Studienarbeit

Vergleich von Java und XL anhand des Agentenmodells "Sugarscape".

von

Bernd Gräber

Matrikelnummer: 9803722

Vom 14.03.06 bis 14.09.06

Lehrstuhl Graphische Systeme an der BTU Cottbus

Gliederung

1 Aufgabenstellung

Das Hauptthema der Studienarbeit ist ein Vergleich zweier Sprachen. Zum einen die Objektorientierte Programmiersprache Java und zum anderen die Implementation einer relationalen Wachstumsgrammatik mittels XL. Es wird hierbei ein bestimmtes Programm in beiden Sprachen erstellt. Das Thema wurde an das Buch „Growing Artificial Societies“ von Joshua M. Epstein und Robert Axtell angelehnt. Der Vergleich selbst beinhaltet die Größe der Programme (LOC), die Prozessor- und die Speicherauslastung, sowie ein Vergleich der Laufzeiten bei ähnlichen Bedingungen.

2. Das Zuckerlandschaft

Im Buch „Growing Artificial Societies“ beschreiben die Autoren eine “Gesellschaft” von sogenannten „Agenten“, welche in einer „Zuckerlandschaft“ existieren. Die Wesen dieser Welt überleben, in dem sie den Zucker von einzelnen Feldern aufnehmen und verbrauchen. Es wird so eine Umwelt simuliert, um bestimmten Verhaltensweisen von Lebensformen auf den Grund zu gehen. Joshua M. Epstein und Robert Axtell programmierten es so weit, dass sie komplexe gesellschaftliche Strukturen, wie z.B. Erbschaften, Freundschaften, Feindschaften, Handel, Reichtum und Armut, Krankheiten und Umweltverschmutzung simulieren konnten. Aus diesen reichhaltigen Möglichkeiten wählte ich einige aus. Im Detail wurden folgende Muster in die Programme aufgenommen:

1. Die grundlegenden Mechanismen, die einen Agenten erst lebensfähig machen
2. Die Möglichkeit, dass sich 2 Agenten fortpflanzen können
3. Die Unterteilung der Agenten in Räuber und Nicht-Räuber
4. Das Altern der Agenten
5. Die Unterteilung der Agenten in verschiedene Stämme und das Ändern der Stammeszugehörigkeit
6. Agenten verursachen durch ihre Aktionen Verschmutzung in ihrer Umwelt
7. Agenten können stammesfremde Agenten angreifen
8. Agenten infizieren sich mit Krankheiten
9. Agenten treiben Handel

In dieser künstlichen Welt wird der Zucker, welcher auf den einzelnen Feldern liegt, von den Agenten verbraucht, um am Leben zu bleiben. Um ein weiteres Überleben zu gewährleisten, muss deshalb Zucker zurückgeführt werden. Dies geschieht über die sogenannten Wachstumsregeln. Auch hier gibt es verschiedene Ansätze, aus denen einige ausgewählt wurden.

1. Der Zucker wächst auf allen Feldern um einen konstanten Wert nach
2. Der Zucker wächst nur in einer Hälfte der gesamten Zuckerlandschaft nach.
3. Der Zucker wächst in Abhängigkeit des Zuckervorrates des Nachbarfeldes nach.

Die einzelnen Regeln werden im nächsten Kapitel genauer erklärt. Der grundsätzliche Ablauf des Lebens auf dem Zuckerfeld sieht folgendermaßen aus: Es wird per Zufall ein Agent aktiviert. Dieser verhält sich seiner Regelung (siehe Regeln der Agenten) entsprechend. Danach wird ein anderer Agent per Zufall aktiviert und wird ebenfalls den Regeln entsprechend abgearbeitet. Dies geschieht, solange bis alle Agenten durchgenommen wurden. Zum Abschluss wird der Zucker auf allen Feldern entsprechend der Wachstumsregel nachwachsen. Eine solche Aktionsfolge mit abschließendem Zuckerwachstum wird von nun an Runde genannt.

3. Die Regeln der Agenten

Dies sind die Regelmechanismen, nach denen ein Agent Entscheidungen trifft. Ebstein & Axtell hatten die Idee diese Regeln so einfach wie möglich zu halten, um zu überprüfen, ob komplexe Verhaltensmuster sich aus einfachen Regeln ergeben können. Dieses Verfahren wird auch hier angewendet. Die Regeln sind nahezu vollständig aus dem Buch selbst übernommen und haben nur einige wenige Anpassungen erlebt.
Der Agent selbst besitzt verschiedene Attribute. Dazu gehören vor allem seine Sichtweite und sein Metabolismuswert. Die Sichtweite wird benötigt, wenn der Agent seine Umgebung beobachten möchte. Dabei zeigt die Größe des Wertes die Anzahl der Zuckerfelder an, die in den jeweiligen Himmelsrichtungen beobachtet werden können. Die Größe des Metabolismus zeigt die Menge des Zuckers, welche vom Agenten bei einer Aktion verbraucht wird. Diese Werte können in den Programmen vom Nutzer frei verändert werden. Standardmäßig ist sowohl für die Sicht, als auch für den Metabolismus die Zahl 5 eingestellt. Je nach Regelwerk können und müssen die Agenten noch andere Attribute aufweisen. Dies wird bei den entsprechenden Regeln näher erläutert.

3.1. Die Grundregel

Diese Regel ist die wichtigste Regel, die es für einen Agenten gibt. Der Agent, der aktiv ist, schaut seiner Sichtweite entsprechend in alle 4 Himmelsrichtungen und sucht sich das Zuckerfeld mit der größten Zuckermenge, welcher nicht von einem anderen Agenten besetzt ist, aus. Er bewegt sich zu diesem Feld und sammelt sämtlichen dort liegenden Zucker auf. Danach verringert sich die Menge des Zuckers, die dieser Agent besitzt, um den Wert seines Metabolismus. Sollte dabei sein Vorrat auf 0 oder darunter fallen, wird er von dem Zuckerfeld entfernt. (Er ist verhungert.) Diese Regel wird im Buch auf Seite 25 beschrieben.

3.2. Fortpflanzung

Es wird bei dieser Regel ein neues Attribut für die Agenten eingeführt. Jeder Agent ist von nun an einem bestimmten Geschlecht zugeordnet. Wenn 2 Agenten unterschiedlichen Geschlechtes nach der Bewegung auf benachbarten Zuckerfeldern stehen und sie genügend große Zuckervorräte haben und es mindestens ein freies Zuckerfeld neben einem von ihnen gibt, dann pflanzen sich die Agenten fort und ein neuer Agent wird auf das Zuckerfeld gesetzt. Die Attribute des neuen Agenten werden durch die Elternteile ermittelt. Einer der möglichen Werte wird per Zufall ausgewählt. Diese Zahl erhält das Kind. Der aktive Agent ist hierbei der Auslöser der Reproduktion und er wird sich mit allen Agenten paaren, welche zum Zeitpunkt in seiner Nachbarschaft stehen und die genannten Kriterien erfüllen. Auf den Seiten 55/56 der Vorlage finden wir den Ursprung.

3.3. Räuber

Auch hier wird mittels eines neuen Attributes eine Trennung unter den Agenten vollzogen. Einige wenige Agenten erhalten die Bezeichnung „Räuber“, während andere die Bezeichnung „Beute“ erhalten. Beute-Agenten verhalten sich wie in den anderen Regeln beschreiben, das bedeutet sie bekommen ihren Zucker aus den Zuckerfeldern. Räuber jedoch suchen in ihrer Sichtweite nicht nach Feldern mit viel Zucker, sondern nach solchen mit Beute-Agenten, deren Zuckervorrat am größten ist. Sie bewegen sich auf diese Punkte, töten den Beute-Agenten und addieren seinen Zuckervorrat zu dem ihren hinzu. Räuber besitzen einen größeren Sichtweite-Wert als Beute-Agenten. Im Gegenzug

werden Beute-Agenten nicht auf Felder gehen, wenn sie dann einen Räuber in ihrer Sichtweite haben. Das Konzept der Räuber ist eine Erweiterung der ursprünglichen Idee und ist nicht im Buch zu finden.

3.4.Alter

Diese Zusatzregel ist recht einfach. Jeder Agent hat ein weiteres Attribut bekommen, das sogenannte Alter. Agenten starten ihr Leben mit einem Alter von 0 und beenden es mit einem Alter von 70 oder 90 Runden. Dies hängt vom jeweiligen Geschlecht ab. Jede Aktion des Agenten erhöht das Alter um einen Punkt. Fortpflanzung (Punkt 3.2.) ist auch nur innerhalb eines bestimmten Alters möglich. Die untere Grenze liegt hier bei einem Wert von 16 und die obere Grenze liegt geschlechtsspezifisch bei 40 bzw. 60.
Auf den Seiten 57/58 wird dieses Thema im Buch behandelt.

3.5.Stämme

Agenten unter dieser Regelung haben einige „kulturelle“ Ansichten. Dies sorgt für eine Unterteilung in verschiedene Stämme. Es gibt z.B. 9 abstrakte „Themen“. Ein Agent kann zu jedem Thema eine von 2 Positionen beziehen. Je nach „Meinungsbild“ des Agenten wird er einem Stamm zugeordnet. Agenten pflanzen sich nur innerhalb desselben Stammes fort. Der aktive Agent wird nach seiner Bewegung mit anderen Agenten interagieren und dort möglicherweise die Meinung zu einem Thema ändern und damit eventuell sogar erreichen, dass der Angesprochene den Stamm wechselt. Ab Seite 71 wird der kulturelle Prozess bei Epstein/Axtell beschreiben)

3.6.Verschmutzung

Agenten produzieren mit allen ihren Aktionen, die mit Zucker zu tun haben, Verschmutzung. Diese Verschmutzung hat keinerlei negative Effekte auf die Agenten oder die Umgebung, sondern sie bewirkt, dass Agenten Felder mit hoher Verschmutzung eher meiden, auch wenn dort viel Zucker liegt.
Verschmutzung wird auf Seite 45 eingeführt.

3.7.Krieg

Agenten können hier Agenten anderer Stämme angreifen und töten. Vorraussetzung dafür ist, dass der zu bekämpfende Agent weniger Zucker besitzt (also schwächer ist) und dass es keinen „feindlichen“ Agenten gibt, der auf der neuen Position das gleiche mit dem ehemaligen Angreifer machen kann. Der Angreifer erhält für diese Aktion nicht nur den Zucker aus dem Feld, sondern auch eine „Beute“ vom besiegten Agenten. Dies ist nötig, da es sonst kaum zu Kämpfen kommen wird, da nichtbesetzte Punkte im Normalfall mehr für das Überleben des Agenten geeignet sind. Räuber sind von diesen Kämpfen ausgenommen, da sie im Normalfall zu wenige sind, um diese Fälle häufig eintreten zu lassen.
Krieg ist als direkte Folge der kulturellen Prozesse auf den Seiten 82/83 erläutert worden.

3.8.Krankheiten

Agenten werden mit einem Immunsystem und verschiedenen Krankheiten ausgestattet. Jede Krankheit, unter der ein Agent leidet, sorgt für einen erhöhten Metabolismus. Das Immunsystem passt sich mit der Zeit den verschiedenen Krankheiten an und bekämpft sie

irgendwann erfolgreich. Krankheiten und Immunsysteme werden innerhalb des Codes als Arrays repräsentiert. Das Kurieren erfolgt mittels mehrerer Vergleiche. Wenn das kleinere Array der Krankheit einem Teil des Arrays des Immunsystems entspricht, wurde sie erfolgreich bekämpft. Pro Runde wird das System, den noch vorhandenen Leiden, an einem Punkt angepasst. Krankheiten werden durch Kontakt mit Nachbarn übertragen. Das Thema wird in Kapitel V ab seite 138 behandelt.

3.9.Handel

Für diese Regel ist es notwendig, einen zweiten Rohstoff in das System einzufügen. Die Agenten benötigen beide um zu überleben. Der zweite Rohstoff wird Gewürz genannt. Wenn 2 Agenten sich treffen, werden sie versuchen einen Handel abzuschließen, der die beiden Rohstoffe verteilt. Dies geschieht nur, wenn beide Agenten aus diesem Handel einen Vorteil ziehen. Hierzu werden 2 Werte berechnet, die auf der einen Seite den Wohlstand (Welfare-Wert) und auf der anderen Seite den Bedarf an einem Rohstoff (MRS-Wert) repräsentieren. Die beiden Zahlen berechnen sich aus den Vorräten an Zucker und Gewürz, sowie den beiden Metabolismen. Der Welfare-Wert wird mittels der Formel:

$$Zucker^{Zuckermetabolismus/Summe\,der\,Metabolismen} * Gewürz^{Gewürzmetabolismus/Summe\,der\,Metabolismen}$$

erstellt. Das bedeutet z.B., dass bei einem niedrigen Zucker-Metabolismus, eben dieser Rohstoff als nicht sonderlich „wertvoll“ eingeschätzt wird, und der Agent sich somit auch mit einem niedrigen Vorrat an Zucker als jemanden betrachten wird, der viel Wohlstand besitzt. Der MRS-Wert ist das Verhältnis der möglichen Überlebenszeiten pro Rohstoff. Wenn berechnet wird, das der Agent mit seinem derzeitigen Gewürz-Vorrat noch 3 Runden und mit seinem Zuckervorrat noch 5 Runden überleben würde, wird Gewürz zum wertvolleren Rohstoff erklärt. Wenn jetzt 2 Agenten anfangen wollen zu handeln, wird als erstes der MRS-Wert der beiden berechnet und miteinander verglichen. Dies bestimmt, ob Zucker oder Gewürz vom aktiven Agenten gekauft wird. Der Umtauschkurs berechnet sich aus der Wurzel des Produktes der beiden MRS-Werte. Nachdem dies festgelegt wurde, wird geprüft, ob ein solcher Handel den Wohlstand beider Agenten erhöhen würde. Sollte dies nicht zutreffen, wird der Handel abgebrochen. Die Agenten tauschen in einer Runde, solange Handelsgüter aus, bis diese Bedingung eintritt.
Wenn Agenten ein gutes ausgewogenes Verhältnis zwischen den beiden Rohstoffen haben, besitzen sie eine Resistenz gegenüber Räubern. Dies wird durch den MRS-Wert bestimmt, bzw. dessen Reziproke sollte er über 1 sein. Dies ist dann die Wahrscheinlichkeit, mit der ein Angriff eines Räubers abgewehrt werden kann.
Handel kann in Kapitel IV ab Seite 94 nachgelesen werden. Die Resistenz von Agenten gegen Raubtiere ist eine Erweiterung des Schemas.

4.Die Regeln des Zuckerwachstums

Agenten sammeln während ihrer Existenz Zucker von den verschiedenen Zuckerfeldern. Um ein Weiterbestehen zu gewährleisten, muss Zucker wieder in das System gebracht werden. Dies wird durch die Wachstumsregeln beschrieben. Ähnlich den Regeln für die Agenten gibt es auch hier bestimmte Mechanismen, welche die Art und Weise beschreiben, wie der Zucker auf das Feld zurückkehren (nachwachsen) kann. 3 verschiedene Systeme wurden implementiert. (Zusätzlich existieren 2 Untersysteme) Die erste und zweite Regel werden im Buch ab Seite 42 erklärt. Die dritte stellt eine Erweiterung des Systems dar.

4.1. die Grundregel

Die einfachste Möglichkeit Zucker nachwachsen zu lassen, besteht darin, auf jedem Zuckerfeld eine bestimmte Menge Zucker aufzufüllen. (z.B. Jedes Zuckerfeld erhält eine Einheit Zucker.) Dabei darf allerdings die maximale Menge an Zucker auf einem Feld nicht überschritten werden. Man erreicht damit einen konstanten Zuwachs an Nahrung in unserer künstlichen Welt.

4.2. Wachstum nach Saison

Mit dieser Regel werden die Auswirkungen von Jahreszeiten und der einhergehenden Nahrungsknappheit simuliert. Nur in einer Hälfte der Zuckerlandschaft wächst Zucker entsprechend der Grundregel nach. Die andere Hälfte erfährt keinerlei Wachstum. Nach einer vorher bestimmten Anzahl Runden wechselt dieses Prinzip. Von diesem Zeitpunkt an wächst der Zucker nur noch auf der vorher brachliegenden Hälfte und auf der anderen gibt es kein Wachstum mehr. So soll der Sommer/Winter-Wechsel auf der Welt aufgezeigt werden.

4.3. Wachstum nach Vorkommen

Eine dritte Möglichkeit besteht darin, das Wachstum von den umstehenden Punkten abhängig zu machen. Neuer Zucker entsteht also vor allem dort, wo noch anderer Zucker existiert, genauso, wie Pflanzen sich erst in der Nähe der eigenen Art ansiedeln. Es wird hierbei das Nachbarfeld ausgesucht, das die größte Anzahl Zuckereinheiten besitzt. Ein vorher eingestellter Prozentsatz dieses Zuckers wird als Wachstum auf dem ursprünglichen Feld hinzuaddiert.

4.4. Grundwachstum und Verschmutzung

Die 1. Subsystemwachstumsregel ist keine eigenständige Regel, sondern wurde erstellt, um mit der Agentenregel 6 (oder höher) zusammenzuwirken (vgl. 3.6. Verschmutzung). Der Zuckerzuwachs läuft genauso ab, wie in Punkt 4.1. beschrieben. Zusätzlich wird nach dem Wachstum die Verschmutzung auf den einzelnen Feldern auf die Nachbarfelder verteilt, das heißt, es wird die durchschnittliche Verschmutzung der 5 Felder errechnet und das Ergebnis ist die neue Verschmutzung auf dem Ausgangsfeld.

4.5. Grundwachstum für Zucker und Gewürz

Die 2. Subsystemwachstumsregel verhält sich ähnlich Regel 4. Auf den Punkten wächst allerdings in Abhängigkeit vom Zufall Zucker oder Gewürz nach.

5. Die Java-Implementierung

5.1.Allgemeines

Bevor ich mich mit den eigentlichen Klassen beschäftigte, machte ich mir Gedanken, wie sich das Programm im fertigen Zustand darstellen sollte. Der Aufbau sollte recht einfach und übersichtlich sein. Ich benötigte also einen Teil zur Darstellung der 3D-Szene und einen Teil, wo ich die verschiedenen Einstellungen tätigen konnte, nach denen dann das

„Leben“ ablaufen sollte. Mir kam der Gedanke die „Einstellungen“ ähnlich zu gestallten, wie es in verbreiteten Textverarbeitungsprogrammen üblich ist. Aus einem Menu werden die entsprechenden Möglichkeiten gewählt und es erscheint ein kleines Extrafenster, in welchem man die Optionen festlegen kann. Folgende Unterteilung der Optionen bot sich an:

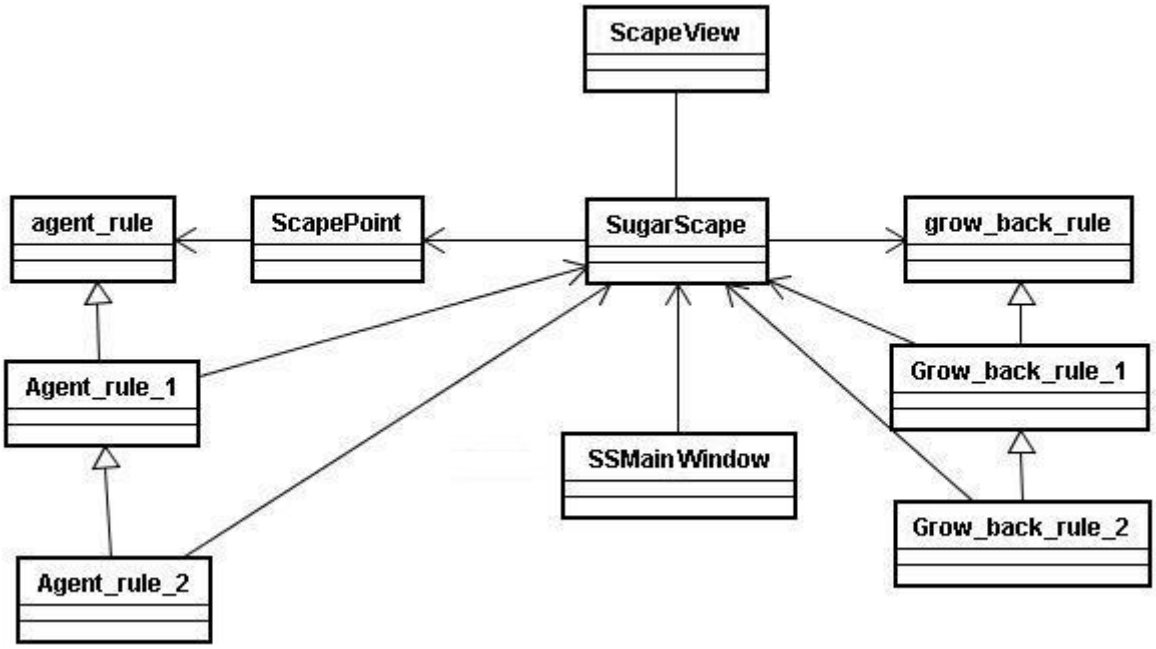
1. Einstellungen für die Zuckerlandschaft.
   Darunter verstehe ich alle Dinge, welche die Zuckerlandschaft selbst betreffen, also die Ausdehnung , sowie die anfängliche Verteilung und das Maximum des Zuckers. Zu diesem Zeitpunkt dachte ich über 2 verschiedene Arten nach, wie ich die Werte der Zuckerfelder generieren könnte. Zum einen ein Ansatz, genaue Werte über eine Datei einzulesen, und zum anderen, nur die Rahmenbedingungen festzulegen und das Feld durch Zufallswerte generieren zu lassen. Beide Varianten haben ihre Vor- und Nachteile. Ich entschloss mich beide zu realisieren.

2. Einstellungen für das Leben in der Landschaft
   Diese Optionen beinhalten die Anzahl und Attribute der Agenten, sowie die Festlegungen der Verhaltensregeln. (siehe Punkte 3-4) Die Agentenattribute geben hierbei immer den maximalen möglichen Wert an.

3. Verschiedene Menupunkte, welche das Starten des Lebenskreislaufes und das Beenden des Programms beinhalten.

Somit blieb nahezu der gesamte Platz des Fensters für die 3D-Darstellung übrig.
Ein weiteres Detail, das geradezu ins Auge fällt, sind die verschiedenen Regeln, welche implementiert werden müssen. Der grundsätzliche Aufbau einer Agentenregel lässt sich etwa so beschreiben: 1. suche einen gutes Feld 2. gehe zum Feld. 3. Sammle und verbrauche Zucker 4. tue andere der Regel entsprechende Dinge. Bei den Regeln für das Nachwachsen lässt sich ebenfalls eine solche Parallele beobachten: 1. Prüfe bei jedem Feld, ob ein Nachwachsen der Regel entspricht. 2. Wenn ja, lass dort Zucker entsprechend der Regel nachwachsen. Bei diesen Parallelen bot sich eine Vererbungsstruktur an. Des weiteren sollte das Programm jede Regel ohne zusätzliche Maßnahmen ausführen können. Dies führte zu den beiden Interfaces „agent_rule“ für die Agentenregeln und „grow_back_rule“ für die Wachstumsregeln.

