„Entwicklungsumgebung“ für zelluläre Automaten

Teilleistung 1 im Kurs „Soft Computing“, Studiengang VAWi, Universität Duisburg-Essen, Sommersemester 2015

Wolfgang Bongartz

Das Programm erlaubt die Definition eines zellulären Automaten mittels einer kleinen Skriptsprache. Es beschränkt sich dabei auf einige wenige Sprachelemente, um den Aufwand nicht allzu hoch werden zu lassen[[1]](#footnote-1). Grundsätzlich lässt es sich aber leicht um zusätzliche Sprachelemente erweitern, sodass prinzipiell auch die Definition komplexer ZA möglich ist. Die Simulation der im Programm definierten ZA wird grafisch dargestellt[[2]](#footnote-2).

Das Programm ist in Java realisiert und sollte auf jedem PC ausgeführt werden können, auf dem eine Java-Laufzeitumgebung installiert ist[[3]](#footnote-3). Das ausführbare Programm befindet sich im File za.jar. Das komplette Eclipse-Projekt[[4]](#footnote-4) inklussive der Sourcen ist im Unterverzeichnis ZA abgelegt[[5]](#footnote-5). Im Wurzelverzeichnis befinden sich ausserdem die zwei Skript-Files GameOfLife.cad und TheTimeMachine.cad, die zwei kleine ZA definieren, mit denen sich die Funktionsweise des Programms beispielhaft nachvollziehen läßt. Beide ZA werden in diesem Dokument kurz erläutert. Das Skript TheTimeMachine.cad wird beim Programmstart automatisch geladen.

# Die Bedienung

Auf der linken Seite des Fensters befindet sich der Editor, in dem das aktuelle Skript bearbeitet werden kann. Auf der rechten Seite des Fensters befindet sich der Anzeigebereich (der beim Programmstart noch leer ist).

## Das File-Menü

* ‚Load definition…‘ lädt ein Textfile in den Editor
* ‚Save definition…‘ speichert den aktuellen Inhalt des Editors in ein Textfile
* ‚Quit‘ beendet das Programm (nicht gespeicherte Änderungen gehen verloren)

## Buttons unterhalb des Editors

* ‚Compile‘ versucht, aus dem aktuellen Inhalt des Editors einen ZA zu erzeugen. Fehlermeldungen erscheinen ggf. in einer Messagebox. Erst nach Betätigung dieses Buttons werden Änderungen aus dem Editor in die Simulation übernommen
* ‚Start‘ startet die Simulation des zuletzt erzeugten ZA (Änderungen müssen erst durch ‚Compile‘ übernommen werden)
* ‚Stop‘ beendet die Simulation
* ‚Step‘ erzeugt nur die nächste Generation des ZA und stoppt dann wieder

# Die Skriptsprache

## Definition eines ZA

Die Definition eines ZA beginnt mit den Schlüsselworten CELLULAR AUTOMAT und einer öffnenden Klammer[[6]](#footnote-6). Sie endet mit einer schließenden Klammer. Innerhalb der Klammern werden das „Habitat“, die Menge der möglichen „Bewohner“ und alle Regeln definiert[[7]](#footnote-7). Zusätzlich läßt sich mit dem Schlüsselwort POPULATE noch angeben, wie viele Zellen nach dem Zufallsprinzip initial mit einem Bewohner „bestückt“ werden sollen (Angabe in Prozent).

Beispiel:

cellular automat (

…

populate=30%

…

)

## Definition des Habitats

Die Menge aller Zellen eines ZA wird hier als ‚Habitat‘ bezeichnet. Ein ZA hat genau ein Habitat. Die Definition des Habitats beginnt mit dem Schlüsselwort HABITAT und einer öffnenden Klammer. Sie endet mit einer schließenden Klammer. Innerhalb der Klammern müssen folgende Angaben gemacht werden:

* Breite des Habitats (Schlüsselwort: WIDTH)
* Höhe des Habitats (Schlüsselwort: HEIGHT)
* Umgebungsart (Moore-Umgebung oder Neumann-Umgebung; Schlüsselwort: ENVIRONMENT gefolgt von MOORE oder NEUMANN)

Beispiel:

habitat (

width=400

height=400

environment=moore

)

## Definition der Bewohner-Arten

Dies ist die Definition der Menge der möglichen Zelleninhalte. Diese Menge darf nicht leer sein.

