Vergleichende Analyse von Genetischen Algorithmen und der Particle Swarm Optimization für den Evolutionären Strukturtest

Arbeit zur Erlangung des akademischen Grades

Diplom-Informatiker

an der Technischen Universität Berlin

in Kooperation mit

DaimlerChrysler Research and Technology

vorgelegt von

Andreas Windisch
Matrikelnummer: 217981

Betreuer: Prof. Dr.-Ing. Ina Schieferdecker
Institut für Telekommunikationssysteme
Technische Universität Berlin
Fakultät IV Elektrotechnik Informatik
Franklinstraße 28/29
10587 Berlin

Dr. rer. nat. Joachim Wegener
DaimlerChrysler Research and Technology
Bereich Software Engineering
Abteilung Methoden und Tools
Alt-Moabit 96a
10559 Berlin

Datum: Berlin, den 3. April 2007

Danksagung

An dieser Stelle möchte ich all denen danken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Arbeit beigetragen haben.

Besonders dankbar bin ich Prof. Ina Schieferdecker für die freundliche und unkomplizierte Betreuung der vorliegenden Arbeit. Dr. Joachim Wegener danke ich für die großartige und engagierte Betreuung meiner Diplomarbeit und meiner gesamten Zeit am Forschungsbereich der DaimlerChrysler AG. Seine hilfreichen Kommentare und Anmerkungen haben sehr zum Gelingen dieser Arbeit beigetragen. André Baresel und Dr. Hartmut Pohlheim danke ich für die hilfreichen Hinweise zur Benutzung des Evolutionären Testsystems.

Mein größter Dank gebührt Stefan Wappler, der mir in den letzten sechs Monaten mit Rat und Tat zur Seite stand. Ich danke ihm für die vielseitige Unterstützung, sein außerordentliches Engagement und die zahlreichen wissenschaftlichen Ratschläge, welche stets zur Verbesserung der Arbeit beigetragen haben.

Natürlich möchte ich mich noch besonders bei meinen Eltern bedanken, die mir das sorgenfreie Studium überhaupt erst ermöglicht haben. Für Eure einzigartige Unterstützung während dieser Zeit, die sicherlich nicht selbstverständlich ist, vielen Dank!

Zum Abschluss möchte ich mich in besonderem Maße bei meiner Partnerin Isabelle Wieprecht für das ausdauernde Verständnis, die unzähligen aufmunternden Worte und die alltägliche Unterstützung während der Fertigstellung dieser Arbeit bedanken. Ihr widme ich diese Arbeit.

Zusammenfassung

Im Bereich der Softwarequalitätssicherung spielt der Modultest eine bedeutende Rolle, da er zur frühzeitigen Fehlererkennung entscheidend beiträgt. Die kritische Aktivität beim Modultest ist die Auswahl der Testdaten. In der Praxis kommen vielfach strukturorientierte Überdeckungskriterien zum Einsatz, um diese Aktivität zu steuern und die Relevanz gegebener Testdaten zu beurteilen. Da der Prozess der Testdatenauswahl sehr zeitaufwändig und fehleranfällig ist, wenn er manuell durchgeführt wird, wurden in den letzten Jahren verschiedene Automatisierungsansätze entwickelt. Einer dieser Ansätze ist der Evolutionäre Strukturtest.

Der Evolutionäre Strukturtest ist ein Verfahren zur Automatisierung des Modultests auf Basis strukturorientierter Überdeckungskriterien. Dabei wird die Aufgabe der Testdatenerzeugung als Optimierungsproblem aufgefasst. Der Test wird in die für das gewählte Strukturtestkriterium zu erreichenden Testziele unterteilt. Jedes Testziel wird als separates Optimierungsproblem aufgefasst: Finde ein Testdatum, durch das ein bestimmtes Strukturelement des zu testenden Quellcodes ausgeführt wird. Zur Lösung dieser Aufgabe kommt ein evolutionärer Algorithmus zum Einsatz. Dabei handelt es sich um ein metaheuristisches Optimierungsverfahren, das sich besonders zur Lösung multimodaler, nichtlinearer und diskontinuierlicher Optimierungsprobleme eignet. Im Allgemeinen haben sich evolutionäre Algorithmen, speziell Genetische Algorithmen (GAs), im industriellen Einsatz vielfach bewährt.

Neben Genetischen Algorithmen gewinnt die Particle Swarm Optimization (PSO) als Suchverfahren immer mehr an Bedeutung. Verschiedene Untersuchungen haben gezeigt, dass die PSO in vielen Fällen den Genetischen Algorithmen überlegen ist: Bei gleicher Effektivität verursacht sie einen geringeren Berechnungsaufwand und durch ihre geringe Komplexität ist sie leichter anwendbar. Aufgrund dieser Eigenschaften stellt die PSO eine viel versprechende Alternative zur Verwendung von Genetischen Algorithmen beim Evolutionären Strukturtest dar.

Im Rahmen dieser Arbeit ist eine vergleichende Untersuchung Genetischer Algorithmen und der PSO für den Evolutionären Strukturtest durchgeführt worden. Das bei DaimlerChrysler entwickelte Testsystem wurde für die Unterstützung der PSO erweitert und besonders geeignete Varianten der PSO implementiert. Anhand selbst erstellter und

ausgewählter industrieller Fallstudien wurde ein empirischer Vergleich hinsichtlich Effektivität und Effizienz beider Optimierungsverfahren für den Strukturtest aufgestellt. Dieser bestätigte die Überlegenheit der PSO gegenüber den Genetischen Algorithmen für die Anwendung im Kontext des Evolutionären Strukturtests. So konnte die PSO im Schnitt eine Effektivitätssteigerung um 8% und eine Effizienzsteigerung um 29% erreichen.

Inhaltsverzeichnis

Kapitel 1

Einführung

Im Bereich der Softwarequalitätssicherung spielt der Modultest eine bedeutende Rolle, da er zur frühzeitigen Fehlererkennung entscheidend beiträgt. Dies ist sehr wichtig, da vor allem die Behebung von Fehlern, deren Ursprung einer frühen Phase des Entwicklungsprozesses entspringen, häufig sowohl mit kostenspieligen und umfangreichen Änderungen von großen Programmteilen als auch mit Folgefehlern verbunden ist. Die kritische Aktivität beim Modultest ist die Auswahl der Testdaten. Für einen Korrektheitsbeweis muss ein vollständiger Test durchgeführt werden, d.h. das Testobjekt muss mit allen möglichen Eingabewerten und all ihren möglichen Kombinationen ausgeführt werden. Da dies jedoch in der Praxis nicht durchführbar ist, muss ein Stichprobentest durchgeführt werden. Für das jeweilige Testobjekt werden dabei fehlerrelevante Eingabedaten gesucht. Zu diesem Zweck kommen vielfach strukturorientierte Überdeckungskriterien zum Einsatz, um diese Aktivität zu steuern und die Relevanz gegebener Testdaten zu beurteilen. Beispiele dafür sind die Anweisungs-, der Zweig- oder der Bedingungsüberdeckung. Bei der Anweisungsüberdeckung wird eine Ausführung aller Anweisungen des Testobjektes gefordert. Für eine vollständige Überdeckung gemäß des Zweigüberdeckungstests müssen alle Zweige des korrespondierenden Kontrollflussgraphen bei der Ausführung durchlaufen werden. Der Bedingungsüberdeckungstest stellt Anforderungen an die Ausführung von hierarchisch aufgebauten Bedingungen.

Da der Prozess der Testdatenauswahl sehr zeitaufwändig und fehleranfällig ist wenn er manuell durchgeführt wird, wurden in den letzten Jahren verschiedene Automatisierungsansätze entwickelt. Einer dieser Ansätze ist der Evolutionäre Strukturtest. Der Evolutionäre Strukturtest ist ein Verfahren zur Automatisierung des Modultests auf Basis strukturorientierter Überdeckungskriterien. Dabei wird die Aufgabe der Testdatenerzeugung als Optimierungsproblem aufgefasst. Der Test wird in die für das gewählte Strukturtestkriterium zu erreichenden Teilziele unterteilt. Jedes Teilziel wird als separates Optimierungsproblem aufgefasst, d.h es wird ein Testdatum gesucht, durch das ein bestimmtes Strukturelement des zu testenden Quellcodes überdeckt wird. Zur Lösung dieser Aufgabe kommt ein evolutionärer Algorithmus zum Einsatz. Dabei handelt es sich um ein metaheuristisches Optimierungsverfahren, das sich besonders zur Lösung

multimodaler, nichtlinearer und diskontinuierlicher Optimierungsprobleme eignet. Diese nutzen die Prinzipien der natürlichen Evolution, um von Generation zu Generation Lösungen mit einer bezüglich des Optimierungsproblems steigenden Güte zu generieren. Dabei werden genetische Operatoren, wie beispielsweise die Selektion, die Rekombination oder die Mutation verwendet. Im Allgemeinen haben sich evolutionäre Algorithmen, speziell Genetische Algorithmen, im industriellen Einsatz vielfach bewährt.

Neben den Genetischen Algorithmen gewinnt die Particle Swarm Optimization (PSO) als Suchverfahren immer mehr an Bedeutung. Dabei handelt es sich ebenfalls um ein metaheuristisches Optimierungsverfahren, welches das kooperierende Verhalten von Vogelschwärmen auf der Suche nach reichhaltigen Futterplätzen nachahmt. Ein Partikel entspricht dabei einem dieser Vögel. Seine Flugrichtung über die zu optimierende Funktionslandschaft wird von seiner Erfahrung und der Erfahrung der anderen Vögel des Schwarms beeinflusst. Verschiedene Untersuchungen, wie beispielsweise (Hod02; CW04; HCW05), haben gezeigt, dass die PSO den Genetischen Algorithmen in vielen Fällen überlegen ist: Bei gleicher Effektivität verursacht sie einen weitaus geringeren Berechnungsaufwand und durch ihre geringe Komplexität ist sie leichter anwendbar. Aufgrund dieser Eigenschaften stellt die PSO eine viel versprechende Alternative zur Verwendung von evolutionären Algorithmen beim Evolutionären Strukturtest dar.

Ziel dieser Arbeit ist die vergleichende Untersuchung von Genetischen Algorithmen und der PSO für den Evolutionären Strukturtest. Das bei DaimlerChrysler entwickelte Testsystem ist für die Unterstützung der PSO zu erweitern und besonders geeignete Varianten der PSO zu implementieren. Dabei soll die Eignung dieser Varianten mithilfe von in der Literatur häufig zu Benchmark-Zwecken verwendeten Testfunktionen bestimmt werden. Die sich dabei als optimal herausgestellte Variante soll dann als Repräsentant der PSO für die Vergleichstests verwendet werden. Es soll anhand ausgewählter Fallstudien ein empirischer Vergleich hinsichtlich Effektivität und Effizienz beider Optimierungsverfahren für den Strukturtest aufgestellt werden. Die Fallstudien sollen einerseits mehrere Testobjekte enthalten, die für diesen Zweck erstellt wurden und spezielle Anforderungen an die daraus resultierende und zu optimierende Zielfunktion stellen. Da deren Ausführung darüber hinaus keinen Nutzen bringt, werden sie in dieser Arbeit als *künstliche Testobjekte* bezeichnet. Andererseits sollen aber auch komplexe industrielle Testobjekte für den Vergleich verwendet werden, um einen Überblick über die Einsatzfähigkeit für reale Probleme zu untersuchen.

Die Arbeit ist wie folgt aufgebaut: In Kapitel 2 werden unterschiedliche Bereiche Evolutionärer Algorithmen und ihre historische Entwicklung vorgestellt. Insbesondere werden sowohl die Genetischen Algorithmen als auch die Particle Swarm Optimization vorgestellt und genauer erläutert. Bei den GAs wird auf die verwendeten Operatoren, wie z.B. die Selektion, die Rekombination oder die Mutation eingegangen. Nach einer allgemeinen Vorstellung der PSO wird detailliert auf einige ausgewählte Varianten eingegangen. Abschließend folgt ein allgemeiner Vergleich beider Verfahren.

Kapitel 3 befasst sich mit der strukturorientierten Testfallermittlung. Zunächst werden Grundbegriffe, wie der Kontrollflussgraph oder die Klassifikationsmerkmale einzelner kontrollflussorientierter Metriken vorgestellt, worauf die Beschreibung des Evolutionären Strukturtests mitsamt der benötigten Metriken zur Konstruktion einer Zielfunktion folgt.

Kapitel 4 und 5 bilden den Kern dieser Arbeit. Kapitel 4 behandelt zunächst die Implementierung des Evolutionären Testsystems. Dabei wird auf das bereits bestehende System eingegangen und die im Rahmen dieser Arbeit erstellten Erweiterungen zur Unterstützung der PSO beschrieben. Des Weiteren wird die durchgeführte Implementierungsverifizierung als Vergleich der implementierten PSO-Varianten präsentiert. Das fünfte Kapitel behandelt den durchgeführten Vergleich der GAs und der PSO anhand von selbst erstellten und industriellen Testobjekten. Die selbst erstellten (künstliche) Testobjekte sind derart aufgebaut, dass bestimmte Bedingungen an die daraus entstehende Zielfunktion erfüllt werden. Es werden die durchgeführten Experimente detailliert beschrieben und die erzielten Ergebnisse für die einzelnen Testobjekte vergleichend zusammengefasst.

Das sechste Kapitel bildet mit einer Zusammenfassung der erzielten Ergebnisse und einem Ausblick auf mögliche zukünftige Arbeiten zur weiteren Verbesserung des Evolutionären Strukturtests den Abschluss der Arbeit.

Kapitel 2
Evolutionäre Algorithmen

Der Begriff *Evolutionäre Algorithmen* ist ein Oberbegriff für alle Optimierungsverfahren, welche auf bestimmten Mechanismen der biologischen Evolution basieren. Sie zeichnen sich sowohl durch die Verwendung einer Population von potentiellen Lösungen und der damit verbundenen Parallelität der Suche, als auch durch die verwendeten probabilistischen Übergangsregeln aus. Des Weiteren sind Evolutionäre Algorithmen häufig von biologischen Paradigmen, wie beispielsweise bei Tieren beobachteten Verhaltensweisen oder der Genetik, inspiriert.

Im Vergleich zu traditionellen Such- und Optimierungsverfahren, bei denen die Suche inkrementell von einem Punkt im Suchraum zum anderen verläuft, starten Evolutionäre Algorithmen die Suche typischerweise mit einer Population von Individuen, welche dann ähnlich einer gerichteten Zufallssuche den Suchraum erkunden. Dabei werden keine zusätzlichen Informationen über die zu optimierende Funktion benötigt, wie z.B. Funktionsableitungen, wie sie beispielsweise bei Gradientenverfahren berechnet werden müssen. Evolutionäre Algorithmen bewerten die Güte, auch Fitness genannt, ihrer Individuen (Positionen im Suchraum) nach dem Wert der zu optimierenden Funktion. Die Fitness eines Individuums entscheidet, wie groß dessen Einfluss auf spätere Generationen ist. Dadurch wird die Suche auf Regionen im Suchraum gerichtet, von denen man sich eine höhere Wahrscheinlichkeit für gute Fitnesswerte erhofft.

Das Feld der Evolutionären Algorithmen lässt sich historisch bedingt in die folgenden vier sich unabhängig voneinander entwickelten Strömungen einteilen (nach Kennedy und Eberhart (KE01)):

- Genetische Algorithmen
- Evolutionäre Programmierung
- Evolutionsstrategien
- Genetische Programmierung

Genetische Algorithmen sind Suchtechniken, die auf Grundlagen der natürlichen Genetik und der Theorie der biologischen Evolution von Darwin basieren. Sie verwenden

Mechanismen wie beispielsweise die Selektion, die Rekombination und die Mutation. Deren Entwicklung und der Einsatz zur Optimierung künstlicher Systeme wurde in den frühen 60er Jahren erstmals von John H. Holland vorgestellt (Hol62; Hol75). Auf Genetische Algorithmen sowie die zugrunde liegenden Operatoren wird im Abschnitt 2.1 detailliert eingegangen.

Von der *Evolutionären Programmierung* wurde erstmals im Jahre 1966 von Fogel, Owens und Walsh (FOW66) berichtet. Das Ziel war die Erschaffung künstlicher Intelligenz durch Simulation der Evolution als ein Lernprozess. Grundlage dieser Arbeit waren einfache endliche Automaten[1], die zur Vorhersage von Zeichenketten verwendet wurden. Die Fähigkeit der Vorhersage wurde als eine Voraussetzung für die Anpassungsfähigkeit an sich ändernde Umgebungen und für intelligentes Verhalten angesehen. Jedes Individuum repräsentierte einen endlichen Automaten, welcher für eine gegebene Eingabesequenz eine spezifische Ausgabesequenz generiert. Für eine vorgegebene Sequenz von Symbolen wurde nun ein Individuum (ein Automat) gesucht, welcher möglichst dieselbe Sequenz generiert, um mit dessen Hilfe die nachfolgenden Symbole vorhersagen zu können. Neue Individuen wurden durch Mutationen in Form von Veränderung der Zustandsübergänge oder der Anfangszustände bzw. in Form von Einfügen neuer oder Entfernen bereits vorhandener Zustände, generiert. Ein Nachkomme wurde jeweils mit seinem Elter verglichen und das Individuum mit der höheren Fitness wurde beibehalten, das andere verworfen. Grundlage dieses Vergleichs war die Übereinstimmung der jeweils generierten Sequenz mit der zu generierenden Sequenz. Ein Merkmal der Evolutionären Programmierung ist die ausschließliche Verwendung der Operatoren Mutation und Selektion. Die Individuen werden als eigene Spezien betrachtet, unter denen eine Rekombination ausgeschlossen ist. Stattdessen werden dabei die Abhängigkeiten der Verhaltensweisen zwischen Eltern und ihren Nachkommen betont, wodurch lediglich die stochastische Anpassung der Spezien (Individuen) an die vorliegende Umgebung realisiert werden soll.

Ebenfalls Mitte der 60er Jahre entwickelten Ingo Rechenberg und Hans–Paul Schwefel Verfahren zur experimentellen Optimierung von verschiedenen physikalischen Problemen, auf die keine traditionellen gradientenbasierten Methoden angewendet werden konnten. Diese Verfahren wurden als *Evolutionsstrategien* bekannt (Sch65). Ausgangspunkt der Verfahren war die bislang beste bekannte experimentelle Konfiguration, welche dann, analog zur genetischen Mutation, leicht verändert wurde. Erzielte die dadurch erhaltene neue Konfiguration bessere Ergebnisse, so wurde diese als neuer Ausgangspunkt verwendet, anderenfalls wurden weitere Mutationen vorgenommen. Bei den Evolutionsstrategien wurde erstmals das Konzept der Selbstanpassung realisiert. Dies geschah in Form einer Schrittweitenregelung der durchgeführten Mutationen. Diese wurde dynamisch an die erzielten Optimierungsfortschritte angepasst, woraus nachweislich eine Verbesserung des Verfahrens resultierte (Bey01).

[1] Ein Automat ist ein mathematisches Modell eines Gerätes, welches eine als Eingabe gegebene Zeichenfolge verarbeitet und eine entsprechende Ausgabe liefert.

Während sich die drei soeben beschriebenen Strömungen Evolutionärer Algorithmen mit der Bestimmung optimaler Parameter in Form von Zeichenketten oder Vektoren beschäftigen, ist das Ziel der *Genetischen Programmierung* die Erstellung von optimalen Programmen zur Berechnung von gegebenen Eingabe-Ausgabe-Paaren. Dazu bestimmen Genetische Algorithmen die gemäß der gegebenen Ein- und Ausgabe optimale Anordnungen von Funktionen innerhalb einer bestimmten Struktur, wie beispielsweise einer Baumstruktur, einer Sequenz (Folge von Assembler-Befehlen) oder auch eines Zustandsautomaten. Es soll ein Programm generiert werden, welches eine bestimmte Aufgabe erfüllt. Dieser Ansatz unterscheidet sich ebenfalls darin, dass die Anzahl, Art und Komplexität der zu bestimmenden Parameter variabel sein kann. Vorreiter auf diesem Gebiet ist John Koza, der in den späten 80er Jahren die Genetische Programmierung zur Erzeugung von einfachen in Baumstruktur vorliegenden LISP[2]-Programmen nutzte (Koz92).

Nachdem sich die vier großen Methodologien Evolutionärer Algorithmen viele Jahre selbständig und unabhängig voneinander entwickelt haben, beginnen deren Grenzen mehr und mehr aufzuweichen. So werden mittlerweile Vorgehensweisen und Techniken eines Paradigmas auf Probleme eines anderen Paradigmas angepasst und angewandt.

Die Particle Swarm Optimization ist ebenfalls als ein Evolutionärer Algorithmus zu klassifizieren, da es sich dabei um ein populationsbasiertes Suchverfahren handelt, welches das Konzept der Fitness verwendet. Sie beinhaltet Techniken sowohl der Genetischen Algorithmen, der Evolutionsstrategien als auch der Evolutionären Programmierung. So ist sie beispielsweise wie die Evolutionäre Programmierung sehr stark abhängig von stochastischen Prozessen und die Anpassung der Flugrichtungen in Abhängigkeit von der lokal und global besten Position ähnelt dem Rekombinations-Operator Genetischer Algorithmen. Ebenso existiert wie bei den Evolutionsstrategien ein Konzept der Selbstanpassung in Form des sich mit voranschreitender Optimierungszeit verringernden Schwungkraftgewichts (inertia weight). Auf die Particle Swarm Optimization sowie einige ihrer Varianten wird im Abschnitt 2.2 detailliert eingegangen.

2.1 Genetische Algorithmen (GA)

Genetische Algorithmen sind eine meta-heuristische Optimierungstechnik, welche die Darwin'sche Theorie der biologischen Evolution nachahmt. In den letzten Jahrzehnten hat vor allem ihre besondere Eignung zur Lösung von nichtlinearen, multimodalen und nicht kontinuierlichen Optimierungsproblemen die Aufmerksamkeit vieler Forscher erregt.

Ein genetischer Algorithmus arbeitet mit einem Satz von potentiellen Lösungen für das

[2] LISP (List Processing) ist eine 1958 entwickelte, syntaktisch einfache, funktionale Programmiersprache und gilt als eine der Programmiersprachen der Künstlichen Intelligenz.

zu lösende Optimierungsproblem, der Population von Individuen. Durch mehrfach iterativ angewandte Modifikationen der aktuellen Population, sollen stetig bessere Lösungen gefunden werden. Die Anwendung des Prinzips der natürlichen Selektion[3] soll schließlich zum Auffinden der optimalen Lösung führen.

In Abbildung 2.1 ist der Ablauf eines einfachen Genetischen Algorithmus' dargestellt. Am Anfang des Optimierungsprozesses steht die Initialisierung, welche die Generie-

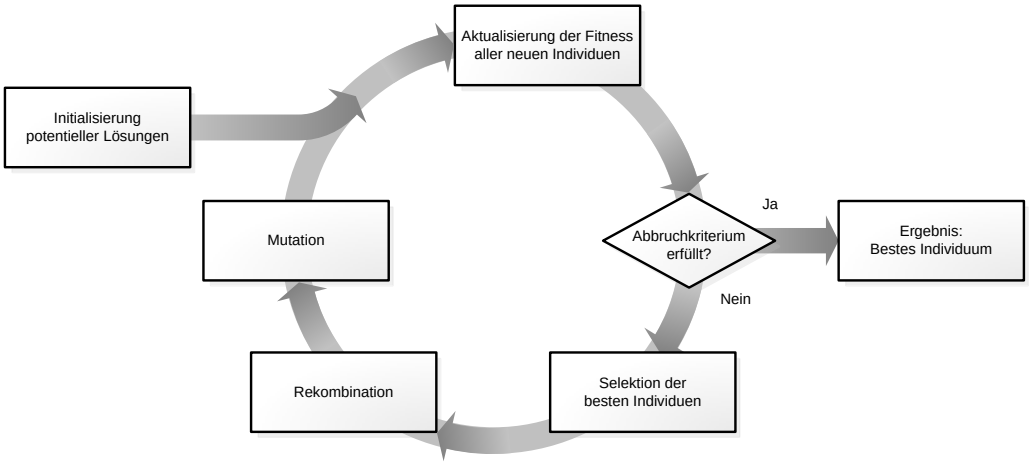

Abbildung 2.1: Allgemeiner Ablauf eines Genetischen Algorithmus

rung einer initialen Population beinhaltet. Falls kein Vorwissen über die Lage der zu suchenden Lösung vorliegt, wird die Startpopulation zufällig und gleichverteilt im Definitionsbereich der Variablen initialisiert. Anderenfalls kann die Startpopulation auch in der Umgebung der erwarteten Lösungsposition verteilt werden. Die Produktion der ersten Populationsgeneration ist mit der Berechnung der zu den Individuen gehörenden Fitness abgeschlossen.