Damit sind die Darstellung und die Regeln für das Leben abgedeckt. Es wurde aber noch eine Möglichkeit gebraucht, die Landschaft und die einzelnen Felder zu verwalten. Die Punkte bekamen eine eigene Klasse namens „ScapePoint“ und das Feld selbst die Klasse „SugarScape“, wobei dort die Objekte der ScapePoint-Klasse verwendet werden, um die Landschaft zu bilden.



agent_rule: Ein Interface, das es dem Programm erlaubt jede unterschiedliche Agentenregel auszuführen, ohne für jede speziell angepasst zu sein.

Agent_rule_1, Agent_rule_2, …. , Agent_rule_X : Implementierungen des agent_rule-Interface. Sie beinhalten das Verhalten der Agenten auf den Feldern.

grow_back_rule: Ein Interface, das es ermöglicht ohne Probleme verschiedene Wachstumsregeln zu nutzen.

Grow_back_rule_1, Grow_back_rule_2,...., Grow_back_rule_X: Implementierungen der verschiedenen Wachstumsregeln.

ScapePoint: Diese Klasse repräsentiert einen Feld in der Landschaft. Sie beinhaltet die Zuckermenge, das Zuckermaximum, eine Kontrolle ob auf dem Punkt ein Agent steht, sowie diverse Funktionen, welche die 3D-Darstellung des Feldes betreffen. Im späteren Verlauf musste hier noch die Möglichkeit der Gewürzexistenz berücksichtigt werden.

SugarScape. Hier werden die ScapePoint-Objekte zu der eigentlichen Landschaft zusammengefasst und verwaltet.

SSMainWindow: Dies bildet das Hauptfenster, in dem die 3D-Darstellung angezeigt wird. Es bildet auch den Schnittpunkt, in dem die Einstellungen aus den Optionsfenstern an die verschiedenen Objekte weitergegeben werden.

ScapeView: Das ist ein Panel, indem das Canvas3D-Objekt aus Java3D liegt. Hier werden alle 3D-Objekte erstellt und in die richtige Position gebracht.

5.2. Beschreibung der wichtigen Klassen

SSMainWindow: Diese Klasse leitet sich aus der Klasse JFrame ab. Sie bildet also das eigentliche Fenster, mit dem gearbeitet wird. Die Funktionen initComponents() und

initComponents2() sind für den Aufbau des Fensters, sowie des Menus verantwortlich.
Zur Auswertung der 5 Menupunkte gibt es die 5 ActionPerformed-Methoden.
CreateSSItemActionPerformed wird aufgerufen, wenn der Menupunkt mit Namen „Sugarscape“ ausgeführt wird. Da die Aufgabe dieses Menupunktes das Erstellen einer neuen Landschaft ist, wird hier als erstes eine eventuell Bestehende samt Agenten und laufender Ausführung gelöscht. Menupunkte wie die Optionen, Agentenerstellung und das Starten des Ablaufes werden deaktiviert, da diese Funktionen hier nicht ausführbar sind.
Nun gibt es 2 Möglichkeiten. In der Ersten wird die Landschaft mit den Rahmenwerten (maximaler Zuckerwert auf einem Feld sowie der Ausdehnung) und einer zufälligen Verteilung der Zuckermenge auf den Feldern erstellt. In der Zweiten wird eine über den Menupunkt eingelesene Text-Datei ausgewertet und die Landschaft so erstellt. Die entsprechende Datei muss folgendes Format aufweisen:
Die Breite der Zuckerlandschaft
Die Länge der Zuckerlandschaft
Der maximale Zuckerwert auf den Feldern
Eine Matrix der genauen Zuckerwerte für die Felder, welche durch Leerzeichen abgetrennt sind.

z.B.

4
3
6
1 4 6 3
2 3 0 1
0 5 2 6

Dies erstellt dann eine Landschaft, welche 4 * 3 Punkte groß ist, deren Zucker den Wert 6 nicht überschreitet und deren genaue Zuckerverteilung am Anfang:
1 4 6 3
2 3 0 1
0 5 2 6
entspricht.
Am Ende wird das SugarScape-Objekt mit den übergebenen Werten erstellt. Der Menupunkt „Optionen“ wird freigeschaltet.
Die Funktion optionsItemActionPerformed sorgt für die Einstellungen der Regeln für den Lebensablauf. Ähnlich wie in createSSItemActionPerformed werden zuerst eventuell existierende Objekte entfernt, die mit den neuen Einstellungen überschrieben werden. Mit den Werten aus dem Optionsfenster werden dann die Agentenobjekte und das Objekt für das Zuckerwachstum generiert. Der „erschaffe Agenten“–Menupunkt wird freigegeben und die 3D-Landschaft erstellt
In der agentsItemActionPerformed-Methode werden die Agenten zufällig auf den Zuckerfeldern platziert und deren 3D-Objekte eingefügt.
„startItemActionPerformed“ bringt ein Auswahlfenster, in dem man die Anzahl der Runden einstellen kann, wie lange der Lebenskreislauf ausgeführt werden soll. Dann wird die entsprechende Funktion aufgerufen, die den Kreislauf in Gang setzt. Sie heißt „startAgentWork“.
„closeItemActionPerformed“ beendet die Anwendung.
Die Funktion „buildView“ fügt ein Panel mit dem Canvas3D-Objekt in das Hauptfenster ein und wird aufgerufen, sobald einige 3D-Figuren existieren, die dargestellt werden können.

„buildAgents“ ist eine der wenigen Dinge, die erweitert werden müssen, sobald eine neue Agentenregel in das Programm eingeführt wird. Hier werden die konkreten Agenten-Objekte erstellt.
Um den Benutzer des Programms die Möglichkeiten zu lassen es während des Lebenskreislaufes zu verändern, oder abzubrechen, werden die entsprechenden Aktionen in einem Extra-Thread abgearbeitet. Dafür existieren die Funktionen „start“ „stop“ und „run“. Mit der oben erwähnten „startAgentWork“ wird der Thread gestartet. Die eigentliche aktionenausführende Methode (doAgentList) wird über „run“ aufgerufen.

„doAgentList“ wählt einen zufälligen Agenten aus der mit „buildAgents“ erstellten Liste und führt seine „doAction“- Funktion aus. „doAction“ wird im Interface „agent_rule“ definiert und lässt somit den Agenten seine durch die entsprechende Regel definierten Aktionen ausführen. Des weiteren wird der Thread für eine halbe Sekunde angehalten, um dem Nutzer die Möglichkeit zu geben, Veränderungen visuell zu verfolgen.

Zusammenfassend kann man sagen, diese Klasse ist für die Menuführung, das Sammeln der vom Nutzer gewünschten Daten, sowie für die Instanzzierung der eigentlichen Regeln und Datenobjekte zuständig.

Objekte der Klasse ScapePoint repräsentieren die einzelnen Zuckerfelder. Dementsprechend beschäftigen sich die Funktionen und Variablen mit den Möglichkeiten eines Feldes. Jedem Feld ist seine jeweilige Rohstoffmenge, das Rohstoffmaximum und das Vorhandensein eines Agenten auf ihm bekannt. Dafür gibt es in dieser Klasse die entsprechenden set- und get-Methoden. Für die 3D-Darstellung existiert hier eine Verbindung zum 3D-Objekt (vom Typ Box), welches das Feld in Canvas3D darstellt. Es wird auch die Farbe der Darstellung der jeweiligen Veränderung des Zuckerinhalts mit der Funktion „updateColor“ angepasst. Die anderen Funktionen dieser Klasse sind dazu da, die je nach Regel auftretende Verschmutzung zu berechnen und darzustellen. (vgl. Punkte 3.6 und 4.4)
Hohe Verschmutzung hebt die Box an. Als erstes benötige ich die Möglichkeit, die Größe des Box-Objektes während der Laufzeit zu verändern. So etwas ist zur Zeit nur mittels einer Skalierung der entsprechenden Dimension und einer Translation, um den Größenzuwachs in der falschen Richtung entgegenzuwirken, zu realisieren. Deshalb finden wir hier auch eine TransformGroup. Diese wird in der Funktion setHeigh genutzt, um die angesprochene Veränderung zu bewirken. Das Attribut pollution zeigt dann die aktuelle Verschmutzung des Zuckerpunktes an. „tempPol“ ist lediglich eine Hilfsvariable zur Berechnung der Verschmutzungsverteilung. (siehe Punkt 4.4).

Die SugarScape-Klasse behandelt die gesamte Landschaft. Hier existiert eine Matrix von ScapePoints sowie eine Liste der Agenten, die dort leben. Die visuelle Darstellung ist eine genaue Umsetzung der Matrix. (Punkt(0,0) ist im 3D-Feld oben links.) Sämtliche Aktionen, die das gesamte, oder auch nur einen Ausschnitt der Zuckerlandschaft benötigen, greifen hier zu. Mit „getPoints“ erhält man die eigentliche Matrix und mit „getAgents“ die Liste aller Agenten. „clearScape“ entfernt alle Agenten und wird benötigt, falls der Nutzer neue Agenten setzten möchte. „getXlength“ bzw. “getYlength” liefern die Breite bzw. die Länge des Zuckerfeldes. Da viele Veränderungen während eines Lebenslaufes bei dieser Klasse und dementsprechend bei der 3D-Darstellung passieren, reiche ich das „view“-Objekt durch SugarScape weiter, um die Änderungen nicht nur in der Klasse selbst, sondern auch gleich in der Darstellung zu ermöglichen. Des weiteren finden wir hier die eingestellte Wachstumsregel. Diese wird von der „doAgentList“-Methode in SSMain_Window aufgerufen, sobald alle Agenten eine Aktion

gemacht habe. Zu diesem Zeitpunkt wird die Funktion „grow_back“ des Interfaces „grow_back_rule“ aufgerufen, und die Rohstoffe des gesamten Feldes wachsen entsprechend der vorher eingestellten Wachstumsregel nach.

ScapeView ist ein JPanel und beherbergt das Canvas3D-Objekt sowie alle wichtigen Funktionen, welche die Darstellung selbst betreffen. Im Konstruktor werden Canvas3D und einige wichtige Sicht- und Lichteinstellungen geladen. Da das Panel erst erstellt wird, wenn die Konfiguration des Zuckerfeldes bekannt ist, werden auch die 3D-Darstellungen der Felder mittel „drawSquares“ gleich erstellt. Um dies zu tun, wird eine Box erstellt, und deren Position innerhalb der Ansicht mittels zweier TransformGroups bestimmt. Die erste stellt eine Translation dar, die das Objekt in die richtige Relation zu den anderen Zuckerpunkten setzt, und die zweite ist eine Rotation des gesamten Feldes um 45°, so dass der Nutzer einen guten Blick auf die Ereignisse hat. Die dritte TransformGroup ist lediglich für die Darstellung der Verschmutzung wichtig und wird in der Klasse ScapePoint erklärt. Die zweite wichtige Funktion ist „createAgentBranchGroup“. Ähnlich wie bei den Zuckerfeldern wird hier ein „Cylinder“-Objekt, welches einen Agenten darstellt, erzeugt und mittels Transformationen an der richtigen Stelle positioniert. Im Gegensatz zu den Zuckerpunkten benötigen wir hier eine Transformation mehr, da ein „Cylinder“-Objekt „liegend“ generiert wird und so vor allen anderen Bewegungen erst aufgerichtet werden muss (Drehung um 90°). Beachtet werden muss weiterhin, dass Objekte eines kompilierten Graphen nur nachträglich verändert werden können, wenn die sogenannten Capabilities gesetzt werden. Nur dann wird bei der Graphenoptimierung eine nachträgliche Änderbarkeit berücksichtigt.

Das Interface „grow_back_rule“ steht stellvertretend für alle möglichen Wachstumsregeln. Es beinhaltet lediglich die Definition der Funktion „grow_back()“. Sie wird vom Programm aufgerufen, wenn es an der Zeit ist, Zucker nachwachsen zu lassen. Die genaue Funktionalität ist den Implementierungen überlassen.

„Grow_back_rule_1“ ist die Verwirklichung der unter 4.1. beschriebenen Wachstumsregel. Die vom Interface vorgegebene Methode ruft für alle Punkte des Zuckerfeldes „grow_back_field“ auf. Dort wird der Zuckerwert um den Wert „alpha“, welcher dem Konstruktor übergeben wurde erhöht.

„Grow_back_rule_2” ist prinzipiell so aufgebaut wie “Grow_back_rule_1”. Sie Implementiert die Wachstumsregel, welche bei 4.2. erklärt wurde. Der Hauptunterschied zur ersten Regel liegt in der Auswahl der Felder, für die „grow_back_field“ aufgerufen wird. Da hier eine Simulation von Jahreszeiten stattfindet, wird entweder in der oberen oder in der unteren Hälfte der Landschaft kein Zucker nachwachsen. Die Variable „gamma“ bezeichnet die Anzahl der Runden, bevor ein Wechsel der „Jahreszeiten“ stattfindet. Wenn „gamma“ gleich 3 ist, wechselt nach 3 Runden die Jahreszeit. Beim Start ist in der nördlichen Hälfte Sommer und nur dort wächst Zucker nach. „grow_back_field“ ist identisch mit der Funktion aus „Grow_back_rule_1“.

Die dritte Wachstumsregel, implementiert in „Grow_back_rule_3“, beschäftigt sich mit dem Nachwachsen auf Grundlage des Zuckergehaltes der Nachbarfelder. Die grundsätzliche Vorgehensweise bleibt die gleiche. Es wird für alle Felder der Landschaft die „grow_back_field“-Funktion aufgerufen. Der Unterschied besteht hier, dass der Wert, um den der Zucker auf dem Punkt ansteigt, erst berechnet werden muss. Zu diesem Zweck hole ich mir die Zuckerwerte aller 4 angrenzenden Felder, wobei Nachbarn, die nicht existieren, einen Wert von 0 haben. (Dieser Fall tritt an den Rändern und an den Ecken

des Feldes ein.) Von dem Punkt, wo der Zuckervorrat am größten ist, wird die Zuwachsrate errechnet. Und zwar mittels „größter Zuckervorrat der angrenzenden Punkte *alpha = Zucker, der dem Feld zugeführt wird“.

“Grow_back_rule_4” bildet eine Ausnahme, da hier keine neue Wachstumsregel implementiert wird. Das Nachwachsen gestaltet sich wie in „Grow_back_rule_1“. Das besondere an dieser Klasse ist die Zusammenarbeit mit der 6. Agentenregel (siehe 3.6. Verschmutzung). Ohne diese Regel (oder andere, die sich davon ableiten) verhält sie sich wie „Grow_back_rule_1“. Mit der Regel schaltet sich ein Mechanismus ein, der für die Verteilung und den Abbau der Verschmutzung auf dem Feld zuständig ist. Er wird mittels der Funktion

„diffusion“ ausgelöst. Alle Punkte des Feldes werden folgendermaßen bearbeitet: Der Durchschnitt der Verschmutzung des gerade bearbeiteten Feldes und seiner Nachbarn wird berechnet. Das Ergebnis ist der neue Verschmutzungswert. Er wird allerdings erst in der ScapePoint-Variable „tempPol“ zwischengespeichert, bis alle Verschmutzungswerte berechnet wurden. Erst dann wird der „tempPol“ – Wert zum eigentlichen Verschmutzungswert. Dies ist nötig, da ansonsten die Berechnung der nachfolgenden Punkte verfälscht würde, z.B. wenn Punkt (0,0) der Matrix berechnet wird und dabei gleich die neue Verschmutzung annimmt, würde die Berechnung des Punktes (0,1) verfälscht, da der eine Nachbar dieses Punktes einen anderen Verschmutzungswert aufweist, als wenn zuerst Punkt (0,1) und danach Punkt (0,0) berechnet würde. Mit der temporären Zwischenspeicherung ist somit die Reihenfolge der Berechnung nicht mehr wichtig.

“Grow_back_rule_5” beinhaltet zusätzlich das Gewürzwachstum nach Regel 1. Es wird per Zufall entschieden, ob auf dem Feld Gewürz oder Zucker nachwächst.

Das Interface „agent_rule“ ist weitaus größer als das Interface der Wachstumsregeln, da Agenten im Gegensatz zu den Zuckerpunkten weitaus mehr Interaktionen mit anderen Objekten haben. Als erste Definition ist hier die „doAction“ – Funktion zu nennen. Wann immer ein Agent seine Aktion ausführen soll, wird diese Funktion aufgerufen. Des weiteren benötigt jeder Agent ein 3D-Objekt, das ihn repräsentiert. Dieses Objekt ist immer einigen Veränderungen unterworfen. Insbesondere sind hier Bewegungen auf dem Feld sowie das Entfernen vom Feld zu nennen. Das 3D-Objekt muss dem Agenten bekannt sein, so dass jede Änderung gleich im 3D-Bild visualisiert werden kann. Des weiteren müssen andere Programmfunktionen die Möglichkeit haben, die genaue Position des Agenten auf der Zuckerfeldmatrix zu ermitteln. Das wird durch „getPosX“ und „getPosY“ ermöglicht. Die Funktion „isAlive“ soll zurückliefern, ob der Agent im Sinne der aktuellen Regel noch als „lebend“ angesehen werden kann. Tote Agenten werden vom Zuckerfeld und aus der Agentenliste entfernt, und es dürfen keinerlei Aktionen mit ihnen durchgeführt werden. Außerdem sollte es möglich sein, die wichtigen Attribute des Agenten auslesen zu können. Hierunter fallen vor allem der Metabolismus und die Sichtweite, aber auch der derzeitige Zuckervorrat des Agenten sollte abrufbar sein. Da der Zucker auch eine veränderliche Ressource des Agenten ist, ist eine entsprechende „set“-Funktion ebenfalls nötig. Zum Schluss kann es sein, das es während des Lebenslaufes einen Grund geben mag, die farbliche Darstellung des Agenten zu ändern. Dazu dienen die Methoden „getColor“ und „setColor“. Die einzelnen Agenten-Regeln, die ich nun beschreibe, bauen aufeinander auf. Wenn eine Funktion des Interfaces in einer der folgenden Beschreibungen nicht auftaucht, so bedeutet dies nur, dass es nicht nötig war, diese zu überschreiben.

„Agent_rule_1“ implementiert das unter Punkt 3.1. erläuterte Grundmodell der Agentenregel.
Die „doAction“ – Methode gliedert sich in 3 Teilabschnitte. Als erstes sucht der Agent nach einer guten neuen Position (checkVision), als zweites bewegt er sich dorthin (moveToPosition) und zuletzt sammelt und konsumiert er den Zucker der dort liegt (collectSugar). Die Funktion, in der die neue Position ausfindig gemacht wird, war eine der schwierigen Aufgaben. Nicht nur, dass ich ständig aufpassen muss, das der Agent nicht über den „Rand“ des Feldes schaut, sondern es muss auch Sorge dafür getragen werden, was passiert, wenn er bei der Suche auf 2 gleich gute Felder stößt. Das erste Problem löste ich, indem ich dem Agenten die Sicht „über“ den Rand mit einem Hilfsattribut versagte. Das Attribut ist die Differenz zwischen der Sichtweite des Agenten und seiner Position zum Rand. Es wird dann von seiner Sicht in dieser Richtung abgezogen. Die allgemeine Vorgehensweise sieht folgendermaßen aus: 1. Alle Zuckerpunkte innerhalb der Sichtweite in der Breite (X-Richtung) des Feldes werden abgesucht. Dabei werden 2 temporäre Variablen namens „bestX_x“ und „bestY_x“ mit der derzeitig besten Koordinate belegt. Der beste Punkt ist derjenige, auf dem am meisten Zucker liegt und wo kein anderer Agent existiert. Sollte er dabei auf einen Punkt stoßen, wo die Kriterien gleich gut sind, wie bei einem, den er gerade als „besten“ Punkt gespeichert hat, wird derjenige genommen, welcher näher an der derzeitigen Agentenposition liegt. Sollte sie Entfernung ebenfalls gleich sein, so wird per Zufall ermittelt, welchen der beiden möglichen Punkte er behält. Dieselbe Prozedur wird in der Länge (Y-Richtung mit den Variablen „bestX_y“ und „bestY_y“) wiederholt. Am Ende wird dann entschieden welcher der beiden möglichen Punkte besser ist (Grundlage wie üblich der Zuckervorrat und die Entfernung) und so „bestX“ und „bestY“ die Koordinate zum besten Punkt gesetzt.

Die Bewegung des Agenten gliedert sich ebenfalls in 2 Phasen. Als erstes wird der Agent auf dem SugarScape-Objekt verschoben, d.h. seine eigenen Koordinaten werden zu „bestX“ und „bestY“, und der entsprechende SugarScape-Punkt erhält mittels setAgent diesen Agenten. Auf dem alten Punkt wird der Agent entfernt. Die 2. Phase befasst sich mit der Visualisierung dieser Bewegung. Dazu wird der BranchGraph herangezogen und die Translations-TransformGroup der neuen Position angepasst. Die gleiche Prozedur muss natürlich auch für das Textobjekt gelten, in welchem der Zuckervorrat dargestellt wird.

Im 3. Teilabschnitt wird der Zucker auf dem besetzten Punkt dem Agenten überschrieben und der Metabolismus vom Zuckervorrat abgezogen. Es wird geprüft, ob der Agent nicht verhungert ist. Sollte das geschehen sein, wird das Objekt aus der Agentenliste und dem Zuckerfeld genommen sowie der entsprechende BranchGraph entfernt. Sollte der Agent noch leben, wird die Darstellung seines Zuckervorrates aktualisiert.

Die restlichen vom Interface vorgegebenen Methoden sind reine „set“- und „get“-Funktionen, auf die ich hier nicht weiter eingehe.

Die „Agent_rule_2“ befasst sich mit der Möglichkeit der Fortpflanzung (siehe 3.2.). Dazu wird dem Agenten im Konstruktor ein zufälliges Geschlecht zugewiesen. Wie bei den Attributen Metabolismus und Sichtweite ist das Geschlecht im Nachhinein nicht mehr veränderbar. Zunächst werden bei einer Aktion die gleichen Methoden wie bei Agent_rule_1 ausgeführt. Danach jedoch sucht der Agent alle benachbarten Felder mittels der Funktion „notice_Neighbours“ nach anderen Agenten ab. Im darauffolgenden werden die gefundenen Agenten einzeln an die mate-Methode weitergereicht. „mate“ überprüft,

ob alle Kriterien für eine Fortpflanzung vorhanden sind. Dazu gehören: unterschiedliche Geschlechter, ein Fruchtbarkeitskriterium mittels „isFertile“ und ein freier Platz in der Nachbarschaft eines der beiden Beteiligten. Sollten alle Kriterien zutreffen, wird „createChild“ aufgerufen und ein neuer Agent wird auf dem Zuckerfeld platziert.

„isFertile“ prüft, ob die möglichen Eltern alle mindestens einen Zuckervorrat von 10 besitzen, da jedes Elternteil dem Kind 5 Einheiten Nahrung mitgibt. Wenn das Kind erstellt wird, so wird jedes seiner Attribute per Zufall von einem Elternteil übernommen. Danach wird ein freier Platz gesucht und das Kind erstellt, wobei den Erzeugern jeweils 5 Einheiten Zucker genommen werden. Die Methode „addAgent“ übernimmt dabei das Eintragen des neuen Agenten in die Agentenliste sowie den Aufbau des BranchGraph. Diese Funktion muss allerdings von allen abgeleiteten Klassen überschrieben werde, um sicher zu gehen, dass die entstehenden neuen Agenten eine Instanz der richtigen Klasse sind.

