“Tool zum Vergleich von stochastischen Mehrzieloptimierungsverfahren“

Janick Cardinale
Zuständiger Assistent: Stefan Bleuler
Professor: Eckart Zitzler

29.07.2005

Inhaltsverzeichnis

1 Einführung

1.1 Problemstellung

Häufig haben praktische Optimierungsprobleme mit mehreren Zielfunktionen einen sehr komplexen Suchraum, sodass eine analytische Methode um das Problem zu optimieren nicht in Frage kommt. Dabei muss man geeignete Kompromisse unter den Zielfunktionen treffen. Ein Ziel ist, die Distanz zu den optimalen Kompromisslösungsvektoren (sog. pareto-optimale Lösungen) zu minimieren. Ein Anderes ist, den mehrdimensionalen Lösungsraum durch die gefundenen Lösungen gut abzudecken. Hier bieten sich stochastische Optimierungsverfahren an, im Besonderen Evolutionäre Algorithmen (EA). Heute bestehen schon sehr viele solche Algorithmen und auch klare Vorgehensweisen, um einen solchen zu entwickeln, nur ist es schwierig, für ein praktisches Optimierungsverfahren die richtigen Adaptionen an einem EA vorzunehmen, sodass die Ziele möglichst gut erreicht werden. Dem Entwickler solcher Algorithmen stellt sich häufig die Frage welche Adaptionen welche Konsequenzen haben d.h. in welche Richtung man sich bewegen soll. Dafür sollen solche stochastischen Optimierungsverfahren verglichen werden. Dies bedarf einer geeigneten, rechnergestützten Plattform.

Es stellen sich 2 grundlegende Probleme:

1. Es ist schwierig, die zu lösenden Probleme so zu modellieren, dass gute Lösungen entstehen. Gleichzeitig soll der entstandene Algorithmus möglichst auch auf andere Probleme anwendbar sein.

2. Es fehlt eine geeignete und klar spezifizierte Umgebung um mit den vielen Aspekten von Evolutionären Algorithmen und deren Vergleichsmethoden die Übersicht zu behalten. Für den Vergleich solcher Algorithmen fehlen zudem geeignete und einfache Methoden, um standardisierte Vorgänge zu automatisieren.

1.2 Was schon existiert

Zur Lösung des ersten Problemes wurde eine Plattform- und von der Programmiersprache unabhängige Schnittstelle für Suchalgorithmen definiert: PISA[1]. PISA gliedert einen Suchalgorithmus klar in zwei Teile auf, dem Optimierungsproblem spezifischen Teil, dem Variator, und dem Algorithmischen Teil, dem Selektor. Durch eine genau spezifizierte Schnittstelle kommunizieren die beiden Teile und machen so einen vollständigen Algorithmus aus.

Der Umgang für einen Entwickler eines Evolutionären Algorithmus mit PISA ist jedoch sehr umständlich, zumal für geeignete Auswertungen sehr

viele Läufe nötig sind und für jeden Lauf die Algorithmen separat gestartet werden müssen. Aus diesem Grund wurde ein Programm entwickelt, welches den Entwickler dabei unterstützen soll, Zwischenresultate einsehen zu können und mehrere Läufe des EA auf einmal zu starten: der Monitor[2] [1]. Das Aufsetzten mehrerer Monitorläufe ist sehr umständlich, zumal die generierten Daten noch sinnvoll abgelegt werden sollen, um diese dann weiterverarbeiten bzw. verglichen zu können. Die Algorithmenteile (Variator und Selektor) wie auch das Monitor-Modul müssen immer noch "von Hand"gestartet, d.h. die PISA Schnittstelle[2] muss kopiert und die einzelnen Pfade der Module müssen dem Monitor als Argument mitgegeben werden. Zudem liefert er kein Konzept, wie und wo die generierten Daten abgelegt werden sollen. Auch möchte man die Algorithmen gegeneinander vergleichen. Zu diesem Zweck werden verschiedene statistische Tools als weitere Module hinzugezogen.

Nun möchte man als Benutzer einen Monitorlauf und das Verwenden der statistischen Tools automatisieren und deren generierten Daten sinnvoll ablegen, sodass man sie gut auswerten und zu jeder Zeit einsehen kann. Zum Beispiel soll es auch möglich sein, Daten auszutauschen, ohne sich über deren Struktur Sorgen zu machen. Es gilt also eine Framework bzw. ein Tool zu entwickeln, welches die verschiedenen Module verknüpft und einen automatisierten Vorgang mit sinnvoller Datenstruktur anbietet. Dabei sollen die in den bisherigen Modulen wichtigen Aspekte der Flexibilität, Modularität, Plattformunabhängigkeit und Erweiterbarkeit beibehalten werden. Neu soll dem Entwickler von Algorithmen auch eine geeignete Übersicht über seine generierten Resultate verschafft.

1.3 Designziele

Flexibilität Daten sollen immer noch von Hand eingesehen und von Hand manipuliert werden können[3]. Das Programm soll also gegenüber den vorgefundenen Daten flexibel sein.

Weiter sollen neue Bausteine bzw. Module, welche an einem genau spezifizierten Ort eingefügt werden, dynamisch eingelesen und mit dem Tool gesteuert werden können, z.B. ein neuer Variator oder ein neuer statistischer Test. Durch das dynamische Einlesen der Daten soll garantiert werden, dass andere Programme bzw. neue Module die Daten bearbeiten können, solange die spezifizierte Datengliederung bzw. Verzeichnishierarchie eingehalten wird. So soll es z.B. möglich sein, schon vorhandene Daten mit dem Tool verarbeiten zu können.

[1] wobei der Name nichts mit einem geschützten Codefragement zu tun hat

[2] Es handelt sich hier um 2x6 Dateien, welche die Kommunikation zwischen Variator-Monitor und Monitor-Selektor gewährleisten

[3] Die Daten sollen also in einer *human readable* Form gespeichert werden

Modularität An der Modularität der bis anhin verwendeten Module soll angeschlossen werden. Das Tool soll sich als weiteres Modul anschliessen, das Programm soll nicht eine eigenständige Applikation darstellen.

Das Tool soll sich auch die bis anhin modulare Struktur zu Nutze machen, d.h. einzelne Berechnungsschritte (z.B. das Auswerten einer Referenzfront, welche für einen statistischen Test verwendet wird) sollen neben einer automatisierten Form auch einzeln ausführbar sein. Ein weiterer wichtiger Aspekt im Bezug auf die Modularität ist die Parallelisierbarkeit. Das Tool soll gleichzeitig bedient werden können, während verschieden Algorithmen im Hintergrund ablaufen. Falls ein Modul beendet, sollen die Resultate dynamisch eingelesen und verarbeitet werden, während im Hintergrund noch andere Algorithmen am Arbeiten sind.

Erweiterbarkeit Das Tool selber soll eine geeignete Architektur aufweisen, sodass es sich schnell und verständlich erweitern lässt. Dabei soll nicht ein eigenes Konzept, sondern auf bekannten Konzepten wie dem MVC Prinzip (Model-View-Controller) zurückgegriffen werden.

Das Programm soll sich in einer spezifizierten Verzeichnishierarchie bewegen, d.h. solange die Hierarchie bestimmte Eigenschaften beibehält, sind die einzelnen Module beliebig veränderbar. Natürlich muss auch die Namensgebung der später definierten Verzeichnisstruktur beim Design neuer Module berücksichtigt werden.

Übersicht Eines der Ziele beinhaltet auch, dass eine insgesamt klare Übersicht über vorliegende Daten sowie über die Vorgehensweise der automatisierten Vorgänge vorliegt. Die einzelnen Module sollen übersichtlich gesteuert, aber auch als eine Kette von Verarbeitungsschritten verwendet werden können. Die generierten Daten sollen nachvollziehbar sein, d.h. aufgrund der verschiedenen Log-Dateien und kopierten Parameterdateien sollen alle Berechnungen, welche vom Tool aus ausgeführt werden, nachgeahmt und reproduziert werden können.

Grundsätzlich sind den bis anhin existierenden Modulen 2 Elemente hinzuzufügen. Erstens eine geeignete Datenstruktur. Sie muss klar definiert werden, sodass auch andere Module sich darin verständigen oder sie sogar ausbauen können. Im Wesentlichen bedarf es einer plattformunabhängigen Datenbank, die hier in Form einer Verzeichnisstruktur und Namensgebung die verschiedenen Beziehungen speichert.

Zweitens bedarf es eines plattformunabhängigen Tools, welches andere Module ansteuert, sich in der Datenstruktur auskennt und durch ein grafisches Benutzer Interface (GUI) dem Entwickler von Algorithmen einen guten Überblick liefert. Das Tool soll mit dem Grundsatzziel entwickelt werden,

Abläufe vollständig zu automatisieren, aber es soll auch ermöglichen, die Module einzeln anzusteuern.

Das Tool, nennen wir es ControlTool, automatisiert also alle standardisierten Vorgaenge zum Start und für den Vergleich von EAs. Dabei können bequem per GUI alle nötigen Variator-, Selektor-, Monitor-, Indikator- und Testparameter eingegeben werden. Diese Parameter werden an der richtigen Stelle in der Verzeichnishierarchie abgelegt und werden vom GUI zu einem späteren Zeitpunkt auch wieder visualisiert, sodass der Benutzer die Berechnungen nachvollziehen kann. Weiter liest das Tool bei jedem Neustart alle in der für das Tool wichtigen Daten in der Verzeichnishierarchie ein, damit das System flexibel bleibt.

2 Architektur

Zum Verständnis des globalen Programmaufbaus, der automatisierten Vorgaenge des Programms sowie der Verzeichnishierarchie werden die vom Tool angesteuerten Module knapp erläutert.

Ein Variator ist der problemspezifische Teil eines evolutionären Algorithmus. Er kreiert eine Startpopulation und verändert die einzelnen Datum bzw. deren Lösungsvektor z.B. durch ein Crossing-over. Zusammen mit dem Selektor, welcher für die Auswahl der Individuen für eine spätere Generation zuständig ist, entsteht ein Variator-Selektor-Paar(VSP) bzw. einen Algorithmus. Die Kommunikation findet über eine Datei statt, welche den aktuellen Zustand des Algorithmus enthält[1]. Sie benötigen als Input eine Datei mit Parametern, den Aufenthaltsort der PISA Schnittstelle und eine Zahl welche als Pollintervall (Intervall zwischen dem Abtasten der Zustandsdatei) verwendet wird. Die Ausgabe ist eine Datei mit den Lösungsvektoren des Variators.

Zwischen dem Variator und Selektor kann der Monitor eingeschleust werden. Dabei agiert der Monitor als "man in the middle"; er unterbricht die Kommunikation vom Variator und Selektor und gaukelt ihnen eine Kommunikation vor, sodass er beide Teile kontrollieren kann. Zum Verdeutlichen dient die Abbildung 1. Als Input verwendet er vom Variator und Selektor jeweils die Parameterdatei, den Aufenthaltsort der PISA Schnittstelle, eine Parameterdatei und auch das Pollintervall. Die Ausgabe des Monitors ist variabel, bzw. hängt von den Parametern in der Parameterdatei ab. Im Wesentlichen können die Individuen der einzelnen Generationen ausgegeben werden.

Bound[4] berechnet von einem Datenset die oberste und unterste Grenze

[4]Eine genaue Beschreibung ist im Sourcecode enthalten, http://tik.ee.ethz.ch/pisa/distribution.tar.gz

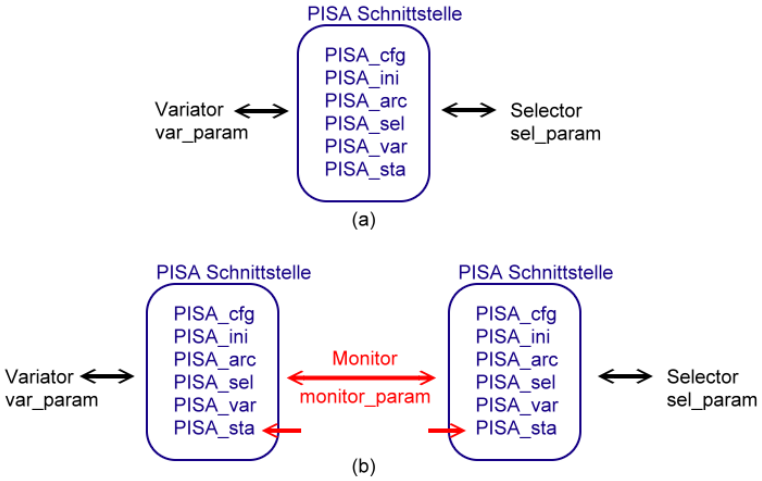

Abbildung 1: Der Monitor als "man int the middle"

für jede Zielfunktion.

Normalize[4] normalisiert für jede Dimension im Datenset die Daten so, dass sie zwischen 1 und 2 liegen. Das Tool benötigt die Grenzen eines Datenset als Input.

Filter[4] berechnet die intern im Datenset nicht dominierte Front für jede Dimension unter bestimmten Annahmen.

Indikatoren berechnen die Qualität von nicht dominierten Fronten im Vergleich zu einer Referenzfront aufgrund von verschiedenen Messungen[4]. Das Tool benötigt als Input eine Datei mit einer oder mehreren nicht dominierten Fronten und einer Referenzfrontdatei. Das Ausgabefile enthält pro Front einen Zahlenwert.

Statistische Tests ermitteln den p-Wert von zwei unabhängigen Stichproben. Falls das Inputfile mehrere Stichproben enthält, wird jede mit allen anderen Stichproben verglichen bzw. der P-Wert daraus wird ermittelt[4]. Die Tests benötigen eine Datei mit den Stichproben und eine Parameterdatei als Input. Ausgegeben werden die entsprechenden p-Werte. So können z.B. das VSP Knapsack-ibea und Knapsack-spea2 gegeneinander getestet werden. Der entsprechende p-Wert liefert Informationen darüber, ob die Null-Hypothese angenommen wird oder nicht, bzw. ob es zwischen den VSPs signifikante Unterschiede gibt.

Das folgende Beispiel für einen Standardablauf eines Tests soll Zusammenhänge verdeutlichen. Im Folgenden sind `var`,`sel` die Platzhalter für einen Variator bzw. für einen Selektor, `ind` steht für einen bestimmten Indikator. Die Zahl hinter dem Punkt bezeichnet eine bestimmte Generation. Mit `in` und `out` angegeben, welches Argument als Inputdatei bzw. als Out-

putdatei verwendet wird.

Nachdem der Algorithmus abgelaufen ist, werden alle Ausgabefiles des Monitor einer bestimmten Generation und einem bestimmten Variator aneinandergefügt (Die Algorithmen können nur bezüglich der Problembasierten Teile der Algorithmen(siehe PISA) verglichen werden, sonst würden Äpfel mit Birnen verglichen):

```
concat var_sel.x(in) var.x(out)
```

Als erstes berechnen wir die Referenzfront:

```
bound var.x(in) var_bound.x(out)
normalize var_bound.x(in) var.x(in) var_norm.x(out)
filter var_norm.x(in) var_ref.x(out)
```

Als zweites werden die Indikatorwerte ermittelt:

```
normalize var_bound.x(in) var_sel.x(in) var_sel_norm(out)
ind var_sel_norm.x(in) var_ref.x(in) var_sel_ind.x(out)
```

Und zum Schluss kann der Test ausgeführt werden. Dazu werden die Indikatorwerte wieder aneinandergefügt, sodass der Test jeden Wert mit jedem anderen vergleicht:

```
concat var_sel_ind.x(in) var_ind.x(out)
test var_ind.x(in) <parameter_file(in)> var_ind_test.x(out)
```

Alle diese Dateien gilt es nun, in einer Verzeichnisstruktur möglichst sinnvoll und ohne Verluste auf Information, d.h. wie sie in Beziehung stehen abzulegen.