Die Definition einer Bewohner-Art beginnt mit den Schlüsselworten INHABITANT TEMPLATE gefolgt von einem Bezeichner, mit dem sich die Bewohner-Art in den Regeldefinitionen ansprechen läßt. Es folgt eine öffnende Klammer. Die Definition endet mit einer schließenden Klammer und der Angabe der Farbe, in welcher die Bewohner dargestellt werden sollen (Schlüsselwort IS gefolgt von der Farbangabe BLACK, BLUE oder RED).

Innerhalb der Klammern kann eine Attributliste definiert werden. Die einzelnen Attribute werden darin mit einem Bezeichner und einem Startwert definiert. Sie werden durch Semikola voneinander getrennt. Bei jedem „neugeborenen Bewohner“ werden diese Attribute mit den hier angegebenen Startwerten belegt. In den Regeln können die Attribute verwendet und geändert werden.

Beispiel:

inhabitant template Eloi (

energy=10;

age=0;

) is blue

## Definition der Regeln

Jede Regel beginnt mit dem Schlüsselwort IF. Es folgt ein boolscher Ausdruck (siehe unten) und dann das Schlüsselwort THEN. Danach wird ein Kommando (siehe unten) angegeben und die Definition mit einem Semikolon abgeschlossen. Es können beliebig viele Regeln definiert werden. Jede Regel bezieht sich auf „die aktuelle Zelle“[[8]](#footnote-8).

Beispiel:

if check(Eloi) and (age>20) then kill();

## Boolsche Ausdrücke

Ein Ausdruck besteht immer aus genau einer Operation der Form

operand1 operator operant2

oder

NOT operand.

Als Operanden werden derzeit unterstützt:

* Die boolschen Konstanten TRUE und FALSE
* Natürliche Zahlen (wobei 0 als FALSE und alle anderen Zahlen als TRUE interpretiert werden)
* Funktionen (siehe unten)
* Bezeichner von Bewohner-Attributen (siehe oben)
* Geklammerte boolsche Ausdrücke

Folgende Operatoren sind derzeit implementiert:

|  |  |
| --- | --- |
| NOT | Logische Negation |
| AND | Logische Konjunktion |
| OR | Logische Disjunktion |
| XOR | Logische Kontravalenz |
| == | Arithmetische Gleichheit |
| <= | Kleiner oder gleich (arithmetisch) |
| >= | Größer oder gleich (arithmetisch) |
| < | Kleiner (arithmetisch) |
| > | Größer (arithmetisch) |

Folgende Funktionen werden unterstützt:

|  |  |  |
| --- | --- | --- |
| **Funktion** | **Parameter** | **Beschreibung** |
| Count | Name einer Bewohner-Art | Liefert die Anzahl der entsprechenden Bewohner in der Umgebung der Zelle zurück |
| getValue | Attribut-Bezeichner | Gibt den aktuellen Wert des entsprechenden Attributs zurück |
| check | optional: Name einer Bewohner-Art | Prüft, ob die Zelle bewohnt ist. Wird der Name einer Bewohner-Art übergeben so wird geprüft, ob die Zelle von genau dieser Art bewohnt ist |

Komplexere Ausdrücke lassen sich derzeit also nur durch Einklammern realisieren.

## Kommandos

Ein Kommando ändert den Inhalt einer Zelle.

|  |  |  |
| --- | --- | --- |
| **Kommando** | **Parameter** | **Beschreibung** |
| KILL |  | Leert eine Zelle. |
| CREATE | Name einer Bewohner-Art | Erzeugt einen neuen Bewohner in der Umgebung der Zelle. Dazu wird eine Zelle nach dem Zufallsprinzip ausgewählt. Ist sie nicht leer wird kein neuer Bewohner erzeugt. |
| ADD | Attribut-Bezeichner, Integer | Addiert oder Substrahiert eine Zahl zu bzw. von einem Attribut, |
| DEPLOY | Attribut-Bezeichner, Name einer Bewohner-Art | Verteilt den Wert eines Attributs auf die Bewohner der Zellen in der Umgebung. Läßt sich auf eine bestimmte Bewohner-Art einschränken. |

# Die Beispiele

## GameOfLife.cad

Hierbei handelt es sich um einen ZA nach den Regeln von „Conway's Game of Life“.

## TheTimeMachine.cad

Dieser ZA basiert auf einer Idee aus dem Roman „The Time Machine“ von H. G. Wells. In diesem Roman existieren auf der Erde in einer fernen Zukunft zwei Arten, die sich aus den Nachkommen des Menschen entwickelt haben: Die Eloi und die Morlock. Die Morlock versorgen die Eloi mit allem, was zum Leben nötig ist. Die Eloi widerum führen ein auf den ersten Blick sorgenfreies und völlig unbeschwertes Leben. Allerdings dienen sie den Morlock als Nahrungsquelle, werden von diesen also letztlich als Nutztiere gehalten.