Ist das ausgewählte Abbruchkriterium erfüllt, so kann das Individuum mit der besten Fitness als Optimierungsergebnis zurückgegeben werden. Falls nicht, werden zyklisch neue Generationen generiert, bis das Abbruchkriterium erfüllt ist. Dieser evolutionäre Kreislauf besteht aus den genetischen Operatoren *Selektion*, *Rekombination* und *Mutation*. Durch die Selektion wird je nach verwendeter Selektionsart ein Satz von Individuen in Abhängigkeit ihrer Fitness, in diesem Zusammenhang Eltern genannt, für die Produktion von neuen Individuen, den Nachkommen, ausgewählt. Die Eltern verwenden dann die Rekombination zur Erstellung ihrer Nachkommen, welche anschließend durch die Mutation, zufällig gemäß einer zuvor definierten Wahrscheinlichkeit, modifiziert werden. Anschließend werden diese in die Population eingefügt und ersetzen damit je nach Wiedereinfügeoperator beispielsweise die im Sinne der Fitnessfunktion schwächsten Individuen. Mit der Bewertung der neuen Individuen ist die Generierung der neuen Ge-

[3] Die natürliche Selektion beschreibt das wahrscheinlichere Überleben von Individuen mit höherer Fitness gegenüber Individuen mit geringerer Fitness.

neration abgeschlossen, von der eine, im Vergleich zur vorherigen Generation, bessere oder mindestens gleichwertige Gesamtfitness erhofft wird.

Es existieren eine Vielzahl von, in diesem Kontext anwendbaren Abbruchkriterien (*termination criteria*), wie z.B. das Erreichen eines bestimmten Fitnesswertes oder auch die allgemeine Konvergenz des Schwarms. Diese Kriterien besitzen jedoch keinen garantierten Abbruch, wodurch es zu keinem Ende der Optimierung kommt, falls der Algorithmus die Kriterien nicht erreicht. Derartige Kriterien müssen aus diesem Grund mit garantiert endenden Abbruchkriterien kombiniert werden. Beispiele dafür sind das Erreichen einer maximalen Anzahl von Generationen oder eine maximal zulässige Rechenzeit.

2.1.1 Genetische Operatoren

Dieser Abschnitt beschreibt die wichtigsten genetischen Operatoren, die Selektion, die Rekombination, die Mutation und das Wiedereinfügen, welche durch Genetische Algorithmen für die Erzeugung von neuen Populationsgenerationen verwendet werden. Zusätzlich wird auf unterschiedliche Populationsmodelle der Individuen und die mögliche Konkurrenz zwischen Unterpopulationen des regionalen Modells eingegangen.

Selektion

Die Selektion nimmt die an verschiedenen Stellen des Genetischen Algorithmus' durchzuführende Auswahl von Individuen in Abhängigkeit ihrer Fitness vor. Die Selektion wird beispielsweise zur Auswahl von Individuen für die Erzeugung von Nachkommen durchgeführt. Die Auswahlwahrscheinlichkeit eines Individuums hängt dabei von seiner Fitness ab, wodurch die Auswahl von qualitativ hochwertigeren Individuen favorisiert wird. Eine Selektion tritt ebenfalls bei der Auswahl der in einer Generation durch neu generierte Nachkommen zu ersetzenden Individuen auf. Ebenso wird sowohl bei der Auswahl der Individuen, die in andere Subpopulationen migrieren als auch bei der Auswahl jener Individuen, die bei Konkurrenz in andere Unterpopulationen wechseln müssen, eine Selektion durchgeführt.

Für die Selektion existieren unterschiedliche Konzepte, wobei im Folgenden genauer auf die beiden fitnessproportionalen Verfahren Rouletteselektion und Stochastic universal sampling sowie auf die Turnierselektion und die rangbasierte Abschneideselektion eingegangen wird.

Die *Rouletteselektion* (*roulette wheel selection* oder auch *stochastic sampling with replacement* genannt (Bak87)) ist das bekannteste Verfahren zur Auswahl von geeigneten Individuen und verfolgt den Ansatz der fitnessproportionalen Individuenauswahl, d.h. die Selektionswahrscheinlichkeit eines Individuums ist proportional zu seinem Fitness-

wert. Dabei werden allen Individuen des Selektionspools[4], entsprechend ihrer Fitness, Abschnitte auf einer Linie zugeordnet, deren Länge der gesamten Fitness des Selektionspools entspricht. Individuen mit hoher Fitness nehmen dabei mehr Platz in Anspruch als Individuen mit niedriger Fitness. Anschließend wird eine gleichverteilte Zufallszahl im Bereich zwischen 0 und der Länge der Linie generiert. Das Individuum, dessen Linienabschnitt diese Zahl beinhaltet wird ausgewählt. Dieser Algorithmus wird nun so oft ausgeführt, bis die gewünschte Anzahl von Individuen ausgewählt wurde. Das Verfahren ist in Abbildung 2.2 dargestellt. Äquivalent lässt sich dieses Vorgehen auch als Roulette

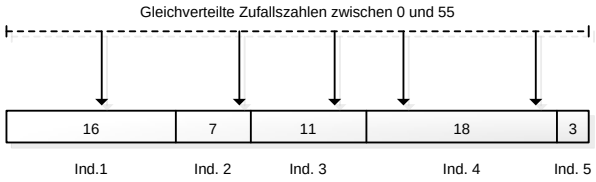

Abbildung 2.2: Illustration der Rouletteselektion. Die Selektionswahrscheinlichkeit eines Individuums (Größe der Abschnitte) ist proportional zu seiner Fitness.

veranschaulichen, bei dem die Individuen, analog zur oben genannten Anordnung auf der Linie, Bereichen eines Rouletterades zugeordnet werden. Das Ziehen einer Zufallszahl entspricht dann dem Drehen des Rades bzw. dem Wurf der Kugel. Bei der Rouletteselektion ist die Auswahl von Individuen mit hoher Fitness zwar wahrscheinlicher als die Auswahl von Individuen mit niedriger Fitness, sie ist allerdings nicht garantiert. So kann es in sehr seltenen Fällen auch vorkommen, dass ausschließlich die Individuen mit geringer Fitness ausgewählt werden. Dies ermöglicht allerdings auch das Entkommen des Algorithmus aus lokalen Optima, da auch Individuen mit geringer Fitness Nachkommen mit hoher Fitness erzeugen können.

Der Nachteil der Rouletteselektion wird mit dem *stochastic universal sampling* (Bak87) verhindert, bei dem die Individuen wie bei der Rouletteselektion entsprechend ihrer Fitness auf einer Linie verteilt sind. Im Gegensatz zur Rouletteselektion werden hierbei jedoch, wie in Abbildung 2.3 gezeigt, entsprechend der Anzahl der zu selektierenden Individuen Zeiger mit konstanten Abständen auf der Linie verteilt. Diese Abstände ergeben sich aus der Division der Linienlänge und der Anzahl der zu selektierenden Individuen. Die Position des ersten dieser Zeiger wird durch das Ziehen einer gleichverteilten Zufallsvariable im Bereich zwischen 0 und dem definierten Zeigerabstand bestimmt. Alle Individuen, auf dessen Linienbereich ein Zeiger zeigt, werden selektiert. Vorteile dieses Verfahrens sind zum einen die hohe Geschwindigkeit, da nur eine einzige Zufallszahl generiert werden muss, und zum anderen die Vermeidung der möglichen, ausschließlichen Selektion von Individuen niedriger Fitness. Anzumerken ist dabei, dass, wie auch bei der Rouletteselektion, Individuen mehrfach selektiert werden können. Während bei der

[4] Der Selektionspool ist eine Menge von Individuen, für die eine Selektion durchgeführt werden soll. Dies kann sowohl die gesamte Population als auch eine Teilmenge (Subpopulation) davon sein.

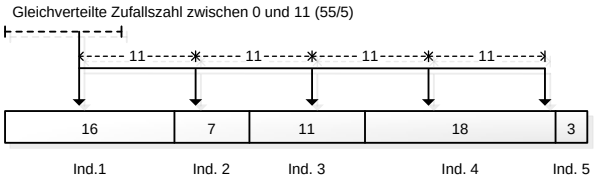

Abbildung 2.3: Exemplarische Illustration des stochastic universal sampling. Die gleichmäßig verteilten Zeiger definieren die zu selektierenden Individuen.

Rouletteselektion jedes Individuum unabhängig von seiner Fitness mehrfach selektiert werden kann, geschieht dies beim stochastic universal sampling nur, wenn das Individuum eine entsprechend hohe Fitness aufweist. Genauer gesagt ist die Selektion eines Individuums genau dann garantiert, wenn dessen normalisierte Fitness größer als der Kehrwert der Anzahl der zu selektierenden Individuen ist. Liegt die normalisierte Fitness zwischen diesem und dem doppelten Kehrwert, so wird es mindestens ein oder zwei mal selektiert.

Ein weiteres Selektionsverfahren ist die *Turnierselektion* (*tournament selection* (GD90)). Dabei werden zufällig und gleichverteilt mehrere Individuen des Selektionspools bestimmt und deren Fitnesswert verglichen. Das jeweils beste Individuum, d.h. jenes mit der höchsten Fitness, wird ausgewählt. Derartige Turniere werden so oft ausgeführt, bis die gewünschte Anzahl an zu selektierenden Individuen erreicht ist. Da bei diesem Verfahren lediglich die Ordnung der Individuen, nicht aber die absolute Fitness in Betracht gezogen wird, handelt es sich dabei um ein rangbasiertes und nicht um ein fitnessproportionales Selektionsverfahren. Abbildung 2.4 zeigt ein Beispiel mit drei Turnieren zwischen verschiedenen Individuen eines Selektionspools. Die Anzahl der für ein Tur-

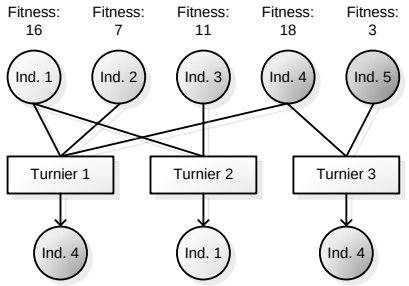

Abbildung 2.4: Exemplarische Illustration der Turnierselektion. Verschiedene Individuen treten in Turnieren gegeneinander an: Das mit der höchsten Fitness gewinnt.

nier zu berücksichtigenden Individuen bestimmt den Selektionsdruck[5]. Je geringer der Selektionsdruck für die Individuen einer Population ist, desto höher fällt die Diversität der von ihnen durchgeführten Suche aus.

[5] Der Selektionsdruck ist die Notwendigkeit eines Individuums, sich seiner Umgebung anzupassen. Er ist ein Maß für die Bevorzugung der besten gegenüber den schlechteren Individuen.

Die Abschneideselektion (*truncation selection*) zählt ebenfalls zu den rangbasierten Selektionsverfahren. Dabei werden alle Individuen gemäß ihrer Fitness sortiert und es werden so lange die besten Individuen ausgewählt, bis die gewünschte Anzahl erreicht ist. Diese Individuen besitzen eine sichere, alle anderen eine unmögliche Selektionswahrscheinlichkeit. In Abbildung 2.5 ist ein Beispiel dargestellt, in dem die besten drei Individuen selektiert werden.

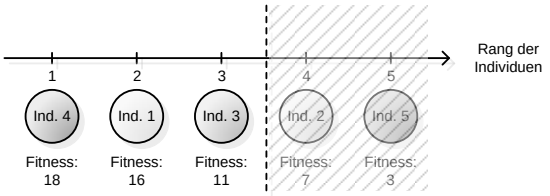

Abbildung 2.5: Exemplarische Illustration der Abschneideselektion. Ausgewählt werden hier die besten drei der nach ihrer Fitness sortierten Individuen.

Rekombination

Die Rekombination ist einer der wichtigsten Operatoren der Genetischen Algorithmen und basiert auf der Metapher der geschlechtlichen Fortpflanzung. Durch den Austausch und die Kombination von Informationen werden dabei aus mindestens zwei zuvor selektierten Individuen Nachkommen erzeugt. Im Folgenden wird der Einfachheit halber und in Anlehnung an die Biologie lediglich auf die Rekombination zweier Elternindividuen eingegangen. Es existieren unterschiedliche Verfahren, die je nach Datenrepräsentation und Anwendungsgebiet verwendet werden können. Häufig verwendete Varianten sind sowohl die *diskrete Rekombination*, welche sich auf alle Datentypen anwenden lässt, als auch die *intermediäre Rekombination*, die sich auf reelle und ganzzahlige Variablen anwenden lässt.

Wieviele Nachkommen innerhalb einer Generation erzeugt werden sollen, wird bestimmt durch die *Generationslücke* (*generation gap*), welche den Anteil der Nachkommen bezogen auf die Gesamtpopulation angibt.

Bei der *diskreten Rekombination* (*discrete recombination* (MSV93)) werden für die Nachkommen alle Variablenwerte unverändert von einem der beiden Elternindividuen übernommen. Welches Elternteil dabei für welche Variablenposition verwendet wird, unterliegt dem Zufall. Geometrisch gedeutet, werden die Nachkommen als Eckpunkte des durch die einbezogenen Eltern aufgespannten Hyperwürfels erzeugt. Gleichung 2.1 stellt den formalen Sachverhalt dar.

$$Var_i^{Nachkomme} = Var_i^{Elter_1} \cdot r_i + Var_i^{Elter_2} \cdot (1 - r_i) \tag{2.1}$$

Var_i^X beschreibt dabei die $i$te Variable des Individuums X und r_i ist eine gleichverteile Zufallsvariable, die die Werte 0 und 1 annehmen kann und für jede Dimension separat

bestimmt wird. Allgemein gesagt, wird für jede Variablenposition (Dimension) eines der zur Verfügung stehenden Elternindividuen durch einfaches Würfeln ausgewählt. Abbildung 2.6 verdeutlicht diesen Zusammenhang für ein zweidimensionales Beispiel mit zwei Elternindividuen.

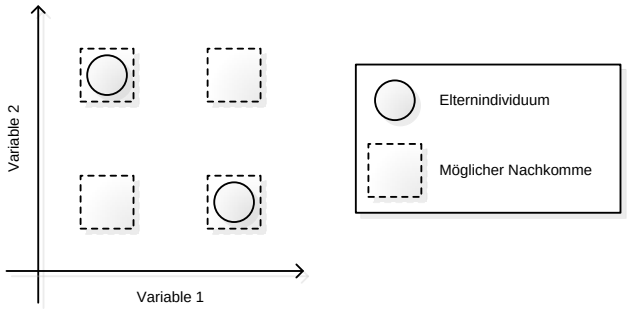

Abbildung 2.6: Exemplarische Illustration der diskreten Rekombination für zwei Dimensionen. Die Nachkommen werden dabei auf den Eckpunkten des durch die Elternindividuen aufgespannten Hyperwürfels erzeugt.

Die *intermediäre Rekombination* (*intermediate recombination* (MSV93)) berücksichtigt bei der Generierung der Nachkommen, im Gegensatz zur diskreten Rekombination, für jede Variable (Dimension) jeweils beide Erzeugerindividuen. Der Einfluss jedes Elternindividuums auf die einzelnen Variablen unterliegt auch hier dem Zufall. Da die gemäß Gleichung 2.2 erzeugten Variablen ein unterschiedlich gewichtetes Mittel der Elternwerte darstellen, muss ebenfalls darauf geachtet werden, dass die erzeugten Variablen dem zugrundeliegenden Datentyp entsprechen.

$$Var_i^{Nachkomme} = Var_i^{Elter_1} \cdot e_i + Var_i^{Elter_2} \cdot (1 - e_i) \tag{2.2}$$

Der Parameter e_i ist dabei eine gleichverteilte Zufallsvariable im Bereich $[-d, 1+d]$. Für den Skalierungsfaktor d wird oft ein Wert von 0.25 gewählt. Geometrisch veranschaulicht entspricht das der Erzeugung der Nachkommen innerhalb des leicht vergrößerten, durch die Variablenwerte aller Elternindividuen aufgespannten Hyperwürfels. Diese in Abbildung 2.7 dargestellte Vergrößerung resultiert aus dem Skalierungsfaktor, der der sonst vorkommenden stetigen Verkleinerung des Variablenbereichs entgegen wirken soll (Poh00).

Mutation

Die Mutation wird in der Regel nach der Rekombination mit einer geringen Wahrscheinlichkeit auf die neu generierten Individuen angewendet. Die Individuen werden geringfügig modifiziert, um die Diversität der gesamten Population zu erhalten und der vorzeitigen Konvergenz in lokale Optima entgegenzuwirken, da durch die Mutation eine

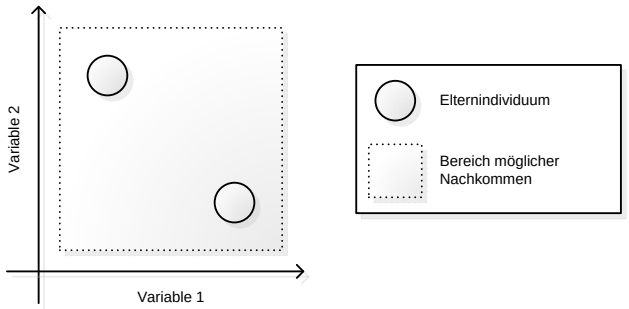

Abbildung 2.7: Exemplarische Illustration der intermediären Rekombination für zwei Dimensionen. Die Nachkommen werden dabei innerhalb des durch die Elternindividuen aufgespannten Hyperwürfels erzeugt.

zu große Ähnlichkeit der Individuen verhindert wird. Das Ausmaß der Modifikationen bezeichnet man als *Mutationsschritt* und deren Häufigkeit als *Mutationsrate*.

Es hat sich gezeigt, dass eine Adaption dieser Parameter während der Suche von Vorteil ist (Bey01). Da der Genetische Algorithmus zu Beginn der Optimierung weite Bereiche des Suchraums erkunden und erst bei voranschreitender Suche detailliertere Auflösungen verwenden sollte, ist eine proportional zum Optimierungsfortschritt zu haltende Mutationsschrittweite vorteilhaft (Selbstadaption (Bey01; ES03)). Die häufigsten Varianten der Mutation sind die *Mutation reeller und ganzzahliger Variablen* sowie die *Mutation binärer Variablen*, welche im Folgenden näher erläutert werden.

Bei der *Mutation reeller und ganzzahliger Variablen* wird gemäß Gleichung 2.3 mit einer geringen Wahrscheinlichkeit (Mutationsrate) jeder Variable der zu mutierenden Individuen ein zufälliger Wert (Mutationsschritt) hinzugefügt oder abgezogen. Dieser Wert wird nach oben sowohl durch den Definitionsbereich der Variable als auch durch den Mutationsbereich[6] r und nach unten durch die Mutationspräzision[7] k beschränkt. Gleichung 2.3 zeigt einen Mutationsoperator ohne Anpassung der Mutationsschrittweite ((Poh00)).

$$
\begin{aligned}
Var_i^{mutiert} &= Var_i + s_i \cdot r_i \cdot a_i \qquad (2.3)\\
mit & \\
& i \in \{1, 2, ..., n\}\, gleichverteilt,\\
& s_i \in \{-1, 1\}\, gleichverteilt,\\
& r_i = r \cdot Definitionsbereich\ der\ Variable\ i,\\
& a_i = 2^{-u \cdot k}, u \in [0, 1]\, gleichverteilt.
\end{aligned}
$$

Die *Mutation binärer Variablen* realisiert einen einfachen Wechsel einiger weniger Variablenzustände des zu mutierenden Individuums. Welche und wie viele der Variablen

[6] Der Mutationsbereich (*mutation range*) definiert den Anteil des Definitionsbereichs der Variablen, der der Mutation zur Verfügung steht.

[7] Die Mutationspräzision (*mutation precision*) definiert die minimal zulässige Mutationsschrittweite.

ihren Zustand wechseln, unterliegt dem Zufall. Da jede Variable zwei Zustände annehmen kann, beträgt die Länge des Mutationsschritts dabei genau 1.

Wiedereinfügen

Das Wiedereinfügen (*reinsertion*) bestimmt, welche der durch die Rekombination und anschließende Mutation erzeugten Individuen in die ursprüngliche Population aufgenommen werden sollen. Der Parameter *Wiedereinfügerate* (*reinsertion rate*) bestimmt dabei den maximalen Anteil der Individuen der ursprünglichen Population, die durch Nachkommen ersetzt werden sollen. Eine Wiedereinfügerate von 1.0 bedeutet dabei, dass alle Individuen durch Nachkommen ersetzt werden. Ist die Anzahl der zu ersetzenden Individuen kleiner als die in Abhängigkeit der Generationslücke erstellten Nachkommen, so muss entschieden werden, welche der Nachkommen eingefügt werden sollen. Dies kann sowohl von deren erreichter Fitness als auch von deren Alter[8] abhängen. Abhängig von der gewählten Generationslücke und der Wiedereinfügerate lassen sich mehrere Strategien unterscheiden.

Beim *einfachen Wiedereinfügen* (*pure reinsertion*) wird für jedes Individuum der Population ein Nachkomme erzeugt, durch das es ersetzt wird. Der Fitnessverlauf des jeweils besten Individuums während der Optimierung ist nicht monoton, d.h. gute Lösungen können wieder verworfen werden. Die Generationslücke beträgt hierbei wie die Wiedereinfügerate genau 1.0.

Ist die Generationslücke oder die Wiedereinfügerate kleiner als 1.0, so muss entschieden werden, welche Individuen der Population durch die Nachkommen ersetzt werden sollen. Das *Zufällige Wiedereinfügen* (*union reinsertion*) ersetzt diese gleichverteilt zufällig ohne Beachtung deren Fitness, das *Elitäre Wiedereinfügen* (*elitest reinsertion*) ersetzt die jeweils schlechtesten Individuen.

Ist die Anzahl der erzeugten Nachkommen größer als die Anzahl der einzufügenden Individuen (Generationslücke $>$ Wiedereinfügerate), so muss entschieden werden, welche der erzeugten Nachkommen eingefügt werden. Dies geschieht üblicherweise mittels der Abschneideselektion, es werden dafür also die besten Individuen ausgewählt.

Populationsmodelle

Eine hilfreiche Erweiterung der Genetischen Algorithmen ist die Einführung und Nutzung verschiedener Populationsmodelle, welche die sonst verwendete Gesamtpopulation in Subpopulationen unterteilt oder anderweitig die Sichtbarkeit der Individuen untereinander regelt. Trotz der durchgeführten Trennung der Subpopulationen stehen sie noch in Verbindung und können Informationen austauschen. Eine bekannte Klassifikation von

[8] Das Alter eines Individuums entspricht hier der Anzahl der bereits überlebten Generationen.

regionalen Modellen stammt von Markus Schwehm (Sch97), der die Populationsmodelle in Abhängigkeit von der Selektionsreichweite der Individuen unterscheidet, also welche Individuen dabei in einem Selektionspool vorkommen können.

Wie in Abbildung 2.8 dargestellt, ergibt sich daraus die Unterscheidung in das globale, das lokale und das regionale Populationsmodell (nach (Poh00)). Beim globalen Populati-

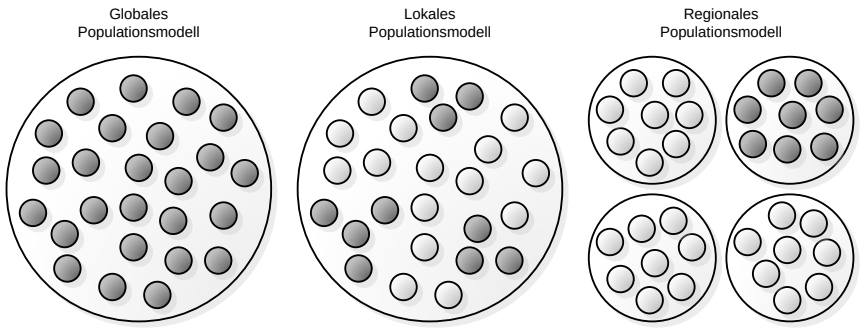

Abbildung 2.8: Klassifikation der Populationsmodelle nach der Selektionsreichweite. Alle dunkel eingefärbten Individuen gehören jeweils zu einem Selektionspool.

onsmodell können sich alle Individuen in einem Selektionspool befinden - es findet keine Unterteilung der Gesamtpopulation statt. Somit entspricht es der klassischen Topologie evolutionärer Algorithmen. Alle Individuen können, unabhängig von deren geographischer Lage, durch die Selektion als Elternindividuen zur Erzeugung von Nachkommen ausgewählt werden.

Beim lokalen Populationsmodell (auch Nachbarschaftsmodell genannt) finden Wechselwirkungen nur zwischen einem Individuum und seinen örtlich nächsten Nachbarn statt. Diese Nachbarschaft kann, abhängig von der Topologie und der Distanz zwischen den Individuen, unterschiedlich groß sein und entspricht genau dem jeweiligen Selektionspool.

Das regionale Populationsmodell (auch Migrationsmodell genannt) unterteilt die Individuen in Subpopulationen, welche sich unabhängig voneinander entwickeln. Alle Individuen einer Subpopulation befinden sich in einem Selektionspool - dabei handelt es sich um mehrere, parallel arbeitende, globale Populationen. Jede Subpopulation wird von einem eigenen Evolutionären Algorithmus bearbeitet. Diese selbständige Entwicklung findet für eine vorgegebene Anzahl von Generationen, der so genannten *Isolationszeit*, statt. Danach werden Informationen, in Form eines Individuenaustauschs, zwischen den Subpopulationen übertragen, was als *Migration* bezeichnet wird. Anschließend entwickeln sich die Subpopulationen wieder selbständig, bis die Isolationszeit erneut abgelaufen ist. Die Migration kann hauptsächlich durch drei Parameter beeinflusst werden: Durch die *Migrationstopologie*, das *Migrationsintervall*, die *Migrationsrate* und durch die *Migrationsauswahl*. Die Migrationstopologie bestimmt, welche Subpopulationen verbunden sind und während der Migration ihre Individuen austauschen. Das Migrationsintervall

bestimmt, wie häufig Migrationen stattfinden und entspricht damit der Isolationszeit. Die Migrationsrate legt den Anteil der für die Migration ausgewählten Individuen fest und die Migrationsauswahl bestimmt die Art und Weise, wie daraus die tatsächlich migrierenden Individuen ausgewählt werden. Dadurch wird der Selektionsdruck zusätzlich erhöht.

