
Individuelles Projekt

# Verifikation und Validierung des Modellierungswerkzeuges 'Kiek/EcoScape'

Nina Marwede
9. Semester Diplom-Informatik

*30. März 2005*

Betreuer:
Prof. Dr. Michael Sonnenschein
Dr. Ingo Stierand

# Abstract

EcoScape is an object-oriented framework for the creation and execution of Hierarchic Asymmetric Cellular Automata, realized in C++ as part of a dissertation. Kiek is a graphical user interface for EcoScape, realized in Java by a student project group.
Investigations in the form of black-box tests have shown that the combined system Kiek/EcoScape falls short of the expectations. For example, a big class of hierarchical models can not be entered, the user guidance is partially mistakable, and the simulation provides reproducible wrong results resp. zero values.
The task of this Individual Project is the verification and validation of the framework and its user interface. Thereby the shortcomings of the software shall be analysed and corrected if possible, with the aim of fitting the expectations of a potential user.



# Zusammenfassung

EcoScape ist ein objektorientiertes Framework zur Modellbildung und -ausführung von Hierarchischen Asymmetrischen Zellularen Automaten, realisiert in C++ als Teil einer Diplomarbeit. Kiek ist eine in Java programmierte grafische Benutzeroberfläche für EcoScape, realisiert von einer Projektgruppe.
Untersuchungen in Form von Blackbox-Tests haben gezeigt, dass das Gesamtsystem Kiek/EcoScape die in es gesetzten Erwartungen nicht erfüllt. Beispielsweise lässt sich eine große Klasse von hierarchischen Modellen nicht eingeben, die Benutzerführung ist zum Teil missverständlich, und die Simulation liefert reproduzierbar falsche Ergebnisse bzw. Nullwerte.
Aufgabe dieses Individuellen Projektes ist daher die Verifikation und Validierung des Frameworks und seiner Oberfläche. Dabei sollen Schwächen der Software analysiert und möglichst behoben werden mit dem Ziel, die Erwartungen eines potenziellen Anwenders zu erfüllen.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1 Einleitung

Im Jahr 2001 stellten Michael Sonnenschein und Ute Vogel in ihrem Artikel “Asymmetric Cellular Automata for the Modelling of Ecological Systems” [SV01] eine neue Klasse von Zellularen Automaten vor, formal erweitert um den Aspekt der Asymmetrie; noch im selben Jahr erschien auch die Erweiterung um Hierarchie. Diese *Hierachischen Asymmetrischen Zellularen Automaten* (HAZA) eignen sich gut zur Modellierung von räumlich heretogenen Strukturen sowie Prozessen auf unterschiedlichen Raum- und Zeitskalen, wie sie beispielsweise in der Ökologie auftreten. Aufgrund ihrer Komplexität sind neuartige Werkzeuge zur Modellerstellung und Simulation erforderlich.

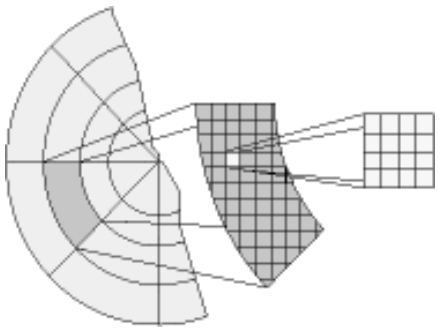

Abbildung 1: HAZA – schematische Struktur [Kie04]

Im Jahr 2002 entwickelte Bert Speckels seine Diplomarbeit mit dem Titel “Entwurf und Realisierung eines Frameworks für Hierarchische Asymmetrische Zellulare Automaten mit Anbindung an räumliche Datenbanken” [Spe02]. Dieses objektorientierte Framework wurde in C++ realisiert, ist in der Lage zur vollständigen und korrekten Abbildung und Berechnung von HAZA und enthält Mechanismen zur Modellbildung und Simulation. Es erhielt den Namen “EcoScape” und konnte mit einigen Beispielmodellen offenbar korrekt arbeiten – obgleich schon festgestellt wurde, dass die Überprüfung des korrekten Verhaltens aufgrund der Komplexität der Modelle relativ schwierig ist: Man kann sie nicht ‘mal eben auf Papier nachrechnen’, wie es bei traditionellen Zellularen Automaten noch möglich ist.

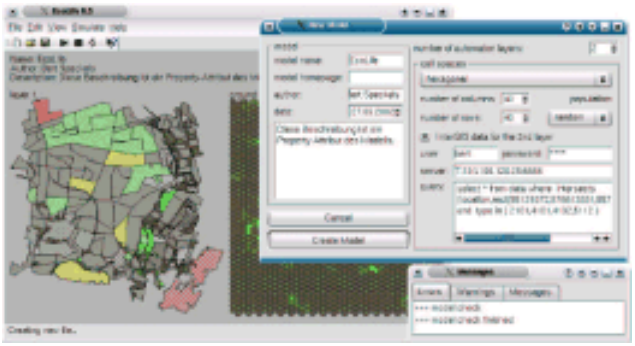

Abbildung 2: EcoLife – Bildschirmfoto [Spe02]

Im Jahr 2003 sollte die Projektgruppe “Entwicklungsumgebung für räumlich strukturierte ökologische und sozioökonomische Modelle” [Kie04], gebildet aus neun Studenten, innerhalb eines Jahres eine grafische Benutzeroberfläche (GUI) erstellen und dabei auf das Framework EcoScape aufsetzen. Die Entwicklungsumgebung wurde in Java programmiert und erhielt den Namen “Kiek”. Sie enthält Assistenten zur Unterstützung der Modellierung, eine eigene Skriptsprache zur freien Definition von Zustandsübergängen sowie eine Komponente zur Visualisierung und Auswertung der Simulation. Allerdings gab es von Beginn an Schwierigkeiten bei der Verarbeitung einiger Klassen von Modellen.

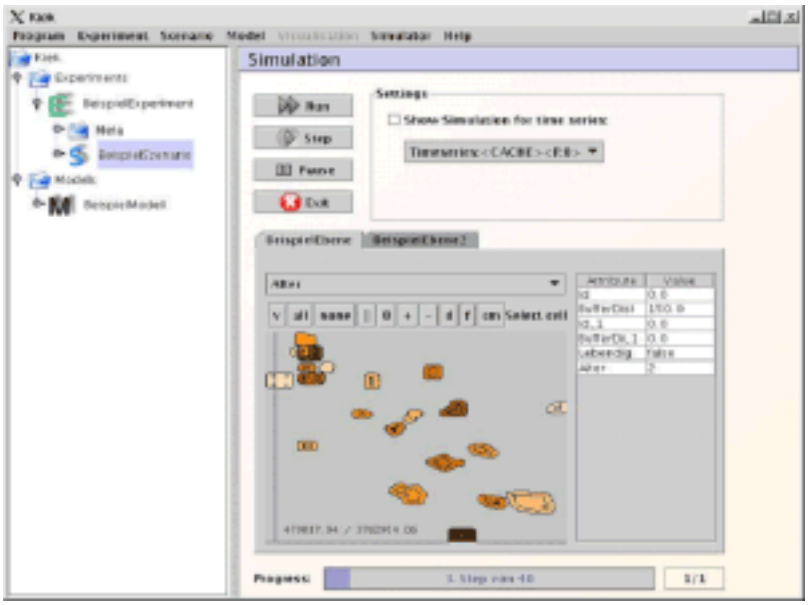


Abbildung 3: Kiek – Bildschirmfoto [Kie04b]

Im Jahr 2004 entdeckte Daniel Müller im Rahmen seines Individuellen Projektes “Evaluation und Erweiterung des Modellierungssystem Ecoscape/Kiek zur Landnutzungsmodellierung” [Mue04] anhand von Blackbox-Tests mit zwei eigens entwickelten Beispielmodellen konkrete Mängel in der Benutzerführung sowie in der Berechnung von Simulationsschritten. Sein Modell zur Winderosion scheiterte bei der Modellierung, die Simulation seines Springbrunnenmodells (zur Überprüfung der hierarchischen Funktionalität) lieferte reproduzierbar falsche Ergebnisse bzw. Nullwerte.

Das vorliegende Individuelle Projekt setzt genau an dieser Stelle an und soll die genauen Ursachen für die offensichtlichen Fehlfunktionen des Frameworks und seiner Benutzungsoberfläche entdecken und beheben. Außerdem soll als Methode der Qualitätssicherung durch Anwendung von systematischen Verifikations- und Validierungs-Prinzipien (V&V) die Wahrscheinlichkeit des Auftretens weiterer Fehlfunktionen verringert werden. Modelle, die der HAZA-Theorie entsprechen, sollen anhand einer zweckmäßigen, logischen Benutzerführung eingegeben und gespeichert sowie mit korrekten Ergebnissen simuliert und ausgewertet werden können.
Aufgrund des eingeschränkten Zeitrahmens ist die Festlegung eines exakteren Ziels schwierig. Falls die Ursachen für die aufgezeigten Fehlfunktionen sich schnell offenbaren würden, könnte relativ viel Zeit für die systematische Überprüfung der auf den ersten Blick fehlerlos funktionierenden Komponenten aufgewendet werden. Andererseits wäre es auch denkbar, dass trotz aller Mühe der Fehler zwar eingekreist würde, doch das Gesamtsystem immer noch nicht fehlerfrei wäre.

Bevor in Kapitel 3 vom praktischen Teil dieses Projektes berichtet wird, soll das Kapitel 2 einige theoretische Hintergründe und Möglichkeiten erläutern. Kapitel 4 enthält schließlich eine Zusammenfassung der gewonnenen Erkenntnisse.
Vorweg kann bereits gesagt werden, dass bei einem derartig komplizierten Stück Software wie dem mir gegebenen sich leider kaum eine der etablierten Strategien, die sich zumeist auf die Entwicklung eines Produktes ‘from Scratch’ konzentrieren, anwenden lässt. Insbesondere Prämissen wie “Test early, test often, test enough” [MS01] hinterlassen ein lachendes und ein weinendes Auge, wenn – salopp gesagt – das Kind bereits in den Brunnen gefallen ist.
Dies ist nicht als Anschuldigung gemeint; schließlich fand die Entwicklung der Software im universitären Umfeld statt, es sind eben keine Profis am Werk, sondern zumeist (im Sinne von Vorwissen, Talent und Motivation) heterogene Gruppen – und hinterher ist man immer schlauer.

# 2 Verifikation und Validierung in der Theorie

Dieses Kapitel enthält einen Überblick über den Prozess der Verifikation und Validierung (V&V), einige grundsätzliche Techniken sowie jeweils Überlegungen, ob und warum für die Bewältigung der Aufgabenstellung welche Verfahren schließlich angewandt werden. Aus der Betrachtung sind marktwirtschaftliche Aspekte im gröberen Sinne (z.B. die Organisation von Teams innerhalb von Firmenstrukturen) weitgehend ausgeschlossen, da sie für das vorliegende Projekt irrelevant sind.

Die Frage, warum überhaupt V&V, erübrigt sich im vorliegenden Falle, aber auch ganz allgemein hat sich im Laufe der Zeit mit zunehmender Komplexität der Programme herausgestellt, dass es wesentlich kostengünstiger ist, sich frühzeitig und in jeder Entwicklungsphase um die erwünschte Funktionalität zu kümmern, als im Nachhinein aufwändige Korrekturmaßnahmen durchzuführen. [Mye91]
Quellen für Fehler gibt es viele, wie sich im Folgenden auch zeigen wird. Angefangen beim Formulieren der Anforderungen eines Softwaresystems über die einzelnen Entwurfsschritte bis zur Implementierung und sogar beim Testen selbst können beispielsweise Missverständnisse entstehen, Übertragungen zwischen Dokumenten (z.B. Entwurf und Quellcode) fehlschlagen oder einfache Tipp- oder Copy/Paste-Fehler auftreten.

Hierzu gibt es zwei relativ berühmte Beispiele in der jüngeren Raumfahrtgeschichte:

- Es wurde der Fehler gemacht, Software von einer Umgebung ohne gründliche Prüfung in eine neue Umgebung zu portieren. Während des ersten Fluges der Rakete "**Ariane 5**" im Jahr 1996 läuft ein unnötiges Kalibrierungsprogramm für die Trägheitssensoren. Die gemessenen Werte der Ariane 5 überschreiten die in der Ariane 4-Software (welche auch in der Ariane 5 eingesetzt wurde) vorgesehenen Bereiche. Die Ausnahme wird durch Anhalten des Steuerungscomputers, der den Wert der Fehlermeldung fälschlicherweise als Messwert interpretiert, behandelt, um auf ein zweites, redundantes System umzuschalten, welches jedoch ebenfalls anhält. In Folge dessen muss die Rakete mitsamt einem 500 Mio. US-$ teuren Satelliten wenige Sekunden nach dem Start notgesprengt werden.

- Im Jahr 1999 verglüht die wissenschaftliche Raumsonde "**Mars Climate Orbiter**" im Wert von 125 Mio. US-$ beim Anflug auf den Mars in einen zu tiefen Orbit. Bei der Berechnung des erforderlichen Bremsschubes werden englische Maße und die zugehörigen metrischen Äquivalente verwechselt.

## 2.1 Begriffsklärung

Gemäß [Som01] ist die Verifikation und Validierung (V&V) ein "Prozess der Überprüfung und Analyse, der sicherstellt, dass eine Software mit ihrer Spezifikation übereinstimmt und die Bedürfnisse der zahlenden Kundschaft erfüllt".