Für den ZA gelten die folgenden Regeln:

* In jeder Zelle kann ein Eloi oder ein Morlock leben. Eloi werden in blau und Morlocks werden in rot dargestellt
* Sowohl Eloi als auch Morlock altern in jedem Durchgang. Außerdem verfügt jedes Individuum über einen gewissen Energievorrat, von dem in jedem Durchgang etwas verbraucht wird. Ist die Energie aufgebraucht stirbt das Individuum
* Lebt ein Eloi in der Umgebung mindestens eines Morlocks, so wird es in jedem Durchgang mit neuer Energie versorgt
* Leben in der Umgebung eines Eloi mindestens zwei weitere Eloi und mindestens ein Morlock, so wird nach dem Zufallsprinzip eine Zelle der Umgebung bestimmt. Ist diese Zelle leer, so entsteht in ihr ein neuer Eloi
* Leben in der Umgebung eines Morlock mindestens zwei weitere Morlocks und mindestens ein Eloi, so wird nach dem Zufallsprinzip eine Zelle der Umgebung bestimmt. Ist diese Zelle leer, so entsteht in ihr ein neuer Morlock
* Leben in der Umgebung eines Eloi weniger als zwei weitere Eloi, so stirbt es an Vereinsamung
* Leben in der Umgebung eines Eloi mehr als fünf weitere Eloi, so stirbt es an Platzmangel
* Leben in der Umgebung eines Morlock mehr als drei weitere Morlock, so stirbt es an den Streitereien mit den anderen Morlocks
* Ist ein Eloi älter als 20 so wird seine Energie an die Morlocks in seiner Umgebung verteilt und es stirbt
* Ist ein Morlock älter als 70, so stirbt es an Altersschwäche

# Die Programmstruktur

Das Programm besteht aus den vier Packages

* automat,
* gui,
* parser und
* rules.

## Das Package ‚automat‘

Hier sind alle Klassen enthalten, die für den Aufbau des eigentlichen ZA nötig sind.

|  |  |
| --- | --- |
| **Klasse** | **Funktion** |
| CellularAutomat | Repräsentiert den ZA und beinhaltet das Habitat (also die Menge aller Zellen), Templates der möglichen Zelleninhalte (‚Inhabitanten‘) und die Regeln. Ist in der Lage, innerhalb eines separaten Threads[[9]](#footnote-9) die jeweils nächste Generation einer Simulation zu erzeugen. |
| Habitat | Repräsentiert die Menge aller Zellen eines ZA. Die Zellen sind in Zeilen und Spalten angeordnet.  Hat eine Breite und eine Höhe und kann entweder Neumann- oder Moore-Umgebungen unterstützen. Kümmert sich um das Füllen und das Leeren der Zellen. Unterstützt das Visitor-Pattern, damit bequem auf alle Zellen zugegriffen werden kann. |
| Inhabitant | Repräsentiert einen konkreten Zelleninhalt. Hat einen Namen und eine Liste von Key-Value-Paaren (Attribute) sowie eine Darstellungsfarbe. |
| CellInformation | Fasst Informationen zu einer Zelle zusammen (Bewohner, Koordinaten, …). |
| Coordinate | Repräsentiert Koordinaten in einem zweidimensionalen Koordinatensystem. Erlaubt das Navigieren und die Überprüfung einer Koordinate gegen ein Referenzsystem. |
| Direction | Repräsentiert die innerhalb eines zweidimensionalen Koordinatensystems möglichen Bewegungsrichtungen. Kann eine zufällig ausgewählte Bewegungsrichtung liefern. |
| Environment | Repräsentiert die Umgebung einer Zelle. Kann Informationen zur Umgebung liefern. |
| CollectCommandsVisitor | Wird für die Implementierung des Visitor-Patterns innerhalb der Run-Methoder der Klasse CellularAutomat verwendet. Besucht alle Zellen und sammelt dabei die aufgrund der zutreffenden Regeln auszuführenden Aktionen ein. |
| PaintHabitatEvent | Implementiert ein Event, dass aus der Run-Methode der Klasse CellularAutomat versendet wird, sobald eine neue Generation des ZA zur Verfügung steht. Dient der Entkopplung des GUI vom Anwenungscode und sorgt dafür, dass die Darstellung und die Simulation parallel ablaufen können. |

