nur um eine Menge an grafischen Symbolen. Das ganze ist nichts anderes als eine Mindmapping-Methode. Dennoch hat sich UML zum Quasistandard herausgebildet. Nicht nur OOP ist unentbehrlich sondern auch UML. Es wird sehr gerne an Universitäten unterrichtet aber auch die industrielle Softwareentwicklung setzt es ein. Leider gibt es gibt heute kein Standard-Tool um UML Diagramme zu zeichnen. Für Eclipse gibt es mehrere Tools, dann gibt es noch Together von Borland, ArgoUML wird häufig noch genannt und in letzterer Zeit Dia. Nach meiner Recherche ist Dia das derzeit

beste Programm. Anders als ArgoUML ist es kein reines UML Casetool, sondern sieht sich in Tradition von Visio. Man kann damit munter beliebige andere Diagramme zeichnen und UML ist nur eine Symbolbibliothek und vielen. Aber, dia besitzt anders als so manch andere Programme mit dia2code einen Code-Generator und autodia kann mit etwas herumprobieren auch reverse Engineering durchführen. Das ist keineswegs selbstverständlich.

Ich erwähne UML deshalb, weil diese Modellierungssprache in der Lage ist die Entwicklung umfangreicher Software zu unterstützen. Angenommen, man möchte ein selbstfahrendes Auto programmieren. Dann benötigt man dafür mit Sicherheit ein UML Diagramm. Komischerweise wird dieser Sachverhalt in der Literatur bisher nur selten thematisiert. Hier [25] ist die bekannte Ausnahme. Dort hat man zumindest im Ansatz eine Ontologie entwickelt. Diese ist zwar viel zu simpel gestrickt aber im vergleich zur sonst verfügbaren Literatur ist es das beste was Google Scholar derzeit zu bieten hat. Man sieht darin mehrere Abbildungen wo sich schon jemand Gedanken darüber gemacht hat, dass man Straßen nach 2-spurig und 1-spurig unterteilt, und das es verschiedene Typen von Fahrzeuugen gibt.

Der Grund warum das in der Literatur nur selten thematisiert wurde, hängt weniger damit zusammen, dass Google, BMW und Co nicht erkannt hätten wie wichtig UML ist, sondern damit dass ein fertiges UML Diagramm verraten könnte, wie die Software funktioniert. Und da es ein ungeschriebenes Gesetz gibt, keinen Sourcecode zu veröffentlichen werden auch keine UML Charts veröffentlicht. Aber das lässt den Hobbyprogrammierer umso mehr Möglichkeiten sich auszutoben. Wann immer man einen Linefollower Mindstorms Roboter, einen Sumo-bot oder ein RC-Car steuern möchte, ist der erste Schritt das Zeichnen eines UML Diagramm. Ob man dazu UML 1, UML 2 oder sogar OWL/RDF verwendet ist fast egal. In jedem Fall geht es darum, Klassen zu bennenen, Vererbungshierarchieen zu zeichnen, Methoden und Attribute festzulegen.

Das ganze ist eine reine Fleißarbeit, die nur selten zu ausführbaren Programmen führt, aber es ist ein wichtiger Schritt hin zu autonomen Fahren. Nur mit objektorientierter Programmierung plus UML lässt sich die Komplexität managen. Ein Meta-Modell was besser wäre als UML wurde bis heute nicht erfunden. Na gut, fast nicht. Die Erklärtexte auf Wikihow wie man sich an einer Kreuzung verhält sind ebenfalls nützlich wenn man Autos programmieren möchte. Wenn man so einen Wikihow Text in ein UML Diagramm überführt hat man schonmal ein sauberes Konzept erarbeitet, womit Programmierer dann weiterarbeiter können.

Es gibt mehrere Paper die sich mit Knowledge extraction von Wikihow beschäftigen. Die Idee ist es, einen Wikihow Text in eine OWL Ontologie zu überführen. Aktuell ist keines dieser System einsatzfähig. Will man eine halbwegs übersichtliche Ontologie erhalten, die sich als UML Diagramm verwenden lässt muss man schon manuell vorgehen. Also den Wikihow-Text lesen, verstehen und dann mit Dia manuell das Diagramm zeichnen.

3.8 Alternative Programmiersprachen

Etwas merkwürdig ist es schon, dass als Programmiersprache der Wahl nach wie vor LISP, Forth und APL gehandelt werden. Nicht nur, dass es dazu viele Bücher gibt, sondern youtube bietet sogar zahlreiche Fachkonferenzen wo man die letzten Geheimnisse der umgekehrten polnischen Notation und von Meta-Compilern erführt. Aber warum werden diese Sprache noch eingesetzt? Das hat weniger etwas mit deren Mächtigkeit zu tun, sondern ist mit Nostalgie zu erklären. So ähnlich wie sich rund um den Commdore 64 eine treue Fangemeinde gebildet hat, hat auch die Generation davor technische Dinge in ihr Herz geschlossen. Früher gab es jedoch keine 8 bit Microcomputer, die galten damals noch als Zukunftsmusik. :Stattdessen hat sich die jugendliche Generation von ganz früher mit LISP, APL und anderen Dingen die Zeit vertrieben. Wenn heute dazu Konferenzen durchgeführt werden, dann in Erinnerung an die als schön erlebte Zeit.

Leider ist keine der genannten Programmiersprachen anschlussfähig. So wenig wie man mit dem C-64 echte Probleme lösen kann, ist es möglich mit Forth echte Software zu entwickeln. Rein formal vielleicht, man kann sich selbstverständlich die J1 Forth CPU herunterladen, diese auf einen FPGA flaschen und damit dann Programme ausführen. Nur, welchen Zweck soll es haben? Richtig, es dürfte überwiegend darum gehen sich in die gute alte Zeit zurückzudenken, als es noch keine CISC Architekturen und keine C-Compiler gab. Wer sich daran gern erinnert ist zu beneiden, aber die meisten werden damit nichts mehr anfangen können.

Der Grund warum die alternativen Programmiersprachen und die Fans von veralterter Technik so leicht haben, ihre Weltsicht auch in der Gegenwart als Mainstream zu verkaufen ist darin begründet, dass die aktuelle Computergeneration ein Leistungsproblem hat. Damit ist gemeint, dass zwar neue Systeme besser sind, aber nicht soviel als das man damit LISP und Co übertrumpfen kann. Böse formuliert, haben wir heute immernoch die selbe alte Hard- und Software die schon im Xerox Parc unsere Großeltern haben erfunden und seitdem nicht besonders viel dazu erfunden . Anstatt über LISP und Co zu meckern, sollte sich die jetzige Generation einmal an die eigene Nase fassen und sich fragen, wo sie denn geblieben ist die Computerrevolution. Also Software, die besser ist als das was früher da war.

Ok, ganz so schlimm ist es dann noch nicht. Zwischen einer ausrangierten PDP/11 und einer modernen Fedora Workstation wo Python läuft ist schon noch ein Unterschied zu erkennen. Im Sinne eines besser / mächtiger.

3.9 Haskell für Roboter?

Haskell ist eine funktionale Programmiersprache welche auf einem hohen Abstraktionsniveau arbeitet. Anstatt Probleme als Step-by-Step Instructions zu befolgen wird ein deklarativer Programmierstil verwendet. Wie das in der Praxis aussieht, kann man anhand eines Primzahl-Finde-Programs sehen. Der Code in Haskell ist nicht nur kürzer sondern wird von der CPU auch in weniger Takten abgearbeitet. Es gibt mehrere Versuche, Haskell zur Robotik-Programmierung einzusetzen. Mein Eindruck ist jedoch, dass es sich dabei um eine Sackgasse handelt. Ausgangspunkt ist die Annahme, dass Robotik-Programmierung zwingend bedeutet eine große Menge an Lines of Code zu schreiben. Je mehr Domain-Wissen man in Software übertragen will desto größer wird der Code. Roboterprobleme wie Selbstfahrende Autos, oder humanoide Laufroboter erfordern sehr viel Domainen-Wissen. Leider eignet sich Haskell nicht gut, wenn man vorhat mittelmäßige Programmierer größere Mengen an Code schreiben lassen zu wollen. Hier gibt es das selbe Problem wie mit LISP und mit Forth. Technisch gesehen sind die Sprachen einem C-Compiler überlegen und auch funktionale Programmierung ist besser ist objektorientierte Programmierung. Nur leider ist besser nicht automatisch besser, und zwar deshalb nicht weil das Interesse der Welt sich auf neues einzulassen begrenzt ist. Damit ist gemeint, dass die meisten Leute mit OOP, Java und Python vertraut sind und nicht auf bessere Alternativen umsteigen wollen.

Wenn man Haskell zur Robotik-Programmierung einsetzt kämpft man an zwei Seiten gleichzeitig. Erstens muss man die Leute überzeugen, das Haskell die bessere Sprache ist und zweitens muss man ihnen erklären wie sie Roboter programmieren sollen. Das ist in der Mischung extrem anspruchsvoll, ich glaube nicht dass sich das durchsetzen wird.

Natürlich sind Leute die bereits mit funktionalier Programmierung

vertraut sind, und die reaktive Patterns für Roboterprobleme aufstellen könne mit Haskell an der richtigen Adresse. Keine andere Programmiersprache ist derart leistungsfähig. Nur wieviele Leute gibt es, auf die das zutrifft? Üblicherweise ist es so, dass Anfänger überhaupt noch nicht programmieren können und allenfalls oberfläche Kenntnisse mitbringen. Ihr Wissen über Roboter ist fast null. Hier ist es besser sie dort abzuholen wo sie schon sind, also Python oder Java zu verwenden und schön prozedural vorzugehen aber nicht funktional.

Best-practice Vielleicht sollte ich an dieser Stelle einmal definieren, wenn Haskell nicht der richtige Weg ist wie man besser Roboter programmieren kann. Folgende Techniken haben sich als sinnvoll herausgestellt:

- interpretierte Programmiersprache wie Python um häufige Edit-Run-Zyklen zu erzielen
- Objektorientierte Programmierung um viele Lines of Code zu verwalten
- Version Control System
- Behavior Trees um Actions zu planen wie sie in Computerspielen eingesetzt werden
- Head-up Displays und Augmented Reality um semi-autonome Systeme zu erzeugen
- texteuelle Heurstiken aus Wikihow und Walkthroughs für Computerspiele verwenden die man dann in Computercode überträgt

Wie man schon sieht taucht in dieser Liste nicht auf, dass man Haskell oder funktionale Programmierung verwenden sollte. Es würde die Dinge nur unnötig schwerer machen.

3.10 Programmiersprachen in der Künstlichen Intelligenz

Welche Programmiersprachen in der Künstlichen Intelligenz bevorzugt eingesetzt werden ist eindeutig ermittelt worden: LISP und Prolog. Fast alle Bücher in denen es um konkiete Implementierungen geht setzen auf diese Sprache. Manchmal wird ergänzend noch APL2, Forth und Smalltalk eingesetzt. Aber könnte das womöglich die Ursache dafür sein warum die 5. Computergeneration als gescheitert gilt? Ich meine, welche Ergebnisse hat die akademische KI Forschung denn hervorgebracht? Richtig gar keine.

Die wichtigste Erkenntnis lautet, dass es durchaus Alternativen zu LISP und Prolog gibt. Ab ungefähr den 1990'er ging ausgehend von Rodney Brooks die "New AI Welle" los, welche auf Behavior Based robotics gesetzt hat. Ebenfalls in den 1990'er verbreitete sich schlagartig Ingame-AI, damit sind Computerspiele gemeint die um Bots angereichert wurden. Üblicherweise kommt bei Ingame AI Lua, C++ oder Java zum Einsatz, weil das die Sprachen sind die Spieleentwickler kennen. Künstliche Intelligenz in Computerspielen hat anders als die akademische KI in den 1980'er konkrete Resultate erzielt, in dem Sinne dass heute NPC Charaktere fester Bestandteil sind und man ziemlich gut beschreiben kann wie man für neue bisher unbekannte Spiele solche AI-Systeme programmiert.

Wenn man mit Non-LISP Sprachen vergleichbare oder bessere Ergebnisse erzielt als mit LISP, dann kann etwas mit dem Paradigma was die Akademische KI Forschung in den 1980'er betrieben hat nicht stimmen. In dem Sinne, dass man auf das falsche Pferd gesetzt hat und zwar alle Wissenschaftler gleichermaßen. Der Hauptgrund dürfte im Selbstverständnis der Universität zu suchen sein. Üblicherweise ist LISP als Lehrsprache ausgelegt, sie erfüllt neben Künstlicher Intelligenz noch einen anderen Zweck: die Didaktik. LISP wiederum baut auf auf den General Problem Solvern welche in den 1950'er ebenfalls an den Universitäten entwickelt wurden. Womöglich hat diese Traditionslinie dazu geführt, dass man eine Art von Betriebsblindheit entwickelt hat die dazu geführt hat, dass die KI-Forschung sich eher verzögert als gefördert hat. Böse formuliert ist LISP ungefähr jene Sprache welche man einsetzen sollte, wenn man auf gar keine Fall intelligente Software entwickeln möchte sondern will dass das Projekt scheitert. LISP ist ein Anti-Pattern der Künstlichen Intelligenz.

Eigentlich sollte man annehmen, die akademische KI Forschung hätte ihr Versagen erkannt und wäre jetzt mit einer Fehleranalyse beschäftigt. Komischerweise gibt es jedoch keine Paper mit dem Titel "Why LISP has failed". Das Thema wurde stattdessen verdrängt. Und auch zu den kommerziellen Spieleentwicklern wo KI eine Selbstverständlichkeit ist ist man auf Abstand gegangen. Computerspiele tauchen im universitären DIskurs praktisch nur am Rande auf. Die Gründe dafür sind, dass Universitäten im gesellschaftlichen Fokus stehen und das eine technikkritische Gesellschaft ganz froh ist, dass die Universitäten mit der 5. Computergeneration gescheitert sind. Erinnert sei an den Lighhill Report wo in den 1970'er Jahren verkündet wurde, dass Computer niemals werden eigenständig denken können. Gewissermaßen hat der Lighthill Report definiert, was das Ergebnis sein wird: das es nicht gelingt Künstliche Intelligenz zu realisieren.

Aus technisch-wissenschaftlicher Sicht ist das natürlich falsch. Man kann durchaus KI Systeme siehe die Bots aus Halo oder anderen Spielen. Nur wird das nicht an den Universitäten betrieben sondern das Thema ist abgewandert zu den Kommerziellen Publishern und zu den Hobbyprogrammierern. An den Universitäten gibt es heute keine ernstgemeinten KI Projekte. Das einzige was noch in diese Richtung geht ist die OpenCOG / AGI Bewegung die weniger die technische Realisierbarkeit untersucht als der Singularity Bewegung zugeordnet wird. Eine moralisch aufgeladene Debatte die weitestgehend ohne konkrete Technik funktioniert, sondern eher moralisch ethisch argumentiert. Also die Schreckensutopie an die Wand malt, wo Maschinen die Menschen versklaven und man sich dagegen ausspricht. Mit Informatik hat das weniger zu tun, eher mit dem Versuch einen Diskurs zu dominieren.

Am besten hat das Versagen der akademische KI-Forschung Rodney Brooks Anfang der 1990'er aufgezeigt. Das erstaunliche war, dass Brooks damals Teil der KI Forschung war. Er war mehr oder weniger der einzige der erläutert hat warum LISP gescheitert ist. Seine Paper lesen sich auch heute noch mit Gewinn. Darin wendet er sich gegen das Topdown Paradigma. Damit sind General Problem Solver, Strips Solver und LISP Expertensysteme gemeint. Gemeinsames Merkmal von Topdown AI ist es, in abstrakter Form die Domäne zu definieren und dann mit Algorithmen darin nach sinnvollen Aktionen zu suchen. Brooks bezeichnet das Vorgehen an sich als Irrweg und schlägt stattdessen vor, die Idee einer Umweltmodellierung ersatzlos zu streichen. Damit hat Brooks das sogenannte Scripting AI vorweggenommen, eine pragmatische Vorgehensweise die ab den 1990'er in kommerziellen Spieletiteln eingesetzt wurde um NPC Charaktere zu steuern. Die bekannteste Realisierung des Bottomup Ansatzes sind Behavior Trees. Es handelt sich um eine Entwicklungsmethode mit der Designer das Verhalten von Bots festlegen.

Von der akademische KI Forschung wurde Brooks weitestgehend ignoriert. Er wird heute ausschließlich im Bereich Game-AI zitiert. Diese Spielart findet – wie oben schon erwähnt – außerhalb der Universitäten statt und darf getrost als eigentliche Künstliche Intelligenz bezeichnet werden. Es sind Systeme die tatsächlich funktionieren, die also noch dem Einschalten etwas tun und zwar autonom.

Der Grund warum sich Brooks bei seinen Universiären Kollegen hochgradig unbeliebt gemacht hat ist darin zu suchen, dass er nicht die KI Forschung als solche kritisiert, sondern seinen Fokus auf die

Software die in diesem Umfeld entstanden ist. In den meisten seiner Paper geht Brooks auf Projekte ein wie Mycin, Lisp und den bekannten General Problem Solver. Damit blendet Brooks das aus, wofür diese Projekte stehen, nähmlich konkrete wissenschaftliche Fragestellungen. Brooks verdreht gewissermaßen die Verhältnisse. Anstatt sich wissenschaftlich mit den Themen Denken und Problemlösen zu beschäftigen schaut er sich die Letzte Version eines Expertensystems an, sagt ziemlich frech dass das Programm Unsinn auf den Bildschirm ausgibt und damit ist sein Urteil gefällt. Ausgehend von der Programmausgabe darf dann der Seitenhieb auf 20 Jahre akademische KI Forschung nicht fehlen und Brooks ist fertig mit seinen Fachkollegen. Dieser respektlose Umgangston ist etwas, was die universitäre KI Forschung und die Welt der kommerziellen Softwareentwicklung entzweit hat. Heute ist Brooks CEO einer Firma und ist nicht länger im universitären Umfeld unterwegs, während gleichzeitig die Universitäten vorgeben, Brooks nicht mehr zu kennen.

3.11 Anmerkungen zum Mergen bei git

Neueinsteiger in die Versionsverwaltung mit git stolpern vor allem über eine Sache: das Mergen. Mag git an sich noch halbwegs leicht zu verwenden sein, spätestens das Mergen ist ein Grund git nicht zu verwenden. Anstatt sich mit den Feinheiten eines 2-Wege Merge unter berücksichtigung der log-historie auseinanderzusetzen, verzichtet man lileber komplett auf eine Versionsverwaltung und bleibt beim gewohnten Texteditor plus Dateisystem, wo man einfach ein Backup von seinem Code erstellt und darin Veränderungen vornimmt. Vielleicht ist es an der Zeit ein wenig über das Mergen aufzuklären. Die gute Nachricht vorneweg: man kann es komplett ignorieren. Mergen ist die unwichtigste Funktion in git. Weder Einsteiger noch Fortgeschrittene Anwender werden damit in Kontakt kommen. Mergen bedeutet üblicherweise, dass in einem git Projekt mehrere Autoren gleichzeitig Veränderungen vornehmen und dabei die selben Textstellen verändern. Dieser Fall mag theoretisch interessant sein, kommt aber in praktischen Projekten nicht vor. Das durchschnittliche git Projekt was bei github liegt oder was Programmierer in Firmen bearbeiten ist ein single-User Projekt. Das heißt, die leute erzeugen mit “git init” in ihrem Verzeichnis ein git Projekt, programmieren da ihren Code hinein, und zur Zusammenarbeit mit anderen Entwicklern werden Foren, Wikis und E-Mails verwendet.

Rein theoretisch kann man die Zusammenarbeit auch über die git merge Funktion erledigen und sich dadurch dann Wikis und Foren sparen. Nur das Problem ist, dass wenn Code ausgestauscht wird, es selten um rein technische Dinge geht, also welchen Parameter man an den Merge Befehl dranhängt, sondern üblicherweise erfolgt die Zusammenarbeit natürlichsprachlich. Und dafür sind Foren und ähnliches weitaus besser geeignet.

Aber mehr noch, selbst wenn man die Merge Funktion von git tatsächlich nutzen möchte, wird man feststellen, dass sie nicht funktioniert, jedenfalls nicht so wie es in den Anleitungen drinsteht. Nehmen wir mal an, jemand hat ein 100 Zeilen Java Programm geschrieben und möchte das mit einem 50 Zeilen Python Script was ein anderer Entwickler in der selben Firmen geschrieben hat vermelzen. Reicht hier ein git merge Befehl aus, oder ist das nicht eine Aufgabe die sehr viel komplexer ist? Richtig, mit git Merge kann man das Zusammenführen des Codes nicht erreichen, sondern stattdessen muss man manuell den Python Code nach Java konvertieren. Dazu legt man am besten in git einen neuen Branch an, kopiert initial den Fremdcode hinein, editiert darin dann herum, erzeugt fleißig Commits und erst ganz am Ende nimmt man per Copy & Paste das Resultat und fügt es in seinen Master Tree hinzu. Dieser Ablauf dauert rund 1 Woche, erforderte mehrere manuell git commits, geht einher mit einer Diskussion auf einer Mailing Liste und erforderte keinen einzigen explizite git Merge Befehl. Kurz gesagt, Mergen ist etwas anderes als Mergen.

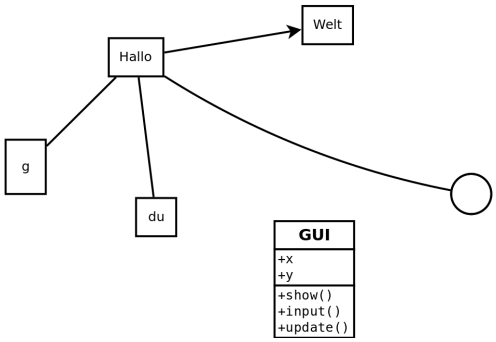

Abbildung 1: Beispiel Diagramm mit Dia

4 UML

4.1 Einführung in Dia

Das UML Zeichenprogramm Dia gilt nicht als besonders leistungsfähig. Es ist ein aufgebohrtes Vektorzeichenprogramm. Es hat jedoch einige Vorteile: zunächst ist es in den meisten Linux-Distributionen als Paket enthalten, zweitens verwendet es eine XML Datei zur Speicherung und zu guter letzt ist es halbwegs aktuell. Zugegeben, die Plugins Autodia und dia2code sind etwas hackelig, aber es gibt sie. Ich habe es selbst an einem Beispiel ausprobiert. Und zwar zuerst ein kleines Python Projekt im Code erstellt, dort ein externes Modul eingebunden und das ganze mit autodia in die Dia Syntax konvertiert.

Leider hat Dia nicht erkannt, dass die Klassen im externen Modul als Package zu formatieren sind, aber zumindest werden die Klassen angezeigt. Man kann sie jetzt manuell in ein Package verschieben. Das geht dadurch, dass man zuerst ein Package erstellt, dann die Klasse darüberzieht, beides markiert und “objects -> adopt” auswählt. Das ist zugegebenermaßen kompliziert, aber es funktioniert. Man kommt damit – etwas Handarbeit vorausgesetzt – zu ansehnlichen UML Diagramme die gerne auch etwas komplexer sein dürfen. Wo also Klassen zu Gruppen zusammengefasst werden. Das beste dürfte es ohnehin sein, wenn man das automatisch erzeugte Klassendiagramm nur als Basis nutzt um from scratch das eigentliche UML Diagramm zu erstellen, wo man dann unwichtige Methoden weglässt. Der Vorteil ist, dass wenn man es manuell macht, es relativ flott geht. Man zieht einfach die Steuerelemente ins Fenster hinein, vergibt noch die Namen und kann das fertige Diagramm dann nach SVG exportieren. Schön ist auch die Ausrichten Funktion, womit man die Klassen genau bündig anordnen kann.

Dia hat aber noch einen weiteren Vorteil. Und zwar kann man anstatt UML Diagramme auch normale Mindmaps damit zeichnen. Es gibt also ganz normale Kreise die man verwenden kann um da Text hineinzuschreiben und die dann mittels Pfeilen zu verbinden. Logisch, dass beim Verschieben der Elemente, die Verbindungen angepasst werden. Und auch hier steht wieder die volle SVG,PDF und JPEG Exportfunktion zur Verfügung. Es gibt für Linux zwar auch andere Mindmapping Tools wie z.B. Labyrinth, doch Dia ist um einiges mächtiger. Der Clou ist, dass man Mindmaps mit UML verbinden kann. Das entspricht zwar nicht der UML 2.0 Notation ist aber hilfreich um Abläufe zu visualisieren.

Als mögliche Alternativen zu Dia kommt einmal LibreOffice Draw in Betracht. Leider ist dieses Tool etwas hackelig zu bedienen, wie auch die übrige LibreOffice Suit. Man kann damit wohl in der Theorie UML Diagramme erstellen ... Und zum Zweiten wären noch UML Case Tools wie die Eclipse Plugins zu nennen, die stärker verzahnt sind mit

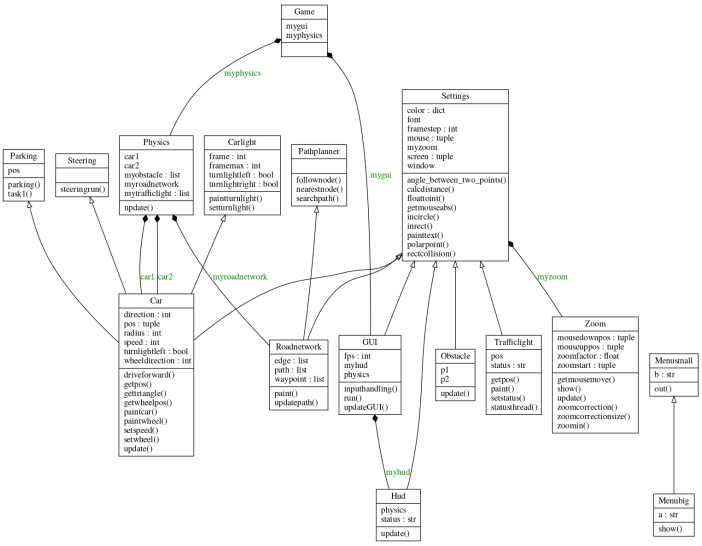

Abbildung 2: pyreverse

dem generierten Sourcecode. Leider ist das sogenannte Roundtrip-Engineering bei dem man mühelos zwischen UML und Sourcecode wechselt nur ein frommer Wunsch, der aktuell von keinem Tool eingehalten wird. In der Theorie mag das mit Eclipse und erst recht mit den vielen IBM Programmen so gehen, doch ich möchte da erstmal ein Paper zu sehen, wo in einem echten Projekt mit 100000 Lines of Code jemand das tatsächlich in Verwendung hat. Mein Eindruck ist eher, dass echte IBM Projekte nach wie vor mit Fortran und APL durchgeführt werden, aber das ist nur eine zynische Unterstellung. Für alle die nicht wissen was APL ist, es handelt sich um einen Vorläufer von LISP der um einiges schwieriger zu bedienen ist.

4.2 Pylint und pyreverse

Der Dia Anwendung merkt man deutlich an, dass sie zum manuellen Zeichnen von UML Diagrammen entwickelt wurde. Es gibt zwar mit autodia einen Parser für Python Code, doch der Parser ist nicht besonders hochentwickelt. Deutlich professioneller ist die Software Pylint. Wenn man damit ein Python Programm auswertet ergibt sich ein ansehnliches UML Diagramm. Darin werden nicht nur die Klassen erkannt, sondern sogar Instanzierungen werden in der korrekten UML Syntax wiedergegeben.

Leider gibt es ein Problem. Die dot-Syntax welche pyreverse verwendet kann man nicht in Dia weiterverarbeiten. Es ist also nicht möglich, die Datei zu öffneen und noch ein wenig die Klassen zu verschieben. Der Vorteil von pyreverse besteht darin, dass die erzeugte dot Datei sich automatisch ausrichtet, das heißt, die Klassen werden übersichtlich dargestellt und zwar mit Hilfe der Graphviz Algorithmen.

Ich würde mal vermuten, wenn man auf die Schnelle eine Python Datei in eine UML Grafik verwandeln möchte, dass pyreverse der beste Ansatz dafür ist. Vielleicht findet sich ja noch weitere Software mit der man .dot Files nachbearbeiten kann.

Die Schwierigkeit besteht darin, dass auch pyreserver nicht alle Abhängigkeiten auflöst und im Zweifel die erzeugte Grafik nichtssagend ist. Es werden zwar hübsch die Verbindungen der Klassen aufgezeigt, aber ein Mehrwert gegenüber dem reinen Sourcecode gibt es nicht. Ich wage mal zu behaupten, dass der Klassenbrowser in der IDE (wo also die Klassen zum Aufklappen angezeigt werden) um einiges übersichtlicher ist, als die obige UML Darstellung. Besonders bei echten Projekten wo nicht 12 Klassen sondern gerne 500 und mehr verwendet werden stößt eine UML Visualisierung schnell an eine Grenze.

Was man mit UML jedoch ausgezeichnet tun kann ist zu erläutern was OOP überhaupt ist. In der obigen Grafik sieht man zwei mögliche Schnittstellen: einmal die Vererbung und einmal die Instanziering. Vererbung ist Klassenorientiert, wo man also eine Vorlage erstellt und Instanzierung bedeutet, aus der Klasse ein konkretes Objekt zu machen was man mit Werten befüllt. Nehmen wir eine Klasse mal heraus: Carlight. Ein Blick auf die Grafik verrät, dass sie nur mit einer Linie mit dem Rest des Netzes verbunden ist. Zunächst einmal ist das ganze also eine Sandbox, mit einer übersichtlichen Anzahl an Attributen und Eigenschaften und lediglich die Verbindungsllinie deutet daraufhin, dass sie innerhalb des Programms eine Funktion ausführt.

Das schöne ist, dass zufällig Carlight aktuell noch einen Bug enthält, genauer gesagt funktioniert die Klasse komplett falsch und muss durch etwas besseres ersetzt werden. Anstatt jedoch im gesammten Code herumzuscrollen und die Software als ganzes zu verbessern reicht es sich auf die 25 Codezeilen von Carlight zu konzentrieren. Diese umzuschreiben oder gar komplett neu zuschreiben ist eine übersichtliche Aufgabe.

4.3 Details zu pyreverse

Derzeit sind mir zwei Tools bekannt um aus Python Code rückwärts ein UML Diagramm zu zeichnen: autodia und pyreverse. Autodia hat den Vorteil, dass man das Diagramm manuell nachbearbeiten kann, pyreverse hingegen ordnet die Klassen übersichtlicher an. Im Zweifel würde ich pyreverse empfehlen. Es gibt dort einen simplen Schalter mit dem man auch größere Projekte grafisch aufbereiten kann: “-k”. Dieser unterdrückt die Attribute und Methoden und gibt nur die Klassennamen aus. Man erhält so relativ unkompliziert einen Überblick über das Gesamtprojekt.

Leider hat pyreverse einen Nachteil: die Dokumentation ist nicht besonders gut und man kann nicht interaktiv in der Grafik browsen. Will man die Ansicht detailierter Anzeigen muss man einen Programmaufruf ausführen. Deutlich effizienter in der täglichen Programmierung ist der Klassenbrowser der IDE, dieser ist allerdings nicht grafisch.

4.4 Pattern Languages

Pattern Languages sind Techniken die oberhalb der objektorientierten Programmierung anzusiedeln sind. Es handelt sich um Meta-Verfahren die ausdrücken wie man zu einem UML Diagramm gelangt, also welche Objekte zu einer Domäne benötigt werden. Erfunden wurde die Pattern Language zuerst im Bereich der Architektur und später auf Softwareengineering übertragen. Aber, darunter darf man sich keine C++ artige Programmiersprache vorstellen und es gibt auch keine Software die eine Pattern Language in Computercode überträgt, sondern das ganze ist ein Projekt-Management-System. Am einfachsten kann man sich eine Pattern Language vorstellen wie Scrum oder das Wasserfallmodell. Also eine Beschreibung wie Projektmanagement funktioniert bei dem mehrere Leute beteiligt sind.

Obwohl das in der Fachliteratur nicht so explizit herausgearbeitet wurde kann man sagen dass die Pattern Lanaguage im Bereich Architektur vor allem darauf abzielt, wie Architekten großprojekte organisieren. Also wie man 10000 Bauarbeiter und noch mehr die auf einer Baustelle gemeinsam etwas machen anleitet, Aufgaben verteilt und mit Problemen umgeht. Also klassisches Scrum-Projektmanagement eben. Und leider liegt hier auch der Nachteil von Pattern Languages. Sie könne im Erfolgsfall nur das leisten was auch Issue-Tracking-Systeme leisten können: die Arbeit auf mehrere Personen verteilen und ein wenig die Abstimmung untereinander verbessern, aber letztlich braucht es Programmierer welche die Arbeit ausführen.

Dennoch ist das Konzept aus theoretischer Sicht interessant. Weil es versucht Softwareengineering zu formalisieren. Also zu

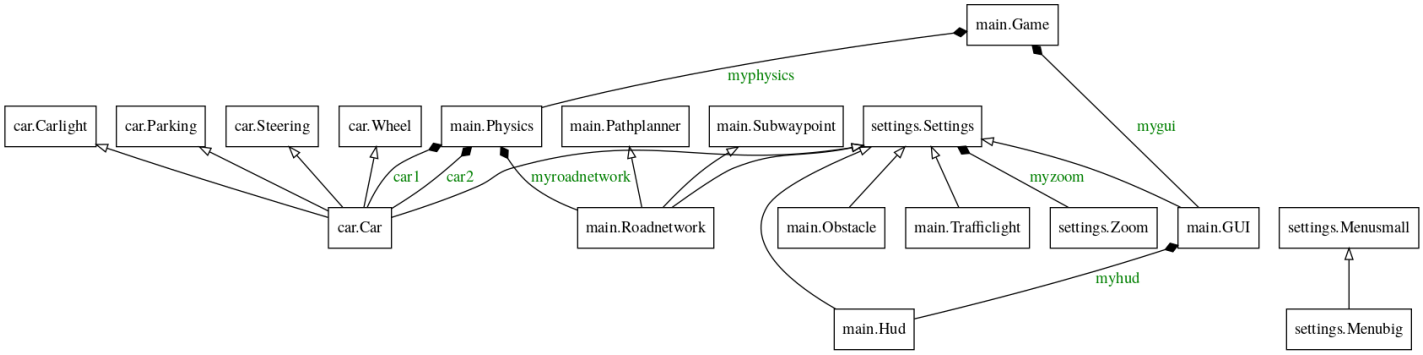

Abbildung 3: pyreverse mit unterdrückten Details

```
pyreverse -o svg *.py -my -k
```

beschreiben wie genau man von einem leeren Blatt zu fertiger Software gelangt.

4.5 Wissensmodellierung

Das Rad zweimal erfinden ist keine gute Idee. Also wäre es sinnvoll, wenn man bereits erstellte UML Diagramme ein zweites Mal nutzt. Leider gibt es kein einheitliches Austauschformat. Der Mindmap Editor den man selber verwendet hat, ist in spätestens 1 Jahr nicht mehr da. Offiziell gilt inzwischen der Eclipse MDT/UML Standard als wegweisend, aber wenn man versucht dazu Bespiele zu finden so gibt es keine. Eclipse MDT ist nur eine Methode unter vielen um UML Diagramme zu zeichnen. Aber zumindest geht das schonmal in die richtige Richtung. Und zwar wird man erkennen, dass die Java Community gerne und viele UML Diagramme zeichnet. Mehr als die C++ Community. Der Grund liegt auf der Hand: Java ist weit verbreitet und ist objektorientiert. Also gehört UML irgendwie dazu.