Boehm beschrieb 1979 den Unterschied so:

- "Validierung: Erstellen wir das richtige Produkt?"
- "Verifikation: Erstellen wir das Produkt richtig?"

Die Validierung orientiert sich also an den Erwartungen eines Kunden, die Verifikation hingegen soll die Übereinstimmung des Produktes mit seiner Spezifikation sicherstellen.

Generelles Ziel von V&V ist der Nachweis, dass die Software "ihren Zweck erfüllt", d.h. nicht unbedingt fehlerfrei, aber "gut genug" für den beabsichtigten Zweck ist.

[Mye91] ist der Meinung, ein Mensch neige dazu, sich unbewusst auf ein Ziel auszurichten: Wenn er die Abwesenheit von Fehlern zeigen soll, dann neigt er zur Auswahl von Testdaten, die wahrscheinlich keinen Fehler aufzeigen. Er definiert:

> "Testen ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden."

Man stelle sich als Analogie einen Arzt vor, der etliche Labortests an einem sich krank fühlenden Menschen durchführt – wird keine Krankheit gefunden, nennen wir den Test 'nicht erfolgreich'; aber schlägt der Test an, dann ist er erfolgreich, weil der Arzt nun eine Behandlung einleiten kann.

Alle Quellen, die sich mit dem Thema beschäftigen, betonen die Wichtigkeit, mit dem Hinterfragen frühzeitig, schon bei der Validierung der Anforderungen, zu beginnen und den Prozess sich über den gesamten Lebenszyklus eines Systems erstrecken zu lassen, auch oder gerade weil sich manche Ungereimtheiten und Schwächen in den Anforderungen zuweilen erst bemerkbar machen, wenn das System vollständig implementiert wurde.

## 2.2 Grundlegende Vorgehensweisen

Bei **Inspektionen** werden Dokumente analysiert, dabei kann es sich um Programmcode oder Dokumentation handeln. Verfahren der formalen Verifikation sind einsetzbar. Die Analyse ist statisch und im Prinzip jederzeit machbar. Unterschieden werden manuelle und automatisierte Verfahren.

Beim **Testen** wird eine Implementierung ausgeführt und mit Testdaten gefüttert; analysiert werden Ausgabe und Verhalten. Die Analyse ist dynamisch, zur Durchführung ist zumindest ein Prototyp erforderlich.

**Validierungstests** zeigen die Entsprechung mit der Spezifikation – das System soll sich "bei der Ausführung vorgegebener Abnahmetestfälle richtig" verhalten. [Som01]

**Fehlertests** hingegen zeigen das Vorhandensein (*nicht* die Abwesenheit) von Fehlern und sollen Inkonsistenzen aufdecken. Um Fehlfunktionen zu provozieren, werden spezielle Testdaten erstellt, beispielsweise kritische Werte wie sehr große oder sehr kleine Zahlen oder Zeichenketten, Sonderzeichen oder ungewöhnliche Kombinationen.

**Statistische Tests** sollen normale und erhöhte Nutzungsbedingungen simulieren, unter denen das Produkt langfristig arbeiten soll. Daraus kann sich eine Abschätzung von Leistung und Zuverlässigkeit ergeben, beispielsweise Antwortzeiten und Anzahl der Benutzer in einer Mehrbenutzerumgebung.

Wichtig sind **Regressionstests**, mit ihrer Hilfe soll vermieden werden, dass durch Fehlerbehebung neue Fehler entstehen. [Mye91] behauptet sogar: "Die Wahrscheinlichkeit für die Existenz weiterer Fehler in einem Abschnitt eines Programms ist proportional zu der Zahl der bereits entdeckten Fehler in diesem Abschnitt."

Bei einem **Back-to-back-Test** werden das zu testende System und ein Referenzsystem (sog. "Orakel") parallel ausgeführt und die Unterschiede analysiert.

Zu beachten ist, dass V&V und Fehlerbehebung (Debugging) verschiedene Vorgänge sind, die nicht unbedingt Hand in Hand gehen müssen: Während ersteres die Fehler lediglich anzeigen und grob lokalisieren soll, kümmert sich letzteres um die Auffindung und Beseitigung. Hilfreich sind dabei vor allem persönliche Erfahrung und spezielle Werkzeuge (Debugger).

## 2.3 Planung

Beim Planen eines V&V-Prozesses sollten festgelegt werden:

- Das Verhältnis von statischen und dynamischen Ansätzen,
- Standards und Prozeduren für Inspektionen und Tests,
- Checklisten für Inspektionen,
- Testfälle für Tests.

Je kritischer (d.h. normalerweise sehr teuer oder Menschenleben-gefährdend) ein System ist, desto mehr Gewicht sollte auf die statische Verifikation gelegt werden. In einigen Fällen kann sogar der formale mathematische Beweis einer korrekten und zuverlässigen Funktionalität erforderlich werden. Da dies jedoch wiederum relativ teuer ist, wird das Verfahren nur bei (besonders im Un-Falle) entsprechend teuren Gesamtsystemen angewendet, Standardbeispiele sind Flugzeuge und Atomkraftwerke.

Der Testplan für ein kommerzielles Produkt könnte folgendermaßen aussehen. [Som01] Bestimmung von:

- Hauptphasen des Prozesses,
- Anforderungen der Benutzer,
- zu testenden Komponenten,
- Zeitplan,
- Protokollierung,
- HW/SW-Anforderungen,
- Randbedingungen (z.B. Personalknappheit).

Bei gehobenen Ansprüchen sollten nicht die Programmierer selbst testen, sondern unabhängige Teams, um die Wahrscheinlichkeit zu verringern, dass die Tests die gleichen Fehler wie das zu testende Programm haben.

## 2.4 Inspektionen

Bei einer Inspektion wird eine Quelltext-Darstellung eines Systems – z.B. Spezifikation oder Programmcode – untersucht. Inspektionen kommen ohne das Ausführen des Programms aus und können daher als Verifikationsverfahren bereits vor der Implementierung Anwendung finden.
Das Finden von Fehlern mit Hilfe von Inspektionen ist billiger als ausgiebiges Testen – [Som01] berichtet von einem Experiment 1987 und der Bestätigung 1993. Der Inspektionsprozess kann außer der Fehlersuche auch andere qualitative Eigenschaften betrachten, z.B. Portierbarkeit, Wartungsfreundlichkeit oder die Einhaltung von Standards. Im Unterschied zu anderen Arten der Qualitätssicherung liegt der Schwerpunkt allerdings im Finden von Fehlern, nicht in der Betrachtung übergreifender Entwurfsprobleme.

Inspektionen können die Übereinstimmung mit der Spezifikation überprüfen, jedoch nicht das dynamische Verhalten. Tests sind weiterhin unverzichtbar zur Beurteilung der Zuverlässigkeit, Geschwindigkeit, Benutzeroberfläche und Anforderungsentsprechung der Benutzer. Eine der wirksamsten Einsatzmöglichkeiten von Inspektionen kann die Prüfung der Testfälle eines Systems sein.

Während der 70er Jahre wurde bei IBM erstmals ein formalisierter Prozess der Programminspektion entworfen [Som01]. Dabei nimmt ein Team aus unterschiedlich erfahrenen Mitgliedern einen sorgfältigen, zeilenweisen Review des Quellcodes vor. Das Inspektionsteam sollte weder Vorschläge zur Fehlerbehebung machen noch Veränderungen an anderen Programmkomponenten empfehlen.

Voraussetzungen für die Programminspektion sind laut [Som01]:

- Spezifikation des Codes – Ohne Dokumentation ist kein Abgleich mit dem Code möglich.
- Vertrautheit der Inspektoren mit den zu prüfenden Kriterien, z.B. Firmenstandards, Checklisten üblicher Fehler.
- Syntaktisch korrekter Code – Es hat insbesondere keinen Zweck, Code zu inspizieren, der 'fast fertig' ist.

Eine Checkliste üblicher Fehler könnte etwa folgende Punkte abdecken:

- Variablen initialisiert/benutzt?
- Konstanten benannt?
- Verletzungen von Arraygrenzen?
- Trennzeichen für Zeichenketten?
- Pufferüberläufe?
- Bedingungen?
- Schleifenterminierung?
- Klammerung?
- Cases vollständig definiert?
- Breaks vorhanden wo nötig?
- Verwendung aller Eingaben?
- Zuweisung aller Ausgaben?
- Abfangen unerwarteter Eingaben?
- Anzahl/Typen/Reihenfolge der Parameter?
- Gemeinsamer Speicherzugriff?
- Zeigerneuordnungen bei Änderungen?
- Speicherallokation/-freigabe?
- Exceptions?

## 2.5 Automatisierte Analyse

Die automatisierte Analyse ist ein statisches Verfahren zur Überprüfung von Programmcode. Dabei kommen spezielle Werkzeuge mit Expertenwissen über die jeweilige Syntax und mögliche Fehler zum Einsatz. Vorteile gegenüber der manuellen Analyse sind vor allem Geschwindigkeit und Präzision des Ablaufs – Nachteil ist, dass einige Fehler, insbesondere solche basierend auf Verständnisproblemen, nicht gefunden werden können. Daher wird der Einsatz automatischer Verfahren in der Praxis nicht alleinig, sondern als eine unter mehreren Möglichkeiten, und zwar relativ frühzeitig eingesetzt. Somit muss man sich bei den Reviews nicht mit Banalitäten abgeben und kann sich auf die wichtigen Dinge konzentrieren.

Als Phasen der automatischen Analyse benennt [Som01]:

1. Steuerungsablauf: Schleifen mit mehreren Abbruch- und Eintrittspunkten, unerreichbarer Code,
2. Datenverwendung: Verwendung von Variablen...
3. Schnittstellen: Konsistenz und Verwendung
4. Informationsfluss: Abhängigkeiten zwischen Eingabe und Ausgabe
5. Pfade: Programmsteuerung

Typische Prüfpunkte für eine automatische Analyse stimmen weitgehend mit der Checklisten für die manuelle Analyse überein:

- Variablen initialisiert/benutzt
- wiederholte Zuweisungen ohne zwischenzeitliche Benutzung
- Verletzungen von Arraygrenzen?
- unerreichbarer Code
- unbedingte Schleifen
- mehrfache Ausgabe ohne zwischenzeitliche Zuweisung
- Parametertypen/anzahl
- nicht benutzte Rückgaben
- nie aufgerufene Funktionen/Prozeduren
- nicht zugewiesene Zeiger
- Zeigerarithmetik

Viele der Kriterien sind heutzutage im Compiler integriert, neuere Sprachen wie Java (bzw. deren Interpreter) erlauben einfache Fehler (z.B. fehlende Initialisierung, Verletzung von Arraygrenzen, Speicherverwaltung) überhaupt nicht.

Ein Beispiel für ein traditionelles spezialisiertes Analysewerkzeug ist "LINT" für die Sprache C – mit C++ kommt es leider nicht zurecht.

## 2.6 Cleanroom

Die Cleanroom-Philosophie (abgeleitet von den 'Reinräumen' in der Halbleiterproduktion) strebt die Entwicklung vollkommen fehlerfreier Software durch rigorose Inspektionen an. Aufwand und damit Kosten für Tests sinken dadurch drastisch.

Voraussetzung ist die *formale Spezifikation* der zu entwickelnden Software.

Die Entwicklung verläuft streng *inkrementell* – während eines Inkrements werden Änderungen der Spezifikation abgelehnt, und jedes Inkrement wird an den Kunden ausgeliefert. Kritische Funktionalität soll frühzeitig implementiert werden.

*Strukturierte Programmierung* kommt zum Einsatz, d.h. Entwicklung durch schrittweise Verfeinerung der Spezifikation – man könnte auch sagen, es wird viel Wert auf 'schönen' Code gelegt.

Die Software wird unter Anwendung rigoroser Inspektionen *statisch verifiziert*.

Auf Basis von Einsatzprofilen soll die Zuverlässigkeit durch *statistische Systemtests* ermittelt werden.

[Som01] merkt an, dass die Cleanroom-Entwicklung funktioniert, wenn sie von gut ausgebildeten, entschlossenen Entwicklern angewendet wird. Bisher findet ihre Verbreitung allerdings nur langsam statt, Erfolgsberichte stammen zumeist von den Erfindern selbst.

## 2.7 Blackbox-Tests, Testfälle und Äquivalenzklassen

**Blackbox-Tests** tragen ihren Namen, weil der innere Aufbau der zu testenden Komponenten unbekannt ist. Das Verhalten des Systems kann nur durch Untersuchung von Eingaben und der zugehörigen Ausgaben festgestellt werden. Diese Vorgehensweise wird auch "funktionales Testen" genannt, wegen der Beschäftigung mit der Funktion, nicht der Implementierung: alle Tests können nur von der Dokumentation abgeleitet werden. Ein Schlüsselproblem ist die Auswahl der Eingaben. Wenn man zu wenig oder die falschen Dinge überprüft, wiegt man sich schnell in falscher Sicherheit – das ist mitunter gefährlicher, als gar nicht zu testen.

**Testfälle** sind Spezifikationen der Testeingaben und der vom System zu erwartenden Ausgaben und enthalten eine Aussage, was damit getestet werden soll. Um bei komplexen Systemen auf eine Teilmenge aller Testfälle verzichten zu können, müssen die Testfälle den einzelnen Programmfunktionen zugeordnet werden. Aber auch bei sehr einfachen Programmen ist es nicht sinnvoll, *sämtliche* möglichen Eingaben und ihre Kombinationen

zu testen, sondern nur eine ausgesuchte Teilmenge. Welche Eingaben in Frage kommen, wird mit Hilfe der Dokumentation und individueller Erfahrung entschieden.