Konkurrenz im regionalen Populationsmodell

Das regionale Modell lässt sich durch das Prinzip der konkurrierenden Unterpopulationen (*competing subpopulations*) erweitern. Die Anzahl der Individuen innerhalb der Unterpopulationen variiert dabei während der Optimierung in Abhängigkeit von deren Leistungsfähigkeit. Erfolgreichere Unterpopulationen erhalten durch Migration Individuen von weniger erolgreichen Unterpopulationen. Das *Konkurrenzintervall* definiert dabei die Anzahl der Generationen, die zwischen zwei Wettbewerben liegen und in denen sich die Subpopulationen selbständig entwickeln können. Die *Konkurrenzrate* bestimmt die Anzahl der Individuen, welche nach einem Wettbewerb von der unterlegenen an die überlegene Unterpopulation abgegeben werden und wird als Anteil der vorhandenen Individuen angegeben.

Um eine vollständige Auflösung einer erfolglosen Unterpopulation zu verhindern, wird eine minimale Anzahl von Individuen festgelegt. Unterpopulationen können somit nach dem Erreichen dieser Grenze keine weiteren Individuen mehr abgeben. Sie entwickeln sich weiter und können aufgrund der eigenen Verbesserung oder der Verschlechterung einer anderen Unterpopulation wieder Individuen zugewiesen bekommen.

2.2 Particle Swarm Optimization (PSO)

Im Vergleich zu den Genetischen Algorithmen ist die Particle Swarm Optimization eine relativ neue Optimierungstechnik des Schwarm-Intelligenz-Paradigmas. Sie wurde 1995 erstmals von Kennedy und Eberhart (KE95; EK95) vorgestellt. Inspiriert von sozialen Verhaltensmetaphern und der Schwarmtheorie wurden Methoden entwickelt, welche die effiziente Optimierung nichtlinearer multimodaler Funktionen ermöglichen. PSO simuliert Kollektive wie Tierherden, Vogel- oder Fischschwärme.

Ähnlich zu Genetischen Algorithmen initialisiert ein PSO-Algorithmus eine Startpopulation bestehend aus Zufallslösungen, welche als Partikel bezeichnet werden. Jedes Partikel kennt seine eigene aktuelle Position im Datenraum, seine derzeitige Geschwindigkeit und Flugrichtung sowie die persönlich beste bisher gefundene Position. Der Schwarm als Ganzes kennt zusätzlich die global beste, bisher von allen Mitgliedern gefundene Position. Die iterative Anwendung von Aktualisierungsregeln führt zu einer stochas-

tischen Manipulation der Geschwindigkeiten[9] und Flugbahnen der Partikel. Während des Optimierungsprozesses erkunden die Partikel den multidimensionalen Raum, wobei deren Trajektorien sowohl von ihrer persönlichen Erfahrung als auch von der anderer Partikel abhängen kann. Partikel können, abhängig von der jeweils verwendeten PSO-Variante (vgl. Abschnitt 2.2.1), beispielsweise nur von Partikeln aus einer bestimmten Nachbarschaft oder aber auch von allen Partikeln des Schwarms lernen. Die verwendete Lernstrategie führt zu einer intensiveren Erkundung von Regionen, die sich als profitabel herausgestellt haben.

Abbildung 2.9 stellt den allgemeinen Ablauf eines PSO-Algorithmus dar. Nach der

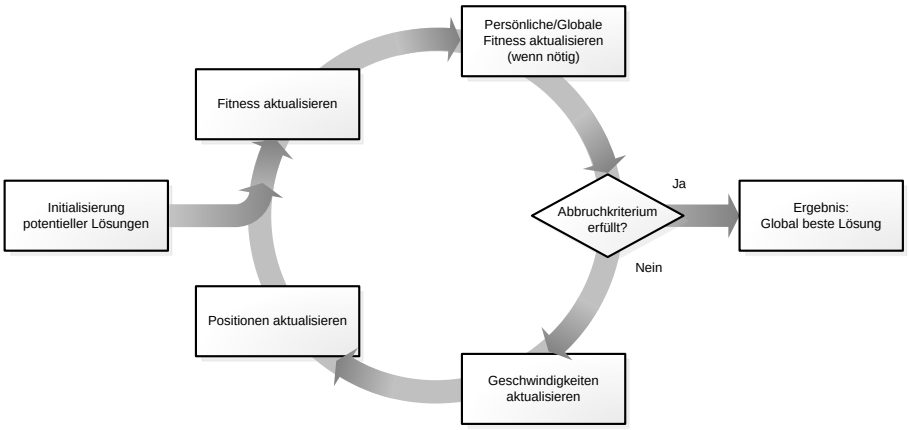

Abbildung 2.9: Allgemeiner Ablauf eines PSO-Algorithmus

gleichverteilt zufälligen Initialisierung der Partikel im Datenraum wird die Fitness selbiger bestimmt und die entsprechenden persönlich besten und die global beste Position aktualisiert. Als Abbruchkriterium kann hier, wie bei den Genetischen Algorithmen, entweder ein bestimmter erreichter Fitnesswert, das Erreichen einer maximalen Anzahl von Iterationen oder aber auch die allgemeine Konvergenz des Schwarms gewählt werden. Ist das Abbruchkriterium noch nicht erfüllt, so werden die Geschwindigkeiten nach einer algorithmenspezifischen Formel aktualisiert, woraus sich neue Positionen für die Partikel ergeben. Anschließend wird die Fitness der Partikel neu bestimmt. Bei Erreichen des Abbruchkriteriums gilt die global beste Position als Lösung.

Im Folgenden soll die ursprüngliche von Kennedy und Eberhart vorgestellte Version des PSO-Algorithmus' erläutert werden. Die D-dimensionalen Vektoren $x_i = (x_i^1, ..., x_i^d, ..., x_i^D)$ und $v_i = (v_i^1, ..., v_i^d, ..., v_i^D)$ repräsentieren die Position und die Geschwindigkeit des $i$ten Partikels. Die Position muss sich im Definitionsbereich der zu optimierenden Variablen befinden, sie wird also durch die maximalen und minimalen Schranken der einzelnen Dimensionen eingeschränkt: $x_i^d \in \left[lb^d, ub^d\right]$. Die Geschwindigkeiten der Partikel sollen ebenfalls auf einen maximalen Wert beschränkt sein: $v_i^d \in \left[-V_{max}^d, V_{max}^d\right]$.

[9] Im Folgenden wird der Begriff Geschwindigkeit eines Partikels für dessen Geschwindigkeitsvektor verwendet. Dieser gibt zusätzlich Aufschluss über die derzeitige Flugrichtung.

Dabei sollte V^d_{max} der Hälfte des Definitionsbereichs entsprechen, um eine Überschreitung der Definitionsbereichsgrenzen möglichst zu verhindern. Die Geschwindigkeit und die Position der $d$ten Dimension des $i$ten Partikels wird zu Beginn gleichverteilt zufällig im Definitionsbereich initialisiert: $v^d_i = rand(-V^d_{max}, V^d_{max})$ und $x^d_i = rand(lb^d, ub^d)$. Anschließend werden die Geschwindigkeiten und danach die Positionen zum Zeitpunkt t wie folgt aktualisiert:

$$\begin{aligned} v^d_i(t) &\leftarrow v^d_i(t-1) + c_1 \cdot \hat{r}^d_i \cdot (pbest^d_i(t-1) - x^d_i(t-1)) \\ &\quad + c_2 \cdot \tilde{r}^d_i \cdot (gbest^d(t-1) - x^d_i(t-1)) \qquad (2.4) \\ x^d_i(t) &\leftarrow x^d_i(t-1) + v^d_i(t) \qquad (2.5) \end{aligned}$$

Die Beschleunigunskoeffizienten c_1 und c_2 gewichten den Einfluss der kognitiven (c_1) und der sozialen (c_2) Komponenten im Verhältnis zur aktuellen Flugrichtung des Partikels. $\hat{r}^d_i$ und $\tilde{r}^d_i$ sind gleichverteilte Zufallsvariablen zwischen 0.0 und 1.0, die für jedes Partikel i und für jede Dimension d separat erzeugt werden. Die beste bislang von Partikel i erreichte Position wird als $pbest_i$ und die beste bislang vom gesamten Schwarm erreichte Position als $gbest$ bezeichnet.

Sowohl die Geschwindigkeiten als auch die daraus resultierenden neuen Positionen müssen nach der Aktualisierung auf die zugelassenen Bereiche beschränkt werden. Dafür werden die folgenden Ansätze verfolgt:

$$v^d_i \leftarrow \begin{cases} -V^d_{max} & , falls\ v^d_i < -V^d_{max} \\ V^d_{max} & , falls\ v^d_i > V^d_{max} \end{cases} \qquad (2.6)$$

$$x^d_i \leftarrow \begin{cases} lb^d & , falls\ x^d_i < lb^d \\ ub^d & , falls\ x^d_i > ub^d \end{cases} \qquad (2.7)$$

Abbildung 2.10 verdeutlicht den Einfluss von sowohl der aktuellen Flugrichtung, der bislang besten gefunden Position $pbest_i$ als auch der besten bislang von allen Partikeln des Schwarms gefundenen Position $gbest$ auf Partikel i gemäß den Aktualisierungsregeln 2.4 und 2.5. Nach der Aktualisierung der Positionen folgt die Bewertung der Partikel anhand der zu optimierenden Fitnessfunktion f und die anschließende eventuelle Überschreibung der persönlich besten Position:

$$pbest_i(t) \leftarrow \begin{cases} pbest_i(t-1) & , falls\ f(x_i(t)) \geq f(pbest_i(t-1)) \\ x_i(t) & , sonst. \end{cases} \qquad (2.8)$$

Nach jeder Aktualisierung der persönlich besten Positionen wird asynchron ebenfalls die globale beste Position aktualisiert:

$$gbest(t) \leftarrow min(f(pbest_1(t)), f(pbest_2(t)), ..., f(pbest_{ps}(t))). \qquad (2.9)$$

Der Parameter ps (*population size*) entspricht der Anzahl der Partikel im Schwarm. Die asynchrone Aktualisierung der global besten Position ermöglicht die sofortige Einbeziehung der eventuell gefundenen vielversprechenden Gebiete in die nachfolgenden Positionsaktualisierungen aller Partikel.

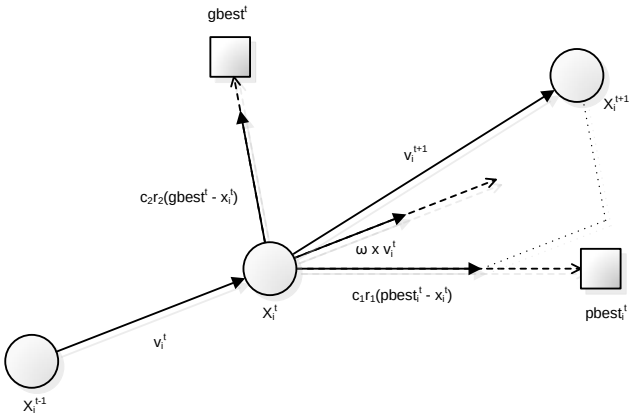

Abbildung 2.10: Einfluss der aktuellen Flugrichtung sowie der kognitiven und der sozialen Komponenten eines Partikels auf die Bestimmung der Folgeposition.

Die ursprüngliche Version der Particle Swarm Optimization aus (KE95) verwendet eine globale Nachbarschaft, d.h. alle Partikel kennen die beste Position aller anderen Partikel des Schwarms. Alle Partikel befinden sich in einer direkten Nachbarschaft. Eberhart und Kennedy bezeichnen diese PSO-Variante auch als GBest-Modell (*global best*) (EK95). Zusätzlich wurde das LBest-Modell (*local best*) vorgestellt. Dieses verwendet eine Ring-Nachbarschaft, bei der jedes Partikel genau n Nachbarn besitzt, die jeweils $n/2$ links und die $n/2$ rechts anliegenden Partikel in der Partikelliste. Jedes Partikel lernt dabei nicht mehr von der global besten Position des gesamten Schwarms sondern nur noch von der besten Position der Partikel seiner Nachbarschaft. Beide Nachbarschaften sind in Abbildung 2.11 dargestellt. Ein Vorteil des GBest-Modells ist dessen Geschwindigkeit, welche sich daraus ergibt, dass alle Partikel sofort in Richtung der bislang besten gefundenen Region des Suchraums fliegen und diese genauer erkunden. Daraus ergibt sich allerdings auch der Nachteil der vorzeitigen Konvergenz in lokale Optima, da der Erfolg der Suche stark von der Initialisierung und der dabei gefundenen besten Position abhängt. Das LBest-Modell ist im Vergleich zum GBest-Modell robuster gegenüber lokalen Optima, da die einzelnen Partikelnachbarschaftsgruppen erst separat verschiedene Regionen nach optimalen Positionen absuchen. Dies reduziert allerdings auch die Konvergenzgeschwindigkeit.

2.2.1 Varianten

In den vergangenen Jahren wurden viele Änderungen und Erweiterungen der PSO erarbeitet mit dem Ziel, diese auf verschiedene Art und Weise zu verbessern. Angestrebt waren dabei hauptsächlich zwei Ziele: eine schnellere Konvergenz und das Erreichen von qualitativ hochwertigeren Lösungen.

Eine dieser Verbesserungen war die von Shi und Eberhart (SE98a) vorgestellte Einfüh-

rung des *inertia weight* Parameters ω in die Aktualisierungsregel der Geschwindigkeiten:

$$\begin{aligned} v_i^d(t) &\leftarrow \omega \cdot v_i^d(t-1) + c_1 \cdot \hat{r}_i^d \cdot (pbest_i^d(t-1) - x_i^d(t-1)) \\ &\quad + c_2 \cdot \tilde{r}_i^d \cdot (gbest^d(t-1) - x_i^d(t-1)) \end{aligned} \tag{2.10}$$

Das inertia weight (die Trägheit) kontrolliert den Einfluss des aktuellen Moments des zu aktualisierenden Partikels auf die Berechnung der neuen Position. Ein großes Gewicht fördert die globale Erkundung des Suchraums während ein kleines Gewicht die lokale Feinabstimmung ermöglicht. Shi und Eberhart haben vorgeschlagen, das inertia weight abhängig vom Optimierungsfortschritt zu verringern (SE98b). Durch das große Gewicht zu Beginn des Optimierungsprozesses wird die Erkundung des Suchraums unter Vermeidung von lokalen Optima unterstützt, wohingegen das sich mit zunehmendem Fortschritt der Optimierung verringernde Gewicht die Konvergenz zum globalen Optimum begünstigt. Aufgrund einer durchgeführten Analyse des Konvergenzverhaltens der PSO wurde von Clerc und Kennedy (CK02) der Elastizitätskoeffizient χ (*constriction factor*) eingeführt:

$$\begin{aligned} v_i^d(t) &\leftarrow \chi \cdot [v_i^d(t-1) + c_1 \cdot \hat{r}_i^d \cdot (pbest_i^d(t-1) - x_i^d(t-1)) \\ &\quad + c_2 \cdot \tilde{r}_i^d \cdot (gbest^d(t-1) - x_i^d(t-1))] \end{aligned} \tag{2.11}$$

$$\chi \leftarrow \frac{2}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|} \; mit \; \varphi = c_1 + c_2, \varphi > 4 \tag{2.12}$$

Dieser kontrolliert und schränkt die Geschwindigkeiten ein und garantiert dadurch die Konvergenz bei gleichzeitiger Verbesserung der Konvergenzgeschwindigkeit.

Ebenso wurde versucht, die PSO durch Variation der Topologie der Schwärme zu verbessern. So untersuchten Kennedy und Mendes (Ken99; KM02) verschiedene Topologien und Nachbarschaften der Partikel. Sie stellten fest, dass große Nachbarschaften zur Lösung von einfachen Problemen und kleine Nachbarschaften zur Lösung von komplexeren Problemen geeignet sind. Die besten Ergebnisse lieferte die Verwendung der Von Neumann Nachbarschaft. In Abbildung 2.11 sind die drei Nachbarschaftsformen globale Nachbarschaft, Ring- und Von-Neumann-Nachbarschaft dargestellt. Es wurden ebenfalls variable Nachbarschaftsmodelle untersucht. In (Sug98) verwendete Suganthan variable, sich vergrößernde Nachbarschaften: Zu Beginn der Optimierung besaßen die Partikel eine sehr kleine Nachbarschaft, welche im Laufe des Optimierungsprozesses dynamisch bis zur globalen Nachbarschaft vergrößert wurden. Hu und Eberhart (HE02) stellten ein Verfahren vor, welches die Nachbarschaft eines Partikels in jeder Generation neu definierte, indem bei jeder Iteration die n räumlich nächsten Partikel als Nachbarn betrachtet wurden.

Nachdem die Beschränkung der Partikelpositionen auf den Definitionsbereich lange Zeit durch simples Setzen der überschreitenden Dimensionen auf den jeweiligen Extremwert sichergestellt wurde (wie in den Formeln 2.6 und 2.7), wurden weitere Ansätze entwickelt, die die Definitionsbereichsgrenzen als Wände interpretieren, an denen die

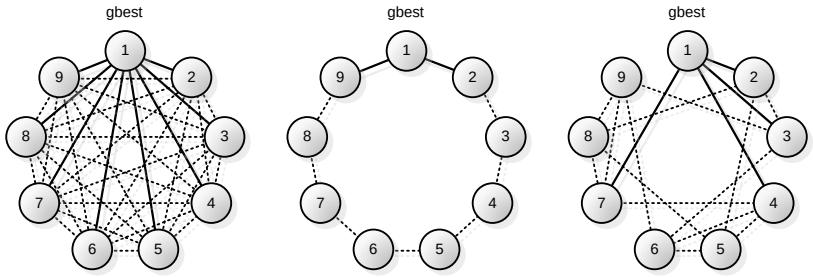

Abbildung 2.11: Exemplarische Darstellung der Nachbarschaften (a) Globale Nachbarschaft, (b) Ring-Nachbarschaft mit einer Nachbarschaftsgröße von 2, (c) Von Neumann Nachbarschaft.

Partikel beispielsweise abprallen können. Für die Beschränkung der Positionen existieren beispielsweise die in Abbildung 2.12 dargestellten Verfahren. Die von Robinson und

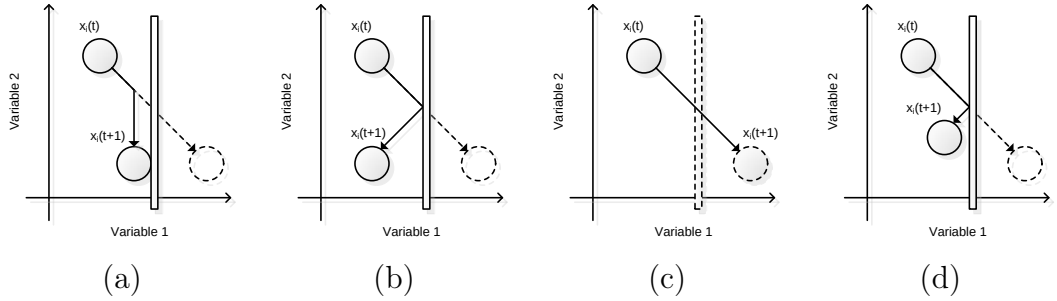

Abbildung 2.12: Verfahren zur Beschränkung der Partikelpositionen auf die erlaubten Bereiche. (a) Absorbierende Wände, (b) Reflektierende Wände, (c) Unsichtbare Wände, (d) Dämpfende Wände

Rahmat-Samii (RRS04) vorgestellten Verfahren der *absorbierenden*, *reflektierenden* und *unsichtbaren Wände*, wurden von Huang und Mohan (HM05) untersucht. Daraufhin wurde ein weiteres hybrides Verfahren vorgestellt: das Konzept der *dämpfenden Wände*, welches die Vorteile der absorbierenden und der reflektierenden Wände vereint. Bei den absorbierenden Wänden wird die Energie des Partikels beim Auftreffen auf die imaginäre Wand einer bestimmten Dimension vollständig absorbiert: Die Bewegung wird für diese Dimension auf Null gesetzt. Die reflektierenden Wände befördern das Partikel beim Auftreffen auf die Wand zurück in den Suchbereich. Die dämpfenden Wände sind eine Mischung aus diesen beiden Verfahren. Hierbei wird in Abhängigkeit einer gleichverteilten Zufallsvariable zwischen 0 und 1 bestimmt, wie stark die Partikel beim Auftreffen auf die Wand reflektiert werden. Beim Konzept der unsichtbaren Wände kann ein Partikel zwar den Definitionsbereich verlassen, dessen Fitness wird allerdings nur dann berechnet und aktualisiert, wenn es sich innerhalb des Suchraums befindet. Da sich die persönlich besten Positionen aller Partikel nur innerhalb des Definitionsbereichs befinden können, wird das Partikel früher oder später in selbigen zurückkehren.

Die Hybridisierung der Particle Swarm Optimization mit Methoden der Genetischen Algorithmen ist ein weiteres Gebiet der Forschung. So konnten Verbesserungen durch die Einführung genetischer Operatoren wie die Selektion, die Rekombination oder die Mutation in vielen Arbeiten erreicht werden (z.B.: (JC06; DKC05; JTR05)).

Zwei weitere vielversprechende Verbesserungsansätze für die Particle Swarm Optimization sind die beiden Verfahren *CL-PSO* und *G-PSO*, die in den folgenden Abschnitten vorgestellt werden.

Comprehensive Learning PSO (CL-PSO)

Ein neuer und vielversprechender Ansatz zur Verbesserung der Leistung der ursprünglich vorgestellten Variante der Particle Swarm Optimization ist der *Comprehensive Learning Particle Swarm Optimizer* (JJLB06) (CL-PSO). Dieser wendet eine neue Lernstrategie an, bei der jedes Partikel i für jede Dimension in Abhängigkeit von der Lernrate Pc_i von unterschiedlichen Partikeln lernt. Dies geschieht, bis das Partikel über eine bestimmte Anzahl von Iterationen, hier *refreshing gap* m genannt, keine Verbesserungen mehr erreicht, woraufhin die Partikelzuordnungen neu gesetzt werden.

Die Lernrate Pc_i des Partikels i ist ein Wert zwischen 0.05 und 0.5, der bestimmt, ob Partikel i für die aktuelle Dimension von seiner eigenen bisher besten gefundenen Position ($pbest_i$) oder von der eines anderen Partikels lernt. Dieser invariante Wahrscheinlichkeitswert wird für jedes Partikel während der Initialisierung separat bestimmt, um variierende Erkundungs- und Sondierungsfähigkeiten aller Partikel zu gewährleisten. Die Wahrscheinlichkeiten werden den Partikeln nach folgender, von den Autoren empirisch bestimmten Formel zugewiesen:

$$Pc_i = 0.05 + 0.45 \cdot \frac{\left(\exp\left(\frac{10(i-1)}{ps-1}\right) - 1\right)}{exp(10) - 1} \tag{2.13}$$

Wenn die betrachtete Dimension von der besten Position eines anderen Partikels gelernt werden soll, wird dieses Partikel ähnlich der Turnierselektion der Genetischen Algorithmen bestimmt. Es werden zufällig zwei Partikel des Schwarms ausgewählt und das bessere selektiert. Nach der Durchführung dieser Vorgehensweise für jede Dimension d liegt eine Liste $f_i(d)$ vor, welche die Indizes der im Lernprozess für die einzelnen Dimensionen einzubeziehenden Partikel enthält. Diese Liste wird in der folgenden Aktualisierungsregel der Geschwindigkeiten verwendet:

$$v_i^d(t) \leftarrow \omega \cdot v_i^d(t-1) + c \cdot r_i^d \cdot (pbest^d_{f_i(d)}(t-1) - x_i^d(t-1)) \tag{2.14}$$

Der Parameter c ist dabei ein Beschleunigungskoeffizient, der die sozialen und kognitiven Komponenten im Vergleich zum aktuellen Moment des Partikels gewichtet. Vorgeschlagen wurde, die folgende Variablenbelegung zu verwenden: $m = 7$, $c = 1.49445$ und ω fortschrittsabhängig von 0.9 auf 0.4 verringern (JJLB06).

Um die Partikel vor dem Verlassen des Definitionsbereichs zu hindern, wird bei diesem Algorithmus das Konzept der *unsichtbaren Wände* verwendet (vgl. Abbildung 2.12).

Die Autoren verglichen CL-PSO mit insgesamt acht anderen PSO-Varianten und konnten zeigen, dass dieser für komplexe, multimodale Probleme signifikant bessere Lösungen lieferte. Schwächen wurden gleichzeitig für einfache, vor allem unimodale Optimierungsprobleme aufgezeigt.