Mit etwas Suchen habe ich doch noch einen Standard gefunden, er basiert nicht auf UML sondern auf RDF. RDF kann man als großen Bruder von UML bezeichnen. Bei UML geht es um das grafische Visualisieren von Klassen und wird auf dem Desktop zum Programmieren verwendet, RDF hingegen gehört zum SemonaticWeb und ist als Internet-Standard geplant. Laut Wikipedia dient RDF dazu, Wissen zu speichern und zwar in der Form Subjekt, Prädikat, Objekt. Wie z.B: “Baum hat Blätter”, hat ist dabei das Prädikat.

Man kann die Tools eingrenzen: die Programmiersprache ist Java, der Standard zur Wissensmodellierung ist RDF und zur Domain-Modellierung wird UML eingesetzt. Das ist zwar noch etwas wage, aber man weiß jetzt wo man nach weiteren Informationen suchen kann. Die von RDF abgeleitete Syntax RDFS geht in Richtung UML Klassenmodellierung. Es ist ein Standard um Klassen in Domänen zu beschreiben.

Zugegeben, bisher klingt das noch alles sehr theoretisch. Wer sagt, dass man damit etwas in der Praxis anfangen kann? [38] beispielsweise, er hat auf Basis von RDF ein Question Answering System entwickelt. Die Idee ist es, alle Fakten in einer RDF Datenbank zu speichern, und mit dieser kann man in einen Dialog treten. Also in natürlicher Sprache eine Frage stellen die das System dann beantwortet. So eine Art von Chatbot, nur dass er funktioniert. Das ganze ausführlicher und auf deutsch hat [11] in petto.

Mit etwas suchen ist es mir gelungen, noch ein Paper ausfindig zu machen, was RDF basierende Wissensmodellierung auf ein “Driver Assistent System” anwendet: [13] Darin wurde zunächst einmal die Spatiale Situation als OWL Ontologie beschrieben (also wo die anderen Fahrzeuge sind) und auf dieser Basis wurden dann Schlüsse gezogen. Die OWL Datei kann man als eine Art von angereichertem UML Diagramm betrachten.

4.6 UML ist mächtiger als gedacht

Anfangs dachte ich noch, dass die UML Diagramme zwar nett anzuschauen wären aber weitestgehend ignoriert werden können. Und in der Tat glaube ich nach wie vor, dass es nicht möglich ist, aus einem UML Diagramm ausführbaren Java Code zu erzeugen und demzufolge UML ungeeignet ist um den Softwareengineering Prozess zu unterstützen. Aber, UML ist in umgekehrter Richtung sehr gut einsetzbar, und zwar wenn die Domänenmodellierung bereits abgeschlossen ist, also das Computerspiel in Form von ausführbarem Javacode bereits bei github hochgeladen ist und jetzt es darum, rückwärts daraus nützliche Informationen zu extrahieren. Beim Durchstöbern der UML Spezifikation sind mir vor allem 3 Diagrammtypen positiv aufgefallen:

- Klassendiagramm, was man als eine Ontologie der Domäne bezeichnen kann
- Sequenzdiagramm, was die Methodenaufruf in zeitlicher Abfolge zeigt
- Aktivitätsdiagramm, was eine Prozedur in Form eines Programmablaufplans darstellt

Wenn man aus vorhandenem Java-Code alle drei Diagrammtypen rückwärts erzeugt, besitzt man ein sehr gutes Domänenmodell. Unklar ist bisher noch, wie genau man das einsetzen kann, um damit einen Domänenparser zu erstellen oder sogar eine Künstliche Intelligenz zu erstellen. Aber zumindest UML scheint hier schon mal eine gute Ausgangsbasis zu sein.

Ich stell mir den weiteren Ablauf so vor. Angenommen wir haben es mit einem Fußballspiel zu tun. Der menschliche Spieler führt in diesem Spiel Aktionen aus. Das UML Reverse Engineering hat ein vollständiges Domänenmodell generiert. Damit müsste jetzt ein Parser die Aktionen des Spielers tracken. Das heißt, wenn er den Ball irgendwohin spielt, werden darüber Methodenaufrufe erzeugt (Einfluss auf das Sequenzdiagramm), es werden Klassen aktiviert (Klassendiagramm) und es werden konkrete Prozeduren / StateMaschines ausgeführt (Aktivitätsdiagramm). Was jeweils aufgerufen wird ist bekannt, weil der Javacode fehlerfrei durchläuft und man ihn tracken kann. Die Frage ist nur noch, wie man das miteinander in Beziehung setzt.

Ich will damit andeuten, dass UML zwar nutzlos ist, um eine Domäne in Programmcode zu überführen (Also für das Softwareengineering) aber ich glaube, dass es danach als Wissensbasis dienen kann um die Interaktionen mit dem Programm zu parsen.

4.7 Event Klassendiagramm

Es gibt durchaus Anleitungen in der Form “C++ Programming for Games”. Dort ist immer auch ein Abschnitt “Künstliche intelligenz” enthalten. Doch leider sind die Anleitungen zu oberflächlich, es wird dem Leser zuviel vorenthalten. An dieser Stelle möchte ich erläutern wie eine Event-Ontologie funktioniert. Dies ist ein wichtiges Element einer Ingame-AI.

Zunächst gehe ich davon aus, dass das Spiel bereits programmiert wurde und fehlerfrei läuft. Die Autos fahren in einem 2D Racecar Game schön im Kreis herum, die Rundenzeit wird ermittelt und die Kollisionsabfrage funktioniert. Jetzt geht es nur noch um die Implementierung der KI. Anders als bei Robotik-Problemen ist eine OpenCV artige Bilderkennung unnötig. In der Klasse Car ist die Position jedes Autos auf den Pixel genau gespeichert. Um daraus eine Künstliche Intelligenz zu entwickeln muss man das vorhandene UML Diagramm erweitern. Genauer gesagt um Klassen zur Beschreibung von Events.

Ein Event ist dabei wie alles was man in C++ programmiert mindestens eine Klasse, meist aber eine Klassenhierarchie. In der Hauptklasse Event könnte man als Attribute festlegen: Framenummer, beteiligtes Auto, beteiligter Bereich, erkannte Aktion. Diese Atrribute werden in weiteren Klassen noch genauer spezifziert, ein Bereich kann beispielsweise das Segment vor einer Kreuzung sein, mögliche Aktionen können sein Kollision, zu geringer Abstand, Geschwindigkeitsüberschreitung usw.

Natürlich gab es die Events auch vorher in dem Spiel. Nur waren sich nicht explizit als Klassendiagram formuliert. Sie waren implizit enthalten in den Pixelinformationen. Das heißt, wenn Auto1 dicht auf Auto2 aufgefahren ist, sah man als Zuschauer dass es zu dicht ist. Und im C++ Programm hätte man auslesen können wie groß der Abstand zwischen beiden Autos beträgt. Aber, es gibt normalerweise keine Klasse umsowas detailiert zu tracken. Und genau für soetwas benötigt man eine Event-Klassendiagramm, also eine seperate Ontologie die die vorhandenen Sensordaten semantisch auswertet.

Es mag ein wenig merkwürdig klingen wenn man eine Ingame-Überwachungskamera programmiert, weil ja eigentlich da nichts zu überwachen ist. Aber ungefähr darum geht es. Die Kamera funktioniert nicht so sehr mit einer Auflösung von 640x480 Pixel, sondern es ist eine semantische Kamera. Also eine Pipeline die vorhanden Informationen ausliest und zu höherwertigem Wissen verdichtet.

5 OWL

5.1 Nachteile von RDF

Im letzten Kapitel wurden RDF Ontologien bereits kurz vorgestellt. Ich habe mir die Mühe gemacht, die OWL Datei welche im Paper [13] verlinkt wurde grafisch zu veranschaulichen. Zunächst die Fakten. Die Datei [6] ist 104 kb groß, wird in Google Chrome als XML Datei textuell angezeigt und lässt sich mit einem der zahlreichen Online-OWL Visualisisier anzeigen. Als Visualisierer habe ich mich für “owlgred” entschieden aber man kann auch eine Desktop Anwendung wie Protege dazu verwenden. Wenn man die OWL Datei lädt wird eine Art von Klassendiagramm angezeigt, also Klassen die Attribute haben und untereinander vernetzt sind. Das ganze entspricht dem RDF/OWL Standard und gilt als ziemlich advanced. Leider kann man mit dieser Ontologie nicht viel anfangen. Man kann hineinzommen und herauszoomen das wars dann aber auch. Vermutlich kann man die OWL Datei auch nach graphviz exportieren und noch ein wenig hübscher Zeichnen, also anstatt orthogonale Verbindungen auch Splines als Linien einzeichnen. Nur, letztlich bleibt das ganze nichts anderes als ein UML Klassendiagramm.

Und wenn man einmal im Detail sich das ganze anschaut so wurde es höchstwahrscheinlich aus einem C++ Programm reversed engineered. Und genau das ist die falsche Vorgehensweise, weil eigentlich RDF und andere Modellierungssprachen entwickelt wurden um daraus dann Sourcecode zu erzeugen, sie wurden erfunden um das Programmieren zu vereinfachen.

Schaut man sich die konkrete OWL Datei jedoch einmal an, so sieht diese zu stark nach echtem Code aus, obwohl in der Datei selber kein Code gespeichert ist. Es drängt sich folgender Verdacht auf: jemand hat für ein Computerspiel einen Bot programmiert, dafür ganz normal einen Behavior Tree erstellt, Sourcecode programmiert und Klassen erstellt. Und als er die 2000 Lines of Code fertig hatte, hat er daraus rückwärts die OWL Datei erstellt. Das wiederum kann man sich dann grafisch anzeigen lassen.

Nun kann man natürlich fragen ob die Reihenfolge nicht egal ist. Nein ist es nicht, weili man so ja ohnehin vorgeht, also zuerst die C++ Datei schreiben, den Bot testen. Bei der Objektorientierten Modellierung bzw. RDF geht es ja darum, das zu vereinfachen, also Metamodelle zu erstellen von denen ausgehend dann Sourcecode erzeugt wird. Nur, schaut man sich das konkrete Beispiel einmal an, so scheint genau das nicht zu klappen. Auch die Sprachsyntax der RDF Datei selber ist nicht mächtig genug um mit Java oder C++ Code zu konkurrieren. Anders formuliert, mein Verdacht ist, dass RDF/OWL ein Flop ist. Es ist nichts anderes als ein einheitliches Dateiformat um graphviz ähnliche Diagramme zu erzeugen. Die Klassen und objekte in einer OWL Datei kann man nicht aus Cyc oder ähnlichen Versuchen ableiten sondern will man eine funktionsfähige OWL Datei haben, muss man vorher den C++ Code schon geschrieben haben und kann dann rückwärts die OWL Datei erzeugen.

Fragen wir doch einmal die Literatur, was es darüber zu berichten gibt. Es gibt relativ wenig Aussagen dazu. Bekannt ist das Java2owl Tool, das bezeichnen die Autoren aber nicht als Sourcecode to Ontology Konverter sondern definieren seine Funktion abgeschwächt als Synchronisation. Das man also Java und OWL Datei immer auf dem selben Stand hält. Ebenfalls abgeschwächt wird über eine Ontologie Extraktion aus einer relationalen Datenbank gesprochen. Wozu es jedoch sehr viele Paper gibt, und was wohl auch die Intention hinter OWL ist, dass sind Verfahren bei dem Natural Language in eine Ontologie umgewandelt wird. Das ist vom Selbstverständnis der OWL Community die richtige Vorgehensweise.

Ein wenig ketzerisch sei die Frage gestattet warum das Gegenteil davon, also Java to OWL, so selten thematisiert wird. Gleichzeitig muss erwähnt werden, das vom technischen Aspekt her, diese Konvertierung ausnahmsweise perfekt funktioniert. So wie man aus einer vorhandenen Java Datei mittels pylint ein UML Diagramm zeichnen kann, kann man auch eine OWL Datei extrahieren. Ja mehr noch, die so gewonnenen OWL Ontologien dürften sehr nahe dran sein am Optimum. .Sie lassen sich verwenden um daraus dann den Sourcecode Template zu erzeugen. Obwohl das natürlich witzlos ist, weil man ja die Original Java Datei ohnehin besitzt.

Fassen wir das ganze zusammen. Das was die OWL Community eigentlich anstrebt ist aus natürlichsprachlichen Texten Ontologien zu extrahieren um damit dann im nächsten Schritt zu ablauffähigen Code zu gelangen. Technisch gesehen funktioniert das schlecht, bzw. gar nicht. Was technisch außerordentlich gut funktioniert ist der umgekehrte Weg, also aus bereits erstelltem Code die OWL Datei zu extrahieren und dann zu behaupten man hätte den Code gar nicht. Das ganze ist eindeutig nicht das worum es bei OWL geht und vermutlich wird es mit Absicht auch nicht in der Literatur thematisiert.

Leider ist die Folge daraus, dass man mit OWL die Erstellung von Software nicht erleichtern kann. Nach wie vor muss man die Software

[6] http://vi.uni-klu.ac.at/ontology/DrivingContext.owl

zuerst manuell programmieren um dann im Schritt 2 daraus das Modell zu erzeugen.

Ich habe mir nochmal eingehender das Paper [13] angeschaut. Im Grunde kann man den Autoren folgendes unterstellen. Sie haben ein kleines Softwareprojekt zum selbstfahrenden Auto durchgeführt, dafür etwas C++ Code geschrieben, diesen getestet, wollten aber den Code nicht auf github veröffentlichen weil sie ihn als wertvoll einschätzen, also haben sie daraufhin aus ihrem eigenen Code eine OWL Datei erzeugt und die veröffentlicht. Das heißt, Außenstehende wissen jetzt grob, welche Klassen und Attribute in der Software enthalten sind, haben aber nicht den genauen Code. Mit Wissensmodellierung hat das ganze wenig zu tun, sondern eher mit Visualisuerung von Sourcecode der schon da war.

Ich kann mich noch an ein weiteres Paper erinnert wo ebenfalls es um OWL Ontologien ging die angeblich mit Protege erstellt wurden. Auch die sahen ziemlich perfekt aus, so als ob jemand vorher schon den ablauffähigen Code in Java geschrieben hat, den Bot oder was auch immer schon in Aktion gesehen hat, und dann nachträglich daraus dann die OWL Datei abgeleitet hat.

Aber warum passiert soetwas, wer ist Schuld an diesem Dilemma? Ich glaube das Hauptproblem ist, dass OWL gänzlich ungeeignet ist zur Wissensmodellierung. Das einzige was wirklich eine Domäne beschreiben kann ist ausführbarer Code. Entweder Java, C++ oder Python. Wenn man darüber ein Problem wie Autofahren, Buchhaltung oder Tetris spezifiziert hat, kann man rückwärts daraus 1A UML Diagramme und OWL Daigramme ableiten. Nur leider hat man davon keine Vorteile, weil ja der Code geschrieben werden musste. Scheinbar ist es schwer bis unmöglich eine Wissensmodellierung zu entwickeln die besser ist als was objektorientierte Sprachen beherschen. Im Grunde ist Java die perfekte Modellierungssprache. Man kann daran Klassen, Objekte und prozedurale Abläufe spezifzieren. Und zwar für jedes Problem.

OWL/RDF ist vom Selbstverständnis als Ebene oberhalb des Sourcecodes gedacht. Wo man also aus Texten zuerst ein formales OWL Modell erzeugt und das dann in Java Code überträgt. Nur in der Praxis funktioniert es so nicht. Es gibt kein einziges funktionierendes Beispiel. Und die vorhandenen OWL Dateien die halbwegs gut designt sind, wurden aller Wahrscheinlichkeit nach nicht aus natürlichen Texten sondern aus fertigem ablauffähigen Java Code rückwärts generiert.

Qdox Wie man aus Java Sourcecode rückwärts OWL Dateien erstellt wird in [7] beschrieben. Verwendet wird ein Parser namens Qdox der aus einer Java Datei alle Klassennamen und Methondenamen extrahiert um diese als OWL Datei abzuspeichern. Qdox ist vergleichbar mit pyreverse und autodia, die aus Python Dateien UML Diagramme erzeugen. Das oben genannte Paper hat als Nutzungsszenario große Firmen im Blick die Unmengen an Sourcecode intern erstellt haben und in einem Repository verwalten wollen. OWL dient als Frontend um leichter in bereits erstelltem Code browsen zu können in der Hoffnung damit die Wiederverwenbarkeit von Code zu erleichtern. Mit Sicherheit könnte man Qdox auch einsetzen um das Stackoverflow Archiv zu durchsuchen und dafür dann riesige Topic-Maps zu erzeugen die sich grafisch durchsuchen lassen. Wie schon weiter oben erwähnt widerspricht das komplett der SemanticWeb Philosophie. Aber, es funktioniert technisch ausgezeichnet.

Anders formuliert: selbst wenn sich das Semantic Web irgendwann durchsetzt führt das nicht dazu, dass das Schreiben von Sourcecode überflüssig wird. Ganz im Gegenteil. Vermutlich wird noch mehr Code firmenintern und bei Stackoverflow erzeugt als jemals zuvor. OWL dient lediglich dazu, diesen Code grafisch zu veranschaulichen und in Beziehung zusetzen.

5.2 OWL ist ein Witz

Je mehr ich mich mit OWL und RDF beschäftige desto stärker beschleicht mich der Verdacht, dass es sich dabei um eine Verschwörung handelt. Ausgangspunkt ist die Vorstellung man könnte aus einer OWL Datei eine Java-Datei erzeugen. Fragt man Google danach, so finden sich unzählige Webseiten und Foreneinträge die genau das suggerieren. Demnach wäre OWL eine Metabeschreibungssprache mit der man Anwendungen entwickeln kann. Das Problem ist nur, dass sich für diese These keine Beispiele finden lassen. Und wenn man selber versucht from scratch erst in Protege eine OWL Datei zu erzeugen um daraus dann Java Code zu generieren wird man erkennen, dass es nicht funktioniert. Der dabei erzeugte Code ist nicht sinnvoll nutzbar um damit weiterzuarbeiten. Das liegt jedoch weniger an den Tools die Schwächen haben sondern vielmehr ist das Konzept OWL an sich das Problem.

Schauen wir uns einen Usecase an, in dem OWL wunderbar funktioniert. Und zwar wenn man den umgekehrten Weg geht, also aus einer Java-Datei eine OWL Datei erzeugt. Wenn man Java2OWL Konverter einsetzt wird man genau das erhalten was man möchte: ein sauber formatiertes Klassendiagramm mit den Linien, in dem man herumbrowsen kann. Auch hier gibt es manchmal noch Bugs, weil die Parser nicht alle Java Statements ausgewertet haben, doch das sind Kleinigkeiten. Man kann sie in späteren Programmversionen verbessern. Es besteht kein Zweifel dass ein java2owl Konverter Sinn macht. Leider gibt es ein Problem. Diese Richtung des Workflows entspricht nicht der Selbstdarstellung der SemanticWeb und UML-Community. Wenn man schon den Java Code geschrieben hat, wozu soll man dafür rückwärts dann eine Metamodell erzeugen? Richtig, es passt nicht ins Selbstverständnis was OWL Sein soll.

Das Problem ist jedoch, dass dies der Wahrheit umso näher kommt. Vergleichen wir dochmal die Ausdrucksfähigkeit von Java Sourcecode mit OWL. Man wird feststellen, dass Java unglaublich mächtig ist. Man darin Spiele programmieren, man kann da Kommentare hineinschreiben, man kann Vererbungshierarchien definieren, man kann Attribute und Methoden festlegen. Eigentlich alles, um Java als die perfekte Metasprache zu bezeichnen. Die Methoden die man noch nicht implementieren möchte, lässt man einfach offen und schreibt lediglich einen Textkommentar hinein, die anderen Sourcecode-Teile hingegen programmiert man aus. Wenn man das ganze noch unter ein git Versionsverwaltungssystem stellt, hat man den perfekten Workflow bei dem man das Klassendiagramm wie auch den Code parallel entwickelt (literate Programming). Und jetzt schauen wir uns im Gegenteil einmal OWL an. Man kann dort gar nichts machen: Prozeduren gibt es nicht, die Attribute folgen einer merkwürdigen Syntax, Kommentare sind keine vorgesehen und ein simpler Befehl wie a = 1+1; ist auch nicht enthalten. Was also soll OWL sein?

Böse formuliert ist das ganze kein Semantisches Modell, sondern OWL ist ein Malprogramm. Es steht auf einer Stufe mit graphviz, Visio und MS-Paint. Kurz gesagt, es ist Spielzeug aber nicht für ernsthafte Probleme anwendbar. Komischerweise sieht das die OWL Community etwas anders. Sie brüstet sich in Papern damit, dass sie aus OWL Modellen (wie sie es nennen, in Wahrheit sind es OWL Strichzeichnungen) Java Code erzeugt hätten. Sie glauben, damit das Programmieren zu revolutionieren und wer es wagt den Kaiser als nackt zu bezeichnen der hat angeblich nicht verstanden was OWL ist.

Ich würde nicht unbedingt behaupten wollen, dass die Welt nur darauf gewartet hat ein 5000 Zeilen langes Java Programm in eine grafische Darstellung zu überführen um zwischen den Klassen herumzubrowsen. Der Mehrwert derartiger Schaubilder ist nicht hoch. Aber, es ist zumindest interessant seinen eigenen code einmal auf diese Weise zu betrachten und korrekt ist das so erzeugte Modell auch. Der umgekehrte Weg jedoch (OWL to Java) endet naturgemäß

in einem Desaster. Es ist ein Anti-Pattern wie Programmieren eben genau nicht funktioniert. So werden keine echte Projekte durchgeführt.

Schaut man sich einmal an, wie die Konvertierung einer OWL Datei in eine Java Datei üblicherweise im akademischen Bereich durchgeführt, so machen die Dozenten folgendes. Sie nehmen sich vorhandenen Java Sourcecode der auf github liebt, erzeugen daraus die OWL Datei rückwärts und behaupten, sie würden jetzt from Scratch ein Projekt beginnen. Sagen also, es gäbe nur die OWL Datei aber noch keinen Code und wenn man jetzt die OWL Datei in Java überführt, erhält man eine sehr saubere Spefzifikation wie das Projekt aussehen soll. Die Studenten sind sprachlos und machen sich eifrig dabei die fehlenden Methoden zu ergänzen. Das ganze ist keine Wissenschaft sondern es ist Betrug. Wenn man noch nicht den Sourcecode kennt ist es nicht möglich eine OWL Datei zu erzeugen. Vor allem nicht wenn man als Input natürliche Sprache verwendet, wie es in dutzenden Papern behauptet wird. Auch das hat mit Wissenschaft nicht zu tun. Das einzige was technisch möglich ist, ist auch hier die umgekehrte Richtung, also: Java -> OWL -> natural language. Das funktioniert ausgezeichnet.

Ich möchte also nicht behaupten, dass OWL an sich nichts taugt, sondern lediglich in einer bestimmten Richtung angewendet werden darf. Nähmlich ausgehend vom Java code über die OWL Datei bis hin zur natürlichen Sprache. Das ist ein Workflow der mit Software gut bewältigt werden kann und wo es Sinn macht wenn man ihn bestreitet.

Aber woran liegt es, dass sich Java zu OWL transformieren lässt aber OWL nicht in Java? Es hat etwas mit Domain-Modellling zu tun. Dazu ein kleines Beispiel. Angenommen es wird eine Trafficsimulation programmiert. Es gibt darin, Autos, Ampeln, eine Fahrbahn und Verkehrsregeln. Die fertige Trafficsimulation also das Modell sieht so aus, dass man 10000 Lines of Code in ausführbarem Java Code hat, der nach dem Start ein Computerspiel anzeigt. Dieses sollte bugfrei sein und alles enthalten was eine gute Software ausmacht. Das ist der Zielzustand der angestrebt wird. Der Javacode mit den 10000 LoC ist das worum es geht. Es enthält alle benötigten Informationen.

Die erste Frage lautet, wie gelangt man zu dem Code. Üblicherweise sichtet ein Programmierer dazu vorhandene Informationen, spricht man Experten und programmiert daraus dann das Spiel. Er verwendet also externe Informationen die weder auf seiner Festplatte noch sonstwo gespeichert sind. Programmieren bedeutet diese wagen Informationen in maschinenlesbaren Code zu überführen.

Und genau hier wird deutlich warum OWL bei diesem Domain-Modelling nichts verloren hat. Nehmen wir mal an, der Programmierer sichtet die Informationen mit dem Ziel daraus eine OWL Datei zu erzeugen. Er weiß natürlich, dass OWL nichts anderes ist als eine grafische Notation. Er braucht sich nicht viel Mühe zu geben sondern legt einfach 20 Klassen an, malt einige Strichen dazwischen und behauptet das wäre die fertige OWL Notation. Natürlich kann man damit nicht weiterarbeiten, sie ist komplett wertlos will man die Software programmieren muss man nochmal from scratch anfangen. Wenn man hingegen als Programmierer mit dem Ziel an die Aufgabe herangeht nicht nur die OWL Datei sondern echten Code zu erzeugen, spezififziert man die Dinge viel exakter. Das so erzeugte Modell ist wesentlich besser geeignet die Realität zu beschreiben.

Die Schwierigkeit besteht darin, dass OWL->Natural Languge und Java->OWL Generatoren technsich zwar möglich sind, aber dass ihre Aussagekraft begrenzt ist. An erster Stelle steht immer der Java Code und den zu lesen ist simpel. Es ist nichts, was man noch großartig verbalisieren oder grafisch visualisieren müsste. Man schaut sich den Code an und weiß im Regelfall was er tut. Besonders wenn er verständlich geschrieben wurde. Die erwähnten Konvertoren (Java->OWL, und OWL->Text) liefern zwar korrekte Ergegnisse und mögen nett sein, wenn man Langeweile hat, doch neue Erkenntnisse über das Programmieren lernt man nicht. Im Regelfall werden dort nur Details weggelassen. so kann man sich nur die Oberklasse anzeigen lassen und die auch noch mit Linien verbinden.

Beim Software-Engineering jedoch geht es um etwas anderes. Und das Programmieren, also die Frage wie man from Scratch zu neuem Java-Code gelangt und genau dieses Problem kann OWL nicht beantworten. Deshalb habe ich etwas ketzerisch das Verfahren in der Kpitelüberschrift als Witz bezeichnet. OWL ist ein Pseudoverfahren was auf Selbstillusion basiert. Die Frage ist weniger wo die Schwächen von OWL liegen sondern die Frage ist eher, warum Java Sourcecode so ausgesprochen mächtig ist, dass sich darin komplette Spiele ausdrücken lassen.

Aber gehen wir nochmal zurück zum Eigentlichen Softwareengineering. Bisher blieb die Frage unbeantwortet was sich oberhalb von Javacode befindet. Bisher wurde nur herausgearbeitet, dass OWL sich dort zumindest nicht befindet, weil das eine Spielzeug-Malsprache ist. Oberhalb von Java Code befindet sich ein soziales Biotop, genannt Community aus Java Programmierern. Also jene Menge aus Hochschulen, Schulen und Betrieben wo Menschen sich eingehender mit der Programmiersprache auseinandersetzen, sich in Foren darüber unterhalten und eigenen Code auf github hochladen. Das ist jener Rahmen auf den man fokussieren muss, wenn man die Erstellung von Software untersuchen möchte. Dieser soziale Rahmen ist für die Informatik nicht zugänglich. Er ist nicht formalisierbar und er folgt keinen Algorithmen. Sondern man kann ihn nur mit den Methoden der Soziologie untersuchen. Es geht um eine Beschreibung dessen wie Programmierer sich weiterbilden, wie sie neues lernen und worüber sie motiviert werden. Das ist kein Thema für das Software-Engineering sondern es ist ein Thema was in den Geistes- und Sozialwissenschaften verortet ist.

Die wichtigste Technik um soziales Coding zu beschreiben ist ein geschichtlicher Zugang, das man also untersucht wie sich das Programmieren im Laufe der Jahre verändert hat. Von den ersten Mainframes über Homecomputer bis hin zur heutigen Java Szene. Dieser Merkmalsraum ist es, der erklärt was Java ist, wie neue Software entsteht und was mögliche Nachfolger zu Java sein werden. Die geschichtliche Fokussierung mag überraschen, weil Programmieren etwas anderes ist als Alterntumskunde oder Literaturgeschichte. Aber ist es das wirklich? Programmieren lässt sich sehr wohl formalisieren, aber in Modellen die die Sozialwissenschaften aufstellt. Es handelt sich um biographische Modelle, Zeitlinien die untersucht werden, sowie Diskurse zwischen Geisteswissenschaftlern in denen These und Anti-These herausgearbeitet werden.

5.3 General Gameplaying ist sinnlos

In der akademisch geprägten Fachliteratur zur Künstlichen Intelligenz gibt es einen ganzen Wust an Papern welche sich der Frage widmen wie man erstellen von Bots vereinfachen kann. In den 1980'er waren Expertensysteme und Prolog angesagt, später folgte OWL, General Gameplaying und Deeplearning. All diese Verfahren haben eine Gemeinsamkeit: menschliches Programmieren wird als ineffizient bewertet und es wird nach einem Metamodell gesucht von dem ausgehend man das Problem lösen kann.

Vorbild für diese deduktiven Verfahren dürften Suchalgorithmen und Pathplanner sein. Das Labyrinth ist vorgegeben und über den Algorithmus wird ein Weg zum Ziel gefunden. Der Weg den es zu finden gilt ist jetzt jedoch das komplette Computerprogramm. Es wird nur selten Hinterfragt ob das Konzept an sich möglicherweise fehlerhaft ist. Wenn man menschliche Programmierer aus der Loop herausnimmt und Systeme anstrebt, die sich selber verbessern verliert man damit etwas sehr wichtiges: den Programmierer. Auch in der Robotik ist es keine gute Idee von manueller Steuerung auf Au-

tomatik umzuschalten, weil man eben nicht klar sagen kann, was anstattdessen die Steuerung übernehmen soll. Und derart naiv gehen auch die Propheten des automatischen PRogrammierens zu Werk. Sie behaupten schlichtweg, dass menschliche Programmierer überflüssig wären, glauben auf sie verzichten zu können, besitzen aber keine Vorstellung davon wie ohne menschliche Programmierer Software entstehen soll.

Wichtig ist zunächst einmal die Erkenntnis dass es bis heute kein Computersystem was einen menschlichen Programmierer ersetzen kann. Was also in der Lage ist, eine Aufgabe in Sourcecode zu überführen, diesen zu testen und zu verbessern. Wenn man also in der Gegenwart Programmgeneratoren einsetzt, wird in Folge dessen die Softwarequalität dramatisch abnehmen und schlimmstenfalls auf der Stufe von Garbage enden.

Die einzig richtige Antwort auf dieses Problem besteht darin, menschliche Programmierer grundsätzlich in der Loop zu belassen und ihnen Hilfsmittel an die Hand zu geben effektiver zu werden. Beispielsweise git, objektorientierte Hochsprachen, IDE Texteditoren und Online-Foren. Zugegeben, der Effekt dieser Tools ist kaum messbar so gering ist er. Aber es ist das beste, was aktuell zur Verfügung steht. Darüberhinaus gibt es nichts womit man die Erstellung von Sourcecode erleichtern kann.

Eine kleine ökonomisch motivierte Überschlagsrechnung zeigt auf, dass dies auch gar nicht nötig ist. Laut der letzten globalen Erhebung gibt es exakt 18 Millionen Programmierer weltweit. Jeder ist in der Lage pro Tag 10 Zeilen Code zu schreiben. Das macht auf ein Jahr bezogen die Summe von 65,7 Milliarden Codezeilen. Zum Vergleich: Windows 10 besteht aus nur 60 Mio Lines of Code. Anders formuliert, selbst bei der heutigen eher geringen Gesamtanzahl von Programmierern und einem sehr niedrigen Arbeitstempo von 10 Zeilen täglich wird mehr Code erzeugt, als man sinnvollerweise benötigt. Eine Notwendigkeit die Produktivität massiv zu erhöhen oder gar automatisierte Programmgeneratoren einzusetzen besteht zu keinem Zeitpunkt. Ja fast möchte man sagen, dass der jährliche Output an Codezeilen eine natürliche Größe ist die ähnlich wie die Fischpopulation in den Weltmeeren automatisch da ist und nur abgefischt zu werden braucht. Man muss da nichts voranbringen oder fördern, sondern die Programmierer sind einfach da und sie erzeugen von ganz alleine den Sourcecode.

Ja ich würde sogar behaupten wollen, dass es ein Überangebot an neu generiertem Code gibt. Weil die oben veranschlagte Menge nicht einmalig entsteht sondern jedes Jahr. Wir haben also im Jahr 2015, 65 Milliarden Zeilen neuen Code, im Jahr 2016 erneut 65 Milliarden Zeilen usw. usw. Es gibt ein Überangebot an Code der durch Menschen erzeugt wird. Tendenz steigend. Wenn man sich die Weltlage so anschaut und welche Anstrenungen in aufstrebenden Ländern wie China und Indien betrieben werden um mithalten zu können mit den us-amerikanischen Programmierkursen, so ist anzunehmen, dass sich die Menge der weltweit verfügbaren Programmierer weiter steigert. Und die werden ebenfalls ihre täglichen 10 Zeilen Code schreiben wollen.

2 Terabyte an Code Bisher waren die Zahlen doch noch sehr abstrakt. Versuchen wir den Codeumfang einzugrenzen: 18 Mio Programmierer schreiben je 10 Zeilen Code täglich. Macht umgerechnet 2,6 Terabyte an jährlichem neuen Code. Viel ist das nicht, im Grunde reicht es wenn github pro Jahr eine neue Festplatte anschafft um den eintreffenden Sourcecode darauf abzulegen. Nur, diese Codemenge ist mehr als ausreichend egal welche Großprojekte man benötigt. Es reicht aus, um Windows jedes Jahr from scratch zu programmieren. Also den vorhandenen Code komplett zu löschen und nochmal neu beginnen. Gleiches gilt für Computerspiele und OpenSource Software. Diese Menge an 2,6 Terabyte neuem Code jährlich ist eine feste Größe. Sie lässt sich nicht beeinflussen sondern sie ist einfach da. Sie ist genauso konstant wie die Zahl der veröffentlichten Twittermeldungen und entstammt aus einem inneren Bedürfniss heraus.

Und jetztnochmal zurück zur Eingangsfrage wie man mit OWL einen automatischen Codegenerator erstellt. Die Ausgangslage die bei OWL fäschlicherweise angenommen wird lautet, dass es keine natürlichen Quellen für Sourcecode gibt. Es also weltweit keinen einzigen gibt, der wüsste wie man Java Code in einen Editor eintippt. Das heißt, das Szeanario lautet dass entweder das Programmieren noch nicht erfunden wurde, oder dass nach einem Atomkrieg alle Programmierer getötet wurden. Und jetzt benötigt man eine Maschine oder ein Verfahren was alleine programmiert. Die also auch dann noch Code generiert, wenn es keine Menschen mehr gibt. Macht ein solches Szenario Sinn? Und ist es realistisch? Wohl kaum, es ist komplett hypothetisch und wird der Wirklichkeit nicht im mindesten gerecht. Insofern muss man der OWL Community unterstellen, dass sie als Nabelschau vor allem mit sich selber beschäftigt. Also versucht einen Diskurs zu dominieren um darüber Macht auszuüben. Motto: menschliche Programmierer sind nicht in der Lage Code zu schreiben weil das zu aufwendig ist und dabei zu viele menschliche Fehler passieren. Genau diese Unterstellung ist falsch. Menschen sind ausgezeichnete Codegeneratoren, sie erzeugen viel und qualitativ hochwertigen Javacode. Und sie tun dies nach wiederholbaren Abläufen, also jedes Jahr erneut. Im Laufe der Jahre sind Menschen darin sogar besser geworden. Die Anzahl der Programmierer ist höher und die geschriebenen Zeilen täglich steigt leicht. Man muss sich mal die Dimensionen auf der Zunge zergehen lassen. Wenn der einzelne Programmierer anstatt 10 Zeilen täglich plötzlich 11 Zeilen am Tag schreibt, dann nimmt dadurch der globale Output an Sourcecode gleichmal um locker 10% zu.