Ein paar Faustregeln helfen, die richtigen Testfälle auszuwählen. [GUV03] Für jeden von einer Funktion akzeptierten Datentyp gibt es Werte, die es zu betrachten lohnt:

- Bei ganzzahligen Typen wie int und long sind das etwa 0, 1, 2, ihre negativen Werte sowie die Konstanten `MAX_VALUE` und `MIN_VALUE`.
- Für Zeichenketten-Parameter sollte man wie auch bei anderen Objekt-Typen überprüfen, ob die Funktion auf die Übergabe von `null` wie vorgesehen reagiert. Der Leerstring `""` ist immer einen Test wert, ebenso wie ein String mit nur einem Zeichen, ein Steuerzeichen wie `"\n"` und ein sehr langer String.

Zur Identifikation weiterer lohnender Testfälle betrachtet man den gesamten Wertebereich, den eine Funktion entgegennimmt. Er lässt sich meist in **Äquivalenzklassen** unterteilen, von denen man erwartet, dass sie für die Fehlersuche identische Ergebnisse liefern. Die Testfälle wählt man aus den Grenzen dieser Bereiche aus.
[GUV03] meinen weiterhin, Schwierigkeiten mit der Bestimmung von (einfachen) Testfällen deuten auf eine Designschwäche und empfehlen in solchen Fällen eine Refaktorisierung des Quelltextes.

## 2.8 Whitebox-Tests, Überdeckungen

Das Gegenteil zum Blackbox-Testen ist ein Blick in den Code (wenn denn verfügbar) – genannt **Whitebox-Test** oder auch "strukturelles Testen". Im Vordergrund steht hierbei die Kenntnis der Softwarestruktur und Implementierung. Es sollen möglichst viele Programmteile durch die Menge an Testfällen abgedeckt werden. Basierend auf dem gleichen Kontrollflussgraphen gibt es Varianten der Überdeckung: [SK1]

- Ziel der *Anweisungsüberdeckung* ist die Ausführung aller Anweisungen jeweils mindestens einmal. Doch bei Erreichen solcher 100%igen Überdeckung kann es nicht getestete Zweige im Graphen geben. Andersherum kann es in Fällen von unerreichbarem Code dazu kommen, dass 100können.
- Ziel der *Zweigüberdeckung* ist die Ausführung aller Zweige (Wegabschnitte) im Graphen, eingeschlossen sind auch z.B. leere Codeabschnitte einer If-Else-Konstruktion.
- Ziel der *Pfadüberdeckung* ist die Ausführung aller Pfade (Gesamtwege) im Graphen. In der Praxis ist dies oft nicht möglich oder sinnvoll, z.B. bei Schleifen. Trotz der großen Anzahl an Tests kann auch mit dieser Variante nicht sichergestellt werden, dass alle Kombinationen von Eingaben geprüft werden.

Offensichtlich ist das Whitebox-Testen verwandt mit der automatisierten Inspektion. An seine Grenzen stößt das Verfahren bei Grafik-orientierten Benutzungsoberflächen und paralleler Ausführung von Komponenten.

## 2.9 Integrationstests

Üblicherweise gegen Ende des Testprozesses finden Integrationstests statt, die vorab bereits einzeln geprüfte Komponenten innerhalb eines Gesamtsystems. Demzufolge liegt der Schwerpunkt der Tests auf den Schnittstellen zwischen den Teilsystemen, der Wechselwirkung zwischen Objekten; dortige Fehler sind beim Testen einzelner Objekte nicht zu entdecken. Je 'schneller' die Integration geschieht, desto schwieriger wird es, einen aufgedeckten Fehler zu lokalisieren, daher sollte ein sukzessiver Aufbau des Gesamtsystems erfolgen.

Für diesen Aufbau gibt es zwei entgegengesetzte Strategien, die in der Praxis allerdings nur selten in Reinform angewandt werden.

**Top-down** bedeutet, mit Integration und Test der übergeordneten Komponenten zu beginnen. In der Architektur darunter liegend befinden sich sogenannte "Stümpfe" (engl. Stubs), deren Aufgabe jeweils die primitive Nachbildung einer Komponente ist – anstelle der Berechnung einer komplexen Funktion könnte eine Konstante zurückgegeben werden. Vorteil dieser Strategie ist die frühzeitige Einsatzfähigkeit im Entwicklungsprozess, der Kunde kann die Benutzungsoberfläche begutachten. Nachteil ist die meist umständliche Erzeugung von Testergebnissen, es müssen Stubs geschrieben werden, und die oberen Schichten müssen zur Produktion von verwertbaren Ergebnissen gezwungen werden.

**Bottom-up** bedeutet, mit Integration und Test der zugrunde liegenden Komponenten zu beginnen. In der Architektur darüber liegend befinden sich sogenannte "Treiber" (engl. Driver), deren Aufgabe die Simulation der Umgebung und die Benutzung der zu testenden Funktionen ist.
Vorteil dieser Strategie ist die einfachere Testbarkeit. Nachteil ist häufig, dass die Ausgabe der Testergebnisse aufgrund unfertiger Benutzungsaberfläche zusätzlich erzeugt werden muss.

[Som01] unterscheidet drei Fehlerklassen:

1. Falsche Verwendung einer Schnittstelle, z.B. bezüglich Parametertyp oder -anzahl,
2. Missverständnis des Verhaltens,
3. Synchronisationsfehler in Echtzeitsystemen.

Als Richtlinien für Schnittstellentests werden genannt:

- Aufrufe an externe Komponenten auflisten, jeweils Grenzwerte (→ Äquivalenzklassen) testen,
- Null-Zeiger testen,
- aufzurufende Komponenten mit Tests versagen lassen, um Missverständnissen vorzubeugen,

- Belastungstests bei Nachrichtenübergabe in Echtzeitsystemen,
- Belastungstests mit großen Datenmengen (Ziel ist ein möglichst "sanfter" Ausfall ohne Datenverlust),
- Variation der Aktivierungsreihenfolge von Komponenten, z.B. zur Suche nach Fehlern in gemeinsamen Daten,
- die Verwendung typisierter Sprachen und statischer Analysewerkzeuge kann vorbeugend wirken.

Beim Testen von Objekten im Vergleich zu einfachen Funktionen wirkt sich neben der erhöhten Komplexität manchmal problematisch aus, dass es keine klare Systemhierarchie gibt, Objekte können nur lose verbunden sein. Ebenfalls typisch für objektorientierte Systeme ist die Wiederverwendung von Komponenten, die nicht immer im Quelltext vorliegen. Des Weiteren wirken Vererbungs- und Überladungs- Strukturen erschwerend auf den Testprozess.
Hilfreich hingegen kann die Betrachtung von Anwendungsfällen oder Szenarien sein, ebenso wie die Betrachtung des Weges von Ereignissen durch das System – hierbei erweisen sich Sequenzdiagramme als nützlich.

## 2.10 Unit-Tests

Im Unterschied zu den Testverfahren im größeren Maßstab untersuchen Unit-Tests stets einen möglichst kleinen, für sich allein funktionierenden Code-Ausschnitt, z.B. eine einzelne Methode oder eine Klasse. Für jedes Stück Code (bis auf Trivialitäten wie get/set-Methoden) muss also eine eigene Testroutine geschrieben werden – somit lässt sich jedoch die Fehlersuche automatisieren und wesentlich beschleunigen. [GUV03]
Auch wenn ein Test niemals fehlschlägt, ist dies hilfreich für den Testprozess. Bei der Entwicklung der Tests fallen insbesondere schlecht entworfene Schnittstellen auf.
Die Prüfroutine eines Unit-Tests füttert etwa eine Methode mit Testdaten und spielt einfache Szenarien durch. Beim Implementieren eines Tests stellt der Entwickler durch festgeschriebene Abfragen sicher, dass das Programm für jede Eingabe (→ Äquivalenzklassen) das erwartete Ergebnis liefert. Werkzeuge wie JUnit[1] unterstützen mit einem einfach gehaltenen Framework.

Unit-Tests werden schon länger für die aufwendige Qualitätssicherung besonders sicherheitskritischer Software eingesetzt. Vergleichsweise neu ist aber die Idee, dass Programmierer solche Tests schon während der Implementierung ihres eigenen Codes anwenden. Diese Art der Unit-Tests wird deswegen auch Entwicklertest genannt.
Im Extremfall implementiert der Entwickler die Tests sogar schon vor den eigentlichen Anwendungsmethoden (Test First, Test Driven Design).

[1] `http://junit.org/`

Unit-Tests fördern quasi nebenbei einen modularen Aufbau der Anwendung. Die ausformulierten Testroutinen dienen zugleich als Dokumentation: Fremde Entwickler haben damit ein konkretes Programmbeispiel zur Hand.

Bei der Arbeit mit Unit-Tests stößt man jedoch auch an Grenzen. Für manche Dinge lassen sich Tests nur schwer programmieren, etwa für Bedienoberflächen oder nebenläufigen Code, bei dem mehrere Threads parallel ablaufen.
Besonders schwierig ist es, mehrschichtige Anwendungen zu testen, die etwa auf Datenbanken oder andere externe Systeme zugreifen. Wer diese Abhängigkeiten nicht auflöst, testet mit den Unit-Tests nicht nur die eigene Software, sondern auch die fremden Systeme samt Dateisystem und Netzwerkverbindung.

## 2.11 Bewertung und Vorauswahl der Strategien

Von vornherein distanzieren will ich mich von der *Validierung*. Es ist unwahrscheinlich, dass genug Zeit bleibt, um Konzepte oder Einsatzbereiche zu hinterfragen – Priorität hat die Fehlersuche bzw. *Verifikation* der gewünschten Funktionalität.

Streng systematische *Inspektionen* des gesamten Quelltextes wären sicher sinnvoll, allerdings sprechen mangelnde 'Manpower' und die starke Objektorientierung dagegen, sodass im Endeffekt nur wenige Komponenten jeweils genau dann inspiziert werden, wenn es Hinweise darauf gibt, dass sie Fehler enthalten.

*Validierungstests* stellen das übergeordnete Ziel dar; durch Implementierung und Simulation von speziellen Beispielmodellen sollen möglichst viele der im 'Normalbetrieb' genutzten Funktionalitäten geprüft werden.

*Fehlertests* kommen hauptsächlich temporär zum Einsatz. Sie sollen Fehler aufzeigen – da ein Hauptziel des Projektes die Fehlerbehebung ist, werden die meisten Fehlertests ihre Aufgabe schnell erfüllt haben und nicht mehr anschlagen.

*Statistische Tests* sind nicht hilfreich zur Bewältigung der vorliegenden Aufgabe. Sie helfen bei der Abschätzung von Leistung und Zuverlässigkeit, aber erstens wird Kiek/EcoScape wahrscheinlich nicht in kritischer Umgebung eingesetzt werden, und zweitens sollte zunächst eine halbwegs erwartungsgemäße Funktionsweise gewährleistet sein.

*Regressionstests* werden insofern durchgeführt, dass alle Testmodelle nach jeder Änderung der Quelltexte erneut ausgeführt werden, um die Entstehung neuer Fehler zu vermeiden.

*Back-to-back-Tests* sind zunächst nicht vorgesehen, doch schließlich bietet sich der direkte Vergleich zweier Testmodelle mit und ohne Verwendung einer bestimmten Komponente an, und zwar in Form von Validierungstest mit paralleler Inspektion.

*Planung* ist kaum nötig – da konkretes Fehlverhalten vorliegt und nur eine einzelne Arbeitskraft zur Verfügung steht, werden die Symptome mehr oder weniger der Reihe nach abgearbeitet. Beispielsweise ist eine Rollenteilung zur Trennung von Inspektion und Fehlerbehebung nicht möglich.

Eine *Checkliste* häufiger Fehler wird nicht speziell erstellt, gesunder Menschenverstand und Erfahrung des Inspekteurs werden als hinreichend eingeschätzt; außerdem werden bereits relativ komplizierte Fehler vermutet.

Auf eine *automatisierte Analyse* mit speziellen Werkzeugen wird mangels Kenntnis verzichtet; die Fähigkeiten der verwendeten Compiler werden als diesbezüglich hinreichend leistungsfähig eingeschätzt, und für die vermuteten komplizierten Fehler sind andere Vorgehensweisen angebracht.

So vielversprechend der *Cleanroom*-Prozess auch klingt, kommt seine Anwendung doch nicht infrage, weil die Voraussetzungen 'formale Spezifikation' und 'in der inkrementellen Entwicklung befindliches Produkt' nicht erfüllt sind.

*Blackbox*-Tests kommen zwangsweise dort zum Einsatz, wo kein Code vorhanden ist (Parsergenerator), oder wo er aufgrund der Schnittstelle (Java/C++) nur schwer einsehbar ist.

*Äquivalenzklassen* finden implizite Verwendung an einigen Stellen – für eine lückenlose Überprüfung fehlt die Zeit.

*Whitebox*-Tests können ebenfalls nicht systematisch durchgeführt werden; unter anderem wird eine Anweisungsüberdeckung an der wichtigsten Schnittstelle (Java/C++) angestrebt.