Gregarious PSO (G-PSO)

Eine weitere Variante der PSO stellt der *Gregarious Particle Swarm Optimizer* (PB06) (G-PSO) dar. Dabei erkunden die Partikel gemeinsam den Suchraum zur Aufspürung von lokalen Optima unter alleiniger Einbeziehung von sozialem Wissen, nämlich der global besten Position. Um jedoch die vorzeitige Konvergenz in lokale Optima zu vermeiden, werden die Geschwindigkeiten jener Partikel, die sich sehr nah an bereits gefundenen lokalen Optima befinden, zufällig reinitialisiert, um diese Optima verlassen und noch bessere Lösungen finden zu können, welche dann als Anziehungspunkt der restlichen Partikel dienen.

Biologisch motiviert ist dieses Verfahren durch Vogelschwärme auf der Suche nach Futter. Hat einer der Vögel eine Region mit Futter gefunden, folgen ihm die anderen, welche allerdings auch im Umkreis dieser Region suchen. Ist der Futtervorrat aufgebraucht, verlässt einer nach dem anderen die Region, um neue Futterstellen ausfindig zu machen.

Gleichung 2.15 zeigt die Aktualisierungsregel der Geschwindigkeiten der Partikel in Abhängigkeit von der euklidischen Distanz des zu aktualisierenden Partikels zur derzeitig global besten Position.

$$\begin{aligned} &if(\|x_i(t-1) - gbest(t-1)\| \leq \varepsilon) \\ &\quad v_i^d(t) \leftarrow rand_i^d(-V_{max}^d, V_{max}^d) \\ &else \\ &\quad v_i^d(t) \leftarrow \gamma \cdot r_i^d \cdot (gbest^d(t-1) - x_i^d(t-1)) \end{aligned} \tag{2.15}$$

Ist die euklidische Distanz zwischen dem betrachteten Partikel und der momentan global besten Position kleiner als der Approximationskoeffizient ε hat sich das Partikel also nah an das Optima angenähert, so wird die Geschwindigkeit im für die einzelnen Dimensionen zulässigen Bereich reinitialisiert. Dabei sind $rand_i^d(-V_{max}^d, V_{max}^d)$ und r_i^d gleichverteilte Zufallsvariablen im Bereich $[-V_{max}^d, V_{max}^d]$ bzw. $[0, 1]$ und werden in jeder Iteration für jedes Partikel i und für jede Dimension d separat bestimmt.

Zusätzlich wird nach jeder Iteration eine reaktive und dynamische Anpassung der Schrittweite γ in Abhängigkeit vom Erfolg der vorherigen Iteration durchgeführt. Große Werte dieser Schrittweite führen dazu, dass die Partikel sich schnell auf das gefundene Optimum zubewegen oder dieses überfliegen, was zu einer Oszillation um selbiges

führen kann. Kleine Werte verringern einerseits die Konvergenzgeschwindigkeit, ermöglichen aber andererseits eine genauere Annäherung an das Optimum. Die Schrittweite wird nach jeder Iteration nach der folgenden Gleichung bestimmt:

$$\gamma \leftarrow \begin{cases} max(\gamma - \delta, \gamma_{min}) & , falls\ f(gbest(t)) > f(gbest(t-1)) \\ min(\gamma + \delta, \gamma_{max}) & , sonst. \end{cases} \tag{2.16}$$

Hat die vorherige Iteration Verbesserungen in Form einer höheren Fitness hervorbringen können, wird die Schrittweite um einen Wert δ verkleinert, um kleinere Schritte gehen und sich somit dem Optimum besser nähern zu können. Anderenfalls wird die Schrittweite erhöht, um durch Oszillationen um das gefundene Optimum bessere Positionen ausfindig machen zu können. Die Schrittweite bewegt sich dabei innerhalb fest definierter Grenzen γ_{min} und γ_{max}. Eine sinnvolle Variablenbelegung (PB06) setzt sich wie folgt zusammen: $\varepsilon = 10^{-8}, \delta = 0.5$ sowie $\gamma_{min} = 2$ und $\gamma_{max} = 4$.

Der Gregarious Particle Swarm Optimizer nutzt die Vorteile des ursprünglich vorgestellten GBest-Modells, versucht jedoch durch einen zusätzlichen Mechanismus zur Reinitialisierung der Partikel und einer dynamischen Schrittweitenanpassung dessen Nachteile zu verhindern.

2.3 Vergleich GA und PSO

Genetische Algorithmen sind hauptsächlich populär aufgrund sowohl ihrer Parallelität der Suche als auch aufgrund ihrer Fähigkeit, nichtlineare, multimodale Probleme zu lösen. Sie können dabei für diskrete und kontinuierliche Variablen verwendet werden und kommen ohne Informationen über die Gradienten der zu optimierenden Funktion aus. Im Vergleich dazu ist die Particle Swarm Optimization bekannt für ihre einfache Implementation, ihren geringen Rechenaufwand und ihre schnelle Konvergenz in optimale Regionen des Suchraums. Obwohl sie die beste Leistung bei der Optimierung von kontinuierlichen Variablen erbringt, kann sie ebenso mit kleinen Veränderungen dazu verwendet werden, diskrete Variablen zu optimieren. Wie bei den Genetischen Algorithmen werden bei der Particle Swarm Optimization keine Gradienteninformationen benötigt.

Beide Verfahren realisieren eine zufallsgesteuerte Suche, da deren Operatoren stochastischen Regeln folgen. Sie basieren beide auf dem Populationskonzept und verfolgen den Ansatz der generativen Vererbung. Das Pendant der Individuen der Genetischen Algorithmen sind bei der PSO die Partikel - beide repräsentieren eine potentielle Lösung des zu optimierenden Problems. Als Analogon des bedeutendsten Operators der Genetischen Algorithmen, der Rekombination, wird bei der Particle Swarm Optimization die Einbeziehung von einem oder mehreren Partikelpositionen in die Berechnung der Geschwindigkeiten, und daraus resultierend der neuen Positionen, gesehen (ES98). Die

Rekombination kann die Nachkommen mehrerer Elternindividuen zu Beginn der Suche, wenn alle Individuen noch weit voneinander entfernt sind, weit entfernt im Suchraum erzeugen und somit eine hohe Diversität garantieren. Gegen Ende des Optimierungsprozesses befinden sich die Individuen tendenziell näher beieinander wodurch der Effekt der Rekombination dann weniger stark ausfällt (ES98). Während GA die Mutationen der Individuen durch stochastisches Wechseln der Bits bzw. durch die Addition einer Zufallszahl zu einigen wenigen Dimensionen realisiert, wird bei der PSO bei der Berechnung der neuen Positionen jede Einbeziehung von anderen Positionen mit einer Zufallszahl gewichtet. Dies geschieht jeweils für alle Dimensionen. Genetische Algorithmen verwenden an vielen Stellen die Selektion zur Bestimmung von besonders erfolgreichen Individuen für unterschiedliche Zwecke. Bei der Verwendung und Einbeziehung der Position des besten Partikels in die Berechnung der Folgepositionen aller anderen Partikel (abhängig vom gewählten Nachbarschaftsmodell) findet auch bei der PSO eine Art der Selektion statt. Es werden allerdings keine Individuen mit schlechter Fitness verworfen, wie es bei den GA der Fall ist - alle Partikel bleiben bis zum Ende der Optimierung erhalten.

Viele Arbeiten haben sich in den vergangenen Jahren mit dem Vergleich beider Optimierungstechniken bezüglich ihrer Effektivität und Effizienz beschäftigt. Hodgson (Hod02) verglich sie durch Anwendung auf das Problem der Optimierung atomarer Cluster. Die Aufgabe bestand darin, eine hochgradig multimodale Energiefunktion der atomaren Cluster zu minimieren. Seine Berechnungen stellten PSO gegenüber sowohl einem allgemeinen als auch einem speziell dafür entwickelten problemspezifischen GA als deutlich überlegen heraus. Clow und White (CW04) verglichen beide Verfahren für das Training künstlicher neuronaler Netze, welche zur Kontrolle von virtuellen Rennautos verwendet wurden. Aufgrund der Kontinuität der zu optimierenden neuronalen Gewichte, stellte sich die PSO für alle durchgeführten Tests gegenüber Genetischen Algorithmen als überlegen heraus. Sie erzielte eine höhere und schneller wachsende mittlere Fitness. Hassan, Cohanim und De Weck (HCW05) verglichen beide Techniken mit einem Satz von acht bekannten Optimierungsfunktionen und kamen zum Schluss, dass die PSO im Allgemeinen genauso effektiv, dafür aber effizienter ist. Die Überlegenheit der PSO stellte sich jedoch als problemspezifisch heraus. Bei der Optimierung von unbegrenzten Problemen mit kontinuierlichen Variablen war der Unterschied der Effizienz größer als bei der Optimierung von begrenzten Problemen mit kontinuierlichen oder diskreten Variablen. Jones (Jon05) verwendete beide Strategien zur Bestimmung der Parameter zweier unterschiedlicher mathematischer Modelle. Obwohl er anmerkte, dass beide Verfahren gleich effektiv waren, kam er zum Schluss, dass die GA der PSO bezüglich der Effizienz überlegen war. Allerdings muss dabei beachtet werden, dass die von Jones verwendete Variante der PSO während des Aktualisierungsschritts der Partikelgeschwindigkeiten die gleichen Zufallsvariablen für alle Dimensionen verwendet, was sich im Vergleich zur Verwendung von unterschiedlichen Zufallsvariablen, wie ursprünglich vorgeschlagen (siehe Gleichung 2.4), als unvorteilhaft für schwierige Optimierungsprobleme herausgestellt hat. Möglicherweise ist das der Grund für die von Jones vorgestellten schlechten Resulta-

te der PSO. Horák, Chmela, Oliva und Raida (JHR06) analysierten die Fähigkeiten der PSO und der GA, ebene Dual-Band-Antennen für mobile Kommunikationsanwendungen zu optimieren. Sie stellten fest, dass PSO einerseits geringfügig bessere Ergebnisse liefern konnte als GA, dafür aber andererseits auch mehr Rechenzeit benötigte.

Kapitel 3
Strukturorientierte Testfallermittlung

Softwaresysteme sind in der Regel in Module unterteilt. Module sind dabei beispielsweise Methoden, Funktionen oder auch einfache Anweisungssequenzen, welche separat angesprochen werden können und autarke Teile des Gesamtsystems darstellen. Zur Qualitätssicherung dieser Module müssen sie getestet werden. Dieser Test wird Modultest (engl. *unit test*) genannt, das zu testende Modul wird dabei auch als *Testobjekt* bezeichnet. Testverfahren, auch Tests genannt, lassen sich grob in statische und dynamische Tests unterteilen.

Statische Testverfahren verwenden zur Untersuchung des Testobjektes lediglich den Quellcode bzw. die Spezifikation des selbigen. Alle statischen Tests haben die Eigenschaften, das zu analysierende Testobjekt nicht auszuführen, keine Eingabewerte (Testfälle) zu bestimmen und die Korrektheit des selbigen nicht beweisen zu können (Lig90). Zu den statischen Tests gehört die Verwendung von Software-Metriken und die Datenflussanomalieanalyse. Software-Metriken sind Vergleichskriterien zur Quantifizierung des Software-Produkts. Beispiele dafür wären die Anzahl der Codezeilen (Lines Of Code) oder die zyklomatische Komplexität. Die Datenflussanomalieanalyse untersucht die Zugriffssequenzen einzelner Variablen des Testobjekts und kann dadurch von einer Norm abweichende Programmstrukturen identifizieren.

Dynamische Tests hingegen führen das Testobjekt zur Identifikation von Fehlern direkt in der realen Umgebung mit zulässigen Eingabewerten (Testdaten) aus. Bei dynamischen Testverfahren handelt es sich um Stichprobenverfahren weshalb sie die Korrektheit eines Software-Produktes nicht beweisen können. Zu bestimmen ist ein Satz von Eingabedaten, auch *Testfälle* genannt, welche das Testobjekt komplett zur Ausführung bringen. Von einem *dynamischen Strukturtest* spricht man, wenn für den dynamischen Test zur Generierung der Testdaten Informationen des Kontroll- oder Datenflusses des zu untersuchenden Testobjekts genutzt werden. Dabei können diverse Überdeckungskriterien verwendet werden, welche einen direkten Einfluss auf den Aufwand des dynamischen Strukturtests haben. Auf datenflussorientierte Verfahren wird im Rahmen dieser Arbeit nicht näher eingegangen; diese sind beispielsweise in (Lig90) ausführlich beschrieben.

In den folgenden Abschnitten sollen die kontrollflussorientierten Strukturtestverfahren

vorgestellt und kurz erläutert werden. In Abschnitt 3.1 werden dazu zuerst die zur Definition der Kriterien des Strukturtests verwendeten strukturellen Bestandteile und der Aufbau des Kontrollflussgraphen beschrieben. Anschließend werden drei wichtige kontrollflussorientierten Verfahren charakterisiert. Diese Ausführungen sollen nur einen Einblick in die Materie verschaffen, detailliertere Informationen liefert auch diesbezüglich Liggesmeyer (Lig90). Abschnitt 3.2 geht anschließend näher auf den Evolutionären Strukturtest und die Konstruktion der für die Optimierungen notwendigen *Zielfunktionen* ein.

3.1 Kontrollflussorientierte Verfahren

Die kontrollfluss- oder kontrollstrukturorientierten Testverfahren beruhen auf dem Kontrollflussgraphen des zu testenden Moduls. Zu erreichende Teilziele lassen sich mittels Strukturelementen, wie zum Beispiel mit Anweisungen, Zweigen oder Bedingungen definieren, woraus sich die Überdeckungskriterien der Anweisungs-, Zweig- und Bedingungsüberdeckung ergeben. Diese beabsichtigen die Generierung von Testfällen, welche die Ausführung aller Anweisungen, aller Bedingungen bzw. aller Zweige des korrespondierenden Kontrollflussgraphen bewirken. Häufig ist in der Praxis eine hunderprozentige Überdeckung nicht möglich, da unerreichbare Zweige existieren können. Aus diesem Grund ist eine Maximierung des Überdeckungsgrades angestrebt.

Im Folgenden soll zunächst auf den Kontrollflussgraphen und anschließend auf drei unterschiedliche kontrollflussorientierte Überdeckungskriterien eingegangen werden.

3.1.1 Kontrollflussgraph

Der Kontrollflussgraph eignet sich zur Darstellung des Kontrollflusses eines Moduls. Dieser ist ein gerichteter Graph $G(N, E, n_{Start}, n_{Ende})$ und besteht sowohl aus einer endlichen Menge von *Knoten* N, einer endlichen Menge an gerichteten *Kanten* E jeweils zwischen zwei Knoten, welche eine Teilmenge von $N \times N$ darstellt, als auch aus einem *Startknoten* n_{Start} und einem *Endknoten* n_{Ende} mit $n_{Start}, n_{Ende} \in N$.

Jede Anweisung oder Bedingung des Moduls korrespondiert mit einem Knoten des Kontrollflussgraphen. Mehrere unbedingte Anweisungen können dabei einem Knoten zugewiesen werden. Der *Kontrollfluss* zwischen den Knoten i und j, d.h. der Übergang von der letzten Anweisung des Knotens i zur ersten Anweisung des Knotens j, wird im Kontrollflussgraphen mithilfe der gerichteten Kante zwischen Knoten i und Knoten j repräsentiert und als *Zweig* bezeichnet. Ein *Pfad* innerhalb des Graphen ist eine Sequenz aus Knoten und Kanten vom Startknoten zum Endknoten. Pfade gelten als *ausführbar* und die korrespondierenden Teilziele als *erreichbar*, wenn Eingabedaten existieren, für

welche das Testobjekt diesen Pfad bei Ausführung durchläuft. Abbildung 3.1 zeigt ein Beispieltestobjekt und dessen Kontrollflussgraphen.

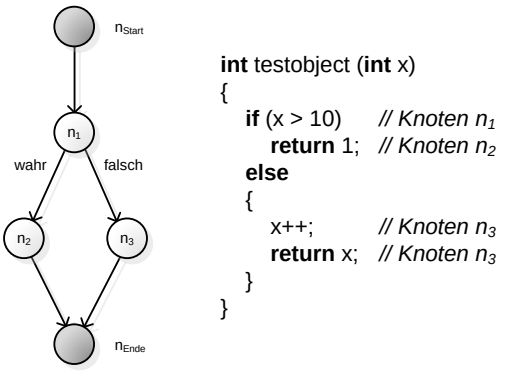

Abbildung 3.1: Der Kontrollflussgraph eines Beispieltestobjektes

Der Einfachheit halber werden im Folgenden einige Redewendungen verwendet, obwohl sie formal so nicht verwendet werden könnten: Nähert sich beispielsweise ein Testfall einem Teilziel an, bedeutet dies, dass sich der Kontrollfluss, der aus der Ausführung des Testobjektes mit diesem geänderten Testfall resultiert, nun näher an der auszuführenden Kontrollstruktur, d.h. der Anweisung, dem Knoten oder dem Zweig befindet. Dies gilt analog für den Abstand eines Testfalls zu einer zu erreichenden Kontrollstruktur. Ebenso können Knoten formal nicht ausgeführt werden, dennoch steht die Ausführung eines Knotens synonym für die Generierung eines Testfalls, der bei Ausführung des Testobjektes zum Durchlaufen des Knotens führt.

3.1.2 Anweisungsüberdeckungstest

Der *Anweisungsüberdeckungstest* (auch C_0-*Test* genannt oder engl. *statement coverage test*) fordert die Ausführung aller Anweisungen des Testobjektes. Es werden also Testfälle gesucht, welche die vollständige Überdeckung aller Anweisungen zur Folge haben. Für jede bestimmte Testdatenmenge kann ein Überdeckungsgrad C_0 angegeben werden, welcher ein Maß für deren erreichte Überdeckung ist. Dieser ist definiert als der Quotient aus der Anzahl der überdeckten und der Anzahl der insgesamt vorhandenen Anweisungen:

$$C_0 = \frac{\text{Anzahl der überdeckten Anweisungen}}{\text{Anzahl aller Anweisungen}} \tag{3.1}$$

3.1.3 Zweigüberdeckungstest

Der *Zweigüberdeckungstest* (auch C_1-*Test* genannt oder engl. *branch coverage test*) fordert die Ausführung aller Zweige des Testobjekts. Gesucht werden Testfälle, welche für den korrespondierenden Kontrollflussgraphen Pfade generieren, in denen alle Zweige mindestens einmal enthalten sind. Der Überdeckungsgrad der Testdatenmengen kann als der Quotient der überdeckten und der Anzahl aller Zweige berechnet werden:

$$C_1 = \frac{\text{Anzahl der überdeckten Zweige}}{\text{Anzahl aller Zweige}} \tag{3.2}$$

3.1.4 Bedingungsüberdeckungstest

Der *Bedingungsüberdeckungstest* (engl. *condition coverage test*) beachtet die logische Struktur von Bedingungen. So sollen Testdaten ermittelt werden, die je nach gewähltem Kriterium beispielsweise einzelne Teile der hierarchisch aufgebauten Bedingungen mit den beiden möglichen Zuständen *wahr* und $falsch$ ausführen können. Die wichtigsten Kriterien des Bedingungsüberdeckungstests sind die *einfache Bedingungsüberdeckung*, die *Mehrfach-Bedingungsüberdeckung* und die *minimale Mehrfach-Bedingungsüberdeckung*.

Einfache Bedingungsüberdeckung

Die *einfache Bedingungsüberdeckung* (auch C_2-*Überdeckung* genannt oder engl. *simple condition coverage*) ist das einfachste Kriterium der Bedingungsüberdeckung. Es sind Testfälle zu generieren, welche alle atomaren Bedingungen jeweils mit *wahr* und mit $falsch$ auswerten. Der Überdeckungsgrad der generierten Testdatenmengen kann als Quotient aus der Anzahl der mit *wahr* und mit $falsch$ ausgewerteten Bedingungen und der doppelten Anzahl aller Bedingungen berechnet werden:

$$C_2 = \frac{\text{Anzahl der mit } wahr \text{ und } falsch \text{ ausgewerteten Bedingungen}}{2 \cdot \text{Anzahl aller Bedingungen}} \tag{3.3}$$

Mehrfach-Bedingungsüberdeckung

Die *Mehrfach-Bedingungsüberdeckung* (auch C_3-*Überdeckung* genannt oder engl. *multiple condition coverage*) stellt das umfangreichste Kriterium der Bedingungsüberdeckung dar. Die zu generierenden Testfälle sollen alle Kombinationsmöglichkeiten der Wahrheitswerte der atomaren Bedingungen aller Verzweigungsbedingungen ausführen. Bei n atomaren Bedingungen sind demzufolge 2^n Testfälle zu erzeugen. Der Überdeckungsgrad der generierten Testdatenmengen kann als Quotient aus der Anzahl der ausgeführten

atomaren Bedingungskombinationen und der Gesamtanzahl aller Bedingungskombinationen berechnet werden:

$$C_3 = \frac{\text{Anzahl der ausgeführten atomaren Bedingungskombinationen}}{\text{Anzahl aller atomaren Bedingungskombinationen}} \tag{3.4}$$

Minimale Mehrfach-Bedingungsüberdeckung

Die *minimale Mehrfach-Bedingungsüberdeckung* (auch $C_2(mM)$-*Überdeckung* oder engl. *minimal multiple condition coverage*) ist der Mehrfach-Bedingungsüberdeckung sehr ähnlich, sie fordert im Gegensatz dazu jedoch nur die Erfüllung jener Kombinationen von atomaren Bedingungen, welche zu einer Wertänderung der gesamten Verzweigungsbedingung führen können. Der Überdeckungsgrad der generierten Testdatenmengen lässt sich demzufolge als Quotient aus der Anzahl der dadurch ausgeführten minimalen atomaren Bedingungskombinationen und der Anzahl aller minimalen Bedingungskombinationen berechnen:

$$C_2(mM) = \frac{\text{Anzahl der ausgeführten minimalen atomaren Bedingungskombinationen}}{\text{Anzahl aller minimalen atomaren Bedingungskombinationen}} \tag{3.5}$$

3.2 Evolutionärer Strukturtest

Beim *Evolutionären Strukturtest* (JSE96; PHP99; WBS01) wird die Aufgabe der Bestimmung relevanter Testfälle, welche ein vorgegebenes Überdeckungskriterium erfüllen, als Optimierungsproblem aufgefasst. Dieses Optimierungsproblem wird dann versucht, von metaheuristischen Suchverfahren, hier von Evolutionären Algorithmen (vgl. Kapitel 2), zu lösen.

Das Testobjekt wird dazu in Teilziele unterteilt, welche aus den jeweiligen Strukturtestkriterien hervorgehen. Deren Ausführung wird zur vollständigen Überdeckung des Testobjektes benötigt. Beispielsweise repräsentiert für die Zweigüberdeckung jeder zu erreichende Zweig des Kontrollflussgraphen des Testobjektes ein Teilziel, welches es zu erreichen gilt. Die Teilziele werden einzeln als Optimierungsproblem aufgefasst und von den Evolutionären Algorithmen optimiert, d.h. es wird für jedes Teilziel ein Testfall generiert, welcher dieses erfüllt. Im Gesamtprozess der Optimierung wird ein Satz von Testfällen generiert, welcher als Ganzes das Testobjekt gemäß des Überdeckungskriteriums vollständig überdecken soll. Der Optimierungsprozess eines dieser Teilziele ist in Abbildung 3.2 dargestellt. Zur Optimierung wird dabei ein Evolutionärer Algorithmus

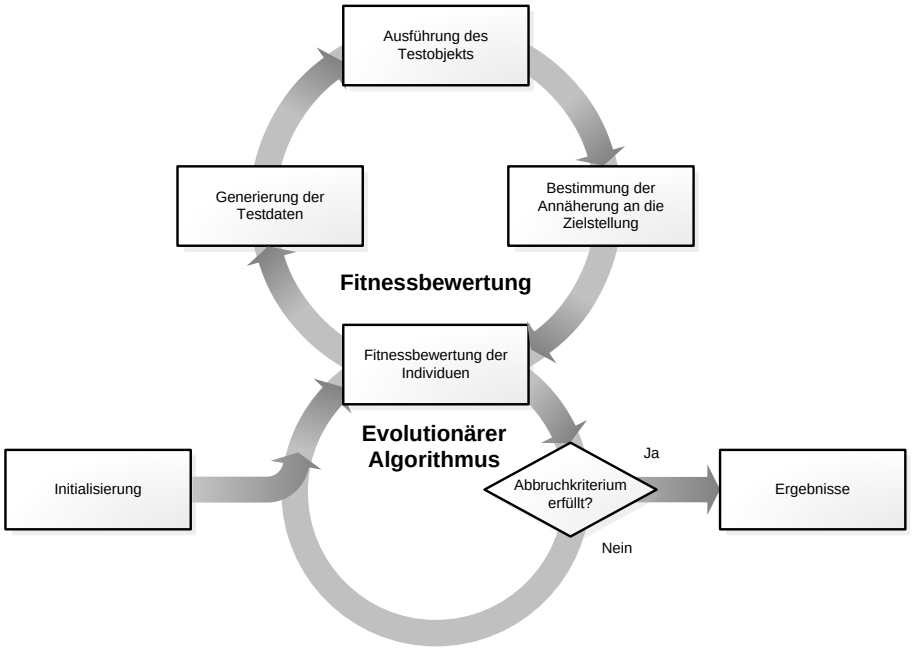

Abbildung 3.2: Allgemeiner Ablauf des Optimierungsprozesses für ein Teilziel im Rahmen des Evolutionären Strukturtests

verwendet, welcher potentielle Lösungen des Optimierungsproblems erstellen und die Bewertungen dieser Lösungen entsprechend berücksichtigen kann. Typischerweise kommen hier GAs zum Einsatz. Die Bewertung der erzeugten Testfälle, d.h. die Zuordnung eines Zielfunktionswertes, erfolgt anhand ihrer Annäherung an das zu erreichende Teilziel. Dafür muss für jedes Teilziel eine Zielfunktion entwickelt werden, deren globales Optimum zur Erreichung dieses Teilziels führt. Diese liefert für jeden Testfall einen numerischen Wert, welcher sich aus der *Abstandsstufe* und der *Abstandsfunktion* ergibt. Dabei hat sich der Ansatz von Baresel et. al. (BS03) durchgesetzt. Die genannten Metriken sollen im Folgenden kurz erläutert werden.