Anders formuliert, ich wage mal die These dass ein beseitigter Bug in Ecilpse der dazu führt, dass Programmierer ihre Produktivität nur minimal erhöhen dazu führt, dass jährlich zigtausende Codezeilen zusätzlich erzeugt werden, einfach so aus dem Nichts heraus. Und das durch ein gut gemachtes Einführugsvideo in die Programmiersprache Python sich gleichmal weltweit 100 Leute mehr fürs Programmieren interessieren.

10 Lines of Code Die obige Überschlagsrechnung basiert auf der Annahme, dass Programmierer täglich 10 Zeilen Code schreiben, also im Jahr auf 3650 kommen. Diese Zahl mag überraschen. Die einen werden sagen, sie ist zu hoch, die anderen sagen dass in Wahrheit viel mehr Code geschrieben wird. Fakt ist jedoch, dass ein menschlicher Programmierer die Zahl von 10 LoC pro Tag gut erreichen kann. Und zwar der Durschnittsprogrammierer. Der also sich nicht besonders anstrengt aber auch nicht komplett dem Rechner fernbleibt. Man muss davon ausgehen, dass es sich um eine Konstante handelt. In der Art, dass der einzelne Programmierer 10 Zeilen täglich schreibt und 100 Programmnierer folglich 1000 Zeilen täglich erzeugen. Was mit diesen Code passiert ist vielfältig. Er wandert vielleicht zu github, fließt hinein in kommerzielle Projekte, wird verworfen oder füllt nur die lokale Festplatte. Was man jedoch sagen kann ist, dass Menschen grundsätzlich in der Lage sind, Computercode zu erzeugen. Es variert je nach den Fähigkeiten, des Alters und des sozialen Status aber grundsätzlich ist Programmieren etwas wozu Menschen in der Lage sind.

Computercode ist nicht irgendeine Ausdrucksform wie Musik oder Prosa, sondern Computercode ist formalisiertes Wissen. Es beschreibt Abläufe in einer expliziten Form. Man kann Computercode unendlich oft ausführen. Einmal erstellter Code kann über Jahre hinweg Maschinen steuern. Wenn man zusätzlich noch über OpenSource die Verbreitung von Code und ganz wichtig von Tutorials fördert in denen erklärt wird wie man programmiert, potenziert sich

der Effekt. Man kann den jährlich erzeugten Code als eine Art von Macht definieren. Etwas das ein Eigenleben führt und das Ergebnis des weltweiten Programmier-Kollektiv darstellt. Es handelt sich um eine Crowd, die keinen Text sondern Sourcecode erzeugt. Dieser Code wird es sein, der die nächste Revolution verursacht. Es ist etwas, was sich nicht kontrollieren lässt. Es ist einfach da und zwar in fast unbegrenzter Menge.

Die Schwierigkeit besteht darin, dass es nicht vorhersehbar ist, was man mit Computercode grundsätzlich machen kann. Das man damit IBM kompatible PCs steuern kann ist allgemein bekannt. Aber vermutlich lässt sich mit Computercode auch Roboter steuern, Krankheiten heilen und noch einiges mehr. Als künstliche Intelligenz wird üblicherweise bereits geschriebener Code bezeichnet der sinnvolle Dinge tut. Das also die Maschine nach dem Einschalten etwas macht was sonst nur Mehschen können. Doch eigentlich beginnt es viel früher. Weil jeder Code zuerst einmal geschrieben werden muss. Und das jährliche Quantum an neu erstelltem Code funktioniert unabhängig von Künstlicher Intelligenz. Der Code hat teilweise etwas mit AI Problemen zu tun, sehr viel Computercode behandelt aber auch Aufgaben wie Betriebssystembiblitheken oder Geschäftsprozesse. Die Urheber dieses Codes sind klar zu bennenen. Es ist biologisch getrieben und ist angesiedelt in den 7 Milliarden Menschen die die Erde bewohnen. Man könnte sagen, die Welt funktioniert wie ein riesiger organischer Supercomputer der Sourcecode generiert.

Codegeneratoren Jetzt nochmal der Rückblick auf die eingangs erwähnten Codegeneratoren. Mag sein, dass es einige funktionierende Beispiele gibt, wo Software neue Software geschrieben hat. Jedenfalls wird das in den Papern behauptet. Und mag sein, dass im Bereich General Game Playing Bemhungen gibt, Algorithmen zu erfinden die lernen können. Doch mit den natürlichen Ressourcen aus echten Programmierern die schon da sind, können diese Ansätze nicht konkurrieren. Das was eine Software an Code schreiben kann ist nichts im Vergleich zur Qualität und zur Menge welche die Crowd aus den 18 Mio weltweiten Programmiern erzeugen kann. Und mehr noch,meine Empfehlung lautet, Projekte einzustellen die automatische Codegeneratoren anstreben und stattdessen komplett auf Human-Intelligence zu setzen. Diese ist weitaus effektiver.

Skynet In dem Film “Terminator II” wurde an einer Stelle erläutert, dass Skynet als automatisches System entwickelt wurde was alleine lernt. Demnach wäre Skynet ein Codegenerator, also ein Computerprogramm was im Stande ist seine eigene Beschreibung anzupassen. Das ist erstens technisch unmöglich und zweitens blendet es auch die Wirklichkeit aus. In Wahrheit erzeugen nicht Computer ihren Code, sondern es sind Menschen die programmieren. Und wenn es irgendwann in Zukunft zu Problemen kommen sollte mit irgendwelchen Robotern die Krieg spielen so steckt mit Sicherheit kein Supercomputer aka Skynet dahinter, sondern wer sich das ausgedacht hat, dürfte auf 2 Beinen herumlaufen und Pizzaschachteln auf dem Schreibtisch stapeln. Kurz gesagt, sollte es wirklich ein Problem mit Sourcecode geben, dann wird die Ursache durch-Menschengenerierter Code sein. Also Programmzeilen die irgendwer der 18 Mio Programmierer sich ausgedacht hat und was Stress verursacht.

Ja mehr noch, wenn immer jemand behauptet dass der Computer von allein etwas gemacht hätte oder sogar seinen eigenen Code so umschreibt dass er böse Dinge tut, kann man mit hoher Wahrscheilichkeit davon ausgehen, dass es sich um Desinformation handelt. Viren, Würmer und trojanische Pferde werden nicht durch Codegeneratoren erzeugt sondern von pubertierenden männlichen Jugendlichen welche unter mangelndem Selbstwertgefühl leiden.

5.4 Sprachen zur Domainmodellierung

Es gibt eine Reihe von Sprachen um Domain-Modellierung zu unterstützen: KQML (knowledge query and manipulation language) und Unified Problem-solving Method Development Language (UPML) sind zwei davon. Beide Sprachen sind Framebasiert, das Wissen wird auf ähnliche Weise abgespeichert wie in einem UML Diagram. Leider sind beide Sprachen nicht mächtig genug. Sie sind nicht die erhoffte Zwischensprache zwischen Maschinenlesbaren Algorithmen und natürlichen Sprachen. Schauen wir uns die beiden Extreme etwas genauer an. Auf der einen Seite gibt es klassische Computersprachen wie Python und Java. Und auf der anderen Seite gibt es Wikihow Texte in denen Domänenspezifisches Wissen abgelegt ist. Kann man hier noch eine Sprache dazwischen formulieren, die beides kann? Leider nein. Was man jedoch versuchen kann ist eine ontologiebasierende semantische Websuchmaschine zu erfinden. Damit ist gemeint, dass vorhandene natürlichsprachliche Texte indiziert und katalogisiert werden, also mit Semantictagging aufbereitet werden. Und jetzt kann man an diese Datenbank Anfragen senden wie z.B. die Frage “Was ist ein Haus?”. Wie abstrakt man die Anfragen formulieren kann hängt davon ab, wie gut die semantische Suchmachine arbeitet. Im einfachsten Fall kann man lediglich per Volltextsuche in den Dokumenten suchen, also so wie es der UNIX grep Befehl kann. Ist die Suchengine leistungsfähiger wird die Anfrage zunächst geparst und mittels Ontologien näher analysiert.

Ich will damit sagen, dass sich Domain-Knowledge nach wie vor am besten in natürlichsprachlichen Texten speichern lässt, also in einer der großen Weltsprachen wie English. Computersprachen hingegen sind gut geeignet wenn man diese Informationen auswerten möchte.

6 Fahrassistenzsystem

6.1 Einparkhilfe programmieren

In einem schlauen Paper aus dem Bereich der Optimierung / Künstliche Intelligenz war einmal ein Solver abgedruckt, der von allein ein Auto in eine Parklücke befördert hat. Die Idee lautete, das ganze als mathematisches Problem zu verstehen, ähnlich wie das Piano Movers Problem. Demzufolge stellt man einen Gametree auf mit möglichen Aktionen und durchsucht diesen nach dem gewünschten Zielzustand. Obwohl das Paper in sich schlüssig klang, und es häufig zitiert wurde, ist das jedoch nicht die Lösung. Was man stattdessen tun sollte, ist eine Software zu programmieren mit dem man Menschen beibringt einzuparken.

Im Computerspiel “Driving School 3D” ist ein solcher Assistent enthalten. Im einfachsten Level #1 muss der Spieler einfach nur geradeaus fahren und in einer gekennzeichneten Box anhalten (ohne Lenken, ohne Rückfährtsfahren). Wenn er das geschafft hat, erhält er einen Pluspunkt. In Level 2 lobt einen der Fahrlehrer wenn man zwei Halteboxen nacheinander befährt und jeweils darin stoppt. In Level 3 sind die Parkboxen dann im Winkel von 90 Grad angeordnet, und so steigt der Schwierigkeitsgrad immer weiter an. Bis irgendwann die Königsdisziplin ansteht wo man rückwärts einparkt, mit mehrmaligem Umsetzen und Radkorrektur.

Das interessante am o.g. Computerspiel ist, dass es dort keine Automatik-Funktion gibt. Vielmehr soll ja der Spieler manuell die Aufgabe lösen. Gleichzeitig besitzt die Software ein umfangreiches Wissen über Parklücken, kann also theoretisch die Aufgabe auch autonom ausführen.

Ein wenig erinnert das an die Theorie des “Lernen durch Lehren”. Wenn man anderen erklären kann, wie sie einparken sollen und zwar dezidiert auf Anfängerniveau, dann kann man es auch selbst. Was vielleicht noch ganz wichtig am Spiel ”“Driving School 3D” ist, dass

man dort nicht irgendwo das Einparken übt, sondern ausgerechnet auf einem Parkplatz wo Polizeifahrzeuge abgestellt sind. Ob das eine gute Idee ist? (Update: das scheint aber so eine Art von Standard zu sein, auch bei anderen Fahrsimulatoren kann man das Parken auf einem Polizeiparkplatz üben). Scheinbar legen die Konsumenten solcher Spiele auf dieses Feature ganz besonderen Wert. Es gibt sogar ein Spiel was dezidiert "3d Police Car Parking" im Namen trägt. Dort ist die Parklücke nicht einfach nur im Boden markiert, nein sie strahlt bis in den Himmel hinein, so als wäre das ganze das 8. Weltwunder.

6.2 Domänenwissen in Sourcecode konvertieren

Das Domänenwissen also das Expertenwissen darüber wie man Auto fährt ist natürlichsprachlich gespeichert. Es liegt bei Wikihow (wo es mehrere Tututorials gibt wie man die Vorfahrt beachtet), es liegt in Prüfungsbögen von Fahrschulen gespeichert und in vielen schlauen Büchern die man bei Amazon kaufen kann. Doch wie bekommt man diese Tutorials welche sich an Menschen richten und aus Text sowie Bildern bestehen in ein lauffähiges Computerprogramm übertragen? Das ist schonmal die richtige Frage, weil genau darum geht es bei der Entwicklung eines autonomen Fahrzeuges. Die schlechte Nachricht vorneweg: einen DeepLearning Algorithmus der sich von alleine durch die Texte frisst und dann die fertige Software generiert habe ich nicht anzubieten. Dafür jedoch einige Überlegungen zu der Thematik wie man es teilautomatisiert durchführen kann.

Zu Beginn sei auf den Flaschenhals verwiesen der beim manuellen Programmieren nicht wegzudiskutieren ist: der Mensch. Ein Durchschnittsprogrammierer erzeugt pro Tag rund 10 Lines of Code, im Jahr also 3650 Zeilen Code. Eine gut programmierte Behavior Engine welche die Feinheiten des Autofahrens beherscht braucht jedoch mehr. Schätzungsweise 1 Mio Lines of Code vielleicht sogar mehr, wenn sie auch noch LKWs rückwärts einparken soll. Und jetzt möchte ich zu den Dingen kommen die schon bekannt sind. Die beste Methode um komplexes Expertenwissen in Computercode zu speichern ist objektorientierte Programmierung. Das heißt, die fertige Behavior Engine besteht aus unzähligen C++ Klassen die als UML Diagramm angeordnet eine Art von Mindmap der Domäne darstellen. Die Klassen sind im UML Schema angeordnet und weiter in Packagess untergliedert. Wenn man davon ausgeht dass die fertige Engine aus 1 Mio Lines of Code besteht und jede Klasse aus 200 Zeilen Code besteht macht das Adam Riese die Anzahl von 5000 Klassen die es schreiben gilt. Beispielsweise eine Klasse für das Auto, eine für die Straße, eine für die Trajektorie, eine für das Vorfahrtsschild, eine weitere für Fußgänger und immer so weiter bis man die komplette Szenerie im Hauptspeicher vorliegen hat.

Jetzt stellt sich die Frage ob man die Domainmodellierung weiter formalisieren kann. Ja man kann. Und zwar gibt es zwei Teilprobleme. Einmal die Modellierung als solche und einmal eine Testfunktion um zu überprüfen ob die Modellierung korrekt ist. Die Testfunktion ist einfacher zu programmieren, sie ist gleichbedeutend mit einem Punktestand in einem Computerspiel. Sie muss nicht echtes Domänenwissen enthalten sondern lediglich erkennen ob eine Aktion richtig oder falsch war. Vielleicht ein Beispiel:

Wenn man in einer Trafficsimualtion einen menschlichen Fahrer herumcruisen lässt wird im Head-up Display in Realtime angezeigt wieviele Fehler er schon gemacht hat. Fährt er zu dicht auf den Vordermann auf gibt es einen Minuspunkt, fährt er bei Rot über die Ampel einen weiteren, verlässt er seine Fahrspur wieder, und fährt er zu schnell gibt es weitere Minuspunkte. Softwaretechnisch muss man unterscheiden zwischen einer Software die obige Minuspunkte berechnet und einer Software die das Fahrzeug steuert und die Null Fehler macht. Technisch gesehen ist der Modellchecker leichter zu programmieren. Ich würde mal schätzen dass rund 20% der Codezeilen ausreichend sind, von 1 Mio LoC also 200000. Und das ist eher hoch angesetzt, vermutlich kommt man auch mit 10% aus. Ein weiteres ´Teilgebiet des Domain-Modells ist ein Head-Up Display, was schätzungsweise 60% des Gesamtcodes ausmacht. Hier mal die Übersicht:

1. Domainchecker, Berechnung des Punktestand: 20% vom Code
2. Head-up Display zur Unterstütung von manueller Fahrweise, 60% vom Code
3. eigentliche Künstliche Intelligenz zur autonomen Steuerung, 20% des Codes

Das sind natürlich nur Richtwerte, sie zeigen woraus das Problem besteht und wieviel Aufmand man im Detail leisten muss.

6.3 Trafficsimulation bei github

Unter dem Stichwort Trafficsimulation weiß github derzeit 75 Projekte aus. Um etwas Licht in die unterschiedlichen Ansätze zu bringen, hier eine längere Tabelle mit folgenden Spalten: ID, URL, Lines of Code, Programmiersprache, Grafikengine, Anmerkung.

Inzwischen hatte ich Gelegenheit mir den Sourcecode etwas genauer anzuschauen. Die Projekte die komplexer sind, verwenden allesamt C# als Programmiersprache und Unity als Gameengine. Der Grund ist, dass es zu Unity bereits ein Addon gibt, mit dem man seine eigene Trafficsimulation inkl. Straßen und Autos zusammenbauen kann. Man muss dann nur noch etwas Glue Code schreiben und erhält dann eine vollausgewachsene Trafficsimulation inkl. realistischer Grafik, Abbiegen an Kreuzungen und Spurwechsel.

Anfangs hatte ich bei github nach fertigen Projekten gesucht in der Hoffnung dort entweder selbst Code zu commiten oder bereits erstellten Code erneut zu verwenden. Leider funktioniert beides nicht. Im Regelfall ist der Sourcecode sehr auf das jeweilige Projekte zugeschnitten. Zu meinem eigenen Projekt gibt es zwar überschneidungen aber nicht so, dass man den Code kombinieren könnte. Das hat nicht nur etwas mit einer anderen Programmiersprache zu tun, sondern mit dem Domainmodelling. Sowohl in meinem Projekt als auch in den Projekten die in der Tabelle aufgeführt sind, wird eine Technologie namens Scripting AI in Kombination mit Objektorientierter Programmierung eingesetzt. Das heißt, es gibt eben keine Meta-Solver die aus einer PDDL Datei etwas ableiten, sondern der Sourcecode wird sehr detailiert auf die Aufgabe zugeschnitten. Benötigt man ein Attribut Geschwindigkeit wird das in die Klasse eingefügt, gleiches gilt für Behaviors oder dem Wendekreis. Es gibt gegenwertig keine Technologie um das Abstraktionsniveau anzuheben, also eine Behavior Engine zu konstruieren die man wiederverwenden kann. Böse formuliert ist sowohl mein eigenes Projekt als auch die anderen so programmiert worden, wie man einen AutoIt Bot schreibt. Also munter drauflos und schön bottom up.

Was sich jedoch wiederverwenden lässt ist die Dokumentation zu den Projekten. Bei Projekten die eine wissenschaftliche Arbeit beinhalteten lässt sich daraus sehr viel ableiten. Meistens ist dort ein UML Diagramm zu sehen und man kann ungefähr erkennen um was es ging. Eigentlich ist also der wissenschaftliche Text das eigentliche Highlight und weniger der Sourcecode. Aber, solche Texte kann man nur schreiben wenn man zuvor ein lauffähiges Projekt durchgeführt hat, also konkreten C# oder Python Code erstellt hat. Weil sonst die Arbeit sehr nichtssagend wird.

Aufgefallen bei der Durchsicht der Projekte ist mir, dass sich verglichen mit sonstigen Programmierprojekten sehr überschaubar sind. Das durchschnittliche github Trafficsimulation Projekt besteht aus 2000 Zeilen Code, die Spitzenreiter bringen es auf 5000 Zeilen. Meine eigene Prognose lautet, dass man eher mit 1 Million Lines of Code

ID	github URL	Lines of Code	Programmiersprache	Anmerkung
1	kevjohnson/TrafficSimulation	561	Python	
2	Nick-Th/TrafficSimulation	61	Python	
3	razabu/TrafficSimulationASM		Assembly	
4	zdsbs/trafficsimulation	296	Clojure	
5	schoerg/TrafficSimulation	960	C#	umfangreiche Dokumentation, zip Datei ist 120 MB groß und enthält Unity Assets, verwendet colllider events
6	johannawad/TrafficSimulation	3148	C#	
7	daria-mih/TrafficSimulation			identisch mit id6
8	eli88p/TrafficSimulationSCE	1154	C#	
9	pontusarfwedson/TrafficSimulation	961	C++	
10	kongo555/TrafficSimulation	1057	C++	SFML
11	hwilliams11/TrafficSimulation	3513	Java	
12	Soniaroxana/TrafficSimulation	1724	Java	
13	polwel/traffic_simulation	2361	C++, Python	inkl. wissenschaftlichem Paper, verwendet Intelligent Driver Model (IDM)
14	JAFS6/Traffic-Flow-Simulator	5976	C#	University project, Unity, inkl. wissenschaftlichem Paper

Tabelle 2: Trafficsimulation bei github

rekursive Lines of Code ermitteln: find . -name '*.cs' | xargs wc -l

rechnen muss für eine funktionierende Trafficsimulation. Das ganze ist also eine reine Fleißaufgabe. Die Programmiersprache ist dabei egal, vorzugsweise natürlich eine objektorientierte Programmiersprache also eine der großen Drei: Java, C# oder Python. Alles sind Virtual Machine basierende OOP Sprachen mit denen sich komplexe Simulationen erzeugen lassen.

6.4 Event-based Trafficsimulation

Eine Trafficsimulation ist eine Art von Computerspiel. Üblicherweise funktionieren diese Time-basiert. Das heißt es gibt eine Game-Loop in der ein Framecounter hochgezählt wird und passend dazu bewegen sich die Objekte. Die Fokussierung auf Zeit als zentrales Ordnungskriterium hat etwas mit der Computerdarstellung zu tun. Ein Computerspiel besteht aus 30 fps, in Folge dessen muss man die Bilder als kleinste Einheit betrachten. Aus Sicht einer Trafficsimulation ist jedoch ein zusätzlicher Event-Queue sinnvoll. Als Event werden semantische Ereignisse bezeichnet wie Kollision mit einem Objekt, ein Mausklick oder das Erreichen einer roten Ampel. Es ist sinnvoll neben der zeitbasierten Game-Loop noch eine dezidierte Event-Loop einzufügen, wo eine Collider Ontologie auf die Game-Objekte angewendet wird.

Eine korrekt verwaltete Event-Loop wird als Ausgangsbasis verwendet um darauf aufbauende Scripte zu schreiben. Wenn bekannt ist, dass das Auto aktuell an einer roten Ampel ist, kann man darauf reagieren und die Geschwindigkeit reduzieren oder Alternativ eine Meldung ausgeben, dass man anhalten soll. Man kann so vom eigentlichen Spiel abstrahieren, es wird zwischen Event und darauffolgender Reaktion unterschieden. Es wird egal, wo genau ein Auto gerade auf der Map ist, wichtig sind nur die Events die gerade eintreffen.

Ontologie Es bleibt noch zu klären wie man Events in einer Programmiersprache darstellt. Eine Möglichkeit ist eine auf OWL basierende Ontologie [30]. OWL jedoch ist relativ komplex, man kann auch verschiedene Klassen erstellen. Im einfachsten Fall kann man auf eine Ontologie sogar verzichten und die Events als Array oder Dictionary im Speicher halten.

7 NASA

7.1 NASA: Auftanken in der Luft

Sehr intensiv hat sich die NASA mit Head-up Displays beschäftigt. Es gibt auf Google Scholar sogar Paper die zurückreichen bis in die 1970'er Jahre wo angefangen wurde mit dieser Technik zu experimentieren. In jühgerer Zeit wurde beispielsweise ein System implementiert was Auftanken in der Luft ermöglicht [10] Auf Seite 10 des Papers kann man gut den Übergang zwischen einem manuellen und autonomen System sehen. Dort ist ein Screenshot des Head-up Displays abgebildet. Also das Videosignal mit überlagerter Objekterkennung. Das ganze diente ursprünglich dazu, dass der Pilot damit leichter an die fliegende Tankstelle andocken kann, lässt sich aber auch zur späteren Entwicklung eines autonomen Systems einsetzen. Mit derartigen Head-up Display Implementieren gelingt die Domänenmodellierungen. Es können Heuristiken für die konkrete Aufgabe ermittelt werden. Derartige Modelle sind teilautonom. Das heißt, bestimmte Funktionen werden durch Computerprogramme ausgeführt, beispielsweise die Objekterkennung, andere Elemente hingegen müssen manuell erledigt werden, wie z.B. das konkrete Steuern das Flugzeuges. Auf Seite 12 des Papers findet sich sogar eine Art von Finite-State-Machine um den Auftank-Prozess zu modellieren. Es ist jedoch keine richtige Finite State-Maschine, weil sie nicht von einem Computerprogramm ausgeführt wird, sondern das ganze ist eine Guideline für menschliche Piloten. Man kann die Abfolge Ready->Trail->Closure->Precontact1->usw. zwar auch automatisiert erledigen, muss es aber nicht. Dadurch beugt man dem Phänomen vor, autonome KI-Systeme zu programmieren die nach dem Einschalten nicht funktionieren.

Einen geschichtlichen Abriss über die Entwicklung bis zum Head-up Display vermag [23] zu geben. Danach gab es zuerst das "Primary Flight Display" (page 92), was aus einer Höhenanzeige, Geschwindigkeitsanzeige und einem künstlichen Horizont besteht, überwiegend als Analog-Instrumente realisiert. Als nächstes wurden digitale Displays entwickelt, die im Zuge von Synthetic Vision um Landkarten erweitert wurden. Das seperate Synthetic Vision Display wurde dann als Head-up Display in die Frontscheibe des Flugzeuges übertragen. Bei Augmented Reality Displays werden die Informationen von mehreren Kameras ausgewertet und um Flugpläne ergänzt. Als State-of-the-art gelten autonome Systeme, wo also ontop der existierenden Augmented Reality Software ein Planner programmiert wird, der die Mission komplett alleine fliegen kann.

Autonome Flugsteuerungen basieren also nicht auf disruptiver Technologie wie neuronale Netze oder neuartigen KI-Algorithmen sondern es handelt sich um die logische Weiterentwicklung der Armaturen, Instrumente und Head-up Displays. Ein autonomes System bedeutet, dass man alle wichtigen Informationen digital vorliegen hat, und die Daten zu höherwertigen Situationsbeschreibungen verdichtet. Im Regelfall erfolgt das domänenspezifisch. Ein Head-up Display für ein Flugzeug ist anders aufgebaut als eines für Autos oder für einen Roboterkran.

7.2 PIXAR: Interactive Animation Control

Bisher wurden Head-up Display allein im technischen Bereich verortet, also bei der Steuerung von Automobilen oder von Flugzeugen. Das gleiche Prinzip lässt sich auch auf Computeranimation übertragen. Computeranimation ist wesentlich freier, was die Themen angeht, es kann einen nützlichen Hintergrund haben, kann aber auch nur der Unterhaltung dienen. Die einfachste Möglichkeit der Computeranimation ist handgesteuert. Das wird im klassischen Puppentheater angewendet wie auch wenn man Figuren auf dem Bildschirm bewegt. Der Animateur hat bestimmte Tasten mit denen er die Arme und Beine bewegen kann. Dadurch werden Geschichten erzählt.

Die Frage ist nun wie man mittels Head-up Display diese manuelle Animation erleichtern kann. Im einfachsten Fall durch Pfade. Das heißt, es wird auf dem Bildschirm angezeigt, wohin die Figur sich bewegen soll und der Animateur steuert dann in diese Richtung. Ebenfalls möglich ist es, wenn der Puppenspieler einen Zettel nutzt, auf dem der Plot draufsteht. So sieht er, wo er sich innerhalb der Geschichte befindet und weiß was als nächstes drankommt. Solche Systeme lassen sich in Software abbilden und zu kompletten Animations-Suiten verdichten. Die Firma Pixar waren die ersten, die sowas entwickelt haben.

Die Gemeinsamkeit zu Head-up Displays in Flugzeugen besteht darin, dass mit manueller Steuerung gearbetiet wird. Primär wird die Animation durch Menschen und nicht durch Maschinen generiert. Aber: es wird angestrebt die kognitive Load für den Puppenspieler zu minimieren. Also ihm Hilfsmittel zu geben, um mit weniger Nachdenken die Animation zu erzeugen.

Die Aufgabe besteht darin, ein User-Interface zu schaffen mit dem man interactive Puppetry betreiben kann. Das User-Interface ist die Modellierung der Domäne.

7.3 Die Siggraph Veranstaltung

Mal angenommen, Künstliche Intelligenz ist die König des Fortschritts wo wird dieser Fortschritt am intensivsten ausgelebt? Bei der NASA jedenfalls nicht, die dortigen Paper sind allenfalls Standardkost, die verbauten Computer die in den Weltraum geflogen sind, waren 20 Jahre und älter, ohne dass es erkennbare Vorteile gab. Auch die akademische KI-Forschung ist nicht gerade ein Leuchtfeuer in Sachen Fortschritt. Üblicherweise ist Hochschul-KI Forschung sehr Philosophie-zentriert, siehe das gescheiterte Projekt OpenCOG, was zwar auf mehrere wissenschaftlichen AGI Tagungen einer breiten Öffentlichkeit schmackhaft gemacht wurde, wo aber bis heute nicht klar ist, was die Software eigentlich tut. Auch die Prolog und 5th generation Projekte der 1990'er sind allenfalls Standardkost.

Es gibt jedoch eine Spielart innerhalb der Künstlichen Intelligenz die sehr fortschrittlich ist, und zwar die Grafik-messe Siggraph. Dort werden seit mehreren Jahren neue Entwicklungen im Bereich der Computeranimation vorgestellt. Auch Marc Raibert hat dort früher einmal Vorträge gehalten, und Pixar ist bis heute der Hauptsponsor. Anders als reine Computerspiele-Messen geht es bei Siggraph wissenschaftlich zu, das ganze ist keine reine Verkaufsveranstaltung sondern auch wissenschaftliche Vorträge und ganz wichtig, Paper, spielen eine wichtige Rolle. Das interessante an den Siggraph Problemen ist, dass sie üblicherweise etwas mit Robotik zu tun haben. Will man in einer Virtuellen Umgebung einen humanoiden Charakter lebensecht animieren und mit High-Level-Kommandos steuern, braucht man dafür ein Robot-Control-System, oder zumindest ein Animation-Control-System.

Vielleicht ist Siggraph nicht die weltbeste Veranstaltung in dieser Richtung, aber es ist zumindest die weltweit beste öffentliche Veranstaltung. Das heißt, mit etwas suchen findet man die alten Paper auf Internet-Servern zum freien Download und die vorgestellten Verfahren zeichnen sich durch einen hohen Praxisbezug aus. Das schöne an der Siggraph Verstaltung ist ausgrechnet, dass dort nicht Künstliche Intelligenz im Vordergrund geht. Sondern KI ist dort nur ein Hilfsmittel um Animationsprobleme zu lösen. Das hat dazu geführt, dass nur jene KI-Technologien eingesetzt werden, die wirklich relevant sind für praktische Probleme. OpenCog hätte dort keine Chance, weil man die Software eben nicht zur Computeranimation einsetzen kann. Anders formuliert, wenn ein Thema offtopic für die Siggraph ist, dann ist es sehr wahrscheinlich generell nicht interessant.

Man könnte die Siggraph als eine Art von Durchlauferhitzer für moderne Technologie bezeichnen. Alle wichtigen Namen der Computergeschichte hatten in igendeiner Weise mit dieser Veranstaltung Kontakt. Offenbar ist die Siggraph durch die Verbindung aus Informatik und Grafik, aus Industrie und Wissenschaft, aus Unterhaltung und Ernst der ideale Nährboden für die Weiterentwicklung von angewandter Künstlicher Intelligenz. Ich würde sogar sagen, die Siggraph ist wichtiger als dezidizierte Robotik-Veranstaltungen wie Robotcup oder IROS.

Unbeantwortet ist natürlich noch die Frage, welche der vielen eingesandten Beiträge wirklich relevant sind. Man kann sich ja schlecht alle Paper durchlesen die jemals veröffentlicht wurden. Da muss ich leider passen, die Siggraph ist einfach zu umfangreich und zu viele Jahre schon aktiv, als dass man da konkrete Empfehlungen geben könnte. Was ich zumindest herausgefunden habe, ist dass die Zeitspanne von 1985-1995 relativ wichtig gewesen zu sein scheint. In dem Sinne, dass auch heutige Vorträge inhaltlich dort aufbauen. Zum Vergleich: das war die Hochzeit der LISP Machinen, Pixar wurde gegründet, objektorientierte Programmierung wurde entwickelt, man sprach von der 5. Computergeneration, die ersten Wissensbasierende Systeme wurde programmiert. Meiner Ansicht nach sind spätere Meilensteine in der Künstliche Intelligenz wie DeepLearning ein Rückschritt zu dieser Hochzeit gewesen und konnten an einstige Erfolge nicht anschließen. Aber das ist eine subjektive These.

Scripted Animation Die Siggraph Veranstaltung hat sich weiterentwickelt. Ein aktuelles Beispiel ist ein Paper aus dem Jahr 2007 [24] was in der Einführung einen geschichtlichen Abriss gibt (angefangen von Luxo Jr. von Pixar von Ende der 1980'er, über Simbicon und ähnliche Projekte) um dann auf Seite 31 den aktuellen Stand zu

beschreiben. Zu sehen ist ein humanoider Charakter der in 3D Studio Max modelliert wurde, und der über eine Scripting Sprache gesteuert wird. Man kann an den Charakter Befehle senden wie “walk 2”, “turn left 30” und diese werden dann ausgeführt und erzeugen die Animation. Auf Seite 33 des gleichen Papers gibt es noch ein Beispielprogramm in der Sprache Dynascript was eine komplexere Animation zeigt. Das Prinzip ist ungefähr das, was auch Marc Raibert in seinen frühen Arbeiten eingesetzt hat, allerdings ist es im obigen Paper sehr viel verständlicher dargestellt. Im Grunde bedeutet Computeranimation in der Gegenwart, ein High-Level-Script zu erstellen. Das wird dann in einer Animation-Engine ausgeführt und später erfolgt dann das 3D Rendering.

Der Trick dahinter ist nicht so sehr die verwendete Scripting-Sprache als solche. Diese ist eher simpel aufgebaut, es ist eine Domänenspezifische Sprache mit vordefinierten Befehlen wie walk, turn, grasp usw. sondern es ist vielmehr das dahinterstehende Framework. Also eine Mischung aus 3D Rigging, Spline Animation, Solver, Behaviors und ontop dann die Dyna+ Scripting-Engine.

8 Nouvelle AI

8.1 Einleitung

Im Jahr 1991 hat Rodney Brooks ein wegweisendes Paper veröffentlicht namens "Intelligence without representation" [2] womit er zugleich das Gründungsmanifest der Nouvelle AI verfasst. Es ist eine Richtung innerhalb der Künstlichen Intelligenz Forschung die radikal bricht mit dem was davor. Was genau Nouvelle AI sein soll wurde nur wage angedeutet. Böse formuliert handelt es sich um mechanische Roboter die Bastler in ihrer Gerage zusammenlöten und dort dann selbstgeschriebene C Programme ausführen. Alles sehr unwissensschaftlich, behelfsmäßig und nicht reproduzierbar. Umso wichtiger ist zu erfahren, gegen was sich Brooks abgrenzte, also was davor 40 Jahre lang unter dem Stichwort Künstliche Intelligenz an den Universitäten unterrichtet wurde.

Gold old fashion AI oder auch als Symbolic AI bekannt bedeutet, die Welt abstrakt und in Objekten zu beschreiben. Man hat eine Banane, welche auf einem Tisch liegt. Und ein Mensch kann jetzt die Banane nehmen und damit etwas tun. Um diese Situation in Programmcode zu formulieren wird überlicherweise LISP, Strips und ACT-R verwendet. Das klingt für sich genommen nicht besonders kritikwürdig, warum also machte Rodney Brooks da ein Problem daraus? Nehmen wir mal an, man codiert die obige Sitatuion in eine Programmiersprache hinein und nutzt dafür die von Marvin Minsky (ebenfalls ein Vertreter der Symbolic AI) vorgeschlagene Methode der semantischen Frames. Das so erstellte Programm braucht vielleicht 100 Zeilen Code und man kann es starten. Das Problem ist folgendes: das Programm tut nichts. Es besitzt kein Grounding. Dinge wie Banane, Tisch und “nehme auf” sind nur Textstrings, die als Attribute den Frames zugewiesen sind und ausgegeben werden können, aber das ganze ist nicht mehr als ein Chatbot. Also ein System was so tut als würde es Sprache verstehen.