*Integrationstests* bilden den Startpunkt: Durch das Scheitern der Testmodelle aus einem vorhergehenden Projekt ist die Aufgabenstellung für dieses Projekt erst entstanden.
*Top-down* ist dabei der erste Ansatz: Die Modelle werden in Kiek erstellt, im Java-Kern verarbeitet und über die JNI-Schnittstelle an EcoScape weitergereicht. Die Ergebnisse sind fehlerhaft, die Ursache ist schleierhaft.
Daher soll nun per *Bottom-up*-Prinzip zunächst die Korrektheit der unteren Schicht (EcoScape) gezeigt werden, bevor die JNI-Schnittstelle, der Java-Kern und die grafische Oberfläche mit einbezogen werden. Wie bereits in Abschnitt 2.9 erwähnt, werden Treiber entwickelt, welche die jeweils darüber liegende Schicht simulieren und die zu testenden Komponenten benutzen sollen; die Ausgabe der Testergebnisse muss zusätzlich erzeugt werden.
*Schnittstellentests* erfolgen dabei eher implizit als systematisch, etwa durch Variation der Aktivierungsreihenfolge von Komponenten.

Für *Unit-Tests* ist es eigentlich zu spät – ähnlich dem Cleanroom-Prozess ist dieser Ansatz während der Entwicklung zu bevorzugen. Eine Nachrüstung wäre relativ aufwändig und wenig sinnvoll, denn eine wichtige Eigenschaft des zu prüfenden Codes ist: Es kommen keine komplizierten Algorithmen zum Einsatz, unübersichtlich ist vor allem die Datenhaltung und die Konstruktion von Objekten sowie deren Zustandsübergänge.

Als allgemeines Problem stellt sich die mangelhafte Kommentierung der Quelltexte heraus – oft ist die gewünschte Funktionalität oder die Interpretation der Rückgabewerte nur schwer ersichtlich. Aufgrund mangelnder persönlicher Erfahrung mit bestimmten Konstruktionen ist die Korrektheit des Quelltextes an einigen Stellen durch bloßes Ansehen nicht entscheidbar.

# 3 Vorgehensweise im Detail

In diesem Kapitel ist aufgelistet, welche Schritte zur Analyse und Behebung der bereits aufgezeigten Mängel am System Kiek/EcoScape unternommen wurden und welche neuen Probleme währenddessen zusätzlich aufgetreten sind.
Zunächst wird ein kurzer Blick auf die Ausgangsproblematik, im Endeffekt ein riesiges Missverständnis, geworfen, sodann geht es auch schon ans konkrete Debugging und Testen.

## 3.1 Ausgangsproblematik

Im Endstadium der Projektgruppe wurde den Teilnehmern klar, dass irgendetwas gewaltig schief gelaufen war: Die Hierarchischen Asymmetrischen Zellularen Automaten konnten zwar mit der grafischen Benutzungsoberfläche von 'Kiek' modelliert werden, aber eine ganze Klasse von ihnen, und zwar ausgerechnet jene mit der kennzeichnenden Eigenschaft 'Hierarchie', wollten bei der Simulation partout keine anderen Werte als `null` bzw. `false` liefern, wo immer eine Kommunikation zwischen verschiedenen Ebenen des Automaten angedacht war. Obwohl fast durchgängig während der Implementierungsphase viel mit Diskussionen über dieses Thema verbracht worden war, ist man sich bis zum Schluss nicht einig geworden, wie die Vorgaben von EcoScape und seiner Dokumentation umzusetzen seien. Falls damals jemand die Lösung kannte, so war er nicht ausreichend durchsetzungsfähig, um die Gruppe auf den richtigen Weg zu stoßen.

Weitere Mängel des Systems – eingeschränkte, verwirrende Modellierung in Kiek, Abstürze der Java Virtual Machine unter Windows beim Start des Simulators – bekommen eine vergleichsweise niedrige Priorität zugewiesen.

### 3.1.1 Unterschiedliche Verständnisse der Realisierung von 'Hierarchie'

Auf Seite 65 der **EcoScape-Diplomarbeit** [Spe02] befindet sich die Abbildung 7.12 "Zustandsübergang: Zustände und Zustandsabbildungen", die aus Sicht einer einzelnen Automatenzelle den Datenfluss zur Berechnung des folgenden Zustands darstellt. (Der erläuternde Text ist leider nicht eindeutig, zu welcher Ebene die jeweils erwähnten Zustände gehörig sind.)

Zu beachten ist die Symmetrie: sowohl 'upperState' als auch 'lowerState' befinden auf der gleichen Ebene wie der 'cellState' der aktuell betrachteten Zelle. In einem hypothetisch dreischichtigen Beispielautomaten besäße die obere Ebene einen lowerState, die untere Ebene einen upperState und die mittlere Ebene beides. Das Konzept lautet: die jeweils anderen Ebenen stellen über ihre upper- bzw. lowerMapper-Funktionen die Werte in upper- bzw. lowerState zur Abholung durch den betrachteten cellMapper bereit.

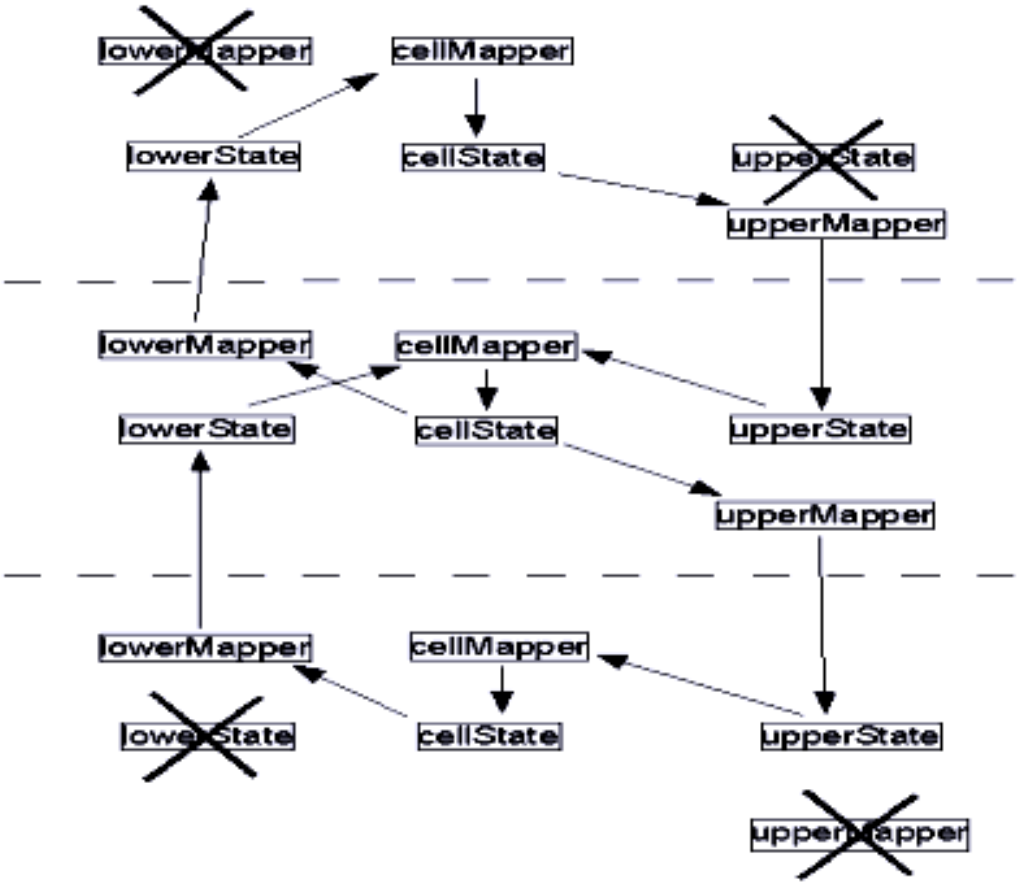


Abbildung 4: Hierarchie-Konzept – EcoScape-Diplomarbeit

Bei der Lektüre des EcoScape-Tutorials **EcoLife** bemerkt der wachsame Leser, dass alle Ebenen außer der untersten einen upperState und upperMapper haben, während alle Ebenen außer der obersten über einen lowerState und lowerMapper verfügen – dies ist ein direkter Widerspruch zur Diplomarbeit!

Der **EcoScape-Check-Code** (Methoden wie `scpModel::check()`) offenbart die Erwartung von EcoScape, dass alle Ebenen außer der untersten einen upperState haben, während alle Ebenen außer der obersten einen lowerState haben sollen – dies entspricht dem Tutorial und widerspricht damit ebenfalls der Diplomarbeit.

Wirft man daraufhin einen Blick in den **EcoLife-Original-Code** (→ `ecolifedoc.cpp`), kann man feststellen, dass der dortige 'ground-layer' einen lowerState bekommt, aber keinen upperState, während alle anderen Ebenen sowohl upper- als auch lowerStates bekommen...

Hieraus könnte man den Eindruck gewinnen, dass sich der Autor bei der Entwicklung selbst nicht so ganz einig war, wie es denn werden sollte; vermutlich hat er sogar sein ursprüngliches Konzept verworfen und daraufhin nicht die gesamte Dokumentation angepasst.

Zu Beginn der Projektgruppe wurde jemandem die Aufgabe zugeteilt, sich mit der Diplomarbeit zu beschäftigen und den anderen Mitgliedern im Rahmen eines **Seminars** [Kie04] die wesentlichen Punkte zu erläutern. Zur Erklärung der Zustandsübergangsberechnung wird die bereits genannte Abbildung 7.12 aus der Diplomarbeit herangezogen, doch der begleitende Text lässt bereits Zweifel vermuten. So heißt es zunächst "Als Zielzustand wird der lowerState der Ebene gewählt.", woraus nicht hervorgeht, welche Ebene gemeint ist.
"Zur letztendlichen Berechnung des neuen Zellzustandes, werden dem cellStateMapper [...] der gepufferte upperState einer evtl. vorhandenen Elternzelle als Eingabe übergeben." – Würde dies nicht bedeuten, der upperState wäre doppelt vorhanden: einmal in der oberen Zelle, aber nochmal 'gepuffert' in der eigenen Zelle?
"Zum Abschluss werden die out- und upperStates der bearbeiteten Zellen für die nächste Berechnung aktualisiert." – Hier plötzlich eine Asymmetrie! Warum nicht auch den lowerState aktualisieren? Es wird sich als korrekt herausstellen, aber die Ursache für diese Erkenntnis bleibt verborgen.

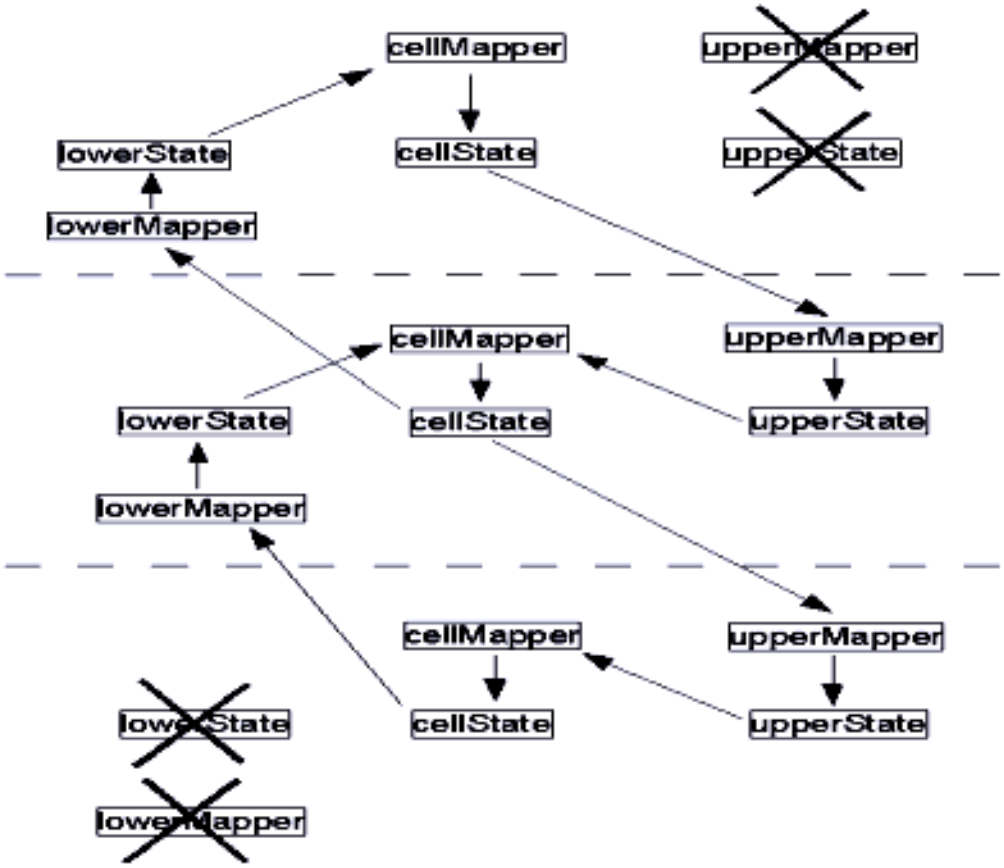


Abbildung 5: Hierarchie-Konzept – Projektgruppe

Auf der Basis des Seminars und der unterschiedlich gründlichen Lektüre der Diplomarbeit durch einige Projektgruppenmitglieder entstand der **Kiek-Model-Wizard**. Obwohl die mit seiner Hilfe konstruierten Modelle in sich konsistent sind, widersprechen sie dennoch allen bisher vorgestellten Konzepten, denn hier sammelt jede Ebene sich selbst die benötigten Werte. Im gedachten dreischichtigen Automaten enthält die obere Ebene einen lowerMapper, die untere einen upperMapper und die mittlere Ebene beides. Die Zuteilung der Zustände erfolgt genau wie in der Diplomarbeit und somit symmetrisch.