3.2.1 Zielfunktion

Die Zielfunktion eines zu erreichenden Teilziels wird basierend auf der erreichten Abstandsstufe und der Abstandsfunktion, welche sich aus logisch verknüpften lokalen Abstandsfunktionen bilden kann, wie folgt berechnet:

$$\text{Zielfunktion} := \text{Abstandsstufe} + \text{Abstandsfunktion} \tag{3.6}$$

Da der global optimale Funktionswert sowohl der Abstandsstufe als auch der Abstandsfunktion dem Wert 0 entspricht, liefert auch die Zielfunktion den optimalen Wert 0, welcher das Erreichen des Teilziels repräsentiert.

3.2.2 Abstandsstufen

Die Abstandsstufen beschreiben den Abstand von den generierten Testdaten zur durch das Teilziel definierten und zu erreichenden Kontrollstruktur. Die Abstandsstufen entsprechen einem ganzzahligen Wert größer oder gleich Null.

Die Unterstützung aller gängigen kontroll- und datenflussorientierten Testmethoden verlangt eine spezifische Berechnung der Abstandsstufen für knotenorientierte, pfadorientierte, knoten-pfadorientierte und knoten-knotenorientierte Testverfahren. Im Folgenden soll nur auf die Berechnung der Abstandsstufen für knotenorientierte Testverfahren eingegangen werden, da diese für die Unterstützung der kontrollflussorientierten Überdeckungskriterien der Anweisungs-, Zweig- und der Bedingungsüberdeckung anzuwenden sind. Die restlichen Berechnungen wurden von Baresel (Bar00) ausführlich beschrieben.

Bei knotenorientierten Testverfahren besteht ein Teilziel aus der Ausführung eines bestimmten Knotens. Die Berechnung der Abstandsstufe hängt in diesem Fall wie folgt von der Annäherung an diesen Knoten ab. Die Unterteilung in die verschiedenen Abstandsstufen basiert auf so genannten *entscheidenden* Verzweigungsknoten. Diese besitzen Zweige, deren Ausführung das Erreichen des Zielknotens unmöglich machen. Wie in Abbildung 3.3 illustriert, entstehen aufgrund dessen verschiedene Distanzen zum gewünschten Teilziel. Je näher die entscheidende Verzweigung dem gesuchten Teilziel liegt,

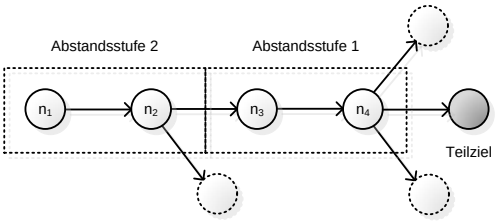

Abbildung 3.3: Abstandsstufen zum Teilziel eines beispielhaften Kontrollflussgraphen

desto geringer ist diese Distanz. Für jeden Testfall, der das Teilziel nicht erreicht, lässt sich bestimmen, an welcher der entscheidenden Verzweigungen ein unerwünschter Zweig ausgeführt wurde. Die Abstandsstufe dieser Verzweigung wird als Ergebnis für den untersuchten Testfall zurückgegeben. Hat der Testfall das Teilziel erreicht, so wird die Abstandsstufe 0 übergeben.

3.2.3 Abstandsfunktion

Die Abstandsstufen reichen als alleinige Bewertungskriterien für eine Optimierung nicht aus, da unterschiedliche Testdaten denselben Fitnesswert erhalten, wenn sie denselben Pfad des Testobjektes durchlaufen und an der selben entscheidenen Verzweigung scheitern. Der Einfluss der Testdaten auf die Erfüllung dieser entscheidenden Verzweigung

wird so nicht beachtet. Es wird also eine zusätzliche und detaillierte Bewertung benötigt. Dies wird durch die Abstandsfunktion Φ realisiert, welche die Bewertung von Testdaten für das Erfüllen von Verzweigungsbedingungen übernimmt, wodurch die Generierung von Testfällen, die die Ausführung dieser Verzweigungsbedingung ermöglichen, gefördert wird. Auch hier gibt der Wert der Abstandsfunktion die Distanz der Testfälle zum Erfüllen der untersuchten Bedingungen wieder. Wird diese erfüllt, so wird die Distanz 0 zurückgegeben.

Besteht eine Verzweigungsbedingung aus mehreren atomaren Bedingungen, so wird die Distanz zu diesen separat berechnet (*lokale Abstandsfunktion*) und anschließend zu einer gesamten Distanz (Abstandsfunktion) zusammengefasst. Da mithilfe der lokalen Abstandsfunktion Testdaten generiert werden sollen, die die untersuchte Bedingung für beide möglichen Wahrheitswerte *wahr* und $falsch$ ausführen, muss für den jeweils gewünschten Wert eine spezielle Abstandsfunktion erzeugt werden.

Lokale Abstandsfunktion für atomare Bedingungen

Die zu erfüllenden Verzweigungsbedingungen können aus zusammengesetzten atomaren Bedingungen bestehen, welche separat betrachtet werden müssen. In Abhängigkeit der verwendeten Relation werden die Distanzwerte nach Baresel (Bar00) und Lammermann (Lam05) wie folgt berechnet:

Für die Äquivalenzrelation ($A = B$) wird das Optimum der Abstandsfunktion $\Phi_{A=B}$ genau dann erreicht, wenn gilt $A = B$. Für alle anderen Punkte ergibt sich entsprechend der Distanz beider Parameter ein Distanzwert im Intervall $[0; 1[$. Die sich daraus ergebende Zielfunktionslandschaft ist in Abbildung 3.4 dargestellt.

$$\Phi_{A=B}(A, B) = 1 - (1 - \varepsilon)^{|A-B|}, \text{ mit } 0 < \varepsilon < 1 \tag{3.7}$$

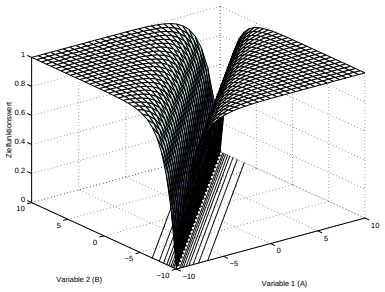

Abbildung 3.4: Funktionslandschaften der lokalen Abstandsfunktion für die Relation $A = B$

Das Optimum der Abstandsfunktion für die Ungleichrelation ($A \neq B$) wird erreicht, wenn die beiden Parameter nicht übereinstimmen. Stimmen sie überein, so wird der

Distanzwert 1 zurückgegeben. Dies führt zu einer wie in Abbildung 3.5 dargestellten, für die Optimierung sehr ungünstigen Funktionslandschaft, welche keinerlei Steigungen oder andere in Richtung des Optimums weisenden Hilfsmittel bereit stellt (Entstehung von Plateaus innerhalb der Funktionslandschaft).

$$\Phi_{A \neq B}(A, B) = \begin{cases} 0 & falls\ A - B \neq 0 \\ 1 & sonst \end{cases} \tag{3.8}$$

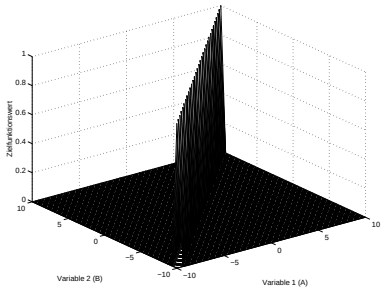

Abbildung 3.5: Funktionslandschaften der lokalen Abstandsfunktion für die Relation $A \neq B$

Die Größergleichrelation ($A \geq B$) gleicht für den Fall, dass $A < B$ ist der Äquivalenzrelation. Anderenfalls liefert sie den optimalen Distanzwert 0. Abbildung 3.6 zeigt die resultierende Zielfunktionslandschaft.

$$\Phi_{A \geq B}(A, B) = \begin{cases} 1 - (1 - \varepsilon)^{|A-B|} & falls\ A - B < 0 \\ 0 & sonst \end{cases}, \text{ mit } 0 < \varepsilon < 1 \tag{3.9}$$

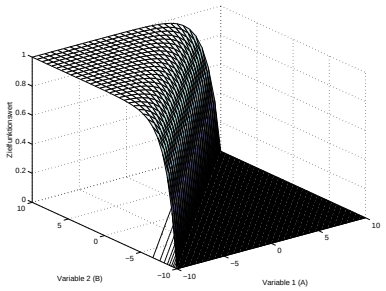

Abbildung 3.6: Funktionslandschaften der lokalen Abstandsfunktion für die Relation $A \geq B$

Die Kleinergleichrelation ($A \leq B$) berechnet sich ähnlich der Größergleichrelation, deren Zielfunktionslandschaft zeigt Abbildung 3.7.

$$\Phi_{A \leq B}(A, B) = \begin{cases} 1 - (1 - \varepsilon)^{|B-A|} & falls\ A - B > 0 \\ 0 & sonst \end{cases}, \text{ mit } 0 < \varepsilon < 1 \tag{3.10}$$

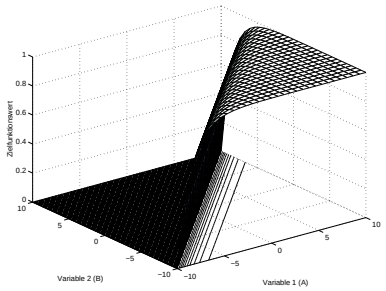

Abbildung 3.7: Funktionslandschaften der lokalen Abstandsfunktion für die Relation $A \leq B$

Die Größerrelation ($A > B$) benötigt die Einführung eines zusätzlichen Parameters κ, mit $0 < \kappa << 1$, welcher die deutlichere Unterscheidung der Zielfunktionswerte der Stelle $A = B$ von der kleinsten Abweichung selbiger in die gewünschte Richtung ermöglicht. An der Stelle $A = B$ liefert diese Funktion den Wert κ zurück und bei der kleinsten Abweichung in die gewünschte Richtung wird der optimale Funktionswert 0 übergeben.

$$\Phi_{A>B}(A,B) = \begin{cases} 1-(1-\varepsilon)^{|A-B|} \cdot (1-\kappa) & falls\ A-B \leq 0 \\ 0 & sonst \end{cases}, \text{ mit } 0 < \varepsilon < 1 \quad (3.11)$$

Die Kleinerrelation ($A < B$) berechnet sich ähnlich der Größerrelation. Die Zielfunktionslandschaften der Größer- und Kleinerrelationen ähneln den der Größergleich- bzw. Kleinergleichrelationen und sind deshalb hier nicht gesondert dargestellt.

$$\Phi_{A<B}(A,B) = \begin{cases} 1-(1-\varepsilon)^{|B-A|} \cdot (1-\kappa) & falls\ A-B \geq 0 \\ 0 & sonst \end{cases}, \text{ mit } 0 < \varepsilon < 1 \quad (3.12)$$

Die logische Negation einer Relation kann nun mit der Verwendung deren komplementären Relation realisiert werden. Die lokale Abstandsfunktion berechnet sich in diesem Fall wie beschrieben.

Lokale Abstandsfunktion für verknüpfte Bedingungen

Eine Verzweigungsbedingung kann aus mehreren logisch miteinander verknüpften atomaren Relationen bestehen. Diese Verknüpfungen müssen ebenfalls in die Berechnung der Abstandsfunktion einbezogen werden.

Die logische *UND-Verknüpfung* wird immer dann zu *wahr* ausgewertet, wenn beide Operanden den Wert *wahr* ergeben. Dies lässt sich im Sinne der Berechnung der Abstandsfunktion durch das arithmetische Mittel beider Operanden beschreiben. Abbildung 3.8 zeigt beispielhaft die Zielfunktionslandschaft zweier UND-verknüpfter Gleichheitsrelationen.

$$\Phi_{UND}(A,B) = \frac{A+B}{2} \quad (3.13)$$

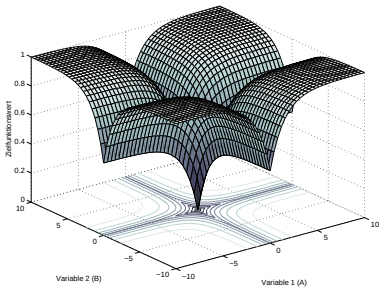

Abbildung 3.8: Funktionslandschaft zweier UND-verknüpfter Gleichheitsrelationen

Demgegenüber wird die logische *ODER-Verknüpfung* immer dann zu *wahr* ausgewertet, wenn mindestens einer der beiden Operanden den Wert *wahr* ergibt. Für die Berechnung der Abstandsfunktion kann dies durch das Minimum beider Operanden beschrieben werden. In Abbildung 3.9 ist die Funktionslandschaft zweier ODER-verknüpfter Gleichheitsrelationen dargestellt.

$$\Phi_{ODER}(A, B) = \min(A, B) \tag{3.14}$$

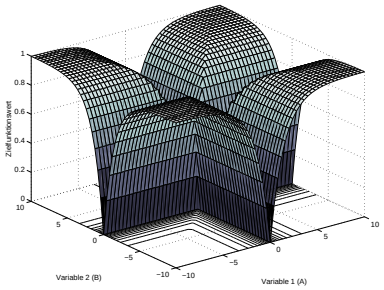

Abbildung 3.9: Funktionslandschaft zweier ODER-verknüpfter Gleichheitsrelationen

Kapitel 4

Implementierung

Dieses Kapitel beschreibt das von DaimlerChrysler entwickelte Testsystem *ETS* (*Evolutionary Testing System*). In Abschnitt 4.1 wird zuerst auf das bereits bestehende System sowie dessen Aufbau und Funktionsweise eingegangen. Dabei werden die darin enthaltenen Komponenten vorgestellt und beschrieben.

Ferner erläutert Abschnitt 4.2 die im Rahmen dieser Arbeit entstandenen Erweiterungen des Systems, bestehend aus einer Optimierungskomponente inklusive Kommunikationsserver zur Umsetzung der Particle Swarm Optimization. Des Weiteren wird auf eine Verifizierung der Implementierung eingegangen und deren Ergebnisse zur Bewertung der verwendeten PSO-Varianten verwendet.

4.1 Beschreibung des bestehenden Systems

Im Laufe der letzten Jahre ist von DaimlerChrysler ein Testsystem entwickelt worden, welches die Konzepte des evolutionären Strukturtests implementiert (WBS01). Das System führt für den als Eingabe übergebenen Quellcode für eine darin enthaltene Funktion (diese wird auch *Testobjekt* genannt) eine automatische Testdatengenerierung auf Basis von evolutionären Algorithmen durch. Die Testdatengenerierung erfolgt dabei strukturorientiert, welches Überdeckungskriterium dafür verwendet werden soll, kann vom Benutzer spezifiziert werden. Zur Verfügung stehen die Kriterien der Anweisungs-, der Zweig- und der Bedingungsüberdeckung. Als Ergebnis wird die Menge der gefundenen Testdaten, sowie diverse Informationen über den Testverlauf, wie beispielsweise die erreichte Überdeckung, die Anzahl der benötigten Fitnessfunktionsevaluationen oder auch die Anzahl der verwendeten Individuen und Generationen, bereitgestellt.

Das System ist momentan zum Testen von Quellcodes der Programmiersprache C konzipiert. Die Erweiterung auf andere imperative Programmiersprachen ist möglich. So wird zur Zeit auch an der Unterstützung von objektorientierten Sprachen, wie beispielsweise JAVA, gearbeitet (WW06).

Die folgenden Abschnitte sollen einen Überblick über den Aufbau des Evolutionären Testsystems und den Ablauf des Evolutionären Strukturtests geben. Zusätzlich soll die Reichweite der vorgenommenen Erweiterungen verdeutlicht werden.

4.1.1 Aufbau und Funktionsweise des ETS

Das Evolutionäre Testsystem basiert auf einer Client-Server-Architektur, welche die Testspezifikation und die eigentliche Testdatengenerierung trennen und auf mehrere Rechner verteilen soll. So können die sehr rechenintensiven Optimierungen auf leistungsfähige Server oder einen Computercluster ausgelagert werden. Beide Seiten bestehen aus mehreren Softwarekomponenten, deren Synergie die Ausführung des evolutionären Strukturtests ermöglicht. Abbildung 4.1 zeigt eine Übersicht über die Gesamtarchitektur.

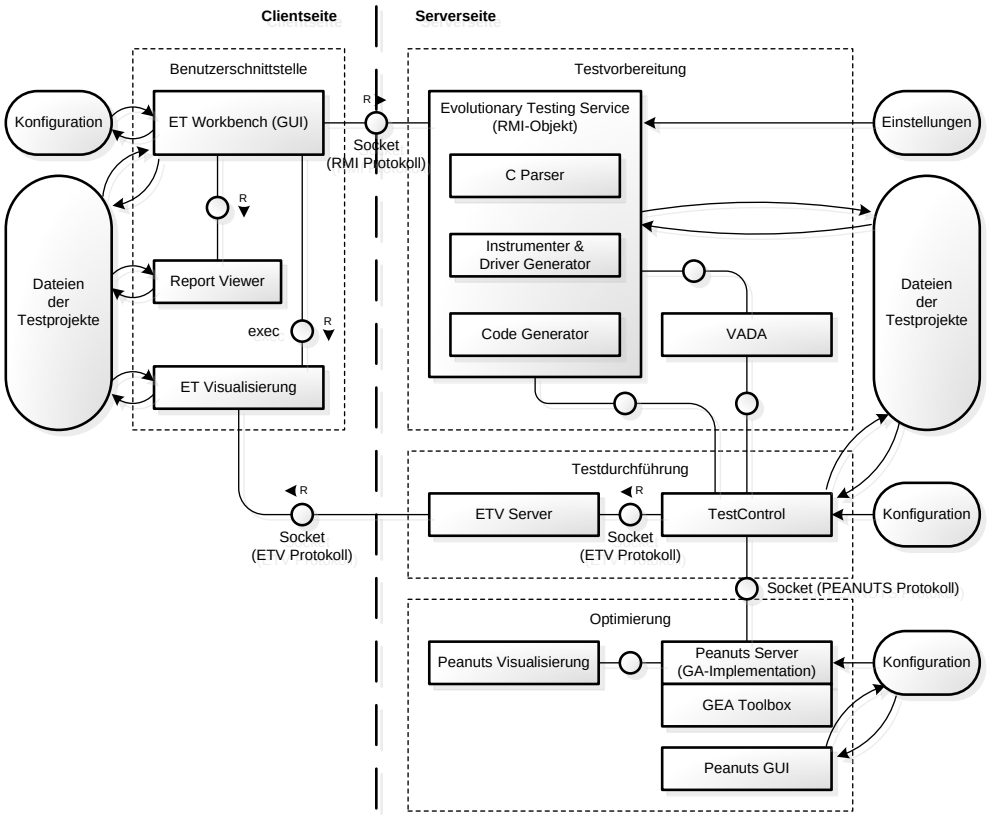

Abbildung 4.1: Der strukturelle Aufbau des evolutionären Testsystems

Das ETS kann grob in vier Bestandteile gegliedert werden: die *Benutzerschnittstelle*, die *Testvorbereitung*, die *Testdurchführung* und die *Optimierung*. Diese sollen im Folgenden mitsamt ihren Komponenten näher erläutert werden.

Benutzerschnittstelle

Auf der Client-Seite befindet sich lediglich die Benutzerschnittstelle, welche aus den Unterkomponenten ET (Evolutionary Testing) Workbench, Report Viewer und ET Visualisierung besteht. Die *ET Workbench* stellt die für den Anwender sichtbare, graphische Benutzeroberfläche (GUI) des Systems dar. Mit deren Hilfe können alle zu bearbeitenden Testobjekte in separaten Projekten initiiert, verwaltet und gesteuert werden. Die Testobjektsinitiierung, also die Erstellung eines neuen Testprojektes, bietet umfangreiche Konfigurationsmöglichkeiten. So kann beispielsweise das zu verfolgende Überdeckungskriterium ausgewählt werden. Diese Auswahl beschränkt sich im Moment auf die drei Strukturtestverfahren Anweisungs-, Zweig- sowie die Bedingsungsüberdeckung. Ebenso werden hier die genauen Spezifikationen des zu testenden Quellcodes, der die Hauptquelldatei, alle benötigten Header-Dateien und die abhängigen Module umfasst, festgelegt. Dazu gehört sowohl die Auswahl der zu testenden Funktion des Quellcodes als auch die Modifikation dessen Schnittstelle. Diese wird vom Parser des Evolutionary Testing Service ermittelt und kann anschließend vom Benutzer modifiziert werden. So können beispielsweise die Wertebereiche der Eingabeparameter verändert oder auch auf feste Werte gesetzt werden. Die Workbench legt für jedes Testprojekt einen eigenen Ordner im Dateisystem an, in welchem alle dazugehörigen Daten abgelegt sind. Diese Dateien werden bei der Erstellung von neuen Projekten ebenfalls an den Testservice übermittelt.

Die *ET Visualisierung* ist eine Visualisierungskomponente, die den Kontrollflussgraphen der zu testenden Funktion und die mit dem Optimierungsprozess fortschreitende Überdeckung dessen Anweisungen, Zweige oder Bedingungen (je nach gewähltem Überdeckungskriterium) grafisch darstellt.

Der *Report Viewer* präsentiert den Testreport eines Testobjekts, für den die Testfallgenerierung bereits abgeschlossen ist. Dieser enthält Informationen über den Erfolg der Optimierung. Es wird neben der Anzahl der insgesamt untersuchten, der davon erreichten und nicht erreichten Teilziele auch die prozentuale Überdeckung angezeigt. Er gibt ebenfalls Aufschluss über die für die Erreichung der einzelnen Testziele benötigten Fitnessfunktionsaufrufe.

Die ET Visualisierung sowie der Report Viewer greifen ebenfalls auf die von der Workbench verwalteten Dateien der Testprojekte zu, aus denen sie Informationen über das Testobjekt, wie beispielsweise Strukturinformationen oder auch die Ergebnisse der Testdurchläufe, erhalten.

Testvorbereitung

Die Testvorbereitung wird vom *Evolutionary Testing Service* und von der Variablenabhängigkeitsanalyse (*VADA*) durchgeführt. Der Testservice führt zuerst eine Syntaxanalyse (*Parser*) und eine Instrumentierung (*Instrumenter*) des während der Testprojekti-

nitiierung übermittelten Quelltextes durch. Dabei werden spezifische Funktionsaufrufe als Messpunkte an charakteristischen Stellen des Quelltextes eingefügt, welche später während der Testausführung Aufschluss über den Programmablauf geben sollen, der aus einem bestimmten Satz von Eingabeparametern resultiert. Anschließend wird vom *Code Generator* der benötigte Testtreiber erstellt. Dieser generiert eine Ausführungsumgebung des zu testenden Testobjekts, welche das Testobjekt mit den übermittelten Testdaten ausführt und die Messungen zurückliefert, die aus den während der Instrumentierung eingefügten Messpunkten resultieren.

Zusätzlich kann die Variablenabhängigkeitsanalyse verwendet werden, um Funktionsparameter, welche nicht zum Erreichen eines Teilziels benötigt werden, für den Optimierungsprozess auszuschließen. Dies verkleinert den Suchraum der zu optimierenden Zielfunktion und soll dadurch die Optimierung vereinfachen.