2.1 Die Verzeichnis- und Dateienhierarchie

Grundsätzlich soll die im Folgenden präsentierte Verzeichnisstruktur nichts anderes als eine plattformunabhängige Datenbank darstellen. Die 2 Hauptziele einer solchen Struktur sind, erstens dem Benützer eine gute Übersicht zu verschaffen und zweitens möglichst viele Beziehungen zwischen den Daten zu speichern. Die beiden Ziele stehen manchmal in Konflikt.

Es gibt 3 Möglichkeiten, Referenzen zwischen Daten in einer Verzeichnishierarchie zu speichern. Die erste ist das Verschachteln von Verzeichnissen. Eine andere Möglichkeit ist, Namenskonventionen zu definieren und in den Namen der Verzeichnisse sowie der Dateien die nötigen Informationen hereinzuschreiben. Die letzte Möglichkeit ist, verschiedene Log-Dateien zu führen, welche die Referenzen zwischen den Daten beinhaltet.

Möglichst viele Inhalte über die Beziehungen der einzelnen Dateien nur mit Namenskonventionen zu speichern bedeutet auch, dass Datei- und Verzeichnisnamen schnell sehr kompliziert werden und so eine schlechte Übersicht resultiert. Ein weiteres Ziel betrifft die Logik gegenüber den vorhandenen Daten. Logik heisst, dass die Struktur des Verzeichnisbaumes der Natur der Daten entspricht und im Verzeichnisbaum genau dann eine Verzweigung vorliegt, wenn die darunter liegenden Daten immer anwachsend sind. Auch dieser Aspekt steht manchmal im Widerspruch dazu, dem Benützer eine gute Übersicht zu gewährleisten. Denn die Logik gegenüber den Daten ist nicht immer auch parallel zu der Intuition und Nachvollziehbarkeit des Benutzers. Ein weiteres Kriterium in unserer Designfrage stellt die Tiefe eines Verzeichnisbaumes dar. Das heisst es stellt sich in jeder Ebene des Baumes die Frage, ob sie überhaupt nötig ist oder ob eine grundsätzliche Entscheidung zur Auseinanderhaltung der Daten in den Dateinamen einbezogen wird.

Ein anderer Designentscheid ist, ob Informationen in Datei- und Verzeichnisnamen gespeichert werden sollen, oder ob für dies jeweils eine Log-Datei geschrieben werden soll. Die zweite Lösung bietet natürlich den Nachteil, dass beim manuellen Hinzufügen von Daten diese Log-Files aufbereitet werden müssen, falls man keinen Verlust der Beziehungsinformation haben möchte. Bei einem Entwurf einer solchen Verzeichnishierarchie muss man also über diese Prinzipien einen Entscheid treffen.

Zur Verdeutlichung ein kleines Beispiel: Wir möchten eine solche Datenbank für Pflanzen herstellen: Die in Figur 2 dargestellte Hierarchie stellt eine vollkommen logische Darstellung gegenüber den vorhandenen Daten dar. Die Pflanzen werden nach Ihren Familien sortiert. Eine Alternative mit einer fast absolut logischen Darstellung wäre es nun, die mittlere Ebene zu entfernen und so eine flachere Struktur herzustellen. Um nun keinen Informationsverlust zu haben, müssten wir die Blätternamen ändern. Zum Beispiel würde `Seerosengewächse_Weisse-Seerose.dat` eine Möglichkeit unter vielen Anderen darstellen. Natürlich würde dies für dieses Beispiel wenig Sinn ergeben. Eine allgemein intuitivere Art dies zu gestalten wäre es, die Pflanzen nach ihren Blütenfarben zu sortieren. Dies wäre aber ein brutaler Verstoss gegenüber dem natürlichen Aufbau der gegebenen Daten. Auch wären die Daten über die Familenart der Pflanzen nicht gespeichert. Zu diesem Zweck könnte man bei jeder Farbe für die Familie ein Unterverzeichnis einführen, was dann in gewisser Weise Datenredundanz zur Folge hätte. Als Alternative könnte man diese Ebene auch weglassen und wie oben vorgeschlagen den Familiennamen in den Dateinamen einbinden. So gibt es sehr viele Möglichkeiten, eine Verzeichnishierarchie zu definieren.

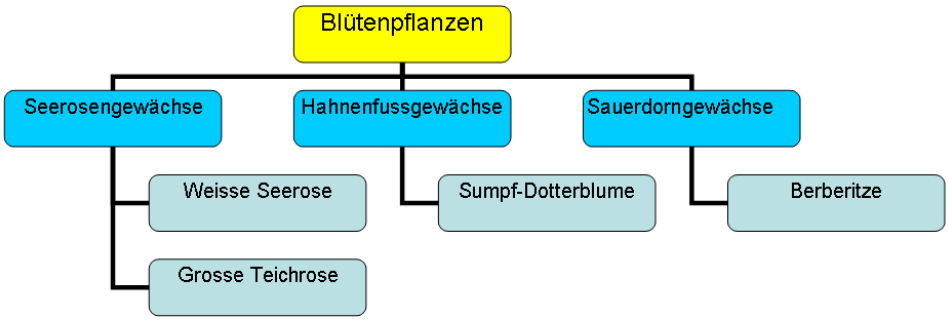

Abbildung 2: Beispiel einer Verzeichnisstruktur

2.1.1 Aufbau und Definition

Als Übersicht und Referenz dient die Abbildung 3, die zum Verständnis der nachfolgenden Gedankengänge eventuell unumgänglich ist.

Das Tool verfügt grundsätzlich über 2 Elemente, das funktionale Element, bzw. alle Module die angesteuert werden, und die Daten, welche generiert wurden oder werden. Zuerst beschreibe ich die Verzeichnishierarchie im Bezug auf die funktionalen Elemente. Der Aufenthaltsort dieser Elemente ist inhaltlich nicht von grosser Bedeutung, wichtig ist nur, dass die Voraussetzungen, die vom Tool gestellt werden, genau spezifiziert sind und eingehalten werden. Alle Modulgruppen bzw. funktionale Elemente des Pakets haben ihr eigenes Verzeichnis.

In den Verzeichnissen `variators/` und `selectors/` befinden sich alle Variatoren bzw. Selektoren in Form eines Unterverzeichnisses. Alle diese Unterverzeichnisse werden potentiell als Variatoren bzw. Selektoren angesehen. Sie enthalten mindestens 2 Dateien, eine ausführbare Datei und die Parameterdatei, welche den selben Namen wie die zugehörige ausführbare Variator- bzw. Selektordatei trägt, gefolgt von der Endung `_param.txt`. In Microsoft Windows trägt die Ausführbare Datei den Suffix `.exe`, welcher zu ignorieren ist. Zum Beispiel befindet sich in `variators/` die Ausführbare Datei `lotz` und ihr zugehörige Parameterdatei `lotz_param.txt`. Alle anderen Dateien in diesem Unterverzeichnis, z.B. eine Dokumentation, werden vom Tool ignoriert. Die zusätzliche, künstliche Ebene nach dem `variators/` bzw. `selectors/` Verzeichnis ist durch den Umstand, dass diese Module evt. selber aus weiteren Modulen bestehen und eine angemessene Dokumentation benötigen, zu begründen.

In ähnlicher Weise sind die Voraussetzungen an die Verzeichnisse `indicators/` , `statistics/` und `tools/`. Der Unterschied zu den Variatoren und Selektoren ist, dass sie flach in ihrem Verzeichnis liegen. Die Indikator-, und Testmodule benötigen alle eine Parameterdatei damit die Indikatoren bzw. statistischen Tests als solche anerkannt werden. Falls ein Modul keine

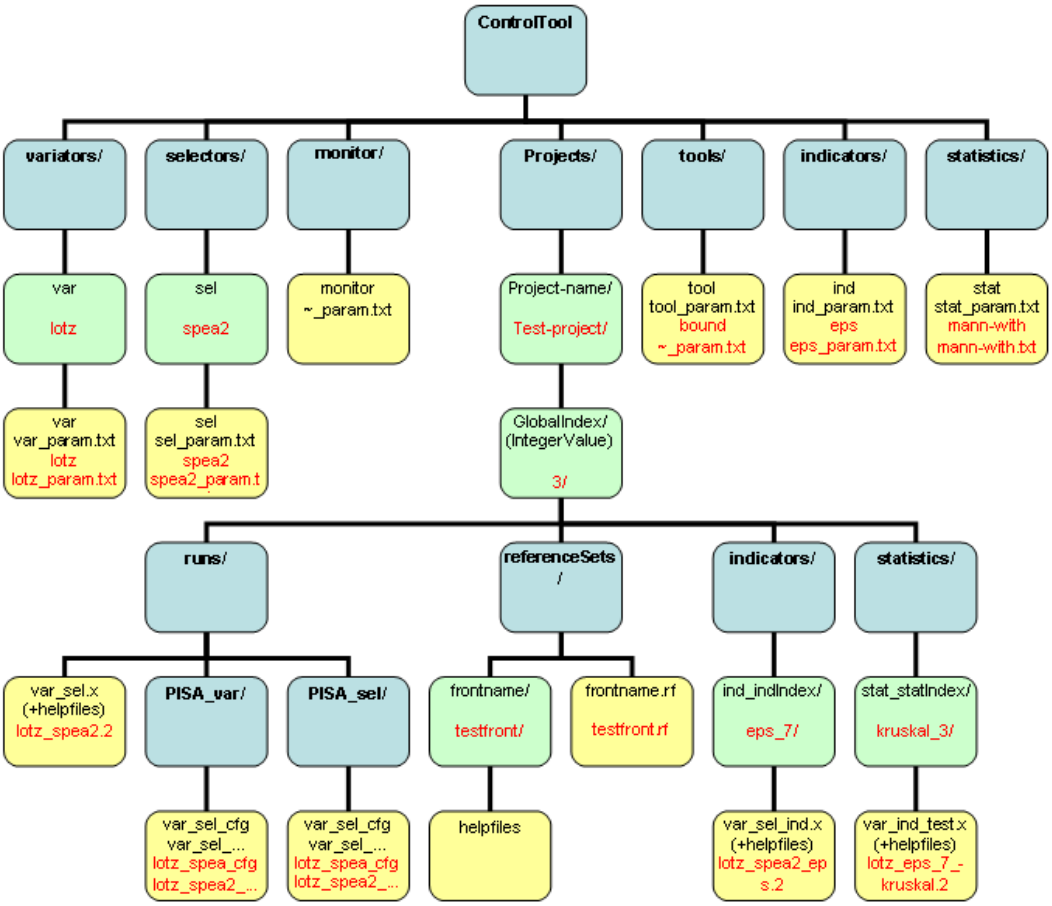

Abbildung 3: Die blau eingefärbten Verzeichnisse sind konstant. Grün eingefärbte werden reproduziert, d.h. für neue Daten wird so ein Verzeichnis neu angelegt (inklusive natürlich alle darunter liegenden Datenstrukturen). Gelb eingefärbt sind die Blätter des Verzeichnisbaumes. Darin befinden sich die eigentlichen Dateien. In den blau eingefärben Verzeichnissen befindet sich der Name. Für die grünen Knoten befindet sich in der ersten Zeile ein Beschrieb, bzw. die Platzhalter und auf der zweiten Zeile ein Beispiel(roter Text).

solche verwendet, kann eine leere Datei benützt werden[5].

Generierte Daten Der zweite Teil, die generierten Daten, werden alle in einem Verzeichnis `Projects/` gehalten. Dieses Unterverzeichnis ist für uns nun von grösserem Interesse.

Aus der Sicht des Bentzers ist wichtig, dass zusammengehörige Daten, d.h. Daten welche zusammen generiert wurden, nicht weit auseinander liegen. Desshalb wird zur Organisation der Daten als erstes eine künstliche Ebene `Projects/` eingeführt. Das Verzeichnis soll klar zusammengehörige Daten von anderen abtrennen. Sie soll so dem Benützer eine bessere Übersicht geben oder ihm dazu verhelfen, ältere Daten von neuen zu unterscheiden. Ein weiterer Grund für diese Ebene ist, dass mit ihr sehr einfach und klar Daten aus der Struktur kopiert oder gar gelöscht werden können. Natürlich ist umgekehrt auch sehr einfach möglich, Daten in die bestehende Struktur einzubinden. Dazu bedarf es nichts anderem, als den betreffenden Projektordner in das Verzeichnis `Projects/` zu kopieren.

Die nächst tiefere Ebene ist nun grundlegend sowohl für den Umgang mit dem Tool als auch für die Übersicht über die Daten. Der Name der Verzeichnisse besteht aus einer natürlichen Zahl. Er dient eher als Indexierung und wird im Folgenden der *globale Index* genannt. Die einem solchen Verzeichnis zugrunde liegenden Daten beschreiben einen Monitorlauf und alle die darauf aufbauenden Auswertungen(Referenzfronten, Indikatorwerte, statistische Tests). Sobald ein neuer Monitorlauf gestartet wird, wird auch ein neues Indexverzeichnis mit allen darunter liegenden Verzeichnissen generiert. Wie oben beschrieben ist dies ein gutes Beispiel dafür, dass die total logische Repräsentation von Daten gegenüber der intuitiven in Konflikt stehen kann. Diese Struktur ist eher intuitiv, denn der Benützer bezweckt mit jedem Monitorlauf etwas Bestimmtes, und nicht mit den einzelnen Algorithmen. Die logische Repräsentation dieser Daten wird weiter unten beschrieben(→ Alternativen). Zudem liegen so alle zusammengehörigen Daten beieinander und bieten einen guten Überblick über deren Zusammenhänge. Jedes dieser Verzeichnisse verfügt über 4 Unterverzeichnisse, diese Struktur ist natürlich beliebig erweiterbar.

Im `runs/` Unterverzeichnis werden alle vom Monitor generierten Daten abgelegt, d.h. zu jedem VSP die vom Benutzer gewünschten Generationen. Zudem läuft hier der Monitor Prozess ab. D.h. hier befindet sich zu jedem VSP die PISA Schnittstelle. Da der Monitor voraussetzt, dass die PISA Schnittstelle für den Variator und den Selektor nicht am selben Ort sein kann, werden zwei Unterverzeichnisse eingefügt: `PISA_var/` und `PISA_sel/`.