Im Zuge der Entwicklung seiner Evaluationsmodelle (→ **Springbrunnen**) hat sich der Autor des vorangehenden Individuellen Projektes [Mue04] noch ein weiteres Konzept einfallen lassen, dessen entscheidendes Merkmal die Asymmetrie ist: Alle Ebenen enthalten sowohl upper- und lowerState als auch upper- und lowerMapper, bis auf die unterste Ebene, sie enthält nichts dergleichen. Im Prinzip könnte man hier sagen: der upperState wird von der oberen Ebene zur Abholung durch den darunter liegenden cellState bereitgestellt, den lowerState als Aggregation der darunter liegenden cellStates hingegen müssen sich die Zellen bei ihrem Zustandsübergang jeweils selbständig einsammeln.

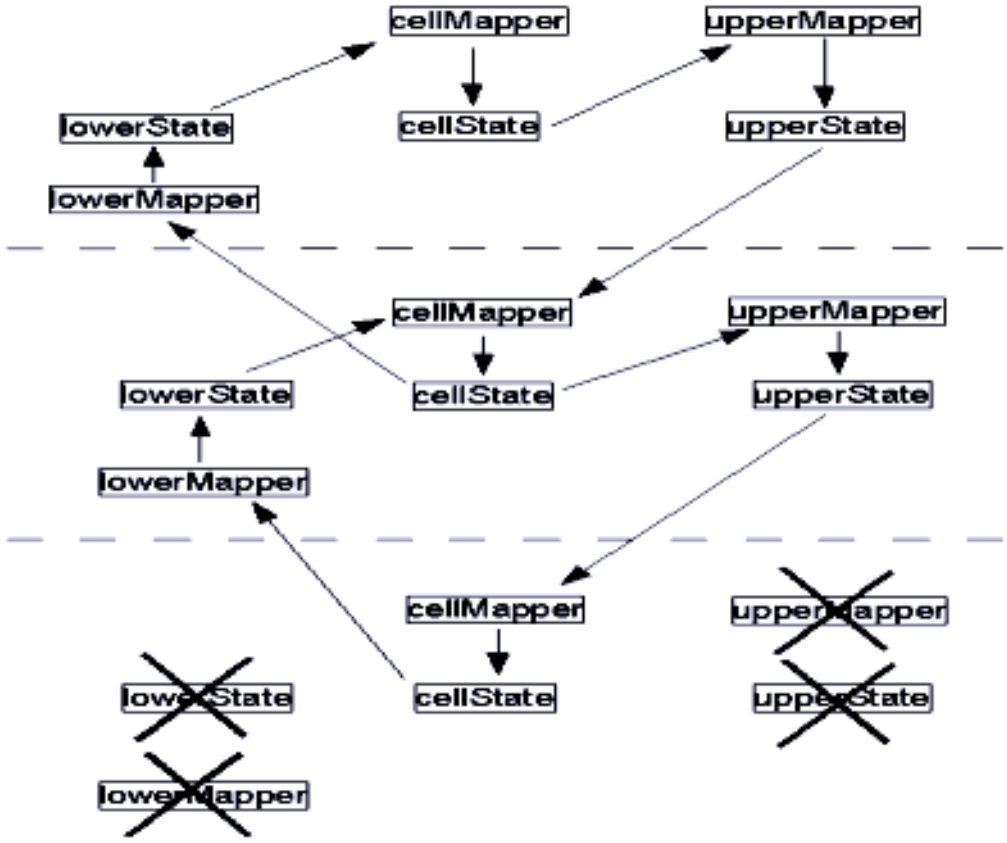


Abbildung 6: Hierarchie-Konzept – Springbrunnen-Testmodell

### 3.1.2 Folgerungen

Um nun endlich das ‘richtige’ Hierarchie-Konzept festzustellen, soll das Testmodell “Springbrunnen” [Mue04] implementiert werden, und auch das EcoScape-Tutorial Eco-Life ist einen Blick wert. Da ein relativ großer konzeptioneller Fehler offenbar schon auf unterster Ebene vorliegt, erscheint ein Wechsel von der Top-down-Strategie (→ Test-modelle in Kiek) zum Bottom-up-Verfahren angebracht. Dies bedeutet zunächst einen Verzicht auf die “Desktop-Applikation”, auf den “Kiekcore” samt Import/Export und auf den “JNI-Wrapper”, bietet somit aber mit der Beschränkung auf “EcoScape” den Vorteil einer reinen C++-Umgebung.

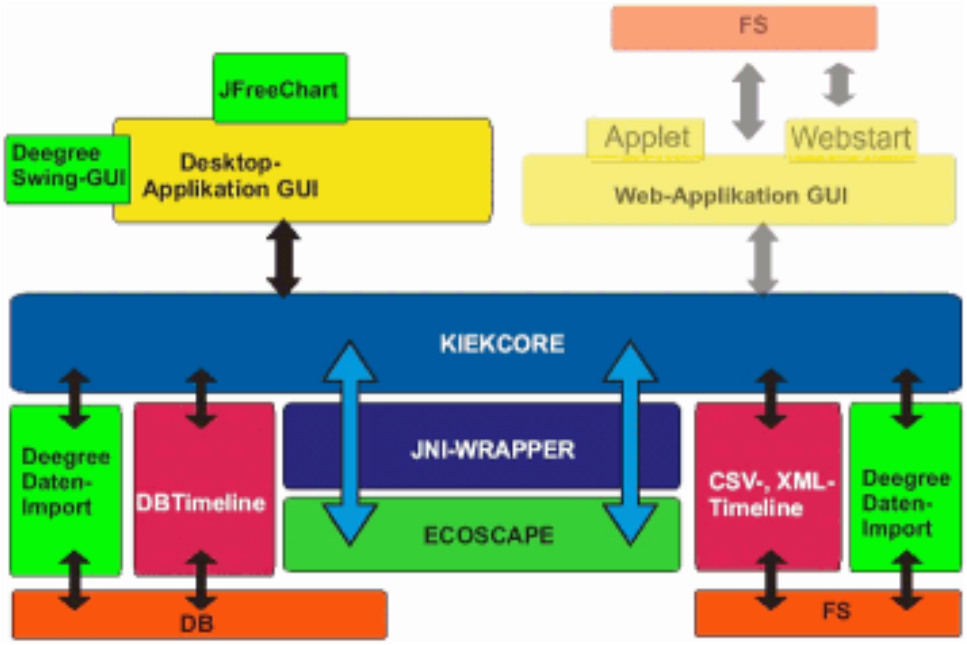


Abbildung 7: Kiek – Architektur [Kie04b]

## 3.2 Vorbereitungen für das Testen

Für einen gründlichen praktischen Test der Modelle ist es sinnvoll, die jeweils erforderlichen Komponenten in einer Debugger-Umgebung auszuführen. Dies erweist sich als problematischer als zunächst angenommen, da sämtliche C++-Komponenten während der Projektgruppe ohne grafische Umgebung entwickelt worden sind – sie lassen sich zwar in einer Unix/Linux-Konsole per `Makefile` kompilieren, doch die Übertragung in ein Projekt einer Entwicklungsumgebung bzw. Portierung nach MS Windows erfordert einigen Aufwand.
Die Alternative zur Entwicklungsumgebung wäre ein Debugging auf ‘printf’-Basis, bei dem die Werte von Variablen an verdächtigen Code-Positionen ausgegeben werden –

[Mye91] bezeichnet dies als die zweit-ineffizienteste Methode nach der Fehlersuche im Speicherauszug (Dump).
Aufgrund von Hardware-Problemen der für dieses Projekt bereitgestellten Linux-Maschine (innerhalb von zwei Monaten wurden nacheinander Mainboard, Festplatte und Netzteil getauscht) beginnt das Testen auf MS Windows.

### 3.2.1 EcoScape mit MS Visual C++ 6.0

Mit aktuellen Versionen des Quellcodes werden die bereits während der PG gesammelten Erkenntnisse [Kie04] genutzt, diese werden hier noch einmal in Kurzform dargelegt.

Das Anlegen eines Projektes mit bereits bestehenden Dateien erfolgt problemlos; es müssen lediglich sämtliche Quellcode-Dateien dem Projekt hinzugefügt werden. Im Fall von EcoScape entstehen drei Projekte mit den drei Komponenten db, model und simulator, Ziel ist jeweils eine dynamische Bibliothek (DLL).
Das Kompilieren einiger Klassen scheitert jedoch zunächst:

- An mehreren Stellen wird der Bezeichner `list` verwendet – für diesen Compiler offenbar ein reserviertes Wort. Es wird provisorisch gegen `list_` ausgetauscht. Die Methoden sind im einzelnen:
  - `scpModelBase::setMessageList()`
  - `scpLatticeCellSpace::cellsInRect()`
  - `scpLatticeCellSpace::cellsAtPoint()`
  - `scpGeoCellSpace::cellsInRect()`
  - `scpGeoCellSpace::cellsAtPoint()`
- Der Methode `EcoShell::clearAttrIndexMap()` wird ein fehlender Rücksprungbefehl (`return`) hinzugefügt.
- Die Methode `scpAttrBoolean::setValueByString()` enthält Code zur Konvertierung der übergebenen Zeichenkette in Kleinbuchstaben. Der MS-Compiler kennt weder `std::tolower` (Linux/GCC) noch `std::towlower` (Borland). Der Code wird mit `#ifdef WIN32` um `_strlwr` anstelle von `std::transform` ergänzt.

Einer Analyse während der PG zufolge ist außerdem die Aufspaltung der kombinierten Zuweisung
`(*itor)=cell=0;`
nach
`(*itor)=cell; cell=0;`
in `scpCellSpace::removeAll()` erforderlich, andernfalls kommt es zu einem Laufzeitfehler. Allerdings scheint `(*itor)=cell;` sogar komplett überflüssig.

Damit die Bibliotheken ihre Funktionalität nach außen zur Verfügung stellen können, wird den DLL-Projekten jeweils eine DEF-Datei hinzugefügt, in der alle Exporte aufgelistet sind. Die während der PG erstellte `model.def` wird um einige in der Zwischenzeit hinzu gekommene Methoden erweitert.

### 3.2.2 JNI-Wrapper und Parser mit MS Visual C++ 6.0

Obwohl für die ersten Tests nicht erforderlich, wird bei dieser Gelegenheit schonmal der JNI-Wrapper, wie bereits in der PG-Dokumentation [Kie04] beschrieben, mit den aktuellen Quellen neu erstellt. Dabei tauchen sowohl bekannte als auch neue Probleme auf.

Als erstes werden mit dem Werkzeug "Parser Generator 2" [2] über die Funktion 'Lib-Builder' einmalig die Bibliotheken für Yacc und Lex erstellt. Damit kann der Parser Generator aus den Quelldateien `flex_in.lex` und `yacc_in.y` die Zieldateien `lex.yy.cpp` und `lex.yy.h` bzw. `y.tab.cpp` und `y.tab.h` erzeugen. Die beiden CPP-Dateien werden dem JNI-Wrapper-Projekt hinzugefügt, ebenso wie die erforderlichen Include-Pfade und Bibliotheken-Verknüpfungen. Der Compiler verlangt bereits bekannte Änderungen an `yylex.h` und `yypars.h`; ein Fehler (`struct symtab*` statt `symTabEntry*`) in der `flex_in.lex` wird behoben. Der Linker verlangt, die Bibliothek `libc` auszuschließen, sowie ein zusätzliches `extern "C"` um `yyin` und `yyreset` in der `yacc_in.y`.

Zum erfolgreichen Kompilieren und Binden sind jetzt noch einige Patches erforderlich, die leider direkt im erzeugten C-Code, nicht im Parser-Code, vorgenommen werden müssen:

- In `lex.yy.cpp` und `y.tab.cpp` muss jeweils zwei mal `using namespace std;` entfernt werden.
- In `lex.yy.cpp` muss `extern "C" {...}` um diverse Variablen von `/* yytext */` bis vor `int yyerror(char *msg)`.
- In `y.tab.cpp` muss `extern "C" {...}` um diverse Variablen von `/* (state) stack */` bis vor `node* constNode(void)`.
- In `lex.yy.cpp` und `y.tab.cpp` muss jeweils YYCONST entfernt werden vor: `yybackup, yymatch, yynontermgoto, yyreduction, yystate, yystateaction, yystategoto, yytokenaction, yytransition`.

### 3.2.3 Andere Werkzeuge

Wo gerade von Entwicklungsumgebungen die Rede ist, sei schonmal vorgezogen die Erkenntnis, dass jene unter Linux durchgehend an der Komplexität des Quelltextes gescheitert sind: Keiner der Umgebungen KDevelop, Anjuta und Eclipse gelingt das Binden der Bibliotheken, Eclipse (mit C++-Plugin) kommt über die Kompilierung gar nicht hinaus. Ein weiteres Werkzeug namens IBM Visual Age verweigert aufgrund von Linux-Unkenntnis des Benutzers die Installation.
Ein reiner Debugger soll einen Debug-Prozess starten (oder sich mit einem laufenden verbinden) und die Ausführung im Einzelschrittmodus anhand des Quelltextes ermöglichen – während kdbg zumindest ersteres schafft, lässt sich xxgdb selbst gar nicht kompilieren.