Testdurchführung

Die Testdurchführung wird von der *TestControl* realisiert, deren Verlauf wird vom *ETV-Server* zur Visualisierung protokolliert. Die TestControl steuert die Ausführung des Testobjektes in Abhängigkeit vom gewählten Überdeckungskriterium. Sie erhält die während der Testvorbereitung erzeugten Daten identifiziert darauf basierend die zu erfüllenden Teilziele, legt die Bearbeitungsreihenfolge dieser Teilziele fest und berechnet während der Optimierung die Zielfunktionswerte (Bewertungen) der von der Optimierungskomponente (*Optimierung*) übermittelten potentiellen Lösungen. Jedes der Teilziele stellt ein separates Optimierungsproblem dar und ist zur Erfüllung des gewählten Überdeckungskriteriums erforderlich. Ein Teilziel gilt dabei als erfüllt, wenn der Evolutionäre Algorithmus Testdaten generiert hat, die als Eingabe verwendet, eine Ausführung des selbigen zur Folge haben. Die Testcontrol überprüft bei der Ausführung eines Teilziels mit der übermittelten Testdatenmenge gleichzeitig, welche anderen Teilziele damit erreicht werden können. Alle während dieser Überprüfung nicht erreichten Teilziele müssen anschließend separat optimiert werden, weshalb die Begriffe Testziel und Teilziel synonym verwendet werden können. Je nach Überdeckungskriterium entspricht ein Testziel unterschiedlichen Elementen des korrespondierenden Kontrollflussgraphen:

- Bei der *Anweisungsüberdeckung* wird ein Satz von Testdaten gesucht, welcher alle Anweisungen der Funktion ausführt. Die zu identifiziernden und zur Ausführung zu bringenden Teilziele sind somit alle Anweisungsknoten des Kontrollflussgraphen, da jede Anweisung darin einem Knoten zugewiesen ist.
- Bei der *Zweigüberdeckung* soll jeder Zweig, d.h. jede Kante des korrespondierenden Kontrollflussgraphen, zur Ausführung gebracht werden.
- Bei der *Bedingungsüberdeckung* sollen alle Verzweigungsanweisungen (Bedingungen) ausgeführt werden. Diese sind ebenfalls jeweils einem Knoten im Kontrollfluss-

graphen zugeordnet und sollen für ihre atomaren Bedingungen *wahr* und *falsch* zur Ausführung gebracht werden.

Nachdem die Generierung der Testdaten für jedes Testziel durchgeführt wurde, wird die dabei ermittelte Menge notwendiger Testdaten über den Testservice (*Evolutionary Testing Service*) an den Benutzer übermittelt.

Optimierung

Die Optimierung, welche in der ursprünglichen Version des Testsystems durch Genetische Algorithmen durchgeführt wird, generiert potentielle Lösungen für das aktuell zu erreichende Teilziel und optimiert diese mithilfe der Bewertungen der TestControl. Diese Optimierungen werden gemäß des in Abschnitt 4.1.2 näher erläuterten PEANuts[1]-Protokolls wie folgt durchgeführt: Zuerst wird der Optimierungskomponente die Schnittstelle der zu testenden Funktion, d.h. die Anzahl und die Typen der Eingabeparameter, übermittelt. Basierend auf diesen Informationen werden Testdaten erzeugt und an die TestControl übermittelt. Diese führt das Testobjekt mit den erhaltenen Eingaben aus und berechnet die dazugehörigen Fitnesswerte des daraus resultierenden Programmablaufs. Anschließend sendet die TestControl diese Fitnesswerte an die Optimierungskomponente und wartet auf die Übermittlung weiterer Testdaten. Dieser Kreislauf wird nun solange ausgeführt, bis ein Testdatensatz gefunden wird, der das zu erreichende Element überdeckt oder bis ein anderes Abbruchkriterium erfüllt ist.

Der *PEANuts-Server* übernimmt dabei zusammen mit der *GEA-Toolbox*[2] die Rolle der Optimierungskomponente. Der PEANuts Server führt die Kommunikation aus und reicht die jeweiligen Daten zur GEA-Toolbox weiter, welche die eigentlichen optimierenden Berechnungen durchführt. Die GEA-Toolbox ist ein Programmpaket, welches eine Vielzahl von Funktionen und Methoden zur Implementierung von Genetischen Algorithmen unter MATLAB zur Verfügung stellt. So werden beispielsweise alle im Kapitel 2 vorgestellten Genetischen Operatoren und Verfahren realisiert und unterstützt. Zusätzlich lässt sich der jeweilige Optimierungsfortschritt mit der *Peanuts-Visualisierung* visualisieren und damit nachvollziehen.

4.1.2 Das PEANuts-Protokoll

Das PEANuts-Protokoll wird innerhalb des Testsystems zur Kommunikation zwischen der Optimierungskomponente und der TestControl verwendet. Der Ablauf dieser Kommunikation ist in Abbildung 4.2 dargestellt. Die TestControl und der PEANuts Server werden durch die Entitäten Client und Server repräsentiert. Die TestControl teilt dem

[1] Program for Evolutionary Algorithm Network communications
[2] Genetic and Evolutionary Algorithm Toolbox for use with Matlab (Poh00)

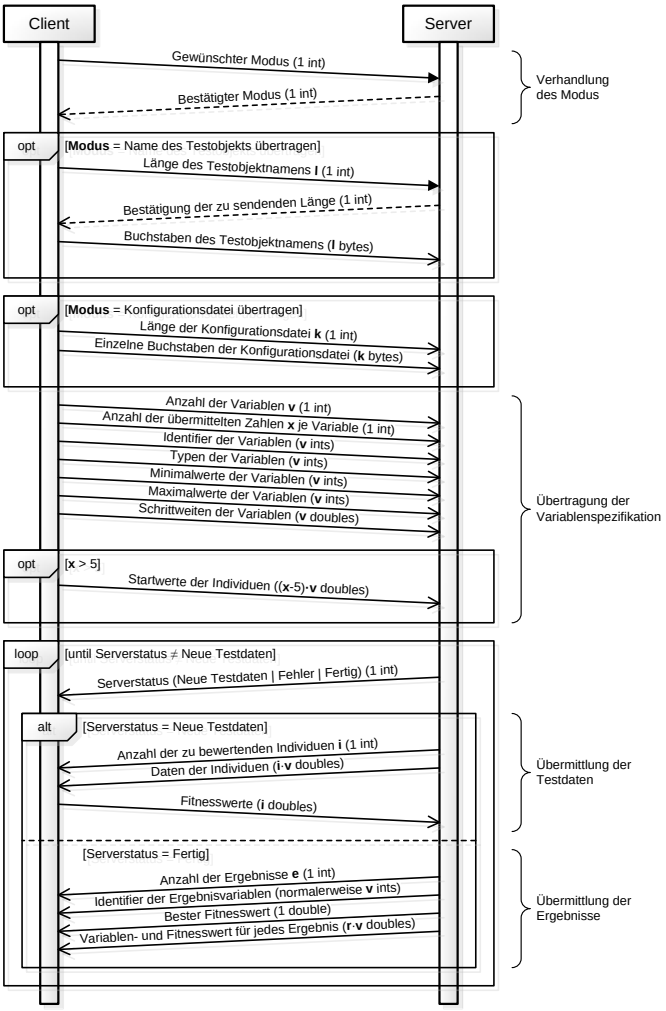

Abbildung 4.2: Das PEANuts-Protokoll zur Kommunikation zwischen Evolutionärem Algorithmus und der TestControl

Server zuerst ihren gewünschten Modus mit und wartet anschließend auf die Übermittlung des vom Server akzeptierten Modus. Der Modus wird dabei von einem ganzzahligen Wert (Integer) repräsentiert, welcher festlegen, ob die Optimierung im Debugmodus durchgeführt werden soll, ob der Server mittels einer optional zu sendenden Konfigurationsdatei spezifischen Einstellungen zugewiesen, oder ob der Name der zu testenden Funktion übermittelt werden soll. Im Debugmodus werden zusätzlich Statusinformationen des Clients an den Server übermittelt, um einen Einblick in den Ablauf der Optimierung zu verschaffen und dadurch die Möglichkeit zu haben, die Ursache für einen eventuell auftretenden Fehler ausfindig zu machen. Falls erwünscht, wird eine für den Server zu verwendende Konfigurationsdatei oder der Funktionsname des zu testenden Testobjekts übertragen. Anschließend wird dem Server die Variablenspezifikation des Testobjekts bzw. dessen zu optimierenden Teilziels mitgeteilt. Diese ist definiert durch die Anzahl der Variablen, deren Typ, deren Definitionsbereich und die zu verwendende minimale Schrittweite. Darüber hinaus werden optional Startwerte für die einzelnen Va-

riablen übermittelt, welche zur Initialisierung der Optimierung verwendet werden. Ist dem Server die Variablenspezifikation bekannt, teilt er dem Client seinen Zustand mit. Dieser kann drei Werte annehmen: die Bereitschaft, neue Testdaten zu senden, die Mitteilung über das Auftreten eines Fehlers und die Benachrichtigung über den Abschluss der Optimierung. Wenn kein Fehler aufgetreten und die Optimierung noch nicht beendet ist, so wird dem Client die gewünschte Anzahl der zu bewertenden Individuen mitgeteilt. Anschließend werden diese Individuen dimensionsweise übertragen, danach vom Client bewertet, der dann mit deren Bewertungen (Fitness) antwortet. Nachdem der Server den Client über die Fertigstellung der Optimierung informiert hat, übermittelt er ihm die endgültigen Ergebniswerte.

4.2 Erweiterungen

Die im Rahmen dieser Diplomarbeit erstellten Erweiterungen des Evolutionären Testsystems betreffen deren Optimierungskomponente, welche um die Unterstützung der Particle Swarm Optimization erweitert wurde. Wie in Abbildung 4.3 dargestellt, kann der Benutzer nun zwischen Genetischen Algorithmen und der Particle Swarm Optimization zur Optimierung auswählen. Für die Genetischen Algorithmen wird weiterhin die

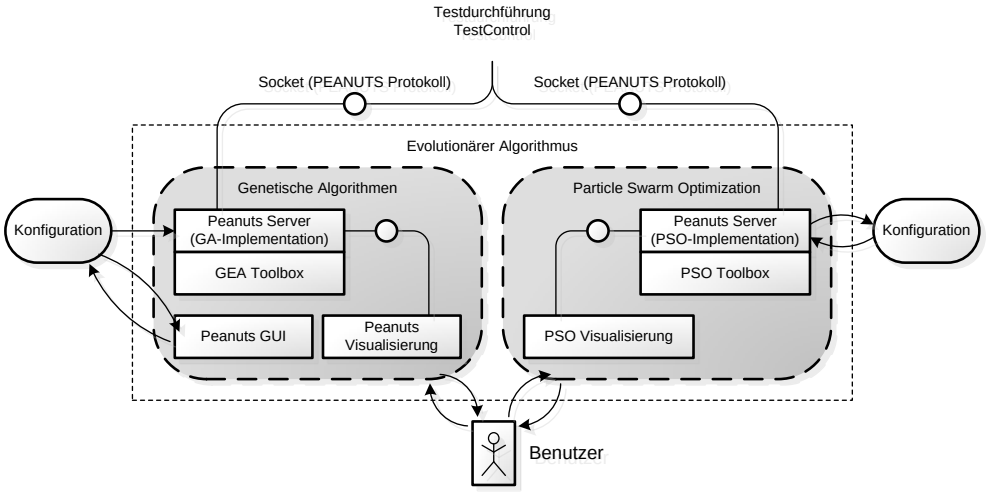

Abbildung 4.3: Die Erweiterung der Optimierungskomponente des Evolutionären Testsystems auf Unterstützung der Particle Swarm Optimization

Kombination aus dem PEANuts-Server und der GEA-Toolbox verwendet, für die neue Unterstützung der PSO wurde ein eigener Server entwickelt. Dieser besteht ebenfalls aus zwei funktionellen Komponenten: Dem PEANuts-Server für die Kommunikation gemäß des PEANuts-Protokolls zwischen TestControl und Optimierungskomponente sowie der PSO-Toolbox, welche für die Optimierung verwendet wird.

In den folgenden beiden Abschnitten werden diese beiden Komponenten näher beschrieben. Abschnitt 4.2.1 beschreibt die Funktionsweise des für die PSO entwickelten

PEANuts-Servers, Abschnitt 4.2.2 geht näher auf die PSO-Toolbox und die darin implementierten Algorithmen ein.

4.2.1 Beschreibung des PEANuts-Servers für PSO

Der neu implementierte PEANuts-Server kommuniziert mit der TestControl mittels des in Abschnitt 4.1.2 vorgestellten PEANuts-Protokolls und verwendet zur Optimierung die PSO-Toolbox. Der Server stellt eine grafische Benutzeroberfläche bereit und ermöglicht somit die komfortable Konfiguration sowohl des Servers als auch der verwendeten Algorithmen. Zusätzlich ist eine einfache Visualisierung des Optimierungsverlaufs mithilfe eines „Farbteppichs“ (farbliche Darstellung der Fitnesswerte aller Individuen) realisiert worden.

Der Dienst des PEANuts-Servers wird einer beliebigen Anzahl von Clients über einen frei wählbaren Port zur Verfügung gestellt. Meldet sich ein Client an, wird ein separater Optimierungsprozess gestartet, um die parallele Kommunikation mit mehreren Clients zu ermöglichen. Alle gleichzeitig angemeldeten und bearbeiteten Clients werden zur Übersicht mitsamt ihres derzeitigen Fortschritts in einer Liste aufgelistet.

4.2.2 Beschreibung der PSO-Toolbox

Die im Rahmen dieser Arbeit entworfene und implementierte Particle Swarm Optimization Toolbox (PSO-Toolbox) ist eine Funktionsbibliothek, die eine Vielzahl von Funktionen und Methoden zur Implementation verschiedener Versionen und Varianten der Particle Swarm Optimization zur Verfügung stellt.

Zur Zeit sind die vier in Kapitel 2 Abschnitt 2.2 vorgestellten Varianten LBest-PSO, GBest-PSO, CL-PSO und G-PSO implementiert. Bei den beiden zuerst genannten Varianten wird zusätzlich eine bezüglich der Optimierungszeit variable Trägheit (inertia weight) der Partikel verwendet, da beide Verfahren dadurch signifikant verbessert werden können (SE98a). Global konfigurierbar ist sowohl die Anzahl der zu verwendenden Partikel, die Richtung der Optimierung, d.h. ob ein Minimum oder ein Maximum gefunden werden soll als auch das zu verwendende Verfahren zur Beschränkung der Partikelpositionen. Implementiert sind hier die vier Metaphern der absorbierenden, der reflektierenden, der unsichtbaren und der dämpfenden Wände (vgl. Abschnitt 2.2.1). Zusätzlich kann neben dem Erreichen des globalen Optimums auch ein garantiertes Abbruchkriterium festgelegt werden: eine maximale Anzahl von Iterationen (Generationen) oder eine maximale Anzahl von Fitnessfunktionsevaluationen.

Bei der Entwicklung wurde Wert auf eine möglichst einfache Erweiterbarkeit der vorhandenen Routinen gelegt. Da sich die verschiedenen PSO-Varianten hauptsächlich in der Berechnung der Folgegeschwindigkeiten und -positionen der Partikel unterscheiden,

konnte eine zentrale Klasse entwickelt werden, welche die Verwaltung des Schwarms übernimmt. So ist es ohne großen Aufwand möglich, neue PSO-Varianten zu implementieren.

4.2.3 Verifizierung der Implementierung

Um die fehlerfreie Implementierung der Algorithmen sicherzustellen, wurden diese auf sieben bekannte Testfunktionen angewandt, welche auch von den Autoren der PSO-Varianten verwendet wurden. Ein Vergleich der Ergebnisse derer und der in dieser Arbeit entstehenden Optimierungen lassen dabei einen Rückschluss auf die Funktionsfähigkeit und Korrektheit der Implementation zu. Ebenso sollten die durchgeführten Optimierungen als Vergleich der PSO-Algorithmen untereinander dienen. Die verwendeten Testfunktionen sind in Tabelle 4.1 aufgelistet und sollen im Folgenden kurz charakterisiert werden. In Abbildung 4.4 sind diese Testfunktionen für zwei Dimensionen dargestellt,

Funktionsname	Formel	
Ackley	$f_1(x) =$	$-20e^{\left(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right)} - e^{\left(\frac{1}{d}\sum_{i=1}^{d}\cos 2\pi x_i\right)} + 20 + e^1$
Griewangk	$f_2(x) =$	$\frac{1}{4000}\sum_{i=1}^{d}\left(x_i^2 - \prod_{i=1}^{d}\cos\frac{x_i}{\sqrt{i}} + 1\right)$
Rastrigin	$f_3(x) =$	$\sum_{i=1}^{d}(x_i^2 - 10\cos 2\pi x_i + 10)$
Rosenbrock	$f_4(x) =$	$\sum_{i=1}^{d}(100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$
Schwefel	$f_5(x) =$	$418.9829 \cdot d - \sum_{i=1}^{d} x_i \sin(\lvert x_i\rvert^{\frac{1}{2}})$
Sphere	$f_6(x) =$	$\sum_{i=1}^{d} x_i^2$
Weierstrass	$f_7(x) =$	$\sum_{i=1}^{d}\left(\sum_{k=0}^{k_{max}}\left(a^k\cos(2\pi b^k(x_i + 0.5))\right)\right)$
		$-d \cdot \sum_{k=0}^{k_{max}}\left(a^k\cos(2\pi b^k \cdot 0.5)\right)$
		mit $a = 0.5, b = 3, k_{max} = 20$

Tabelle 4.1: Übersicht über die für die Verifikation genutzten Testfunktionen

um ein gutes Verständnis des Aussehens und der Charakteristika dieser Funktionen zu erhalten.

Alle im Folgenden vorgestellten Funktionen sollen minimiert werden, der zu findende minimale Funktionswert beträgt jeweils 0. Die Sphären- (Sphere-) und die Rosenbrock-Testfunktion gelten dabei als sehr einfache Optimierungsfunktionen. Die Sphärenfunktion ist unimodal, besitzt also nur einen Extremwert: das globale Optimum im Ur-

sprung des Koordinatensystems. Die Funktion von Rosenbrock (auch Bananenfunktion genannt) ist für zwei Dimensionen ebenfalls unimodal und besitzt in höheren Dimensionen ein lokales Optimum. Ihr globales Optimum befindet sich innerhalb eines schmalen, parabolisch geformten und flachen Tales. Das Auffinden dieses Tales ist dabei sehr einfach, wohingegen die eigentliche Konvergenz zum Optimum sehr schwierig ist. Schwieriger, da hochgradig multimodal, sind die restlichen Funktionen. Die Ackley-Funktion besitzt ihr enges globales Optimum im Ursprung umgeben von vielen lokalen Optima. Die Griewangk-Funktion ist, auf ihren gesamten Definitionsbereich bezogen, der Sphärenfunktion sehr ähnlich, bis auf die Tatsache, dass deren gesamte Oberfläche von kleinen Erhebungen und Vertiefungen gekennzeichnet ist. Sie weist die Eigenschaft auf, für kleinere Dimensionen schwieriger lösbar zu sein als für höhere (WRDM96). Die Rastrigin-Funktion basiert auf der Sphärenfunktion und wurde durch Einbeziehung eines Kosinus-Terms um viele lokale Optima erweitert. Die Komplexität von Schwefels Funktion beruht auf dem Vorhandensein eines tiefen lokalen Optimums, welches weit vom globalen Optimum entfernt liegt. Daher werden Suchalgorithmen möglicherweise getäuscht, indem sie sich auf dem Weg zu dem erfolgversprechenden tiefen lokalen Optimum vom gesuchten globalen Optimum entfernen. Die Weierstrass-Funktion ist zwar kontinuierlich, allerdings nur in wenigen Punkten differenzierbar, weshalb sie häufig zum Vergleich von stochastischen mit gradientenbasierten Optimierungsverfahren verwendet wird. Sie besitzt unzählige lokale Optima innerhalb des gesamten Definitionsbereichs, ihr globales Optimum befindet sich im Ursprung.

Die Testfunktionen wurden nacheinander von jedem der Algorithmen optimiert, wobei der Suchraum jeweils 30 Dimensionen umfasste. Zur Sicherstellung der statistischen Aussagekraft wurde dafür eine Wiederholungsrate der Experimente von 100 verwendet und anschließend das arithmetische Mittel berechnet. Bei der Lösung von realen Problemen nimmt die Berechnung der Fitnesswerte gewöhnlicherweise, wie beispielsweise beim Evolutionären Strukturtest, die meiste Zeit in Anspruch. Aus diesem Grund werden hier nicht die Ausführungszeiten der einzelnen algorithmenspezifischen Berechnungen verglichen, sondern das Erreichen eines Fitnesswertes in Abhängigkeit der dafür benötigten Aufrufe der Fitnessfunktion. Als garantiertes Abbruchkriterium gilt das Erreichen von 200.000 Fitnessfunktionsevaluationen. Die während der Optimierung verwendeten Such- und Initialisierungsbereiche der einzelnen Testfunktionen sind in Tabelle 4.2 dargestellt. Diese wurden von den Autoren des G-PSO (PB06) bzw. des CL-PSO (JJLB06) verwendet und beziehen sich jeweils auf alle Dimensionen. Alle Varianten weisen eine Schwarmgröße von 40 Partikeln auf. Die sozialen und kognitiven Beschleunigungsparameter c_1 und c_2 des GBest- und des LBest-Modells der Standardvariante der PSO wurden beide auf 2.0 gesetzt, die Trägheit der Partikel wird im Laufe des Optimierungsprozesses linear von anfänglich 0.9 auf 0.4 verringert. Der Approximationskoeffizient ε des G-PSO wird auf 10^{-8} gesetzt, die Schrittweite wird durch 2 und 4 begrenzt und in Schritten von 0.5 adaptiert. Das refreshing gap des CL-PSO wird, wie von den Autoren vorgeschlagen, auf 7 und der Beschleunigungskoeffizient c auf 1.49445 gesetzt. Das inertia weight wird wie

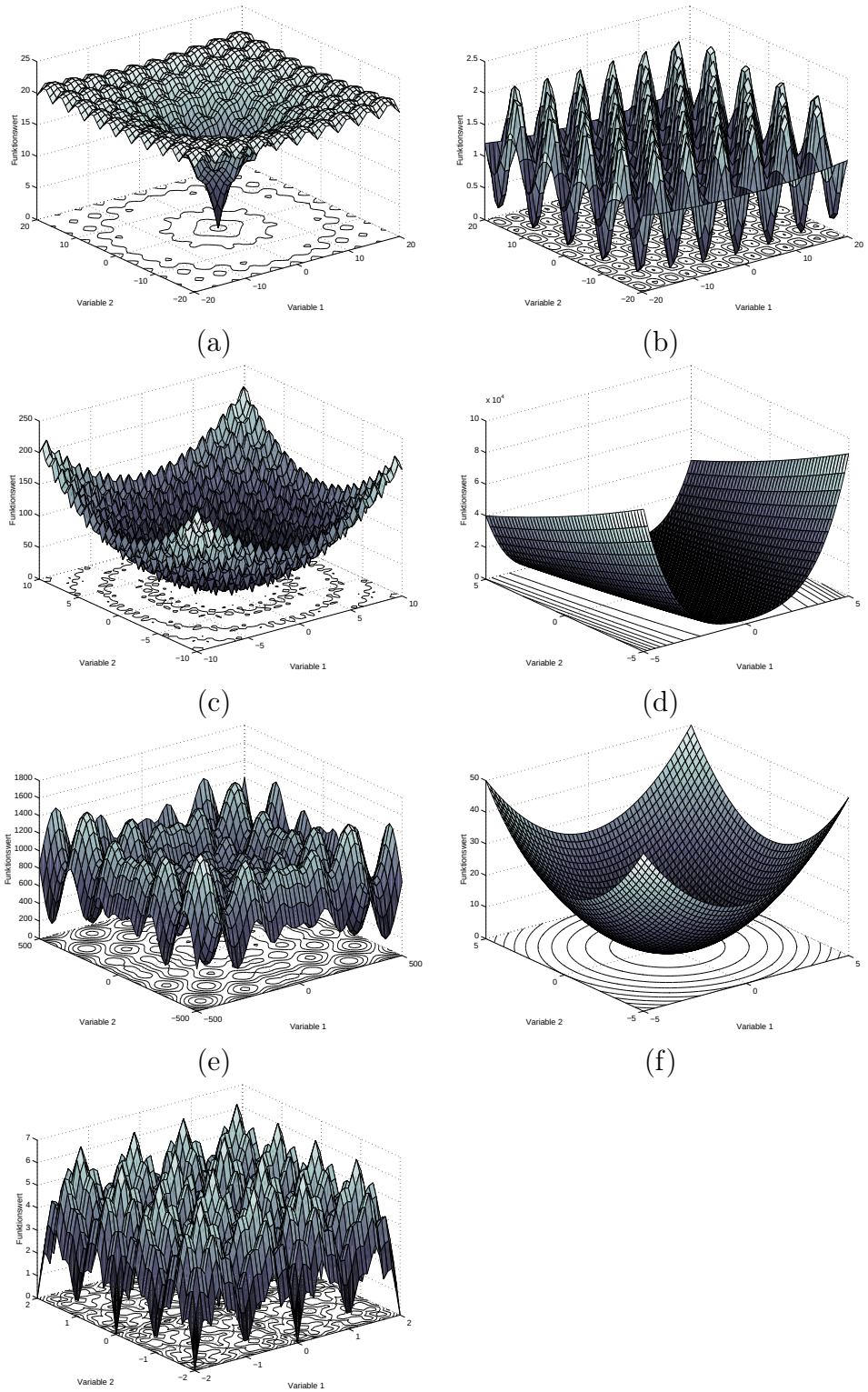

Abbildung 4.4: Dreidimensionale Darstellung der Funktionslandschaften der während der Verifikation genutzten Testfunktionen. (a) Ackley, (b) Griewank, (c) Rastrigin, (d) Rosenbrock, (e) Schwefel, (f) Sphere, (g) Weierstrass

Funktionsname	Suchbereich	Initialisierungsbereich
Ackley	$[-32, 32]^d$	$[15, 32]^d$
Griewangk	$[-600, 600]^d$	$[300, 600]^d$
Rastrigin	$[-10, 10]^d$	$[2.56, 5.12]^d$
Rosenbrock	$[-100, 100]^d$	$[15, 30]^d$
Schwefel	$[-500, 500]^d$	$[-500, 500]^d$
Sphere	$[-100, 100]^d$	$[50, 100]^d$
Weierstrass	$[-0.5, 0.5]^d$	$[-0.5, 0.2]^d$

Tabelle 4.2: Übersicht über die verwendeten Such- und Initialisierungsbereiche der Testfunktionen

bei den Standardvarianten im Laufe der Optimierung linear von 0.9 auf 0.4 verringert. Die drei zuerst genannten Varianten verwenden zusätzlich das Konzept der dämpfenden Wände zur Beschränkung auf den vorgegebenen Suchraum, wobei der CL-PSO dafür den Ansatz der unsichtbaren Wände nutzt.