Das `referenceSets/` Unterverzeichnis enthält alle im aktuellen globa-

[5]Im Anhang befindet sich zur Konfliktvermeidung eine Liste an Anforderungen an neue Module

len Index generierten Referenzfronten, sowie auch alle Referenzfronten, die benötigt werden, um im aktuellen globalen Index Indikatoren zu generieren. Jede Referenzfront verfügt zusätzlich über ein am selben Ort liegendes Verzeichnis. Dort sind alle Hilfsdaten, die zum berechnen der Hilfsfront benötigt wurden, abgelegt. Der Grund für dieses Hilfsverzeichnis ist, dass Referenzfronten über die globalen Indices hinweg benötigt werden. So können sehr einfach die zur Referenzfront zugehörigen Daten ausfindig gemacht und kopiert werden. Eine Datei, in welcher eine solche Referenzfront gespeichert ist, trägt die Dateiendung `.rf`.

Bei dem Aufenthaltsort der Indikatordaten ist nun wichtig, dass die Referenzen zu den verwendeten Daten gegeben ist. Z.B. muss klar sein, welche Selektoren involviert sind und mit welchen Parametern der Indikator gestartet wurde. Darum ist das Verzeichnis `indicators/` bzw. die darunter liegende Ebene auch von grosser Bedeutung für die Erhaltung der Datenzusammenhänge. Sie wird, wie die globale Indexierung, auch indexiert. Der Verzeichnisname besteht aus dem Namen des Indikators, gefolgt von einen Unterstrich und einer natürlichen Zahl. Genau wie beim Prinzip der globalen Indexierung ist hier das Ziel, zusammen ausgeführte Berechnungen zusammenzuhalten. Ein solches Verzeichnis bezieht sich auf die verwendete Referenzfront sowie die Parametereinstellungen. Haben also zwei Verzeichnisse denselben Index, können wir davon ausgehen, dass die darin liegenden Indikatorwerte mit derselben Referenzfront generiert wurden. Falls auch der Name des Indikators gleich ist, wissen wir, dass für den jeweiligen Indikator auch dieselben Parameter verwendet wurden. Leider haben wir umgekehrt aber keine Information darüber, welche anderen Indikatoren mit derselben Parameterkonfiguration und demselben Referenzset existieren. Man könnte auch sagen, die Verzeichnisse beziehen sich nur auf die Entstehung der jeweiligen Indikatoren, nicht aber auf deren Inhalt. Auch hier wird der Aspekt, die Gliederung intuitiv zu gestalten, der logischen Gliederung vorgezogen. Die Indexierung wird verwendet, um zusammengehörige Daten am selben Ort zu bewahren und die danach mit statistischen Tests ausgeführten Daten auf Ihren Ursprung zurückführen zu können. Die Verzeichnisse enthalten die Zwischenresultate, die jeweils benötigt wurden um die Indikatoren zu ermitteln, sowie die Indikatordaten selber. Der Dateiname einer solchen Indikatordatei besteht aus dem Variatornamen, Selektornamen, Indikatornamen separiert durch Unterstriche gefolgt von einem Punkt und der Generation die verwendet wurde, um die Daten zu generieren.

Nach dem gleichen Prinzip funktionieren die Unterverzeichnisse von `statistics/`. Auch sie sind nummeriert. Der jeweilige Index bezieht sich auf die vom Test verwendete Parameterkonfiguration. Der Dateiname besteht aus dem Variatornamen, Indikator und Testnamen durch Unterstriche aneinandergefügt, gefolgt von einem Punkt und der verwendeten Generation. Aus diesem Namen geht nicht hervor, welche Selektoren für den jeweiligen

Test verwendet wurde. D.h. eine derartige Rekonstruktion der Beziehungen von Daten ist verunmöglicht(man könnte natürlich alle verwendeten Selektornamen anhängen, daraus resultierten aber unter Umständen sehr lange Dateinamen). Aus diesem Grund wird in diesem Verzeichnis auch eine Datei erzeugt, welche die verwendeten Selektoren in der richtigen Reihenfolge auflistet. Die Datei trägt den gleichen Namen, wie die, welche die Daten über den Test enthält, gefolgt von einem `_log`.

Verwendete Prinzipien Im Allgemeinen habe ich beim Modellieren dieser Verzeichnis- und Dateistruktur darauf geachtet, dass möglichst viele Beziehungen zwischen den Daten gespeichert werden können.

Nebenbei ist ein Konzept entstanden, welches dem Benutzer intuitiv einen guten Überblick verschafft. Dieser Grundsatzentscheid basiert auf dem Inhalt der Daten. Auf diese Weise wird für den Benützer eine sehr gute Übersicht vermittelt. Wichtig ist, dass zusammengehörige Daten nicht zu weit auseinander liegen. Weiter ist eine chronologische Abfolge der ermittelten Daten gegeben, was dem Benützer intuitiv auch helfen wird, die Generierung von Daten nachzuvollziehen.

Weiter ist die oben spezifizierte Struktur gegenüber einer logischen Struktur sehr flach. Auch dies ist dadurch zu begründen, dass zusammen erstellte Daten sich am selben Ort befinden sollen. Die Ziele, gute Beziehungen zwischen den Daten und deren Nachvollziehbarkeit, wurden also der allgemeinen Logik der Daten vorgezogen.

Beziehungen bzw. Referenzen zu anderen Daten soweit als möglich in die Dateinamen eingebunden und dafür nicht eine Log-Datei verwendet. So bleibt es einfach möglich, Daten manuell in die Verzeichnishierarchie einzubinden ohne mühsames Registrieren.

Die Verzeichnisstruktur ist vollumfänglich erweiterbar, d.h. ein Programm welches sich in der obigen Verzeichnishierarchie bewegt, sollte nicht davon ausgehen, dass alle Dateien oder Verzeichnisse an einem bestimmten Ort eine bestimmte Bedeutung haben, sondern nur gewisse Daten an einem gewissen Ort voraussetzten.

2.1.2 Alternativen zur Verzeichnisstruktur

VSP-Parameter-1run Der gegenüber den Daten *logische* Ansatz wäre, für jedes Variator-Selektor-Paar mit einer Parameterkonfiguration und einem Lauf ein Verzeichnis zu erstellen. Dies wäre ein Blatt im Verzeichnisbaum. Die nächste darüber liegende Ebene enthielte für dieses VSP und die Parameterkonfiguration alle Läufe. Und das wiederum darüber liegende Vaterverzeichnis enthielte alle zu diesem VSP existierenden Parameterkon-

figurationen. Als oberstes Verzeichnis dieser Datenstruktur fände man einen Pool mit allen VSP. Eine andere Möglichkeit wäre, noch mal eine Ebene einzuführen, d.h. jeder Variator bildet das Vaterverzeichnis für die darin liegenden Selektorpartner.

Da sich Referenzfronten, Indikatorwerte und statistische Tests immer auf mehrere Läufe beziehen, würde dies für jedes dieser Module einen Umstand ausmachen, alle Blätter des Baumes zu durchforschen und alle gefundenen Resultate einer bestimmten Generation einzulesen und aneinander zu knüpfen. Die berechneten Daten würden dann in die zweit tiefste Ebene im Verzeichnisbaum kommen, d.h. für ein VSP mit bestimmten Parameterkonfigurationen und mehreren Läufen. Wollte man aber einen statistischen Test ausführen, müsste man eine Ebene höher arbeiten, denn man will ja zwei verschiedene Algorithmen gegeneinander testen. Dies würde allerdings zu einer Verteilung der ausgewerteten Daten führen und die Referenzen zu verwendeten Indikator und Referenzfronten erheblich komplexer gestalten. Ein weiterer Nachteil ist, dass der Austausch von bestimmten Daten zu einem anderen Benutzersystem sehr schwierig ist. Es müssen verschiedene Verzeichnisse durchforscht werden, da die Trennung der verschiedenen VSP an der Wurzel des Baumes liegt. Einfachheitshalber könnte man auf das Ablegen von Auswertungen verzichten und nur ein globales Referenzfrontverzeichnis führen. Dies insofern Möglich, als dass die Auswertungen der Monitordaten im Vergleich zum tatsächlichen Ablauf eines EA sehr kurz dauert.

Dieser Ansatz verfolgt als Grundprinzip eine tiefe Struktur. Die Referenzen zu anderen Daten würden jedoch für dieses Prinzip kaum mehr durch den Dateinamen beschrieben werden können(natürlich funktioniert dies immer, bei zu langen Dateinamen bietet sich aber eher eine Log-Datei oder ein Zwischenverzeichnis an). Die so abgelegten Daten vermittlen kaum einen intuitiven Zusammenhang für die Daten unter sich, d.h. die Nachvollziehbarkeit wäre sehr schlecht. Der Ansatz würde dem Benutzer eine gute Übersicht über die Menge aller gesamten ermittelten Daten liefern. Jedoch würde bei diesem Prinzip bei grösserer Menge an Daten die mittleren Ebenen(interne Knoten im Verzeichnisbaum) schnell überfüllt werden.

VSP-Parameter-n runs Eine leicht abgeänderte Form dieses Prinzips bringt schon wesentlich mehr Vorteile und kommt der tatsächlich verwendeten Hierarchie schon näher. Es ist eine Kompromisslösung zwischen totaler Übersicht über alle vorhandenen Daten und einer gewissen Struktur, welche erlaubt, ältere von neueren Daten zu unterscheiden.

Jeder Monitorlauf beschreibt ein Blatt des Baumes, das heisst für jedes VSP und n Läufen würde ein Verzeichnis angelegt. Zusätzlich könnte man hinter jedes dieser Verzeichnisse einen Laufindex anhängen, dies würde dem Benutzer eine bessere Übersicht verschaffen, da er sich an einer Chronologie

orientieren könnte. Das Problem der Auswertungen der Monitorläufe wäre jedoch noch nicht gelöst. Auch hier würde ein Laufindex wieder zu einem gewissen Mass Abhilfe schaffen, jedoch wäre der zu betreibende Aufwand, um die Referenzen zwischen den ermittelten Daten zu speichern, erheblich, da auch hier die Auswertungen auf verschiedene Ebenen Verteilt wären.

Dieser Ansatz ist um eine Ebene weniger tief als der gegenüber den Daten vollkommen logische Ansatz. Dafür ist die Datenablegung für den Benutzer intuitiver und besser nachvollziehbar. Intuitiver weil alle Daten eines Monitorlaufs flach in einem Verzeichnis liegen und nachvollziehbarer wird der Ansatz durch den angefügten Laufindex, welcher ein chronologisches Mass darstellt.

Allgemein sieht man, dass die Auftrennung zwischen den verschiedenen Variator-Selektor-Paaren in verschiedene Verzeichnisse in der Nähe der Wurzel des Verzeichnisbaumes das eigentliche Problem bereitet. Man möchte die Algorithmen gegeneinander Vergleichen, d.h. dass die statistische Auswertung verschiedene Daten aus mehreren Ebenen kombinieren muss. Dies bedeutet wiederum einen grossen Mehraufwand um Referenzen zwischen den Daten konsistent zu halten. Darum scheint der tatsächlich verwendete Ansatz, welcher die logische Hierarchie im Prinzip auf den Kopf stellt, sehr vernünftig. Als Wurzel dient nicht das eigentlich VSP, sondern eine künstliche Ebene, welche die Chronologie repräsentiert. In den Blättern des Baumes finden wir dann alle Monitorläufe mit gleichen Parametern von mehreren VSP flach im selben Verzeichnis. Dadurch sind auch alle statistischen Auswertungen auf dieselbe, nämlich unterste, Ebene des Baumes bezogen, was uns eine relativ einfache Konsistenthaltung der Referenzen ermöglicht.

2.2 Das ControlTool

Das ControlTool wurde mit Java programmiert. Dies erstens weil die Plattformunabhängigkeit des Tools ein Designziel darstellt und zweitens, weil Java objektorientiert ist und für GUI Anwendungen sehr viele nützliche vorprogrammierte Klassen zur Verfügung stellt. Im Weiteren soll das ControlTool parallelisert ablaufen, d.h. das Ziel ist, dass verschiedene Module bzw. deren Prozesse gestartet werden können und das Tool weiterhin verwendet werden kann, d.h. derweilen nicht blockiert ist. Java bietet auch hier mit Multithreading eine einfache und saubere Lösung an.

2.2.1 Die Klassenhierarchie

In diesem Abschnitt werden die wichtigsten Klassen vorgestellt. Es sollen auch nicht einzelne Methoden, sondern eher die Aufgabenbereiche präsentiert werden. Weiter möchte ich grundsätzliche Entscheide betreffend der Hierar-

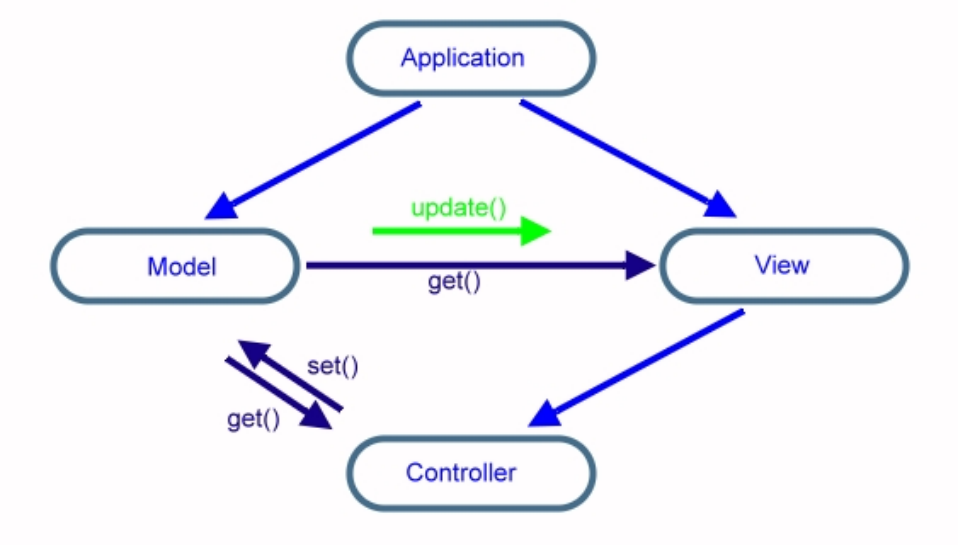

Abbildung 4: Eine Möglichkeit für ein MVC Projekt

chie der einzelnen Klassen und deren Trennung dokumentieren.

Ein Ziel beim Design einer Klassenhierarchie ist eine logische Trennung der Datenstrukturen und Funktionalitäten. Eine gute Umsetzung dieses Ziels garantiert auch eine gute Übersicht und eine gute Leserlichkeit für andere Programmierer. Auf der anderen Seite läuft man bei einer zu starken Trennung der Funktionalitäten in Klassen Gefahr, dass für die Kommunikation zwischen den einzelnen Klassen zu grosse Kosten entstehen. Zum einen wird durch die Kommunikation zwischen Klassen vieles komplizierter, zum anderen wird aber auch die Rechnerleistung eingeschränkt, vor allem wird der Hauptspeicher stark beansprucht(ein in Java bekanntes Problem). Ein anderes Ziel ist, die Grundstruktur der Klassen so zu erstellen, dass sie einfach und schnell erweiterbar ist. Z.B. soll ein neues GUI erstellt werden können, ohne dabei Veränderungen am Datenmodell vorzunehmen und umgekehrt. Dafür wurde das MVC(Model, View, Controller) Konzept verwendet. Dies soll hier kurz erläutert werden.