In „Agent_rule_3“ werden Räuber eingeführt (siehe 3.3.). Sie starten mit einigen veränderten Werten. Der Sichtbereich ist im Schnitt 5 mal größer und die Startmenge an Zucker ist 3 mal so groß. Die Veränderungen beziehen sich dabei vor allem auf die Suche nach dem besten Zuckerfeld und auf das Aufsammeln des Zuckers. Beide Dinge müssen in zwei verschiedenen Versionen erfolgen, denn ein Räuber sucht sich seine Nahrung anders aus als ein Beute-Agent. Die entsprechenden Funktionen für den Beute-Agenten funktionieren genauso wie in den Elternklassen.

Ein Räuber sucht seinen Ziel nicht, indem er Felder mit Agenten meidet, sondern indem er nur Positionen mit Nicht-Räuber-Agenten sucht. Das Verfahren ist allerdings das gleiche. Das Aufsammeln des Zuckers wird dadurch erreicht, dass der Agent, der vorher auf dem Feld war, entfernt wird und sein Vorrat dem Räuber zugeführt wird. Das Opfer wird in der Funktion „moveToPosition“ entfernt. Die Fortpflanzung erweitert sich um ein Kriterium. Räuber pflanzen sich nicht mit Beute-Agenten fort und vice versa. Die Fruchtbarkeitsbedingung wird für Räuber auf 30 Zuckereinheiten erhöht, da sie ihrem Kind jeweils 15 Einheiten Nahrung überschreiben.

„Agent_rule_4“ ist eine recht kleine Erweiterung der Funktionalität. Agenten erhalten hier ein Attribut „age“, welches das Alter des Agenten anzeigt. (siehe 3.4.). Bei jedem Aufruf von „doAction“ wird das Alter um einen Punkt erhöht. Des weiteren können sich Agenten je nach Geschlecht nur innerhalb eines bestimmten Altersfensters fortpflanzen. Dies wird in “notice_Neighbours“ und „mate_neighbours“ abgefragt. Ab einem bestimmten Alter werden die Agenten entfernt.

In der „Agent_rule_5“ wird Stammesverhalten eingebaut (siehe 3.5.). Die kulturellen Ansichten, welche am Ende einen Agenten einem bestimmten Stamm zuordnen, werden durch ein Boolean-Array der Größe 9 repräsentiert. Der Stamm des Agenten wird aus der Anzahl der „true“-Werte des Arrays ermittelt. Um die Agenten der verschiedenen Stämme auseinander zuhalten, werden sie jeweils mittels der Funktion „resetColor“ anders eingefärbt. Die kulturelle Einstellung wird im Konstruktor durch Zufall festgelegt. Wenn ein Agent sich nach seiner Bewegung in direkter Nachbarschaft mit einem anderen Agenten befindet, dann wird er möglicherweise die Einstellung seines Nachbarn ein wenig ändern können. Dabei wird ein zufällig ausgewähltes Element des Arrays miteinander verglichen. Sollten die Werte unterschiedlich sein, nimmt der nichtaktive Agent dort den anderen Wert an. Der Austausch ist zwischen Räubern und Beute-Agenten nicht möglich. Das wird mittels „transCulture“ erreicht, welche nach allen anderen

Aktionen (Suchen, Bewegen, Essen, Fortpflanzen) durchgeführt wird. Die Funktion der Fortpflanzung („mate“) wurde erweitert, so dass es Agenten unterschiedlicher Stämme nicht möglich ist, sich untereinander fortzupflanzen.

Eine der größten Ausbauten des Programms kam mit „Agent_rule_6“. Mit dieser Regel wurde die Verschmutzung der Zuckerfelder eingeführt (siehe 3.6.). Die Auswahl, nach der Agenten eine neue Position suchen, wird nun durch das Verhältnis Zucker/(1+Verschmutzung) bestimmt. (Die 1 verhindert eine Division durch 0, falls das Feld keinerlei Verschmutzung aufweist.) Je besser dieses Verhältnis, desto attraktiver ist die Position. Deswegen mussten die beiden Such-Funktionen (checkVisionNonPrey und checkVisionPrey) überschrieben werden. Die 2. Veränderung betrifft die Funktionen, in denen der Zucker gesammelt und konsumiert wird. Jede Einheit Zucker, die von diesen Aktionen betroffen ist, produziert ein gewisses Maß an Verschmutzung. Die Berechnung sieht folgendermaßen aus:

„polAmount“ * aufgenommener Zucker + „polAmount“ * verbrauchter Zucker. „polAmount“ ist ein Faktor, der im Optionsfenster bei „beta“ eingestellt werden kann. Die betroffenen Methoden sind „collectSugar“ und „collectSugarNonPrey“. Das größte Problem betrifft die Visualisierung des Verschmutzungslevels. Nach einigen Überlegungen entschied ich mich dafür, das Zuckerfeld in seiner Höhe zu verändern. Dazu wurde eine zusätzliche TransformGroup bei den Box-Objekten eingeführt. Diese TransformGroup wird mittels “ScapePoint.setHeight” verändert. Um die Agenten an das veränderte Höhenlevel anzupassen, konnte die Translations-TransformGroup herangezogen werden. „setAgentHeight“ ist das Pendant zum obengenanten „setHeight“ und sorgt dafür, dass Agenten das Verschieben der Höhe mitmachen.

In der Agentenregel 7 (Agent_rule_7, siehe Punkt 3.7.) wird die Möglichkeit des Kampfes zwischen einzelnen Agenten beleuchtet. Auch hier mussten die Suchfunktionen überschrieben werden, um den Beute-Agenten die Möglichkeit zu geben, Felder mit Agenten eines anderen Stammes als potenzielle Ziele zu erkennen. Es können nur Agenten angegriffen werden, die einen geringeren Zuckervorrat als der Angreifer aufweisen. Es gibt noch 2 andere Kriterien, nach denen ein Angriff erwogen wird. Das Erste wäre die Gefahr einer Vergeltung durch andere Agenten. Das bedeutet, wenn ein Angriff vorgenommen wird, darf kein Gegner innerhalb der Sichtlinie auf der neuen Position sein, der einen größeren Zuckervorrat hat. Die Bedingung wird mit „isVulnerable“ überprüft. (Das Feld zählt also als „verwundbar“, wenn ein solcher Gegner existiert.) Das zweite ist der Gewinn, den ein Agent aus dieser Aktion bekommen kann. Er errechnet sich aus (Zucker + Bonus) /(1+ Verschmutzung). Der Bonus ist die Beute, die aus dem Kampf entsteht. Er ist nötig, da Agenten meistens auf leeren Zuckerfeldern stehen und es ohne ihn kaum zu Kämpfen kommt. Räuber werden keine Stammeskämpfe ausfechten.

Agentenregel 8 (Agent_rule_8) lässt die Agenten erkranken. Jeder Agent wird mit einem Immunsystem erstellt. Es besteht ähnlich wie die Kultur aus einem Boolean-Array. Krankheiten sind ebenfalls so aufgebaut, allerdings ist das Array dort nicht so lang. Der Agent speichert seine Immunität in 2 verschiedene Arrays. ImmunGeno wird nicht verändert, und beinhaltet die Anfangsverteilung der Werte. Nur durch diese Maßnahme kann man später das Immunsystem möglicher Kinder berechnen. Das zweite Array (immun) wird zur eigentlichen Abwehr der Krankheiten genutzt. Eine Krankheit gilt als kuriert, wenn das Array der Krankheit einem Subarray des „immun“-Arrays entspricht. Um Krankheit bei Agenten zu visualisieren, nutzte ich eine transparente Textur. Die

Funktion „IRR“ ruft für alle Krankheiten, welche den Agenten plagen, die Funktion „checkImmun“ auf. Dort wird überprüft, ob die Krankheit einem Subarray des Immunsystems entspricht. Sollte dies nicht der Fall sein, wird beim ähnlichsten Subarray-Teil ein Wert des Immunsystems verändert, in der Hoffnung, dass beim nächsten Mal die Krankheit kuriert werden kann. Mittels „infectNeighbours“ werden benachbarte Agenten mit einer zufälligen Krankheit des Aktiven infiziert. „checkDisease“ überprüft am Ende, ob der Agent noch (bzw. wieder) krank ist und entfernt (bzw. fügt hinzu) die Textur. Die Berechnung des Immunsystems möglicher Kinder ist der Berechnung der Kultur des Kindes ähnlich. Es werden die ImmunGeno-Arrays der Eltern genommen und verglichen.

Agentenregel 9 (Agent_rule_9) bereitete mir einige Schwierigkeiten. Die erste und wichtigste Änderung ergab sich im Sammeln und Konsumieren des Essens. Es musste plötzlich der zweite Rohstoff beachtet werden - sowohl beim Aussuchen der Zielpunkte, als auch beim Konsumieren und Sterben. Beim ersteren half mir allerdings der im Buch benutzte „Welfare“-Wert. Er zeigt an, wie „wohl“ sich ein Agent fühlt, und ist abhängig von den beiden Metabolismuswerten, sowie seinem Zucker- und Gewürzvorrat. Er wird mittels

$$Zucker^{Zuckermetabolismus / Summe\ der\ Metabolismen} * Gewürz^{Gewürzmetabolismus / Summe\ der\ Metabolismen}$$

berechnet. Ein Feld wird dann als „besser“ bezeichnet, wenn er den „Welfare“-Wert des Agenten steigert. Der eigentliche Handel wird nach allen anderen Aktionen des Agenten durchgeführt. Hierzu wird für alle Nachbarn der sogenannte MRS-Wert berechnet.

Er ist das Verhältnis der möglichen Überlebenszeiten pro Rohstoff. Wenn berechnet wird, das der Agent mit seinem derzeitigen Zucker-Vorrat weniger Runden überleben würde, als mit seinem Gewürzvorrat, wird Zucker als wertvoller eingestuft. Des weiteren wird dadurch auch der Tausch-Kurs festgelegt. Er berechnet sich aus der Wurzel des Produktes der beiden MRS-Werte der beteiligten Agenten. Nachdem die Tauschrichtung und der Preis festgestellt wurden, wird überprüft, ob der Handel den „Welfare“-Wert der beiden Agenten steigern würde. Wenn dies zutrifft, wird der Handel durchgeführt. Wenn nicht, findet kein Handel mehr zwischen den beiden Beteiligten in dieser Runde statt. Es wird solange gehandelt, bis dieses Ereignis eintritt. „calculateMRS“ und „calculateWelfare“ sind für die Berechnung der Werte verantwortlich.

Der Handel findet in der Funktion „trade“ statt. Außerdem musste die Fortpflanzung an die Tatsache angepasst werden, dass 2 Rohstoffe zum Überleben nötig sind. Darunter fallen unter anderem die Fruchtbarkeitsbedingungen und ein Metabolismus für Gewürz.

Es existieren noch einige kleine Dialogklassen, die durch die verschiedenen Menupunkte aufgerufen werden. Ihnen allen ist der gleiche Aufbau gemein. Es gibt eine initComponents-Methode, die den Inhalt des Fensters erstellt, sowie verschiedene Actionlistener zum reagieren auf Benutzer-Eingaben. Wichtige Werte erhalten eine get-Funktion, damit sie im SSMain_Window abgefragt werden können. Darunter fallen die Klassen StartDialog, SSOptionsDialog, ErrorDialog und OptionsDialog. Der „ErrorDialog“ wird gezeigt, wenn bei der Erstellung des Zuckerfeldes eine Datei verwendet wurde, die nicht dem geforderten Aufbau entspricht. „StartDialog“ wird beim Menupunkt „start“ aufgerufen. Dort kann man die Anzahl der Runden einstellen. „SSOptionsDialog“ beinhaltet die Einstellungen für die Zuckerlandschaft, während „OptionsDialog“ die Regeln- und Agentenoptionen besitzt.

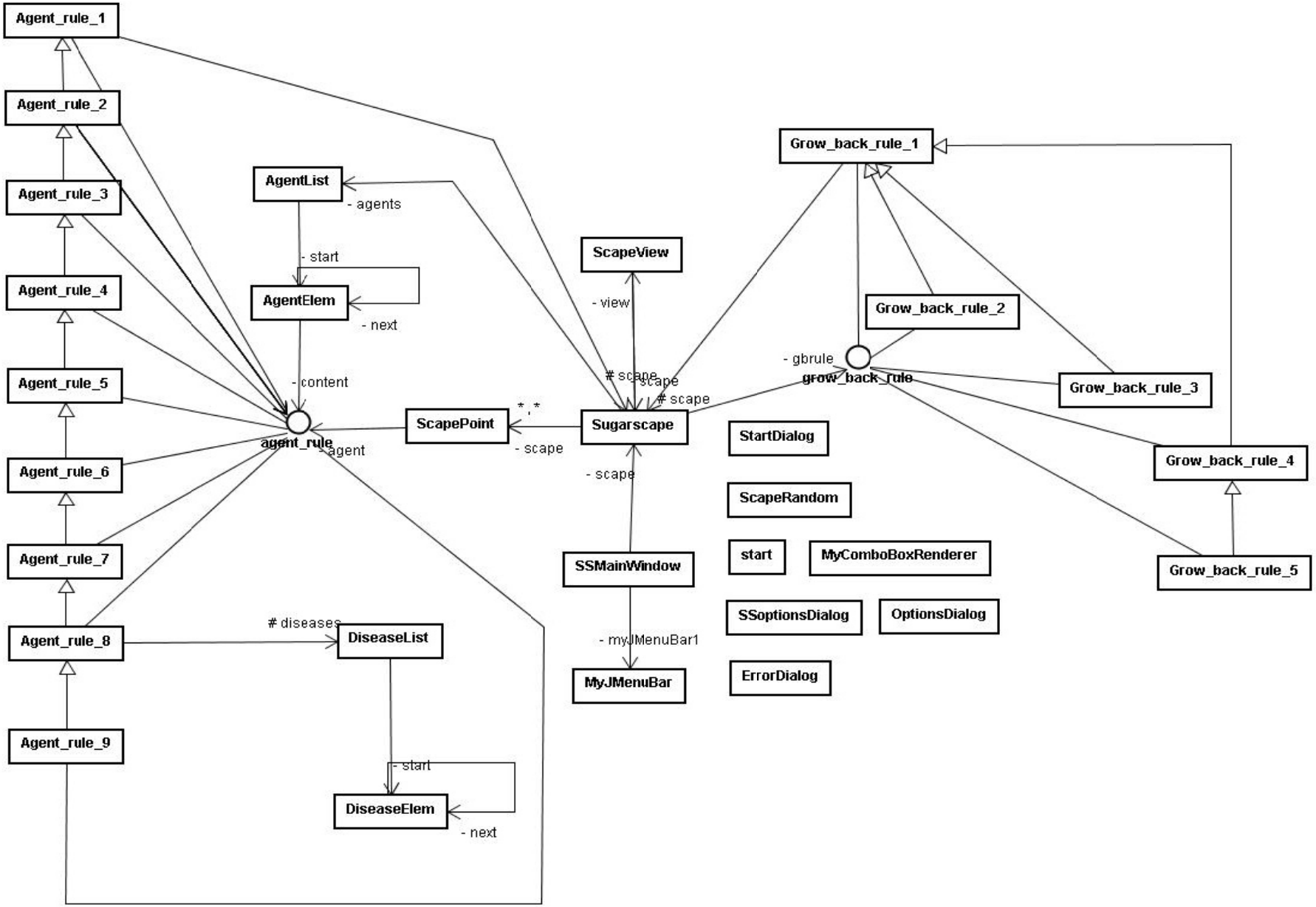
5.3. Besonderheiten und nicht vorhergesehene Probleme

Die in Punkt 5.2 beschriebenen Klassen sind eigentlich die einzigen, die wirklich zur Realisierung nötig sind. Leider gab es während der Implementierung einige Schwierigkeiten und Unstimmigkeiten, welche zur Bildung weiterer Klassen führten. Darauf möchte ich in diesem Punkt näher eingehen. Die erste Überraschung erreichte mich, als ich das Canvas3D-Objekt in das Hauptfenster einfügte. Solange es da war, verdeckte es immer die Menupunkte, welche in die Canvas-Darstellung hineinragten. Einige Recherchen zeigten mir die Ursache des Problems. Canvas3D ist eine sogenannte „heavy-weight-Component“, während Swing (und damit das Menu) „light-weight-Components“ sind. Das bedeutet, das Canvas eine Swing-Komponente immer verdecken wird. Die Lösung liegt darin, die verdeckten Swing-Elemente künstlich zu „heavy-weight-Components“ zu machen. In diesem Programm ist nur die „JMenuBar“ betroffen, und so entstand die Klasse „MyJMenuBar“. Innerhalb des Konstruktors muss lediglich folgende Zeile aufgerufen werden:
„ javax.swing.JPopupMenu.setDefaultLightWeightPopupEnabled(false);“
Damit wird nun das Menu über das Canvas-Objekt gezeichnet.
Das nächste Problem offenbarte sich mir bei der Implementierung der 2. Agentenregel. Ich hatte bis dahin alle Agenten in einem normalen Array gehalten. Wenn einer gestorben ist, so wurde sein Platz lediglich mit „null“ gekennzeichnet. Mit Einführung der Fortpflanzung jedoch ist ein statisches Array nicht mehr möglich. Ich brauchte eine gewisse Dynamik in meiner Agentenliste. 1. Möglichkeit: Je nach Änderung der Agentenzahl ein neues Array zu generieren. 2. Möglichkeit: die in Java implementierten dynamischen Listen nutzen.

Beide Dinge sagten mir nicht zu, vor allem, da die dynamischen Java-Listen auf Array-Objekten aufbauten und ich somit ebenfalls bei Möglichkeit 1 gelandet wäre. Ich erarbeitete mir also eine verkettete Liste, in der die Agenten dynamisch gehalten werden konnten. Diese Liste besteht aus 2 Klassen. „AgentElem“ beinhaltet ein Objekt vom Typ „agent_rule“ und jeweils eine Kette zum Nachfolger in der Liste. „AgentList“ beherbergt sämtliche für eine Liste wichtigen Operationen. Darunter fallen z.B.: Element hinzufügen/entfernen, Elemente suchen, Anzahl der Elemente zurückgeben sowie Liste kopieren/vergleichen. Damit konnte ich dann ohne Probleme fortfahren.

Außerdem basiert dieses Programm auf recht vielen Zufallselementen und zwar so vielen, dass nahezu jede Klasse auf ein Random-Objekt zugreifen musste. Ich schuf eine Extra-Klasse namens ScapeRandom, in der ich ein Random-Objekt lagerte. Es wird einmal instanziiert. Die Funktion, welche den Zugriff auf das Objekt ermöglicht, machte ich static, so dass jede Klasse, welche ein Random benötigt, auf es zugreifen kann.

Da die Beschreibungen in den Combo-Boxen zur Auswahl der verschiedenen Regeln nicht aussagekräftig waren, empfahl mir Prof. Dr. Kurth, diese mit einem Tooltip-Text zu versehen. Leider bietet Java keine bequeme Möglichkeit, einzelne Auswahlen mit einem Tooltip zu versehen. Die einzige von mir gefundene Möglichkeit bestand darin, den „BasicComboBoxRenderer“ der Box zu überschreiben. Dort werden dann die einzelnen Listeneinträge extra mit einem Tooltip versehen. Sie werden als Array von Strings übergeben.
Zum Schluss lagerte ich die Main-Funktion in eine Extra-Datei namens „start“ aus.


Agent_rule_1
Agent_rule_2
Agent_rule_3
Agent_rule_4
Agent_rule_5
Agent_rule_6
Agent_rule_7
Agent_rule_8
Agent_rule_9
AgentList
- agents
- start
AgentElem
- next
- content
agent_rule
- agent
ScapePoint
* , *
- scape
ScapeView
- view
scape
scape
scape
Sugarscape
- scape
SSMainWindow
- myJMenuBar1
MyJMenuBar
diseases
DiseaseList
- start
DiseaseElem
- next
Grow_back_rule_1
Grow_back_rule_2
Grow_back_rule_3
Grow_back_rule_4
Grow_back_rule_5
- gbrule
grow_back_rule
StartDialog
ScapeRandom
start
MyComboBoxRenderer
SSoptionsDialog
OptionsDialog
ErrorDialog

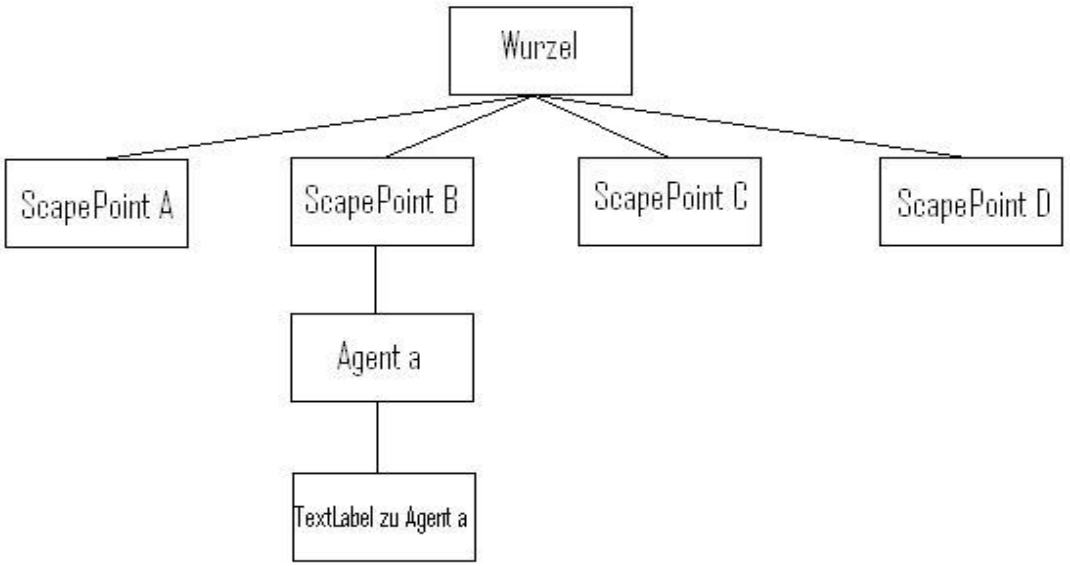
6. die XL-Implementierung

6.1. Was ist XL?

XL ist eine Realisierung von Relationalen Wachstumsgrammatiken (kurz RGG). RGGs haben ihren Ursprung in den „L-Systemen“ und deren Erweiterung der sensitiven Wachstumsgrammatiken. Diese beiden Systeme stoßen bei komplexeren Modellen und Daten an ihre Grenzen. Man muss in diesen Fällen die Aufgabe soweit vereinfachen, dass sie trotzdem dargestellt werden kann. Hier setzten RGGs an. Zusätzlich zu den Regeln der L- oder SGG-Systeme transformiert ein RGG-System Graphen, um komplexe Modelle innerhalb eines regelbasierten Kontextes zu realisieren. „XL“ ist eine Implementation des RGG-Prinzips auf Grundlage der Programmiersprache „Java“. Zur Zeit wird am Lehrstuhl „Graphische Systeme“ an der BTU Cottbus im Rahmen der Dissertation von Ole Kniemeyer eine Plattform namens GroIMP entwickelt, mit der XL interpretiert werden kann.
http://www.grogra.de/

6.2. Allgemeines

Um das Programm in XL zu verstehen, sollte man sich die interne Struktur verdeutlichen. Der Graph für unsere Anwendung besteht aus 3 verschiedenen Knotenarten. Zuerst finden wir den Zuckerfeld-Knoten, genannt „Sugarscape“. Er beinhaltet Attribute für die Menge des Zuckers, welche aktuell auf diesem Feld liegt, sowie die Menge der Verschmutzung und auch für das Gewürz. Die zweite wichtige Art ist der Agentenknoten mit der Bezeichnung „Agent“. Er beinhaltet verschiedene Werte, welche je nach angewendeter Regel genutzt werden. Die Attribute „vision“, „metabolism“, „sugar“ und „worked“ werden generell gebraucht, um die Grundfunktionalität des Agenten zu gewährleisten. Inhalte wie „sex“, „predator“, „age“, „culture“ und „tribe“ werden erst mit den entsprechenden Regeln benutzt. Der letzte Knoten beinhaltet nur ein TextLabel. Er wird dazu genutzt, den aktuellen Zuckervorrat eines Agenten zu visualisieren. Der Graph, den ich mit diesen Knoten aufbaue, ist ein Baum. An der Wurzel des Baumes in der 1. Ebene finden wir die ScapePoints, in der 2. Ebene an den Punkten die Agenten, welche auf dem Punkt stehen, und in der 3. Ebene das TextLabel an den Agenten.