Der gemittelte Optimierungsverlauf der Algorithmen für die einzelnen Testfunktionen wird in Abbildung 4.5 dargestellt. Die Diagramme verwenden logarithmische Skalen, um die sehr schnell fallenden Funktionsgraphen besser darstellen und interpretieren zu können. Die erhaltenen Ergebnisse waren vergleichbar mit den Ergebnissen der Autoren und der Konvergenzverlauf ist fast identisch. Somit kann von einer korrekten Implementation ausgegangen werden.

Die beiden Standardalgorithmen brachten erwartungsgemäß unterschiedliche Resultate hervor. Während das LBest-Modell bei multimodalen Funktionen innerhalb der gegebenen Zeit bessere Lösungen finden konnte, wies das GBest-Modell eine höhere Konvergenzgeschwindigkeit auf. Gleichzeitig konvergierte es allerdings auch in lokale Optima, beispielsweise bei der Ackley-, Griewank- und Weierstrassfunktion. An komplexeren Funktionen wie die von Rastrigin und Schwefel scheiterten beide durch Konvergenz in eines der lokalen Optima. Der G-PSO verfügt im Allgemeinen über eine sehr hohe Konvergenzgeschwindigkeit und ist im Vergleich zum GBest-Modell robuster gegenüber lokalen Optima. Dennoch konvergiert auch der G-PSO bei komplizierten Funktionen (Griewangks und Schwefels Funktion) vorzeitig in lokale Optima. Demgegenüber verlief sich der CL-PSO in keiner der optimierten Funktionen in einem lokalen Optima. So fand er als einziger Algorithmus im Test die Region des globalen Optimums von Schwefels Funktion. Die Stärken des CL-PSO bestehen also vor allem, wie bereits von dessen Autoren beschrieben, in der Optimierung von komplexen multimodalen Funktionen, wobei die Optimierung von einfach zu lösenden und sogar unimodalen Funktionen mit einer langsameren Konvergenz einhergeht.

Zum weiteren Vergleich der Genetischen Algorithmen und der Particle Swarm Optimization für den Evolutionären Strukturtest wurden aufgrund dieser Ergebnisse zunächst

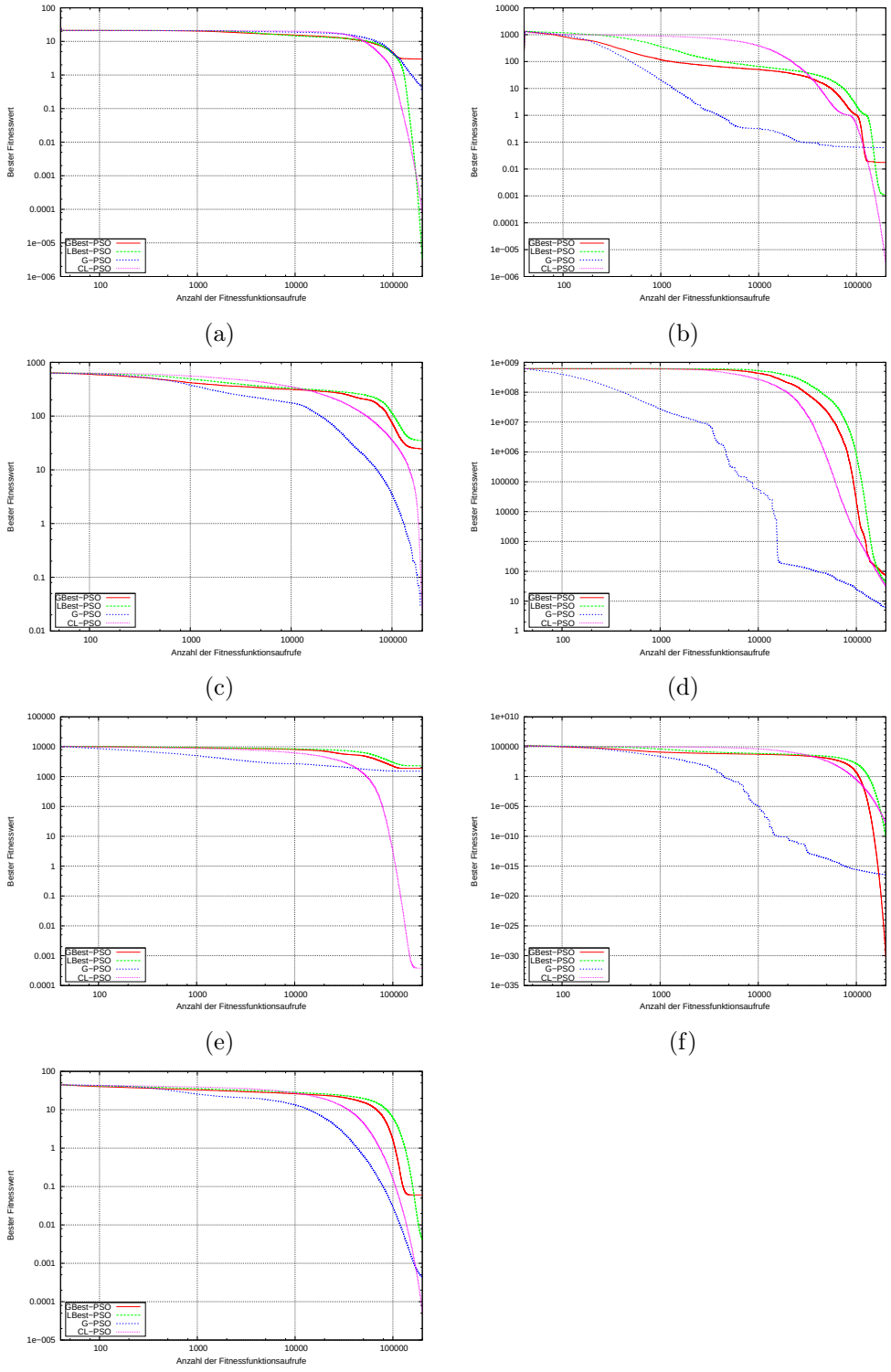

Abbildung 4.5: Gemittelte Konvergenzcharakteristika der implementierten PSO-Algorithmen für die Testfunktionen von (a) Ackley, (b) Griewank, (c) Rastrigin, (d) Rosenbrock, (e) Schwefel, die (f) Sphärenfunktion (Sphere) und (g) die Testfunktion von Weierstrass

der G-PSO und der CL-PSO als Repräsentanten der PSO verwendet. Da die dabei zu optimierenden Zielfunktionen eine sehr hohe Komplexität (u.a. durch eine Vielzahl an lokalen Optima) aufweisen können, muss Wert auf eine hohe Robustheit der Algorithmen gegenüber lokalen Optima gelegt werden. Da die Berechnung der Zielfunktionswerte zudem sehr rechenintensiv ist, sollte ebenfalls eine schnelle Konvergenz gegeben sein. Wie sich jedoch bereits bei den ersten industriellen Testobjekten herausstellte, konvergierte der G-PSO dabei oft vorzeitig in lokale Optima, was die Anwendung als Optimierungskomponente innerhalb des Evolutionären Testsystems ausschließt. Aus diesem Grund wurde die Verwendung des CL-PSO als Repräsentant der PSO beschlossen.

Kapitel 5
Experimente

In Kapitel 4 ist das Evolutionäre Testsystem mitsamt der Erweiterung auf die Unterstützung der Particle Swarm Optimization für den Evolutionären Strukturtest vorgestellt und beschrieben worden. In diesem Kapitel soll geklärt werden, wie gut die Particle Swarm Optimization auf dieses Anwendungsgebiet angewandt werden kann und ob sie im Vergleich zu den bislang verwendeten Genetischen Algorithmen Vorteile in Bezug auf die Effektivität und die Effizienz des Optimierungsprozesses aufweisen kann.

In Abschnitt 5.1 werden die für den Vergleich beider Optimierungsverfahren verwendeten Konfigurationen der Optimierungskomponenten genauer spezifiziert.

Abschnitt 5.2 widmet sich dem Vergleich beider Optimierungskomponenten für die Anwendung auf künstlich erzeugte Testobjekte. Diese unterscheiden sich beispielsweise in der Anzahl der zu optimierenden Variablen und der vorhandenen lokalen Optima, weshalb sie erste Rückschlüsse auf sowohl die Verwendbarkeit und Eignung der PSO für den Evolutionären Strukturtest als auch einen ersten Vergleich mit den GAs zulassen.

Abschnitt 5.3 erläutert anschließend die Anwendung beider Techniken auf reale Testobjekte aus dem industriellen Automobilkontext. Diese sind vor allem aufgrund ihrer unterschiedlichen Struktur und Komplexität von besonderem Interesse für den durchzuführenden empirischen Vergleich.

5.1 Konfiguration der Algorithmen

Im Folgenden soll eine Übersicht über die verwendete Konfiguration beider Optimierungskomponenten für den durchzuführenden Vergleich der Genetischen Algorithmen mit der Particle Swarm Optimization, beschrieben werden.

Für alle durchgeführten Experimente wurde die Zweigüberdeckung als Überdeckungskriterium ausgewählt, da ihre Relevanz von vielen industriellen Qualitätsstandards akzeptiert und ihr Einsatz empfohlen wird. Da der Funktionswert der Zielfunktion aller

Testobjekte im zu erreichenden globalen Optimum genau 0 beträgt, kann für beide Optimierungskomponenten als Abbruchkriterium das Erreichen dieses Fitnesswertes verwendet werden.

Im Laufe dieser Arbeit werden häufig die Bezeichnungen der erreichten Effektivität bzw. der Effizienz der Suche verwendet. Mit der Effektivität ist dabei die Erfolgsrate, d.h. der Anteil der erfüllten Teilziele zur Anzahl aller zu erfüllender Teilziele des Testobjektes gemeint. Die Effizienz ist synonym als ein Maß für die Anzahl der benötigten Zielfunktionsaufrufe zu verstehen. Eine hohe Effizienz steht dabei für eine geringe Anzahl an benötigten Zielfunktionsaufrufen.

5.1.1 Konfiguration der GEA-Toolbox

Tabelle 5.1 stellt die Konfiguration der GEA-Toolbox dar. Diese Einstellungen haben

Parameter		Wert
Subpopulationen	Anzahl	6
	Größe	jeweils 40 Individuen
Konkurrenz	Intervall	10
	Rate	0.1
	Minimale Größe	10
Migration	Topologie	Komplettes Netz
	Intervall	13
	Rate	0.1
Selektion	Name	Stochastic Universal Sampling
	Druck	1.7
	Generation gap	0.9
Rekombination	Typ	Diskret
	Rate	1.0
Mutation	Typ	Für reelle Variablen
	Präzision	17
Mutationsbereich	Subpopulation 1	0.1
	Subpopulation 2	0.01
	Subpopulation 3	0.001
	Subpopulation 4	0.0001
	Subpopulation 5	0.00001
	Subpopulation 6	0.000001
Wiedereinfügen	Typ	Fitnessbasiert

Tabelle 5.1: Konfiguration der GEA-Toolbox

sich während der letzten Jahre durch zahlreiche Experimente mit verschiedenen industriellen Testobjekten ergeben. Die ersten beiden Zeilen deuten eine Verwendung des regionalen Modells mit insgesamt sechs Unterpopulationen an. Aufgrund der variierenden Mutationsrate unter den Subpopulationen können einige davon erfolgreicher sein als andere. Konkurrenz zwischen diesen Unterpopulationen führt während der Suche zu einem Übergang von Individuen erfolgloserer zu erfolgreicheren Subpopulationen.

5.1.2 Konfiguration der PSO-Toolbox

Die Konfiguration ist in Tabelle 5.2 dargestellt. Bis auf die beiden im Folgenden be-

Parameter	Wert
Anzahl der Partikel	40
Algorithmus	CL-PSO
Inertia weight ω	Linear von 0.9 auf 0.4 verringert
Beschleunigungskoeffizient c	1.49445
Maximale Geschwindigkeit V^d_{max}	$(ub^d - lb^d)/2$
Refreshing gap m	7
Suchraumbegrenzung	Dämpfende Wände

Tabelle 5.2: Konfiguration der PSO-Toolbox

schriebenen kleinen Veränderungen wurden die Parameter des CL-PSO auf die von dessen Autoren vorgeschlagenen Werte gesetzt (JJLB06). Die erwähnten Veränderungen betreffen sowohl die Fähigkeit der Verwendung und Optimierung von diskreten Variablen als auch die Verwendung unterschiedlicher Suchraumbegrenzung. Ersteres wird, wenn die untersuchten Variablen diskret sind, durch einfaches Runden der potentiellen Lösungen auf den nächsten ganzzahligen Wert erreicht. Die Entscheidung, dämpfende Wände zu verwenden, entstammt im Rahmen dieser Arbeit durchgeführten empirischen Untersuchungen mit den verschiedenen Wand-Typen. Dabei kann es bei Verwendung des Konzepts der unsichtbaren Wände in einigen wenigen Fällen zu einem wiederholten und unerwünschten Ausbleiben der Partikelaktualisierungen kommen. Besser geeignet waren die absorbierenden und reflektierenden Wände. Welches dieser Verfahren dominierte, hing jedoch stark vom jeweiligen Problem ab. Das Paradigma der dämpfenden Wände geht einen stochastischen Mittelweg und ist daher für diese Voraussetzungen am besten geeignet.

5.2 Experimente mit künstlichen Testobjekten

Zur Untersuchung der Leistungsfähigkeit der Particle Swarm Optimization und zum Vergleich dieser mit den Genetischen Algorithmen wurden Testobjekte erstellt, für welche dann vom evolutionären Testsystem eine automatische Testdatengenerierung durchgeführt werden soll. Deren unterschiedliche Komplexitätsgrade sollen einen Vergleich der zu untersuchenden Suchtechniken ermöglichen. In den folgenden Abschnitten werden die dazu verwendeten Testobjekte genau beschrieben und die Ergebnisse der dafür ausgeführten Strukturtests unter Verwendung beider Optimierungsverfahren präsentiert.

Als Vergleichskriterium werden dabei die während der Optimierung erreichten Zielfunktionswerte verwendet. Ein Teilziel gilt als erfüllt, wenn der Zielfunktionswert 0 erreicht werden konnte. Die Effektivität der Optimierungsverfahren ist somit durch die Differenz

des erreichten Zielfunktionswertes zur 0 gegeben, während die Effizienz durch die Anzahl der benötigten Zielfunktionsaufrufe zum Erreichen eines bestimmten Wertes gegeben ist. Im Bezug auf den Softwaretest und die dafür notwendige Testdatengenerierung spielt die Nähe zum globalen Optimum keine Rolle, da dafür das Erreichen des Zielfunktionswertes 0 notwendig ist. Sie soll hier allerdings dennoch zum Vergleich der Leistungsfähigkeit und der Konvergenzgeschwindigkeiten beider Optimierungsverfahren verwendet werden.

5.2.1 Kriterien zur Fallstudienauswahl

Für den Vergleich der Genetischen Algorithmen und der Particle Swarm Optimization waren die folgenden drei Testkriterien von Interesse:

- Anzahl der Parameter (Variablen im Datenraum)
- Datentyp der Variablen
- Anzahl der lokalen Optima

Während die Anzahl der gleichzeitig zu optimierenden Parameter die Suchraumgröße und damit auch den Optimierungsaufwand offensichtlich erhöht, ist eine Variabilität der Datentypen der zu optimierenden Parameter für einen Vergleich interessant, weil die PSO vor allem bei kontinuierlichen und weniger bei diskreten Optimierungsproblemen ihr Stärken aufweisen kann. Funktionen mit einer höheren Anzahl an lokalen Optima sind im Allgemeinen schwieriger zu optimieren. Deshalb soll diese Anzahl dem Vergleich der Techniken auf die Robustheit gegenüber lokalen Optima dienen.

Insgesamt wurden 25 künstliche Testobjekte unterschiedlicher Komplexität erstellt. Diese unterscheiden sich sowohl in der Anzahl und dem Typ der zu optimierenden Parameter und in der Anzahl der vorhandenen lokalen Optima. Die lokalen Optima beziehen sich dabei auf den Suchraum der Zielfunktion, welche aus dem, gemäß des Zweigüderdeckungskriteriums zu erreichenden Zweig resultiert.

Jedes der Testobjekte besteht aus einer Bedingung, welche es zur Erreichung des Zielzweiges auf *wahr* zu setzen gilt. Die Testobjekte sind nach dem folgenden in Pseudo-Notation dargestellten Schema aufgebaut:

```
function Testobjekt(Formale Argumentliste)
{
  if (Bedingung)
    // Zu erreichender Zweig
}
```

Listing 5.1: Allgemeine Struktur der künstlichen Testobjekte

Die formale Argumentliste des Testobjektes bestimmt die Anzahl und Art der Parameter. Diese werden innerhalb der Bedingung verwendet. Die Bedingung besteht da-

bei jeweils aus mehreren logischen *Und*-Verknüpfungen, welche die in der Signatur spezifizierten Parameter gleichzeitig betrachtet. Zusätzlich verursachen logische *Oder*-Verknüpfungen lokalen Optima innerhalb der erzeugten Zielfunktion. Mit dieser Herangehensweise können Zielfunktionen konstruiert werden, welche sowohl mehrere Parameter als auch mehrere Optima besitzen.

Eine *Oder*-Verknüpfung von mindestens zwei Variablen führt zu einer Zielfunktionslandschaft mit ebenso vielen Optima bzw. einem Plateau (vgl. Kapitel 3.2.3). Zur Konstruktion eines zusätzlichen globalen Optimums müssen diese Terme zusätzlich mit dem gewünschten globalen Optimum konjugiert werden. In Abbildung 5.1 ist dieser Zusammenhang für zwei Beispielbedingungen dargestellt. Da die Konjugation zweier Operan-

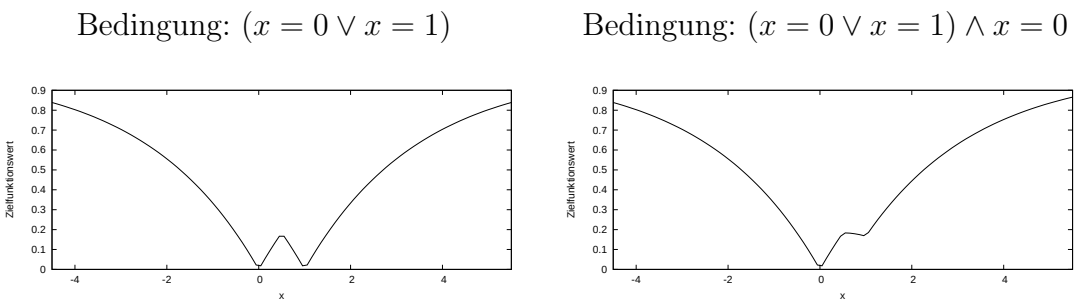

Abbildung 5.1: Beispielhafte Konstruktion eines globalen Optimums durch Konjugation

den während der Zielfunktionsberechnung die Berechnung des arithmetischen Mittels selbiger zur Folge hat, kommt es zu einer Anheben aller lokalen Optima, mit Ausnahme des zusätzlich konjugierten und gewünschten globalen Optimums, welches seinen Funktionswert beibehält.

In Tabelle 5.3 sind die Charakteristika der künstlichen Testobjekte dargestellt. Die Spalte *Variablen* beschreibt die Anzahl der Eingabeparameter, die das Testobjekt besitzt, die Spalte *Lok. Optima* gibt die Anzahl der lokalen Optima im von den Eingabeparametern aufgespannten Suchraum an. Die Spalten *boolean*, *integer* und *double* geben die verwendeten Variablentypen an. Als Beispiel sollen an dieser Stelle die Eigenschaften des Testobjekts $f12$ kurz erläutert werden. Bei diesem handelt es sich um eine Funktion mit zwei Parametern (einer vom Typ integer und einer vom Typ double), welche neben dem globalen Optimum im Ursprung insgesamt 24 lokale Optima besitzt. Abbildung 5.2 stellt die dabei entstehende Zielfunktionslandschaft dar. Der Quellcode aller Testobjekte befindet sich im Anhang.

Jedes Testobjekt wurde insgesamt 30 Mal von jedem Optimierungsverfahren optimiert, woraus anschließend, zur Sicherstellung der statistischen Relevanz, ein Mittelwert gebildet wurde. Um die Konvergenzgeschwindigkeit beider Verfahren zu beschleunigen, wird der extrem große Definitionsbereich der Parameter vom Typ *integer* und *double* auf den Bereich von -10^6 bis 10^6 eingeschränkt.

Testobjekt	Variablen	Lok. Optima	boolean	integer	double
f01	1	0	x		
f02	1	0		x	
f03	1	0			x
f04	1	1	x		
f05	1	1		x	
f06	1	1			x
f07	1	4		x	
f08	1	4			x
f09	2	0	x		
f10	2	0		x	
f11	2	0			x
f12	2	24		x	x
f13	3	0	x		
f14	3	0		x	
f15	3	0			x
f16	3	0	x	x	x
f17	3	49	x	x	x
f18	6	0	x		
f19	6	0		x	
f20	6	0			x
f21	6	0	x	x	x
f22	12	0	x		
f23	12	0		x	
f24	12	0			x
f25	12	0	x	x	x

Tabelle 5.3: Übersicht über die erzeugten künstlichen Testobjekte

5.2.2 Erwartete Ergebnisse

Beide Verfahren werden die Testobjekte $f01$, $f04$, $f09$, $f13$, $f18$, $f22$ und $f25$ vermutlich gleichermaßen schnell lösen können, da dabei lediglich Parameter vom Typ *boolean* optimal bestimmt werden müssen. Da die Parameter in diesem Fall jeweils nur zwei mögliche Belegungen annehmen können, werden die entsprechenden Lösungen voraussichtlich bereits während der Initialisierung gefunden. Aufgrund der kontinuierlichen Suche der PSO wird diese einerseits bei den Testobjekten, welche Parameter vom Typ *double* verwenden, möglicherweise bessere Ergebnisse liefern als die GAs. Diese könnten andererseits Vorteile bei der Optimierung von Testobjekten mit diskreten Parametern vom Typ *integer* aufweisen. Insgesamt jedoch sollte sowohl die PSO als auch die GAs in der Lage sein, alle Zielfunktionen der Testobjekte erfolgreich optimieren zu können, da diese sehr einfache Optimierungsprobleme darstellen. Falls die Zielfunktion eines Testobjektes nicht in der gegebenen Zeit (Anzahl der Zielfunktionsaufrufe) optimiert werden kann, so wird dies voraussichtlich an dem vorzeitigen Abbruch und weniger an der schlechten Effektivität des jeweiligen Verfahrens liegen. Dies sollte dann anhand der aufgezeichneten Konvergenzverläufe ersichtlich sein.

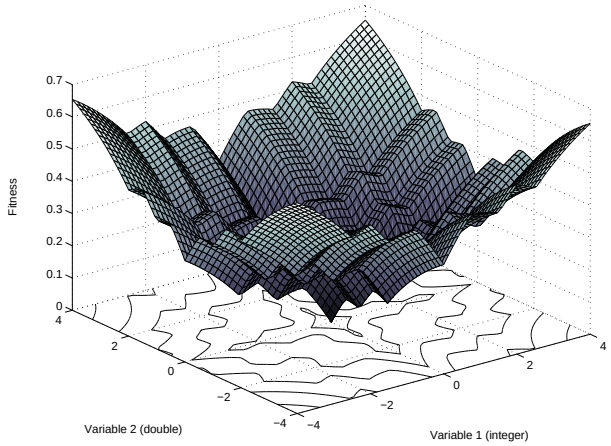

Abbildung 5.2: Zielfunktionslandschaft des zu erreichenden *wahr*-Zweigs des Testobjekts $f12$

5.2.3 Auswertung

Die Konvergenzcharakteristika der künstlichen Testobjekte sind in Abbildung 5.3, 5.4 und 5.5 dargestellt. Jeder Konvergenzverlauf ist jeweils in zwei Auflösungsstufen dargestellt. Auf der linken Seite ist der Wertebereich zwischen 0 und 1 dargestellt, für besonders schnell fallende Funktionswerte wurde dieser zusätzlich eingeschränkt. Auf der rechten Seite ist nur ein sehr kleiner Ausschnitt um den Funktionswert 0 zu sehen, um zu erkennen, welches Suchverfahren diesen zuerst erreicht. Die Ergebnisse für die Testobjekte $f01$, $f04$, $f09$, $f13$, $f18$, $f22$ und $f25$ sind davon ausgenommen, weil deren Zielfunktionen aufgrund der ausschließlichen Verwendung von Parametern des Typs *boolean* erwartungsgemäß von beiden Verfahren bereits während der Initialisierung erfolgreich optimiert werden konnten.