MVC MVC[6] übernimmt Java von Smalltalk und ist heute ein sehr verbreitetes Konzept. MVC nimmt eine klare Trennung zwischen dem Datenmodell einer Anwendung, graphischen Teil und dem für den Ablauf des Programms zuständigen Teil, dem Controller vor. Dies bietet den Vorteil, dass alle drei Teile unabhängig voneinander weiterentwickelt werden können.

In Abbildung 4 ist das Grundprinzip des Aufbaus vom Tool abgebildet.

[6]The Model-View-Controller (MVC) Its Past and Present Trygve Reenskaug, University of Oslo, http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf

Es besteht aus 4 Teilen. Unter den vielen Möglichkeiten, dieses Konzept umzusetzen, wird hier eine kurz skizziert:

Die Anwendung Sie ist im Wesentlichen Verantwortlich für den Programmstart. Sie erstellt eine Instanz aller nötigen Datenmodelle. Diese Instanz wird als Referenz dem nachher konstruierten GUI weitergegeben.

Das Datenmodell Wie der Name sagt, werden hier Daten eingelesen, verwaltet, manipuliert und können abgefragt werden. Dabei schreibt das MVC Konzept vor, dass die verschiedenen Daten mit set() und get() Methoden manipuliert bzw. abgefragt werden können. Was innerhalb des Datenmodells abläuft, bzw. in welcher Form die Daten verarbeitet werden, ist für einen GUI Designer uninteressant, er operiert nur über die set() und get() Methoden.

Die View Die View übernimmt mit dem Controller den restlichen Teil einer Applikation nach MVC. Dabei ist die View verantwortlich für alles was auf dem Bildschirm erscheint. Nicht aber über die Abfolge der einzelnen Aktionen. Dafür ist der jeweils zu einer View gehörende Controller verantwortlich. Er wird von der View aufgerufen, und erhält von ihr eine Referenz auf das Datenmodell wie auch eine Referenz auf die View selber.

Der Controller Er ist zuständig für alle Interaktionen mit dem Benutzer. Das heisst, dass z.B. bei einem Mausklick der Controller überprüft, welches Objekt angeklickt wurde. Nachher werden alle nötigen Aktionen mit dem Datenmodell abgearbeitet. Wenn dies geschehen ist, wird der Controller die View weiterverarbeiten, d.h. zum Beispiel ein anderes Fenster öffnen.

Dieses so skizzierte Konzept hat einen schweren Mangel. Der Controller muss von allen verschiedenen Fenstern(bzw. von allen GUI-Klassen, meistens wird in Java aber pro Fenster eine GUI Klasse implementiert) wissen, was in ihnen angezeigt wird. Dies ist bei einer komplexen GUI Applikation fast unmöglich. Zweitens muss so jeder Controller alle nötigen Änderungen am gesammten visuellen Erscheinungsbild vornehmen, was das Designziel Erweiterbarkeit praktisch verunmöglichen würde. Es wäre besser wenn die View direkt von dem Datenmodell informiert würde und alle in der View liegenden Klassen darauf reagieren könnten. So wären alle GUI-Klassen und ihre Controller voneinander unabhängig. Dabei ist es unwichtig ob ein Update einer View nach einer Veränderung des Datenmodells vom Controller, oder von der View selber ausgeführt wird. Normalerweise wird dies aber vom Controller ausgeführt.

Um genau diese Kommunikation zwischen dem Datenmodell und der View zu gewährleisten, unterstützt Java das MVC-Konzept mit dem Interface `java.uitl.Observer` und der Klasse java.util.Observable. Die Klasse Observable wird an das Datenmodell bzw. an alle darin liegenden Klassen vererbt. Nun können alle anderen Klassen am Datenmodell horchen und bei neuen Daten ihre Veränderungen selber durchführen. Dabei spielt es für das Datenmodell keine Rolle, wie viele andere Klassen am horchen sind. Man kann also das Tool ohne Problem um ein GUI ausbauen, ohne grössere Strukturveränderungen vorzunehmen. Die neuen Klassen müssen sich nur beim Datenmodell registrieren um über neue Daten informiert zu werden. Welche Nachrichten an die vom Observable intern geführte Verteilerliste gesendet werden, ist natürlich vom Programmierer definiert. Mit den beiden Aufrufen

```
this.setChanged() // setzt ein Flag um
this.notifyObservers(Message m) // versendet die Nachricht
```

wird vom Datenmodell aus so eine Nachricht verschickt. Im Gegenzug versprechen alle Observers, dass sie eine Methode update(Observable o, Object arg) implementiert haben, sodass sie auf die Meldungen des Datenmodells eingehen können. Dies ist ein gutes Beispiel für den Mehraufwand den man betreiben muss, um die Kommunikation zwischen den einzelnen Klassen zu gewährleisten. Im Gegenzug entsteht unter einer MVC Struktur programmierten Applikation ein äusserst schnell und einfach ausbaubares Programm und auch ein sehr übersichtlicher Programmcode. Genau aus diesem Grund habe ich dieses Konzept gewählt um das ControlTool zu strukturieren. Der dabei entstehende Mehraufwand der zur Kommunikation aufgebracht werden muss, ist im Vergleich zur Rechenzeit und Speicheraufwand der einzelnen Algorithmen vernachlässigbar.

2.2.2 Die Klassenhierarchie des ControlTool

Es gibt viele Möglichkeiten das MVC Konzept umzusetzen. Im Wesentlichen hält das Tool aber an dem oben skizzierten Schema fest. Ich möchte nicht zu detailliert auf jede einzelne Klasse eingehen, der interessantere Teil beschränkt sich auf den Applikationsteil (*Application*) sowie auf das Datenmodell(*Model*). Die *View* und die zugehörigen *Controller* sind sehr spezifisch auf das Design des GUIs bezogen und aus programmiertechnischer Sicht uninteressant. Zur Übersicht dient Abbildung 5, welche die Klassen zu ihrem zugehörigen MVC Teil aufzeigt. Eines der nicht ganz gelösten Probleme im MVC Prinzip entsteht gleich beim Programmstart. Nämlich braucht das Datenmodell schon Informationen vom Benutzer um überhaupt konstruiert zu werden, in unserem Fall den Projektnamen. Wie soll nun eine View mit dem globalen Datenmodell erstellt werden, wenn dieses eine Information der View

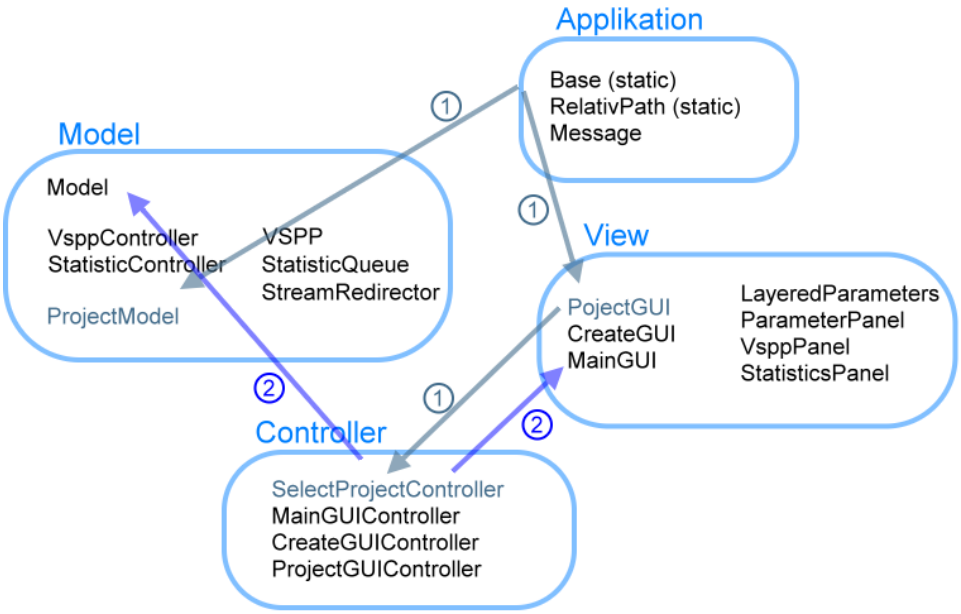

Abbildung 5: Die Klassenhierarchie des ControlTool. Die Zahlen beschreiben die Abfolge der Instanzierung. Weiter sind alle Klassen nach MVC klassifiziert. Die View ist gegliedert in Fensterklassen(Frames) und Hilfsklassen, sog. Bohnen welche abgekapselt wurden, damit sie wiederverwendbar sind. Auf den Aufbau des Model wird später eingegangen.

benötigt? Es ergibt sich eine Deadlock Situation. Um dieses konzeptionelle Problem zu lösen, kann man ein eigentlich vom Rest abgetrenntes MVC-Konstrukt bauen. Dieses verfügt über eine kleine, separierte Datenstruktur, einer View und einen separierten Controller. Eigentlich ein eigenständiges Programm welches im selben Java-Package eingebunden ist. Nach der Benutzereingabe(Selektion des Arbeitsraumes, nämlich das Project) wird als zweiter Schritt der Controller von diesem Vorprogramm das eigentliche Datenmodell Model und das MainGUI starten. Das separierte Programm besteht im ControlTool aus der Datenmodellklasse `ProjectModel`, der View `SelectProjectGUI` und aus dem Controller `SelectGUIController`.

Der Applikationsteil Der Applikationsteil besteht aus 3 Klassen. Die Klasse `Base` enthält die main-Methode, wo das Tool gestartet wird. Sie übernimmt das Starten des separierten Vorprogramms. Des Weiteren bietet sie eine Vielzahl von nützlichen Methoden, welche von allen Programmteilen verwendet werden können, da die Klasse und somit auch alle ihre Methoden statisch definiert sind, können alle Teile des Programms diese Methoden abrufen. Z.B. kann mit Hilfe dieser Methoden Dateien kopiert oder aneinandergefügt werden. Eine andere Möglichkeit wäre, diese statischen Methoden in das Datenmodell einzubauen.

Die Klasse `RelativePath` verfügt über nur eine statische Methode, welchen den relativen Pfad aus 2 absoluten Pfaden berechnet(eine Grundidee von Java ist, dass nicht 2-mal der gleiche Code von verschiedenen Autoren geschrieben wird, sondern dass solche Standardklassen verteilt und wieder verwendet werden. Die RelativePath Klasse wurde von einem anderen Autor geschrieben und wird deshalb als eigene Klasse gehalten). Die dritte Klasse in dieser Gruppe heisst Message. Sie definiert eine programminterne Kommunikationsschnittstelle. Eine Message enthält einen Namen vom Typ String und ein Argument vom Typ Object. Zum Beispiel wird eine Message verwendet um zwischen dem Datenmodell und der View via Observer Interface zu kommunizieren. Man könnte diese Klasse auch zum Datenmodell hinzuzählen, aber auch sie ist nicht Datenmodell abhängig und wird deshalb in diesem Rahmen zum Applikationsrahmen hinzugezählt(eigentlich gehört diese Klasse wie auch die Klasse RelativePath zu keinem von MVC definierten Teil; es sind eher sogenannte utility-Klassen und können kaum nach MVC klassifiziert werden).

Das Datenmodell (Model) Das Datenmodell ist das Herzstück des ControlTool. In unserem Fall ist das Datenmodell eine Schnittstelle zwischen der Verzeichnishierarchie und den GUI-Controllern. In diesem speziellen Fall aber unterstützt das Datenmodell keine set() Anweisungen, da es solche nicht gibt. Es kann nur dazu aufgefordert werden, Berechnungen vorzuneh-

men, welche dann direkt auf der Festplatte gespeichert werden. Mit der spezifizierten Verzeichnishierarchie zusammen resultiert so im Grunde nichts anderes, als eine plattformunabhängige, persistente Datenbank. Falls neue Daten auf der Festplatte liegen, werden alle Observer dieses Datenmodells darüber informiert.

Um die Klassen zu beschreiben, beginne ich mit den kleinsten Objekten(VSPP) und fahre mit der Verwaltung von ihnen fort. Nachfolgend wird die Klasse StatisticController, welche für die Auswertungen der generierten Daten verantwortlich ist, beschrieben. Zum Schluss beschreibe ich, wie die parallelisierten Vorgaenge mit Hilfe der `StatisticQueue` Klasse abgefangen und organisiert werden.

Wenn das Datenmodell aus mehreren Klassen besteht(so auch im ControlTool) müssten beim konstruieren von View- und Controllerklassen immer Referenzen auf alle die Objekte im Datenmodell weitergegeben werden. Wenn jetzt eine neue Datenmodellklasse hinzukommt, müssten alle Konstruktoren der View- und Controllerklassen auch erweitert werden.

Um eben dieses Problem zu umgehen und eine einfache Erweiterung des Datenmodells zu garantieren, wird eine künstliche Klasse `Model` konstruiert, welche von allen Klassen aus dem Datenmodell eine Referenz speichert. Dies erleichtert das Weitergeben der Referenz auf das Datenmodell zwischen dem Applikationsteil und den GUI-Klassen sowie auch zwischen den GUI-Klassen und den jeweiligen Controllern. Anstatt mehrere Referenzen auf alle Klassen weiterzugeben, muss so nur eine auf die Instanz der Klasse `Model` übergeben werden.

Die Klasse `Vspp` definiert ein Variator-Selektor-Paar mit einer bestimmten Parameterkonfiguration(VSPP). Sie hält Informationen über solche VSPPs fest, z.B. die Anzahl der Läufe oder den Namen des zugehörigen Variators und Selektors. Der Beschluss für ein VSPP eine eigene Klasse zu definieren ist unumgänglich, weil die Instanzen VSPPs parallel verarbeitet werden sollen. So implementiert die Klasse das Interface `java.lang.Runnable` welche die Klasse Thread fähig macht. Auch dieser Beschluss ist unumgänglich für eine Parallelisierbarkeit. Die Klasse hat die Aufgabe 3 neue Prozesse zu starten, nämlich den Variator-, Selektor- und Monitorprozess, und zu warten bis diese beenden. Dabei werden auch die Standard- sowie Fehler-Outputstreams umgelenkt.

Die einzelnen Vspp-Objekte müssen jetzt noch dynamisch gespeichert werden. Dynamisch heisst, dass wir nicht wissen, wieviele dieser Objekte existieren werden. Dafür ist die Klasse VsppController zuständig.