[2] `http://www.bumblebeesoftware.com/`

Zwar funktioniert die Erstellung der Bibliotheken per `Makefile` nach wie vor, sodass Ergebnisse unter Linux überprüft werden können, doch der weitere Testprozess findet aufgrund der wesentlich komfortableren Entwicklungsumgebung fast ausschließlich unter MS Windows statt.

Wegen der oben genannten Probleme mit der Parser-Komponente wird ein Ersatz für den "Parser Generator 2" gesucht; ein vielversprechender Kandidat scheint das Gespann `bison.exe/flex.exe`[3], angeblich eine direkte Open-Source-Portierung der Unix-Äquivalente. Leider gelingt die Übersetzung der erzeugten C-Quelltexte nicht, es werden nicht vorhandene Dateien verlangt.

## 3.3 Validierungstest: EcoLife

Das EcoScape-Tutorial "EcoLife" [Spe02] macht keine Aussage über geeignete Startwerte der Attribute – damit bei der Simulation etwas 'passiert', wird nun die Initialisierung der 'lebendig'-Attribute mit Zufallszahlen durch `(bool)(rand()%2)` gewählt.
Bei der Implementierung des Modells gemäß der Anleitung stößt man auf etliche ziemlich eindeutige Tippfehler in den Quelltextfragmenten, die eine Entstehung 'auf die letzte Minute' vermuten lassen.
Eine konsolenbasierte Visualisierung (Abb. 8) vermittelt schließlich den Eindruck, dass alles so funktioniert, wie es sollte: Zwar würde eine systematische Prüfung jedes Einzelwertes zuviel Zeit kosten, aber die stichprobenartige Kontrolle bestimmter Zustandsübergänge weniger Zellen entdeckt keine offensichtlichen Fehler.

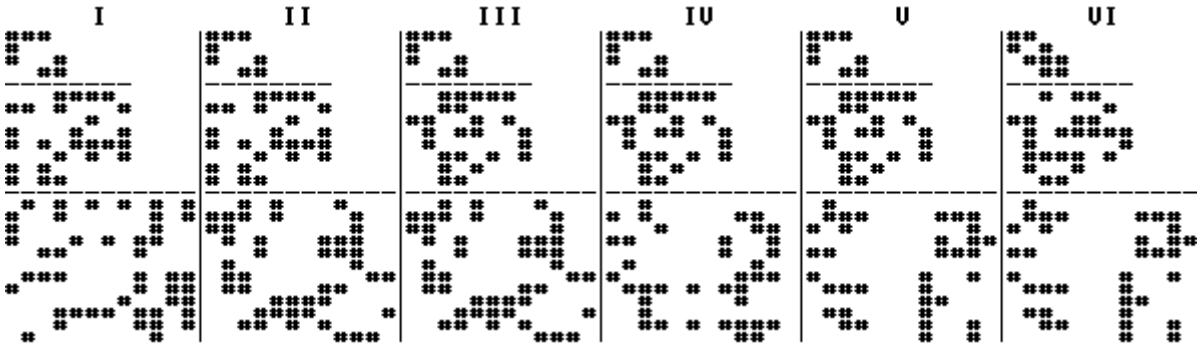


Abbildung 8: EcoLife – Bildschirmfotos einiger Simulationsschritte mit Zufallsinitialisierung – sechs Schritte von links nach rechts

Zudem genießt dieser Test schon im Vorfeld großes Vertrauen, da er parallel zum zu testenden Framework entwickelt worden ist – Vertrauen insofern, dass er wahrscheinlich keinen Fehler zeigt, *nicht* dass er die Abwesenheit von Fehlern zeigt! Der Leser sei an die Unstimmigkeit der Hierarchie-Konzepte erinnert: EcoLife steht im Widerspruch zur EcoScape-Diplomarbeit, aber auch zu den anderen Varianten. Experimente mit Hinzufügen oder Weglassen einzelner upper- oder lowerMappers bewirken auf den ersten

[3] http://www.monmouth.com/ wstreett/lex-yacc/lex-yacc.html

Blick lediglich eine leichte Veränderung der ‘Aussterbewahrscheinlichkeit’ der einzelnen Ebenen.

## 3.4 Fehlertest: Springbrunnen

Ursprünglich zum Testen von Kiek ist das Modell “Springbrunnen” entstanden. [Mue04] In Kiek implementiert, zeigt es deutlich die Fehlfunktion in der Ebenenkommunikation, nämlich zu viele Nullwerte nach wenigen Simulationsschritten.
Das Modell besteht aus drei Ebenen, nach unten hin nimmt die Zellengröße ab und die Zellenanzahl zu. Das virtuelle Wasser (in Form von Integer-Werten) wird in einem Kreislauf vertikal durch das System transportiert; auf dem Weg nach oben werden Summen gebildet.

Der Springbrunnen wird nun auf Basis des EcoLife-Tutorials, ebenfalls mit einer primitiven Visualisierung, implementiert. Die Benennung der Ebenen und Attribute hätte der Autor allerdings besser wählen können, denn im Original kommt man mit den Begriffen ‘upper’ und ‘lower’ leicht durcheinander. Ein Refaktorisierung macht die Sache wesentlich übersichtlicher.
Ergebnis der Simulation: Zwar nur bei gleicher Taktung aller Ebenen, aber das Prinzip funktioniert! Streng genommen ist dies ein Fehlschlag, denn dieser Fehlertest zeigt keinen Fehler auf.

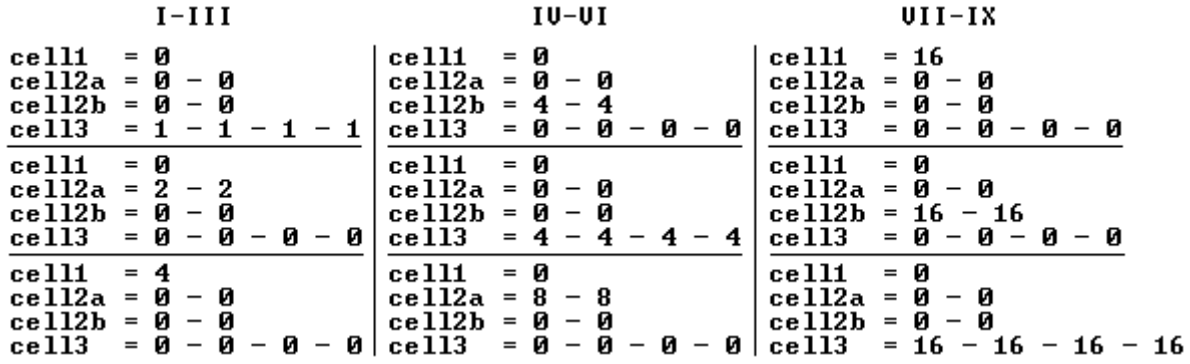


Abbildung 9: Springbrunnen – Bildschirmfotos einiger Simulationsschritte

## 3.5 Zwischenergebnis

Nach intensivem E-Mail-Kontakt mit dem Autor der EcoScape-Diplomarbeit kann jetzt endlich mit hoher Wahrscheinlichkeit bestätigt werden, dass das letztgenannte Hierarchiekonzept (Abb. 6) korrekt ist bzw. zumindest der Realisierung in EcoScape entspricht, und dass somit die Diplomarbeit an dieser Stelle fehlerhaft ist, zumindest die dortigen Abbildungen 7.12 bzw. 7.19 sind eindeutig falsch.

Damit ist auch klar, warum Kiek nicht funktionieren kann: Die Projektgruppe ist tatsächlich von falschen Voraussetzungen ausgegangen. Das Springbrunnenmodell lässt sich über den Assistenten nicht vollständig eingeben: Zwar kann man die nicht angebotenen Attribute manuell eintippen, doch die notwendige upperState-Transition ist gesperrt und muss nachträglich eingegeben werden. Bei der Simulation ist keine Änderung erkennbar, und der Versuch einer Visualisierung wird mit Ausnahmebehandlung quittiert.

## 3.6 Klassisches Debugging: Kiek-Abstürze

Unter MS Windows scheitert Kiek an technischen Problemen: Die Debug-Versionen der DLLs lassen die Java Virtual Machine bereits bei der Eingabe des Modelltitels abstürzen bzw. beim Laden eines gespeicherten Modells, die Release-Versionen schaffen es bis kurz vor die Simulation.
Die zeilenweise Ausführung in der Entwicklungsumgebung stoppt im ersten Fall bei einer einfachen Zuweisung in `scpModelElement::setName()`, und zwar ohne offensichtliche Ursache; diese Stelle wird häufig genutzt und war bislang nicht negativ aufgefallen. Der zweite Fall ist wesentlich unangenehmer – zugleich einfach (die lokale Ursache ist klar) und kompliziert (der Ursprung ist unerklärlich) – und hat eine Menge Zeit verschlungen. Im Quelltext der EcoScape-Simulator-Komponente ist an Kommentaren, auskommentierten Zeilen und infantilen Debug-Ausgaben erkennbar, dass mindestens einer meiner Vorgänger dort schon vergeblich gearbeitet hat. Das Problem ist: Im Simulator-Konstruktor werden zwei Zeitpunkte (`scpTimePoint`) erzeugt und initialisiert. Beim Verlassen des Konstruktors wird der TimePoint-Destruktor aktiviert und will per `delete`-Operation Speicher freigeben. Doch das interne Attribut `scpAttrFloat` ist ungültig und das Programm stürzt ab. Außer einem Entfernen der Speicherfreigabe (und damit dem Schaffen eines Speicherlecks ) wird hier zunächst keine Lösung gefunden.
Abstürze gibt es damit nun keine mehr, doch die Visualisierung gelingt auch hier nicht.

## 3.7 Neue Strategie: JNI-Wrapper ohne JNI

Die unterste Schicht der Bottom-up-Vorgehensweise kann durch die Tests mit EcoLife und Springbrunnen als validiert gelten. Es ist Zeit für einen Schritt nach oben im Architekturdiagramm (Abb. 7), die Einbeziehung der Komponente “JNI-Wrapper”.

Ein paar Worte zur Funktionsweise: Das *Java Native Interface* [Sun1] ist eine Programmier-Schnittstelle zur Verbindung von Java mit anderen, ‘nativen’ Anwendungen und Bibliotheken, diese können in beliebigen Programmiersprachen implementiert worden sein. Die Kommunikation zwischen den Objekten auf beiden Seiten ist ‘direkter’ als bei Konkurrenzlösungen wie SOAP oder CORBA, während RMI sogar nur zwischen Java-Teilsystemen funktioniert [Hae00]. Damit von Java auf native Methoden zugegriffen werden kann, müssen diese in spezielle Bibliotheken verpackt werden, dabei benötigt jede Methode ihre eigene, besondere Signatur.
Während der Entwicklung von Kiek sollte der Quelltext von EcoScape so wenig wie

möglich geändert werden, aber es gab auch andere Gründe dafür, eine Zwischenschicht zur Abwicklung sämlicher Kommunikation zwischen den Java- und C++-Komponenten einzuführen [Kie04]. So besteht der Kiek-JNI-Wrapper aus 52 JNI-Funktionen, die sich einerseits um die Datenhaltung auf C++-Seite kümmern (dazu enthält ein globaler `vector` Referenzen auf sämtliche verarbeiteten Modell-Objekte Modelle, Ebenen, Zustände, Attribute und Zellen) und andererseits nebst etwas Verwaltung (z.B. Konvertierung von UTF8- und Unicode-Zeichenketten) die Anfragen von Kiek an EcoScape weitergeben.

Um die Grenze zu Java noch nicht zu überschreiten, sollen die folgenden Tests zwar die JNI-Wrapper-Methoden benutzen, so wie es Kiek sonst eigentlich tun würde, aber die echte JNI-Funktionalität ist nicht erforderlich.

Als eine Möglichkeit fällt bei der Lektüre der JNI-Dokumentation ein Tutorial [4] auf, das den direkten Aufruf ("invocation") der Java Virtual Machine erklärt – das wäre wirklich eine schöne Lösung, denn dann könnte der Quelltext des JNI-Wrappers unverändert bleiben. Aber, kurz gesagt, es klappt nicht: Unter MS Windows erwartet die Methode `JNI_CreateJavaVM` entgegen der Dokumentation als zweiten Parameter einen `void**`, bei Zwangskonvertierung lautet der Rückgabewert `-1`. Grund dafür könnte allerdings auch sein, dass `JNI_GetDefaultJavaVMInitArgs()` keinen `classpath` setzt. Unter Linux gelingt schon die Kompilierung nicht aufgrund fehlender Definition benötigter Datentypen.

Mangels Alternativen wird nun doch der JNI-Wrapper geändert, oder zumindest eine Kopie: Da die Java-VM nicht zur Verfügung steht, müssen alle damit in Zusammenhang stehenden Elemente aus dem JNI-Wrapper herausoperiert werden. Es handelt sich dabei hauptsächlich im die Signaturen, die im Original jeweils zwei zusätzliche Parameter (für die JNI-Umgebung und das aufrufende Objekt) enthalten, und die Zeichenkettenkonvertierung. Die meisten primitiven Datentypen können erhalten bleiben, da es sich lediglich um Neudefinitionen handelt (Bsp. `typedef unsigned char jboolean;`), umständlicher ist der Umgang mit Feldern.