Versucht man diese Software zu nutzen um damit echte Roboter zu steuern oder ein Computerspiel automatisch zu spielen wird man feststellen dass es nicht geht. Es gibt nicht direkt eine Fehlermeldung auf dem Bildschirm, sondern vielmehr besteht das Problem darin, dass es gerade keine Bugs gibt. Wir haben also einmal die KI-Software welche aus Frames besteht und in LISP programmiert wurde, welche perfekt funktioniert, und dann haben wir die Konkrete Aufgabe wo der Roboterarm etwas tun soll, der ebenfalls mechanisch super funktioniert. Nur, ohne Bugs gibt es nichts zu verbessern, das Projekt steht auf der Stelle. Und genau hier kam Rodney Brooks ins Spiel. Er hatte nicht etwa an den semantischen Frames etwas auszusetzen, sondern er war der Meinung, dass man irgendwas benötigt an dem man sich abarbeiten kann.

Heute würde man sagen, dass die Software gegen etwas ein Grounding benötigt. Also etwas worüber man sehen kann, was die Probleme sind. Ein gutes Beispiel dafür sind Roboterwettbewerbe die von Brooks und anderen veranstaltet wurden. Hat man einen Linefollow-Roboter der eine Linie entlangfahren soll, damit aber komplett überfordert ist und von der Linie abweicht hat man einen sogenannten Issue. Also die Erkenntnis das etwas mit der Software und dem Roboter nicht stimmt. Das macht depressiv, zeigt aber dass die Sache wahr ist. In dem Sinne, dass Wissenschaft so funktioniert, dass Dinge als ungelöst gelten.

Versuchen wir ein wenig näher einzugehen, auf die Symbolic AI. Vom Prinzip her ist der Ansatz die Welt in abstrakten Begriffen zu beschreiben nicht verkehrt. Nur ist das etwas, was in einem Robot-Control-System erst ganz am Schluss möglich ist, also wenn man bereits Sensorwahrnehmung, Lowlevel Action Primitive, Planner usw. implementiert sind. Das heißt, wenn man bereits eine Software hat, die über 2000 Zeilen Code lang lowlevel Berechnungen ausführt, die etwas mit Sinus-Winkel, Steering und pathplanning zu tun haben, kann man als Krönung des ganzen dann in den Zeilen 2000 bis 2100 einen symbolischen Planner einbauen. Also ein System was Befehle implementiert wie “moveto(Banana)”. Die ersten 2000 Zeilen Code dienen dazu, den Roboter zu grounden,also auf lowlevel Ebene manuell zu programmieren und nur der letzte kleine Rest wird dann als Krönung auf sehr abstraktem Niveau abehandelt.

Um eine Roboterprogrammiersprache zu schaffen wo man in quasi-natürlicher Sprache mit dem System kommuniziert muss man sehr viel Vorarbeit leisten, die das genaue Gegenteil von Akademischer Informatik ist und als Hacking, Coding und Ausprobieren bezeichnet wird. Brooks hat diesen Aspekt des Adhoc Herumspielen in den Mittelpunkt gebracht.

Ein wenig kann man den Diskurs vergleichen mit einer anderen Spielart der akademischen Informatik. Softwareentwicklung. In den Vorlesungen an den Unis wird unterrichtet, dass man topdown vorgehen soll. Also im Schritt 1 das UML Diagramm zeichnet, mit diesem dann Java Klassen erzeugt und so schrittweise zum fertigen Programm gelangt. Was jedoch richtige Programmierer tun ist den umgekehrten Weg zu bestreiten. Sie programmieren zuerst das Java Programm und ganz am Ende wird dann rückwärts daraus das UML Diagramm erzeugt. Es geht also um die Frage welche Reihenfolge man verwendet und was man als erstes angeht.

Wenn man die Texte von Brooks etwas genauer ließt unterstellt er seinen Fachkollegen von der GOFAI Fraktion nichts geringeres als mythisches Denken. Das sie also Künstliche Intelligenz als Projektionsfläche verwenden um ausgedachten Unsinn in die Welt zu blasen und darüber dann ihre Weltanschauung zu formulieren. Brooks fragt mehrmal polemisch, was denn die KI Forschung all die Jahre an verwertbarer Software zu Tage gefördert hat. Und recht hat er. Wenn man eine Software wie ACT-R startet passiert exakt gar nichts. Das gleiche gilt für angeblich so wundervolle Eliza Program von Weizenbaum, über den LISP Interpreter sowie für die General Problem Solver. Allesamt hundertfach zitierte Projekte, ohne jede praktische Bedeutung. Selbst Shakey the robot ist bei Lichte betrachtet mythisch aufgeladen. In den dazugehörigen Erklärvideo erzählt eine Stimme aus dem Off, was Shakey gerade denkt und wie er lernt eine neue Aufgabe zu lösen. Natürlich ist das Quatsch, der Roboter lernt überhaupt nichts, die Software ist komplett unterentwickelt.

Auch die Roboter die Rodney Brooks gebaut hat, sind weit davon entfernt konkrete Probleme zu lösen. Nach den Bildern zu urteilen konnten seine Walker gerademal über ein Buch drüberklettern. Der Unterschied ist der, dass sich Brooks dessen bewusst war, weil er den passenden C-Code selber gehackt hat und weiß dass er Bugs

enthält. Und Brooks macht sich nicht einmal die Mühe in seine Maschine irgendwas hineinzuinterpretieren, sondern für ihn ist es einfach nur lausige Hardware und schlechte Software. Damit ist Brooks in der Lage seine eigene Grenzen viel besser zu erkennen.

In dem Text [2] vergleicht Brooks die AI Forschung mit der Artifical Flight Forschung der 1890'er Jahre. Vermutlich eine Anspielung auf Otto Lilienthal, der um 1891 herum zum ersten Gleitflug ansetzte und noch keine Ahnung hatte von Flugzeugen. Der Text von Brooks ist demnach der Versuch, Künstliche Intelligenz von einem Engineering Standpunkt aus zu bewerten. Also selbstkritisch zu bilanzieren wo die Schwächen sind und dann versuchen, das Flugzeug in die Luft zu bekommen.

8.2 Intelligent Tutoring Systems

Rodney Brooks hat die Frage unbeantwortet gelassen, wie die Umgebung in den Roboter hineinkommt, also wie der Ablauf ist um Robotersoftware bottom up zu erstellen. Natürlich kann man sie programmieren, aber womöglich gibt es auch eine Abkürzung dazu. Eine Ontologie um das Verhalten eines Roboters zu beschreiben wäre hilfreich dabei. Solche Ontologien wurden im Rahmen von "intelligent Tutoring Systemen" entwickelt, allerdings unter einem anderen Vorzeichen. Normalerweise werden solche Systeme programmiert um die Ausbildung zu unterstützen, also um Menschen etwas zu erklären. Im Ergebnis sind solche Systeme jedoch nichts anderes als eine maschinenlesbare Ontologie der Domaine. Wenn man ein Java Programm womit normalerweise Fahrschüler trainiert werden, muss man nichts weiter tun als daraus die Klassen zu extrahieren und man erhält eine vorzügliche Ontologie um einen Roboter zu steuern.

Wie man Intelligent Tutoring Systeme programmiert, darüber gibt es unterschiedliche Meinungen. Es wurde im Laufe der Zeit viele Authoring Systeme vorgestellt. Doch im wesentlichen dürfte es darauf hinauslaufen, dass ein Tutoring System ähnlich erstellt wird, wie ein Computerspiel auch, also schön mit manuell erstelltem Code der von einem Entwicklerteam verwaltet und verbessert wird. Das Erstellen soll uns jedoch nicht groß aufhalten, weil im Laufe der Zeit bereits sehr viele solche Systeme erstellt wurden, es gibt sie entweder als kommerzielles Produkt oder sogar als OpenSource zum Download. Und sie alle besitzen eine mehr oder wenige hochentwickelte Klassenstruktur, wo also der C++ bzw. der Java Sourcecode thematisch unterteilt ist, Methoden/Attribute enthält und eine Ablauflogik. Das heißt, wenn es irgendwo eine Software gibt mit der man lernt ein Mikroskop zu benutzen, dann ist in dieser Software zwangsläufig irgendwo eine Klasse für das Mikroskop eine für das Präparat und eine weitere um zu beschreiben wo der Ein/Ausschalter ist. Ohne diese Informationen könnte das Intelligent Tutoring System schwerlich Feedback geben und erklären wie so ein Mikroskop bedient wird.

Anders formuliert, man muss keineswegs bei 0 anfangen wenn man eine Ontologie benötigt um Roboter zu steuern und man muss diese Ontologien auch nicht aus natürlichsprachlichen Texten extrahieren sondern kann sich an dem orientieren was im Bereich Edutainment / Computer based Training bereits programmiert wurde.

Laut Google Scholar gibt es immerhin 41 Paper in denen von einer Hochzeit zwischen Rodney Brooks und "intelligent tutoring system" fabuliert wird, also eine Synthese aus künstlicher Intelligenz einerseits und Computerunterstützten Unterricht für Menschen. Mein Eindruck ist, dass das Feld des Iintelligent Tutoring Systems schon sehr gut ausgeleuchtet ist. Bisher war lediglich die Übertragung der dort erstellten Ontologie in Robotik-Anwendungen noch nicht weit verbreitet. Weil, oberflächlich betrachtet sind das zwei sehr weitauseinanderliegende Bereiche. Bei dem einen geht es darum, Software zu schreiben mit denen Menschen etwasa anfangen können und bei der Robotik geht es um autonome Systeme, also wo Menschen gerade nicht in-the-loop sind. Die Gemeinsamkeit wird als Domain-Model bezeichnen. In beiden Fällen benötigt man eine manschinenlesbare Ontologie welche die Umwelt im Detail beschreibt. Also die Regeln des Spiels, mögliche Aktionen und Parameter auf die man achten muss.

Beispiel Bei github gibt es ein Textadventure was in Java geschrieben wurde, und wodurch der Spieler lernen soll, sich in finsteren Dungons gegen Monster zu kämpfen[7] Der dazugehörige Javacode wurde bereits erstellt, das Spiel ist mehr oder weniger fertig. Wenn man sich den Sourcecode anschaut finden sich dort Klassen für die verschiedenen Monsterarten, Positionsangaben auf der Karte und mögliche Items die man aufnehmen kann. :Wenn man das Java Programm in eine UML Datei verwandelt hat man eine hübsche UML Datei mit einer vollständigen Domainenbeschreibung. Inkl. der möglichen Aktionen die man als Spieler ausführen kann. Im Grunde ist das die Ontologie die auch Roboter benötigen, wenn sie autonom agieren sollen. Man kann die Ontologie verwenden um daraus eine Künstliche Intelligenz zu basteln.

Offen ist noch die Frage wie man einen UML Chart dazu verwenden kann, Entscheidungen eines Roboters zu erzeugen. Der erste Schritt besteht darin, zunächst einmal die Ansprüche zu reduzieren und nicht gleich wichtige Entscheidungen zu planen, sondern lediglich verstehen zu wollen wie die Szene funktioniert. Also einen Parser zu entwickeln der ausgehend von der bekannten Ontologie und der Ist Situation eine Beschreibung abgibt, was gerade im Spiel stattfindet. Eine Ontologie für die Modellierung von Spielen ist vergleichbar mit einer Grammar die von Parsergeneratoren eingesetzt wird. Es dient dazu die möglichen Aktionen in ein Schema einzufügen.

Die Antwort auf diese Frage liefert ironischerweise der UML Standard. Und zwar wenn man ihn rückwärts anwendet. Also nicht von einem Use-Case Diagramm den Code erzeugen will, sondern aus dem oben verlinkten github Projekt rückwärts das dazugehörige Use-Case Diagramm generiert. Weil, der erstellte Java Code ist im oben verlinkten Projekt sehr ausführlich. Alle wesentlichen Informationen sind darin enthalten. Er ist deshalb so gut geraten, weil er eben nicht automatisiert erzeugt wurde, sondern weil der Hauptautor und viele Commiter an dem Projekt mitgearbeitet haben, und das Spiel im Echtbetrieb verbessert haben.

Die Frage wie man aus vorhandenem Code rückwärts ein Use-Case Diagram erzeugt ist unüblich, stößt auf Erstaunen wird aber durchaus ernsthaft diskutiert [8] Laut dem Codecrunch Forum ist "Rational XDE " ist in der Lage rückwärts zumindest ein Sequenz-Diagramm zu erzeugen. Warum das trotz allem sehr ungewöhnlich ist, kann man erläutern wenn man sich anschaut wie normalerweise UML und dessen Ableger thematisiert werden.

Üblicherweise geht es um die Frage wie man from Scratch Software entwickelt, also Wissensengineering betreibt. Der übliche Ablauf ist so, dass sich der Programmierer mit den Domain-Experten unterhält, UML Diagramme zeichnet und zunächst einmal die Anforderungen aufnimmt, daraus entwickelt er dann in einem agilen Verfahren nach und nach die Software, und passt währenddessen seine Diagramme an. Es wird also für eine unscharfe Domäne ein Computerprogramm erzeugt.

Nun, wenn man aber die Java Software schon vorliegen hat, und sie bereitss mehrfach getestet und für gut befunden wurde, kann man sich den Domain-Modelling-Prozess sparen. Es geht also nicht mehr darum, die Domäne zu verstehen, sondern es geht darum, aus einer in Java codierten Ontologie einen Bot zu entwickeln. Also etwas, was das Domain-Modell verwendet um Entscheidungen zu treffen, oder

[7] https://github.com/Progether/JAdventure

[8] https://coderanch.com/t/99409/generate-Case-diagrams-automatically-java

Stufe	Name	Beispiel
1	Symbolic Games	Textadventure
2	Games	Pacman
3	Serios Game	Driver School 3D
4	Serios Game + HuD	
5	Serios Game + HuD + teilautonomer Bot	
6	Serios Game + HuD + vollautonomer Bot	

Tabelle 3: Hierarchie von Autonomie

zumindest um die Entscheidungen der menschlichen Spieler zu kommentieren.

In einem solchen Szenario ist es nicht nötig, mit Domain Experten noch über die Realisierung in Software zu debattieren, weil das ja bereits erfolgt ist. Die Software ist geschrieben, die 2000 Lines of Code sind gut so wie sie sind. Worum es stattdessen geht ist den vollständigen Umfang der UML Spezifikation auf diesen Code anzuwenden. Ich wage mal die steile These, dass genau dafür UML eigentlich erfunden und nicht für die Modellierung von Domainen in Software.

8.3 Bug: Reverse Engineering von Java-Code bringt nichts

Leider hat sich herausgestellt, dass das Reverse Engineering von Java Code um daraus UML Diagramme zu erzeugen nichts bringt. Stattdessen bilden Computerspiele eine Hiearchie aus, die man nur mit manuellem Programmieraufwand abwandern kann. In der Abbildung ist das Prinzip häher erläutert. Je höher man die Stufe hinaufgeht, desto leichter fällt es eine Künstliche Intelligenz für das jeweilige Spiel zu schreiben.

Vielleicht ein Beispiel: angenommen man findet auf github ein Spiel der Kategorie 3, also ein Lernspiel bei dem es noch kein Head up Display und auch keinen Bot gibt. Selbst wenn man den Sourcecode des Spiels kennt und daraus rückwärts ein Sequenzdiagramm erzeugt ist es nicht möglich, daraus automatisiert ein Head-up Display zu erzeugen. Also eine Softwarekomponente welche dem Spieler mit Hilfen zur Seite steht. Der Grund ist folgender: wenn es so leicht wäre, hätte der ursprüngliche Programmier dieses Head-up Display selber verbaut. Also wird man als externer Programmierer das nicht mal so nebenbei machen.

Was man jedoch tun kann, ist Arbeit in den Sourcecode zu stecken, um das Spiel von Stufe 3 auf Stufe 4 anzuheben, also manuell das benötigte HuD zu programmieren. Die Rangfolge der Stufe ergibt sich dadurch wieviel Aufwand es bedeutet, das Spiel auf das Niveau eines vollautonomen Bots zu bringen. Hat man ein Spiel der Stufe 5 vorliegen fällt es leichter dafür eine Künstliche Intelligenz zu schreiben, als wenn man ein Spiel der Stufe 2 vor sich hat. Mit leicht ist gemeint, der benötigte Sourcecode der ergänzend geschrieben werden muss. Also die Anzahl der Codezeilen die sich in Mannjahre umrechnen lassen.

Die schlechte Nachricht lautet, dass es keine Wundergeneratoren gibt, um ohne Mühe ein Head-Up Display oder sogar teilautonome Bots in ein Spiel einzuprogrammieren. Man muss von außen Energie hinzufügen, also die jeweilige Domäne analysieren und dafür passend die Erweiterung vornehmen.

8.4 Realisierung der Subsumption Architektur

Was die Subsumption Architektur ist, brauche ich nicht groß erläutern. Jeder wird das berühmte Paper von Rodney Brooks gelesen haben. Was jedoch noch unklar ist, ist die Frage wie man die unterschiedlichen Layer konkret in Software realisiert. Meiner Meinung nach geht das über objektorientierte Programmierung. In seinem Aufsatz ermittelt Brooks unterschiedliche Aufgabe die ein Roboter quasigleichzeitig ausführt:

- Umgebungserkennung
- Pfadplannung
- Microbewegungen ausführen
- Gegenstände aufnehmen
- High Level Planner
- Karte erstellen
- Aktoren ansteuern

Brooks hat die Layer als Schaubild eingezeichnet so wie es Elektrotechniker tun. Besser wäre es gewesen die UML Syntax zu nutzen. Jeder der oben aufgeführten Tasks ist eine Klasse, und das Domainmodel ist das identisch mit dem UML Model. Solche UML Modelle sind im Bereich von Business-Applikationen üblich. Man kann damit Geschäftsprozesse modellieren wo Lieferanten, Kunden und die Rechnungsabteilung involviert sind. Genausogut lässt sich UML aber auch in der Robotik sinnvoll einsetzen. Es gibt einige wenige Paper dazu, aber es funktioniert ausgezeichnet. Der Clou bei UML ist, dass selbst wenn ein Task komplizierter sein sollte wie z.B. “Karte erstellen” man die Klasse weiter aufteilen kann in Unterklassen. Am Ende erhält man eine Hierachie aus Objekten die sich gegenseitig Nachrichten zusenden. Das tolle an diesem Modell ist, dass es 1:1 in Java Sourcecode überführt werden kann.

Die Idee Subsumption objektorientiert auszuführen vertreten auch: [3, 5]

9 Ontologien

9.1 Scene Understanding mit Ontologien

In mehreren Papern wurden Grammar based Scene Parser vorgestellt. Zuletzt von Prof. Aloimonos Yiannis von der University Maryland [37]. Er beschreibt eine Knowledgedatenbank die als kontextfreie Grammatik gespeichert ist und mit der man eine Szene parsen kann. Ein Mensch führt Aktionen aus, erzeugt dadurch eine Timeline in der er Gegenstände aufnimmt und Manipulationen ausführt und der Scene Parser verwendet eine Domänenspezifische Sprache um darin einen Sinn zu erkennen. Das Konzept ist nicht neu sondern orientiert sich am Parsen von Computersprachen. Leider gibt es mit diesem Ansatz ein Problem. Es ist nicht effizient genug. Was nicht bedacht wurde ist, dass eine kontextfreie Grammatik eben keine Wissensdatenbank darstellt, sondern nur eine Markupsprache ist ähnlich wie HTML. Damit lassen sich zwar Aktions taggen, aber in sehr engen Grenzen.

Deutlich fortschrittlicher geht hingegen [34] dessen Paper ich ausführlicher würdigen möchte. Er nutzt keine Grammar sondern eine Ontologie. Der Unterschied besteht darin, dass Ontologien normalerweise als Ausgangspunkt verwendet werden um komplexe UML Diagramme und dann Java Code zu implementieren. Das Aufstellen einer Ontologie ist im Softwareengineering ein wichtiger Meilenstein und wird um Use-Case-Diagramme ergänzt. Vielleicht gleich zu Anfang eine Mini-Kritik an dem Paper. Natürlich fragt man sich, wenn da eine Ontologie verwendet wurde, wo bitte ist der dazu passende Sourcecode in Java. Weil, eine Ontologie allein ist noch keine Software. Nun, vermutlich wird der ausführbare Java Code irgendwo existieren und

die gezeigten UML Diagramme in dem Paper sind über Reverse Engineering entstanden. Der Grund warum sie den Weg in das Paper gefunden haben, nicht jedoch der komplette Java Code dürfte etwas mit dem Patentrecht zu tun haben. Weil, hätte man den kompletten Java-Code publiziert, wäre es möglich das System nachzubauen und genau das ist nicht erwünscht. Dies ist jedoch keine inhaltliche Kritik, sondern es ist eher eine Frage ob man OpenSource oder propritäre Software verwendet. Die Idee als solche eine Ontologie zu nutzen ist stimmig.

Doch gehen wir auf das Paper etwas genauer ein. Auf Seite 2 erfährt der Leser dass es eine Klasse namens MobileObject gibt. Darin sind weitere Unterklassen enthalten wie Crowd, Group, Hand, Person und Vehicle. Die Klasse Person ist noch detailierter beschrieben und besteht aus Height, Weight, Speed, Siluhette und weiteren Parametern. Auf Seite 4 werden noch weitere Klassen eingeführt. Neben MobileObject gibt es noch statische Objekte wie Door und Zone, und es gibt eine Klassenhierarchie die sich nur um Events kümmert. Darin wird gespeichert, ob jemand springt, eine Zone betritt, ob eine Zone überfüllt ist usw. Die komplette Ontologie bzw. das UML Diagramm ist in der Arbeit nicht abgedruckt, es scheint aber etwas größer zu sein. Ich würde mal schätzen es dürften so 30 Klassen und mehr sein. Das ist schwer zu sagen. Jedenfalls kann man das ganze sich vorstellen als eine Klassenhierarchie eines Videoüberwachungssystems. Wo man alles was wichtig ist für diese Aufgabe als Klassen implementiert hat.

Der nächste Schritt wäre dann die Klassen mit Methoden anzureichern, den Code mittels Agiler Techniken zu verbessern, Issues zu tracken usw. bis irgendwann die Software einsatzfähig ist. Davon steht in dem Paper nichts drin, sondern es wird nur die Ontologie selber beschreiben. Vom inhaltlichen Ansatz her ist das sehr fortschrittlich. Anders als bei dem eingangs zitierten Paper was mit Kontextfreien Grammatiken arbeitet ist das Konzept objektorientierte Ontologien zu nutzen extrem mächtig. Es läuft darauf hinaus, dass das fertige Programm nicht einfach nur ein Template darstellt um eine Sprache zu parsen, sondern es erinnert mehr an ein richtiges Softwareprojekt mit vielen hundert Lines of Code.

Schwierigkeiten oder ernsthafte Probleme sehe ich nicht mit Hinblick auf die gestellte Aufgabe. Vielmehr dürfte nur das selbe Spießroutenlaufen beginnen wie bei jedem anderen Java / C++ Projekt. Die implementierten Klassen enthalten Bugs, die User melden haufenweise Fehler, der Code muss refactored werden usw. Aber das sind Detailfragen die etwas mit Software-Engineering zu tun haben. Aus Sicht der Künstlichen Intelligenz ist der gewählte Ansatz grundsätzlich der richtige. Genügend Manpower vorausgesetzt lässt sich das System beliebig leistungsfähigk gestalten.

UML vs. Grammar Anhand der beiden vorgestellten Paper kann man gut die Unterschiede aufzeigen. In Variante 1 wurde eine kontextfreie Grammar verwendet, welche in der traditionellen Informatik weit verbreitet ist. Nach diesem Prinzip wurden auch die ersten Algol Compiler implementiert und die ersten Maschinenübersetzer entwickelt. Bis heute werden Parsergeneratoren eingesetzt um via Bison/Yacc C-Code in Maschinencode zu übersetzen und es liegt auf der Hand das Konzept auch für das Parsen von Videos zu verwenden.

Im Beispiel 2 wurden eine Ontologie verwendet, welche in der Informatik-Geschichte sehr viel jünger ist. Die objektorientierte Programmierung hat sich erst Mitte der 1980'er durchgesetzt und die UML Spezifikation ist erst um das Jahr 2000 herum entstanden. Insgesamt ist die Variante 2 leistungsfähiger, einfacher zu erweitern und der grammar-basierenden Lösung überlegen.

Es gibt jedoch auch Punkte die für die Grammar Lösung sprechen. Und zwar ist diese besser dokumentiert. Es gibt mehr Paper die sich mit dieser Thematik beschäftigen und es gibt sogar wenigstens einen Youtube Vortrag wo eine Live Präsentation gezeigt wurde. Die Ontologie-basierende Lösung ist noch sehr geheimnisvoll. Im oben zitierten Paper fehlt der Sourcecode und es gibt nur wenige Paper die es im Detail beschreiben.

9.2 Metaprogramming

Zunächst die schlechte Nachricht: es gibt keinen magischen Algoirthmus der eine vorhandene Ontologie via Makroprogrammierung in ausführbaren Sourcecode überführen kann. Sogenanntes Sourceless Metaprogramming ist ein Mythos. Er leugnet die Wirklichkeit, dass Sourcecode mit einer Produktivität von rund 10 Codezeilen am Tag durch Firmen, Organisationen oder wem auch immer erstellt wird. Was man jedoch tun kann ist eine Ahnung davon zu vermitteln wie ein fertiges Robot-Control-System aussieht, wenn man schon eines hat.

Nehmen wir das Serios Game "Driver School 3d" als Vorbild. Es handelt sich dabei um ein kommerzielles Computerspiel, was irgendwer programmiert hat und relativ bekannt ist. Wenn man auf den Sourcecode dieses Spiels (was leider kein OpenSource ist) via Reverse Engineering einen UML Chart erstellt erhält man ein riesiges Diagramm mit hunderten von Klassen welche alle Aspekte eines intelligent Tutoring System aus der Domäne Fahrschulprüfung beschreiben. Dieses Wissensnetz kann man als maschinenlesbare Ontologie beschreiben. Es gibt dort Klassen für den menschlicher Fahrer, gesetzten Blinker, die Fußgänger, Trajektorien, Straßenverläufe ja sogar für das Einparken und überholen. Als ist fein säuberlich in dem Diagramm enthalten.

Solche Ontologien sind nicht bottom up entstanden wo jemand mit argoUML zuerst das Diagramm gezeichnet hat um daraus dann C++ Sourcecode zu generieren, sondern sie funktionieren Codegetrieben. Das heißt, die Herstellerfirma programmiert zuerst das Spiel "Driver School 3D", verwendet im Optimalfall dazu ein Versionscontrol-System und bereits erstellte Game-Engines und baut daraus in einem Projekt was über viele Monate geht die fertige Software. Erst ganz am Ende erhält man das UML Diagramm in seiner gänze.

In solchen UML Ontologien wird die Domäne vollständig abgebildet. Es ist die Übertragung von Wirklichkeit in den Computer. Es gibt dafür keine konkrete Algorithmen oder Meta-Strukturen, sondern im wesentlichen sind nur die Elemente enthalten, welche der UML2 Standard hergibt. Also Klassendiagramme, Zustandsdiagramme und Paketübersichten. Klassendiagramme wiederum sind die wichtigste Gruppe und bestehen aus Attribute und Methoden. Methoden lassen sich ebenfalls grafisch veranschaulichen als Programmablaufpläne. Mehr Elemente gibt es nicht, darin kann man eine komplette Domäne beschreiben. Es gibt keinen Aspekt der Wirklichkeit der sich nicht in diese Beschreibung darstellen ließe.

Die spannende Frage ist jetzt: wie erstellt man solche ausführbaren Ontologien from Scratch? Mit Hinblick auf die eingangs zitierte Produktivitätskennzahl von 10 Lines of Code am Tag, wird Software und die daraus ableitbaren Ontologien durch die Zusammenarbeit von Menschen erstellt. Meist in kommerziellen Spielefirmen, manchmal auch als OpenSource Projekte. Je mehr Manpower man bereitstellt, desto realistischer kann man die Domäne modellieren.

"Driver School 3D" ist nicht irgendein Spiel, wo man aus dem Sourcecode den UML Chart ableitet, sondern es ist ein Intelligent Tutoring System, hat also einen edukativen Anspruch. Damit ist gemeint, dass der Ablauf üblicherweise Stressfrei funktioniert, man also nicht irgendwelche Raumschiffe abballern muss, dafür gibt es jede Menge abstrakter Regeln wie das man an der Kreuzung langsamer fahren muss oder die Rotphase der Ampel respektieren muss. Genau genommen sind die Spielinhalte als Unterricht ausgelegt und haben das Ziel Einsicht zu vermitteln, also in die Domäne des Straßen-

verkehrs. Es handelt sich um eine Form des selbstbestimmten Lernens, die ohne Bücher und sogar ohne menschliche Lehrer auskommt. Stattdessen wird das pädagogische Lernziel allein in Software formuliert. Am ehesten ist “Driver School 3D” mit Schulfernsehen zu vergleichen, nur eben auf das Medium Computerspiel übertragen. Ähnlich wie die Telekolleg Reihe steht dahinter ein erfahrenes Autorenteam, nur dass in dem Team keine Fernsehexperten sondern Computerspiele-Programmierer die eigentliche Realisierung übernahmen, beraten wurden sie dabei von Fahrschullehrern und Buchautoren die sich der Domäne Verkehrserziehung angenommen haben. Man kann sagen, dass “Driver School 3D” ein konkretes Beispiel für die 5. Computergeneration ist, also ein Expertensystem darstellt.

9.3 Golog

Die wichtigste Frage innerhalb der Robotik lautet wie man die Komplexität senken kann. Das Ziel besteht darin, möglichst wenig zu programmieren. Trotzdem soll am Ende ein funktionsfähiges Programm entstehen, was einen Sinn ergibt. Schauen wir uns zunächst einmal an, wie Roboterwettbewerbe durchgeführt werden. Im Regelfall sind Micromouse und Robocup Challanges eine Mischung aus Hard- und Software. Wo man also einerseits fit sein muss in Microcontrollern aber gleichzeitig auch Software erstellt. Man kann diese Wettbewerbe soweit vereinfachen, dass man den Teil mit der Hardware einfach weglässt und nur noch einen Controller in der Simulation programmiert. Man kann den Hardwareteil deshalb ignorieren, weil die Aufgabe zu leicht und zu unwichtig ist. Es ist nicht besonders schwer, einen ferngesteuerten Roboter von der Hardware in Betrieb zu nehmen, es gibt sogar kommerzielle Roboter out-of-the-box wo man nur die Batterien reintun muss und das Teil fährt los. Doch bleiben wir bei der reinen Software-Simulation.

Angenommen man hat nur noch eine Game-Engine plus Computerspiel und soll darin jetzt den Roboter steuern. Damit hat man schonmal auf das wirklich schwierige Thema fokussiert. Wenn man jetzt Behavior Trees und User-Interfaces implementiert konzentriert man sich automatisch auf das was als Kern-AI gilt. Aber, auch bei Simulationen sind viele Dinge überflüssiges Beiwerk, das betrifft die komplette Bildschirmausgabe, sowie alles was normalerweise im Bereich Game-Development durchgeführt wird, beispielsweise das Anzeigen von Sprites auf dem Bildschirm, das Setzen der Framerate usw. Wenn man jetzt noch weiter sich auf das Kernthema reduziert wird man sehen, dass es eigentlich um Domainmodelling geht. Also die Frage wie man ein Spiel wie Robocup, Micromouse usw. in Form von abstrakten Regeln formuliert. Diese können, müssen aber nicht, visuell ausgegeben werden.

Eine wichtige Programmiersprache zur Domänenmodellierung ist Golog. Es handelt sich dabei um ein Spiel im Spiel. Mit Golog kann man anders als mit C++ keine Grafiken zeichnen oder User-Interface gestalten sondern man kann darin nur das Spiel als solches modellieren. Vergleichbar mit der Game-Description Language. Und genau hier reduziert man das Problem auf seinen Kern. Und zwar lautet die Herausforderung für eine beliebige Domaine den Golog-sourcecode zu programmieren. Hat man diesen, kann man ihn mittels Game-Engine grafisch darstellen und die Game-Engine kann wiederum einen echten Roboter im Reallife steuern.

Machen wir es etwas konkreter: in meinem eigenen Robot-Control-System habe ich schön mit Python gearbeitet. Dort konkret mit Hilfe der Pygame-Engine ein komplettes Spiel programmiert. Python eignet sich dafür außerordentlich gut, weil es eine general purpose objectorientierte Programmiersprache ist. Man kann damit alles Programmieren, also auch Spiele. Nur, der Sourcecode hat einen Nachteil, es wird dort die Domäne (eine Trafficsimulation) vermischt mit der GUI. Das heißt, die meisten Anweisungen in dem Programm haben etwas mit Grafikausgabe, Abfrage des Keyboards und Textausgabe zu tun. Das ist zwar schön, dass Python das alles unterstützt und für die Software ist das auch wichtig, nur ist es eben keine minimale Domänenmodellierung. Meiner Meinung nach wäre es nötig, innerhalb des Python Programms einen Bereich nur für die Domain-Modellierung zu reservieren. Wie man um eine Domäne herum das Computerspiel programmiert und ob man es in 2D, 3D oder wie auch immer macht ist ähnlich simpel wie das Bauen von Physischer Hardware für den Roboter. Worum es geht ist es, sich auf jenen Teil zu fokussieren der wirklich relevant ist. Also die 20% des Projekts die am schwersten sind.

Aktuell kann ich das Gebiet noch nicht richtig eingrenzen. Ich vermute mal dass man mit Golog, Indigolog und General Game Description Language die Domäne modellieren muss. Für das Beispiel Robotcup also sehr formal definiert, was überhaupt in den Spiel passieren darf. Man könnte es als Game-Physik bezeichnen. Die Kunst besteht darin, die Physik-Engine von dem eigentlichen Spiel zu isolieren.

Vielleicht etwas konkreter: Aktuell besteht men Robot-Control-System aus rund 1000 Zeilen Code in Python. Zur Domänenmodellierung selber wurden vielleicht 300 Zeilen verwendet, mehr nicht. Im Grunde könnte man diese 300 Zeilen extrahieren und als eigenes Projekt pflegen. Aktuell wurde das nicht gemacht stattdessen hat ein Game-Object wie ein einzelnes Auto gemäß dem Objektorientierten Paradigma auch Methoden mit dabei um sich auf den Bildschirm zu zeichnen. Aus Sicht der Spieleprogrammierung ist das so Standard, aber besser wäre es wenn man das in einem extra Layer ausführt. Hier mal die grobe Skizze:

Physical Robot -> Computersimulation -> Domainmodel

No Python Schauen wir uns einige Teile eines Python Programs an, die mit der eigentlichen Domain-Modellierung nichts zu tun haben. Einmal betrifft das alle Grafikausgaben mit Hilfe von Pygame, also das Zeichnen von Linien und Kreisen, dann das Abfragen von Tastaturevents, ebenfalls nichts mit Domain-Modelling zu tun hat ein Framecounter, oder die Python Threading Funktion um Prozesse parallel auszuführen. Zwar kann man keines dieser Befehl in dem Python Programm weglassen, aber es sind Funktionen die sehr auf die konkrete Bildschirmausgabe hin zugeschnitten sind. Sie kommunizieren direkt mit dem Python-Interpreter und dem Betriebssystem. Machen wir es etwas konkreter: angenommen man konvertiert den Python Sourcecode in ein UML Diagramm, dann hat mindestens die Hälfte der Klassen nichts mit dem eigentlichen Domain-Model zu tun.

Schaut man sich hingegen ein Indigolog-Programm [9] an, so ist das sehr viel konzentrierter auf die Domäne hin zugeschnitten. Indigolog besitzt einen sehr klar definierten Anwenduungszweck.