Wie in Abbildung 7 gezeigt, werden die einzelnen Vspp Instanzen von der Klasse VsppController(die Namensgebung ist ein bisschen verwirrend, der VsppController sowie der StatisticController haben mit dem Controller

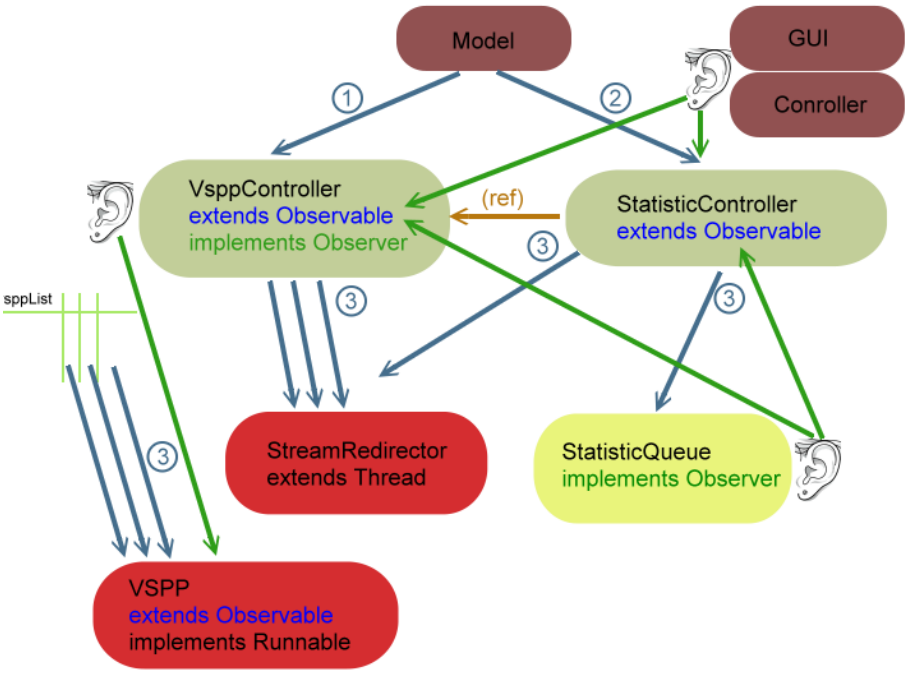

Abbildung 6: Die Zahlen beschreiben den zeitlichen Ablauf der Instanzierungen im ControlTool. Die grün und gelb eingefärbten Klassen werden nur einmal Instanziert. Die 2 rot eingefärbten Klassen entspringen aus der Klasse Thread und werden mehrere Male instanziert. Die Graphik ist von oben nach unten zu lesen. Die oberste Ebene ist die Klassifizierung nach MVC. Die beiden grünen Klassen sind der Hauptbestandteil des Model. Sie enthalten den aktuellen Bestand aller Daten, welche in einer java.util.Set oder in einer java.util.Map Kollektion gespeichert sind. Sie unterstützen die get()-Methoden für das GUI und können von allen anderen Klassen abgehört werden. Die 3 darunterliegenden Klassen sind Hilfsklassen fürs Datenmodell.

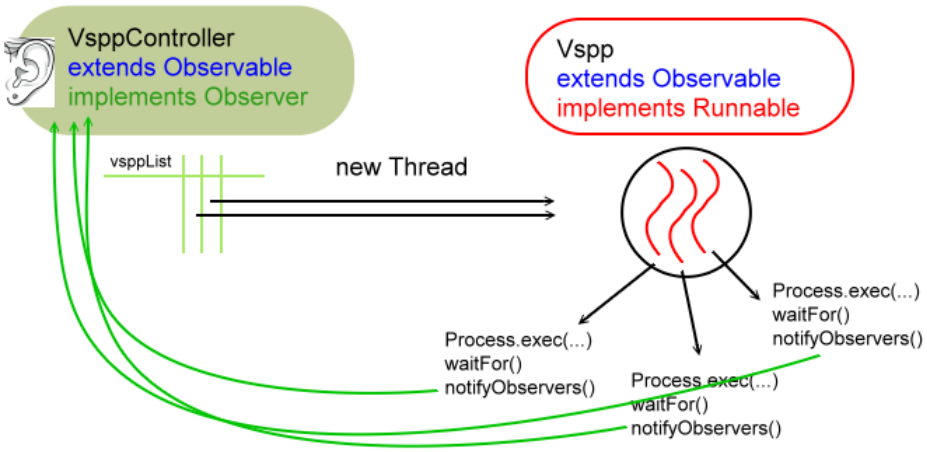

Abbildung 7: Die Graphik soll illustrieren, wie der VsppController neue Vspp-Threads instanziert. Diese Objekte können dann mit der run() Methode angewiesen werden, einen Prozess zu starten. Pro angesteuertes Modul wird also ein Thread und ein separater Prozess gestartet. Der Thread wartet dann bis der Prozess zu Ende ist und informiert wiederum den VsppController über die Neuigkeiten. Dieser wird die Message an die Controller und die View verschicken.

Teil der MVC Klassifikation nichts zu tun) verwaltet. Der Aufgabenbereich dieser Klasse zieht sich über das Führen einer Liste aller berechneten Vspps. Das heisst, beim Start des ControlTool werden alle Monitorausgabedateien auf der Festplatte durchsucht und in eine Liste, der vsppList eingetragen. Das VsppController Objekt (die Klasse wird nur einmal instanziert) startet die einzelnen Vspp-Threads. Falls ein Vspp-Thread beendet, wird der Vspp-Controller darüber informiert und bindet die Vspp Instanz bzw. eine Referenz darauf in die List ein. Danach kommuniziert die Instanz dieser Klasse an alle an ihr horchenden Objekte via Observer Interface. Wie oben beschrieben können die Controller die Vspp Bestände per `get()` Anweisung abfragen. Hingegen gibt es keine `set()` Anweisung, nur den Befehl `launchVspp()` welche einen Monitorlauf startet. Dies ergibt sich aus dem Grund, dass bei unserem Problem keine internen Daten(Daten im RAM)verändert werden sollen, sondern nur auf die Festplatte geschrieben wird.

Beim Prozessstart durch einen Java-Thread hat sich ein Problem betreffend dem Outputstream des aufgerufenen Prozesses offenbart. Dieser OutputStream muss regelmässig geleert werden. Diese Aufgabe übernimmt die im Model enthaltene StreamRedirector Klasse. Sie agiert auch als Thread. Für detailliertere Dokumentation möchte ich auf den Implementierungsteil verweisen.

Nun müssen noch alle Auswertungen der Daten, d.h. Referenzfronten, Indikatorwerte und statistische Tests, verwaltet werden. Dafür ist die Klasse `StatisticController` zuständig. Sie verfügt über eine Referenz des Vspp-Controller und weiss somit auch, wie die VSPP Bestände sind. Mit den Methoden `calculateReferenceFront()`, `calculateIndicators()` und `calculateTest()` anstelle von `set()` Operationen können die Controller die verschiedenen Berechnungen ausführen lassen, welche auch wieder direkt auf der Festplatte gespeichert werden. In dieser Klasse wird auch das Ziel umgesetzt, die Module zur Auswertung einzeln zu steuern. So unterstützt die Klasse Methoden wie `bound()`, `normalize()` und `filter()`. Wenn die Berechnungen fertig sind, informiert der StatisticController wiederum alle Observer. Mit get() Methoden können die verschiedenen Auswertungsbestände abgefragt werden.

Nehmen wir an, ein Benutzer möchte alle Einstellungen vom Monitorlauf bis zum letzen Parameter im statistischen Test zum vornherein tätigen und alles automatisiert starten lassen. Dann haben wir das Problem, dass wir alle nötigen Resultate für eine Auswertung abwarten müssen. Derweilen müssen die hängenden Auswertungen zwischengespeichert werden.

Dieser Aufgabe widmet sich die Klasse `StatisticQueue`. Sie ermöglicht der View bzw. deren Controllern einen vollautomatischen Ablauf von Monitorläufen, den Referenzfrontenberechnungen, den Indikatorberechnungen und den statistischen Tests. Das heisst, die StatisticQueue horcht am Vspp-Controller wie auch am StatisticController und überprüft ob neue Resultate vorliegen. Falls ja, wird überprüft ob genug Daten berechnet wurden um mit den Auswertungen der Daten fortzufahren. Falls nicht, wartet die StatisticQueue weiter. Die Klasse gleicht einer Warteschlange für offene Auswertungen. Sie folgt aber keinem Prinzip wie FIFO oder LIFO sondern berechnet immer sofort alle mit den gegebenen Daten möglichen Auswertungen. Für eine genauere Beschreibung der Implementierung möchte ich auf den Implementierungsteil verweisen.

Die hier vorgestellte und auch implementierte Lösung zur Aufteilung der Klassen im Datenmodell gibt dank MVC eine gute Übersicht und ist sehr einfach erweiterbar. Es können sehr einfach neue Klassen hinzugefügt werden, dazu muss wie oben beschrieben nur eine Referenz in der Klasse Model abgelegt werden.

Weil die Model Instanz(auch diese Klasse wird nur einmal instanziert) nach dem MVC Modell auf jedes konstruierte Objekt mitgegeben wird, haben alle anderen Klassen im gesamten Package (ausser den Klassen im Datenmodell selber) automatisch auch eine Referenz auf die neu implementierte Klasse. Dies ermöglicht das Ausbauen von schon vorhandenen GUIs sehr komfortabel.

Des Weiteren entspricht die Aufteilung einer intuitven Art, das Gesamt-

paket(das ControlTools und alle angesteuerten Module) zu betrachten. Man hätte auch den VsppController und den StatisticController in eine Klasse packen können. Aus Übersichtsgründen und Trennung von Funktionalitäten wurde aber eine Auftrennung bevorzugt. Die Nachteile des MVC Prinzip sind im Wesentlichen bekannt und haben sich auch bei der Implementierung des ControlTool gezeigt. Zum einen muss ein gut ausgedachtes Konzept der Messageverbreitung ausgearbeitet und implementiert werden. Für kleinere Anwendungen resultiert aus der Sicht des Informatikers ein grosser Kostenaufwand(engl. Overhead) an im Grunde genommen überflüssigen Objekten. Würde man auf das MVC Prinzip verzichten, also z.B. alles sehr kompakt und auf das zu lösende Problem bezogen programmieren(z.B. die ganze Anwendung in 2 Klassen verpacken, was theoretisch möglich ist), wäre eine komplexe Anwendung sicherlich wesentlich schneller und würde vor allem weniger Hauptspeicher benötigen. Natürlich wäre dann das Programm weder gut erweiterbar noch übersichtlich.

Bemerkung zu den Controller Klassen So wie die Datenmodellklassen aufgebaut sind, muss ein Controller sehr genau wissen, wie das Datenmodell zu manipulieren ist. Zum Beispiel muss der Controller das Datenmodell dazu anweisen, ein neues GlobalIndex Verzeichnis zu erstellen, bevor die VSPPs gestartet werden. Geschieht dies nicht, werden möglicherweise vorhandene Daten überschrieben. Das liegt aber auch in der Natur einer persistenten Datenbank. Dies ermöglicht einerseits dem Controller vollumfaengliche Kontrolle über die Datenstruktur, andererseits aber trägt ein Controller viel Verantwortung.

3 Implementierung

In diesem Kapitel möchte ich erläutern, wie die wichtigsten Mechanismen im ControlTool funktionieren. Dazu gehört die möglichst effektive Datenablagerung in den RAM. Dabei sollen die Zusammenhänge der Daten, welche aus der Verzeichnishierarchie resultieren, auch gespeichert werden. Ein weiteres aus programmiertechnischer Sicht interessantes Konstrukt ist die Klasse `StatisticQueue`, welche die Vspp-Threads abfängt und die berechneten Daten zur Auswertung bereitlegt. Zum Schluss soll noch kurz ein sehr Java-spezifisches Problem mit Prozessstarts dargestellt werden.

3.1 Die Datenstrukturen

Als Erstes stellt sich die Frage, wie viele Daten in den Hauptspeicher geladen werden sollen. Diese Entscheidung hängt stark damit zusammen, wie viele

Daten überhaupt vorliegen. In unserem Fall sehr viele. Dann würde sich zuerst die Option anbieten, die möglichst nur die Daten im RAM aufbewahrt, welche gerade angezeigt werden. Der grosse Nachteil an diesem Ansatz ist aber, dass sich der Programmablauf verlangsamen würde, da ein Zugriff auf die Festplatte sehr lange dauert.

Als eine Alternative dazu steht das Prinzip der Indexierung. Man könnte jedes Objekt mit einem Index versehen, dies ermöglicht eine sehr schnelle Suche der Daten und ist zusätzlich Platz sparend.

Eine weitere Alternative wäre, ein Datenbankobjekt zu kreieren. Ein grosser Vorteil wäre die einfache Handhabung der Daten. Weil die Daten immer in einer human-readable Form auf der Festplatte liegen sollen, müsste das Erstellen der Datenbank in unserem Fall immer beim Programmstart geschehen, was sehr viel Zeit in Anspruch nehmen würde. Ausserdem hätte man, weil die Daten ja sowieso auf der Disk liegen, eine grosse Datenredundanz. Ein zweiter Grund auf diese Methode zu verzichten ist, das ControlTool möglichst einfach und vor allem plattformunabhängig zu halten.

Aus diesen Gründen habe ich mich für eine Indexierung der vorliegenden Dateien entschieden. Um solche Datenmengen zu verwahren, bieten sich in Java5 die Collections-Klassen an. Die neuen generischen Datentypen von Java[7] zeigten sich als sehr nützlich für die Implementierung des ControlTool.

Das Datenmodell im ControlTool verfügt im Wesentlichen über 3 verschiedene vorliegende Datenobjekte. Die Ausgabe der Monitorläufe(VSPP), Indikatorresultate und die berechneten statistischen Tests:

Die VSPPs werden vom VsppController verwaltet. Er führt eine sogenannte `vsppList` vom abstrakten, generischen Datentyp

```
java.util.TreeMap<Vector<Integer>,Vspp>
```

Diese Java Collection bildet einen Vektor mit ganzzahligen Einträgen auf eine Vspp Instanz ab. Der Vektor ist 3-dimensional und dient uns zur Indexierung, er wird im Allgemeinen der *Schlüssel* genannt. Der erste Eintrag bezieht sich auf den GlobalIndex, worin sich die vom Monitor generierten Daten befinden. Die zweite Zahl indexiert den Variator, die dritte den Selektor. Dieses 3-tuple genügt um eine VsppInstanz zu referenzieren.

Damit die Daten im Bezug auf den GlobalIndex geordnet im RAM liegen, habe ich die TreeMap verwendet. Die TreeMap Collection implementiert das `java.util.SortedMap` Interface nach einem Rot-Schwarz-Baum, so finden wir unsere Vsppobjekte innerhalb logarithmischer Zeit in Abhängigkeit der vorhandenen Vspps. Mit Hilfe einer sortierten Collection müssen die Daten nicht immer zuerst sortiert, bevor sie angezeigt werden. Die Klasse TreeMap

[7]Ein Tutorial zu den generischen Datentypen von java findet man unter: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

ist abstrakt; zu einer Instanzierung benötigt sie einen definiertes Vergleichsobjekt vom Interfacetyp `java.util.Comparator<Vector<Integer>>`, welcher für die Sortierung der Schlüssel zuständig ist. Die hier implementierte Form sortiert die Vektoren indem jeweils die ersten Einträge verglichen werden, dann die 2ten usw.

Eine Alternative zu dieser Wahl liegt auf der Hand. Ein 3-dimensionales Vspp Array. Der Nachteil an dieser Variante wäre aber, dass das Array bei jedem neuen Vspp-Objekt kopiert werden müsste(Java löst intern das Problem natürlich auch mit einem Array, nur wird dort schon im voraus immer das 1.5fache an Platz reserviert, sodass das Array nicht immer kopiert werden muss; ausserdem wird dem Programmierer so viel Arbeit erspart).