Eine Kante zwischen ScapePoint und Agent besteht nur, wenn der Agent auf genau diesem Punkt steht. Leere ScapePoints besitzen keine anderen Kanten außer zur Wurzel. Auf der anderen Seite hat aber jeder Agent eine Verbindung zu einem TextLabel, mit dem sein Zuckervorrat dargestellt wird.

Auf dieser Struktur basieren die Operationen der XL-Implementierung. Zur Auswertung und Darstellung wird das Programm GroIMP genutzt. Die verschiedenen Einstellungen (Anzahl der Agenten, benutzte Regeln, Größe des Zuckerfeldes etc.) können bei GroIMP mittels des Reiters „Meta Objekts“ und dem darin enthaltenen Klassennamen „Scape“ verändert werden. Dazu genügt es, auf den Klassennamen doppelzuklicken, um den Attributeditor aufzurufen. Wenn Optionen dort verändert wurden, muss das Feld mittels der Reset-Taste neu gestartet werden.

6.3. Beschreibung der Funktionen

Das Programm basiert auf 2 Schritten. Zuerst werden die Objekte initialisiert und in den Graphen eingefügt. Danach wird es möglich, Operationen auf den Graphen auszuführen.

Die Initialisierung wird mit der Funktion „init“ eingeleitet. „GroIMP“ führt sie gleich nach dem Start aus, bzw. immer dann, wenn der Button „Reset“ gedrückt wird. Als erstes werden die Werte der Attribute aus den Optionen Variablen zugeordnet, welche innerhalb des Programms verwendet werden. Dann beginnt der eigentliche Initialisierungsprozess. Es werden die Zuckerpunkte nacheinander nach folgendem Algorithmus erstellt:

1. Erstelle Zuckerpunkt
2. Prüfe, ob noch Agenten platziert werden müssen
3. Berechne die Wahrscheinlichkeit, dass auf diesem Punkt ein Agent steht (noch zu setzende Agenten/ noch zu erstellende Zuckerpunkte)
4. Setzte einen Agenten mit TextLabel, wenn mittels Random die entsprechende Zahl erwürfelt wurde

Damit wäre der Graph vollständig aufgebaut, und der Ausführung des Programms steht nichts mehr im Wege. Dazu wird die „run“-Funktion benötigt. Sie lässt genau einen Agenten eine Aktion ausführen. Eine kontinuierliche Abfolge der Aktionen erreicht man über den Button „Run run“. Der Ablauf an sich sieht folgendermaßen aus:

1. ein Agent führt eine Aktion aus.
2. Wiederhole dies, bis alle Agenten eine Aktion ausgeführt haben
3. Lasse den Zucker nachwachsen

Agenten haben ein Attribut namens „worked“. Wenn er eine Aktion ausgeführt hat, wird es auf true gesetzt. Existiert kein Agent mehr, wo „worked“ false ist, so wird in der „run“-Funktion das Zuckernachwachsen initialisiert. Je nach Wachstumsregel wird eine von 5 Funktionen ausgeführt. Ähnlich geht „run“ bei den Agentenaktionen vor. Je nach eingestellter Regel wird eine andere Funktion aufgerufen.

„runStandard“ repräsentiert die Agentenregel 1 (siehe 3.1.) Sie wählt aus allen Agenten, welche in dieser Runde noch keine Aktion durchgeführt haben, einen per Zufall aus. Für diesen ruft „runStandard“ die Funktion „doAction“ auf. Hierin finden wir wieder das bekannte Muster der Standardagentenaktion. Suchen, Bewegen, Essen! Alle anderen zusätzlichen Funktionalitäten der Regeln werden durch die entsprechenden „run“-Methoden abgedeckt. Das Suchen nach dem geeigneten Punkt wird durch eine sogenannte „bestPoint“-Methode durchgeführt. Es gibt 5 Inkarnationen davon. Jede ist für einen anderen Regel- und Agententyp-Bereich zuständig. Die Methode „bestPoint“ gilt für alle

Nicht-Räuber der Regeln 1-6. Sie sucht einen unbesetztes Zuckerfeld in Sichtweite und gibt dieses Objekt zurück. „bestPointPredator gilt für alle Räuber der Regeln 3-8. Sie sucht einen Agenten in Sichtweite, der den größten Zuckervorrat besitzt. Die Methode „bestPointTribe“ wird für Beute-Agenten der Regeln 7-8 benötigt. Ähnlich wie in „bestPoint“ werden hier Felder mit hohem Zuckergehalt gesucht. Zusätzlich werden gegnerische Agenten als potentielle Zielpunkte behandelt. „bestPointExchange“ und „bestPointPredatorExchange“ sind für die Änderungen durch den zusätzlichen Rohstoff Gewürz nötig und gelten nur für Agentenregel 9.

Es gibt in „doAction“ eine Besonderheit zu beachten. Es ist unter anderem die Aufgabe dieser Funktion, zu prüfen, ob der Agent trotz seiner Bemühungen verhungert ist. In einem solchen Fall soll der Agent sterben. Nun ist die Bewegung des Agenten auf ein anderes Feld und das entfernen desjenigen jeweils eine Operation auf dem Graphen. GroIMP verarbeitet Graphenoperationen jedoch parallel, was in diesem Fall bedeutet, das ich einen Agenten entfernte, den ich parallel mit einem Zuckerpunkt verband. Um diesem Dilemma zu entgehen, wird schon vor der eigentlichen Bewegung des Agenten geprüft, ob er sterben würde. Sollte dies der Fall sein, wird der Agent entfernt, keine Bewegung durchgeführt und der Zuckervorrat (respektive der gegnerische oder der Beute-Agent) des Ziels auf 0 gesetzt (getötet). Der Zucker wird wie gehabt aufgesammelt und verbraucht. Letzteres geschieht auf jeden Fall, wenn der Agent überlebt. Er wird zusätzlich noch auf das abgeerntete Fels bewegt.

Die Funktionen „runReproduction“, „runPredator“, „runAge“, „runCulture“, „runPollution“, „runTribe“, „runDisease“ und „runExchange“ bilden die Inkarnationen der eingestellten Agentenregeln. Ihre Funktionsweise ist ähnlich aufgebaut wie in „runStandard“. Zuerst wird ein zufälliger Agent gewählt und die für diese Regel nötige “doAktion“-Methode aufgerufen. Danach werden im Normalfall weitere Anweisungen für den Agenten abgearbeitet. „runReproduction“ prüft nach dieser Grundaktion, ob in der Nachbarschaft ein Agent steht, mit dem eine Vermehrung stattfinden kann. In der „mate“-Methode wird sie dann vollzogen. Dabei wird zuerst eine freie Position neben einem der beteiligten Agenten und die Werte des Kindes durch die der Eltern durch Zufall bestimmt, der Agent erstellt und zum Schluss mittels „addAgentToPointInit“ in den Graphen eingefügt.

In „runPredator“ wird eine Fallunterscheidung vorgenommen. Diese bezieht sich aber nur auf die unterschiedlichen „doAction“- Methoden für Räuber und Beute-Agenten.

„runAge“ verhält sich wie „runPredator“ und ergänzt die Aktion durch die Inkrementierung des Alters und die Überprüfung, ob der Agent dadurch sterben sollte.

Die Regel 5, welche durch „runTribe“ realisiert wird, fügt dem ganzen mittels „transCulture“ den möglichen Austausch von Kultur und den dadurch bedingten Stammeswechsel hinzu. Hierbei werden in der Nachbarschaft Agenten gesucht und mit allen Gefundenen ein per Zufall ausgesuchter Bestandteil des Kulturarrays dem des aktiven Agenten angepasst.

„runPollution“ bildet eine kleine Ausnahme zu dem oben beschriebenen Schema. Das Erzeugen von Verschmutzung und die graphische Veränderung die zum Anzeigen nötig ist, würden es erfordern, nahezu alle grundlegenden Funktionen für diese Regel neu zu schreiben. Stattdessen ging ich einen anderen Weg. Die Verschmutzung und die Visualisierung werden in der Methode „moveAgent“ durch eine Abfrage mittels der

globalen Variablen „staticRule“, welche die ausgewählte Agentenregel anzeigt, durchgeführt. Allerdings nur, wenn „staticRule“ die richtige Regel anzeigt. Die Auswirkungen der Verschmutzung auf die Wahl des Agenten für einen geeigneten Zielpunkt wurden direkt in die „doAction“-Methoden integriert. Hier allerdings ohne Abfrage, da das Kriterium sowohl mit als auch ohne Verschmutzung mit der Formel (Zucker des Punktes/(1+Verschmutzung)) abgedeckt ist. (Mit Verschmutzung gleich 0 wird nur der reine Zuckervorrat betrachtet und ohne Regel 6 oder höher wird keine Verschmutzung auf die Punkte gelegt.)

„runTribe“ beinhaltet Regel 7 und integriert den Kampf zwischen Nicht-Räubern. Um das zu erreichen, musste eine neue „doAction“-Methode erstellt werden, die Methode „doActionTribe“. Sie unterscheidet sich nur in 2 Punkten von „doAction“. Erstens wird mit „bestPointTribe“ eine Suchroutine aufgerufen, welche auch Zuckerfelder, die von gegnerischen Agenten besetzt, sind als Zielpunkte zurückliefern kann, und zweitens, dass auf besagtem Feld der besiegte Agent entfernt wird. Es existiert eine „Prämie“ von 10 Zuckereinheiten für einen Sieg, um den Kampf etwas attraktiver zu gestalten.

„runDisease“ ruft „runTribe“ auf und versucht mittels „IRR“ und „checkImmun“ Krankheiten vom Agenten zu entfernen. „infectNeighbours“ infiziert Agenten auf Nachbarfeldern, und mit „checkDisease“ wird überprüft, ob der Agent mittels Textur als krank gekennzeichnet werden muss.

Die letzte Agentenregel ist „runExchange“. Die Funktionen „trade“, „calculateWelfare“ und „calculateMRS“ konnten nahezu vollständig aus dem Java-Programm entnommen werden.

Die zweite Riege der „run“-Funktionen beinhaltet die Wachstumsregeln.
„run1“ sorgt dafür, dass auf allen Zuckerfeldern der Wert Alpha an Zucker nachwächst. Dazu wird für alle Felder die Funktion „growthNormal“ in welcher der Zucker um die Zahl „alpha“ nachwächst, aufgerufen. Danach wird das „worked“-Attribut aller Agenten auf false gesetzt, so dass eine neue Runde anlaufen kann.

Die Funktionen „run2“ und „run3“ laufen nach dem gleichen Schema ab und rufen jeweils „growthSummerWinter“ und „growthPoint“ auf, in denen jeweils die Wachstumsregeln 2 und 3 realisiert sind.

Bei „growthSummerWinter“ wächst beim aufgerufenen Feld nur Zucker nach, wenn er je nach Saison in der richtigen Hälfte der Landschaft liegt.

An „run4“ ist insofern besonders, dass hier keine neue Wachstumsregel eingeführt wird, sondern die Regel 1 mit der Verteilung und dem Abbau von Verschmutzung verbunden wird. Es wird empfohlen, sie nur in Verbindung mit den Agentenregeln 6,7 und 8 zu verwenden. Für alle anderen Regeln verhält sie sich wie Wachstumsregel 1. Um diese Verteilung zu berechnen, existiert die Funktion „diffusion“. Sie berechnet den Durchschnitt der Verschmutzung für das Feld selbst und alle seine Nachbarn. Dies ist der neue Verschmutzungswert für ihn.

„run5“ verhält sich wie „run4“, allerdings wird per Zufall bestimmt, ob Zucker oder Gewürz nachwächst.

Ich habe für dieses Programm einige Hilfsfunktionen definiert, welche im allgemeinen

reine Graphoperationen darstellen. Zum einen bot mir dies eine größere Übersicht über das Programm, zum anderen konnte ich so besser die einzelnen Befehle lernen. „findPoint“ liefert den Zuckerpunkt, auf dem der Agent steht, der übergeben wurde. „getAgent“ ist das genaue Gegenteil davon. Die Funktion „sterben“ entfernt den übergebenen Agenten aus dem Graphen. „isAgent“ prüft, ob ein Agent auf diesem Zuckerpunkt steht, und „getText“ gibt das TextLabel zurück, wo jeder Agent eine Verknüpfung mit einem solchen besitzt. „vonNeumann“ liefert ein Array aller direkten Nachbarn des Agenten zurück.

6.4. Anmerkungen

Auch wenn XL auf Java basiert und es ohne Probleme möglich ist, Java-Klassen einzubinden, sollte man sich immer vor Augen halten, dass wir es nicht mit einer Sprache wie Java oder C++ zu tun haben. XL basiert auf weitaus grundlegenderen Prinzipien, und es ist möglich, Strukturen nahezu 1:1 in XL umzusetzen, anstatt sie eher abstrakt in einen Algorithmus zu interpretieren. Als ich mit diesem Teil der Arbeit anfing, beging ich den Fehler, es als eine andere Art von Java mit einer anderen Art der Interpretation zu sehen.

Deshalb möchte ich diesen Teil dazu verwenden, um anderen Einsteigern aus der Java- oder C++ -Linie eine kleine Hilfe zu geben. Als erstes solltet Ihr euch darüber klar werden, das Ihr einen Graphen aufbaut und auf diesem Konstrukt auch arbeitet. XL bietet viele Operatoren an, um Knoten und Kanten des Graphen zu finden und zu manipulieren. Unterteilt eure Arbeit am besten in 3 Schritte. Zuerst solltet Ihr den grundlegenden Graphen aufbauen. Danach überlegt Euch, welche Veränderungen Euer Graph durchlaufen soll, und erstellt Funktionen, die genau eine solche Aktion durchführen. Danach könnt Ihr ein Programm schreiben, was nahezu reines Java ist. Es ist dabei zu beachten, dass mehrere Graphenoperationen parallel abgearbeitet werden.

Zurzeit ist es leider noch nicht möglich, XL-Dateien zu importieren. Der Import von Java-Dateien in eine XL-Datei ist ohne Schwierigkeiten möglich, aber um größere Programme besser zu strukturieren, wäre es besser, XL-Dateien importieren zu können, oder eine Möglichkeit zu schaffen auf Klassen und Module in anderen XL-Dateien zuzugreifen.

Bedingt durch den Zwang, nur eine Datei zu nutzen, finde ich das Programm im Moment zu unübersichtlich, wenn es derartige Ausmaße annimmt.

Bei meinem XL-Programm verwende ich 2 Java-Dateien, um die Krankheiten zu verwalten. Diese werden durch ein Jar-Archiv (dlist.jar) in das Programm integriert. In GroIMP muss diese Datei in das „ext“-Verzeichnis kopiert werden.

7. Der Vergleich der Implementierungen

Der eigentliche Zweck der Programmierung dieses Themas ist der Vergleich zwischen XL unter GroIMP und der direkten Java-Implementierung. Es gibt verschiedene Dinge, die man vergleichen könnte, aber ich habe mich hier für 4 verschiedene entschieden. Das erste Kriterium sollen die reinen LOC-Anzahlen sein, ohne Kommentare oder Leerzeilen. Das zweite ist die Speicherauslastung über einen Zeitraum von 10 min, das dritte die Prozessorauslastung für denselben Zeitraum. Jeweils für alle Agenten- und Wachstumsregeln.
Zum Schluss soll noch ein kleiner Laufzeitvergleich stattfinden.

7.1 LOC-Vergleich

Um die LOC-Zahl des Java-Programms zu ermitteln, benutzte ich, bedingt durch die Menge der Dateien und Kommentaren, ein Tool zum automatischen Zählen. Es ist die Shareware-version von „Code Counter Pro“ der Firma „GeroneSoft“ Version 1.32. Das Ergebnis sieht folgendermaßen aus.

Dateiname	Quellcode	Kommentare	Beides	Leerzeilen	Total:
SSMainWindow.java	456	123	14	46	639
SSoptionsDialog.java	194	97	11	33	335
start.java	8	10	0	3	21
StartDialog.java	77	38	5	19	139
Sugarscape.java	51	70	0	15	136
agent_rule.java	19	65	0	16	100
Agent_rule_1.java	206	182	12	30	430
Agent_rule_2.java	156	108	0	23	287
Agent_rule_3.java	460	215	22	40	737
Agent_rule_4.java	57	80	0	15	152
Agent_rule_5.java	223	153	0	17	393
Agent_rule_6.java	304	182	22	23	531
Agent_rule_7.java	225	148	16	21	410
Agent_rule_8.java	234	188	3	15	440
Agent_rule_9.java	665	300	22	34	1021
AgentElem.java	18	34	0	4	56
AgentList.java	79	58	0	12	149
DiseaseElem.java	20	27	0	7	54
DiseaseList.java	90	63	0	13	166
ErrorDialog.java	48	17	3	9	77
grow_back_rule.java	3	8	0	1	12
Grow_back_rule_1.java	19	27	0	3	49
Grow_back_rule_2.java	33	37	0	6	76
Grow_back_rule_3.java	50	37	0	5	92
Grow_back_rule_4.java	49	41	0	3	93
Grow_back_rule_5.java	24	28	0	4	56
MyComboBoxRenderer.java	25	15	0	3	43
MyJMenuBar.java	7	16	2	6	31
OptionsDialog.java	288	90	9	44	431
ScapePoint.java	98	122	0	26	246
ScapeRandom.java	10	18	0	3	31
ScapeView.java	211	57	2	24	294
Total	4407	2654	143	523	7727

Die Spalte “Beides” bezeichnet Zeilen, die sowohl Source-Code, als auch Kommentare enthalten. Damit errechnet sich eine Gesamt-LOC von 4550. Wenn wir die verschiedenen Workarounds (MyJMenuBar.java, AgentElem.java, AgentList.java, MyComboBoxRenderer.java, DiseaseElem.java, DiseaseList.java) nicht mitzählen, bleiben immer noch 4309 LOC. Da in XL einzig die Funktionalität und nicht die 3D-Darstellung implementiert wird, müsste der Vergleich der Programme ebenfalls nur solche Java-Klassen beinhalten. Das bedeutet eine Zählung der Interfaces „agent_rule“ und „grow_back_rule“ sowie ihrer Implementationen, die Klassen „ScapeRandom“, „Scape_Point“, „SugarScape“, „SSMain_Window“ und „start“.
Diese ergeben eine LOC-Zahl von 3461.
Da es noch keinen auf XL zugeschnittenen LOC-Zähler gibt, musste ich es hier selbst tun. Ich kam auf eine LOC-Zahl von 1795, wobei man noch ergänzen muss, dass einige

Befehle so lang waren, das ich sie auf mehrere Zeilen verteilt habe. Probehalber habe ich die XL-Datei als Java-Datei maskiert und durch das Programm zählen lassen, da durch den Aufbau der Sprache so möglicherweise auch ein brauchbares Ergebnis gewonnen werden kann. Folgende Zahlen ergaben sich daraus:

Dateiname	Quellcode	Kommentare	Beides	Leerzeilen	Total:
Scape	1781	841	16	78	2716
Total	1781	841	16	78	2716

Zu den 1797 LOC muss ich bei beiden Varianten die LOC der Klassen „DiseaseList“ und „DiseaseElem“ hinzuaddieren, da ich sie für das XL-Programm benutze. Wir haben also 1905 (1907) LOC für XL.

Je nach Betrachtungsweise haben wir also ein Verhältnis von 4550 : 1905 (etwa 2,39 : 1) oder 3461 : 1905 (etwa 1,81 : 1)

Dabei muss noch ergänzt werden, dass ich durch meine Unerfahrenheit in dieser neuen Sprache sicherlich einiges mittels Java realisierte, obwohl es eventuell eine kleinere XL-Variante gegeben hätte. Dies fiel mir insbesondere in den höheren Agentenregeln auf (Agentenregel 7,8,9). Dort kann man sehen, das die Funktionalität der Java-Programmierung nahezu vollständig mit der XL-Implementierung übereinstimmt. Je komplexer der zu erschaffende Mechanismus ist, desto mehr griff ich auf reine Java-Programmierung zurück.

7.2 Speicher- und Prozessorleistungsvergleich

Um die Speicher- und Prozessorauslastung der Programme zu messen, verwendete ich den „Performance Monitor“ von Microsoft. Ich ließ jedes der beiden Programme mit nahezu jeder Kombination von Wachstums- und Agentenregeln jeweils 10 Minuten lang arbeiten. Alle 30 Sekunden wurden die Prozessorauslastung und Speicherbelegung gemessen und aufgezeichnet. Berücksichtigt wurden alle Kombinationen der Agentenregeln 1-5 mit den Wachstumsregeln 1-3. Eine Ausnahme bilden hierbei die Agentenregeln 6, 7, 8 und 9, welche nur mit den Wachstumsregel 4 und 5 zusammen ein vollständiges System ergeben. Jedes der beiden Programme hat eine erzwungene halbsekündige Pause zwischen den Agentenaktionen. Dies geschah ursprünglich, um dem Nutzer die Möglichkeit zu geben, die Veränderungen visuell zu erfassen. In Anbetracht der Messung gibt uns diese Pause überhaupt erst die Option, Prozessorauslastung zu erfassen. Ohne sie würden wir nur die gesamte Messzeit über 100% erfassen.