9.4 Was ist falsch mit Wissensmodellierung?

Unter Wissensmodellierung versteht man die Übertragung von Expertenwissen in eine maschinenlesbare Form. Tools die häufig eingesetzt werden sind OWL, Protege, PDDL, Golog und Domain-specific languages. Häufig gibt es weitere Tools die zwischen diesen Formaten hin- und herkonvertieren. Leider gibt es ein Problem. Nichts daron ist praxistauglich. Protege ist böse formuliert, pseudoscience. Das gleiche gilt für PDDL. Das fein säuberlich erstellte Modell ist nicht ausführbar, es ist nichts anderes als eine MIndmap, also ein Mockup wie die künftige Anwendung einmal aussehen soll. Golog Programme sind zwar ausführbar, sie sind leider häufig sehr abstrakt so dass es keinen Sinn macht sie direkt auszuführen, sondern stattdessen braucht man weitere Solver die das Golog Programm nach Zielen absuchen. Warum diese negativ-Punkte auftreten hat etwas mit dem Selbstverständnis von Wissensmodellierung zu tun. Üblicherweise will

man eine möglichst allgemeine Darstellung in der Hoffnung damit die Komplexität zu senken. Meiner Erfahrung nach übertreibt man es. OWL und darauf aufbauende Verfahren senken die Komplexität zu stark.

Schauen wir uns die nächst niedrige Stufe an, wenn man nicht so stark abstrahiert wie bei Golog sondern weniger gelang man zu Python. Einer High-level-Programmiersprache mit der sich Software erzeugen lässt. Python kann all das was Golog auch kann, ist aber für die Praxis nutzbar. Mein Eindruck ist, dass sich Python für echte Projekte eignet, während die nächst höhere Stufe wie OWL und Golog ungeeignet sind. Um diesen Gap näher zu beschreiben ein kleiner Exkurs in die Geschichte der Informatik. Als Programmiersprachen der 3. Generation werden prozedurale Sprachen bezeichnet wie C und Pascal. Als Sprachen der 4. Generation dann objektorientierte Sprachen wie Python und Java. Die eingangs zitierten Tools zur Domänenmodellierung werden als Sprachen der 5. Generation bezeichnet. Gemeint sind Sprachen die auf OWL, Prolog und PDDL aufbauen. Leider wissen wir heute, dass die 5. Generation gescheitert ist, sie konnte sich nicht durchsetzen, einen Layer oberhalb von objektorientierter Programmierung gibt es nicht.

Was man stattdessen machen kann um komplexe Systeme zu entwickeln ist das Programmieren von Libraryies. Also mit einer Sprache wie Python eine Library programmieren um die Domäne Micromouse abzubilden, und eine zweite Library für Robotcup. Solche Librarys haben den Vorteil, dass sie anders als Protege-Modell ausführbaren Code enthalten. Man kann sie starten und dann passiert etwas. Die Frage bleibt nur noch zu klären, wie man solche Librarys erstellt. Meiner Meinung nach mit den Tools die allgemein bekannt sind: also Texteditoren auf der technischen Seite und github-Kollobaration auf der informellen Seite. Anders formuliert, die Programmiersprachen der 5. Generation sind nicht so sehr Programme mit denen man Software erstellt, sondern die 5. Computergeneration bedeutet, dass man Technologie aus der 4. Generation einsetzt und sie mit Internet-Kollaboration wie Wikis und OpenSource Entwicklung erweitert.

Github wurde bereits erwähnt. Github ist anders als Protege keine Software sondern ein Online-Dienst. Wenn man seine Funktionen voll ausnutzt hat man dort ein Wiki, einen Issuetracker, eine Versionsverwaltung und viele Forks. Das ist die eigentliche Infrastruktur um Domain-Modelling zu betreiben. Innerhalb dieses Rahmens lässt sich jede beliebige Domain machinenlesbar darstellen. Vorausgesetzt die Community ist fleißig engagiert, das heißt man braucht Leute die Code hochladen, Issues fixen und das Wiki aktualisieren. Domain Modelling ist weniger eine Sache für Prolog, als vielmehr ein Software-Entwicklungsprozess. Er scheitert nicht am Situationskalkül sondern an einer fehlenden Community. Wenn also sich niemand für das Github Projekt interessiert und der Issue-Tracker verwaist bleibt.

9.5 Domainknowledge speichern

Domainen-spezifisches Wissen ist der zentrale Punkt von Künstlicher Intelligenz. In der Vorstellungswelt von Expertensystemen wird es als knowledge-base also als Wissensbasis bezeichnet. Leider wird häufig nicht gesagt, wie konkret Domainenwissen in Software modelliert wird. Die Antwort ist simpel: mit Hilfe der UML Notation. Genauer gesagt durch objektorientierte Programmierung. Es handelt sich um jenen Teil der Software die bei Computerspielen als Physik-Engine bezeichnet wird. Also ein Modul was selbst keine Grafikausgabe besitzt sondern nur dafür da ist, die Objekte der Simulation zu verwalten. Einfache Physik-Engines wie box2D sind lediglich in der Lage Kollisionen zu erkennen und Schwerkraft zu berechnen. Komplexere Beispiele wie die DART-Engine (DART: Dynamic Animation and Robotics Toolkit) verwalten darüberhinaus auch noch inverse Kinematiken und Pfadplanner. In Autorennspielen wird eine Physik-Engine noch um weiteres Wissen aus dem Bereich des Racing erweitert, wie z.B. spezielle Module um die optimale Streckenführung zu bestimmen oder um zu ermitteln wann ein Auto in den Boxengasse muss.

Aber egal welches Domänenwissen man implementiert, es wird immer objektorientiert implementiert, also in einer Hochsprache wie C#, Java oder Python. Zur Veranschaulichung wird häufig die UML Notation verwendet und um sehr komplexe Ontologien zu erzeugen die aus mehreren hundert Klassen bestehen werden verteilte Versionsverwaltungen wie git eingesetzt. Und hier sieht man bereits den Flaschenhals des Domain-Modelling. Es ist abhängig von menschlichen Programmierern. Hat man 10 Developer kann man nur eine kleine Domäne in Software überführen, hat man hingegen 1000 Developer kann man sehr komplexe Simulationen erstellen.

Es mag für viele Einsteiger ein wenig sonderbar erscheinen, dass ausgerechnet die Knowledge-Base einer Künstlichen Intelligenz nicht automatisch erzeugt wird durch hochentwickelte Algorithmen sondern das menschliche Programmierer benötigt werden, die sehr langsam Sourcecode in die Tastatur eintippen. Aber das ist die Wahrheit. Egal ob in OpenSource Projekten oder bei kommerzieller Domain-Modellierungen kommen immer und ausschließlich menschliche Programmierer zum Einsatz. Auch bei sehr hochentwickelter Software wie IBM Watson wurde das System von Menschen programmiert. Dort wurden vor allem Computerlinguisten eingesetzt weil die Domäne des IBM Watson etwas mit Sprache zu tun hatte.

Im Laufe von mehr als 50 Jahre Softwaregeschichte wurde viel darüber nachgedacht ob man die Entwicklung von Software effizienter gestalten kann. Hier ist insbesondere “Automatic Programming” zu nennen. Doch bisher ohne Erfolg. Heute wird Software noch auf die selbe Weise programmiert wie vor 50 Jahren, ja auch die Effizienz wieviel Lines of Code pro Tag ein Programmierer schreibt hat sich nur minimal erhöht. Das derzeitige Optimum was bei Google und Spielefirmen eingesetzt wird ist eine Mischung aus einer Hochsprache wie Java, ein Versionsverwaltungssystem wie git plus Mailing-Listen zur Projektorganisation. Das hilft dabei den manuellen Aufwand zu minimieren aber man kann damit menschliche Programmierer nicht ersetzen. Will man größere Software programmieren, genauer gesagt will man komplexe Domänen in Software modellieren benötigt man nach wie vor sehr viele Programmierer dafür. Google verfügt derzeit über 40000 Software-Entwickler [9] Microsoft hält 50k unter Vertrag [10], Apple 16k [11] und Bigblue IBM bringt es auf stolze 275k [12].

Die Gemeinsamkeit der Software von diesen Firmen ist, dass sie erstens aus nominell sehr vielen Lines of Code besteht und das sie objektorientiert programmiert wurde. Ein Blick in die Zukunft verrät dass genau das so weitergehen dürfte. Die Software wird nominell noch umfangreicher und sie wird noch mehr objektorientiert sein. Man darf daraus schlussfolgern, dass Software weniger etwas mit Compilern, Hochsprachen oder Hardware zu tun hat, sondern dass Software ein soziales Grounding besitzt. Der äußere Rahmen in dem Software entsteht ist gesellschaftlicher Art, lässt sich also mit Begriffen wie Arbeitsvertrag, Firma und Ausbildung beschreiben. Die oben erwähnten Zahlen der beschäftigten Software-Entwickler sind dadurch gekennzeichnet dass sie einen eigenen Arbeitsmarkt darstellen. In dem Sinne dass es Angebot, Nachfrage und Geld gibt welche darüber entscheidet, welche Software erstellt wird und was sie zu leisten im Stande ist.

Wesentliches Merkmal dieser gesellschaftlichen Struktur ist, dass sie weitestgehend intransparent für die Öffentlichkeit ist. Der Großteil der Programmierung findet hinter verschlossenen Türen statt, nur ein

[9] https://www.quora.com/How-many-software-engineers-does-Google-have
[10] https://www.quora.com/How-many-engineers-does-Microsoft-have
[11] https://www.quora.com/How-many-software-engineers-does-Apple-have
[12] https://www.quora.com/What-is-the-average-number-of-software-engineers-at-IBM

sehr kleiner Teil wird als OpenSource veröffentlicht, die meiste Software wie beispielsweise die Google Suchmachine wird überhaupt nicht kommerziell verkauft. Man kann bei Google lediglich auf die API zugreifen nicht jedoch auf den Sourcecode selber der die Suche ausführt. Der Grund warum Softwareentwicklung verdeckt ausgeführt wird hat damit zu tun, dass Software etwas sehr mächtiges ist. Würde IBM den Sourcecode zu IBM Watson offenlegen, würden sie frewillig Macht aufgeben.

Softwareentwicklung kann man nicht lernen im eigentlichen Sinne. Der durchschnittliche Hobbyprogrammierer der in C++ ein Jump'n'run Game zusammenfrickelt besitzt bereits die selbe Produktivität die auch ein Apple Programmierer hat. Der Unterschied ist lediglich der, dass bei Apple die Leute im Team arbeiten und so größere Repositorien erstellen können.

Vom Softwarekonzern Microsoft gibt es auf Youtube einen Clip von der CES 2016 Party. Man sieht darin eine Großraumdisko in der mindestens 1000 Leute sehr eng zusammenstehen, schwitzen und sich zu Techno-Musik bewegen. Scheinwerfer sind auf das Party-Volk gerichtet, viele sind angetrunken, von der Decke seilen sich Tänzerinnen mit Regenschirmen ab.

9.6 Domain specific Game-Engine

Als GameEngine wird normalerweise ein System wie pygame oder Unity3d verstanden, also ein Satz von Grafik und Physik-Befehlen mit deren Hilfe man Spiele erstellen kann. Man kann aber auch domainspecific Game-Engines verwenden, beispielsweise ein Framework um Soccer-Games zu erstellen. Genauer gesagt handelt es sich um ein ausprogrammiertes Spiel was als Librarys konzipiert ist. Dieser Ansatz ist relativ selten vertreten. Beim Mario AI Wettbewerb wurde so eine Engine bereitgestellt, das heißt der Veranstalter hat ein 2d Jump'n'Run inkl. API bereitgestellt und die Teilnehmer mussten dafür dann Bots schreiben. Manchmal kann man die Schnittstelle auch mittels AutoIt erweitern um so nachträglich Bots für Spiele zu schreiben.

Generell lässt sich anhand dieser Thematik erklären was Domain-Knowledge eigentlich ist. Eine normale Game-Engine wie pygame oder eine Physik-Engine wie box2d Enthält noch kein Domain-Knowledge. Wenn man jedoch eine Plattform für Robocup Soccer programmiert wo die Bots schon Standard-Befehle wie moveto, dribble und shot beherschen hat man Domänenwissen implementiert. Die Messlatte liegt weniger in dem Programm als solchen, sondern es kommt auf die Schnittstellen nach Außen an. Der Ablauf geht wie folgt.

Auf der untersten Ebene liegt die Physik-Engine. Darüber kommt dann das eigentliche Spiel, z.B. eine Fußballsimulation, und darüber kommt dann die Schnittstelle zu anderen Programmen, also die API um Bots für das Spiel zu schreiben.

10 Programmiersprachen

10.1 Welche ist die beste?

Einen Mangel an Programmiersprachen gibt es wahrlich nicht. Leider ist bis heute nicht so ganz geklärt was davon die beste ist. Am liebsten würde ich erklären, dass Forth die beste Programmiersprache ist. Forth entspricht ungefähr einem tiefergelegten Rennauto, es ist die unangefochtene Nummer 1 und die absolute Hackersprache. Damit ist gemeint, dass die Sprache so unglaublich mächtig ist, dass niemand da mithalten kann. Gegenüber Forth ist C++ nur eine müde Krücke, eine Sprache für Leute die gerade erst angefangen haben mit der Programmierung. Profis hingegen setzen auf Forth. Sie schreiben darin nicht nur ihre Programme, sondern das Betriebssystem, den Compiler und die VHDL Beschreibung für den IP Core gleichmit. Leider gibt es mit Forth eiin Problem. Wie bei jedem Profi-Rennwagen braucht man dafür einen Fahrer der das Auto steuern kann. Ich habe es versucht und bin mit Forth einige Runden gefahren. Und was soll ich sagen, ich hatte Probleme den Gang reinzubekommen, das Auto ist mir in der Kurve ausgebrochen, ich habe mehrmals den Motor abgewürgt, und beim Gasgeben bin ich mehrmals in die Leitplanke hineingefahren. Anders formuliert, Forth ist ein wenig zu stark für mich, ich bin der Sprache nicht gewachsen, das ist was für Leute die sich wirklich auskennen.

Eine Sprache die etwas leichter in der Bedienung ist Python. Dort braucht man nicht viel Fachwissen mitbringen sondern kann die Sprache als solche erlernen. Es gibt in Python leistungsfähige Bibliotheken und man kann damit relativ stressfrei Programme schreiben. Mein Eindruck ist, dass Python wesentlich mehr kann als einfach nur 100 Zeilen Scripte und Prototypen auszuführen sondern mit Python kann man vollwertige Programme schreiben. Aber, es gibt Sprachen die als Professioneller gelten. Python ist eben keine richtige Programmiersprache sondern nur ein Aufsatz für Betriebssysteme oder für die JVM. Als wirkliche Programmiersprache gelten hingegen C, C++, Java und C#. Das sind die Schwergewichte, es sind Sprachen mit denen man größere Anwendungen entwickeln kann, die grafikintensiv sind und die Multithreading auf mehreren CPUs benötigen.

Java gilt bei vielen als leichter zu bedienen. Doch die Sprache hat Schwächen. Java kommt nicht an den Komfort heran den Python bietet, es ist eine Sprache die genau zwischen den Extremen steht. Es ist keine kompilierte Sprache wie C++ aber auch keine interpretierte Anfängersprache wie Python. Noch dazu fehlt die Unterstützung für Pointer so dass Java niemals wird mit C/C++ konkurieren können was die Entwicklung von Betriebssystemroutinen angeht. Insofern tendiere ich zu der Haltung, dass C++ die bessere Sprache ist. C++ gilt zwar als überholt und steht im Kreuzfeuer von mehreren Seiten, dennoch gibt es bis heute keine ernsthafte Alternative. Die Kritikpunkte an C++ sind schnell benannt. Langjährige C Programmierer kritisieren die schwer verständliche Syntax, es gibt in C++ zuviele Befehle die man eigentlich nicht braucht, wenn man strukturiert programmieren möchte. Nur, wenn man sich einmal anschaut, was man als Programmierer benötigt dann erfüllt C++ ziemlich gut die Anfordungen. Eine moderne Sprache muss kompiliert sein, sonst ist sie nur eine Scrpting Sprache, und sie muss Objekte enthalten, sonst ist es nur ein C.

Ob C++ nun die beste Sprache ist will ich mal offenlassen. Eines ist sicher, es ist die meistgehassteste Sprache. Man kann in C++ durchaus Software entwickeln. Es gibt sogar Tools wie cpp2dia, womit man aus C++ Code UML Diagramme erzeugen kann. Ferner gibt es für die Sprache alle möglichen Erweiterungen und Handbücher so dass man damit eigentlich alles machen kann was man möchte. Bleibt noch die Frage zu klären ob OOP und Pointer zusammenpassen. Oder ob man nicht lieber die Dinge trennen sollte. Die Alternative zu C++ besteht darin, dass man für Lowlevel Aufgaben C nimmt und für Highlevel Programmierung Python.

Mein Eindruck ist, dass man um C++ einen großen Bogen machen sollte. Es ist zwar der Ort wo sich der Mainstream tummelt, aber so sieht nicht die Zukunft aus. Microsoft hat das erkannt und zu C++ längst einen Nachfolger entwickelt: C#. Diese Sprache ist halbwegs vernünftig ist aber für Linux Anhänger nicht optimal. Schaut man sich C++ selber an, so fällt vor allem auf, dass es keine Standard-Verfahren gibt die man lernen könnte, sondern dass es für jede Sachen mehrere Wege gibt. Allein die Frage wie und ob man Pointer einsetzt ist unklar. Einmal kann man normale Pointer einsetzen, was die meisten tun, dann gibt es noch Smartpointer und dann gibt es doppelt referenzierte Pointer (-> vs. "."). Also nichts gegen Pointer, ich liebe sie. Aber wenn dann bitte in reinem C, dort machen sie Sinn.

Fakt ist eines: C++ ist nicht so mächtig, dass es C ersetzt hat. Und es wird es auch niemals schaffen. Gleichzeitig fehlt C ein wichtiges

Detail: objektorientierte Programmierung. Meiner Meinung nach ist also die bereits erwähnte Mischung aus C plus Python das Optimum. Wo man als Hochsprache und zum Prototypen Python einsetzt und für die Lowlevelprogrammierung für zeitkritische Applikationen auf C setzt. Diese Mischung ist zwar nicht ganz so mächtig wie Forth, dafür sind jedoch die Anforderungen an die Programmierer kleiner. C und Python bietet zusammen soetwas wie eine Plattform innerhalb derer man in überschaubarer Zeit Code entwickeln kann. Nichts revolutionäres sondern nur Standardkost.

Kommen wir nochmal zurück zu Forth. Eigentlich ist Forth die Sprache die am meisten Spaß macht. Wannimmer man die Möglichkeit hat in Forth zu programmieren sollte man es tun. Und zwar explizit deshalb weil man dort für den fertigen Code rund 10x länger braucht als würde man es in Python schreiben. Die zusätzliche Zeit geht für das Lesen von sehr obskuren Manuals drauf, aus denen man sehr viel lernen kann. Das heißt, wenn der Code zu einem fixen Termin fertig sein muss, ist Forth vielleicht nicht gerade die Sprache der Wahl. Wenn man jedoch nicht nur an der Lösung interessiert ist, sondern so ganz nebenbei noch tiefer in die Informatik einsteigen will dann ist Forth perfekt. Bei Python hingegen ist die Programmierung sehr viel stromlinienförmiger. Man analysiert das Problem, sucht sich die passenden Librarys mit Hilfe von Stackoverflow zusammen und ist dann fertig damit. Man lernt nur das, was unbedingt nötig ist. Bei Forth hingegen wird aus einem simplen Primzahl-Algorithmus gleich eine Frage die etwas mit 6502 Opcodes zu tun hat und wo man ernsthaft darüber nachgrübelt wie man den Code sehr viel kompakter formuliert.

Was man von Forth nicht erwarten sollte, ist dass man das Problem damit gelöst bekommt. Üblicherweise endet der Ausflug zu Forth damit, dass man feststellt, dass das eigene Wissen nicht ausreicht, so dass man überhaupt keinen Code vorzuweisen hat, bzw. er Fehler aufweist.Forth ist vergleichbar als wenn man den Freeclimbing Sport ausführt, also ohne Seil die Felswand hochsteigt. Wenn der Berg nicht sehr hoch ist mag das angenehm sein, aber wehe man ist nicht wirklich gut im Klettern ...

Im Vergleich zu Forth kann man Python nur als absolute Looser Sprache bezeichnen. Sie ist nicht stackorientiert, sie benötigt um zu funktionieren einen C-compiler und ein komplettes Betrtebssystem und auf andere Hardware kann man die Sprache auch nur schwer übertragen. Noch dazu gibt es bei Python objektorientierte Programmierung out-of-the-box sowie jede Menge dokumentierte Libraries. Es ist also eine langweilige Programmiersprache die sich an Einsteiger richtet, die sich nur oberflächlich mit der Sprache auseinandersetzen wollen. Das Hauptproblem mit Python ist, dass man auch nach längerer Benutzung nichts neues lernt. Am Anfang vor vielen Jahren hat man sich mal das Handbuch angeschaut wo drinsteht wie das mit den Variablen und den Klassen geht und seitdem programmiert man so vor sich hin und schmort in seinem eigenen Saft. Die Ausführungsgeschwindigkeit von Python ist extrem langsam. Laut den letzten Benchmarks ist der Interpreter rund 20x langsamer als reines C. Und C ist bereits eine von den schlechteren Sprachen, weil auch sie nicht wirklich machinennah ist.

10.2 C++ und SFML

Wenn man Programmiersprachen an sich betrachtet, ist wohl Python der Spitzenreiter. Es ist eine sehr übersichtliche wie auch mächtige Programmiersprache. Das schöne an Python ist, dass man die Sprache selbst fast gar nicht bemerkt, man wird nur relativ selten mit Pointern oder Fehlermeldungen belästigt und kann sich stattdessen voll auf sein Problem konzentrieren. Es gibt nur ein kleines Problem. Informatik besitzt eine Geschichte und die von Python ist nicht besonders intensiv. Python wurde erfunden als andere Sprachen schon längst etabliert waren. Wäre Python in den 1970'er erfunden worden, wäre es sicherlich anders verlaufen.

Jetzt kann man sicherlich auf dem Standpunkt stehen, dass es nicht um das Gestern geht sondern um die Zukunft. Leider wurden Bibliotheken immer in der Vergangenheit geschrieben und auch Programmierer haben eine Biographie. Wenn Python die beste Sprache der Gegenwart ist, so ist C++ wohl die beste Sprache der Vergangenheit. C++ gab es schon zu MS-DOS Zeiten. Darin wurden sehr viele Computerspiele erstellt. Keine andere Sprache hat die Softwareentwicklung stärker geprägt. Einige sagen, C++ wäre tod weil Python alles besser macht, aber will man wirklich auf das Knowhow der C++ Community verzichten? Die Verwendung von C++ unterscheidet sich grundsätzlich von Python. Bei C++ hat man permanent mit dem physischen Computer zu tun, man tüftelt aus, ob man eine Variable als Variable oder lieber als Pointer übergibt oder ob man ein struct oder eine Klasse verwendet. Dadurch gelten C++ Programme als unwartbar. Auf der anderen Seite sind alle wichtigen Desktop-Applikationen in dieser Sprache geschrieben. Nicht so sehr weil C++ die beste Sprache ist, sondern weil es Ausdruck einer Ideologie ist.

Python Programmierer haben üblicherweise an der Sprache kein Interesse. Sie starten das Programm und es ist ihnen egal wenn es 20x langsamer läuft als eigentlich nötig. Python Programmierer sind eher am Problem orientiert, sie wollen das Spiel zum laufen bekommen oder sie wollen Messreihen auswerten. Üblicherweise sehen sich Python Nutzer nicht als Programmierer sondern als Nicht-Informatiker. Sie kommen aus angrenzenden Disziplinen wie Linguistik, Physik oder Medien und sind an internen Dingen nicht interessiert. Leider hat das zur Folge, dass sie auch an der Kultur der Informatik nicht interessiert sind. Sie wissen nicht, wie früher programmiert wurde, ihnen fehlt der Bezug zur Vergangenheit.

C++ ist da anders. C++ ist ein offenes Geschichtsbuch. Wer sich auf die Sprache einlässt wird nicht nur mit einer hohen Effizienz belohnt sondern auch mit Software die vor 30 Jahren geschrieben wurde.

Gaming Culture Man kann Computer auf sehr unterschiedliche Weise programmieren. In der universitären informatik gibt es eine Kultur die sich an mathematischen Funktionen orientiert, gemeint ist das Lamda Kalkül was seinen Niederschlag in Programmiersprachen wie LISP und Haskell gefunden hat. Damit sind nicht nur die Sprachen als solche gemeint sondern ein funktionaler Programmierstil. Man programmiert weniger in LISP als vielmehr für eine Community die eine Historie besitzt. Und exakt dasselbe ist auch mit C++ festzustellen. Man programmiert nicht wirklich objektorientiertes C++ sondern orientiert sich zunächst einmal an einer bestimmten Community. Genauer gesagt an einer Gaming-Community die sich bewusst vom akademischen Programmieren abgrenzt und auf Homecomputern Sidescroller programmiert. C++ ist die Sprache der Computerspiele der 1980'er und es ist auch die Sprache der kommerziellen Softwareentwicklung. Ob nun funktionale Programmierung oder imperative Programmierung besser ist kann man nicht genau sagen, was man jedoch sagen kann ist, dass es diese beiden Communitys gibt und man sich mit einer davon identifzieren muss.

Die Sprache Python würde ich im Bereich der akademischen Informatik und dort als Einsteigersprache einordnen. Sie wurde entwickelt für eine bestimmte Zielgruppe im Hinterkopf und zwar den Nicht-Programmierer der sich von Außen der Informatik nähert um darin konkrete Probleme mit wenig Programmieren zu lösen. Leider hat das zur Folge dass man von der Python Community nicht sehr viel lernen kann. Jedenfalls nichts über das Programmieren.

Konferenzen Vergleichen wir einmal zwei wichtige Konferenzen miteinander: Die Pycon auf der einen mit der cppcon auf der anderen

Seite. In beiden Fällen geht es um eine Programmiersprache. Von der Mächtigkeit her sind beide Sprache ungefähr gleichauf. Mit C++ kann man zwar ein wenig mehr Performance erzielen aber gegenüber pypy ist der Unterschied nur Minimal. Und selbst bei zeitkritischen Spielen fällt der Unterschied kaum ins Gewicht, auch mit Python lassen sich 60fps butterweich auf den Bildschirm zaubern. Ebenfalls gleichauf sind bei beiden Sprachen die objektorientierten Features. Ja fast möchte man sagen, dass eine Programmiersprache an einem Projekt wohl das unwichtigste überhaupt ist. Im wesentlichen gibt es nur wenige Befehle wie for, if und class und damit wird das Programm erzeugt.

Dennoch gibt es zwischen der Pycon und der Cppcon einen gewaltigen Unterschied. Auf der Cppcon sind die Redner sehr viel selbstbewusster. Sie glauben dort, dass sie in der besten Programmiersprache der Welt programmieren würden, was natürlich Interpretationssache ist, aber es ist typisch für die C++ Community. Während man in der Python Community weiß, dass man selber ein Anfänger ist und die anderen auch nur mit Wasser kochen. Die Frage ist jetzt, welche Lernatomosphäre einem persönlich mehr zusagt, und wo man glaubt am meisten zu lernen.

Häufig wird als Hauptvorteil von C++ angepriesen, dass es extrem schnell sei. Das ist zwar richtig, bringt aber in der Praxis keinen Vorteil. Der Bottleneck heutzutage ist nicht so sehr die Ausführungsgeschwindigkeit sondern die Programmiergeschwindigkeit. Diese wiederum ist abhängig von der Community. Und das ist der eigentliche Vorteil von C++. Das heißt, nicht C++ an sich ist schnell, sondern C++ besitzt eine Community die von sich behauptet schnellen und hocheffizienten Code zu schreiben. Python hingegen ist laut Selbstdefinition nur eine Sprache unter vielen. Python geht sehr viel unideologischer an die Dinge heran und koexistiert friedlich mit anderen Hochsprachen wie Java. Die C++ Developer hingegen führen einen Krieg gegen den Rest der Welt. Sie betrrachten C++ als die einzig gütlige Sprache und alles andere als Zumutung. Die Fallhöhe steigt dadurch natürlich an und der Anspruch an sich selbst auch. Und genau diese alles oder nichts Haltung ist auf der Cppcon zu spüren. Es finden sich dort Programmierer die sich nirgendwo integrieren sondern die die Wahrheit für sich gepachtet haben. C++ besitzt den Anspruch in jedem Bereich der beste zu sein.

Wie man vielleicht schon ahnt, besitzt die C++ Programmiersprache keine inhärenten Features um diesem Anspruch gerecht zu werden. Böse formuliert, ist C++ einfach nur ein kompiliertes Python, es gibt dort die selben miesen Befehle, und wenn man etwas in Software codieren will muss man mühsam den Code zu Fuß schreiben. Nur, damit hat die C++ Community ausgiebig Erfahrung sammeln können, die ersten 3D Spiele wurden schon in den 1990'er Jahren programmiert. Damals galt das Motto: entweder man realisiert es in C++ oder in keiner Sprache. Heute ist aufgrund der schnelleren Computer die Lage sehr viel entspannter, für grafiklastige Spiele kann man heute auf eine Vielzahl von Programmiersprachen zugreifen, aber eben auch auf C++ was immernoch da ist.

Produktivität Der Grund warum Python so viele Anhänger besitzt während C++ als die meist gehasste Programmiersprache gilt hat damit zu tun, dass man mit Python produktiver ist. Damit ist gemeint, dass man dort stressfreier ein Programm zum laufen bekommt und in kürzerer Zeit mehr Code schreiben kann. Python liest sich so ähnlich wie Pseudocode. Das mag zunächst wie eine gute Idee klingen, weil der Flaschenhals der Programmierer ist und wenn dieser produktiver wird umso besser. Leider hat Python einen großen Nachteil: die Produktivitätssteigerung ist nicht groß genug. Ich würde mal grob schätzen, dass man mit C++ pro Tag 8 Lines of Code netto schreiben kann (also Zuwachs für ein bestehendes Projekt) während mit Python pro Tag 12 LoC möglich sind; beides mal pro Kopf natürlich. Was man auch mit Python nicht hinbekommt sind täglich 100 LoC. Und das heißt nichts anderes, als das man ein Großprojekt wie ein größeres Spiel oder eine Robotersoftware nicht wird fertigbekommen. Will man 1M LoC schreiben und hat als Einzelperson pro Tag einen Output von 12 LoC, dann dauert es stolze 228 Jahre bis das Projekt abgeschlossen ist. Zu lange.

Eine Methode um die Produktivität sowohl von C++ als auch von Python zu erhöhen gibt es nicht. Der einzige Ausweg wie man größere Projekte auf den Weg bringt sind Betriebssystembibliotheken und die Zusammenarbeit im Team. Und das dürfte vielleicht verraten wiso einerseits die Produktivität von C++ Programmierern verglichen mit Python niedrig ist, aber gleichzeitig die wirklich großen Applikationen in C++ erstellt werden. So ähnlich wie C++ Developer Respekt vor ihrer Sprache haben, haben sie auch Respekt vor größeren Projekten. Sie wissen, dass man dort mindestens 1000 Leute für benötigt sonst braucht man gar nicht erst damit anzufangen. C++ Bibliotheken werden im Regelfall einmal geschrieben und werden dann 20 Jahre lang verwendet.

Warum die Produktivität in Python zwar höher ist als in C++ aber zu niedrig um als Einzelperson größere Projekte zu bearbeiten ist bisher ungeklärt. Schaut man sich den Sprachumfang von Python an, so hat der Erfinder der Sprache alles richtig gemacht. Es gibt für alles genau eine richtige Vorgehensweise, wenn der Interpreter eine Fehlermeldung ausgibt, ist sie sie selbsterklärend und um Integer-Variablen braucht sich der Programmierer auch nicht zu kümmern. .Die wesentlichen Stolpersteine von C++ wurden entfernt, das Programmieren geht reibungslos. Und tatsächlich, seit ich Python verwende gab es nichteinmal den Fall wo die Sprache an sich ein Hinderniss war. Nur, auch mit Python ist die Produktivität des Programmierers nicht unendlich hoch. Er muss immernoch Variablen benennen, Methoden aufrufen, Code überarbeiten, Bugs bearbeiten. Objektorientierte Programmierung ist zwar das leistungsfähigste Programmierparadigma was es aktuell gibt und mit keinem anderen Verfahren kann man effektiver programmieren, aber bis ein größeres Projekt fertig ist, vergehen viele Monate. Diese Erkenntnis ist zu selten verbreitet, gerade Anfänger welche mittels Copy&Paste ein Hello World Programm auf den Bildschirm bringen glauben instinktiv sie könnten am Tag 100 Zeilen Code schreiben. Wenn man jedoch seine eigene Produktität über einen längeren Zeitraum trackt und sich vor allem einmal größere Projekte wie Windows 10 oder den Linux Kernel anschaut wird man erkennen, dass selbst Profiprogrammierer die Standardprobleme lösen selten die Marke von 10 LoC/Tag überschreiten. Das heißt, in absoluten Zahlen sind sie unglaublich unproduktiv.

Wer schonmal selber programmiert hat wird ahnen woran das liegt, weil man bei 10 geöffneten Fenstern, plus Bugtracker plus Stackoverflow zum Nachschlagen sich manchmal schwertut und dann 30 Minuten herumsucht nur um Festzustellen dass irgendwo eine Klammer falsch gesetzt wurde. Und leider tritt dieses Phänomen sowohl bei Python als auch in jeder anderen Programmiersprache auf.

Nach der Cocomo Kostenschätzung geht man davon aus, dass eine Zeile Code rund 10 US$ kostet. Das ist extrem viel. Bevor man also anfängt größere Mengen an Code zu schreiben, sollte man sich eine Sprache aussuchen die für die Ewigkeit gedacht ist. Worin man also Betriebssystemroutinen schreibt und die auf maximale Geschwindigkeit hin optimiert wurde. Python kann man zwar etwas preiswerter programmieren als C++ aber nur allzuoft landet der fertige Code danach in der Mülltonne und wird nicht weiterverwendet.

Worauf es hingegen wirklich ankommt ist, Communities zu bilden. Derzeit gibt es ca. 20 Mio Programmierer weltweit. Wenn jeder davon nur 10 Zeilen Code am Tag schreibt, haben wir mehr Software als je benötigt wird. Das Problem ist nur, dass der meiste Code nicht dauerhaft ist, er findet nicht den Weg in Bibliotheken oder in konkrete Anwendungen. Und genau hier liegt das Geheimnis warum C++ so er-

folgreich ist. Im Regelfall tritt die C++ mit dem Anspruch an, dass ihr Code den Weg in die Systembibliotheken findet. Er also für die x86 Architektur nativ kompiliert wird und dann in Windows oder Linux als Routine bereitsteht um von anderen Programmen genutzt zu werden. Und das heißt, dass man den Code einmal schreibt und er danach dann verfügbar ist. Dieser Anspruch ist es, der im Laufe der Jahre dazu geführt hat, dass C++ überall Fuß fassen konnte, und das obwohl die Sprache viele Fehler enthält.