Die Indikatordaten bzw. Dateien werden in ähnlicher Form in der "indicatorList" im RAM gehalten. Die Dateien werden auch wieder indexiert aber nicht auf ein Objekt abgebildet. D.h. es wird nur der Bestand der aktuellen Dateien angeschaut, nicht aber deren Inhalt. Diesmal wird die Java-Collection `java.util.HashSet<Vector<Integer>>` verwendet. Das Set ist eine Konstruktion welche keine doppelten Elemente speichert. Die darin enthaltenen Objekte sind 5-dimensionale ganzzahlige Vektoren. Die ersten 3 Zahlen entsprechen der Indexierung eines Vspp-Objektes. Es können also nur Indikatorwerte vorhanden sein, von welchen auch die zugehörigen VSPP Daten vorliegen. Die 4 Zahl indexiert den verwendeten Indikator und die 5te Zahl representiert die in der Verzeichnishierarchie vorgestellte Indikatorlaufnummer(IndicatorGlobalIndex).

Die statistischen Tests wiederum sind in Form einer

`java.util.HashMap<Vector<Integer>,Vector<Integer>>`

im RAM abgelegt, nennen wir die HashMap `statisticList`. Ein 6- dimensionaler ganzzahliger Vektor bildet auf einen 1-dimensionalen ganzzahligen Vektor ab. Ein Schlüsselvektor enthält dieselben Einträge wie ein indicatorList Eintrag ohne den Selektorindex, da sich ein Test immer auf mehrere Selektoren bezieht. Die statisticList bildet aber auf einen Vektor ab, welcher alle Indices der im Test verwendeten Selektoren enthält. Auf diese Weise projiziert die statisticList bijektiv auf die vorhandenen Daten. Man hätte auch den Selektor in den Schlüssel einbinden können, doch dann hätte man in der statisticList mehrere Einträge, welche auf dieselbe Datei verweisen. Die statistischen Tests müssen nicht sortiert sein, darum habe ich aus Geschwindigkeitsgründen die HashMap dem TreeMap-Konstrukt vorgezogen.

Mit diesen Kollektionen und der Indexierung wurde ein vernünftiger Kompromiss geschlossen zwischen Schnelligkeit und Platzaufwand im Hauptspeicher. Zusätzlich bieten die so implementierten Collectons einen grossen Komfort, wenn es darum geht, zugehörige Daten ausfindig zu machen. Zur Verdeutlichung eins aus hunderten von Beispielen: Man möchte gerne alle

statistischen Tests zu einem bestimmten Variator suchen. So reicht es, die statisticList in linearer Zeit auf das zweite Element im Schlüsselvektor zu durchsuchen.

3.1.1 Die Hilfsklasse StatisticQueue

Das Parallelisieren des Tools verursacht in unserem Fall 2 Probleme. Ich werde diese nacheinander kurz schildern und meine Lösungsvorschläge praesentieren:

Fehlende Daten Die Automatisierung von einem Monitorlauf bis zu den berechneten statistischen Test und die Parallelisierbarkeit sind zusammen schwierig umzusetzen. Das Problem liegt darin, dass das GUI dem Datenmodell die vom Benutzer eingestellten Parameterkonfigurationen für die Auswertungen vor dem Beenden der Vspp-Threads übergibt. Somit fehlen die für die Auswertung benötigten Daten, also müssen die angeforderten Berechnungen in eine Warteliste aufgenommen bzw. darin zwischengespeichert werden. Wenn ein VSPP-Thread beendet, muss die Schlange überprüfen, ob nun alle nötigen Daten vorliegen. Wenn ja, kann mit dem Auswerten begonnen werden.

Zur Lösung des ersten Problems baut die StatisticQueue eine Zwischenablage für Daten, welche noch ausgewertet werden müssen. Damit die Klasse über neue Resultate informiert wird, implementiert sie das Interface `java.util.Observer`. Die Klasse horcht an dem VsppController- und am StatisticController-Objekt, welche für die Controller- und Viewklassen sowieso Nachrichten verschicken. Die Abbildung 8 illustriert zusammen mit dem Pseudocode den Ablauf vom Beenden des Vspp-Thread bis zu den fertigen statistischen Tests. Die StatisticQueue führt für jedes Zwischenresultat zur Auswertungen von Daten eine eigene Datenstruktur. Für den korrekten Ablauf der Tests sind folgende 3 Methoden zuständig:

Die Methode lookForReferenceFront() Die Methode durchsucht im die Datenstruktur:

```
HashMap<Set<Vector<Integer>>, ReferenceFronts> referenceFrontQueue
```

Der Schlüssel der Map ist ein Set aus allen nötigen Vspps zum berechnen einer Referenzfront. Die Werte der Map sind in der StatisticQueue interne Klassen, welche alle Werte zum Neustart einer Berechnung enthält. Zudem verfügt die interne Klasse über die Methode restart(), welche die Referenzfront wieder an den StatisticController schickt. Um zu sehen ob eine Referenzfront mit dem neuen Indexvektor `v` berechnet werden kann, wird die

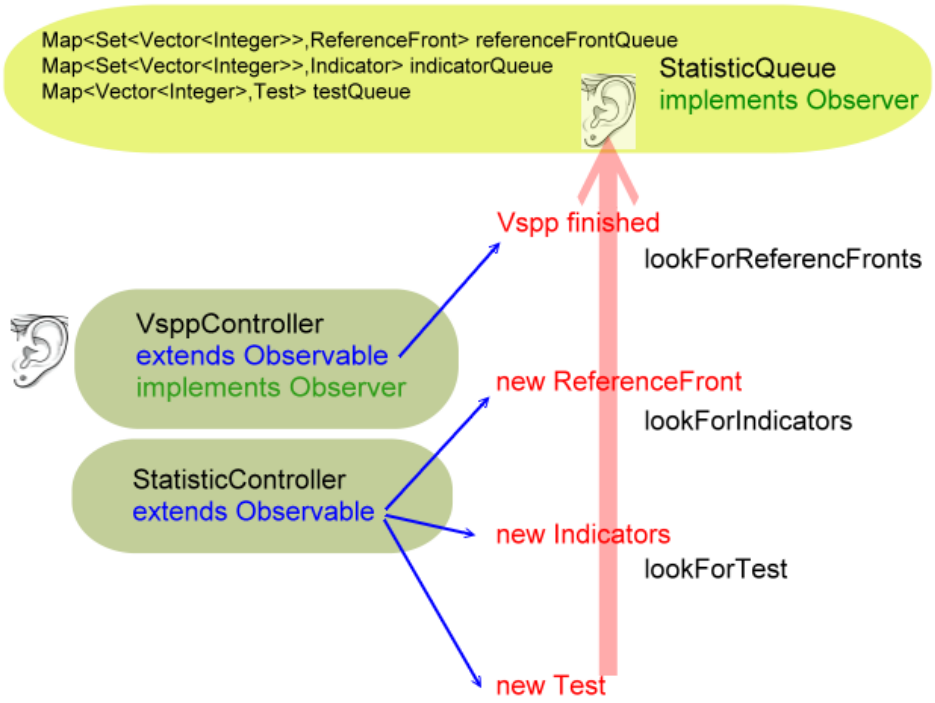

Abbildung 8: Die Klasse StatisticQueue horcht an den Klassen VsppController und StatisticController und wartet auf neue Ergebnisse. Falls eine Message (roter Text) eintrifft, werden die nötigen Aktionen durchgeführt. Der schwarze Text ist die nach dem Eintreffen der Message ausgeführte Methode.

ganze referenceFrontQueue sequentiell nach einem solchen Schlüssel durchsucht. Falls in einem Set von Indexvektoren(`aKeySet`) der Vektor `v` enthalten ist, wird `aKeySet` genauer überprüft. Wenn alle nötigen Vspp Daten in der vsppList liegen, wird die Berechnung der Referenzfront neu gestartet. Danach wird aKeySet und das zugehörige ReferenceFront Objekt aus der Warteliste entfernt.

```
for(aKeySet in refFrontQueue.keySet()){
 if(aKeySet.contains(v)){
        if(vsppList.keySet().containsAll(aKeySet))
   continue;
        referenceFrontQueue.get(aKeySet).restart();
        referenceFrontQueue.remove(aKeySet);
        }
}
```

Wenn das Datenmodell die Referenzfront berechnet hat, wird es alle Observer darüber benachrichtigen. So wird die Klasse StatisticQueue wiederum überprüfen, ob es nun möglich ist, neue Indikatorwerte zu berechnen.

Nach dem gleichen Prinzip funktionieren die indicatorQueue und die statisticQueue:

Die Methode lookForIndicator(Vector aVsppKey) Diese Methode funktioniert im Ablauf genau gleich wie `lookForReferenceFront(..)`. Nur das hier eine andere Datenstruktur durchsucht wird, diese enthält die Zwischengespeicherten Indikatoraufrufe in Form von der Collection:

```
HashMap<Set<Vector<Integer>>, Indicator>  indicatorQueue
```

Analog zur referenceFrontQueue verfügt die indicatorQueue über eine Abbildung von Sets von 3-dimensionalen Vspp-Schlüsseln auf eine Referenz einer Instanz der internen Indicator Klasse. Zusätzlich wird hier aber vor dem Aufruf der Methode restart() überprüft, ob die nötigen Referenzfrontdaten vorliegen.

Die Methode lookForTest(Vector aIndicatorKey) Analog zu den beiden vorherigen Methoden wird zur Zwischenspeicherung eines Tests eine vom Typ:

```
HashMap<Vector<Integer>,Test> testQueue
```

verwendet. In dieser Warteliste besteht die Abbildung von einem 6- dimensionalen Integervektor auf eine Referenz des internen Datentyps Test. Der

Schlüssel entspricht der gleichen Indexierungskonvention wie die Schlüssel in der statisticList in der Klasse StatisticController. Falls ein Indikator zu einem Test passt, wird überprüft ob alle Indikatordaten, die für einen Test notwendig sind, zur verfügung stehen. Wenn ja, wird die Methode `Test.restart()` aufgerufen und der Eintrag aus der statisticQueue gelöscht.

Datenstrukturen und Threads Ein anderes Problem ist, dass die Java Collections nicht threadsicher sind. Das heisst, die einzelnen Methoden sind nicht als synchronized deklariert. So kommt es, dass die Warteschlange überbeansprucht wird. Z.B. wenn ein Thread einen neuen Eintrag an einer Warteschlange anhängt, während ein anderer Thread mit Hilfe eines Iterators eine Collection durchsucht, gibt es einen Fehler. Auch müssen die Threads bevor sie auf die Datenmodell Collections zugreifen können, irgendwie kanalisiert werden, da sonst die Collectoins im Datenmodell überbeansprucht werden könnten.

Um das zweite Problem zu lösen, unterstützt die Collection Klasse die statische Methode `synchronizedMap`. Sie kann zum konstruieren einer threadsicheren Instanz einer beliebigen Collection Instanz verwendet werden. So sind alle die in der StatisticQueue vorhandenen Collections auf diese Art konstruiert worden. Damit ist aber das Problem erst für die Collections in der StatisticQueue gelöst. Die Threads müssen also noch kanalisiert werden, so dass die Klassen VsppController und StatisticController niemals von 2 Threads gleichzeitig manipuliert werden.

Die Lösung dazu kann so gestaltet werden, dass in StatisticQueue alle Methoden mit einem `synchronize` Schlüsselwort versehen werden. So wird die Methode für alle anderen Threads gesperrt, solange sie besetzt ist. Also wird der Aufruf der Methode restart() immer nur von einem Thread möglich sein. Natürlich gibt es für jede interne Klasse(ReferenceFront, Indicator und Test) eine Methode run(). Also ist es durchaus möglich, dass der StatisticController immer noch parallel verarbeitet wird. Dies ist nicht schlimm, denn die StatisticController Klasse beansprucht während ihren Berechnungen immer nur die zugehörige Collection(z.B. wird die Collection statisticList während einer Indikatorberechnung nicht verwendet). Diese Art mit dem Problem umzugehen ist zwar einfach aber gefährlich. Gefährlich weil, bei einer Erweiterung des Datenmodells genau darauf geachtet werden muss, in welcher Methode welche Collection manipuliert wird. Eine sicherere Lösung wäre es, die Collections aus der StatistcController Klasse auch threadsicher zu konstruieren. Dies würde sich aber wiederum auf alle anderen Klassen im Programm auswirken(weil theoretisch Messages parallel versendet werden könnten), so müssten diese ihrerseits auch wieder für Threadsicherheit garantieren. Aus diesem Grund habe ich die einfachere Lösung zur Behebung des Problems vorgezogen.

4 Fallstudie

4.1 Einleitung

Das Ziel der Studie ist zu zeigen, dass das Tool einen automatisierten Ablauf vom Start der VSP bis zur Visualisierung der Daten unterstützt und dass das Tool auch unter realen Voraussetzungen arbeitet. Dazu haben wir den Variator ZDT3 mit 30 Läufen und 1000 Generationen mit dem Selektorset {ibea, spea2, nsga2} gestartet. Für die Auswertungen wählten wir alle Indikatormodule: additiver epsilon, R und den Hypervolumen Indikator.

4.2 Resultate

Das Tool löst nun verschiedene Probleme. Einerseits können mehrere Monitorläufe zu verschiedenen VSP auf einmal per Knopfdruck gestartet werden. Weiter können die Indikator- und Testmodule bequem alle miteinander kombiniert und gestartet werden. Damit ist das Problem des umständlichen Starten von vielen Monitorläufen und das Starten der einzelnen Berechnungsschritten in allen Kombinationen gelöst. Auch mussten wir uns überhaupt keine Gedanken machen darüber, wie und wo die Daten abgelegt wurden. Das Tool hat die Daten für uns verwaltet und in sinnvoller Art auf die Disk abgelegt.

In Abbildung 9 sind die vom Tool und einem Matlabmodul erstellten Boxplots abgebildet. Man sieht, dass spea2 für den R Indikator signifikant besser arbeitet als die anderen beiden Selektoren. Für den hypervolumen Indikator stellt man nur einen signifikanten Unterschied zwischen spea2 und nsga2 fest. Um genauere Informationen zu erhalten, müsste man einen statistischen Test zu ziehen. Aus der letzten Graphik geht hervor, dass spea2 und nsga2 die signifikant schlechteren Resultate liefert als ibea. Dies ist insofern nicht erstaunlich, als dass ibea versucht, die Resultate für den epsiolon Indikator zu optimieren.