## 3.8 Back-to-back-Test: JNI-Springbrunnen

Auf Basis des kastrierten JNI-Wrappers wird wiederum das Springbrunnenmodell implementiert. Dabei fällt auf, dass die Reihenfolge der Modellierungsschritte verändert werden muss: Der JNI-Wrapper erlaubt das Initialisieren von Attributen erst nach dem Erstellen des zugehörigen Zellenraumes. Um die Vergleichbarkeit zu gewährleisten, werden die gleichen Modifikationen parallel am Original-Springbrunnen vorgenommen. Eingeschobene *Regressionstests* zeigen keine Änderung der Ergebnisse.
Neben der Attributwertzuweisung muss auch die Definition der Zustandsübergangsfunktionen nach hinten verlagert werden: zuvor wollen Nachbarschaften, Zeitpunkte sowie Zustandsübergänge ("Mappers") im Rohzustand erzeugt werden. Falsche Reihenfolgen machen sich sehr deutlich an Zugriffsverletzungen zur Laufzeit bemerkbar.

[4] `http://java.sun.com/docs/books/tutorial/native1.1/invoking/invo.html`

Die komfortable Erzeugung der Zellenräume mit `scpLatticeCellSpace` ist im JNI-Wrapper nicht vorgesehen, stattdessen muss jede Zelle umständlich als Feld von Koordinaten definiert werden.
Gänzlich anders als zuvor werden die Zustandsübergangsfunktionen definiert, die Definition zur Kompilierzeit steht nicht zur Verfügung. Aus diesem Grund ist von der Projektgruppe eine einfache Skriptsprache mit Java-ähnlicher Syntax entwickelt worden, mit deren Hilfe sich zur Laufzeit relativ komplexe Zustandsübergänge kreieren lassen.

```
//layer1->setCellState( cellState1 );
Java_kiekcore_jniEcoScapeImpl_KiekModelLayer_jniSetCellState( layer1, cellState1 );
...
//cell = new scpModelCell();
//ring = new scpGeoLineRing();
//ring->insertPointAtEnd( new scpGeoPoint(  100,  100 ) );
//ring->insertPointAtEnd( new scpGeoPoint(  600,  100 ) );
//ring->insertPointAtEnd( new scpGeoPoint(  600, 1100 ) );
//ring->insertPointAtEnd( new scpGeoPoint(  100, 1100 ) );
//cell->setGeometry( new scpGeoPolygon( ring ) );
//cellSpace->addCell( cell );
double pts02[] = { 100,  100,
                   600,  100,
                   600, 1100,
                   100, 1100 };
res = Java_kiekcore_jniEcoScapeImpl_KiekScenarioLayer_jniAddModelCell( layer2, pts02, 8 );
assert( res == 32 );
...
//layer1->setLowerStateMapper( new SprbrnLowerMapper1() );  // lower1 = sum(cell2a)
prg = Java_kiekcore_jniEcoScapeImpl_KiekScenarioLayer_jniSetLowerStateMapper( layer1,
  "LSA(lower1) = 0; \
   FOREACH EMBEDDED { \
      LSA(lower1) = LSA(lower1) + CSA(cell2a); \
   }", 0 );
assert( prg );
assert( prg[0] == 0x0000001a );   // 1a PUSHCONSTI
assert( prg[1] == 0x00000000 );   //    <const>
assert( prg[2] == 0x00004022 );   // 22 POPI      40 lower  00 index    lower1
assert( prg[3] == 0x00100131 );   // 31 BEGINLOOP 01 ctr    10 cell     FOREACH {
assert( prg[4] == 0x00004012 );   // 12 PUSHI     40 lower  00 index    lower1
assert( prg[5] == 0x00031012 );   // 12 PUSHI     10 cell   03 index    cell2a
assert( prg[6] == 0x00000091 );   // 91 ADD                             +
assert( prg[7] == 0x00004022 );   // 22 POPI      40 lower  00 index    lower1
assert( prg[8] == 0xfffc0132 );   // 32 ENDLOOP   01 ctr    fffc adr    }
assert( prg[9] == 0x000000f0 );   // f0 RETURN
delete[] prg;
test = Java_kiekcore_jniEcoScapeImpl_KiekModelState_jniGetParserErrorString();
assert( strlen(test) == 0 );
```

Abbildung 10: JNI-Springbrunnen – Quelltextfragmente – als Kommentar darüber jeweils das Original

Wo immer möglich, wird der Debug-Befehl `assert()` eingesetzt, um eine Zusicherung bestimmter Eigenschaften des Modellierungsvorganges zu erreichen. Darunter fällt die Prüfung, ob Elemente nach ihrer Verarbeitung noch den korrekten Titel tragen und ob sie mit einem vorgesehenen Index gespeichert worden sind. Die Korrektheit des aus den Skripten erzeugten Bytecodes wird anhand der EcoShell-Dokumentation [Sti04] überprüft.
Dabei ist aufgefallen, dass `jniGetIntAttribute()` im Fehlerfall 0 zurückgibt – die aufrufende Methode kann die Fehlermeldung nicht vom Wert unterscheiden – aber eine Lösung kann momentan nicht angeboten werden.

## 3.9 Fehler in Parser und EcoShell

Die Erleichterung ist groß, als tatsächlich endlich der erste Fehler im System sichtbar wird: Nicht nur im Kiek-Model-Wizard, sondern auch im Parser ist ein `FOREACH EMBEDDED` analog zum `FOREACH NEIGHBOR` nicht (vollständig) enthalten! Zwar ist das Token deklariert, aber die Bytecode-Erzeugung (der Block `case EMBEDDED` in `assm()`) fehlt komplett und muss jetzt (dem Betreuer Ingo Stierand wird diese Ehre zuteil) nachgerüstet werden.
Aufgefallen ist der Fehler nicht etwa durch eine konkrete Fehlermeldung, sondern durch den eben erwähnten Zusicherungsvorgang – dem generierten Bytecode mangelte es an einer Schleife.
Bei einer spontanen *Code-Review* der gesamten Parser-Quellen fiel noch ein potenzieller Fehler in der `flex_in.lex` ins Auge: die Länge des Schlüsselwortes `EMBEDDED` wurde im Gegensatz zu `NEIGHBOR` mit 9 statt 8 Zeichen deklariert.
Des Weiteren wurden in diesem Zusammenhang wenig später in `EcoShell::getNumCells()` ein Fehler (`STATE_CELL` war `STATE_LOWER`) und eine 'Unstimmigkeit' (`ATTR_STATEOUT` war `ATTR_STATENEIGHB`, aber dies wurde an anderer Stelle beachtet) bereinigt.

## 3.10 Fehler im Simulator

Nach erfolgreichem Abschluss der Modellierung des JNI-Springbrunnens soll dieser in der Fortsetzung des *Back-to-back-Fehlertests* Unterschiede zum Standalone-Springbrunnen zeigen. Dies tut er schneller als erwartet, indem er den Simulator abstürzen lässt. Bei den Vorbereitungen zur Berechnung der Zustandsübergänge, genauer gesagt bei der Sammlung der Quellzustände, zeigt der Neighborhood-Iterator in `scpSimulator::calculateCell()` ein völlig absurdes Verhalten: Zunächst gibt er die Anzahl der Nachbarn jeweils korrekt mit 1, 2 bzw. 4 an, dann jedoch liefert er unabhängig von der tatsächlichen Anzahl erst genau ein gültiges, dann genau ein ungültiges Zellenobjekt! Es darf vorweggenommen werden, dass die Untersuchung dieses Problems relativ viel Zeit in Anspruch genommen hat und dennoch keine Lösung vorweisen kann...

Unter MS Windows bekommen offenbar (zumindest im Debug-Modus) ungültige Zeiger den Wert `0xcdcdcdcd`. Auf diese Weise kann der fehlerhafte Iterator erkannt und die Bearbeitung abgebrochen werden, sodass der JNI-Springbrunnen jetzt tatsächlich die gleichen Ergebnisse liefert wie sein `Orakel`, der Original-Springbrunnen.

Interessant ist: Obwohl Nachbarschaften im Original-Springbrunnen keine Rolle spielen, wird der Neighborhood-Iterator dort benutzt und funktioniert auch korrekt.

Die Herangehensweise ist nun einerseits geprägt durch unsystematische Inspektionen von irgendwie mit dem Iterator in Zusammenhang stehenden Quelltextfragmenten, andererseits durch die Untersuchung der defekten Objekte im Speicher zur Laufzeit mit Hilfe der Entwicklungsumgebung und Vergleich mit ihren fehlerfreien Verwandten. Leider findet keine der Varianten auch nur den geringsten Anhaltspunkt. Vielleicht sollte eine Fehlfunktion im Basiscode der verwendeten STL-Bibliothek in Betracht gezogen werden –

immerhin scheint der Neighborhood-Iterator der einzige in Verbindung mit einer `map` zu sein, andere Iteratoren arbeiten stattdessen mit einer `list`.

Als positiver Nebeneffekt kann jedoch eine Lösung für das Problem im `scpTimePoint`-Destruktor angeboten werden. Zunächst kam auch hier ein Workaround ähnlich wie beim Neighborhood-Iterator infrage, indem das Objekt vor Speicherfreigabe auf Gültigkeit getestet wird. Konkret für das `scpAttrFloat`-Attribut des `scpTimePoint` existiert sogar bereits eine interne `check()`-Methode für diesen Zweck, die sich in diesem Fall über ein fehlendes `scpAttrScheme` beschwert.
Im Zuge einer *Review* wurde allerdings entdeckt, dass in `scpTimeLine::addTimePoint()` mit einer gänzlich anderen Methode das gleiche Ziel erreicht werden soll wie im `scpSimulator`-Konstruktor, nämlich die Erzeugung von Zeitpunkten, und zwar durch die Ableitung vom `scpAttrScheme` eines bestehenden Modells. Dieses Modell ist im `scpSimulator`-Konstruktor jedoch nicht verfügbar, Lösung wäre ein überladener Konstruktor, wie er bereits (fälschlich) im EcoLife-Tutorial verwendet wird. Mit dieser neuen Konstruktion und angepasstem JNI-Wrapper tritt dieses Problem nicht mehr auf.

## 3.11 Fehlertest: Game Of Life

Auf Wunsch der Betreuer wurde mit der Entwicklung eines um Nachbarschaften erweiterten Springbrunnenmodells begonnen... doch nach kurzer Zeit abgebrochen. Entweder es würde ähnlich unübersichtlich wie EcoLife oder die beiden Teilautomaten 'Hierarchie' und 'Nachbarschaft' müssten völlig unabhängig voneinander (durch disjunkte Attribute) arbeiten. Beide Fälle bieten für die aktuelle Validierungsphase keine Vorteile. Daher wird die Implementierung eines bewährten Zellularen Automaten, nämlich "Game Of Life"[5], bevorzugt und parallel sowohl mit als auch ohne Nutzung der JNI-Funktionen erstellt.
Das Standalone-GameOfLife funktioniert tadellos – Objekte wie der berühmte Gleiter verhalten sich erwartungsgemäß –, während das JNI-GameOfLife wie erwartet die bekannten Probleme mit dem Neighborhood-Iterator hat, d.h. die Simulation funktioniert zwar, doch gibt es keine Änderungen über die Zeit.

Bei der Implementierung wird allerdings ein weiteres Missverständnis offenbart: Die bitweisen logischen Operatoren in `EcoShell` funktionieren nicht so, wie es die Projektgruppe bei der Entwicklung des Kiek-Model-Wizard angenommen hatte! Eigentlich sollten Konstruktionen wie diese möglich sein:

```
IF( (NSA(counter) < 2) OR (NSA(counter) > 3) ){
    CSA(alive) = FALSE;
}
```

Doch die Dokumentation der vorliegenden EcoShell-Version sagt sinngemäß, dass bei Anwendung des bitweisen `OR` die beiden obersten Elemente vom Stack entfernt und das

[5] `http://de.wikipedia.org/wiki/Game_of_life`

Ergebnis dort abgelegt wird, während aber bei der Berechnung der logischen Operatoren (größer, kleiner etc.) beide Elemente auf dem Stack verbleiben und das Ergebnis in einem speziellen Test-Flag abgelegt wird, welches wiederum von der `IF`-Anweisung ausgewertet wird. Kiek bräuchte also im Prinzip eine Auswertung der letzten beiden `TEST`-Ereignisse, doch mit einem einzelnen Test-Flag ist dies nicht möglich.

Workaround 1 – Vermeidung der bitweisen Operatoren:

```
IF( NSA(counter) < 2 ){
    CSA(alive) = FALSE;
}
IF( NSA(counter) > 3 ){
    CSA(alive) = FALSE;
}
```

Workaround 2 – Verwendung temporärer Variablen:

```
VAR(var1) = NSA(counter) < 2;
VAR(var2) = NSA(counter) > 3;
IF( VAR(var1) OR VAR(var2) ){
    CSA(alive) = FALSE;
}
```

Als langfristige Lösung könnte man eine Änderung in EcoShell in Betracht ziehen.

## 3.12 Direkter Vergleich der Varianten

Um die Ursache für die Fehlfunktion des Neighborhood-Iterator einzugrenzen, wurden die wesentlichen Teile der Quelltexte von Original-Springbrunnen, JNI-Springbrunnen und JNI-Wrapper in einer tabellarischen Übersicht einander gegenübergestellt. Da der Original-Springbrunnen zuvor bereits an sein JNI-Äquivalent angeglichen worden war, fanden sich erwartungsgemäß nur geringe Unterschiede: die Konstruktion der Zustandsübergänge und die Zuweisung der Übergangsfunktionen.