10.3 Produktivität von Python erhöhen?

Python gilt als die produktivste Programmiersprache. Gemessen in Codezeilen am Tag kann man mit keiner anderen Sprache mehr programmieren. Der Grund ist, dass in Python auf viele Stolpersteine wie Pointer und manuelle Variablendeklaratiionen verzichtet wurde und man nur noch Pseudocode hinschreibt. Was auf den ersten Blick wie eine sinnvolle Evolution hin zu einer besseren Programmiersprache klingt erweist sich bei näherer Betrachtung als Sackgasse. Das Problem ist nicht so sehr die heutige Produktivität in Python sondern die Tatsache dass man sie nicht mehr großartig steigern kann. Das Python 3 brachte gegenüber Python 2 praktisch keinerlei Verbesserung in dieser Richtung, man muss immernoch manuell Klassen definieren und for-Loops schreiben. Früher in den 1990'er hat man einmal an Programmiersprachen der 5. Generation geforscht, sogenannten deklarativen Sprachen. Die Idee war es, einfach nur den Unittest für eine Methode zu schreiben und dann auf Knopfdruck die Methode von allein zu erzeugen (Stichwort Genetic Programming). Obwohl es für Python derartige Ansätze gibt, wie pyevolve, ist das Konzept jedoch weit von der Praxisreife entfernt. Böse Stimmen sagen, dass auch in 20 Jahren noch immer die imperative objektorientierte Programmierung das wichtigste Programmierdogma ist und es immernoch nicht möglich ist, Code automatisch zu erzeugen. Interessanterweise ist auch ein weiteres Konzept gescheitert, und zwar aus UML Klassen Code zu erzeugen. Auch dazu gibt es zwar Forschungesprojekte aber konkret einsatzfähig ist es nicht. Offenbar ist Programmieren deutlich komplexer als angenommen und der Man-in-the-loop unvermeidbar. Womöglich ist es prinzipiell nicht möglich, das OOP Paradigma zu verlassen egal welche Hochsprache man verwendet.

Und hier wird die Tragik von Python deutlich. Es ist unzweifelhaft eine High-Level-Language. Die Verwendung einer reduzierten Syntax plus die Verwendung einer virtuellen Maschine ist ziemlich advanced. Aber es ist nicht so gut um von dort aus weitere Verbesserungen anzustreben. Auch mit Python ist der Programmierer eingesperrt in eine Produktivität von 10 LoC/Tag. Es ist nicht möglich in Python 100 LoC/Tag zu erzeugen.

Die Schwierigkeit ist, dass imperative OOP Sprachen bottom up funktionieren. Das heißt, man fängt mit einer Game-Klasse an, wo man nur die GUI anzeigt, baut dann eine zweite Klasse hinzu, wo man die Figuren auf dem Spielfeld speichert und baut das Programm dann zu einem Spiel aus. Das ist der gängige Programmierstil und offensichtlich ist er Alternativlos. Alle Versuche aus Meta-Informationen Sourcecode zu erzeugen um darüber die Produktivität drastisch zu erhöhen sind gescheitert. Und das bedeutet es ist Zeit sich auf die Kernelemente zu fokussieren. Eine alte wie mächtige Möglichkeit die Prodktivität von Programmierern zu steigern ist die Verwendung von Biblitoheken und die Zusammenarbeit im Team. Darüber kann man wirklich große Projekte realisieren. Leider gibt es ein Problem. Je mehr Programmiersprachen es gibt, desto schwerer ist die Teamarbeit. Die beste Methode die Produktivität wirklich zu steigern ist es, wenn man nur noch eine Programmiersprache verwendet. Selbst wenn das Programmieren in dieser Sprache etwas langsamer geht als in Python wäre die Gesamtleistung höher. Welches könnte die einzige Sprache sein? Python mit Sicherheit nicht. Es ist nur yet-another-language, go ist es auch nicht. Go ist nicht konkurrenzfähig mit C. Und Java als einzige Progarmmiersprache ist auch keine gute Idee. Welche Sprache als Universalsprache übrig bleibt dürfte auf der Hand liegen.

Das Problem mit Python ist, dass Python zwar eine Verbesserung gegenüber anderen Sprachen besitzt. Es ist beispielsweise gut portabel und man kann darin schnell Code schreiben, aber leider ist Python sehr ähnlich wie C++ aufgebaut. Beide Sprachen sind imperativ und objektorientiert. Python wurde so konzipiert sich von C++ zu emanzipieren, dass also die Leute lieber darin die Programme schreiben bevor sie große C++ Compiler in Stellung bringen und sich durch die Manuals kämpfen. Aber das sind nur Detailverbesserungen. Den eigentlichen Flaschenhals konnte Python nicht beseitigen.

Rein nominell gibt es in der Computerbranche ein Paradox. Auf der einen Seite gibt es weltweit 20 Mio Programmierer die genug Code schreiben, auf der anderen gibt es nach wie vor eine Softwarekrise wo Sourcecode ein äußerst knappes gut ist. Nach wie vor gibt es die meiste Software nur gegen Geld und im Bereich Robotik-Software gibt es noch nichtmal kommerzielle Software, sondern es gibt überhaupt keine Software. Meiner Meinung nach besteht die Produktivitätsbremse Nr.1 im Sprachenwirrwar was sich im Laufe der Zeit etablieren konnte. Es gibt heute Programmiersprachen für fast alles, die untereinander inkompatibel sind. Sogenannte Wrapper haben das Problem nur weiter verschärft. Wir haben also die seltene Situation, dass 20 Mio Developer weltweit nicht im Stande sind den Bedarf nach Software zu decken, so dass immere weitere Entwickler ausgebildet werden. Für die kommenden Jahre wird geschätzt dass die Zahl auf 40 Mio ansteigt. Aber auch die werden nicht im Stande sein gute Betriebssysteme und autonome Autos zu programmieren weil sie ihre Manpower auf sehr viele Sprachen aufteilen anstatt sie zu bündeln.

1 Mio Developer sind ausreichend Machen wir eine kleine Überschlagsrechnung. Angenommen man hat 1 Mio Entwickler weltweit die in C++ 10 Codezeilen am Tag produzieren. Im Jahr macht das die Summe von 3,6G LoC. Das ist ausreichend um Betriebssysteme, Spiele und Robotik-Software neu zu schreiben und die Bugs zu beheben. Effektiv reichen also 1 Mio Entwickler vollkommen aus, um den weltweiten Bedarf nach Software zu befriedigen. Und das obwohl der einzelne mit einer lächerlich geringen Produktivität arbeitet.

Was wir stattdessen sehen ist, dass es derzeit 30 Mio Entwickler gibt, die in 500 unterschiedlichen Programmiersprachen programmieren und es nichtmal schaffen die einfachsten Anwendungen zu realisieren. Das neue Windows 10 hat Fehler, Gnome unter Linux funktioniert nicht, und das ROS Betriebssystem ist einfach nur eine Katastrophe. Schauen wir uns nochmal die Sprache C++ genauer an. Was genau fehlt dort? Welches Feature muss dorthinein um die Produktivität anzuheben? Gar nichts fehlt da, C++ enthält alles was man braucht. Man kann damit lowlevel wie highlevel Programmieren, man kann damit Betriebssysteme wie auch Spielebibliotheken programmieren. Ja man kann C++ sogar in einer virtuellen Maschine ausführen. Das Problem ist nur, dass es nicht gemacht wird. Stattdessen werden alle möglichen Sprachen gepusht, an den Universitäten gerne Haskell oder Java, in Firmen hingegen C# und einige Apple Angestellte haben tatsächlich Swift entwickelt in der Hoffnung, dass damit alles besser wird.

Zugegeben, ich selber bin ein dezidierte C++ Hater, ich mag die Sprache nicht besonders, das Problem ist nur, dass es ohne C++ nicht geht. Man kann sie nicht durch etwas besseres ersetzen.

Im neuen C++11 Sprachstandard gibt es das Wort "auto". Python Programmierer werden darüber nur müde lächeln, weil man dort ohnehin nicht angeben muss, ob ein Int oder Boolean deklariert wird. Auto macht C++ also ein wenig Python kompatibler. Und vielleicht gibt

es in einer zukünftigen C++ Version sogar die Möglichkeit auf Deklarationen komplett zu verzichten und die Sprache nähert sich damit noch mehr Python an. Was bedeutet es konkret? Es bedeutet, dass C++ Programmierer dann nicht mehr 8 Zeilen Code täglich schreiben, sondern 8,5. Es ist eine Verbesserung aber noch immer ist der einzelne Programmierer gezwungen mit anderen im Team zu arbeiten wenn er wirklich große Projekte umsetzen möchte. Und das ist ein wenig die Schwäche der heutigen Informatik. Man kann das Programmieren nicht grundlegend neu erfinden. Am Ende hat man immer eine imperative OOP Sprache.

Den eigentlichen Produktivitätskick erreicht man nicht durch die Programmiersprache sondern durch den Kontext wo sie eingesetzt wird. Wenn es gelingt, mit 1000 Leuten die alle die selbe Programmiersprache beherschen ein Projekt durchzuziehen kann man dort rechnerisch 1000*10 = 10k LoC/Tag schreiben. Das ist sehr viel. Man kann da quasi stündlich zusehen wie die Anwendung immer komplexer wird. 'Genau nach diesem Modell werden übrigens reine C++ Projekte durchgeführt. Man kann so zu überschaubaren Kosten beliebige Mengen an Code erzeugen. Und zwar Code der inhaltlich bereits getestet ist und auf der Hardware nahe am Optimum ausgeführt wird.

Arbeit im Team Üblicherweise werden die Vorteile und Nachteile von C++ anhand der Sprachdefinition festgemacht. So wird verglichen wie man in C++ ein Array deklariert, während es bei Python leichter geht. Dieser Vergleich zielt auf den Einzelnen Programmierer ab, also was er letztlich beachten muss wenn er sein Minispiel entwickeln möchte. Nur, Softwareentwicklung hat fast nichts mit Technologie zu tun, sondern worum es geht ist Teamarbeit. Der Grund dafür ist, dass es bis heute nicht gelungen ist automatische Codegeneratoren zu entwickeln und man stattdessen immernoch menschliche Programmierer einsetzt. Schauen wir uns einige Großprojekte an: der Linux Kernel hat 20M LoC, Windows 10 hat 50M LoC, ROS besteht aus 7M LoC und eine Game-Engine wie Unity ist ebenfalls oberhalb von 1M LoC verortet. Genaugenommen wird also Software nicht in C, Java oder C++ programmiert sondern Software entsteht in sozialen Strukturen, also mittels github, bei Microsoft oder in Projekten die von Redhat gesonsert werden. Die eigentliche Keytechnologie um Software zu entwickeln liegt in einer gemeinsamen Programmiersprache plus Versionsverwaltungssystem. Es geht nicht darum, ob C++ die beste Sprache ist, sondern es geht darum, dass sie von mehreren Leuten eingesetzt wird, die die Änderungen dann auf einen git-Server pushen.

Unumstritten ist ein C++ eine Universalsprache die sowohl Lowlevel als auch High-Level eingesetzt werden kann. Und mit C++ kann man jede andere Sprache ersetzen. Manchmal mit Reibungsverlusten aber technisch gesehen geht es. Demzufolge ist C++ der kleinste gemeinsame Standard auf den sich alle einigen können. und dadurch wird aus der Sprache C++ eine Software-Community. Also Leute die darum herum gemeinsam Software erstellen.

Einige sagen, dass nicht C++ sondern dessen Vorgänger C der kleinste gemeinsame Nenner wäre. Doch diese Sichtweise halte ich für überholt. Mit C kann man nicht objektorientiert programiert und OOP ist wiederum zwigend erforderlich wenn man größere Spiele erstellen möchte.

C++ verbessern Schaut man sich aktuellen Sourcecode in C++ an so fällt einem Python Programmierer auf, dass man da vieles leserlicher machen könnte. Und tatsächlhich, was das betrifft ist C++ in der Tat im Nachteil. Aber nehmen wir mal an, man bessert nach und verändert C++ so dass ähnlich wie Python aussieht, wieviel kann man dadurch gewinnen? Doch nur sehr geringe Verbesserungen. Eine Klassendefinition sieht ohne C++ leicht übersichtlicher aus, vielleicht fehlen dann dann die Deklarationen und man spart 10% der Zeilen ein, aber es bleibt immernoch eine Klassendefinition. Sie besteht aus 10 willkürlichen Methoden, mehreren Arrays und implementiert mehrere komplexe Algorithmen. Das heißt, selbst wenn man C++ einsteigerfreundlicher macht oder sich der Programmierer an die Syntax gewöhnt wird er immernoch genug Probleme haben den Sourceocode zu verstehen. C++ ist zwar eine gruselige Sprache aber so viel schlimmer als Python ist sie nicht.

Python Programme sehen nur schick und übersichtlich aus, wenn sie weniger als 200 Lines of Code besitzen. Größere Python Programme mit mehr als 10k LoC liegen bereits gleichauf mit einem entsprechenden C++ Programm was die Lesbarkeit angeht. Und oberhalb von 50k LoC gibt es Gleichstand. In der Art, dass Neueinsteiger in das Projekt erstmal sehr viel in den Dateien herumscrollen müssen und sich weitere Schaubilder zeichnen bevor sie die erste Zeile Code produktiv schreiben können.

Nicht C++ macht die Welt unübersichtlich sondern es sind Programme die länger sind als 50k LoC.

10.4 Künstliche Intelligenz in C++

Wenn es an das Thema Künstliche Intelligenz geht werden C++ Programmierer nachdenklich. So richtig scheint niemand die Antwort zu wissen wie genau man sowas einprogrammiert. Dabei ist es eigentlich nicht so schwer. Als erstes benötigen wir ein normales C++ Spiel. Das kann im einfachsten Fall mit SFML worden sein und sollte butterweiches Scrolling besitzen sowie über eine spritzige Musiik verfügen. Um für dieses Spiel eine AI zu programmieren benötigen wir eine sogenannte Wissensbasis. Diese besteht aus C++ Klassen welche als UML Notation visualisiert werden. Die Klassennamen richten sich nach der konkrete Domäne. Bei einem Fußballspiel benötigt man andere Klassen als bei einem Racing Game. In diesen Klassen werden Zusatzfunktionen implementiert die im eigentlichen Spiel noch nicht enthalten sind. Das betrifft zwei Bereiche:

- Head-up Display
- Intelligent Tutorial System

Man kann sich das so vorstellen, dass man ein Racing-Game um didaktische Elemente erweitert. Also einen tutor programmiert wo der Spieler das Fahren erlernen soll. Wenn die Lektion 1 lautet, dass man den Motor anlassen sollen und bis zur Markierung fahren soll, dann muss im Worldmodel als Klasse definiert sein, was ein Motor ist und was eine Markierung ist. Da im ursprünglichen Spiel diese Elemente fehlen muss man sie manuell einprogrammieren. Ein Head-up Display wiederum dient dazu, Zusatzinformationen in das Spielgeschehen einzublenden, beispielsweise über mögliche Kollisionen mit Gegnern. Eine Künstliche Intelligenz erhält man wenn man das semi-autonome Spiel was bereits ein UML Domainmodell besitzt mit einem Script automatisiert. Das Script benutzt die angelegten Klassen und High-Level-Definitionen um konkrete Aktionen auszuführen. Beispielsweise fährt es zur Markierung, hält an einer roten Ampel an und vermeidet Kollisionen.

Das Programmieren einer künstlichen Intelligenz bedeutet für C++ Programmierer vor allem, dass sie viele neue Klassen anlegen müssen die untereinander erben können. Es führt dazu, dass sich der Codeumfang erhöht und höherwertige Befehle bereitgestellt werden. Wenn im ursprünglichen Spiel lediglich das Auto mit up,down,left und right gesteuert werden konnte ist in der aufgerüsteten Künstlichen Intelligenz auch das Ansteuern einer beliebigen Zielkoordinate inkl. Ausweichen bei Hindernissen als C++ Klasse implementiert.

10.5 Warum C++ eine ausgezeichnete Idee ist

Der Neueinsteiger wird zwar mit etwas Mühe ein lauffähiges C++ Programm hinbekommen sogar inkl. GUI wird aber bemerken, dass die Programmierung ausgesprochen langsam von der Hand geht. Er muss unzählige Compiler-Fehler beheben und ziemlich in die Details gehen. Das führt dazu, dass sowohl seine subjektiv empfundene aber auch die objektiv messbare Produktivität absinkt. Konkret führt das dazu, dass Neueinsteiger in C++ am Tag vielleicht 5 Zeilen neuen Code schreiben.

Es ist vollkommen unverständlich warum sich jemand soetwas freiwillig antut – so die weitverbreitete Meinung und es liegt auf der Hand sich von C++ abzuwenden zugunsten einer Sprache mit der sich leichter programmieren lässt. Allen voran Java, C# und ganz besonders Python. Python ist so ungefähr das genaue Gegenteil von C++. Auch in Python kann man objektorientiert und imperativ programmieren benötigt aber weniger Code und kann diesen auch noch schneller erstellen. Wiederum sind das sowohl subjektive als auch objektive Maßzahlen. Dennoch ist am Ende die Produktivität in C++ höher. Damit ist nicht etwa die gemessene Geschwindigkeit der Programmausführung gemeint sondern in welcher Zeit man Code erstellt hat. Am schnellsten bekommt man eine Software dann zum Laufen wenn man überhaupt nichts programmiert, also anstatt den Compiler zu bemühen den Paketmanager seines Vertrauens um dort mit einem flinken apt-get install die benötigte Software einfach nachinstalliert. Und wenn das nicht möglich ist, sollte man als zweitbeste Lösung auf vorhandene Bibliotheken aufsetzen, also auf OpenGL, Unity3D usw. um darüber dann ans Ziel zu gelangen. In solchen Fällen ist die Produktivität am allerhöchsten. Und genau hier liegt der Schlüssel um Python mit C++ zu vergleichen. Rein formal kann man in Python ausgewachsene Libraries erstellen. Doch es gibt ein Problem: Libraries werden traditionell in C/C++ geschrieben weil man damit die maximale Performance erzielt. Auch Box2D wurde in C/C++ geschrieben und OpenGL sowieso. Wenn man sich jetzt dranmacht, eine umfangreiche mathematische Bibliothek zu schreiben und man dafür die Programmiersprache Python verwendet wird eines ganz sicher nicht passieren. Das irgendein Debian Packagemaintainer aus Versehen auf die idee kommt, das als Standard-Library mit auszuliefern. Sondern er wird entweder eine vergleichbare C++ Bibliothek auswählen oder er wird überhaupt keine Bibliothek zum Betriebssystem mitliefern. Und genau das ist die Stärke von C++. Das ist jene Sprache mit der sich Systembibliotheken schreiben lassen. Diese wiederum haben den Vorteil, dass andere Programmierer darauf aufbauen können. Wenn man also nicht nur seine eigene Produktivität im Blick hat sondern über den Tellerrand blickt ist C++ die bessere Sprache.

Ein fairer Vergleich C++ gegen den Rest der Welt geht immer so aus, dass C++ natürlich gewinnt. Der Grund ist, dass man in Java, Python, PHP oder go keine systemnahen Bibliotheken schreiben kann und vor allem keine die in Assemblercode compiliert werden. Man kann zwar in Python umfangreiche Software erstellen, aber die ist für die Tonne. Außer einem selbst führt sie niemand sonst aus. Und das wiederum ist extrem unproduktiv. Es bedeutet, dass man das Rad neu erfindet und wertvolle Ressourcen vergeudet.

Bis heute ist es nicht möglich, Python Sourcecode nach C++ automatisch zu konvertieren. Sondern man muss zwingend per Hand alles nochmal neu schreiben. Wenn man jedoch zunächst 1000 Zeilen in Python schreibt um das danach nach C++ zu konvertieren ist das wiederum doppelte Mehrarbeit. Man muss sich erstens mit C++ herumplagen und zweitens auch noch mit Python.

Wenn man stattdessen vernünftig C++ lernt und darauf seinen Workflow konzentriert wird man bemerken, dass man mit etwas gutem Willen durchaus viel Code schreiben kann. Nicht so locker flockig wie bei Python, aber OOP kann auch C++. Das wichtige bei C++ ist nicht so sehr der Sprachstandard, sondern dass man für eine Community programmiert, also Leute die C++ Code weiternutzen wollen. In den anderen Programmiersprachen hat man immer das Gefühl, als ob da jeder für sich arbeiten würde. Außerhalb der Java Welt will beispielsweise die Java Klassen niemand aufrufen und so ähnlich ist es auch mit Ruby. Beides sind sehr spezielle Programmiersprachen deren Reichweite gering ist. Wenn man hingegen mit viel Mühe und im Schneckentempo von 1 Codezeile am Tag eine C++ Bibliothek schreibt und es schafft die in einer Linux-Distribution unterzubringen können Millionen Anwender davon profitieren.

11 Agenten

11.1 Jadex Agent

Bevor ich zu Agenten-Frameworks komme noch ein kurzer Exkurs wie Künstliche Intelligenz grundsätzlich in Computerspiele realisiert wird. Üblicherweise werden die Spiele ohne eingebaute KI ausgeliefert. Das beste Beispiel ist Super Mario Bros oder der Klassiker Pong. Der Spieler hat die Aufgabe mit Hilfe des Joysticks den Charakter sicher durchs Level zu bugsieren. Gewissermaßen sind die meisten veröffentlichten Spiele also eine Plattform für die es erst noch eine KI zu entwickeln gilt. Also einen Agenten der das Spiel alleine spielt. Rund um diese Aufgabe hat sich eine regelrechte Community gebildet welche aufbauend auf der AutoIt Programmiersprache versucht solche Spiele zu automatisieren. AutoIt ist eine Scripting Sprache mit der sich Tastenbefehle an vorhandene Applikationen senden lassen und worüber Pixelwerte abgefragt werden können. Die Frage ist jetzt: wie programmiert man mit AutoIt für ein vorhandenes Spiel eine KI?

Die Antwort darauf ist bekannt. Man muss die Domäne des Spiels als UML Modell realisieren. Bei Supermario würde das UML Modell aus Klassen wie Pfad, Umgebung, Gegner usw. bestehen die um Methoden angereichert sind. Der aktuelle Spielzustand wird auf der untersten Ebene mit Hilfe von AutoIt grob geparst, um zu erkennen wo die Gegner sind, wo das Hinderniss und wo der Charakter. Eine Ebene darüber wird dann eine Event-Queue wo man hineinschreibt, dass Mario gerade vor einem Abgrund steht und dann braucht man weitere Klassen die einen reaktiven Planner beinhalten der ermittelt was zu tun ist.

Der Programmieraufwand für solche UML Modelle ist hoch. Er übersteigt den Aufwand für das Spiel selbst um einiges. Einen Pong Klone kann man locker in 200 Lines of Code realisieren, das dazugehörige UML Modell um den Paddel an die richtige Position zu bringen benötigt ein vielfaches davon. Und hier kommt die Jadex Agent Architektur ins Spiel. Jadex wurde entwickelt um den Zeitaufwendigen Prozess des Domain-Modelling zu erleichtern. Die Idee ist es, ein Framework zu schaffen mit dem man für beliebige Spiele solche Event-Queue, reaktiven Planner und Domain-Modelle erzeugen kann. Nicht ganz vollautomatisch aber einfacher, als wenn man alles from scratch programmiert. Jadex beginnt dort wo die meisten Computerspiele enden. Es geht darum für ein vorhandenes Computerspiel sinnvolle Keyboard-Eingaben zu erzeugen. Also die Frage zu beantworten, wann Mario springen muss und wann er sich ducken muss.

BDI In einer Präsenation zur Jadex Plattform wurde die grundlegende Struktur erläutert. Im Mittelpunkt stehen die Events, um sie herum werden Pläne, Beliefs und Goals angeordnet. Was ist damit gemeint? Die Basis um Künstliche Intelligenz für Spiele zu entwickeln ist ein Event. Ein Event stellt ein semantisches Mapping zwischen einer Lowlevel und einer Highlevel Aktion da. Ein Beispiel: wenn der Spieler auf einen Gegner trifft überlappen sich auf der unter-

sten Ebene zwei Rechtecke. Mathematisch gesehen gibt es zwischen den Pixeln eine Überschneidung. Im Hintergrund lässt man jetzt eine Loop laufen und testet auf das Eintreten dieses Ereignis. Gibt es eine Kollision schreibt man in den Event-Queue hinein: Frame432,Collision,Player-Enemy1. Aus einer Pixelüberschneidung wird dadurch ein semantisch angereichertes Ereignis. In Computerspielen treten solche Events permanent auf. Sie bilden die Grundlage um darauf aufbauend einen Agent zu konstruieren. .Also ein Computerprogramm was die Events als Input verwendet, um erst eine Umgebungsanalyse zu erstellen und darauf aufbauend dann Folgeaktionen zu berechnen.

Jadex im Detail Inzwischen hatte ich Zeit mir Jadex etwas genauer anzuschauen. Leider ist es nicht so sehr eine einsatzbereite Software sondern eine Anleitung wie man eigene Programme erstellt. Vergleichbar mit Behavior Trees, die ebenfalls nur sehr allgemein eine Sammlung von Symbolen sind um die konkrete Domäne zu visualisieren. Jadex geht über Behavior Trees weithinaus. Man kann es am ehesten mit der GOAP Architektur vergleichen. Man kann sich zwar eine zip-Datei mit der aktuellen Jadex Version 3.0 herunterladen die auch ausführbaren Java-Code enthält doch schaut man sich die Beispiele etwas genauer an wird schnell deutlich, dass man von dem Sourcecode exakt 0 für die eigene Domäne wiederverwenden kann. Am meisten Sinn macht Jadex noch, wenn das Schaubild nutzt, um seine eigene BDI Architektur zu erstellen. Das heißt konkret, dass man in seiner Lieblingsprogrammiersprache und unabhängig von Jadex eine Klasse für Events, eine für Beliefs, eine für Goals und eine für Plans erstellt und darin dann seine konkrete Domäne verwaltet. Genauer gesagt muss man also from scratch modellieren was man eigentlich vorhat mit dem Agenten. Ein BDI Agent der Auto fährt ist gänzlich anders aufgebaut als einer der Super-Mario spielt. Die Gemeinsamkeit ist nur, dass es in beiden Fällen eine Klasse Evens und eine für Beliefs gibt. Aber was man da konkret speichert und vor allem wie das mit den Plänen verbunden wird hängt stark von der Domäne ab.

Dennoch würde ich Jadex als nützlich bezeichnen, weil man zumindest anhand der Beipsiele sieht, wie grundsätzlich eine BDI Architektur aufgebaut ist. Im wesentlichen geht es darum, eine semantische Beschreibung der Domäne zu verwenden. Also Fakten aus der Wirklichkeit in Sourcecode abzubilden.

Jadex selber setzt auf Java auf. Und es übeschneidet sich mit dieser Programmiersprache. Das ist das Dilemma. Im Grunde kann man auch ohne Jadex eine BDI Architektur modellieren. Java bringt die benötigten Features bereits mit. Es läuft darauf hinaus, dass man in seinem Sourcecode Klassen erstellt und diese interagieren lässt. So ähnlich wie man Behavior Trees auch in normalem Java schreiben kann, ohne dafür einen Addon zu bemühen. Man schreibt einfach eine Klasse, definiert darin Methoden und das ist dann der Behavior Tree.

11.2 Domain-Knowledge

In nahezu jeder Agentenarchitektur taucht es irgendwann auf, das sogenannte Domain-Knowledge. Der Begriff ist in der Literatur nicht eindeutig definiert, meint aber üblicherweise eine Knowledge-Base. Wenn man sie erstellt hat, besitzt man ein UML Modell, also ausführbaren Sourcecode in der Sprache Java oder C++. Das UML Modell für die Domäne Soccer sieht anders aus, als für die Domäne Driverless Car. Insofern ist es schwer einen allgemeinen Rahmen vorzugeben. Doch es gibt ihn. Im Reallife wird Domain-Knowledge lexikonartig aufbereitet. In einem Lexikon über Super Mario Bros würde man ähnlich wie bei Wikipedia Kategorien erstellen, in den Artikel alphabetisch angeordnet sind. Man erhält eine Wissensdatenbank die man nach verschiedenen Stichworten durchsuchen kann. Man erfährt wie das Spiel funktioniert, welche Gegner es gibt, wie ein konkretes Level funktioniert usw. Leider gibt es mit derartigem Domain-Knowledge ein Problem: es liegt als natürlichsprachlicher Text vor, und das auch nur wenn fließige Gamer zuvor so ein Wiki erstellt haben.

Manchmal liegt Domainwissen noch nicht als dezidiertes Lexikon vor, sondern häufig finden sich auch nur Tutorials. Wikihow und ehow sind zwei Beispiele. Der Unterschied ist quantitativer Art. Das heißt, ein Wikihow Text ist kurz und knapp, während ein dezidiertes Wiki zu einem Thema umfangreicher ist. Aber auch wikihow Texte liegen als natürlichsprachlicher Text vor. Diese haben das Problem, dass sie eben nicht auf einem Computer ausführbar sind. Man muss also das Wissen noch mühsam in ein UML Modell überführen. Leider geht das nicht automatisch sondern das erfolgt in einem Software-Engineering-Prozess. Die Produktivität bei diesem Vorgang ist bekannt und sie ist niedrig. Ein Einzelprogrammierer arbeitet ungefähr mit einer Geschwindigkeit von 10 Zeilen Code am Tag. Man gibt ihm das natürlichsprachliche Wiki, lässt ihm 30 Tage Zeit und dann hat er 300 Zeilen Javacode geschrieben, was die ersten Klassen beinhaltet. Dieses UML Klassenmodell ist dann ausführbar.

Jetzt stellt sich natürlich die Frage wie man diesen Vorgang beschleunigen kann. Zunächst einmal ist es sich wichtig sich klarzumachen dass das fertige macshinenlesbare Domänmodell in einer objektorientierten Hochsprache wie Java oder C++ erstellt wird, während die Baseline an dem sich dieses Modell orientiert ein natürlichsprachliches Lexikon darstellt. Genau dazwischen befinden sich Programmierwettbewerbe wie “Starcraft AI”, Robocup oder “Mario AI” bei denen die Teams die Aufgabe haben zu einer konkreten Domäne Agenten zu programmieren. Bei solchen Wettbewerben gewinnt üblicherweise das Team, was am meisten der Domäne in Software übertragen hat.

Programmiersprachen Manchmal wird versucht, Domain-Knowledge in einer dezidierten höheren Programiersprache wie KQML (knowledge query and manipulation language) zu speichern. Leider ohne Erfolg, solche Modellierungssprache sind akademisch orientiert und nicht tauglich für den praktischen Einsatz. Was man hingegen tun kann ist, aus bestehenden Agents die bei “Mario AI” auf den vordersten Plätzen waren, rückwärts das UML Diagramm abzuleiten. Also aus dem C++ Code oder Java Code rückwärts das UML Diagramm zu erzeugen.

Das geht deshalb gut, weil C++ und java praxistaugliche Programmiersprachen sind. In ihnen kann man vollständig das Verhalten eines Agenten spezifizieren. Und ich wage einfach mal die These, dass es keine besseren Sprachen dafür gibt. Insofern ist C++ bereits eine universale Sprache um Domain-Knowledge machinenlesbar zu speichern. Es bleibt jetzt noch die Frage zu beantworten wie man das eigentliche Codieren effizienter durchführt, also wie man schneller aus einem Wikihow Text die dazugehörige C++ Datei erzeugt.

Im Grunde läuft diese Zielstellung auf automatic Programming hinaus. Ein Konzept, bei dem man den Code nicht direkt schreibt, sondern ihn aus allgemeinen Spezifikationen heraus generiert:

> “Program synthesis is the process of automatically deriving executable code from (non-executable) high-level specifications.” [4]

Der Wikihow Text wäre der Unit-Test gegen den man den Solver laufen lässt der den eigentlichen Sourcecode erzeugt. Leider gibt es ein Problem: Automatic Programming ist ebenfalls nur eine akademisch-theoretische Disziplin die sich nicht in der Praxis einsetzen lässt.

Program synthesis Meine persönliche Meinung ist, dass Program synthesis nicht funktioniert. Will man ein natürlichsprachliches Lexikon

in ein ausführbares UML Modell übertragen benötigt man menschliche Programmierer. Um die Produktität zu erhöheh muss man sich auf die Community der C++ und Java Developer fokussieren. So ähnlich wie es möglich ist, Game Ressourcen wie animierte Sprites und wiederverwendbare Wav-Dateien auf Community Portalen bereitzustellen ist es möglich, komplette UML Modelle allgemein bereitzustellen. Die best-practice Methode besteht folglich darin, dass man einen ausprogrammierten "Mario AI" Agenten anschließend bei github hochlädt, damit ihn andere weiterverwenden können. Das ist das wichtigste Prinzip wenn man die Effizienz erhöhen möchte.

Natürlich ist das keine echte Program-Synthese, es gibt eben keinen magischen Generator der aus einer Spezifikation den Sourcecode erstellt, sondern es läuft eher darauf hinaus, eine Plattform bereitzustellen auf der C++ Programmierer ihre Ressourcen bündeln. Der Vorteil ist, dass diese Methode praxistauglich ist. Man also in absehbarerer Zeit und wiederholbar für beliebige Domäne einen Agenten erstellt.

Einge "Mario AI" Agent sind tatsächlich bei github im Sourcecode verfügbar.[13] Die Gemeinsamkeit aller Einreichungen besteht darin, dass sie manuell programmiert wurden. Grundsätzlich orientieren sich die Programmierer zwar an bekannten Prinzipien wie Reinforcement Learning, A* Search, BDI Architektur und State Machines aber im Resultat sieht das so aus, dass sie sich hinsetzen, ihre Eclipse IDE starten, dort Sourcecode erstellen und das ist dann der Agent-Controller.

11.3 UML Aktivitätsdiagramme

Domainenwissen lässt sich mit Hilfe von UML formalisieren. UML besteht aus Diagramm-Notationen. Das bekannteste ist das Klassendiagramm, aber ebenfalls mächtig ist das Aktivitätsdiagramm. UML Diagramme haben den Vorteil, unabhängig von Programmiersprachen zu funktionieren. Man kann darin Abläufen sehr allgemein darstellen. Womöglich bildet die UML Notation einen Zwischenschritt zwischen einem natürlichsprachlichen Lexikon wie Wikihow und einem ausführbarem Programm in C++. In [19] werden Suchmaschinen vorgestellt um Prozessdiagramme im Internet zu suchen. Leider ist das Paper sehr allgemein gehalten und ich konnte nur die ersten 24 Seiten einsehen. Im wesentlichen geht es wohl darum, dass man nicht allein auf Wikihow aufbaut, sondern stattdessen Repositorien mit formalen Workflows in der UML Activitydiagramm Syntax verwaltet. Diese Diagramme haben den Vorteil, dass sie einerseits den Prozess beschreiben, aber sowohl für Maschinen als auch für Menschen lesbar sind. Anders als natürlichsprachlicher Text können solche Diagramme sehr viel einfacher geparst werden. Das bemerkenswerte an dem Paper ist, dass es den Begriff der "Process base" einführt. Gemeint ist nicht ein einzelner Prozess, sondern eine Sammlung von vielen Prozessen um eine Domäne vollständig zu beschreiben.

In [20] wird der Begriff der "Process based search engine" näher erläutert. Darin wird zunächst erläutert wie eine Dokumentenzentrierte Suchmaschine wie Google funktioniert: Die Anfrage wird dadurch beantwortet, dass ein einzelnes Dokument zurückgegeben wird. Bei der empfohlenen Process-based engine funktioniert die Suche wie bei einem Solver. Der Anwender gibt ein Zielzustand ein wohin er gerne möchte, und die Suchmaschine sucht in den vorhandenen Prozess-Diagrammen nach einem Weg dorthin.

Es gibt jedoch ein Problem. Die UML Aktivitätsdiagramme können nicht automatisch erzeugt erstellt, will man sie für eine Domäne wie "Mario AI" haben, muss man sie vorher manuell hinschreiben. Ein Prozess-Modell kann man sich als Prototypen vorstellen, also eine grobe Vorlage wie das Programm einmal aussehen soll. Es gibt hier zwei Möglichkeiten: entweder man schreibt direkt den C++ Code hin, das dauert jedoch lange. Oder man erstellt zuerst das Prozess-Diagramm und leitet daraus den Sourcecode ab. Dieses Vorgehen wird gerne an Universitäten unterrichtet. Angeblich könnte man nach dieser Methode auch UML Klassendiagramme in ausführbaren Sourcecode überführen. Leider ist das Verfahren nicht praxistauglich. Wer schonmal versucht hat, aus einem UML Diagramm vorwärts den Sourcecode zu generieren wird erkennen, dass es nicht funktioniert. Ja, die best-practice Methode in der agilen Softwareentwicklung besteht gerade darin, diesen Zwischenschritt auszulassen und direkt aus dem natürlichsprachlichen Text den Sourcecode zu schreiben.