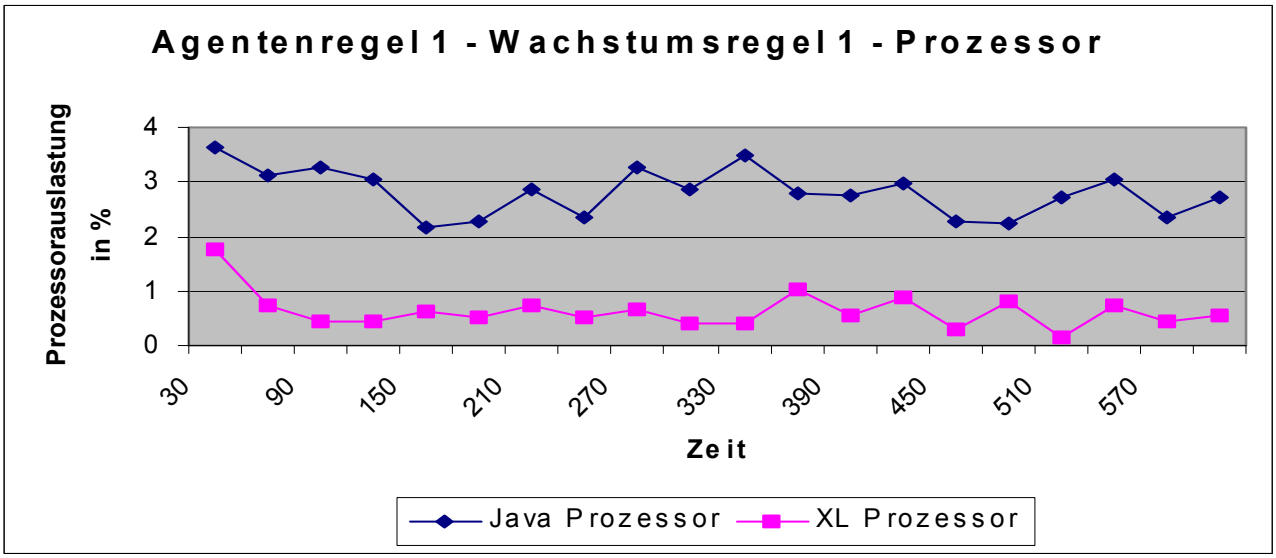
Das eigentliche Messverfahren mittels des Performance Monitors gestaltete sich denkbar einfach. Man erstellt ein sogenanntes Leistungsindikatorprotokoll. In diesem werden das Dateiformat für die Ergebnisse, das Messintervall sowie Anfangszeit und Laufzeit bestimmt. Die zu messende Größen waren hierbei die Prozessorauslastung und der noch verbleibende freie Speicher des Systems. Es liefen nur Windows XP, meine Firewall und das Antivierenprogramm neben dem Messobjekt und dem Performance Monitor. Ich führte auch einen Durchlauf ohne Java/XL-Programm durch, um die normale Belastung des Systems zu sehen. Die gewonnenen Daten nutze ich, um durch die Differenz die von Java/XL genutzten Ressourcen zu errechnen. Die Hauptschwierigkeit der Messungen lag vor allem in der Tatsache, dass nicht jeder Versuch zu einem auswertbaren Ergebnis führte.

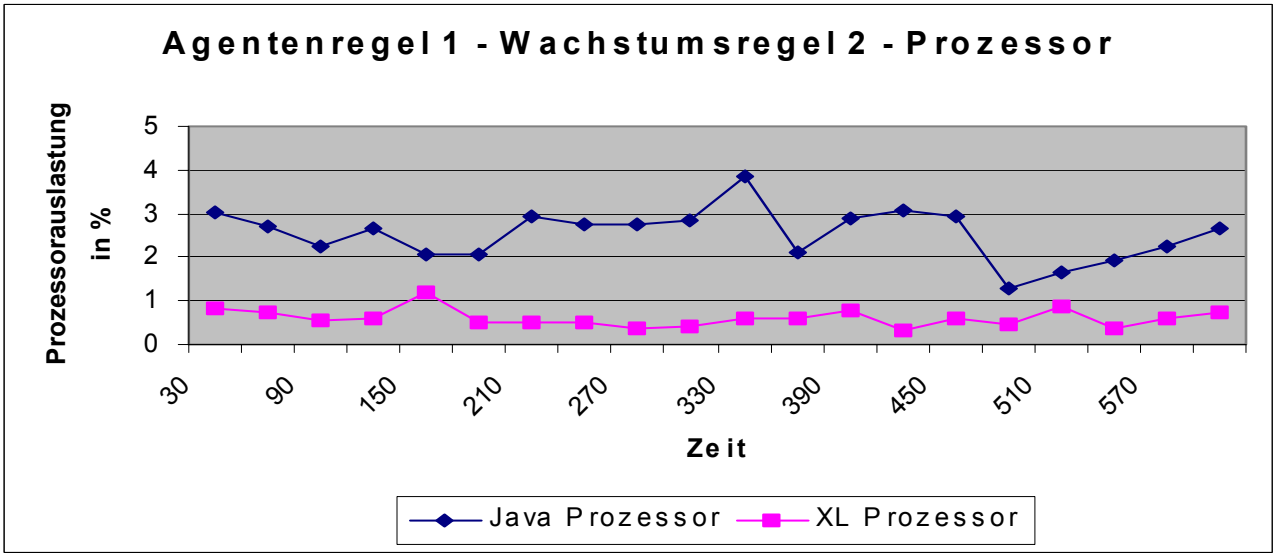
In den höheren Agentenregeln gibt es immer öfter Möglichkeiten, wie Agenten vom Sugarscape verschwinden konnten. Diese hohe Sterberate sorgte nicht selten dafür, das nach einer gewissen Zeit keine Agenten mehr lebten und so keine nennenswerten Daten gemessen werden konnten.

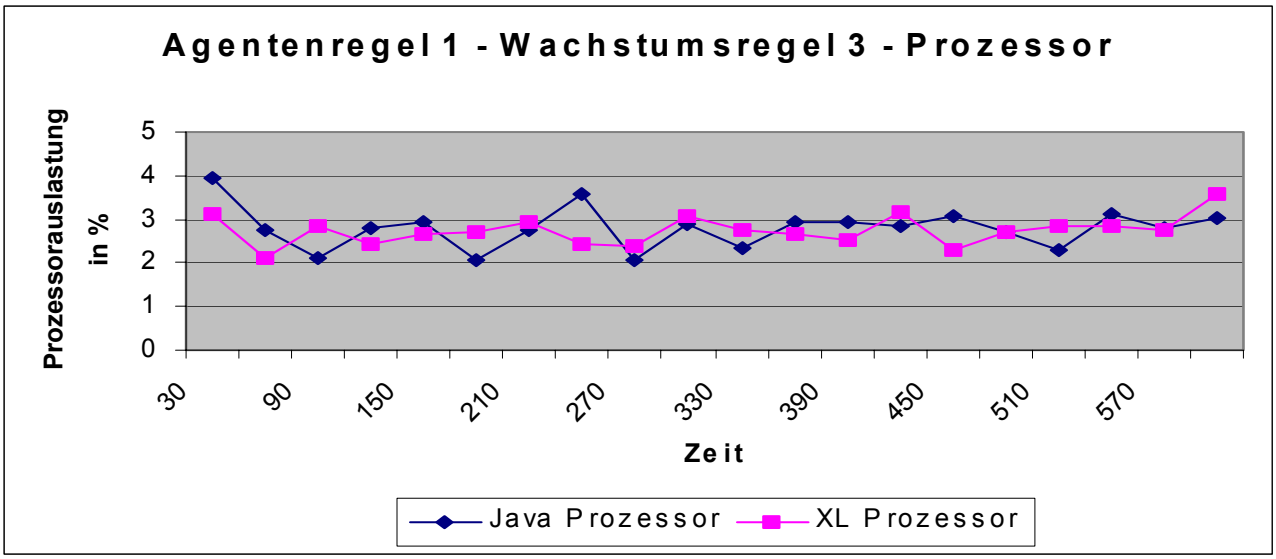
Bei den Prozessorwerten musste ich jedes Mal den ersten Wert entfernen, da dort der Performance Monitor die Last in die Höhe trieb.

Insgesamt habe ich so 19 erfolgreiche Messungen pro Programm vorgenommen. Ich werde einige Interessante auswählen und auswerten.

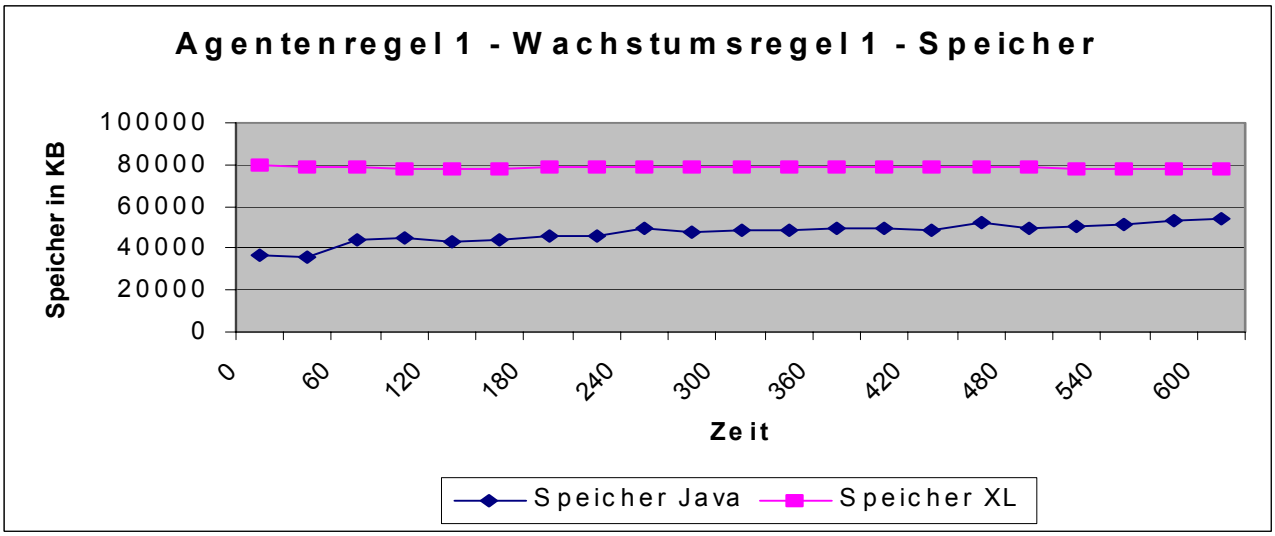
Als erstes mache ich einen Vergleich der 3 Wachstumsregeln. Ich benutze hierzu die Agentenregel 1, da sie von allen die wenigsten Interaktionen aufweist.

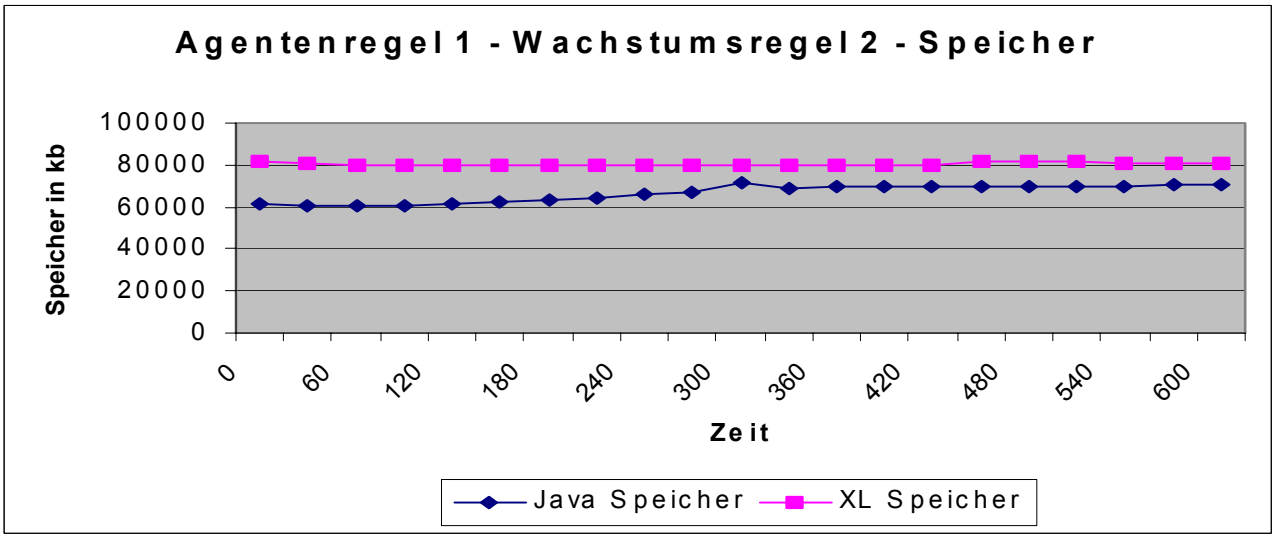


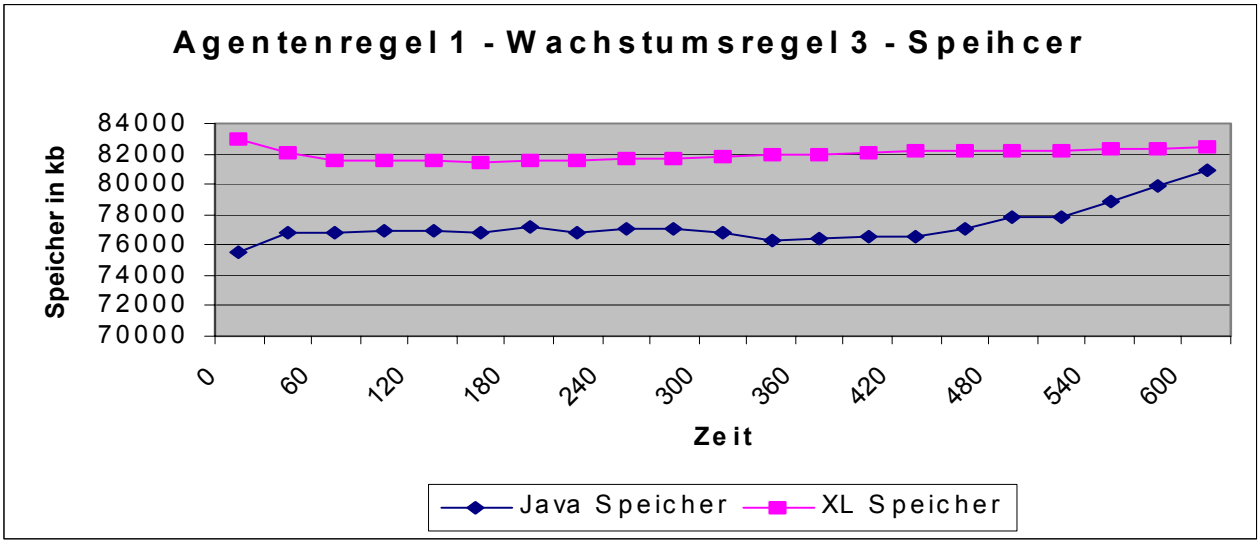




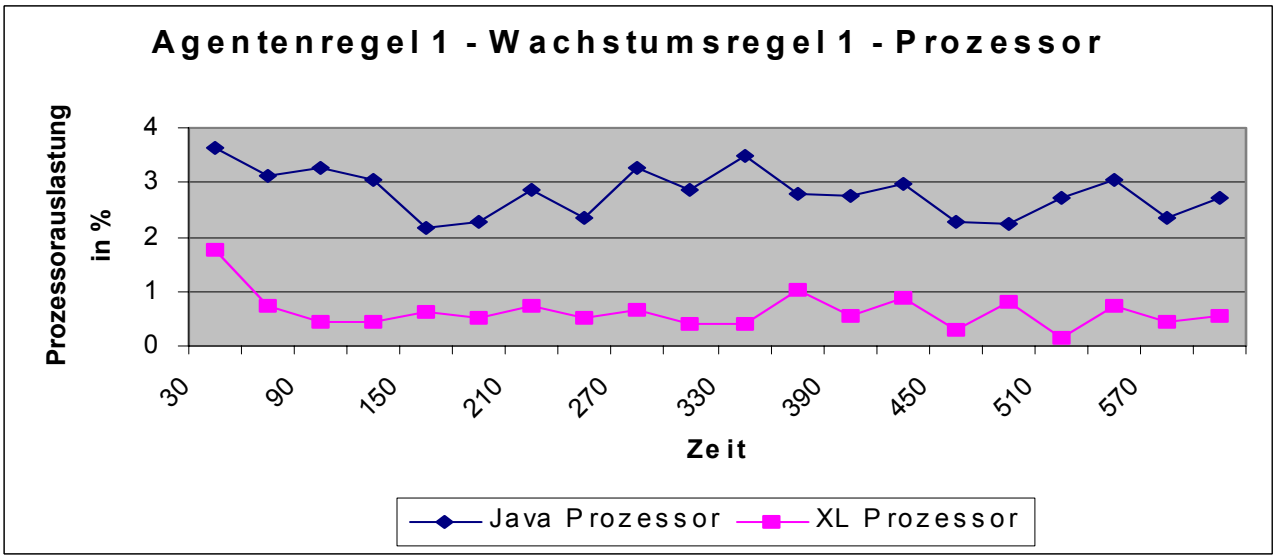
Bis auf die 3. Wachstumsregel haben wir hier den Trend, das XL immer weniger Prozessorlast hat als die Java-Version. Die erhöhte Auslastung im 3. Fall ist vermutlich auf die zusätzliche Graphensuche zurückzuführen.

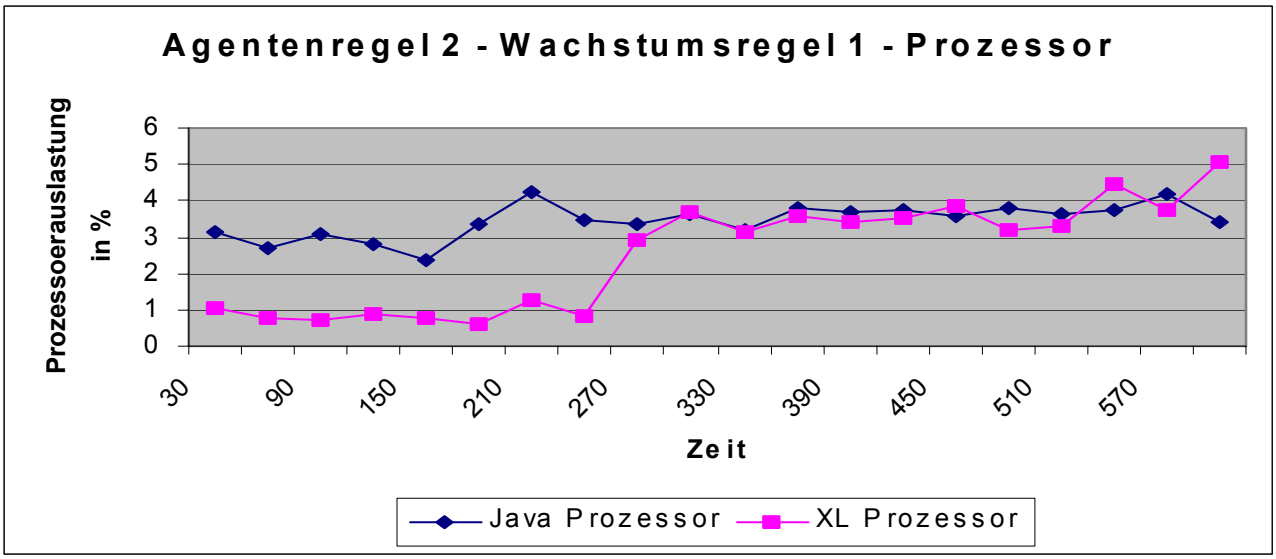


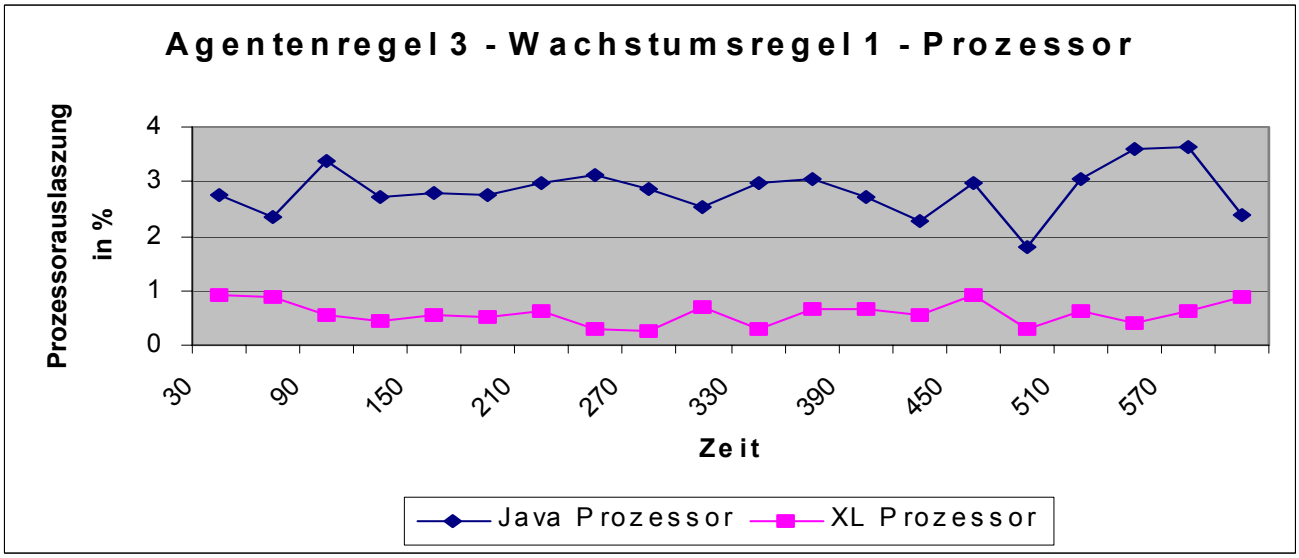


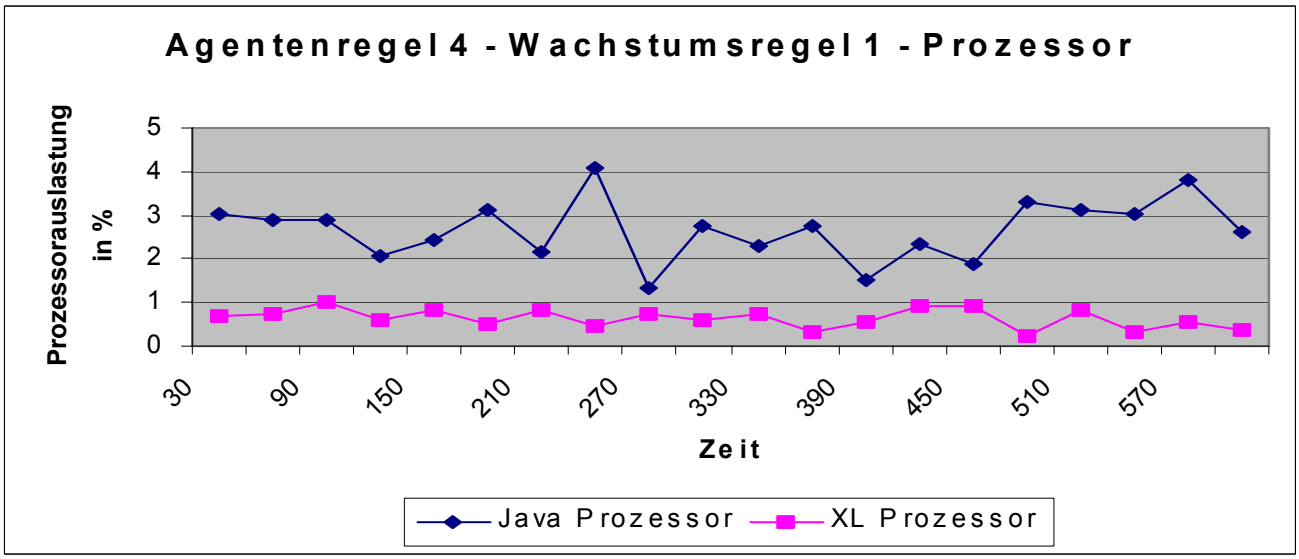


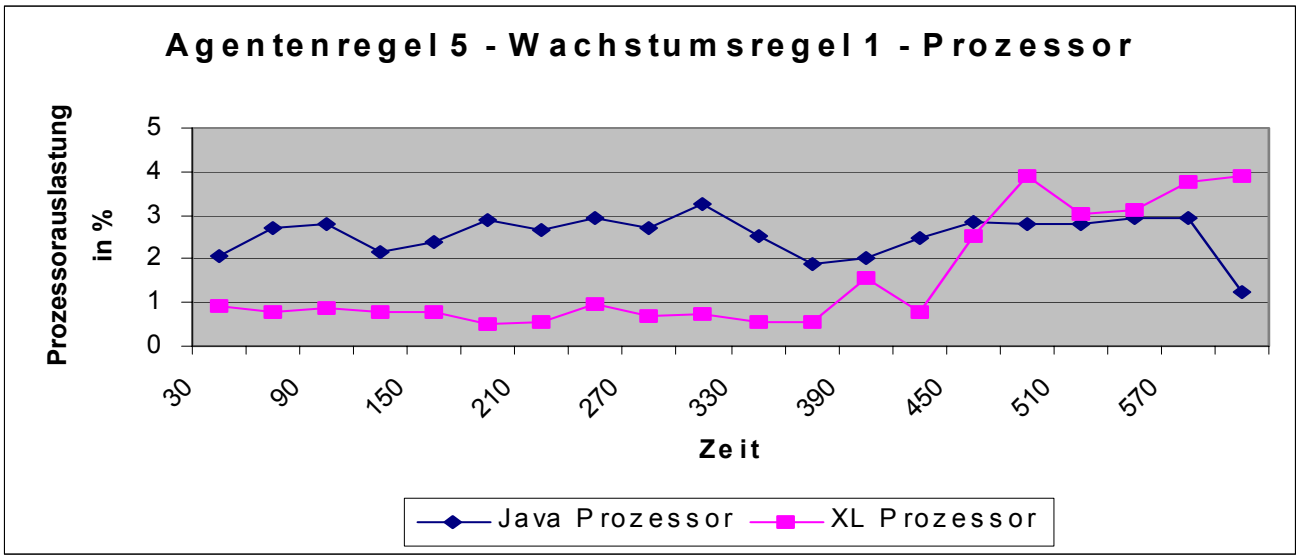
Die Speicherauslastung zeigt uns einen hohen, aber stabilen Wert für GroIMP/XL. Java dagegen verzeichnet einen geringeren Speicherverbrauch, ist aber nicht so stabil.
Als nächstes betrachten wir die Agentenregeln 1-5 unter der Wachstumsregel 1.