5 Zusammenfassung und Ausblick

Ein Vergleich von evolutionären Algorithmen besteht aus mehreren einzelnen Teilschritten oder Teilberechnungen, welche alle durch Parameter konfiguriert werden können. Von einem solchen Vergleich entstehen viele Daten, die man möglichst sinnvoll ablegen möchte. Ein systematisches Testen eines solchen Algorithmus ist aus diesen Gründen aufwändig. Trotz der vielen Möglichkeiten die es gibt um zwei evolutionäre Algorithmen zu vergleichen, gibt es ein gewisses Standardprozedere, dass uns erlaubt, einen solchen Vergleich zu automatisieren. Das ControlTool ist eine GUI gesteuerte Applika-

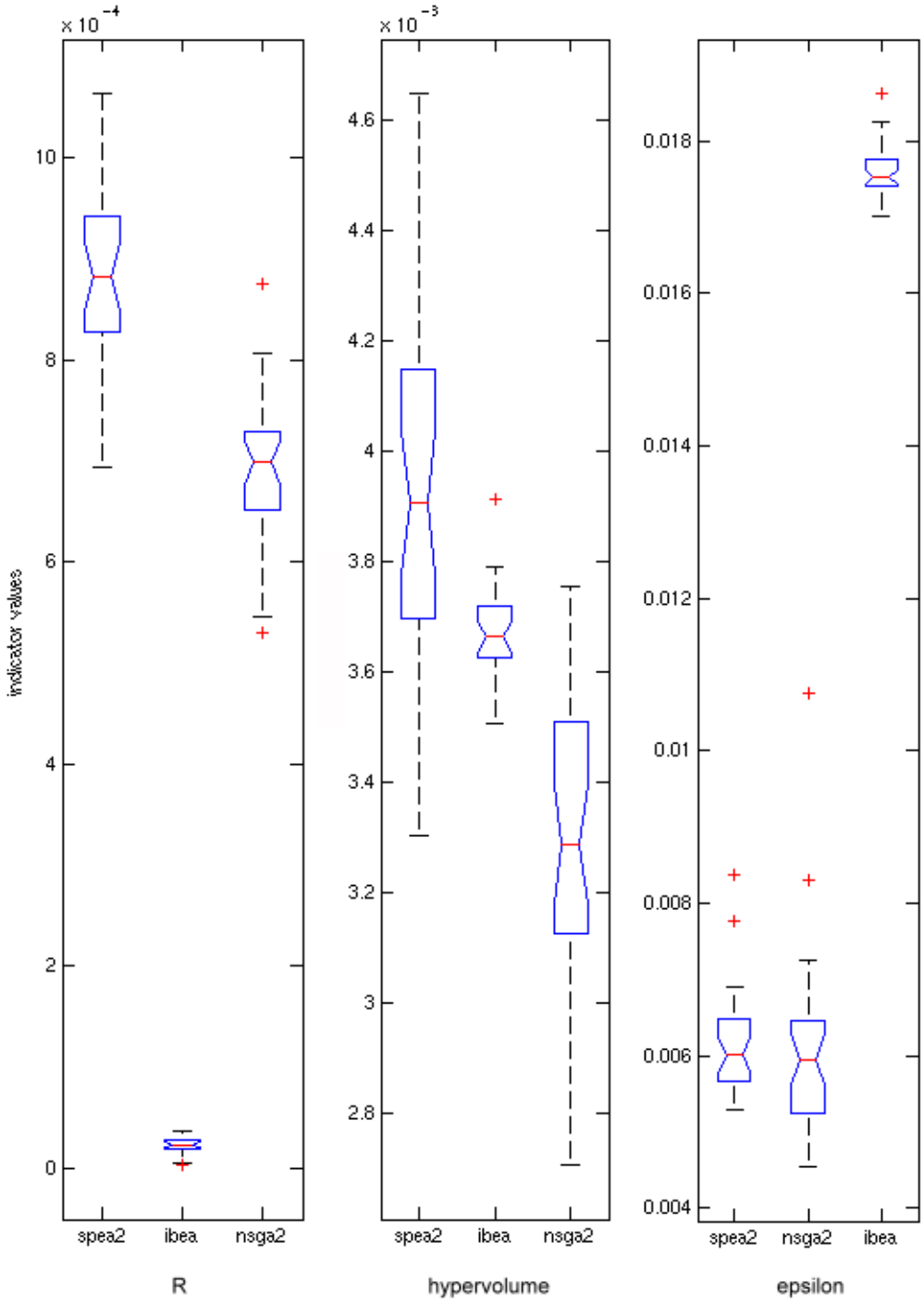

Abbildung 9: Boxplots: Variator ZDT3, Selektoren: ibea, spea2, nsga2; 30 Läufe zu je 1000 Generationen

tion, welche einem Benutzer das Vergleichen von EAs vereinfacht.

Das Tool steuert verschiedene vorhandene Module an, welche für die einzelnen Berechnungsschritte zuständig sind und koordiniert sie. Das Tool erlaubt sowohl einzelne Berechnungsschritte als auch einen ganz automatisierten Ablauf eines Vergleiches bzw. eines statistischen Tests. Z.B. können alle Variator-Selektorpaare ohne statistische Auswertungen gestartet werden. Es können aber auch einzelne Teilschritte wie das Berechnen einer Referenzfront von vorhandenen Daten bequem und übersichtlich von einem GUI aus erstellt werden.

Die berechneten Daten werden vom Tool in einer übersichtlichen Verzeichnisstruktur abgelegt, sodass sie nachvollzogen und rekonstruiert werden können. Das Tool stellt zusammenhängende Daten in einem GUI dar und ermöglicht dem Benutzer eine gute Übersicht über vorhandene Daten. Diese können nach den vollzogenen Auswertungen durch das GUI einfach an ein Plotmodul geschickt werden.

Es gibt viele Möglichkeiten, welche den Vergleich der Evolutionären Algorithmen noch einfacher gestalten würden. Häufig stellt sich z.B. das Problem, dass ein Benutzer gerne mehr Läufe an den vorhandenen anschliessen möchte, um seinen Daten sicher zu sein. Dazu könnten bestehende Parameterkonfigurationen in einen neuen GlobalIndex geladen und das Tool dazu aufgefordert werden, in dieser Konfiguration weitere Läufe zu generieren. Diese Läufe könnten dann z.B. an schon vorhandene angeknüpft werden.

Ein anderes Beispiel wäre ein interaktiver Plotter. So wäre es mit den vorhandenen Daten(so wie sie im nach der Verzeichnisstruktur abgelegt sind) möglich, wenn man ein geplottetes Individuum anklickt, dass das GUI automatisch die zugehörigen Parameter anzeigen würde.

Es wäre z.B. auch schön, einen Plotter für 3-dimensionale Fronten zu bauen. Dies ist, so modular wie das Tool aufgebaut ist, nicht sehr utopisch, zumal ganz einfach andere Programme als Module eingebunden werden könnten, wie zum Beispiel Matlab oder Mathematica.

Ein weiterer Vorschlag wäre, viel mehr Überprüfungen von Benutzereingaben auf der GUI Seite einzubauen. Alle diese Vorschläge sind dank einer sehr flexiblen und modularen Struktur des Tools sicherlich in die Realität umsetzbar.

A Die Voraussetzungen des ControlTool an die Verzeichnishierarchie und an die Module

Es folgt eine vollständige Spezifikation welche die Voraussetzungen des ControlTool an die gegebenen Daten stellt, damit das Tool einerseits funktioniert und andererseits vorhandene Daten korrekt einlesen kann. Zur Definition der regulären Ausdrücke wird hier eine kleine Auswahl an Platzhaltern kurz erläutert:

- `^`:Anfang des Ausdruckes
- `$`: Ende des Ausdruckes
- `var, sel, ind, test`: Stehen als Platzhalter für Variatorn, Selektoren, Indikatoren bzw. statistische Tests
- `+`: der Ausdruck vor dem + wird mindestens 1mal wiederholt
- `_`: ein Punkt
- `[0-9]+`: eine natürlich Zahl

Das ControlTool benötigt für eine korrekte Funktionsweise:

- Kein Modul darf einen Unterstrich im Namen tragen; Unterstriche werden zur Separation von Modulnamen benutzt.
- In jedem Variatorverzeichnis `ControlTool/variators/var/` wird jene Datei als Variator betrachtet, welche den gleichen Namen trägt wie das Vaterverzeichnis und sich am selben Ort eine dazu korrespondierende Parameterdatei aufhält. Dabei werden die beiden Suffixe `.exe` und `.bat` für die ausführbare Datei ignoriert. Die Parameterdatei heisst gleich wie der Variator gefolgt von dem Suffix `_param.txt`. Zudem muss in allen Variatorverzeichnissen die Datei `PISA_cfg` befinden. Sie beschreibt die Defaultwerte für Parameter in der PISA Schnittstelle. Es muss mindestens ein Variator in diesem Verzeichnis sein, welche diese Auflage erfüllt.
- In genau gleicher Manier findet die Suche nach den Selektoren statt. Der einzige Unterschied zur Variatorsuche ist, dass das `PISA_cfg` file nicht im Selektorverzeichnis sein muss.
- Die ausführbare Datei *monitor* und das zugehörige Parameterfile `monitor_param.txt` muss sich im Verzeichnis `ControlTool/monitor/` befinden.

- Damit Dateien als Referenzfronten gelten, müssen sie die Dateiendung `.rf` aufweisen und sich im Verzeichnis `referenceSets/` befinden
- Indikatoren und statistische Tests liegen in `indicator/` bzw. `statistics/` und werden als solche angesehen, falls eine zugehörige Parameterdatei auch im jeweiligen Verzeichnis liegt.

Die im Folgenden angesprochenen Verzeichnisse liegen immer im Vaterverzeichnis `ControlTool/Projects/"project name"/"globalIndex"/`.

- Ausgewertete Daten mit Hilfe des Monitors, muss den Dateinamen `var_sel.x` haben, damit diese als generierte Läufe angesehen und ins Programm eingebunden werden. Ausserdem befinden sich solche Daten im Verzeichnis `runs/` im jeweiligen global Index.
- Ausgewertete Daten mit Hilfe eines Indikatoren befinden sich im Verzeichnis, welches mit dem regulären Ausdruck

  ```
  indicators/ind_[0-9]+/$
  ```

  beschrieben wird, liegen. Die einzelnen Dateinamen werden mit dem regulären Ausdruck `^var_sel_ind\.[0-9]+$` beschrieben. Wobei die letzte Zahl die verwendete Generation beschreibt. Der Indikatorname im Vaterverzeichnis der Datei muss derselbe sein, wie der Indikatorname im Dateinamen.
- Ausgewertete Daten mit Hilfe eines statistischen Tests liegen im Verzeichnis welches durch den regulären Ausdruck `^stat_[0-9]+/$` beschrieben wird liegen. Der Dateiname wird durch

  ```
  ^var_ind_[0-9]+_test_\.[0-9]+$
  ```

  beschrieben. Dabei muss der Test im Dateinamen derselbe sein, wie jener im Vaterverzeichnis.

A.1 Vom ControlTool generierte Daten

Natürlich folgt das ControlTool, wenn Daten generiert werden, den oben spezifizierten Auflagen. Aber es werden noch viele andere Daten generiert. Damit Konflikte mit anderen Modulen vermieden werden können, werden sämtliche Daten, welche erzeugt werden, aufgelistet. Dies dient nicht nur zur Konfliktvermeidung, andere Module können sich diese Daten natürlich auch zu Nutze machen. Ausserdem ist es für den Benutzer wichtig, dass sämtliche Resultate nachvollziehbar sind. Also sollen auch die Zwischenresultate nach einem geeigneten System eingebunden und genau spezifiziert werden.

Die im Folgenden angesprochenen Verzeichnisse liegen immer im Vaterverzeichnis `ControlTool/Projects/"project name"/"globalIndex"/`.

Bei einem gestarteten Monitorlauf, werden im Verzeichnis `/runs/` folgende Dateien generiert:

- `commandOf_var_sel`: Eine Datei welche den genauen Startbefehl des Monitorprozesses enthält.
- `var_output.txt`: Hier werden die Lösungsdaten bezüglich des Variators der letzten berechneten Generation dargestellt. In Zukunft sollen von allen Generationen die Lösungsdaten ausgegeben werden. Dies kann z.B. in Form eines angefügten Punktes und der zugehörigen Generation geschehen. Im übrigen kann der Name dieser Datei dem Variator als Parameter mitgegeben werden.
- `var_sel.x`: Hier werden die Individuen aller vom Monitor lancierten Läufe einer Generation `x`dargestellt. Wieviele von diesen Dateien entstehen, ist von der Parameterwahl des Monitors abhängig.
- `var_param.txt`: Kopien der verwendeten Variatorparameterdateien
- `sel_param.txt`: Kopien der verwendeten Selektorparameterdateien
- `monitor_param.txt`: Eine Kopie der verwendeten Monitorparameterdatei
- `var_sel.txt`: Eine vom Monitor generierte Log-Datei. Sie enthält Informationen über die gemeinsam verwendeten Parametern von Variator und Selektor, sowie die Monitorparameter.
- `PISA_var/var_sel_cfg,arg,ini,sel,sta,var`: Dieses Unterverzeichnis und ihre darin enthaltenen Dateien dienen zur Kommunikation zwischen dem Monitor und dem Variator.
- `PISA_sel/var_sel_cfg,arg,ini,sel,sta,var`: Dieses Unterverzeichnis und ihre darin enthaltenen Dateien dienen zur Kommunikation zwischen dem Monitor und dem Selektor.

Wenn mit dem Tool eine Referenzfront generiert wird, werden im Verzeichnis `/referenceFronts/` folgende Daten generiert. `refname` dient als Platzhalter für den vom Benutzer eingegebenen Namen. Man kann im den Namen auch generieren lassen. Dann wird der Name `var_ref.x.rf` verwendet.

- `commandOf_refname`: Diese Datei enthält den genauen Prozessaufruf des filter-Modul.

- `refname.rf`: In dieser Datei ist die Referenzfront enthalten.

Das Verzeichnis `refname/` enthält die Zwischenresultate zur Berechnung einer Referenzfront. In ihm befinden sich:

- `commandOf_var.x`: Eine Aufliestung der Dateien, welche aneinander gefügt wurden.
- `var.x`: Die Aneinanderreihung von `var_sel.x` Dateien vom `../runs/` Verzeichnis.
- `commandOf_var_bound.x`: Der genaue Befehl, welcher verwendet wurde, um das bound-modul zu starten.
- `var_bound.x`: Das vom bound-Modul generierte Resultat
- `commandOf_var_normalize.x`: Der genaue Befehl, welcher verwendet wurde, um das normalize-modul zu starten.
- `var_normalize.x`: Das Zwischenresultat des normalize-Moduls

Um Indikatorwerte zu berechnen, legt das Tool dieselben 6 letzten Dateien in das Verzeichnis `/indicators/ind_indIndex`. Dazu werden die 2 folgenden Dateien erzeugt:

- `commandOf_var_sel_ind.x`: Diese Datei enthält den genauen Programmaufruf der verwendet wurde um den Indikatorprozess zu starten.
- `var_sel_ind.x`: In dieser Datei befinden sich die Indikatorauswertungen bezüglich dem zugehörigen Variator und Selektor in der Generation x.

Zum Schluss werden beim Aufruf eines statistischen Tests 3 Dateien erzeugt:

- `commandOf_var_ind_indIndex_test.x`: Der genaue Aufruf des statistischen Tests wird in dieser Datei dokumentiert.
- `var_ind_indIndex_test_log`: Hier drin befinden sich alle Selektoren, welche in den Test mit einbezogen wurden.
- `var_ind_indIndex_test_x`: In dieser Datei steht das Resultat des jeweiligen Tests.

Falls das Tool aufgefordert wird, alle die Schritte automatisch vorzunehmen, werden die 4 Schritte aneinandergefügt. Bei der Namensverteilung ändert sich dabei nichts.

Literatur

[1] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. Pisa – a platform and programming language independent interface for search algorithms. In C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, volume 2632/2003 of *LNCS*, pages 494 – 508. Springer-Verlag Heidelberg, 2003.