Das Reverse Engineering hingegen, also ausgehend vom C++ Sourcecode das UML Aktivitätsdiagramm zu erzeugen, macht Sinn und die damit gewonnen Diagramme sind akurat. Es gibt mehrere Ansätze wie Flowgen [17]und Moritz. Letzteres läuft zusammen mit Doxygen und wird zur Dokumentation von existierendem Sourceocode eingesetzt. Wenn man sich damit erzeugten UML Aktivitätsdiagramme einmal anschaut wird deutlich warum es keinen Sinn macht, sie from scratch also ohne C++ Sourcecode zu schreiben. Sie sind extrem komplex aufgebaut. Üblicherweise benötigen sehr einfache Prozesse mehrere DIN A4 Seiten, und der Workflow besteht aus mehreren Subtasks. Der Ablauf um solche UML Diagramme zu erzeugen ist folgender:

Wikihow -> C++ Code -> UML Aktivitätschart

Das mag überraschen, weil viele Paper aus dem akademischen Mileau folgenden Ablauf beschreiben: Wikihow -> UML Aktivitätschart -> C++ Code Das heißt, das Prozessmodell wird als Zwischensprache verstanden um damit die Programimerung zu erleichtern. Nur wie schon weiter oben angedeutet, ist diese Ansicht weltfremd. Es funktioniert nur auf dem Paper, in der Praxis hingegen wird Variante 1 verwendet.

11.4 Agenten schreiben in C++

Um einen Agent zu schreiben geht man wie folgt vor. Zuerst surft man zu github, gibt dort ins Suchfenster ein "agent game", schränkt dann die Suche ein auf die Programmiersprache C++ und sieht jetzt 52 Agents die zur Auswahl stehen. Für das Spiel seiner Wahl, z.B. Torcs ruft man jetzt den Agenten auf, liest sich die Readme Datei durch und installiert den Agent mit Hilfe des Makefiles. Wenn einem das Konzept gefällt, forkt man das Projekt und erweitert den Agent.

Dieses Mini-Tutorial klingt banal, weil es nichts zu tun hat mit neuronalen Netzen oder mit BDI Frameworks, hat aber den Vorteil dass es in der Praxis funktioniert. Das erstellen von Agent ist eine soziale Aufgabe. Sie hat etwas damit zu tun, dass man mit github interagiert und auf dem Code aufbaut der schon existiert. Was man darüberhinaus noch tun kann ist es, bei wikihow nach Tutorials zu suchen wie man das jeweilige Spiel spielt. Beispielsweise hat Wikihow mehrere Tutorials wie man Racing-Games spielt. Teile dieser Anleitung kann man versuchen in C++ zu programmieren. Das Hauptproblem mit den meisten Agenten bei github ist, dass sie mangels Manpower nur einen sehr geringen Umfang besitzen. Üblicherweise ist der Codeumfang kleiner als 500 Lines of Code. Das ist natürlich viel zu niedrig um die Domäne vollständig zu beschreiben. Der Grund dürfte darin liegen, dass es sich um 1-Mann-Projekte handelt, wo also die Programmierer from scratch ihre Dateien erstmal ins Netz gestellt haben und jetzt darauf warten dass jemand das Projekt forkt um es zu erweitern. Genau das ist der Part, den man selber machen kann. Wenn man einen Agent um weitere 500 Codezeilen ergänzt, wird seine Spielstärke davon profitieren und vielleicht freut sich auch der ursprüngliche Autor über soviel Interesse aus der Community.

[13] https://github.com/jumoel/mario-astar-robinbaumgarten
https://github.com/search?utf8=%E2%9C%93&q=%22mario+ai%22&type=

11.5 Visualisierung der BDI Architeltur

Bei den gängigen Agentenframeworks wie Jadex wird das vorhandene Weltwissen meist textuell darstellt. Man sieht auf der Konsole eine Ausgabe wie “enemyahead=true”. Das ist nicht optimal. Besser wäre es, wenn die Beliefs und Goals des Agenten grafisch dargestellt werden. Aber wie bitteschön visualisiert man abstrakte Ziele? Die Antwort lautet: UML. In der UML Syntax gibt es Diagrammarten für alles mögliche. Ziele könnte man als Use-Case diagramm darstellen, das Weltwissen als Class-Diagramm.

Anders ausgedrückt, ein hochentwickeltes Agent-Framework besitzt eine Ausgabefunktion bei der die internen Goals als UML Diagramm grafisch dargestellt werden. Inhaltlich bleibt alles beim alten, sondern neu ist lediglich dass in Echtzeit eine hübsche Grafik auf den Bildschirm ausgegeben wird. So kann der User wie auch Programmierer mitverfolgen, was der Agent gerade denkt und was er von der Umwelt weiß.

Rein technisch ist die Realisierung simpel: angenommen, die Goals werden einfafch in einem Array gespiechert, also hintereinanderweg und jeweils ein simples Wort. Dann würde der UML Visualisizer dieses Array lediglich auslesen und dafür dann ein Schaubild zeichnen was dem Usecase Diagram entspricht. Es wird also keinewegs der Sourcecode des Agenten selbst visualisiert, sondern UML wird zweckentfremded um Werte aus Tabellen anzuzeigen.

11.6 BDI Architektur vs. subsumption Architektur

Eine BDI Architektur lässt sich auf vielfältige Weise realisieren. In seiner optimalen Form mit Strips und PDDL. Damit ist ein sehr altes Konzept eines symbolischen Planners gemeint. Der Ablauf im Detail: Angenommen, ein Auto kommt an eine Kreuzung. Dann werden im Event-Queue folgende Ereignisse aktiviert: Auto-an-Kreuzung, Ampel-ist-rot, VorAuto-nocheinAuto, Fussgänger-aufFahrbahn. Dieser Event-Queue transformiert die reale Welt in eine symbolische Beschreibung. Es gibt nicht länger 2 LIDAR Sensorbilder und eine HD Kamera, sondern die Welt ist mit den Events vollständig beschrieben. Die Idee hinter der BDI Architektur bzw. PDDL/Strips lautet nun, ausgehend von der Ist-Situation einen Pfad zum Ziel zu bestimmen. Das Ziel könnte lauten, dass das Auto geradeausfahren will, und damit es das kann müssen Randbedingungen eingehalten werden: Warten bis die Ampel grün wird, Abstand zum Vordermann korrigieren usw.

Leider zeigt sich hier ein grundlegendes Problem was symbolische Planner haben: man muss die komplette Welt darin formulieren. Wenn der Agent kein Modell besitzt kann er auch keine Entscheidungen treffen. Nur, wie will man die Reale Kreuzung als symbolisches PDDL Model beschreiben? Richtig, genau daran scheitert es. Es funktioniert nicht. Sowohl die BDI Architektur als auch PDDL funktioniert nur bei synthetischen akademischen Beispielen, wo die Welt sehr übersichtlich ist.

Schauen wir uns jetzt an, was Brooks mit seiner Subsumption Architektur vorgeschlagen hat. Er hat gesagt, dass man auf PDDL, BDI und GOAP verzichten kann. Also keinen symbolischen Planner benötigt. Ohne Planner braucht man auch kein symbolic Worldmodel. Es bleibt natürlich noch die Frage zu beantworten wie jetzt im obigen Beispiel das Auto über die Kreuzung kommt. Die Antwort ist simpel: gar nicht. Es ist nicht möglich, aus den Events eine Handlung für den Agenten abzuleiten, stattdessen gibt es einen Event-Queue Überlauf und das Fahrzeug macht vorsorglich den Motor aus und aktiviert die Warnblinkanlage, als Zeichen mit der Situation komplett überfordert zu sein.

Natürlich war das ein kleiner Scherz. Versuchen wir etwas sachlicher auf die Nachteile von PDDL einzugehen. Grundsätzlich handelt es sich dabei um einen symbolic Planner. Damit ist gemeint, dass die Software Entscheidungen trifft auf grund der Ist-Situation. Und genau das ist das Problem. Eine AI die eigenständig Entscheidungen trifft ist genau das was zu vermeiden ist. Besser ist es, Nonplanning Ansätze zu verwenden. Konkret statische Behavior Trees, Also vordefinierte Taskmodelle die nur genau in der Reihenfolge abgearbeitet werden, wie sie zuvor manuell einprogrammiert wurden. Und wenn das Taskmodell auf die Situation nicht passt, bricht das Programm mit einer Fehlermeldung ab. Obwohl das für viele Visionäre der Künstlichen Intelligenz schwer verständlich ist, sind Meta-Programming Konzepte und vollautonome Planner nicht erwünscht. Wir wollen ja eben nicht, dass die Software anfängt eigenständige Pläne zu entwickeln oder Lösungen findet an die die Programmierer nicht gedacht haben, sondern wir wollen dass die Software ein vorhandenes UML Modell penibel abarbeitet. Anders gesagt, deklarative Programmierung gilt es zu vermeiden, stattdesen ist konsequent auf imperative Programmierung zu setzen. Das bedeutet für die eingangs beschriebene Kreuzungssituation dass man dafür natürlich eine eigene Klasse benötigt. In dieser werden mögliche Aktionen definiert, die man in einer Kreuzung machen sollte. Beim Heranfahren an die Kreuzung wird die Geschwindigkeit reduziert, dann wird die Ampel ausgewertet, dann die richtige Abbiegespur gewählt usw. Der Vorteil von derlei Programmierstil ist, dass es etwas ist was man debuggen kann. Das heißt, es gibt eben keinen Planner der sehr frei entscheidet was zu tun ist sondern die Software ist klar definiert.

Anders formuliert, ich halte symbolic Planner für generell den falschen Ansatz. Das passt nicht zusammen, weil die Domäne um die es geht, viel zu umfangreich ist, um sie mit einigen PDDL Statements beschreiben zu können. Planning, das klingt nach einem sehr kurzen Programm, wo in 100 Zeilen Code die optimale Handlung ermittelt wird. Nur leider haben kurzer Programme immer das Problem, dass sie nicht praxistauglich sind.

Verkehrswissenschaft Weiter oben habe ich mich skeptisch gezeigt, was die Leistung von symbolischen Plannern angeht. Das generelle Problem besteht darin, dass die Informatik überfordert ist, zu beurteilen wie sich ein Auto korrekt an einer Kreuzung verhalten muss. PDDL, Solver und soweiter helfen da nicht weiter. Was man braucht sind Verfahren die außerhalb der Informatik entwickelt werden. Gemeint ist das Gebiet der Verkehrswissenschaft. Das hat zunächst mit Computern nur sehr entfernt etwas zu tun. Sondern Verkehrwissenschaft ist ein Gebiet was sich durch eine umfangreiche Literaturiiste auszeichnet und über eine lange Historie verfügt. Das Domänenwissen was man für ein autonomes Auto benötigt wird nicht etwa in einem BDI Agenten formalisiert sondern es liegt im Fachbereich Verkehrswissenschaft vor. Und genau hier kommt die Subsumption Architektur ins Spiel. Es handelt sich um nichts anderes als das Erlernen von Bescheidenheit. Das also die Informatik zugibt, es nicht zu wissen wie man den Roboter steuern soll, und das man sich Hilfe holt von Außerhalb. Also Verkehrsexperten, Unfallforscher, Verkehrsdidakten zu Rate zieht.

Der äußere Rahmen in dem darüber diskutiert wird, wie die Software mit dem Event-Queue einer Kreuzung verfahren soll, wird nicht durch den Computer definiert sondern muss mit menschlichen Experten erörtert werden. Üblicherweise innerhalb der Gutenberg-Galaxis auf Deutsch oder Englisch. Erst dieser Rahmen ist weit genug gefasst um eine Kreuzungssituation umfassend zu beschreiben. Genau genommen muss man also wissenschatliche Literatur zitieren wenn man wissen will, was man an einer Kreuzung tun darf und was nicht.

Es bleibt noch die Frage zu beantworten wie man das gesammelte Wissen der Verkehrsiwssenschaft in ein Computerprogramm unterbringt. Die Antwort lautet, dass man Wissen als Spiel modelliert. Und genau an diesem Punkt wird es interessiert. Ein Spiel haben wir ja bereits, die Autosimulation. Für dieses Spiel wollten wir ontop einen

Bot schreiben, also eine Künstliche Intelligenz. Und jetzt erfahren wir, dass um diesen Bot zu schreiben wir zunächst ein Spiel benötigen um die Domäne zu formalisieren? Häh wie jetzt? Man kann dieses Paradoxon ungefähr so auflösen. Es gibt keinen Unterschied zwischen einem Computerspiel auf der einen Seite und dem Bot auf der anderen Seite. Soetwas wie einen dezidierten Bot der unabhängig vom Spiel agiert gibt es nicht. Hier die Details:

Ein Computerspiel stellt den Versuch da, komplexes Domänenwissen aus einem Gebiet zu formalisieren. Beispielsweise wird die Verkehrswissenschaft in einem Autosimulation erlebbar. Wenn man jetzt plant für dieses Spiel ein Bot zu schreiben und merkt, dass das Domänenwissen nicht vollständig ist kann man nicht damit beginnen den Bot schlauer zu machen. Sondern die Lösung lautet, das Projekt "Schreiben eines Bots" einzustellen und stattdessen die Simulation zu verbessern. Also mehr Domänenwissen darin abzubilden. Wenn die Simulation genug Wissen über Verkehrssicherheit enthält braucht man keinen dezidierten Bot der ein Weltmodell erstellt sondern das Spiel ist das Weltmodell.

Diese Erkenntnis überrascht etwas, weil normalerweise die Idee darin besteht einen Bot ontop eines bestehenden Spiels zu programmieren. Also die Künstliche Intelligenz und das Domänenwissen quasi nachträglich einzubauen. Das funktioniert jedoch nicht. Das einzige was man tun kann, ist ein Computerspiel zu programmieren. Soetwas wie ein Technik um eine AI zu programmieren gibt es nicht. Weil eine AI würde Domainwissen benötigen, und das ist entweder im Spiel vorhanden oder muss erst dort einprogrammiert werden.

11.7 Theorie vs. Praxis

In der Theorie ist das Erstellen eines BDI konformen Agenten simpel. Man liest sich zunächst einmal die Paper durch welche zu Jade und Jadex veröffentlicht wurden, modelliert dann entsprechend der Beispiele seine Domäne, baut dort noch eine Meta-Learning Funktion ein und schon hat man saubere wissenschaftliche Arbeit geleistet. Doch irgendwas stimmt nicht mit diesen Agenten, sie wurden nicht unter Live-Bedingungen getestet. Und genau in diese Lücke stoßen Robotik-Wettbewerbe wie Robocup, Torcs Challange, Mario AI und ähnliche hinein. Wichtigstes Merkmal dieser Challanges ist, dass es egal ist, ob man verstanden hat was BDI bedeutet oder ob man fit ist in LISP sondern worum es geht ist die Punktzahl. Das schöne ist, dass man den Sieger ermitteln kann, selbst wenn man keine Ahnung von Künstlicher Intelligenz hat. Bei Torcs dadurch dass man die Rundenzeit misst. Welche Programmiertechnik der Sieger verwendet ist egal.

Schaut man sich einmal die real durchgeführten Wettbewerbe an und vergleicht die Ergebnisse mit dem was in den Agenten-orientierten theorie Papern drinsteht so könnte der Unterschied größer nicht sein. Womöglich ist BDI kompletter Unsinn, es traut sich aber keiner zuzugeben weil das ja heißt, man hat das Konzept nicht verstanden, und vielleicht ist auch SOAR und PDDL Unsinn aber auch das traut sich keiner zu sagen. Was man jedoch sagen kann ist wer auf den Roboter-Challanges gewinnt. Dort steht der Bewertungsmaßstab fest, es gibt Regeln die für alle gelten.

Aber woher kommt das starke Gegensatz zwischen Theorie und Praxis? Nun die BDI Architektur wurde entwickelt um auf wissenschaftlicher Ebene zu klären was Künstliche Intelligenz ist. Sie baut auf gängigen Theorien auf und entwickelt diese weiter. Genauer gesagt fügt sich BDI ziemlich gut ein, in vorherige kognitive Architekturen wie ACT-R und LISP. Die Schwäche der BDI Architektur ist, dass sie zu einer Projektionsfläche für alles mögliche geworden ist. Vielleicht ist der Ansatz genial, vielleicht ist er aber auch eine Sackgasse, wer könnte das schon sagen?

12 Semantic video surveillance

12.1 Motivation

In der Einleitung zu diesem Paper ist zu lesen, dass die Zielstellung darin besteht ein Head-Up Display zu programmieren. Und zwar deshalb, weil sich darüber die Künstliche Intelligenz entwickeln lässt. Diesen Ansatz kann man noch weiter präzsieren. Es geht darum ein Head-up Display zur semantischen Videoanalyse zu programmieren.

Machen wir es etwas konkreter. Ausgangspunkt ist ein beliebiges Computerspiel wie Super Mario Bros. Und die Aufgabe lautet dafür eine Künstliche Intelligenz zu entwickeln. Die Programmiersprache ist klar: C++ es fehlt nur noch ein Plan wie man die KI realisiert. Verabschieden sollte man sich von der Idee from scratch eine autonome KI zu erzeugen, wo man also auf den Knopf drückt und das System übernimmt dann die Steuerung. Besser ist es, zunächst ein Player-Support System zu realisieren. Konkret besteht das aus dem eingangs erwähnten Head-up Display, wo also der Spieler Zusatzinformationen eingeblendet bekommt. Offen war bisher noch, was das für Informationen sein sollen, also wie das Domain-Modell für Super Mario Bros aussieht. Und genau dafür benötigt man eine Ingame-Video surveillance. Die Gameengine erzeugt die Events, diese werden zu semantischen Informationen umgewandelt. Und genau für diesen Teil ist das Surveillance System zuständig. Es hat die Aufgabe, aus Pixeldaten semantische Informationen zu erzeugen. Aus Lowlevel Daten wie "x=120, 123, 126" erzeugt die Videoüberwachung das Event "Mario läuft auf den Abgrund zu".

Üblicherweise werden Video Surveillance Systeme für echte Kameras entwickelt. Wo man auf einem öffentlichen Platz eine Kamera hinstellt, das Signal dann mit OpenCV auswertet und daraus dann die semantischen Informationen haben möchte. Man kann solche Systeme aber auch wunderbar für die ingame Überwachung einsetzen. Den Teil mit OpenCV kann man sich sparen, weil ja die Pixeldaten bereits in der maximalen Qualität vorliegen und die Gameengine weiß auf welcher (x,y) Position Mario gerade ist. Was die Game-Engine jedoch nicht weiß ist was es bedeutet. Also jenes Aufgabengebiet was in den wissenschaftlichen Paper als spatiotemporal-Reasoning bezeichnet wird. Das ist nichts anderes als eine semantic Video surveillance Parser. Links gehen die Pixeldaten rein und rechts kommt die Ontologie heraus.

Halten wir fest: eine Mischung aus Videosurveillance plus Head-up Display macht gar nichts. Der Spieler muss Mario nach wie vor manuell steuern. Sondern das System läuft im Hintergrund mit, analysiert die Szene und blendet Kontextinformationen ins Head-up Display ein.

In der Literatur wird das Thema für echte Video-Surveillance relativ ausführlich behandelt, inkl. Protege Ontologie die das Klassendiagramm zeigt in das man die Events speichert. Seltener sind hingegen Paper die sich speziell mit Spielen wie Fußball und noch seltener mit Ingame-Videosurveilance beschäftigen. Für die Echtzeit-Analyse von Fußballspielen ist es noch relativ naheliegend eine semantische Videoanalyse einzusetzen: man kann dort schön den Ball tracken und die Spieler zuordnen. Leider sind solche Systeme extrem aufwendig, viel interessanter hingegen ist es die Videoauswertung von Computerspielen. Weil man dort nicht erst mehrere Kameras aufbauen muss sondern die Daten ohnehin vorliegen. In so einem Setting kann man sich komplett auf den Semantic Parser konzentrieren, also jenen Programmteil der Pixeldaten in Sinn-Zusammenhänge konvertiert.

Vielleicht noch ein weiteres Beispiel. Video-Surveillance Software um einen Parkplatz zu überwachen kann unterscheiden ob ein Auto gerade fährt oder ob es steht. Das interessante ist, dass man so ein System auch ingame nutzen kann. Zwar weiß dort die Game-Engine bereits, wo das Auto gerade ist, aber das heißt nicht dass

Video-Surveillance überflüssig wäre. Das Problem ist, dass zwar die aktuelle Position irgendwo in einer Klasse gespeichert ist, aber die Software noch nicht dezidiert weiß, dass ein Auto stillsteht, wenn die Position konstant bleibt. Ein Mensch sieht das sofort. Er schaut sich das Computerspiel an, und sieht das ein Auto sich nicht bewegt. Für die Game-Engine hingegen fehlt diese Information.

12.2 High-level Events

Eine genauere Literaturrecherche hat ergeben, dass "video surveillance ontology" optimal geeignet sind, wenn man für ein vorhandenes Computerspiel ein Domänenmodell erstellen will. Wenn man in der Lage ist, ein Video semantisch zu parsen hat man auch ein maschinenlesbares Modell. Leider besitzt die aktuelle Literatur zwei Nachteile. Erstens, fokussiert sie allein auf Videosurveillance ist also an der Interpretation von Echt-Videos interessiert. Beispielsweise um den laufenden Verkehr zu überwachen oder um ein Kamerasystem in einer U-Bahn zu installieren. Viel spannender ist hingegen videoüberwachung von Computerspielen, wo man also zuerst das Spiel erzeugt und es dann interpretiert. Das zweite Problem mit der aktuellen Literatur ist, dass der Begriff Ontology zu inflationier verwendet wurde. Eigentlich ist eine Ontologie nichts anderes als ein UML Klassendiagramm, wo also mehrere C++ Klassen interagieren. Das auf reine OWL Ontologie zu reduzieren wird der Sache nicht gerecht.

Trotz dieser kleineren Hindernisse ist jedoch das Thema als solches ausgezeichnet geeignet um eine Künstliche Inteligenz für Computerspiele zu entwickeln. Insbesondere sogenannte High-Level-Eventdetection ist in der Lage aus lowlevel Ereignisse höherwertige Rückschlüsse zu erzeugen. Wenn ein Standard-Computerspiel wie Super Mario Bros gestartet wird passiert normalerweise folgendes: es werden eine Abfolge von Lowlevel Daten erzeugt. Genauer gesagt wird mit 25 Bildern pro Sekunde Pixeldaten auf den Monitor gezeichnet. Oder noch genauer, auf dem Bildschirm sind Objekte mit x/y Koordinaten zu sehen, die sich bewegen.

Die Kunst besteht darin, diese Zahlenwerte semantisch aufzubereiten. Und zwar anhand des Domänenmodells. Bei Super Mario Bros und bei Starcraft AI sind die Events unterschiedlich. In dem einen Spiel gibt es ein Event wie "hochspringen" in dem anderen Spiel heißt das Event "Einheiten bewegen". Auf Lowlevel Ebene sind es ähnliche Ereignisse. Ein Objekt auf dem Bildschirm verändert seine Position im x/y Raum. Unterschiedlich ist jedoch die semantische Bedeutung. Die muss man aber kennen wenn man eine Künstliche Intelligenz realisieren will.

Nehmen wir mal an, man hat einen solchen Event Parser bereits realisiert. Ist also in der Lage entsprechend der Domäne die Szene zu interpretieren. Das bedeutet nichts geringeres als dass das Computerspiel maschinenlesbar wird. Bei Unity3d heißt diese Funktion Collider und meint dass man in einer Loop mehrmals pro Sekunde das Spiel untersucht ob etwas wichtiges passiert. Ein High-Level-Collider würde bei einer Trafficsimulation nach folgenden Dingen suchen:

- Stau
- Verkehrsunfall

Und wenn soetwas eintritt wird eine Variable von False auf True gesetzt. Im Grunde ist das die eigentliche KI. Der Programmierer muss nur dafür sorgen, dass die Events korrekt geparst werden. Üblicherweise enthalten Computerspiele noch keine High-Level-Event Detektion. Zwar verfügen alle Spiele über eine Kollisionserkennung inkl. Hochzählen des Punktestandes wenn ein gewünschtes Ereignis eintritt, aber das ist nur eine sehr grobe Event-Detektion. Wenn man das Spiel von einer KI spielen lassen will, benötigt man sehr viel detailliertere Angaben über das Spielgeschehen, man muss also seinen eigenen Event-Parser programmieren.

```
Autorennen
- Strecke
  - Kurven
  - Pit stop
  - Regen
- Auto
  - Drifting
  - Gangschaltung
- Überholen
  - Kollision
  - Unfall
```

Abbildung 4: Linguistic Ontologie für Autorennen

12.3 Domain Knowledge

Eine Videoüberwachung muss domain-spezifisch sein. Damit ist gemeint, dass es einen Unterschied macht, ob man ein Autorennspiel oder ein Jump'n'Run Spiel automatisch analysiert. Die Domäne selber wird außerhalb der Informatik definiert. Im Bereich Autorennen durch die Wissenschaft vom Fahren und im Bereich Jump'n'Run durch die Wissenschaft vom Geschichten erzählen. Der Kanon einer Domäne definiert sich durch die Lehrbücher, Glossare und Studiengänge die es zu einem Thema bereits gibt. Eine halbwegs formale Beschreibung findet sich in sogenannten Walkthroughs die es für jedes Computerspiel gibt. Ein Walkthrough ist eine textuelle Anleitung wie man ein Spiel löst welche mit Screenshot und ergänzenden Markierungen versehen ist.

Die Aufgabe der Domain-Modellierung besteht darin, ein vorhandenes Walkthrough in ein Video-Surveillance-System zu überführen. Das Videosurveillance System selber ist wie jedes andere Computerprogramm auch in einer Hochsprache wie C++ formuliert. Manchmal wird in der Literatur nach weiteren Beschreibungslogiken gesucht, wie der "Temporal Description Logic" oder Bayschen Netzen doch all diese Verfahren lassen sich innerhalb eines C++ Programms ausdrücken. Man kann also sagen, dass fertig implementiertes Domain-Knowledge ein C++ UML Diagramm mit ausformuliertem Sourcecode ist.

Die Erstellung eines solchen kann man durch die Verwendung einer "Linguistic Ontology" vereinfachen. Damit ist eine Mindmap gemeint, welche die wesentlichen Begriffe einer Domäne hierarchisch auflistet. In der Abbildung habe ich eine solche Ontology für die Domäne Autorennen dargestellt. Es handelt sich dabei um eine stark vereinfachte Beschreibung die angelehnt ist an einen Outline-Editor.

In älteren wissenschaftlichen Veröffentlichungen ist zu lesen, dass man die Knowledge Base als Teil eines Expertensystems betrachten müsste. Folglich benötigt man ein Expertshell und eine spezielle Programmiersprache wie Prolog. Diese Sichtweise ist nicht mehr zeitgemäß. Stattdessen lässt sich die Konstruktion einer Knowledgebase als Software-Engineering-Prozess verstehen. Er wird mit den selben Methoden durchgeführt, als wenn man ein Betriebssystem programmiert, also mit Hilfe eines Bugtrackers, Versioncontrol-Systeme, OpenSource Repositorien und natürlich auf Basis der C++ Programmiersprache. Knowledge-Engineering unterscheidet sich in keinster Weise von Anwendungsprogrammierung, man benötigt auch keine dezidierten OWL Ontologien die mit Protege erstellt werden, sondern eine normale Programmier-IDE plus C++ Classes sind mehr als ausreichend. Selbst Scripting Sprachen wie Lua oder Golog sind überflüssig, man kann das komplette Wissen als C++ Sourcecode formulieren.

Leider ist der Aufwand dafür hoch bis sehr hoch. Das ist auch der Grund, warum es nur wenige funktionierende Video Surveillance Systeme gibt. Insbesondere im Bereich ingame-Surveillance besteht ein akuter Mangel.

VERL [35] verwendet zur Beschreibung einer Domäne nicht C++ sondern greift auf speziell für diesen Zweck konstruierte Sprachen

zurück. Auf Seite 2 wird für das Beispiel einer Zutrittskontrolle einer Anlage einmal die Sprache “Video Event Representation Language (VERL)” verwendet und zwei Absätze später das gleiche nochmal in der Sprache “Semantic Web Rule Language (SWRL)” (einem auf OWL aufbauenden Dialekt) formuliert. In beiden Fällen wird eine Art von Logic-Chaining betrieben, das heißt, es werden Bedingungen überprüft und die Funktion gibt dann entweder true oder false zurück. Leider teile die Euphorie des Autors über diese Spezialsprachen nicht, würde man das Problem in normalem C++ formulieren, wäre der Sourcecode nicht viel länger. Ein Vorteil durch die Verwendung von diesen Event-Sprachen sehe ich nicht.

Der Grund warum solche Sprachen verwendet werden, hat weniger technische Gründe sondern ist Ausdruck einer bestimmten Programmierkultur. Die Idee ist es, ähnlich wie bei einer Firewall auch, Programmlogic von der Anwendung zu trennen und in einer high-Level-Sprache die Rules zu definieren. In manchen Computerspielen wird ebenfalls der Behavior Tree nicht in C++ sondern in einer eigenen Scripting-Sprache wie Lua formuliert, ebenfalls mit der Idee darüber die Programmierung zu vereinfachen und/oder möglicherweise sogar die Anwendungslogik automatisiert zu erzeugen. Nur, die Erfahrung zeigt, dass Scripting Sprachen keinen Vorteil gegenüber C++ aufweisen, man kann zwar potentiell schneller die Anwendung fertigstellen hat aber anschließend das Problem, dass man dann zwei unterschiedliche Sprachen in dem Projekt hat.

Was man eigentlich bräuchte, wäre eine High-Level-Sprache die aber nicht zugleich eine Programiersprache ist, wie z.B. HTML oder die Wikisyntax. Wenn man darin die Anwendungslogik formulieren könnte, wäre das gegenüber C++ tatsächlich eine Verbesserung. Aber, ich habe den Verdacht dass genau solche Sprachen nicht mächtig genug sind um spatio-temporale Events zu beschreiben. Am Ende braucht man doch wieder klassischen Unterfunktionen, if-then-Bedingungen und Sprünge in andere Programmteile.

Beispiel Schauen wir uns eine konkrete Domäne an: Super Mario Bros. Einen Bot der automatisch durchs Level hüpft ist auf github verfügbar. [14] Er wurde in der Sprache Lua geschrieben und besteht aus 853 Lines of Code. Genausogut hätte man auch die oben erwähnte Video Event Representation Language (VERL) verwenden können. Die Syntax wäre vergleichbar gewesen. In beiden Fällen wird die Domäne Super Mario Bros in einer Computersprache beschrieben. Wenn man sich den konkreten Lua Sourcecode einmal anschaut, so ist dieser keineswegs deklarativ oder wiederverwendbar sondern es ist ein Yet-another-Programming Projekt. Das heißt, jemand mit zuviel Zeit hat sich in sein Zimmer eingesperrt, mit der IDE den Lua Code reingehackt und solange verbessert bis es funktioniert. Und genau das ist das Problem. Anders als in den theoretischen Papern angedacht ist es eben nicht möglich, eine Domäne abstrakt zu beschreiben und ohne dezidiertes Programmieren sondern es läuft immer darauf hinaus, dass man eine Hochsprache nutzt, darin Funktionen definiert und die dann testet. Es ist also immer ein Software-Engineering-Projekt von dem bekannt ist, wie lange es ungefähr dauert. Das obige Lua Programm besteht aus 850 Zeilen. Bei einer unterstellten Produktivität von 10 Zeilen Code am Tag hat der Programmierer also 85 Tage damit zugebracht es zu schreiben. Würde man eine andere Sprache verwenden wie LISP, VERL oder gerne auch C++ wäre die Länge des Sourcecodes identisch und der Zeitaufwand vergleichbar.

Worauf ich hinausmöchte ist, dass Domänemodellierung immer bedeutet, dass man ausführbaren Sourcecode erstellt. Dafür gibt es zwar geschätzt 100 unterschiedliche Programmiersprachen, man kann aber auch einfach C++ nehmen und erzielt damit die selbe Produktivität. Die Leistung des Programs definiert sich über die Codezeilen und je mehr man schreiben will, desto aufwendiger wird es.

[14] https://github.com/haseeb-heaven/LuaRio_Bot/blob/master/LuaRio_Bot.lua

12.4 Domain-Modelling mit Versioncontrol

In der Literatur auf Google Scholar sind die meisten Autoren der Überzeugung, dass man Domain-Modelling mit Hilfe von Petrinetzen, neuronalen Netzen und OWL realisieren könnte. Sie erfinden Semantic Event Description Language um darin Domain-Knowledge auszudrücken. Dieser Ansatz ist eine Sackgasse. Die oben erwähnten Konzepte sind nicht mächtig genug. Was hingegen mächtig ist, sind zwei gegensätzliche Formen von Domain-Knowledge: einmal die textuelle Beschreibung in Form eines Walkthroughs und auf der anderen Seite eine formalisierte maschinenlesbare Variante als C++ Sourcecode. Beide Darstellungsformen sind in der Lage Domain-Knowledge umfassend zu beschreiben. Um von Darstellungsform A zu Darstellungsform B zu gelangen gibt es nur eine Option: Versioncontrol-Systeme. Gemeint sind git und Co. Also ein System was das Schreiben von Sourcecode unterstützt. In Git lassen sich Branches anlegen, Änderungen rückgängig machen und ganz wichtig es wird die Arbeit im Team unterstützt. Das Erzeugen einer Domain-Knowledgebase lässt sich am besten als git logfile abbilden. Wo also unterschiedliche Leute commits an das Coderepository senden und dadurch den C++ Sourcecode immer weiter verbessern.

Viele halten diesen Prozess für langatmig aber es ist die derzeit einzige Möglichkeit dies es gibt. Es bedeutet, dass man die Knowledge-Base per Hand programmiert. So ähnlich als wenn man ein Computerspiel programmiert. Dieser Vergleich ist ziemlich gut geeignet um auszudrücken worum es geht. Auch beim Programmieren eines Computerspiels wird Domain-Knowledge in Computercode übersetzt. Nur dass es bei Computerspielen ausreicht, wenn am Ende die Grafik ruckelfrei läuft, wenn man hingegen eine semantische Ontologie formalisiert, muss man sehr viel mehr Aspekte der Domäne in C++ Code implementieren. Die Abläufe bleiben jedoch dieselbe: in beiden Fällen benötigt man Programmierer, setzt einen github Server auf und erstellt commits.

12.5 Action to Language

In Videospielen werden normalerweise Joysticks verwendet um Einfluss zu nehmen. Bewegt man den Joystick nach oben bewegt sich auch das Auto nach oben. Für eine manuelle Kontrolle ist diese Interaktion ausreichend, die Aufgabe des Spieleprogrammierers ist erfüllt. Wenn man jedoch vorhat, das Spiel von einer KI spielen zu lassen benötigt man eine Zwischenebene, genauer gesagt ein Taskmodell. Ein Beispiel: das Auto fährt eine Straße entlang, der Befehl lautet “links abbiegen”. Wenn das Auto auf dem Bild nach oben fährt, heißt links tatsächlich links, wenn es jedoch von oben nach unten fährt, meint links: links aus Sicht des Autos. Man drückt also den Joystick nach links aber auf dem Monitor fährt das Auto nach rechts.