Agentenregel 2 - Wachstumsregel 1 - Prozessor
Prozessoerauslastung in %
0
1
2
3
4
5
6
30
90
150
210
270
330
390
450
510
570
Zeit
Java Prozessor
XL Prozessor

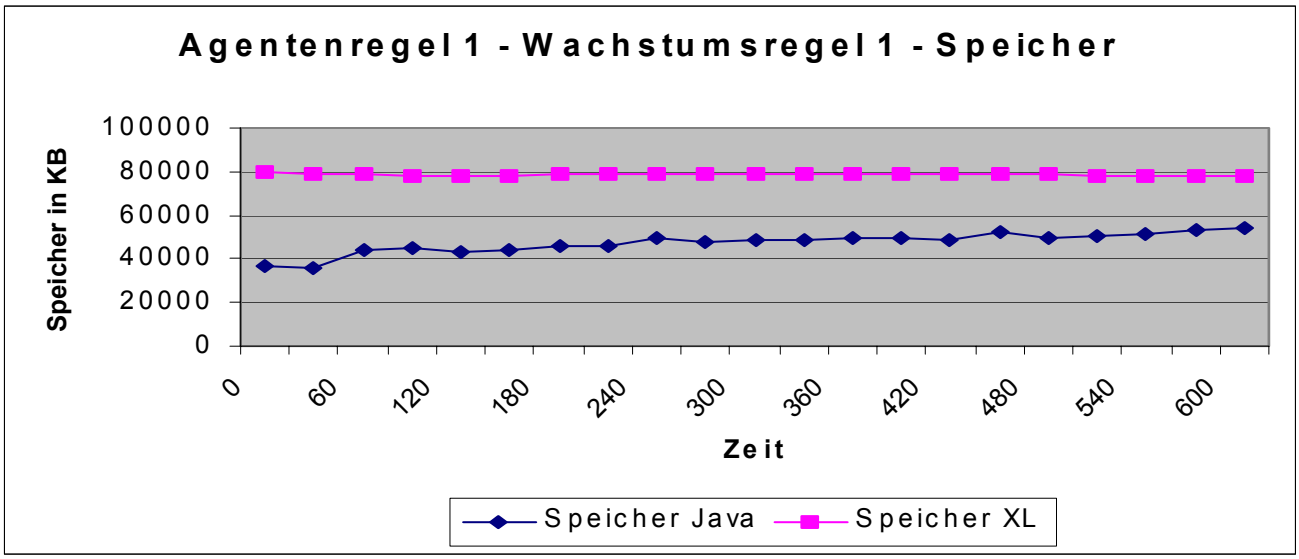

Agentenregel 3 - Wachstumsregel 1 - Prozessor
Prozessorauslaszung in %
0
1
2
3
4
30
90
150
210
270
330
390
450
510
570
Zeit
Java Prozessor
XL Prozessor

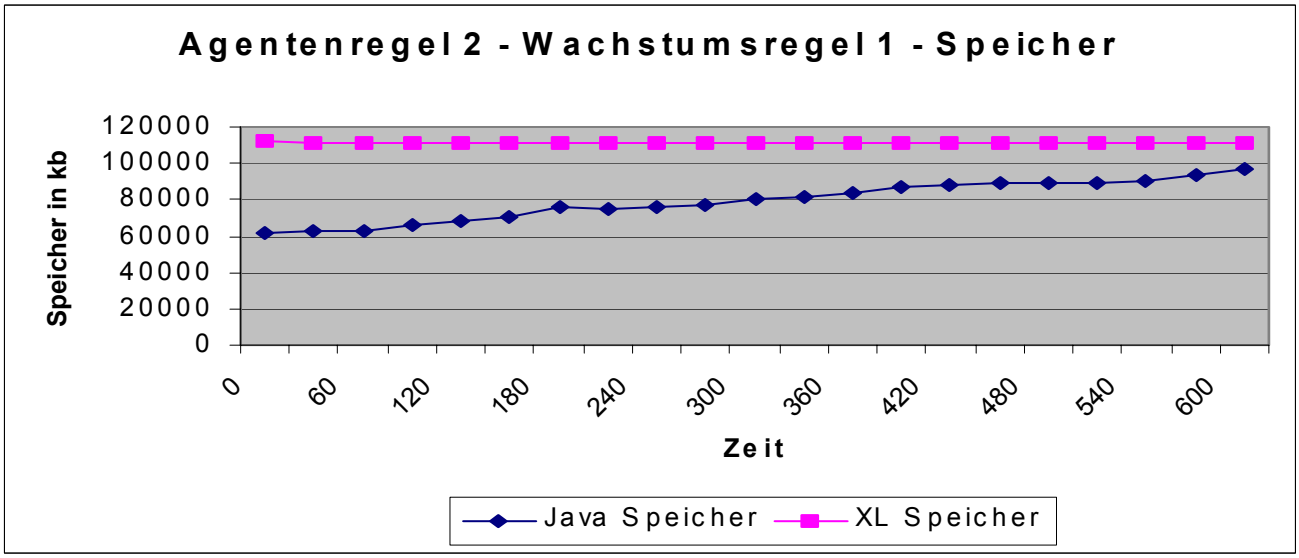

Agentenregel 4 - Wachstumsregel 1 - Prozessor
Prozessorauslastung in %
0
1
2
3
4
5
30
90
150
210
270
330
390
450
510
570
Zeit
Java Prozessor
XL Prozessor

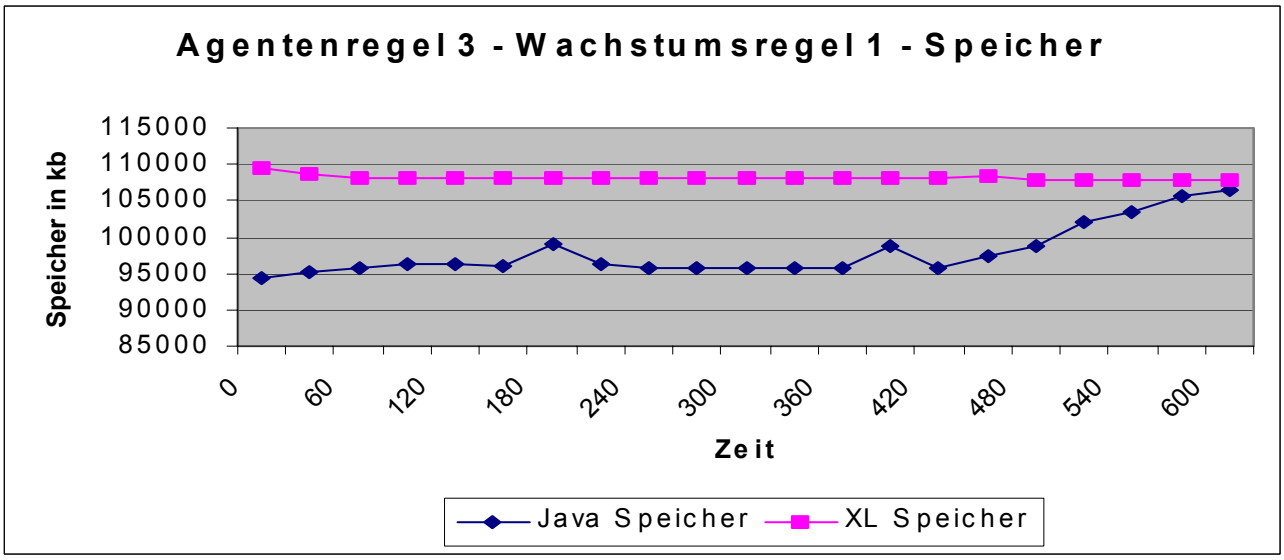


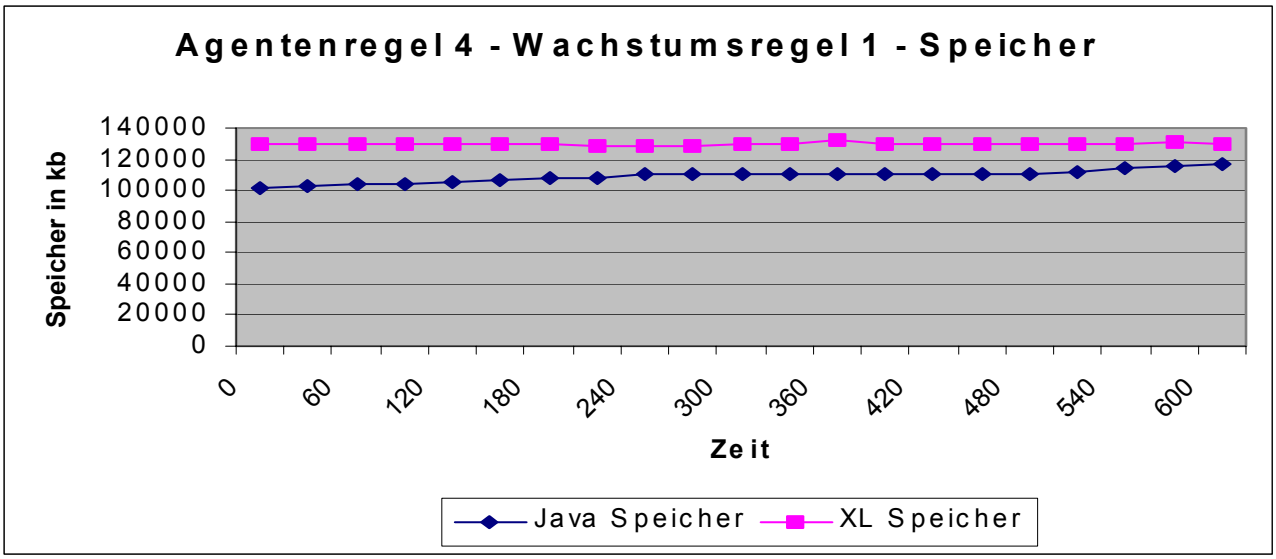
Mit immer höheren Agentenregeln erhöht sich auch die Fluktuation der Agenten auf dem Sugarscape. In Regel 1 bleibt die Anzahl bis auf wenige Ausnahmen konstant.
Mit Regel 2 erleben wir ab einem bestimmten Zeitpunkt eine explosionsartige Vermehrung. Durch die Räuber in Regel 3 gibt es dagegen eine zusätzliche Möglichkeit zu sterben. Die Regeln 4 und 5 zögern die Vermehrung der Agenten durch das Mindestalter und durch die geringere Wahrscheinlichkeit einen Partner zu finden hinaus. Oft passiert es gerade hier, dass alle Agenten vor Messende sterben. Der relativ gleichmäßige Verlauf in Regel 1 wird in Regel 2 unterbrochen. Die erhöhte Agentenpopulation sorgt in XL für einen starken Mehraufwand der Berechnungen. Je mehr Agenten wir haben, desto öfter muss GroIMP den gesamten Graphen durchsuchen, während bei Java nur bestimmte Punkte einer Matrix abgefragt werden. In Regel 3 hat sich die Kombination von Fortpflanzung und Sterben durch Räuber so gut die Waage gehalten, dass wir ein Ergebnis ähnlich der Regel 1 bekommen konnten. (Dies kann aber auch in die Extreme abweichen.)

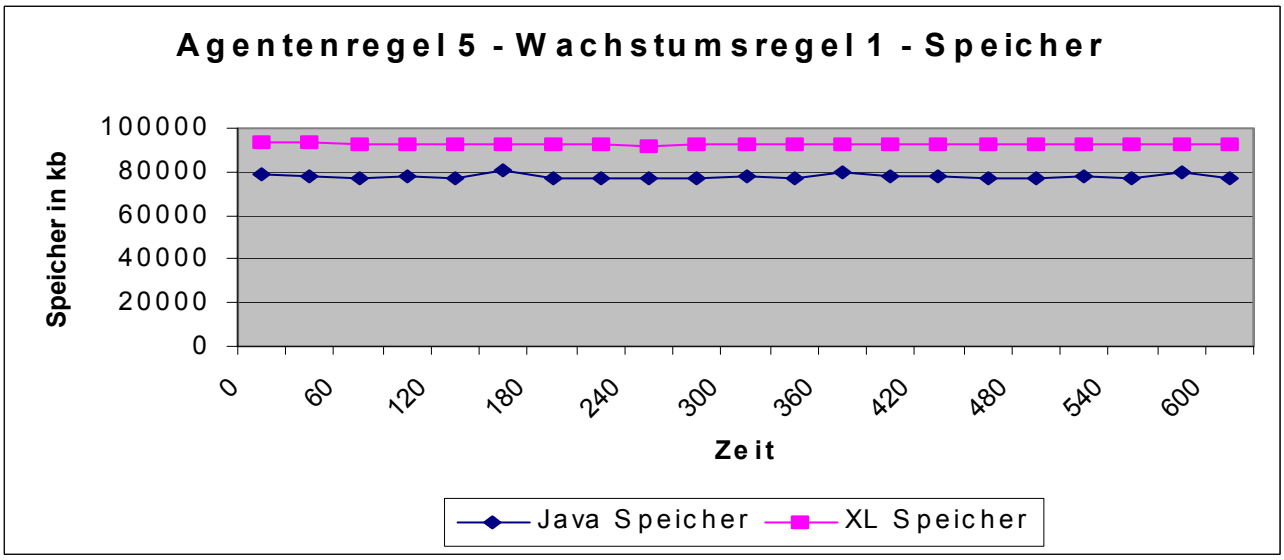
Bei den Messungen für die Regeln 4 und 5 kam es wieder zu einem Geburtenüberfluss. Ähnlich wie in Regel 2 sehen wir auch den Anstieg der Prozessorlast bei XL. Java scheint keinen erkennbaren Trend bei hoher oder niedriger Bevölkerung zu haben.


Agentenregel 1 - Wachstumsregel 1 - Speicher
Speicher in KB
100000
80000
60000
40000
20000
0
0
60
120
180
240
300
360
420
480
540
600
Zeit
Speicher Java
Speicher XL


Agentenregel 2 - Wachstumsregel 1 - Speicher
Speicher in kb
120000
100000
80000
60000
40000
20000
0
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher

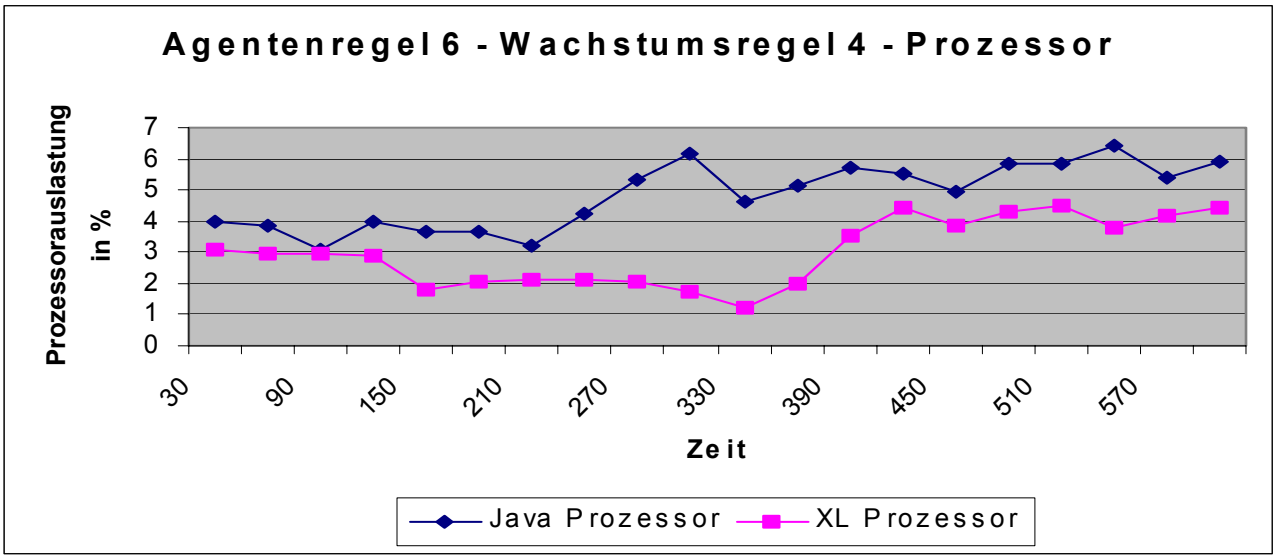

Agentenregel 3 - Wachstumsregel 1 - Speicher
Speicher in kb
115000
110000
105000
100000
95000
90000
85000
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher

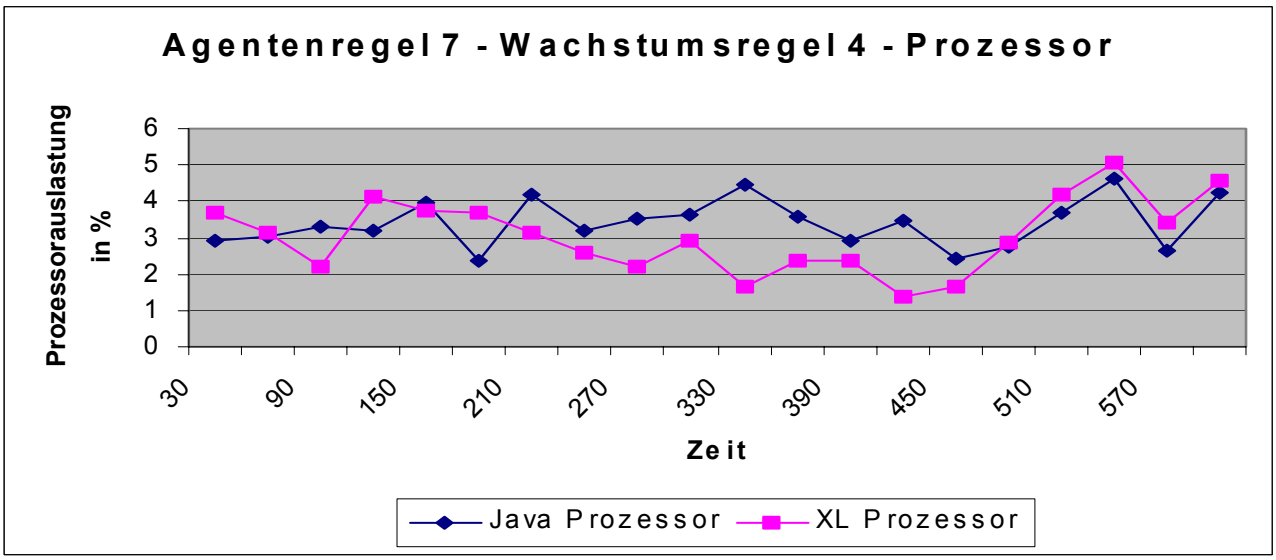


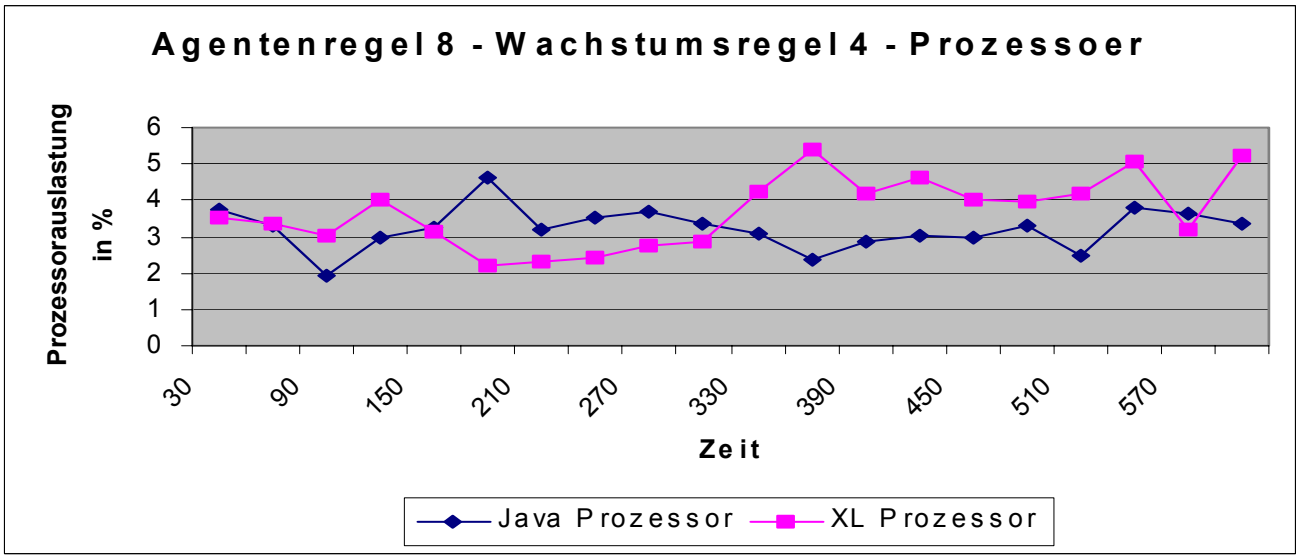


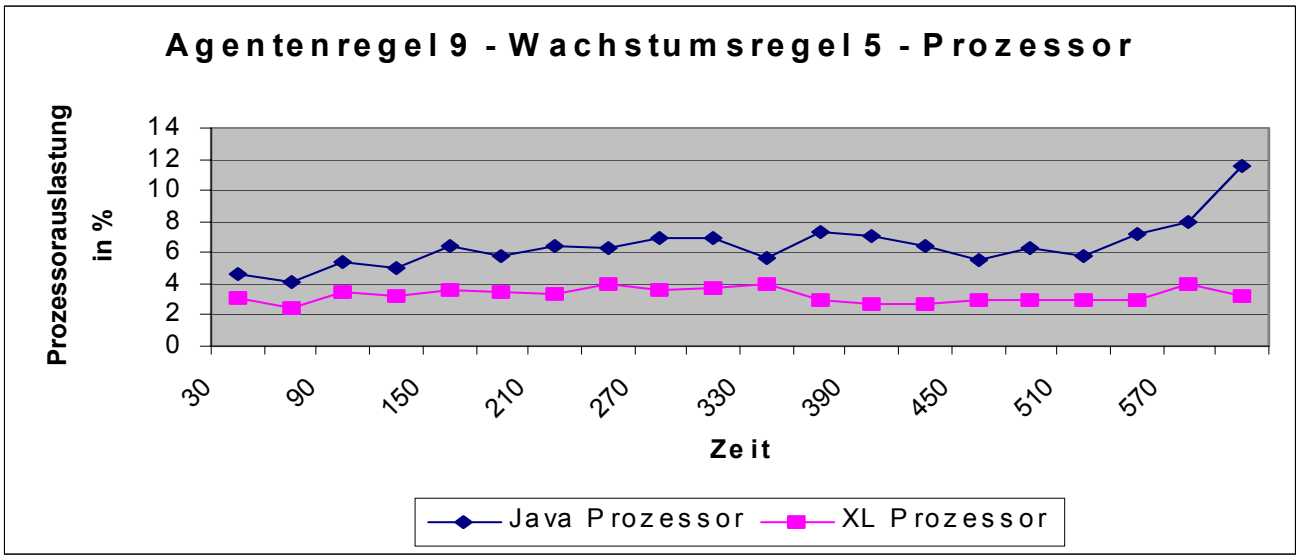
Die Speicherauslastung zeigt wenig Überraschungen. XL benötigt mehr Speicher als Java, ist dafür aber konstant in seinem Verbrauch. Java benötigt je nach Agentenzahl andere Speichermengen. Besonders gut zu sehen bei Regel 2 und 3.