[2] L. Thiele. Using the monitor in pisa. http://www.tik.ee.ethz.ch/pisa/monitor/monitor_documentation.pdf.

Manual

This manual is gives you a very short introduction about the goal of the ControlTool. Then the manual shows how to setup the application. In another section the manual describes how to generate new data and how to select specific data to visualize.

1 Introduction

The ControlTool is a tool for comparison of multi objective optimization algorithms. The tool is a framework to simply and comfortably control several modules to start Algorithms and to fulfil all needed steps for a statistical comparison. The tool saves intermediate and finished result data in a specified directory hierarchy which contains all the information. This guarantees you that all results are reproducible by starting the runs or analysis tools by hand. All these results are saved in a human readable form on your disk, so you can insert, delete and manipulate all results by hand. The specified directory hierarchy gives you a good overview over the data for this. The goal is of course, that the user doesn't need to care about how data are saved on the hard disk, but the user has the possibility to manipulate them with a standard text editor.

The tool is based on PISA[1] which separates an algorithm in two parts. A problem dependent part, called the *variator*, and a problem independent part, the *selector*. The tool provides to launch several parallel running variator-selector pairs(VSP) with a specific parameter configuration (VSPP).

The tool provides an automatic way to perform the following steps:

1. launch of several parallel running algorithms
2. selection and launch of indicator- and statistical test modules.
3. selection of results and the launch of a module to visualize them, i.e. plots.

1.1 Requirements

The tool requires a java virtual machine, at least version 1.5.0[1].

[1] http://www.java.com/en/download/download_the_latest.jsp

2 Set up the tool

All required modules can be downloaded on the PISA webpage[2]. In the directory where you whish to start the tool, the following folders and modules need to exist in order to set up the program. The paths in the following items are relative to the path where the application is started.

- The monitor module. The executable `monitor` (`.exe`) file and the parameter file `monitor_param.txt` must be in the directory `monitor/`.
- A `variators/` folder containing subfolders with the name of your variator you like to use. Therein an executable variator module with corresponding parameter and `PISA_cfg` file must exist.
- A `selectors/` folder containing all the selector modules. Each selector lies in a separate subfolder (with the same name as the selector) and the corresponding parameter file.
- The three modules *bound*, *normalize* and *filter* must lie in a directory called `tools/`. All the indicators and test modules as well as the corresponding parameter exist in a folder `indicators/` and `statistics/`, respectively.

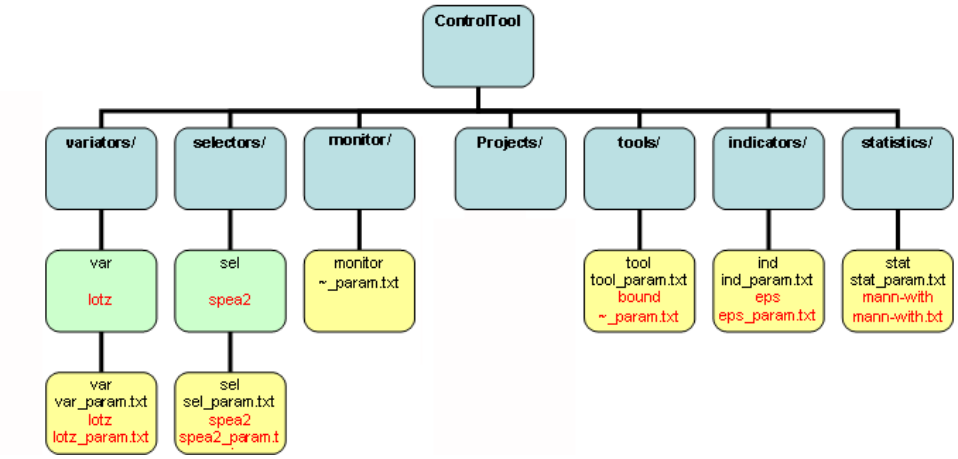

Figure 1: This figure illustrates all necessary directories(in blue) to run the program. To add a variator or a selector you have to add a green directory. Red text denotes an example.

With this setup you can now start the ControlTool with the command

[2]http://www.tik.ee.ethz.ch/pisa

```
java -jar controlToolPackageName.jar
```

or simply by an icon click on the jar file (if the command is not found, make sure that the `java/bin` directory is in the classpath).

3 The ControlTool

3.1 Application start

On every program start, the tool checks for new modules and browses all data. This may need a certain time, depending on the size of the data in the project. First, the project has to be selected. The project is a workspace to clearly separate research projects. For a faster application start, it is recommended to often change the workspace.

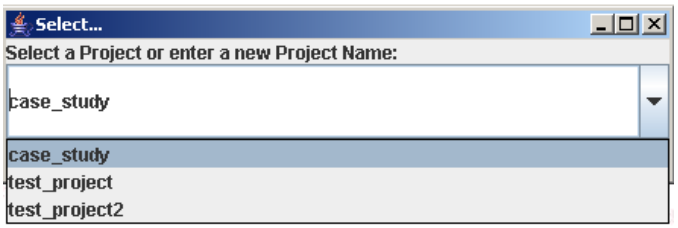

Figure 2: The select a project frame

In the "Select a project" frame you can choose an available project or you can enter a new name. So a new project is created.

3.2 Starting Runs and Analysis Tools

To generate new data, use the menu item ControlTool - New... . A new window is opened. This will leads you throw four configuration steps. You must not open two such windows at the same time at the current version.

Step 1 - variators and selectors Select these variators you want to test. For better overview you should choose not too many or even only one.

Select all selectors you want. The tool will build the superset; this means it will run all possible combinations of variators and selectors. Note that the selectors you choose here are tested against each other for a specific variator.

All tests performed afterwards are always applied to all VSPPs (the superset) you select now.

Step 2 - parameter configuration For all chosen modules in step 1, you can now configure their parameter setting. With the button "set back to default" you can undo your entries. With a click on the button "save as default" you overwrite the default values. This operation is not undoable.

In the monitor parameter tab you can change for example the number of runs to perform or the number of generations to calculate. These parameters are the same for all VSPs chosen in the first step.

The configuration information of the variator consists of two files: First, the parameter setting for the variator, i.e. the `variator_param.txt` file. Second, the parameters for the `PISA_cfg` file. For example can be determined the size of the start population. For each VSP, this file is related to the variator. So for a variator, the same `PISA_cfg` configuration will be used for every selector partner.

At the moment at which you click on the button `next`, the program copies the parameter files to the right directory. If you abort now, the directory and the parameterfiles are not deleted.

Step 3 - Indicators and Tests Here you can select indicators you like to use to assess the outcome of the multi-objective evolutionary algorithms. You can also choose none, if you like to do the analysis of the algorithms afterwards.

For all VSPs there will be an indicator file generated for all the here selected indicators. Note that for each indicator the parameter configuration rests the same. If you like to use different configurations, you can do this after the algorithms have finished.

Select the tests you like to perform on the data. Each test is executed for each selected indicator.

Step 4 - Parameter configuration for Indicators and Tests First you have to select a generation on which the analysis is performed. There is a default value which represents the latest generation. It's up to the user to select a generation that was written on the disk by the monitor.

You can now configure indicator and test parameters for all chosen modules. For the indicators you can also use a default configuration that, for example, reads the dimension parameter from the data file. If you like to use your own parameters, you must disable the check box under the indicator parameter panel. Note that, if this checkbox is selected, the user defined parameters are ignored.

Click finish! Now all runs are started and the analysis is put into a waiting list. If necessary results are available, the program will begin to

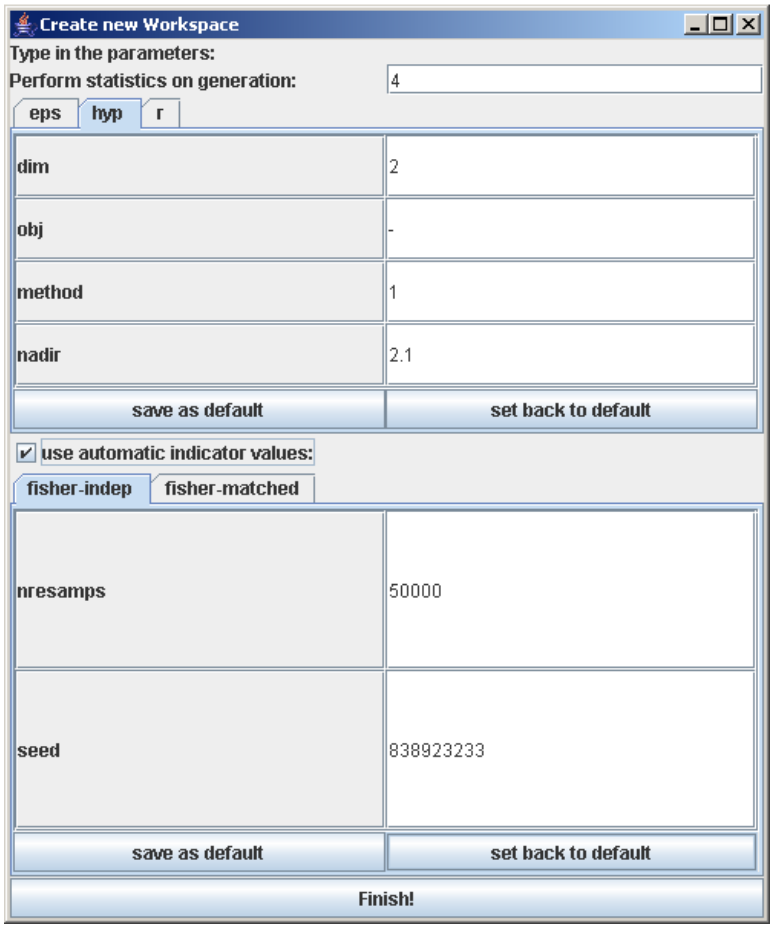

Figure 3: A parameter panel

calculate and is blocked for a certain moment. If the parameter "debug" iUn the monitor parameter panel is equal to 1, you can watch the state of the running modules on the standard output stream.

During the generation of data, you may use the program parallel to the calculations. You can watch on data you generated before or even start new runs.

3.3 Using the Main Control Window

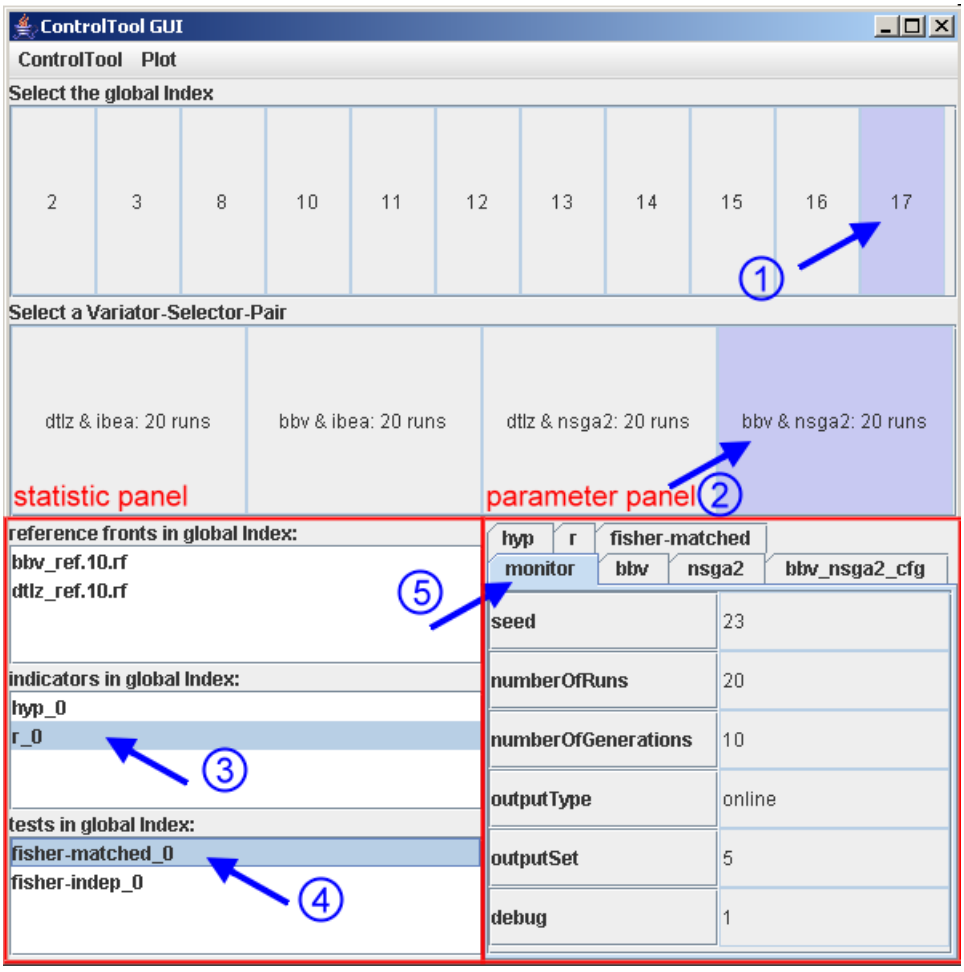

Figure 4: This figure shows the mainframe. The numbers are in order to select correctly the data. The numbers: 1.global index - 2.VSP - 3.Indicator - 4.Test - 5. Parameterpanel

Generally you handle this window from the top to the bottom.

First you select a global index, in fig.4 denoted by a 1. This specifies a subworkspace; this means for all the VSPs in the global index the same monitor parameters were used (you can also think about the global index as "together generated data"). Further, all the variators and selectors have the same parameter configuration.

Second you can select a VSPP, in fig .4 denoted by a 2. All parameters used for this VSP are now shown in the parameter panel. Also all available test data are displayed in the statistic panel.

If you have done some analysis, you can chose now a available indicator to show the parameters used. If the automatic parameters were selected for an indicator, there will be an empty tab in the parameter panel corresponding to this indicator.

If you have done some tests in this global index, they are displayed in the statistic panel. If you click on it, the corresponding parameters are displayed in a new tab in the parameter panel. To see the data in the test, i.e. p-values, you can make a right-click on the selected test, a little window will open that shows the corresponding p-values. Note that these results depend on the selected indicator before. If you have done new calculations, it's up to the user to know which indicator belongs to which statistical test.

3.4 New calculations

If you like to perform new indicator calculations or new tests, which means with another parameterset, you can choose the menu item `ControlTool`→ `new calculations...`. This opens a well known window. You can now choose a new indicator and test set. By a click on next you may configure the parameters. The program is blocked while calculating the tests.

3.5 Results and plotting

To visualize the results of your tests you may do 2 plots.

Front plot For two dimensional problems only, you can plot a front. For this you have to select a VSP and an available reference front. Now you can choose the menu item `plot`→`plot a front`. A module, which plots the VSP- and the reference front in the same graphic, is started.

Boxplot You can start a boxplot module for a specific variator and a specific indicator. First select a VSP to select the variator (the selector is ignored; all selectors in the current global index are plotted). Then select

an indicator you like to use for your boxplot. After that you can start the boxplot module with the menu item `plot→boxplot...`

References

[1] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. Pisa – a platform and programming language independent interface for search algorithms. In C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, volume 2632/2003 of *LNCS*, pages 494 – 508. Springer-Verlag Heidelberg, 2003.