Derartige Abstraktionsstufen gibt es für andere Bereiche ebenfalls. Um sie zu ordnen benötigt man natürliche Sprache. Anstatt einen Joystick-Befehl an das Spiel zu senden, sendet man ein natürlichsprachliches Kommandos, das wird von dem Taskplanner in einen Lowlevel Befehl übersetzt. Das schöne ist, dass man Domainwissen grundsätzlich mit natürlicher Sprache beschreiben kann. Egal ob man die Interaktion beim Fußballspiel oder die Aktionen von Autos beschreiben möchte, in beiden Fällen ist natürliche Sprache das beste Mittel. Jetzt gibt es nur ein Problem: Computer können lediglich Programmiersprachen interpretieren nicht jedoch natürliche Sprache. Das wichtigste Element einer KI besteht darin, den fehlenden Layer zu implementieren, also einen Action to language Parser und einen lan-

guage to action Generator zu programmieren. Solche Konverter werden in der Literatur üblicherweise als Grammatik beschrieben (Stichwort Compilergenerator). Besser ist gedoch, wenn man das Taskmodel objektorientiert formuliert, also mit Hilfe eines UML Diagramms die Szeneninterpreation durchführt. Grammar-Basierende Ansätze sind nur ausreichend, wenn man C++ nach Assembly Language transformieren möchte, um jedoch eine Verkehrskreuzung in Sprache zu übersetzen sind sie nicht mächtig genug.

Die möglichen Aktionen bewegen sich in einem spatio-temporalen Kontext. Damit ist gemeint, dass es um einen Zeitpunkt geht, an dem ein Objekt eine Position besitzt. Auto1 fährt beispielsweise eine Straße entlang, und gleichzeitig kommt ein anderes Auto aus einer Seitenstraße gefahren. Auf der Lowlevel Ebene werden diese Abläufe durch die Grafikkarte angezeigt, also auf 800x600 Pixeln wo bestimmte Pixel sich farblich verändern. Auf High-Level-Ebene hingegen erfolgt die Beschreibung als natürlichsprachliches Logfile, also so wie eben beschrieben:

Frame 100: Auto1 fährt die Straße entlang

Frame 100 (gleichzeitig): Auto2 kommt die Seitenstraße entlang

Domain Modelling warum? Angenommen man betreibt viel Aufwand um einen Action-to-language Parser zu erstellen. Mit diesem kann man ein Computerspiel kommentieren. Aber was will man mit so einem System anfangen? Denn eigentlich will man ja keinen Märchenerzähler haben sondern eigentlich soll die KI das Spiel selber spielen. Aber der Zeitaufwand zur Domain-Modellierung ist keineswegs vergeblich. Ganz im Gegenteil. Schauen wir uns einmal an, was es bedeutet in einer Traffic-Simuation eine Handlung bzw. eine Aktion auszuführen. Eigentlich gibt es nur 4 mögliche Aktionen: links, rechts, beschleunigen und bremsen. Die möglichen Aktionen sind also sehr überschaubar, anders gesagt ein Auto zu steuern ist das geringste Problem. Man muss da nichts planen oder berechnen, sondern man sendet einfach einen Befehl wie “action(left)” und das Auto reagiert. Der Clou ist, dass man in eine Aktion auch nicht mehr Aufwand investieren möchte wenn man es möchte. Sondern die Intelligent einer Entscheidung definiert sich darüber was vor der eigentlichen Aktion kam, und das war die Spielanalyse.

Man kann sagen, dass der Programmieraufwand für den Action-to-language Parser bei 95% liegt und das eigentliche Aktion Modul was eine Entscheidung ausführt nur 5% des Programmieraufwands einnimmt. Ist die Szenenanalyse korrekt ist also der Parser in der Lage das Geschehen zu kommentieren kann man daraus sehr leicht die erforderliche nächste Aktion bestimmen. Vielleicht ein Beispiel:

Angenommen es ist klar, dass sich das Auto an einer Kreuzung befindet, es links abbiegen will, der Blinker gesetzt ist, und der Gegenverkehr frei ist. All diese Meldungen würde die Event-Recognition ausgeben, in wohl formuliertem Englisch versteht sich. Dann ist handlungstechnisch die Sache simpel: Die Straße ist frei, es ist klar was das Auto vorhat, dann kann man nach links abbiegen. Die Aktion welche der Spieler ausführt ist nur das letzte Element einer langen Ketten von Prozessen. Das Blabla hingegen was davor war, also worüber das Spielgeschehen beschrieben wurde, das ist die eigentliche KI. Und darüber wird eine Aktion auch vorbereitet, ja entschieden.

Auch bei anderen Spielen gilt, dass die eigentliche Aktion lächerlich simpel ist. Bei Super Mario Bros hat der Spieler nur die Möglichkeit nach links oder rechts das Steuerkreuz zu drücken und mit einer dritten Taste zu springen. Mehr Optionen bestehen nicht. Bei Pong ist die Sache noch übersichtlicher, dort kann man nur up oder down sagen. Und bei einem Flipperautomaten gibt es sogar nur eine Entscheidungsmöglichkeit: entweder kickt man gegen das Paddel oder man macht gar nichts. Davon unbenommen ist jedoch die Spielanalyse, also der Action-to-language Parser extrem kompliziert. Es gibt in den Spielen unendliche viele Feinheiten zu beachten.

13 Sonstiges

13.1 Versionsverwaltung als Meta-Struktur

Um die Programmierung zu vereinfachen verwendet die Informatik Meta-Modelle. Genauer gesagt forscht sie, ob es solche Modelle gibt. Eines von diesen Meta-Modellen ist UML, eine Syntax in der sich Klassen beschreiben lassen. Nur leider reicht UML nicht aus, um eine Domäne zu spezifizieren, weil eben nicht klar ist, welche Klassen überhaupt benötigt werden und wie sie interagieren sollen. Oberhalb von UML kann man Versionsverwaltungssysteme ansiedeln. Es handelt sich dabei um Computerprogramme welche Änderungen tracken. Also den Fortschritt beim Programmieren erfassen. Die Frage ist jetzt, ob git und Co bereits das Ende der Fahnenstange darstelle oder ob es nicht noch weitere Meta-Modelle gibt. Vorstellbar wäre eine domänenspezifische Versionsverwaltung, die also sehr viel stärker auf das konkrete Probleme hin zugeschnitten ist. Also ein Template zur Erstellung einer Versionsverwaltung.

In [14] wird für eines Autopiloten eine Mischung aus Domain-Specific-Language, Versionsverwaltung und objektorientierter Programmierung verwendet. Gegenüber einem Standardworkflow wie er in der Industrie eingesetzt wird, ist eigentlich nur die Domain Specific Language neu. Das man also nicht nur mittels git Java-Code trackt, sondern git einsetzt um einen Domain-Specific Language (DSL) zu überwachen:

> “A DSL is a metamodel, which not only captures the domain concepts, but also the domain-specific constraints and rules” [14, page 32]

Das Problem was ich persönlich mit dem Ansatz sehe ist, dass auch normaler Java-Code domain-specific ist. Wenn man in einer Klasse eine Methode definiert wie “opendoor”, dann ist diese auf das konkrete Problem hin zugeschnitten. Wozu braucht man noch eine explizite DSL? Eine mögliche Antwort könnte sein, dass man zu einer DSL einen "abstract syntax tree" erzeugen kann. Das geht bei klassischer Java Programmierung nicht, dort wird zwar auch ein "abstract syntax tree" verwendet, aber nur für die Java Sprache an sich (Sourcecode -> JVM) nicht jedoch für die Methoden und Klassen in der Sprache selber.

Ein gutes Beispiel für eine DSL ist die Game-Description Language (GDL), eine Programmiersprache um Spiele zu programmieren. Aber, bisher fehlt der Nachweis, dass man damit effizienter ein Spiel programmiert. Gegeneinüber dem klassischen Ansatz eine vorhandene Game-Engine zu verwenden und darin sein Spiel zu programmieren gibt es praktisch keinen Unterschied. Die Schwierigkeit besteht darin, dass das was der Programmierer tut, sich auch mit einer domänenspezifischen Sprache nicht abbilden lässt sondern sich der Geschichte der Computerspiele orientiert. Also an seinem Verständnis für 2D Roleplaying Games, an den Pattern um eine Gameloop zu kontruieren oder an Vorbildern an denen er sich orientiert. Diese höhere Abstraktionsebene die beim Programmieren verwendet wird ist nicht formalisierbar sondern entspricht dem was die Geisteswissenschaft als Diskursraum kennt. Also ein Kanon an vorhandener Literatur, Memen, wichtigen Autoren in dessen Fahrwasser man sich bewegt. Wenn überhaupt lässt sich dieser Kanon als Querry an die Google Suchmaschine definieren. Das heißt, die Anfrage lautet “Alle Informationen über computerspiele von 1980 bis heute” und das ist dann die Basis um neue eigene Ideen zu realisieren. Leider sind Domänenspezifische Sprache nicht im Stande mit Google zu konkurrieren. Selbst wenn man sich eine 10 kb große Grammar ausdenkt und diese ordnungsgemäß parst reicht deren Informationsgehalt nicht aus.

Odyssey-VCS Unter der Maßgabe, dass eine domänenspezifiische Sprache mit UML Profilen sehr viel leichter modellliert werden kann als das man die DSL als Gegenmodell zu UML versteht, bleibt UML als das einzige Modellierungstool übrig. Also muss man das Erstellen eines UML Modell lediglich mit Hilfe von Versionsverwaltungen tracken. Eine Software dafür ist Odyssey-VCS. Leider ist kein Vorteil gegenüber einer simplen git-Installation erkennbar. Auch Odyssey-VCS kann nur Änderungen tracken und womöglich eine Issue-ID speichern.

Meiner Meinung nach ist die nächst höhere Stufe nach UML ein Online-Kollobarationstool wie github. Wo man also den Prozess dadurch aufwertet, dass man eine Community mit in die Softwareerstellung einbezieht. Dadurch erhält man oberhalb von UML noch eine weitere Ebene. Das jedoch ist nichts neues, sondern ist weit akzeptierter Standard in Sachen Softwareentwicklung.

13.2 Subsumption architecture mit Memory

Die Subsumption architecture von Rodney brooks geht davon aus, dass die Umwelt eines Roboters zu komplex ist um sie zu modellieren. Selbst wenn man Domain-Specific Languages, UML Metamodelle und Dictionarys einsetzt wird es nicht gelingen ein vollständiges Umgebungsmodell zu erzeugen. Der Ausweg den Brooks anbietet ist sich von unten nach oben vorzuarbeiten. Also mit Lowlevel Primiven zu beginnen und diese dann schrittweise zu komplexeren Methoden zusammenzubauen. Ein Lowlevel Primitive wäre eine inverse Kinematik. Diese besteht aus 3 Zeilen Java-Code welcher eine methamtische Formel beinhaltet. Mit einer solchen Prozedur ausgestattet ist der Roboter zwar noch nicht intelligent aber er kann zumindest einen Teil der Aufgabe erfüllen.

Leider hat der Ansatz von Brooks ein Nachteil: er unterstellt, dass die Subsumption architecture 1:1 auf die Servomotoren gemappt wird. Das heißt, eine Prozedur wird ausgeführt und hat zur Folge dass sofort der Roboter etwas macht. Das ist gerade bei komplexen Systemen nicht sinnvoll. .Besser ist es, einen zwischenlayer, genannt Head-up Display einzufügen. Diesen kann man als Memory des Roboters betrachten. Es bedeutet, dass man zwar eine Subsumption Architektur verwendet, aber diese nichts tut. Das heißt, der Roboter bewegt sich keinen Zentimeter obwohl die Software arbeitet. Stattdessen werden lediglich Bilddaten als Overlay Image erzeugt.

Anstatt also Behavior zu spezifizieren die wie der Name andeutet, Aktionen implizieren werden Memory-Chunks programmiert. Also Teile eines Head-up Displays die zur Wahrnehmung der Umwelt verwendet werden. HIer gibt es ein kleines Problem. Ein Roboter ist laut Definition für das Ausführen von Aktionen da. Wenn der Roobter stillsteht und gar nichts macht, hat man auch keinen Roboter. Sondern man hat einen Sensor.

Dennoch ist der Ansatz von Brooks gar nicht mal so verkehrt. Das Prinzip um das es geht lautet Bottom Up, also ausgehend von kleinen Motion Primitive komplexere Systeme zu programmieren. Die mögliche Ergänzung besteht darin, dass nicht ein Roboter das Ziel ist, sondern ein Parser. Dieser überwacht einen Menschen beim Ausführen von Aktionen. Der Human-Operator führt Aktionen aus, der Parser trackt diese und visualisiert es auf dem Head-up Display in Echtzeit. So die Idee. Wie man das konkret umsetzt ist derzeit noch unklar. Den Ansatz dafür neuronale Netze zu vewenden halte ich für falsch. Sondern ähnlich wie die Subsumption Architektur von Brooks muss man das System manuell programmieren, also in C, LISP oder womit auch immer.

In [28] wird ein Beispiel erläutert wie man einen Walk-Zyklus von einem Menschen parsen kann. Es wird dazu eine Domain-Specific-Language plus Grammar verwendet. Vermutlich in der Annahme, damit ein universelles Ausdrucksmittel zu besitzen. Aber, man sollte bedenken, dass grammar-based Language ursprünglich zum Parsen von Computersprachen entwickelt wurden und nicht besonders mächtig sind. Besser wäre es einen objektorientierten Parser zu verwenden. Also ein UML Modell was in der Lage ist, Sensordaten zu interpetieren. Aber das ist nur die konkrete Methode. Im Kern geht es um den Parser als solchen. Also um eine Software welche eine Ist-Sitatuation auswertet. Ein wenig umgangssprachlich formuliert geht es darum, eine Head-up Display für eine Domäne zu programmieren.

Das bedeutet folgendes: angenommen ein Mensch nimmt eine Banane auf. Dann wird dies selbstverständlich auf dem Display angezeigt und wenn sich das ein Mensch ansieht sieht er es auch. Aber, der Vorgang ist nicht maschinenlesbar. Das wird er erst wenn explzit auf dem Bildschirm eingeblendet wird: Subjekt=Mensch, Action=Task, Object=Banana. Das muss nicht textuell erfolgen, sondern andere Visualisierungsmethoden sind ebenfalls denkbar. Aber ohne ein solches Head-up Display hat der Parser die Situation auch nicht erkannt.

Ein weiteres konkretes Beispiel ist das Poeticon Projekt, was laut Selbstdefinition eine minimalist action Grammar und ein semantic memory darstellt.

13.3 Scene Parsing ohne Grammar

Relativ dominant bei Google Scholar sind Papers die sich mit “Scene Parsing with grammar” beschäftigen. Die Idee ist es, ein Pixelbild in eine semantische Beschreibung überführen und zwar mit Hilfe einer kontextfreien Grammatik wie sie normalerweise zum Parsen von Computersprachen wie LISP und Java verwendet wird. Auf den ersten Blick ist das Verfahren vielversprechend. Die Grammar enthält die domänenspezifische Sprache um das Bild zu verarbeiten und gibt in Echtzeit aus was zu sehen ist. Leider hat das Verfahren ein Nachteil, es ist nicht erweiterbar. Jedenfalls nicht viel. Um den Flaschenhals näher zu bewerten ist es wichtig sich an der Maßzahl Lines of Code zu orientieren. Damit ist gemeint, dass ein Parser der aus 100 Lines of Code besteht schlechte Ergebnisse liefert und einer der aus 1000 Lines of Code besteht gute Ergebnisse.

Kontextfreien Grammatiken und domänenspezifischen Sprachen sind jedoch auf Minimalismus hin ausgelegt. Sie sind dann besonders gut, wenn sich das Problem eingrenzen lässt und die zu modellierende Sprache nicht groß verändert. Will man damit BASIC nach Assembler übersetzen ist das die optimale Wahl. BASIC besteht aus vielleicht 30 Wörtern und Opcodes einer 8-Bit CPU sind so umfangreich auch nicht. Bei Scene Understanding ist jedoch die Domäne ungleich komplexer. Hier braucht man Parser die sehr viel mehr leisten. Kurzum man braucht nicht nur einen rekursiven Stack der ein wenig in den Pixeln herumsucht, sondern was man braucht ist ein komplettes UML Modell. Also wo mehrere Klassen sich gegenseitig aufrufen wo eine mehrstufige Scene Understandig Hierarchie ausgeführt wird.

Mein Vorschlag lautet, auf die Grammar zu verzichten und stattdessen die Software schön mit objektorientierten Elementen zu programmieren. Nur so ist sichergestellt dass man sie nach oben skalieren kann, also viele Programmzeilen anfügen kann und die Komplexität weiter erhöht. Das was aktuell in den Papern und als Youtube Video zu sehen ist mit einer Grammar die Bilder parst ist nur der Anfang. Es ist ein minimalistischer Scene Understanding Prototyp der strukturelle Schwächen besitzt.

Die Begriffe Parser, Interpreter oder Transducer sind bereits belegt und zwar meint man üblicherweise Grammar based parsing damit. Also so wie heutige Compiler arbeiten wenn sie Computersprachen übersetzen. Das ist wie oben erwähnt jedoch nicht die optimale Lösung. Besser ist es, etwas allgemeiner von Scene Understanding zu sprechen und damit die Umsetzung bewusst offen zulassen, also auch non-grammar-based Verfahren einzubeziehen.

Ein wenig mehr gehen Paper die sich mit “ontology based scene description” auseinandersetzen. Eine Ontologie ist zwar noch kein UML Diagramm und erst recht kein 2000 Zeilen Java Programm mit ausformulierten Klassen, aber es ist zumindest ein Anfang. Eine Ontologie meint im Regelfall, dass da Objekte sind, die mehrere Methoden enthalten, was impliziert, dass der fertige Parser oh pardon, die fertige Scene Understanding Software ruhig etwas komplexer werden darf.

Ein Paper was in diese Richtung geht und was ich hier lobend erwähnen möchte ist [29]. Auf Seite 4 findet sich eine “Spatio-Temporal Visual Ontology” welche in einem OWL Designer unter Windows erstellt wurde, und aus der man sehr gut erst ein UML Diagramm und später dann Java Klassen machen kann. In dieser Ontologie werden bestimmte Eigenschaften der erkannten Objekte abgelegt. Leider sind noch keine Methoden enthalten, aber man kann sie erweitern. Solche Ontologien, UML Diagramme bzw. Java-Klassen eignen sich ausgezeichnet um komplexere Scene Understanding Systeme zu programmieren, also welche die mehr als 100 Zeilen Code aufweisen und die mit genügend Manpower ausgebaut werden können.

Im Gegensatz dazu sind herkömlmiche Grammars, stochastic Grammars und Domain-Specific Languages eine Einbahnstraße. Es sind keine echten Computerprogramme sondern es sind Bison/Yacc-Eingabedateien. Man kann damit zwar etwas parsen, aber nur wenn die Komplexität gering ist. Das ganze hat weniger etwas mit der Grammar als solche zu tun, sondern eher damit auf wieviel Lines of Code man diese hochskalieren kann. Das heißt, die Vermutung lautet wie folgt: Scene Understanding = hochkomplexe Aufgabe = viele Lines of Code = UML plus Javacode.

13.4 Class Tree

Auf den ersten Blick sind größere Projekte die nach der objektorientierten Programmierung durchgeführt werden hochgradig unübersichtlich. Ich habe mir mal die Mühe ein RPG Game was bei github gehostet ist, und aus 4000 Zeilen Code besteht in ein UML Diagramm zu überführen und anzuzeigen. Das ergebnis war eine 2 MB große PNG Datei die epische Ausmaße annahm und wo sehr viele Klassen und nochmehr Verbindungslinien unter diesen Klassen enthalten waren. Sehr unübersichtlich das ganze insbesondere weil der Code von jemand anderem stammte.

Auf den ersten Blick ist das ein klarer Punktsieg für dezidierte Non-OOP-Programmierung wo man also auf ein derartiges Chaos verzichtet und lieber alle Methoden linear untereinander schreibt, alphabetisch geordnet versteht sich. Doch nicht so schnell. So unübersichtlich wie zunächst gedacht ist das UML Diagramm nicht. Weil, der Hauptgrund warum es konfus aussah war nicht OOP sondern der Parser der das UML Diagramm erzeugt hat. Würde man das UML Diagramm manuell erstellen oder einen Parser verwenden der intelligenter ist als pyreverse würde man zunächst einmal Packages im Diagramm erkennen, also größere Strukturen zu denen Klassen zusammengefasst werden und man würde die Verbindungslinien kreuzungsfrei zeichnen. Ich kanne zwar keinen Parser der sowas automatisch kann, aber manuell könnte man es in jedem Fall so machen.

Aber nicht nur das. Wenn man eine leicht abstraktere UML Darstellung wählt, reduziert sich die Komplexität weiter. Das obige 4000 Zeilen Beispiel git-hub Projekte gleiderte sich in Unterdirektories wo jeweils die Packages drin waren. Man kann neben der grafischen Notation also auch einen Klassen-Baum textuell zeichnen, wo man ähnlich wie bei einem Directory Baum die Packages hierarchisch anordnet und darunter dann die Klassen hineinsortiert. Freilich gehen dabei die Verbindungen zwischen den Klassen verloren, die sieht man nur im UML Chart, aber dafür passt das ganze jetzt auf eine handliche DIN A4 Seite drauf. Und wenn das noch immer zu unübersichtlich sein sollte, kann man weitere Tools nutzen um die Klassen zu durchbrowsen, fast alle großen IDEs bieten sowas an, wo man links in der Outline Ansicht die Klassennamen sieht und sie ausklappen kann.

Anders formuliert, wenn man unbedingt möchte, kann man natürlich ein unübersichtliches UML Diagramm zu erzeugen um zu zeigen, dass OOP generell der falsche Ansatz ist. Doch mit etwas mit Liebe zum Detail stellt sich dieser vorwurf als Haltlos heraus. Im Gegenteil, UML und OOP ist die beste Erfindung seit geschnitten Brot. Insbesondere größere Projekte lassen sich damit übersichtlich gestalten, selbst wenn der Code buggy sein sollte und hastig dahinprogrammiert wurde.

Nebenbei bemerkt, was das Spiel inhaltlich nicht besonders gut. Schon im Startmenü ist das Programm abgestürzt, ohne wichtigen Grund, normalerweise ein klarer Fall für die Issuetracker doch so tief wollte ich nicht einsteigen. Und außerdem sind abstürtzte Programme nichts ungewöhnliches sondern der Normalfall.

Aber zurück zur Objektorientierten Programmierung. Der große Vorteil besteht darin, dass man sich aus einem unbekannten Projekt eine einzelne Klasse heraussuchen kann und relativ leicht sagen kann, wie sie funktioniert. Man muss nur schauen von wo sie etwas erbt und wo sie eingebunden ist und schon weiß man alles über diese Klasse. Wenn sie einem nicht gefällt, schreibt man sie komplett um, und lässt die anderen Codezeilen so wie sie sind. Ich will damit sagen, dass OOP nicht irgendein Paradigma ist wo man dafür und dagegen sein kann, sondern OOP ist Here-to-stay. Es kommt dabei gar nicht mal auf die konkrete Programmiersprache an, das oben erwähnte github Projekt war in Python geschrieben, sondern auch in Java, C#, C++ und PHP kommen die Vorteile zum Tragen.

Auch wenn in vielen religiösen Debatten a la C++ vs. Java die Unterschiede betont werden, sind doch die Maintreamsprachen sich sehr ähnlich. Die Unterschiede zwischen C++ und Python sind so groß nicht. Klar, es gibt sie. Die eine Sprache wird kompiliert die andere nicht. Aber auch das ist nicht so sicher wie es den Anschein hat. Beispielsweise kann man Python Code mittels Jython für die JVM kompilieren, während man C++ interpretiert ausführen kann und sogar auf Pointer verzichten. Das verbindene Element in Programmierprojekten ist, dass es einmal eine Maßzahl Lines of Code gibt, die den Projektfortschritt angibt und zweitens den bereits erwähnten objektorientierten Programmierstil.

Es gibt derzeit keine automatische Compiler mit denen man Python nach Java und von dort nach C# konvertieren kann, wenn man das jedoch manuell ausführt ist der Aufwand so groß nicht. Keine Ahnung wie lange sowas dauert, aber mehr als 10 Zeilen Code am Tag kann man auf jeden Fall von einer Sprache in die andere übersetzen. Ich würde mal schätzen 100 Zeilen am Tag oder sogar nochmehr sind möglich. Wohlgemerkt, wenn man das C++ Programm schon hat, was nach Python übertragen werden soll. Das ist im Grunde eine Arbeit die relativ zügig geht, wenn man in beiden Sprachen zu Hause ist.

Auch Onine-Portale wie Repl.it machen den Eindruck, als ob es letztlich nur die eine Programmiersprache namens “Java/Python/PHP/C/C#/C++/Ruby” gibt, in der die Algorithmen und UML Diagramme formuliert sind. Mein Eindruck ist eher, dass die wahren Unterschiede in der Codegröße zu suchen sind. Das heißt, ein Projekt aus 100 Llnes of Code in Python hat fast nichts mehr gemeinsam mit einem 4000 Lines of Code Projekt was ebenfalls in Python geschrieben wurde. Wärend ein 4000 LoC Java und ein 4000 LoC C++ Projekt sehr große Ähnlichkeit besitzen.

13.5 Grammar based Action parsing

Die Idee kontextfreie Grammatiken zu nutzen um Videos zu parsen ist nicht neu. Die ersten Paper wurden in den 1990’er Jahren veröffentlicht. In der leichten Variante wird der Inhalt des Bildes in einen

Scene Graphen überführt. Das heißt, die Bäume gehören zur Straße, der Strauch des Baumes zum Baum usw. bis man den kompletten Bildinhalt als Abstract Syntax Tree ausgewertet hat. Anspruchsvoller wird es, wenn auch der zeitliche Ablauf geparst werden soll. Also beispeilsweise das Überqueren eines Fußgängers des Zebrastreifens. Dafür benötigt man dann bereits eine Sprache, so beispielweise “crossing(Person1,ZoneB)”, worin man den Event ausdrückt.

Schaut man sich die Vielzahl der Veröffentlichungen an, so scheinen die Robotik-Leute Gefallen an grammar basierendem Scene Parsing gefunden haben. Die Paper werden häufig zitiert und es gibt immer neue Ideen in dieser Richtung. Das Problem was ich mit Scene Grammars habe ist folgendes: Als Technologie wird die Grammar und die Domain-Spezifische Sprache in den Mittelpunkt gestellt, das man also damit das Scene Parsing Problem lösen könnte. Obwohl beides durchaus mächtige Werkzeuge sind, reicht es leider nicht aus. Sondern die eigentliche Technologie die man benötigt lautet Lines of Code. Das heißt konkret: wenn der Scene Parser gut sein soll muss er viele LoC besitzen, sonst klappt es nicht. Und ja, diese simple Maßzahl nach dem Motto: je mehr Metall in dem Flugzeug, desto wird es fliegen reicht aus, um die Software zu bewerten.

Anders formuliert lautet die Basis Technologie um einen Scene Parser zu schreiben nicht etwa dass man einen Stack hat, den man rekursiv durchsucht um einen Graphen zu erstellen, sondern die Basistechnologie lautet, dass man in einem UML Layout Programm sich Klassen notiert und diese in einem agilen Softwareprojekt dann in einer Hochsprache ausformuliert. Mit Sicherheit wird man in einer Klasse auch eine Grammar verwenden, aber eben nicht nur. Man braucht noch sehr viel mehr Module bis das System zuverlässig funktioniert. Genau genommen lässt sich der Fortschritt an den Lines of Code ausmachen.

Der Grund warum in der Literatur so gerne Grammars und Domänenspezifische Sprache zitiert werden hat damit zu tun, dass die Informatik versuchen möglichst wenig zu programmieren. Jeder der schonmal Java Code editiert hat wird sich möglichst davor drücken. Insofern lautet die Idee, dass man das umgehen will. Das Ziel was derzeit in den Papern verfolgt wird ist einen Scene Parser zu bauen der bitteschön klein und handlich ist. Also effektiv aus nicht mehr als 500 Lines of Code besteht in der Idee dass es eben nicht auf LoC ankommt sondern die Grammar im Mittelpunkt steht.

Das diese Idee eine Sackgasse ist erkennt man wenn man sich heutige grammar based Scene Parser in Echt anschaut. Sie parsen zwar irgendwas und einige geben auch natürliche Sprache aus, doch irgendwas fehlt noch. Richtig, es fehl eine Idee wie sich auftretende Bugs zuverlässig beseitigen lassen. Also die Erkennungsleistung auf 100% zu steigern. Wenn man davon ausgeht, dass eine bisschen Grammar Hokus Pokus plus eine Domänensprache ausreichen wird man die Skalierung nach oben nicht hinbekommen. Weil eine Grammar nichts anderes ist, als eine Template Datei wo man in seiner sehr kompakten Schreibweise eine Sprache definiert. Leider ist diese Sprache nicht besonders mächtig, es lassen sich damit nur sehr künstliche Szene auswerten.

Die bessere Antwort lautet, dass man die Domänenmodellierung über das UML Diagramm druchführt. Das heißt, die Anzahl von 30 und noch mehr Klassen sind die Domäne. Die Klassen sind ausführbar, lassen sich debuggen und enthalten ablauffähigen Code. So und nicht anders sehen leistungsfähige Scene Scrapper aus.

Literatur

[1] Samir Araujo and Luiz Chaimowicz. A synthetic mind model for virtual actors in interactive storytelling systems. In *AIIDE*, 2009.

[2] Rodney A Brooks. Intelligence without representation. *Artificial intelligence*, 47(1-3):139–159, 1991.

[3] Joanna Bryson and Brendan McGonigle. Agent architecture as object oriented design. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 15–29. Springer, 1997.

[4] Tomas Bures, Ewen Denney, Bernd Fischer, and Eugen C Nistor. The role of ontologies in schema-based program synthesis. 2004.

[5] Greg Butler, Andrea Gantchev, and Peter Grogono. Object-oriented design of the subsumption architecture. *Software: Practice and Experience*, 31(9):911–923, 2001.

[6] Jackie Chi Kit Cheung, Hoifung Poon, and Lucy Vanderwende. Probabilistic frame induction. *arXiv preprint arXiv:1302.4813*, 2013.

[7] OWL Creation. To generate the ontology from java source code. *IJACSA Editorial*, 2011.

[8] Oscar Danielsson, Anna Syberfeldt, Rodney Brewster, and Lihui Wang. Assessing instructions in augmented reality for human-robot collaborative assembly by using demonstrators. *Procedia CIRP*, 63:89–94, 2017.

[9] Giuseppe De Giacomo, Yves Lespérance, Hector J Levesque, and Sebastian Sardina. Indigolog: A high-level programming language for embedded reasoning agents. *Multi-agent programming: Languages, platforms and applications*, pages 31–72, 2009.

[10] Ryan P Dibley, Michael J Allen, and Nassib Nabaa. Autonomous airborne refueling demonstration: Phase i flight-test results. 2007.

[11] Andre Eisenmenger. Semantic web: Konzept einer owl-ontologie fÃŒr einen chatbot. Master’s thesis, HAW Hamburg, Berliner Tor 5, 20099 Hamburg, 2008.

[12] Gabriel Farah, Juan Sebastian Tejada, and Dario Correal. Openhub: a scalable architecture for the analysis of software quality attributes. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 420–423. ACM, 2014.

[13] Simone Fuchs, Stefan Rass, and Kyandoghere Kyamakya. Integration of ontological scene representation and logic-based reasoning for context-aware driver assistance systems. *Electronic Communications of the EASST*, 11, 2008.

[14] Pedro Azevedo Isidro. *Automatic code generation for attitude and orbit control systems using domain-specific languages*. PhD thesis, Instituto Superior Tecnico, Lisbon, Portugal, 2014.

[15] Han-byul Jang, Jang-woon Kim, and Chil-woo Lee. Augmented reality cooking system using tabletop display interface. In *International Symposium on Ubiquitous VR*, page 1, 2007.

[16] Capers Jones. A short history of lines of code (loc) metrics. *Capers Jones & Associates LLC, Narragansett*, pages 1–12, 2008.

[17] David A Kosower and Juan J Lopez-Villarejo. Flowgen: Flowchart-based documentation for c++ codes. *Computer Physics Communications*, 196:497–505, 2015.

[18] James J Kuffner and J-C Latombe. Fast synthetic vision, memory, and learning models for virtual humans. In *Computer Animation, 1999. Proceedings*, pages 118–127. IEEE, 1999.

[19] Yaling Liu. *A process-based search engine*. PhD thesis, University of Kansas, 2009.

[20] Yaling Liu and Arvin Agah. Crawling and extracting process data from the web. *Advanced Data Mining and Applications*, pages 545–552, 2009.

[21] Minhua Ma and Paul McKevitt. Building character animation for intelligent storytelling with the h-anim standard. In *Eurographics Ireland Workshop Series (EG-Ireland)*, volume 2, pages 9–15. University of Ulster, 2003.

[22] Minhua Eunice Ma. Confucius: An intelligent multimedia storytelling interpretation and presentation system. *First year report: School of computing and intelligent systems, faculty of informatics, University of Ulster, Magee*, 2002.

[23] Marco Melega. Autonomous collision avoidance for unmanned aerial systems. 2014.

[24] Marek Mittmann, Jarosław Francik, and Adam Szarowicz. *Physics-based animation of human avatars*. PhD thesis, Citeseer, 2007.

[25] Philippe Morignot and Fawzi Nashashibi. An ontology-based approach to relax traffic regulation for autonomous vehicle assistance. *arXiv preprint arXiv:1212.0768*, 2012.

[26] Hansrudi Noser, Olivier Renault, Daniel Thalmann, and Nadia Magnenat Thalmann. Navigation for digital actors based on synthetic vision, memory, and learning. *Computers & graphics*, 19(1):7–19, 1995.

[27] Hansrudi Noser and Daniel Thalmann. Sensor based synthetic actors in a tennis game simulation. In *Computer Graphics International, 1997. Proceedings*, pages 189–198. IEEE, 1997.

[28] Abhijit Ogale, Alap Karapurkar, and Yiannis Aloimonos. View-invariant modeling and recognition of human actions using grammars. *Dynamical vision*, pages 115–126, 2007.

[29] Joanna I Olszewska and Thomas L McCluskey. Ontology-coupled active contours for dynamic video scene understanding. In *Intelligent Engineering Systems (INES), 2011 15th IEEE International Conference on*, pages 369–374. IEEE, 2011.

[30] Janne Parkkila, Filip Radulovic, Daniel Garijo, María Poveda-Villalón, Jouni Ikonen, Jari Porras, and Asunción Gómez-Pérez. An ontology for videogame interoperability. *Multimedia Tools and Applications*, 76(4):4981–5000, 2017.

[31] Anu Rastogi, Paul Milgram, and Julius J Grodski. Augmented telerobotic control: a visual interface for unstructured environments. In *Proceedings of the KBS/Robotics Conference*, pages 16–18, 1995.

[32] Craig W Reynolds. Computer animation with scripts and actors. In *ACM SIGGRAPH Computer Graphics*, volume 16, pages 289–296. ACM, 1982.

[33] Craig William Reynolds. *Computer animation in the world of actors and scripts.* PhD thesis, Massachusetts Institute of Technology, 1978.

[34] Juan Carlos SanMiguel, José M Martinez, and Álvaro Garcia. An ontology for event detection and its application in surveillance video. In *Advanced Video and Signal Based Surveillance, 2009. AVSS'09. Sixth IEEE International Conference on*, pages 220–225. IEEE, 2009.

[35] Lauro Snidaro, Massimo Belluz, and Gian Luca Foresti. Domain knowledge for surveillance applications. In *Information Fusion, 2007 10th International Conference on*, pages 1–6. IEEE, 2007.

[36] Daniel Thalmann. A new generation of synthetic actors: the real-time and interactive perceptive actors. In *Proc. Pacific Graphics*, volume 96, pages 200–219, 1996.

[37] Yezhou Yang, Anupam Guha, Cornelia Fermüller, and Yiannis Aloimonos. Manipulation action tree bank: A knowledge resource for humanoids. In *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*, pages 987–992. IEEE, 2014.

[38] Lei Zou, Ruizhe Huang, Haixun Wang, Jeffrey Xu Yu, Wenqiang He, and Dongyan Zhao. Natural language question answering over rdf: a graph data driven approach. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 313–324. ACM, 2014.