Zum Schluss betrachten wir noch die Regeln 6-9 mit ihren relevanten Wachstumsregeln.

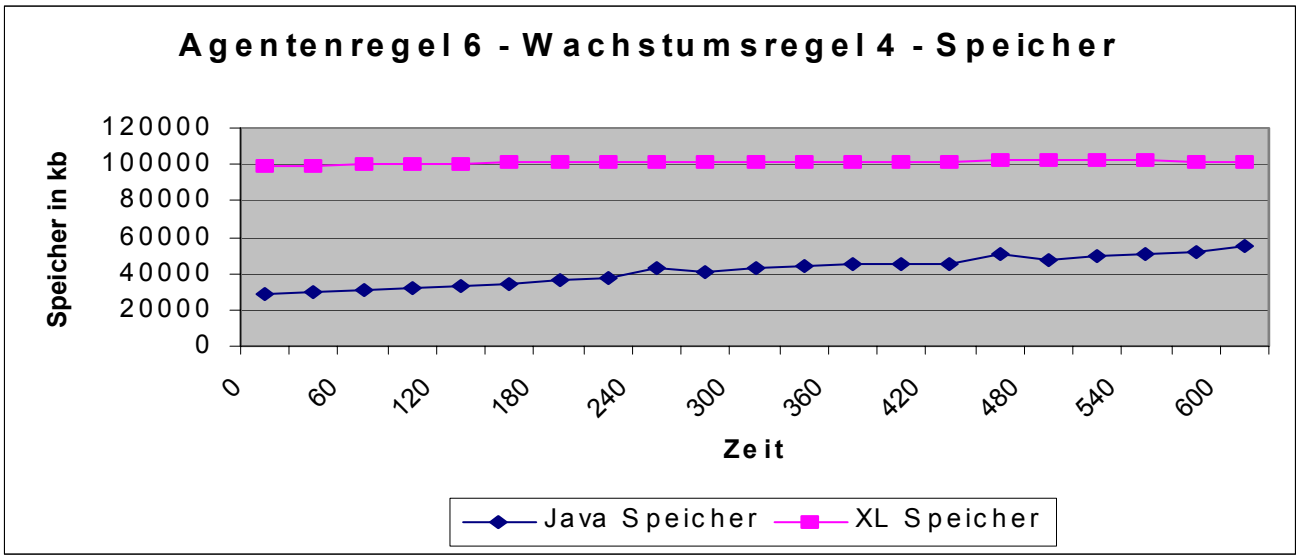


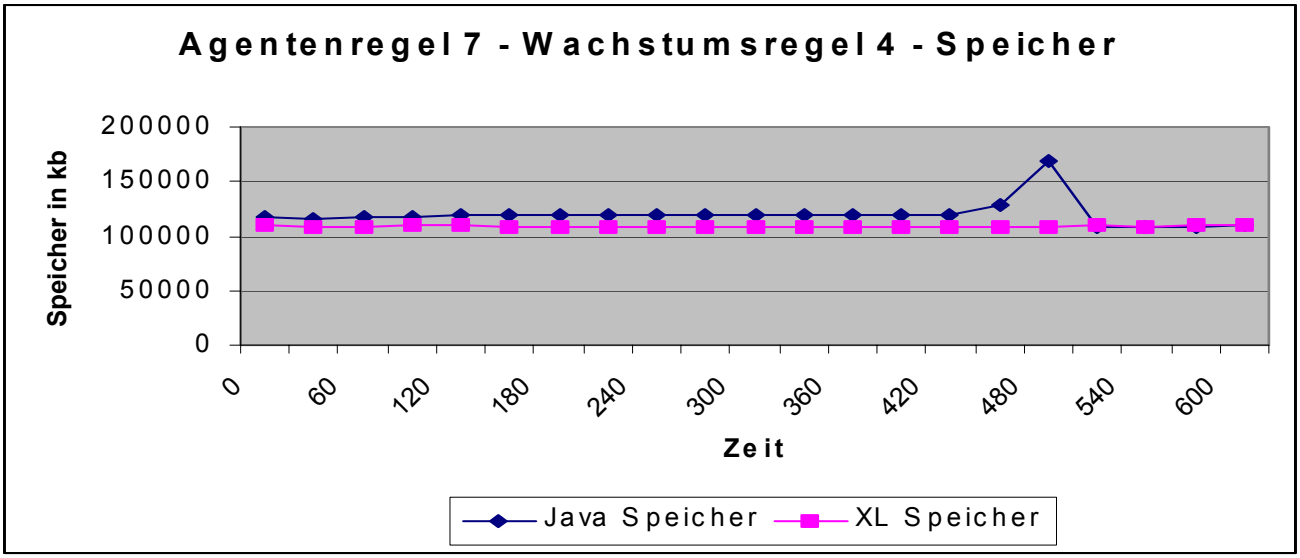


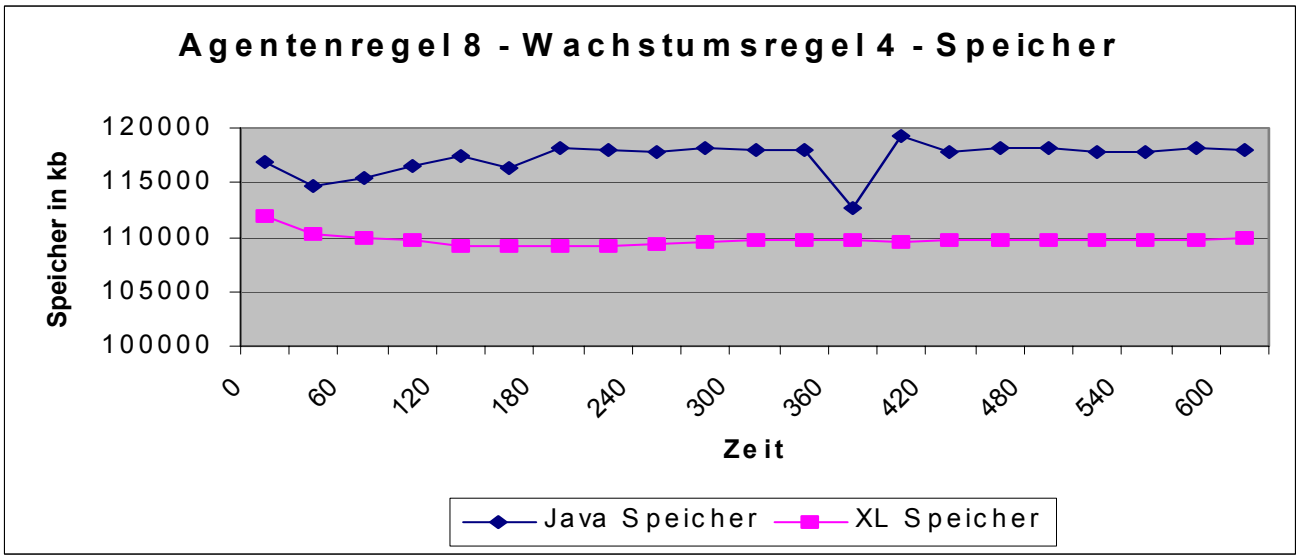


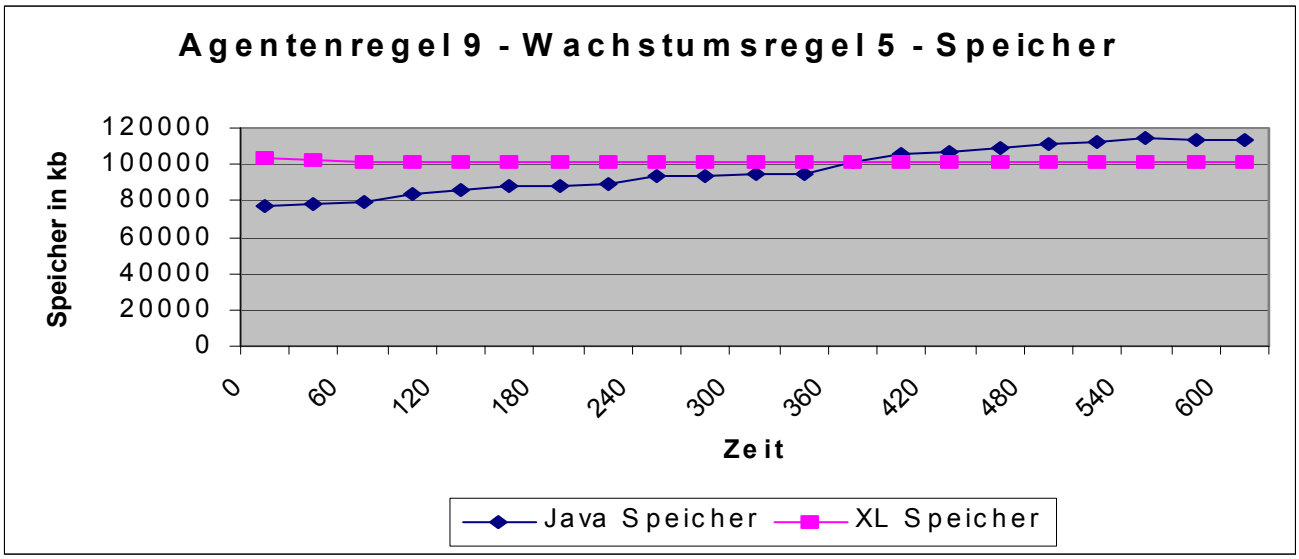


Wie wir sehen, verwischt sich der Unterschied zwischen Java und XL mit fortschreitender Komplexität der Agentenregeln immer mehr. Das kommt daher, dass der Code kaum mehr größere Unterschiede aufweist.


Agentenregel 6 - Wachstumsregel 4 - Speicher
Speicher in kb
120000
100000
80000
60000
40000
20000
0
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher


Agentenregel 7 - Wachstumsregel 4 - Speicher
Speicher in kb
200000
150000
100000
50000
0
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher


Agentenregel 8 - Wachstumsregel 4 - Speicher
Speicher in kb
120000
115000
110000
105000
100000
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher



In diesen Bereichen, wo Populationsfluktuationen gang und gäbe sind und es nahezu unmöglich ist, die beiden Testreihen mit einem ähnlichen Verlauf zu gestalten, erreichen auch mögliche Interpretationen ihre Grenzen. Das einzige, was wir auf jeden Fall bemerken, ist die Konstanz der Speicherauslastung des XL-Programms im Gegensatz zu Java.
Abschließend ist zu bemerken, dass XL eine stabile Speichernutzung und eine im Allgemeinen geringere Prozessorlast besitzt. Java hingegen hat eine größere Auslastung der Prozessorzeit, sowie einen meist geringeren, aber instabileren Speicherverbrauch.

7.3. Laufzeitvergleich

Ich habe beide Programme 50 Runden lang mit Agentenregel 1 und Wachstumsregel 1 laufen und per „java.util.Date()“ die Zeit ausgeben lassen. Diese Regelkombination ist die stabilste, was die Population betrifft. Der Vergleich kann auch nur stattfinden, wenn beide Programme so ähnlich wie möglich ablaufen. Die entsprechende Funktionalität findet sich auskommentiert noch in den Programmen. (XL am Anfang der run()-Funktion und Java innerhalb der Thread.run()-Funktion in SSMain_Window)
Java benötigte für die 50 Runden 8 Minuten und 20 Sekunden. XL schaffte es in 8 Minuten und 28 Sekunden. Das bedeutet eine Abweichung von 1,6%.

8. Abschlussbemerkungen

Die Arbeit an diesen beiden Programmen sowie die anschließende Auswertung hatte sowohl gute als auch schlechte Momente erlebt. Ich habe hier vieles gelernt, insbesondere, dass man 3D nicht auf die leichte Schulter nehmen sollte. Die Vielfalt der Möglichkeiten der Darstellungen können für einen Anfänger sehr verwirrend sein. Auch die XL/GroIMP-Programmierung stellte mich vor das Problem der Unerfahrenheit. Zusätzlich muss hier noch gesagt werden, dass noch keine vollständigen Dokumentationen und Tutorials vorliegen. Die mitgelieferten Beispiele beziehen sich meist nur auf kleinere Programme. An dieser Stelle möchte ich Herrn Prof. Dr. Winfried Kurth und Herrn Ole Kniemeyer danken, die mir mit Ihrem Rat zur Seite standen und zum Teil GroIMP meinen Bedürfnissen anpassten.
Des weiteren hatte sich XL/GroIMP als die bessere Testumgebung herausgestellt. Wenn ich das Verhalten zweier spezifischer Agenten untersuchen wollte, musste ich den Java-Code so umschreiben, das die Testbedingungen erfüllt wurden. In GroIMP konnte ich die Werte der durch Zufall erstellten Agenten dem durchzuführenden Test anpassen.
Als eine weitere große Schwierigkeit stellten sich die Messungen selbst heraus. Viele Messversuche musste ich abbrechen, da alle Agenten bereits gestorben waren. Im Idealfall sollten die 2x 19 Messungen etwa 6h und 20 Minuten dauern. Durch diese Probleme dauerte diese Phase des Projektes nahezu 2 Tage, in denen ich ununterbrochen auf sich bewegende Zylinder starrte.
Zum Abschluss kann ich nur sagen, dass ich, wie bei nahezu jedem Projekt, wichtige Erfahrungen gemacht habe und diverse Fehler sich so bei mir hoffentlich nicht wiederholen werden.

9. Literatur und verwendete Werkzeuge

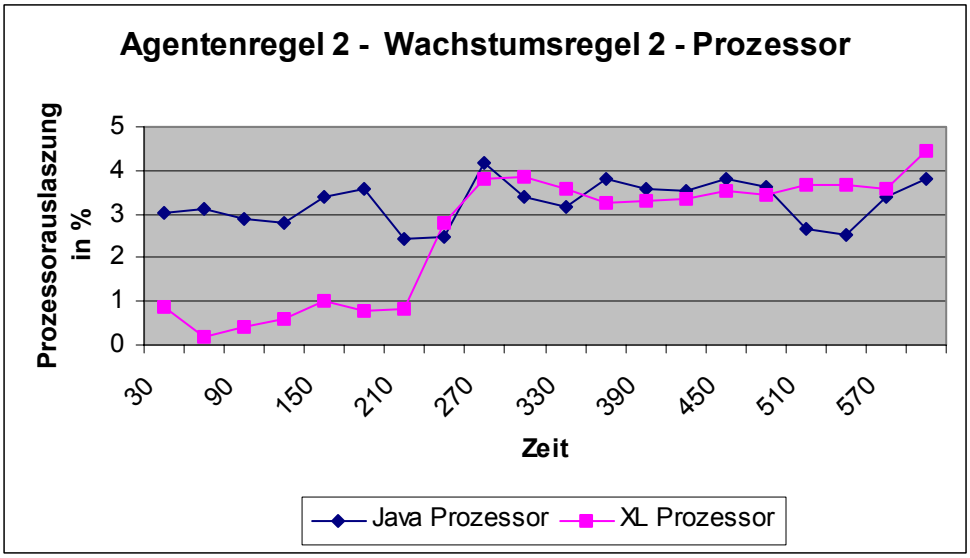
9.1. verwendete Programme

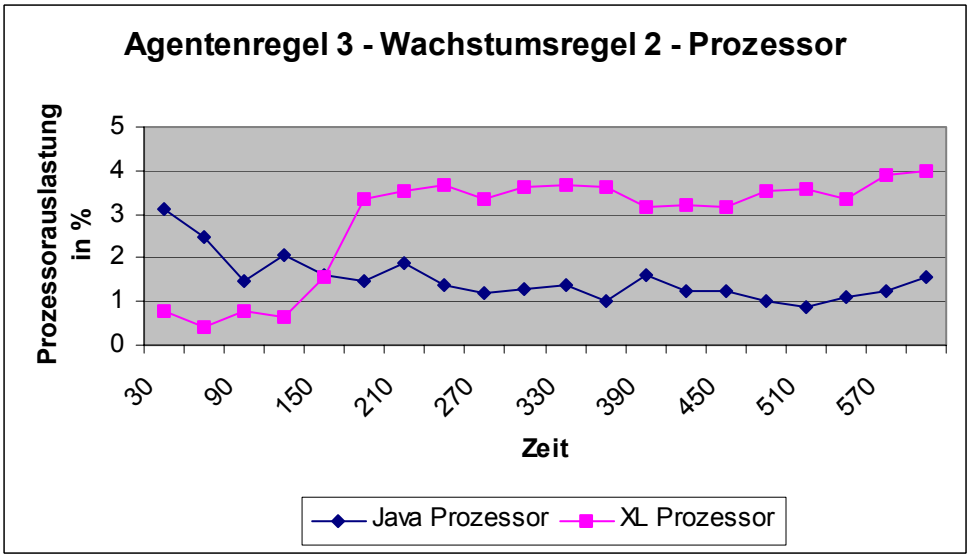
- Netbeans IDE 5.0 http://www.netbeans.org/ (13.9.2006)
- GroIMP Version 0.93 http://www.grogra.de/ (13.9.2006)
- Microsoft Word 2000
- Microsoft Excel 97
- JUDE 2.5.1 http://jude.change-vision.com/jude-web/index.html (13.9.2006)
- Microsoft Paint 5.1
- Java jdk 1.4.2_05
- Java3D 1.4.0_01

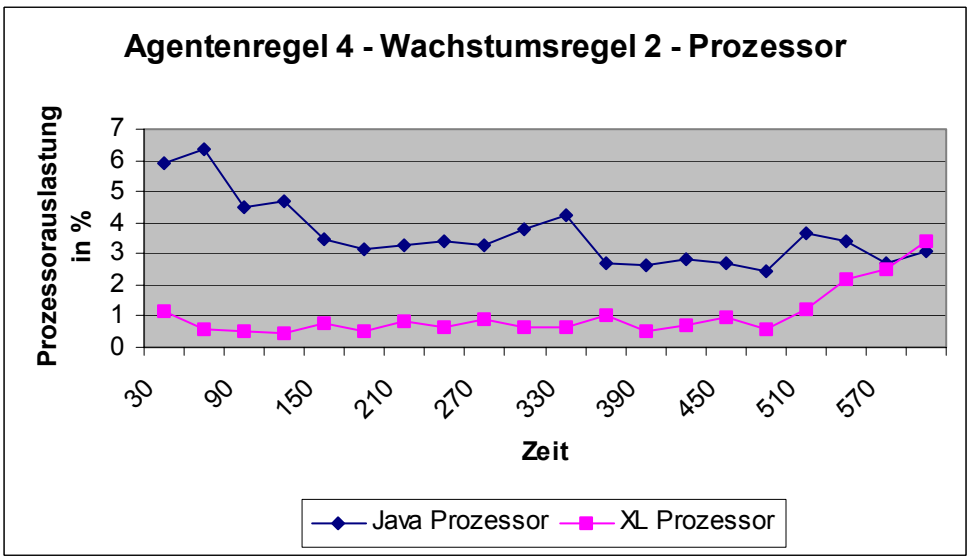
9.2. Literatur

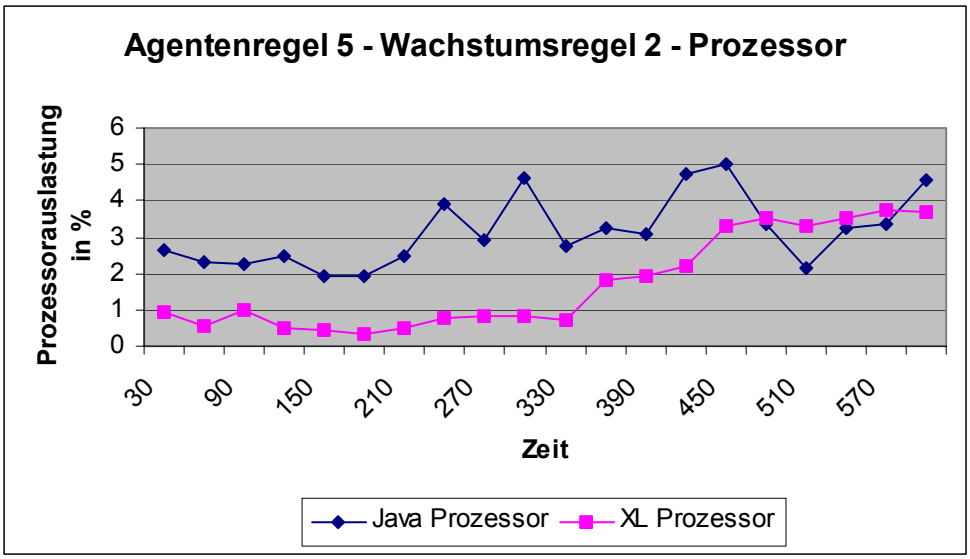
- Joshua M. Epstein / Robert Axtell , “Growing Artificial Societies” The Brookings Institution 1996
- Java3D API Dokumentation http://download.java.net/media/java3d/javadoc/1.4.0/index.html (13.9.2006)
- Java API Dokumentation http://java.sun.com/j2se/1.3/docs/api/
- (13.9.2006)
- XL-Language-Specification http://www.grogra.de/ (13.9.2006)
- GroIMP User-Manual http://www.grogra.de/ (13.9.2006)
- Rule-based modelling with the XL/GroIMP software : Ole Kniemeyer http://www.grogra.de/ (13.9.2006)
- Demonstration of the GroIMP software: Ole Kniemeyer http://www.grogra.de/ (13.9.2006)
- Java3D API Tutorial http://java.sun.com/developer/onlineTraining/java3d/ (13.9.2006)
- Vorlesungsskript Computergrafik : Prof. Dr. Winfried Kurth (12. 2. 2003) http://www-gs.informatik.tu-cottbus.de/~wwwgs/cg2_vorles.htm (13.9.2006)