|  |  |
| --- | --- |
| **Interface** | **Funktion** |
| PaintHabitatEventObserver | Wird von der Klasse MainWindow implementiert, da diese auf das Event PaintHabitatEvent reagiert. Sorgt für die Bereitstellung der für die Eventverarbeitung nötigen Methoden. |
| ReferenceSystem2D | Wird von der Klasse Habitat implementiert. Sorgt für die Bereitstellung der Methoden, die für die Überprüfung von Koordinaten nötig sind. |
| Visitor | Wird von der Klasse CollectCommandsVisitor implementiert. Sorgt für die Bereitstellung einer Methode, die beim Besuchen einer Zelle ausgeführt wird. |

## Das Package ‚gui‘

|  |  |
| --- | --- |
| **Klasse** | **Funktion** |
| MainWindow | Stellt die main()-Methode und (mit Ausnahme des Darstellungsbereichs für die Simulation) auch alle Elemente des GUI bereit. |
| DrawPanel | Stellt den Darstellungsbereich für die Simulation zur Verfügung. |

## Das Package ‚parser‘

Dieses Package enthält den Code des Parsers, der aus dem Skript das Modell des ZA erzeugt. Dieser Parser wurde mit dem Tool JavaCC auf Basis der im File parser/Grammar.jj definierten Grammatik generiert. Alle weiteren Quelltexte in diesem Package wurde also von JavaCC erzeugt. Das File Grammar.jj enthält alle Konstrukte der Skriptsprache.

## Das Package ‚rules‘

In diesem Package sind alle Klassen enthalten, welche die Regeln des ZA repräsentieren. Eine Regel besteht immer aus einer Bedingung und einer Aktion. Die Bedingung kann aus mehreren miteinander verknüpften Operationen bestehen und ist daher als Baum abgebildet. Regeln werden beim Erzeugen einer neuen Generation in zwei Schritten verarbeitet. Im ersten Schritt werden die Bedigungen aller Regeln für jede Zelle ausgewertet. Ergibt diese Auswertung das Ergebnis TRUE, so wird die Aktion der jeweiligen Regel zunächst in einer Liste gespeichert. Im zweiten Schritt werden alle Aktionen auf die einzelnen Zellen angewendet.

Eine zentrale Rolle in diesem Package spielt die Klasse Rule, die eine Bedingung (bzw. die in einer Baumstruktur organisierten (Sub-) Bedingungen) und die ggf. durchzuführende Aktion enthält. Die Klasse stellt die Methode apply zur Verfügung, die einer als Parameter übergebenen Liste von Aktionen eine weitere Aktion hinzufügt, falls die Auswertung der Bedingung TRUE ergibt.

|  |  |
| --- | --- |
| **Interface** | **Funktion** |
| Evaluable | Wird von allen Klassen implementiert, die einen auswertbaren Ausdruck repräsentieren. Stellt die Methode evaluate() bereit. |
| InhabitantFactory | Wird von der Klasse CellularAutomat implementiert, da sie aus den dort hinterlegten Templates neue Zellinhalte (‚Inhabitanten‘) erzeugen kann. Wird von allen Aktionen verwendet, die neue Zellinhalte erzeugen (bspw. von Create). |

### Ausdrücke

Bedingungen bestehen aus einem Ausdruck. Ein Ausdruck besteht aus einem Operator sowie einem oder zwei Operanden. Der Operator wird von einer Klasse repräsentiert, die von der Klasse Operation abgeleitet ist. Operanden werden von Klassen repräsentiert, welche das Interface Evaluable implementieren. Da auch die Klasse Operation das Interface Evaluable implementiert können auch Operatoren widerum als Operanden auftreten. Auf diese Weise ergibt sich eine Baumstruktur. Ein Ausdruck wird durch den Aufruf der evaluate()-Methode ausgewertet.

|  |  |
| --- | --- |
| **Klasse** | **Funktion** |
| Operation | Basisklasse aller Ausdrücke. |
| AndExpression | Implementiert die Und-Verknüpfung. |
| ConstExpression | Implementiert Ausdrücke, die nur aus einer Konstanten bestehen (TRUE und FALSE). |
| EqualsExpression | Implementiert ‚ist gleich‘. |
| LessThanExpression | Implementiert ‚ist kleiner‘. |
| LessThanOrEqualExpression | Implementiert ‚ist kleiner oder gleich‘. |
| MoreThanExpression | Implementiert ‚ist größer‘. |
| MoreThanOrEqualExpression | Implementiert ‚ist größer oder gleich‘. |
| NotExpression | Implementiert die Negation. |
| OrExpression | Implementiert die Oder-Verknüpfung. |
| Skalar | Implementiert Ausdrücke, die nur aus einem Integer bestehen[[10]](#footnote-10). |
| XorExpression | Implementiert die exklussive Oder-Verknüpfung. |