Für das GameOfLife gilt das gleiche Prinzip, und da hier die Auswirkungen größer sind, wird die Angleichung mit der Ausrüstung des JNI-GameOfLife mit den Zustandsübergangsfunktionen des Original-GameOfLife fortgesetzt. Doch der Neighborhood-Iterator lässt sich davon nicht beeindrucken und produziert weiterhin defekte Zeiger. Leider enthält der tabellarische Vergleich nunmehr keine Abweichungen zwischen den Modellvarianten – damit beginnt die Suche nach der Ursache wieder von vorn.

## 3.13 Weiterführende Überlegungen

Experimente mit den Compiler-Optionen unter MS Windows ließen in dieser Angelegenheit bislang ebenfalls keine Hoffnung auf Besserung.

Unter Linux gibt es zwar keinen Fehler, aber auch keine Bewegung in der Visualisierung.

Vielleicht stimmt doch irgendetwas an der Reihenfolge der Aufrufe der JNI-Wrapper-Methoden nicht? Zur Klärung dieser Frage wurde zunächst eine Auflistung erstellt, welche JNI-Methoden in Kiek überhaupt referenziert werden – es sind alle 52. Um die Reihenfolge der Aufrufe im praktischen Einsatz zu finden, wurden jeweils zusätzliche Debug-Ausgaben in jede JNI-Wrapper Funktion integriert und ein Modell in Kiek erstellt und simuliert. Leider zeigen die ersten Analysen keine signifikanten Abweichungen.

## 3.14 Zusammenfassung

Wie befürchtet, ist der größte Teil der Quelltexte tatsächlich derart beschaffen, dass systematische Tests nur schwierig anzuwenden sind. Insbesondere die Objektorientierung mit ihrer Vererbung, Überladung und gegenseitigen Verschachtelung macht das Leben in diesem Fall nicht leichter. Der EcoScape-Autor hat mit seinen internen `check()`-Methoden bereits einen guten Weg eingeschlagen, doch hätte er dies ruhig konsequent weiterführen können.

Bei der rückblickenden Betrachtung der Vorgehensweisen sticht vor allem der Back-to-back-Test überraschend positiv hervor, auch wenn die planlosen, spontanen Inspektionen wahrscheinlich die meiste Zeit beansprucht haben. Im Allgemeinen sind die Annahmen in Kapitel 2.11 jedoch durchaus zutreffend: Die richtige Mischung macht's!

Am Ende fehlt leider wie üblich die Zeit, um die noch verbleibenden, zum Teil erst in der Endphase der Dokumentation aufgekommenen Ideen in die Praxis umzusetzen. Doch wenn lange Zeit nicht feststeht, ob gen Ende eines solch zeitlich beschränkten Projektes überhaupt *irgendein* Fortschritt stehen wird, dann sollte man wohl froh sein über jeden gefundenen Fehler, denn er bedeutet schließlich einen erfolgreichen Test.

# 4 Ergebnisse

Dieses Kapitel bietet einen Überblick über die Erkenntnisse, die während der Analyse der Fehlfunktionen des Gesamtsystems Kiek/EcoScape gewonnen wurden.

## 4.1 Fehler in der EcoScape-Diplomarbeit

Das Konzept der Hierarchie und der Kommunikation ist missverständlich ausgedrückt oder sogar falsch im Vergleich zum tatsächlich realisierten Framework.
Die Abbildungen 7.12 und 7.19 sind definitiv falsch.

## 4.2 Fehler im EcoScape-Tutorial

Hierbei handelt es sich um Kleinigkeiten. Die Zeilennummern entsprechen denjenigen in den Quelltextfragmenten im Tutorial.

- 029 `layer1->setOutState( neighbState );`
  sollte wohl sein:
  `layer1->setNeighborhoodState( neighbState );`
- 043 `layer1->setOutStateMapper( new EcoLifeOutState Mapper() );`
  sollte wohl sein:
  `layer1->setOutStateMapper( new EcoLifeOutMapper() );`
- 058 `setCellMapperMapper`
  sollte wohl sein:
  `setCellStateMapper`
- 090 `scpModelTimeLine* timeline;`
  sollte wohl sein:
  `scpTimeLine* timeline;`
- 103 `simulator->init(myModel)`
  gibt es nicht, sondern nur
  `simulator->init()`
  zusätzlich sollte vielleicht
  `simulator->setModel(myModel)`
  aufgerufen werden
- `EcoLifeOutMapper`: `out_state.m_alive = cell_state.m_alive;`
  sollte wohl sein:
  `out_state->m_alive = cell_state->m_alive;`

## 4.3 Fehler im EcoScape-Quellcode

Hier sind außer den echten Fehlern auch einige potenzielle Fehler und Merkwürdigkeiten aufgelistet.

- `scpAttribute::scpAttribute()` – seltsame Initialisierung
- `scpCellSpace::removeAll()` – fehlerhaft, scheint aber nicht benutzt zu werden – `(*itor)=cell;` auskommentiert
- `scpGeoCellSpace::cellAtPoint()` – Kommentar irreführend
- `scpGeoCellSpace::cellsAtPoint()` – unvollständig, scheint aber nicht benutzt zu werden
- `scpGeoCellSpace::cellsInRect()` – unvollständig, scheint aber nicht benutzt zu werden
- `scpLatticeCellSpace::cellsInRect()` – unvollständig, scheint aber nicht benutzt zu werden
- `scpModel::check()` – falsche Prüfung, ob upper/lower existieren, führt aber nur zu Warnungen
- `scpModel::check()` – fehlende Prüfung, ob überhaupt ein Layer existiert
- `scpModel::check()` – schlägt nicht Alarm, wenn unter Nutzung von JNI keine Attribute existieren, weil irgendwoher das zusätzliche Attribut `neighborhoodGrade` erstellt wird
- `scpModelCell::removeNeighborCell()` – unvollständig, scheint aber nicht benutzt zu werden
- `scpSimulator::scpSimulator()` – die ursprüngliche Methode scheint fehlerhafte TimePoints zu erzeugen – Auswirkung (Absturz im Destruktor) aber nur bei Nutzung des JNI-Wrappers
- `scpSimulator::calculateCell()` – Abstürze im Neighborhood-Iterator, wenn aus JNI-Wrapper aufgerufen
- `scpTimePoint:: scpTimePoint()` – Abstürze, wenn vom Simulator aus JNI-Wrapper aufgerufen

## 4.4 Fehler in EcoShell

`vMapper->numSourceStates(STATE_CELL)` in `EcoShell::getNumCells()` war zuvor `vMapper->numSourceStates(STATE_LOWER)`.

Die bitweisen logischen Operatoren (OR, AND etc.) funktionieren nicht so wie in Kiek erwartet.

## 4.5 Fehler im Parser

`symTabEntry * symlook(char* s, locEnum loc);` in `flex_in.lex` war zuvor `struct symtab* symlook(char* s, locEnum loc);`.

`"EMBEDDED { tokenpos+=8"` in `flex_in.lex` war zuvor `"EMBEDDED { tokenpos+=9"`.

`FOREACH EMBEDDED` in `yacc_in.y` war unvollständig und fehlerhaft implementiert.

## 4.6 Fehler im JNI-Wrapper

- `jniCalculateNext()` – gibt im Fehlerfall `FALSE=unfinished` zurück – potenzielle Endlosschleife
- `jniCheckMapperFct()` – gibt konstant 0 (=korrekt) zurück, `kiekParse()` ist (leider berechtigt) auskommentiert
- `jniGetIntAttribute()` – liefert 0 im Fehlerfall → unpraktisch!
- `jniInitForSimulation()` – `layer->lowerState()` – copy-paste-Fehler → wahrscheinlich harmlos
- `jniSetCellStateMapper()` – fehlende Initialisierung → wahrscheinlich harmlos
- `jniSetCellStateMapper()` – `upperAttrs = &modelAttrs[index-1]->upper;` war zuvor `upperAttrs = &modelAttrs[index-1]->cell;`
- `jniSetLowerStateMapper()` – Copy/Paste-Fehler: locUSA und locLSA verwechselt

## 4.7 Fehler in Kiek

Im Quelltext von Kiek wurden keine Fehler gesucht, aber die grafische Benutzungsoberfläche wurde sporadisch genutzt.

Die Eingabe von Modellen ist nur sehr umständlich, über Assistent nicht vollständig möglich, dort können Attribute versteckt bzw. der falschen Ebene zugeordnet sein. Auch fehlt ein Hinweis auf `FOREACH EMBEDDED`.

Bei der Implementierung des aktuellen Springbrunnenmodells in Kiek verlangte der Ebenenassistent die Eingabe von Nachbarschaftsattributen und der Modellassistent verbietet die Eingabe des lowerState für die oberste Ebene.

## 4.8 Fehler im Kiek-Tutorial

Das Game Of Life in Abschnitt 21.3.7 ist nicht ganz korrekt: beim `NeighborhoodState` fehlt die Initialisierung, beim `CellState` muss die erste Zahl 3 statt 2 sein.

## 4.9 Verbliebene Fehler

Der Neighborhood-Iterator in `scpSimulator::calculateCell()` ist irgendwie kaputt, darum funktionieren die Nachbarschaftsbeziehungen nicht korrekt.

Unter MS Windows mit Debug-Bibliotheken stürzt Kiek schon beim Eingeben des Modellnamens ab.

## 4.10 Fazit

Auch wenn es schön wäre, ein einfaches, aber vollständiges HAZA-Testmodell zu haben, mit dem sowohl Hierarchie als auch Nachbarschaft getestet werden können – im Laufe dieses Projektes gewann die Erkenntnis, dass zwei separate Modelle momentan mehr Sinn ergeben.

Es wurden einige kleine und wenige große Fehler aufgezeigt und größtenteils behoben. Die wirklich entscheidenden für den Fortschritt des Gesamtsystems sind:

- die unklare Dokumentation der Hierarchie,
- die Missverständnisse und Nachlässigkeit der Projektgruppe,
- das Fehlen von `FOREACH EMBEDDED` im Parser,
- die Verwechslung von States in EcoShell,
- die schlechte Kontruktion des Simulators,
- der defekte Nachbarschafts-Iterator.

Damit kann das Hauptziel eines jeden Verifikations/Validierungs-Prozesses, die signifikante Erhöhung der Qualität eines Systems, als erreicht betrachtet werden, obgleich es noch Einiges zu tun gibt.

Auch die Autorin hat etwas gelernt, insbesondere über verschleppte Projektplanung...

## 4.11 Ausblick

In der Idee der Annäherung des funktionierenden und des nicht funktionierenden Modells liegt noch einiges Potenzial.

Wenn die C++-Probleme gelöst sind, könnte ein Mini-Kiek erstellt werden, das den Modellierungsprozess auf einer höheren Ebene im Sinne der Bottom-up-Strategie ausführt: Ähnlich den JNI-Testmodellen (Springbrunnen und Game Of Life) würde das Modell direkt im Quelltext erstellt, nicht über eine grafische Benutzungsoberfläche. Dabei sollte auf eine vollständige Abdeckung aller JNI-Methoden geachtet werden.

# Stichwortverzeichnis

# Literatur

[Boe79] B. W. Boehm: Software engineering - R & D trends and defense needs; Research Directions in Software Technology; MIT Press, 1979.

[GUV03] Steffen Gemkow, Martin Uhlig, Karsten Violka: Testfieber – Unit-Tests als Sicherheitsnetz für Programmierer; c't, 2003(13):226-229, 2003.

[Hae00] Michael Härtfelder: Kaffee mit Vitamin C – JNI als ultimativer Zweikomponentenkleber; 2000; `http://www.haertfelder.com/jni.html`, URL zuletzt geprüft am 30.03.2005.

[Kie04] Projektgruppe "Kiek", Dokumentation; Universität Oldenburg, 2004.

[Kie04b] Projektgruppe "Kiek", Präsentation; Universität Oldenburg, 2004.

[MS01] John D. McGregor, David A. Sykes: A Practical Guide To Testing Object-Oriented Software; Addison-Wesley, 2001.

[Mue04] Daniel Müller: Evaluation und Erweiterung des Modellierungssystem Ecoscape/Kiek zur Landnutzungsmodellierung; Universität Oldenburg, 2004.

[Mye91] Glenford J. Myers: Methodisches Testen von Programmen; 4. Auflage, Oldenbourg, 1991.

[SK1] Virtuelles Software Engineering Kompetenzzentrum: White-Box Test; `http://www.software-kompetenz.de/?10745`, URL zuletzt geprüft am 30.03.2005.

[Som01] Ian Sommerville: Software Engineering; 6. Auflage, Addison-Wesley, 2001.

[Spe02] Bert Speckels: Entwurf und Realisierung eines Frameworks für Hierarchische Asymmetrische Zellulare Automaten mit Anbindung an räumliche Datenbanken; Diplomarbeit, Universität Oldenburg, 2002.

[Sti04] Ingo Stierand: EcoShell Handbuch; PG Kiek, Universität Oldenburg, 2004.

[Sun1] Sun Microsystems Inc: Java Native Interface; `http://java.sun.com/j2se/1.4.2/docs/guide/jni/`, URL zuletzt geprüft am 30.03.2005.

[SV01] Michael Sonnenschein and Ute Vogel: Asymmetric Cellular Automata for the Modelling of Ecological Systems; Sustainability in the Information Society - 15th International Symposium Informatics for Environmental Protection, pages 631-636, 2001.