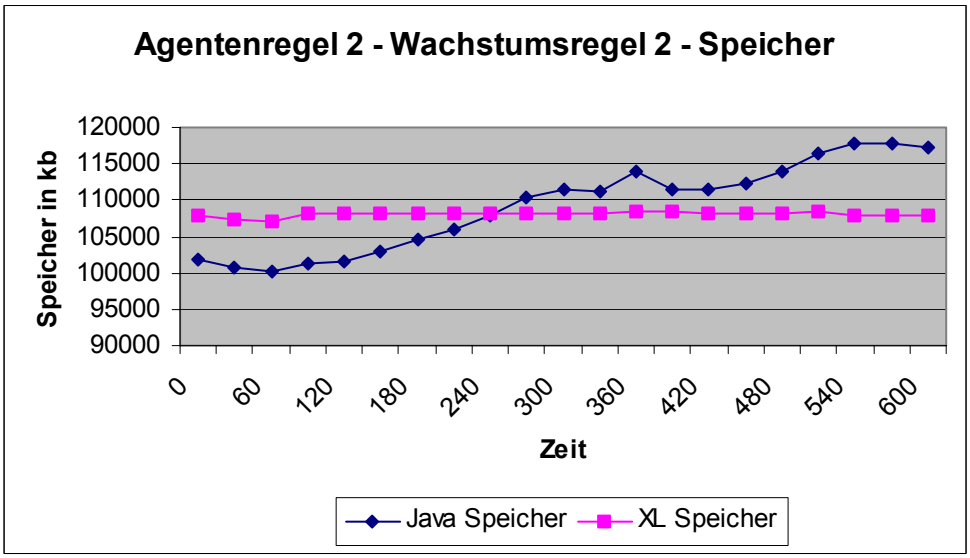
Diagramme, welche nicht in die Auswertung einflossen:

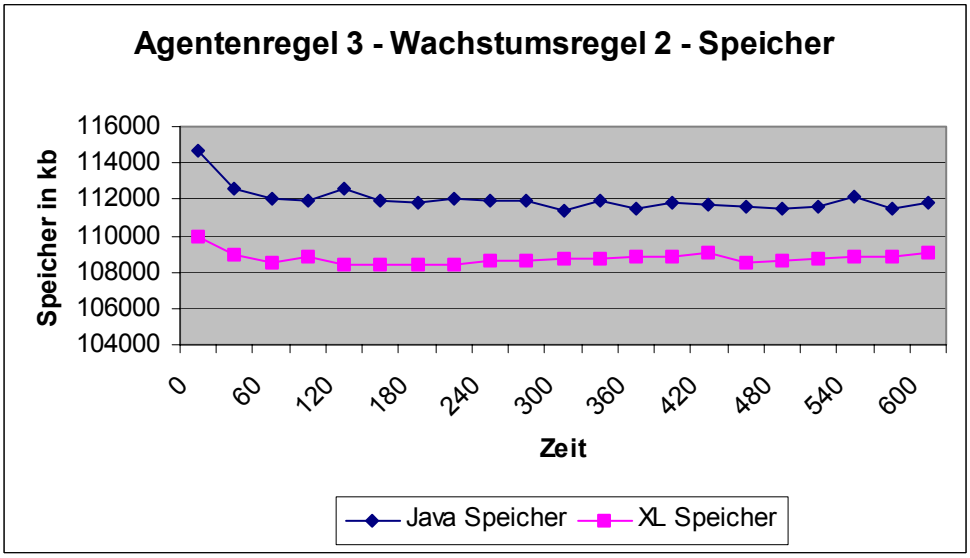


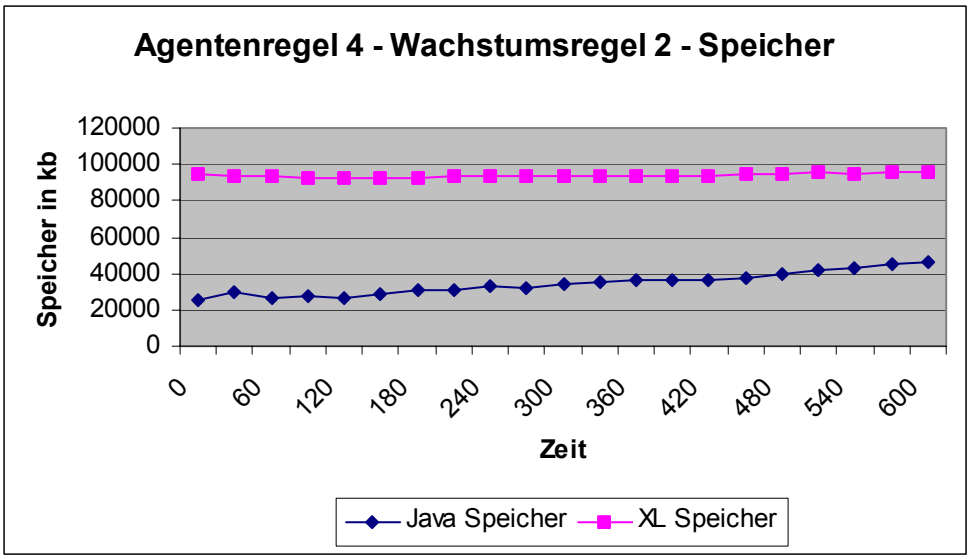


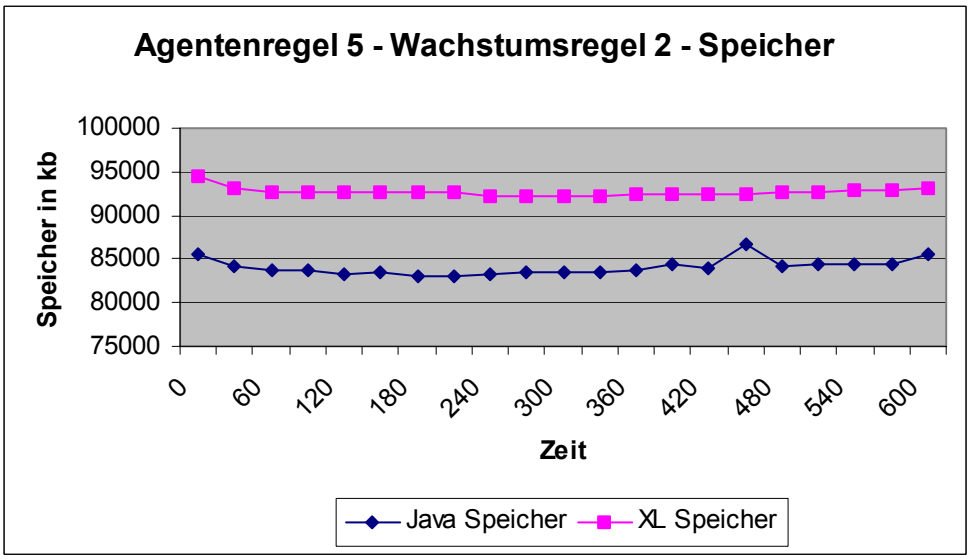


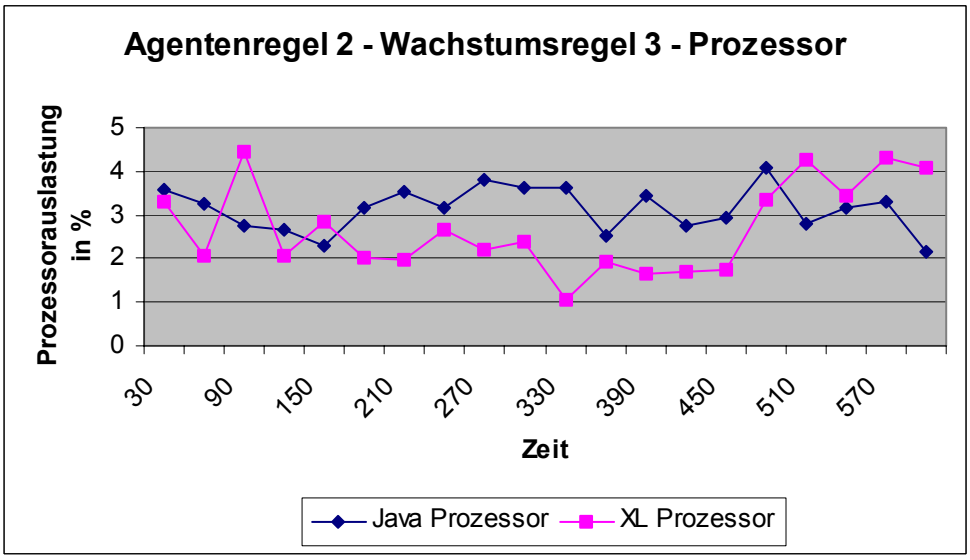

Agentenregel 5 - Wachstumsregel 2 - Prozessor
Prozessorauslastung in %
0
1
2
3
4
5
6
30
90
150
210
270
330
390
450
510
570
Zeit
Java Prozessor
XL Prozessor

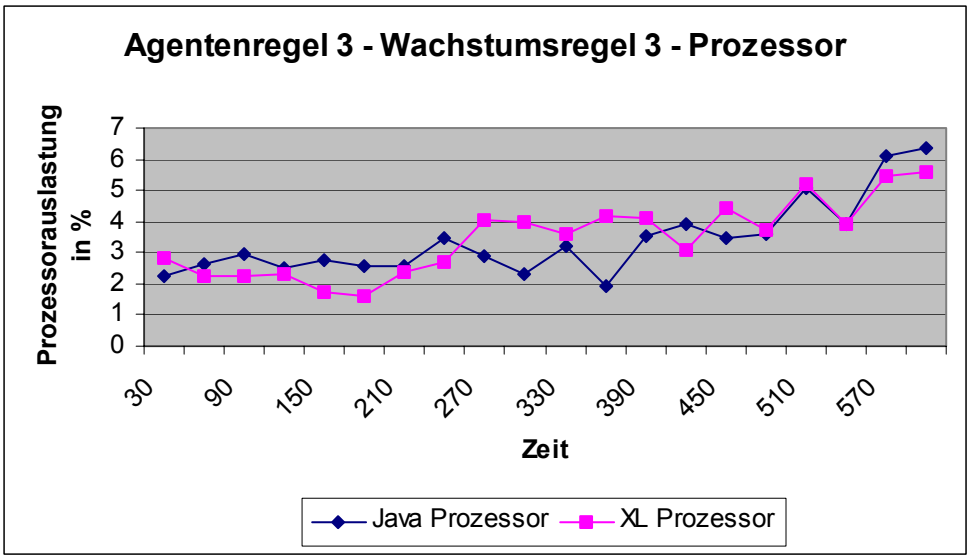

Agentenregel 2 - Wachstumsregel 2 - Speicher
Speicher in kb
90000
95000
100000
105000
110000
115000
120000
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher

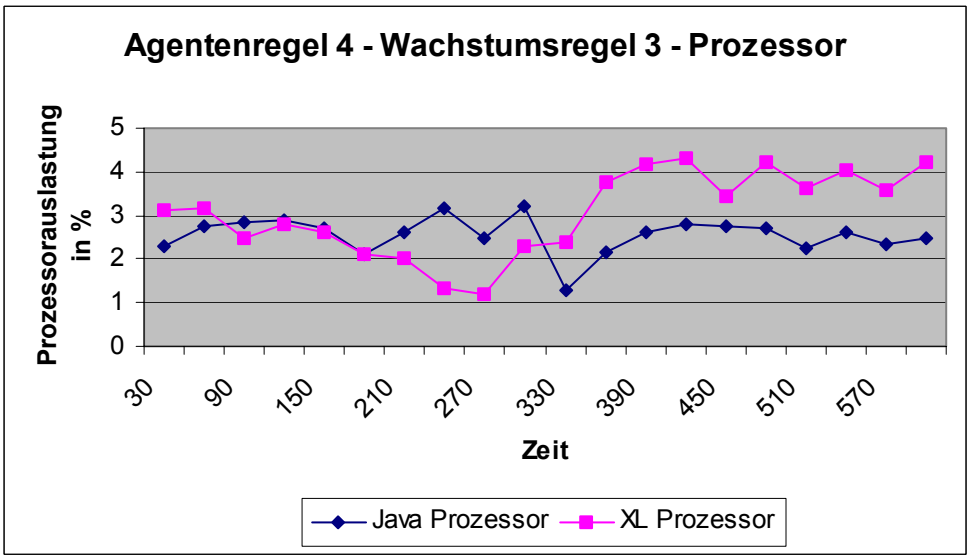

Agentenregel 3 - Wachstumsregel 2 - Speicher
Speicher in kb
104000
106000
108000
110000
112000
114000
116000
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher

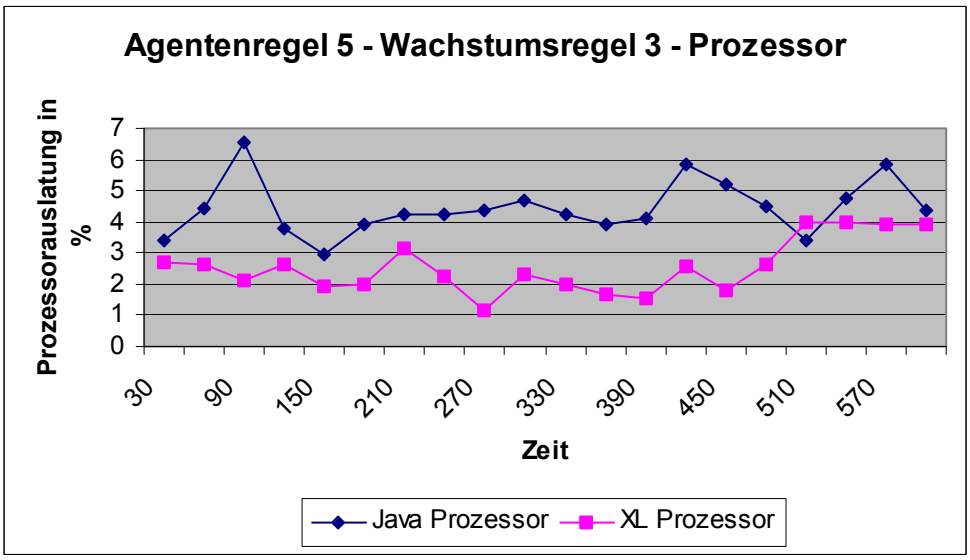

Agentenregel 4 - Wachstumsregel 2 - Speicher
Speicher in kb
120000
100000
80000
60000
40000
20000
0
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher

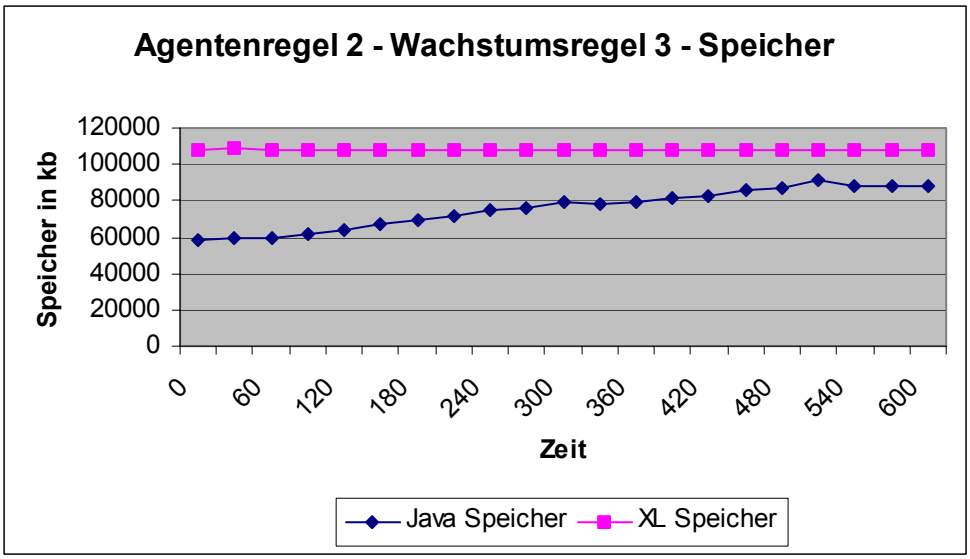

Agentenregel 5 - Wachstumsregel 2 - Speicher
Speicher in kb
100000
95000
90000
85000
80000
75000
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher

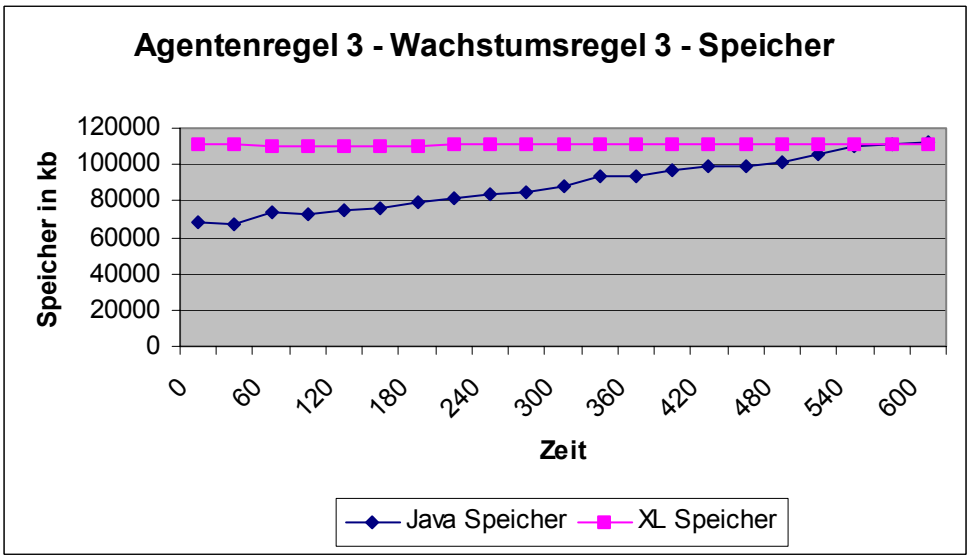

Agentenregel 2 - Wachstumsregel 3 - Prozessor
Prozessorauslastung in %
5
4
3
2
1
0
30
90
150
210
270
330
390
450
510
570
Zeit
Java Prozessor
XL Prozessor

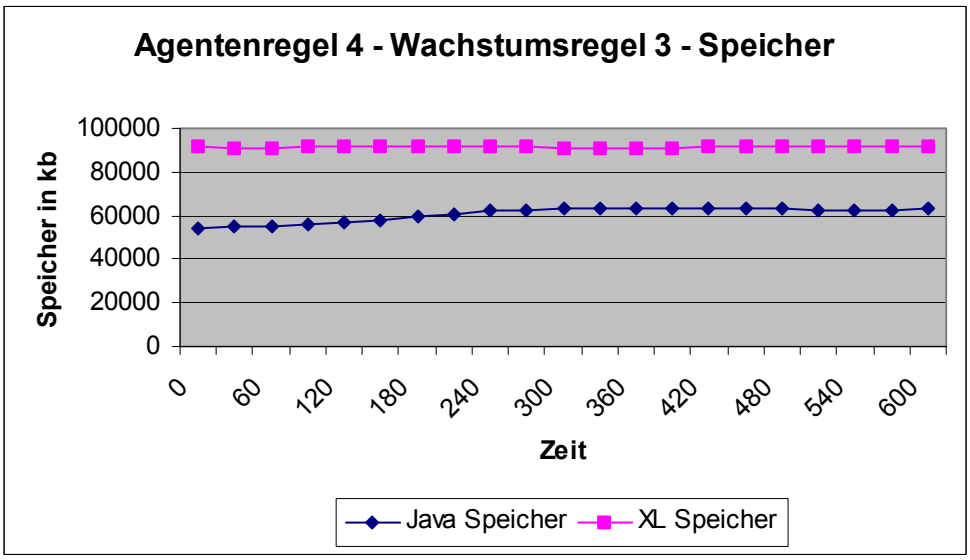

Agentenregel 3 - Wachstumsregel 3 - Prozessor
Prozessorauslastung in %
0
1
2
3
4
5
6
7
30
90
150
210
270
330
390
450
510
570
Zeit
Java Prozessor
XL Prozessor

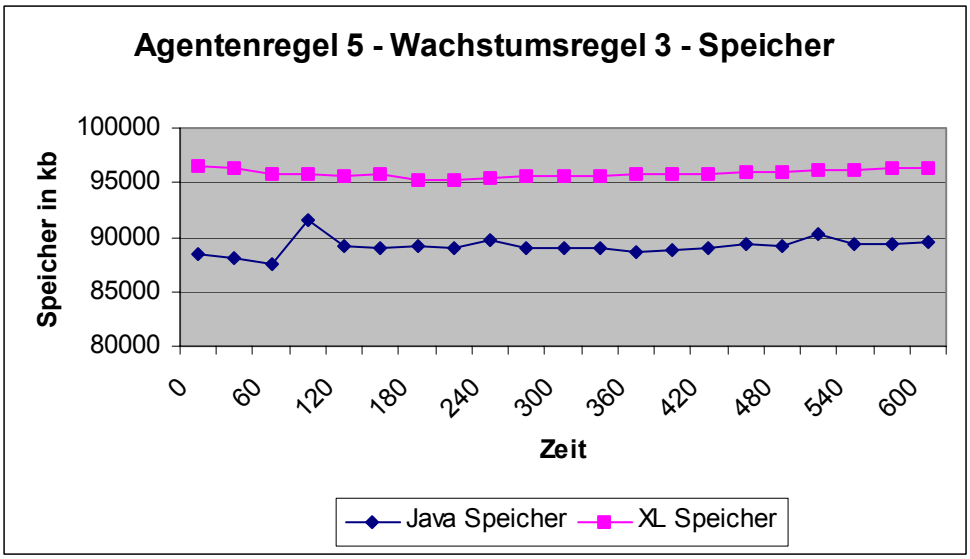

Agentenregel 4 - Wachstumsregel 3 - Prozessor
Prozessorauslastung in %
0
1
2
3
4
5
30
90
150
210
270
330
390
450
510
570
Zeit
Java Prozessor
XL Prozessor


Agentenregel 5 - Wachstumsregel 3 - Prozessor
Prozessorauslatung in %
0
1
2
3
4
5
6
7
30
90
150
210
270
330
390
450
510
570
Zeit
Java Prozessor
XL Prozessor


Agentenregel 2 - Wachstumsregel 3 - Speicher
Speicher in kb
120000
100000
80000
60000
40000
20000
0
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher


Agentenregel 3 - Wachstumsregel 3 - Speicher
Speicher in kb
120000
100000
80000
60000
40000
20000
0
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher


Agentenregel 4 - Wachstumsregel 3 - Speicher
Speicher in kb
100000
80000
60000
40000
20000
0
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher


Agentenregel 5 - Wachstumsregel 3 - Speicher
Speicher in kb
100000
95000
90000
85000
80000
0
60
120
180
240
300
360
420
480
540
600
Zeit
Java Speicher
XL Speicher