### Funktionen

Funktionen können innerhalb eines Ausdrucks als Operanden auftreten. Eine Funktion liefert also immer einen Wert zurück.

|  |  |
| --- | --- |
| **Klasse** | **Funktion** |
| GetValueFunction | Implementiert eine Funktion, die den aktuellen Wert eines als Parameter übergebenen Attributs zurückliefert. Bezieht sich immer auf den Inhalt (also den ‚Bewohner‘) der gerade untersuchten Zelle. |
| CountFunction | Implementiert eine Funktion, welche die Anzahl der ‚Bewohner‘ in der Umgebung der Zelle zurückliefert. Falls ein Bezeichner als Parameter übergeben wird werden nur solche ‚Bewohner‘ gezählt, deren Art-Bezeichnung mit dem Parameter übereinstimmt. |
| CheckFunction | Implementiert eine Funktion, die prüft, ob die Zelle ‚bewohnt‘ ist. Falls ein Bezeichner als Parameter übergeben wird so wird überprüft, ob die Zelle von einem ‚Bewohner‘ der betreffenden ‚Art‘ bewohnt wird. |

### Aktionen

Eine Aktion führt eine Veränderung an einer Zelle durch.

|  |  |
| --- | --- |
| **Klasse** | **Funktion** |
| Command | Basisklasse aller Aktionen. |
| Action | In Instanzen dieser Klasse werden alle für die (spätere) durchführung einer Aktion erforderlichen Informationen zusammengetragen. |
| AddCommand | Implementiert die folgende Aktion: Addieren oder substrahieren eines Werts zu/von einem Attribut[[11]](#footnote-11). |
| CreateCommand | Implementiert die folgende Aktion: Erzeugen eines neuen Feldinhalts (‚Inhabitanten‘). |
| DeployCommand | Implementiert die folgende Aktion: Verteilen eines Attributswertes auf die ‚Bewohner‘ der Nachbarzellen. |
| KillCommand | Implementiert die folgende Aktion: Leeren einer Zelle. |

1. Schon in der vorliegenden Form lag der Implementierungsaufwand bei circa 60 Stunden. Beim Funktionsumfang mussten daher leider gewisse Abstriche gemacht werden. [↑](#footnote-ref-1)
2. Die grafische Ausgabe ist nicht besonders elaboriert. So wäre bspw. eine Zoomfunktion sicherlich wünschenswert. Auch die Performance ließe sich durchaus noch verbessern. Doch auch darauf wurde aus Aufwandsgründen verzichtet. [↑](#footnote-ref-2)
3. Mindestens die JRE 1.6 [↑](#footnote-ref-3)
4. Das Projekt läßt sich in Eclipse compilieren. Um in Eclipse auch den Parser neu generieren zu können wird allerdings JavaCC benötigt. [↑](#footnote-ref-4)
5. Im Unterverzeichnis ZA/doc findet sich außerdem die mit javadoc generierten Klassendokumentation. [↑](#footnote-ref-5)
6. Da derzeit ohnehin nur ein einziger ZA definiert werden kann wäre es eigentlich gar nicht nötig, die Definition eines ZA so zu umschließen. Das Programm könnte aber so erweitert werden, dass auch mehrere ZA definiert werden können. Auch wäre die Definition anderer Konstrukte als ZA möglich. [↑](#footnote-ref-6)
7. Diese Bezeichnungen lehnen sich an das Game of life an. Gleichwohl sind die definierbaren ZA natürlich nicht auf das Game of life beschränkt. [↑](#footnote-ref-7)
8. Also die Zelle, die während der Simulation beim Erzeugen der nächsten Generation gerade untersucht wird. [↑](#footnote-ref-8)
9. Damit das Programm auch während der Simulation bedienbar bleibt wurde die Simulation in einen eigenen Thread ausgelagert. [↑](#footnote-ref-9)
10. Bei der Auswertung werden 0 als FALSE und alle anderen Werte als TRUE interpretiert. [↑](#footnote-ref-10)
11. Gemeint ist hier das einem ‚Inhabitanten‘ zugeordnete Attribut. [↑](#footnote-ref-11)