Betrachtet man die Funktionen, welche mehrere Parameter desselben Typs verwenden (Funktionen $f02$, $f10$, $f14$, $f19$ und $f23$ für den Typ *integer* und die Funktionen $f03$, $f11$, $f15$, $f20$ für den Typ *double*), so wird deutlich, dass beide Techniken bei einer höheren Anzahl an zu bestimmenden Parametern auch umso mehr Zeit für die Optimierung benötigen. Eine detailliertere Analyse dieser Resultate zeigt außerdem, dass die GAs nicht in der Lage waren, das gesuchte globale Optimum vor Eintreten des garantierten Abbruchkriteriums zu finden, wenn mehr als nur ein Parameter zu bestimmen war. Im Gegensatz dazu erreichte die PSO bei gleicher Anzahl von Zielfunktionsevaluationen das globale Optimum bei der Optimierung von einem, zwei oder drei Parametern unabhängig von dessen Typ. Obwohl beide Verfahren für die Optimierung von sechs oder zwölf Parametern keine erfolgreiche Konvergenz aufweisen konnten, erreichte die PSO bei der Verwendung der gleichen Ressourcen Lösungen, welche sich näher am globalen Optimum befinden, hier also kleinere Zielfunktionswerte besitzen. Die Einführung von künstlich konstruierten lokalen Optima (Funktionen $f05$, $f06$, $f07$ und $f08$) bewirkte keine bemerkenswerten Veränderungen.

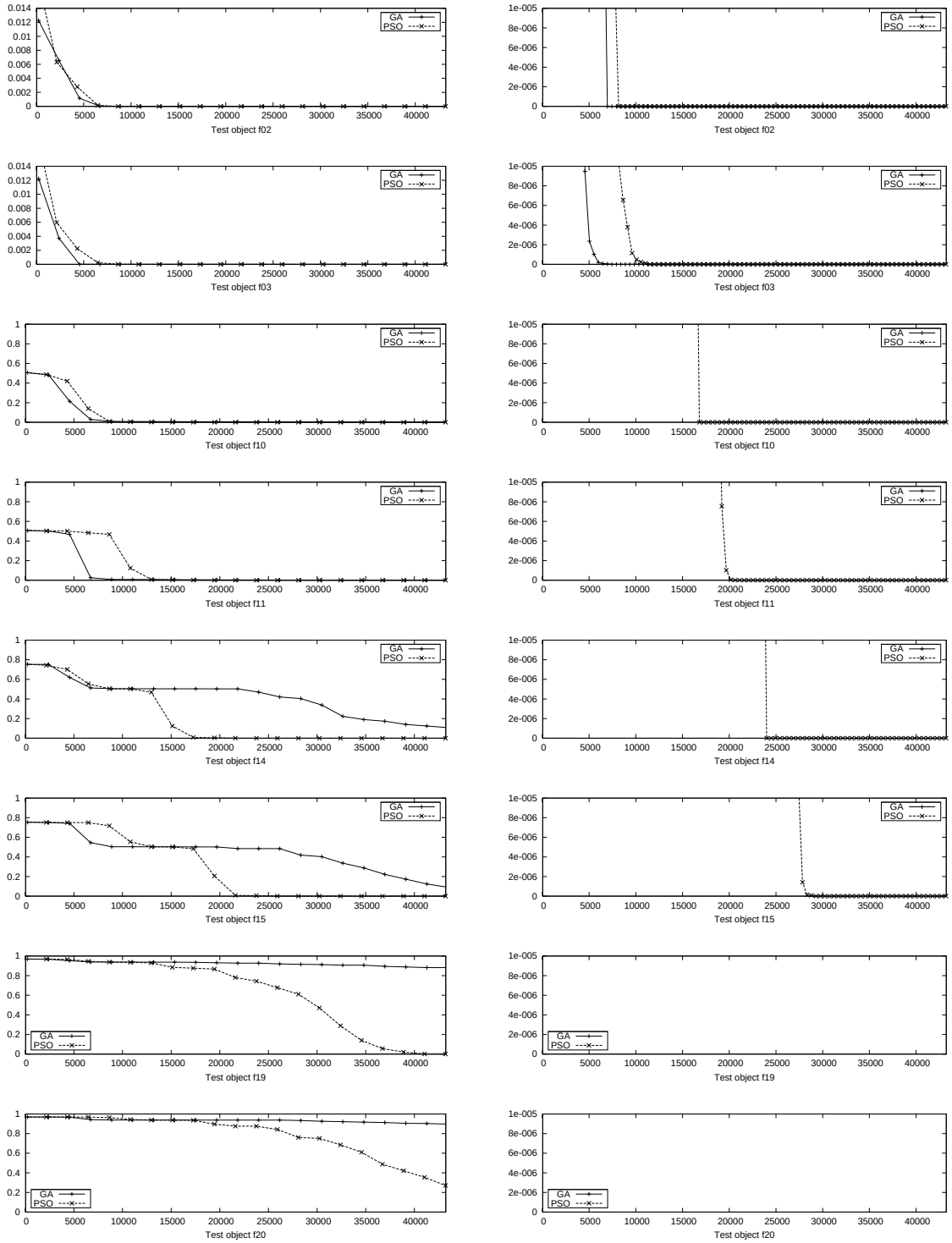

Abbildung 5.3: Die Konvergenzcharakteristika der GA und PSO für die künstlichen Testobjekte (f02 - f20): Die X-Achse stellt die Anzahl der Zielfunktionsevaluationen dar, die Y-Achse gibt den Fitnesswert wieder. Die Ergebnisse sind über 30 Läufe gemittelt.

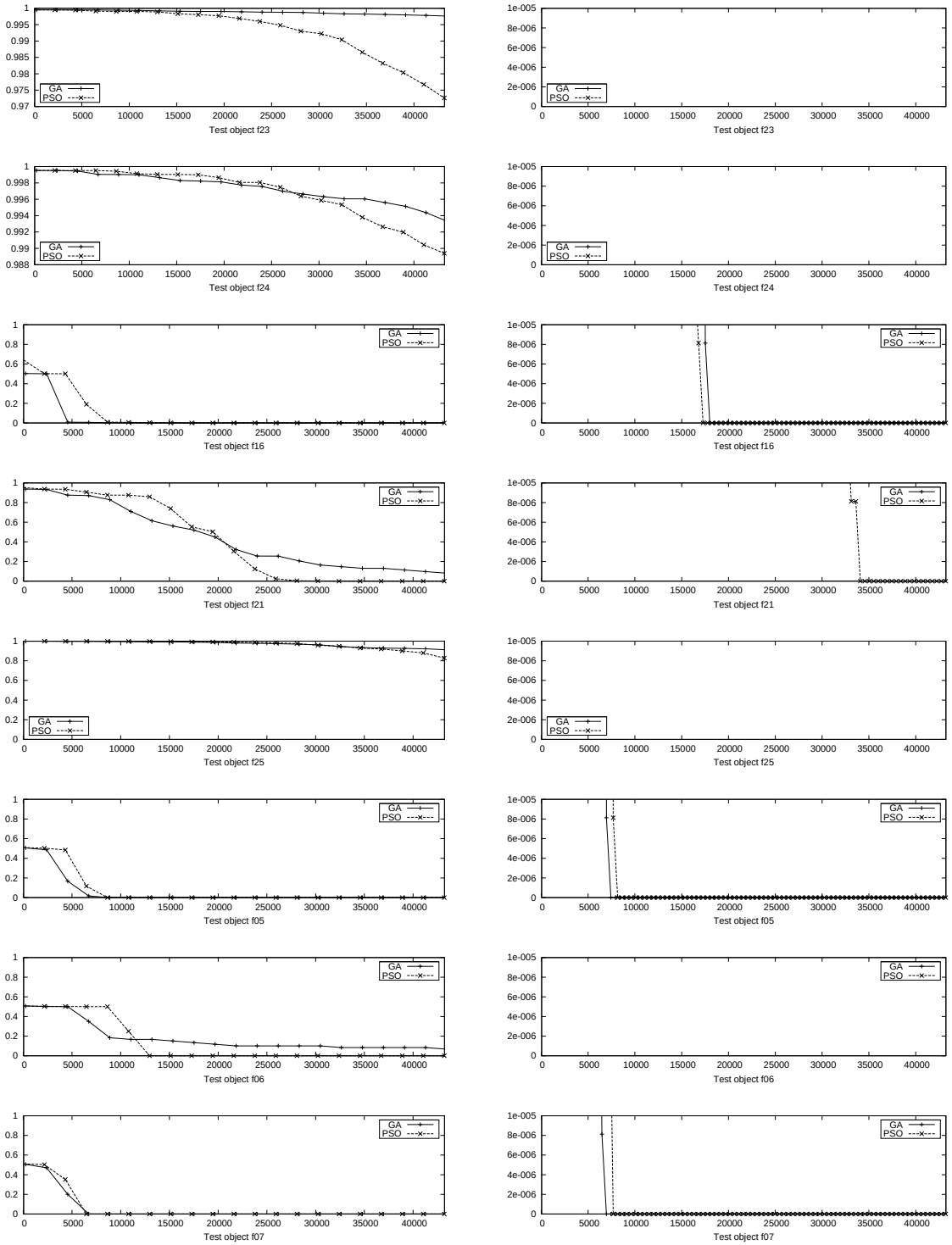

Abbildung 5.4: Die Konvergenzcharakteristika der GA und PSO für die künstlichen Testobjekte (f23 - f07): Die X-Achse stellt die Anzahl der Zielfunktionsevaluationen dar, die Y-Achse gibt den Fitnesswert wieder. Die Ergebnisse sind über 30 Läufe gemittelt.

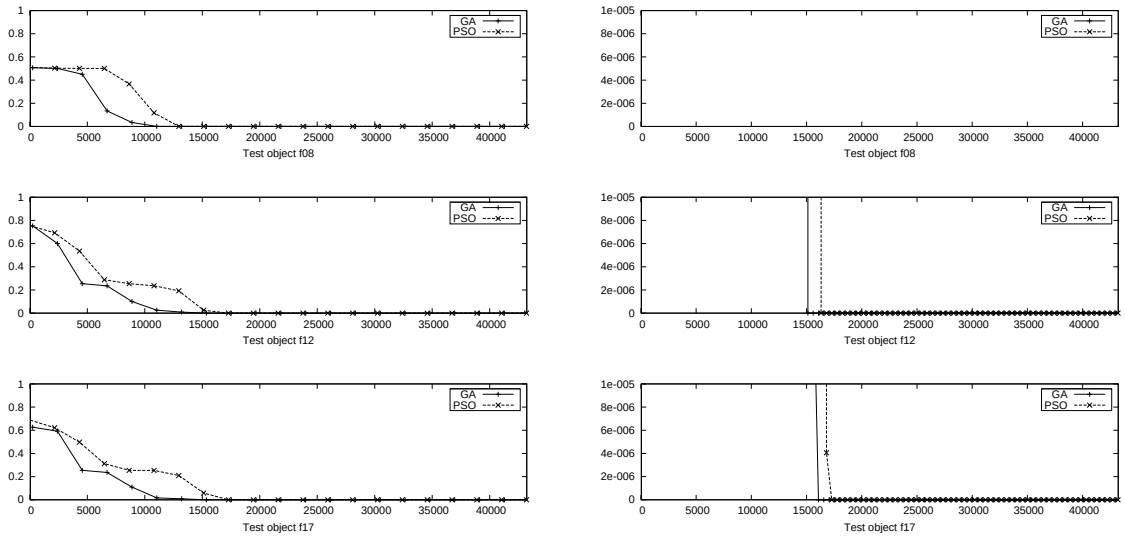

Abbildung 5.5: Die Konvergenzcharakteristika der GA und PSO für die künstlichen Testobjekte (f08 - f17): Die X-Achse stellt die Anzahl der Fitnessfunktionsevaluationen dar, die Y-Achse gibt den Fitnesswert wieder. Die Ergebnisse sind über 30 Läufe gemittelt.

Für die Funktionen, die verschiedene Arten von Parametertypen verwenden (Funktionen $f16$, $f21$ und $f25$), übertrafen die Leistungen der PSO die der GA: Entweder konnte das globale Optimum nach einer geringeren Anzahl von Zielfunktionsevaluationen erreicht werden, oder sie konnten ein besseres Ergebnis nach dem Ablauf der zulässigen Evaluationen erzielen. Das Vorhandensein von lokalen Optima innerhalb der Zielfunktion zweier Testobjekte mit unterschiedlichen Parametertypen (Funktionen $f12$ und $f17$) resultierte in einer geringfügig schnelleren Konvergenz der GA gegenüber der PSO.

In 13 der 19 gezeigten Fälle war die PSO den GAs überlegen, wohingegen in den restlichen sechs Fällen die GAs überlegen waren. Während allerdings die Überlegenheit der PSO bezüglich der Effektivität der Suche gegenüber den GAs in acht der 13 Fälle signifikant ist, existiert kein Fall, in dem die Überlegenheit der GA gegenüber der PSO signifikant ist.

Die Ergebnisse weichen demnach von den Erwartungen insofern ab, als dass sich die Überlegenheit der PSO gegenüber den GAs nicht ausschließlich auf Testobjekte mit kontinuierlichen Variablen beschränkt, sondern diese für alle gewählten Parametertypen innerhalb der durchgeführten Tests gegeben war. Im Allgemeinen kann aufgrund dieser Ergebnisse gesagt werden, dass die GAs eine geringfügig schnellere Konvergenz für einfache Optimierungsprobleme aufweisen, während die PSO hauptsächlich bei komplexeren Funktionen mit großen Suchräumen effizienter ist. Zusätzlich war die PSO im Vergleich zu den GAs in der Lage, entweder eine gleichwertige oder aber eine signifikant bessere Lösung für alle künstlichen Testobjekte zu finden.

Die Effektivität des Evolutionären Strukturtests konnte durch die Verwendung der PSO als Optimierungskomponente erhöht werden, ebenso wie die Effizienz bei komplexen Problemen. Bei weniger komplexen Optimierungsproblemen wiesen die GAs eine geringfügig höhere Konvergenzgeschwindigkeit auf.

5.3 Experimente mit industriellen Testobjekten

In den vorherigen Abschnitten konnte gezeigt werden, dass die Particle Swarm Optimization für den Evolutionären Strukturtest verwendet werden kann und dabei - zumindest für die künstlich erstellten Testobjekte - Vorteile gegenüber den GAs in Bezug auf die Effizienz bei gleicher Effektivität aufweisen konnte. Dies gilt es auch für reale Testobjekte zu untersuchen.

In den folgenden Abschnitten wird zuerst die Auswahl an verwendeten Testobjekten mitsamt einer kurzen Charakterisierung vorgestellt und anschließend werden die Ergebnisse der durchgeführten Experimente für beide Optimierungsverfahren präsentiert.

Wie in Abschnitt 5.1 angesprochen, gilt das Erreichen des Funktionswertes 0 als Abbruchkriterium für beide Optimierungsverfahren. Da dieses allerdings nicht garantiert werden kann, müssen zusätzlich garantierte Abbruchkriterien definiert werden. Bei der GEA-Toolbox wird dafür die maximale Anzahl an zu erzeugenden Generationen auf 200 festgelegt. Als garantiertes Abbruchkriterium für die PSO-Toolbox wurden 43224 Zielfunktionsevaluationen gewählt, weil die GEA-Toolbox aufgrund ihrer Konfiguration abhängig vom gewählten generation gap des Selektionsoperators innerhalb der maximal möglichen 200 Generationen genau 43224 Aufrufe der Zielfunktion benötigt.

Zur Sicherstellung der statistischen Relevanz wurden für die Experimente alle Teilziele der Testobjekte einzeln von beiden Optimierungskomponenten insgesamt 30 Mal optimiert und die Ergebnisse aller Testläufe der einzelnen Teilziele anschließend gemittelt. Man erhält dadurch also für beide Techniken eine gemittelte Anzahl von Zielfunktionsevaluationen, die zum Erreichen des globalen Optimums notwendig waren und somit einen Überblick über deren Effektivität und Effizienz.

5.3.1 Auswahl

Tabelle 5.4 zeigt die Auswahl an industriellen Testobjekten. Die Größe der Testobjekte reicht von 36 bis 990 Codezeilen, die Anzahl der Zweige von 18 bis 264 und die Größen der zu optimierenden Datenräume umfassen sieben bis 776 Dimensionen. Insgesamt erstrecken sich die durchgeführten Experimente über mehr als 4.600 Zeilen C-Quellcode, verteilt auf 15 Funktionen und über eine Gesamtanzahl von 1126 Zweigen. Diese Funktionen entstammen beispielsweise aktuellen Entwicklungsprojekten von Mercedes und

Testobjekt / Funktion	Codezeilen	Anzahl der Zweige	Anzahl der Variablen
BatteryModel	142	18	48
BrakeAssistant1	405	108	33
BrakeAssistant2	405	108	33
ClassifyTriangle	36	26	8
De-icer1	171	56	12
De-icer2	79	18	12
EmergencyBrake	202	62	31
ICCDeterminer	60	30	7
Ignition	58	32	24
LogicModule	259	70	38
Multimedia	861	264	66
Preprocessor	990	92	33
PrototypingTool	232	46	135
Reaction	310	96	776
Warning	404	100	13

Tabelle 5.4: Übersicht über die getesteten industriellen Testobjekte

Chrysler. Sie unterscheiden sich sowohl in der Länge und der Struktur des Quellcodes als auch der Anzahl und Typen der Variablen. Sowohl *BrakeAssistant1* und *BrakeAssistant2* werden zur Bremskoordination im Bremsassistenzsystem verwendet, während *De-icer1* und *De-icer2* zur Kontrolle der Heizeinheiten der Scheibenenteiser benötigt werden. Das Testobjekt *EmergencyBrake* ist ein Teil eines Notfallbremssystems. *Preprocessor* realisiert Teile der Objektvorverarbeitung, welche zur Situationsanalyse benötigt wird, und *Warning* verwaltet auftretende Warnungen. *ClassifyTriangle* ist eine oft verwendete Testfunktion und führt eine Klassifikation von Dreiecken mittels ihrer Seitenlängen durch. *Multimedia* wird zur Bedienung von Peripheriegeräten und *LogicModule* zur Bestimmung des aktuellen Betriebsmodus des Motors verwendet. *PrototypingTool* steuert zu Testzwecken die Interaktion von neuem Code mit einem Prototyp des Motorsystems. *ICCDeterminer* bestimmt die Gründe, weshalb sich der Tempomat des Fahrzeugs nicht aktivieren lässt. Das Testobjekt *Ignition* bestimmt, welche Zylinder aufgrund der aktuellen Kalibrierung gezündet werden sollen, *BatteryModel* schätzt die Temperatur der Autobatterie aufgrund von äußeren Faktoren. *Reaction* überwacht die möglichen Systemzustände und bestimmt die bei auftretenden Fehlern jeweils zu erwartende Reaktion.

5.3.2 Erwartete Ergebnisse

Aufgrund der Ergebnisse der Experimente mit den künstlichen Testobjekten kann nun auch bei den industriellen Testobjekten eine Effektivitätssteigerung des Evolutionären Strukturtests erwartet werden.

Die Effizienz beider Verfahren wird voraussichtlich ebenfalls problemabhängig sein, d.h. dass die PSO für komplexe Teilziele möglicherweise weniger Zeit in Anspruch nehmen wird als die GAs und umgekehrt für weniger komplexe Teilziele. Abhängig von der Anzahl komplexer und weniger komplexer Teilziele wird sich die Effizienzsteigerung entsprechend auf die Testobjekte als Ganzes übertragen.

5.3.3 Auswertung

Für den Vergleich der Effektivität bezüglich eines zu erreichenden Teilziels wird wie folgt vorgegangen: Ein Teilziel gilt als erreicht, wenn das globale Optimum in allen 30 Läufen gefunden wurde, bevor das garantierte Abbruchkriterium zum Tragen kommt. Die *Erfolgsrate* einer Optimierungskomponente bezüglich eines Testobjektes ist dann die relative Häufigkeit dessen erreichter Teilziele.

Die gemittelte Anzahl der zur Bestimmung einer global optimalen Lösung notwendigen Zielfunktionsevaluationen ist für alle Teilziele aller Testobjekte für beide Optimierungstechniken im Anhang angegeben. Zusätzlich ist für jedes Teilziel die statistische Signifikanz des Unterschieds beider Optimierungsverfahren angegeben. Diese wurde mithilfe eines T-Tests berechnet und ist wie folgt visualisiert: Ein Sternchen über einem Vergleichspaar deutet einen signifikanten Unterschied an (Irrtumswahrscheinlichkeit $< 5\% = 0.05$) und zwei Sternchen signalisieren einen sehr signifikanten Unterschied (Irrtumswahrscheinlichkeit $< 1\% = 0.01$). Ist einer dieser Unterschiede nicht signifikant, so kann dieser auch durch Zufall zustande gekommen sein. In den folgenden Abschnitten wird auf die erhaltenen Ergebnisse der einzelnen Testobjekte eingegangen und diese näher analysiert. Prozentuale Angaben sind dabei stets gerundet.

Für einige Teilziele werden Code-Elemente in Pseudo-Notation dargestellt und die daraus resultierende Zielfunktionslandschaft visualisiert. Dabei ist zu beachten, dass diese nur Ausschnitte der tatsächlich zu optimierenden Quelltexte darstellen. Häufig werden dabei viele geschachtelte einfache Bedingungen ausgelassen, da diese zu Visualisierungszwecken nicht als essentielles Merkmal der zu optimierenden Funktion anzusehen sind. Da für die Visualisierung nur drei Dimensionen zur Verfügung stehen, können jeweils nur zwei Parameter gleichzeitig betrachtet werden. Zum besseren Verständnis der Funktionslandschaft sind die Grafiken jeweils in zwei Auflösungsstufen dargestellt: auf der linken Seite in einer für alle visualisierten Funktionslandschaften gleichen Auflösungsstufe und auf der rechten Seite mit einer genaueren Auflösung, welche die direkte Umgebung des globalen Optimums besonders gut erkennen lässt.

Einige der Testobjekte enthalten Teilziele, welche nicht erreichbar sind. Dies kann aufgrund von logischen Bedingungen oder aber auch aufgrund der Verwendung von Kalibrierungskonstanten der Fall sein. Kalibrierungskonstanten sind feste Werte, die für verschiedene Laufzeitumgebungen gesetzt werden und die zur Ausführung von bestimm-

ten Zweigen des korrespondierenden Kontrollflussgraphen auf einen bestimmten Wert gesetzt werden müssten. Die Bestimmung deren optimaler Belegung zählt nicht zum Optimierungsproblem.

Testobjekt BatteryModel

Für das Testobjekt *BatteryModel* konnten beide Suchverfahren fast identische Resultate liefern. Sie erzielten beide eine Überdeckung von 6%. Sie waren beide lediglich in der Lage eines der insgesamt 18 Teilziele zu erreichen. Dieser sehr geringe Überdeckungsgrad ist in der Unerreichbarkeit dieser Teilziele aufgrund von Kalibrierungskonstanten begründet.

Es ließe sich also lediglich der Vergleich der Effizienz beider Verfahren für das erfolgreich optimierte Teilziel heranziehen. Dieses wurde von den GAs und von der PSO bereits während der Initialisierung gelöst, weshalb sich ein minimaler Vorteil für die PSO ergibt, welcher aus der geringeren Individuenanzahl der PSO resultiert.

Testobjekte BrakeAssistant1 und BrakeAssistant2

Da sich die beiden Testobjekte *BrakeAssistant1* und *BrakeAssistant2* sowohl im Aufbau und in den erzielten Ergebnissen sehr stark ähneln, werden diese hier gemeinsam betrachtet.

Die GAs konnten nur 89% bzw. 87% beider Testobjekte überdecken, während die PSO in beiden Fällen eine Überdeckung von 98% erreichte. Insgesamt konnten 96 bzw. 94 der 108 Teilziele von beiden Verfahren erreicht werden. Zehn bzw. 12 Teilziele wurden nur von der PSO in allen 30 Läufen erreicht und die restlichen zwei Teilziele konnten für beide Testobjekte weder von den GAs noch von der PSO in jedem Lauf erreicht werden. Die PSO stellte somit für dieses Testobjekt das effektivere Suchverfahren dar.

Für jedes Testobjekt konnten insgesamt 44 der 108 Teilziele bereits während der Initialisierung gefunden werden. Acht weitere Teilziele konnten von den GAs schneller gefunden werden, während die PSO für die restlichen 56 Teilziele eine schnellere Konvergenz aufwies. Die PSO benötigte, gemittelt über alle Teilziele, nur 40% der Zielfunktionsaufrufe der GAs. Alle acht Teilziele, welche die GAs schneller erreichen konnte, haben die in Listing 5.2 vereinfacht dargestellte Struktur.

```
function BrakeAssistant (int i1, int i2)
{ ...
  if (i1 == 0)
    if (i2 == 2 || i2 == 3 || i2 == 4 || i2 == 5)
      // Teilziel 20.0
  ...
```

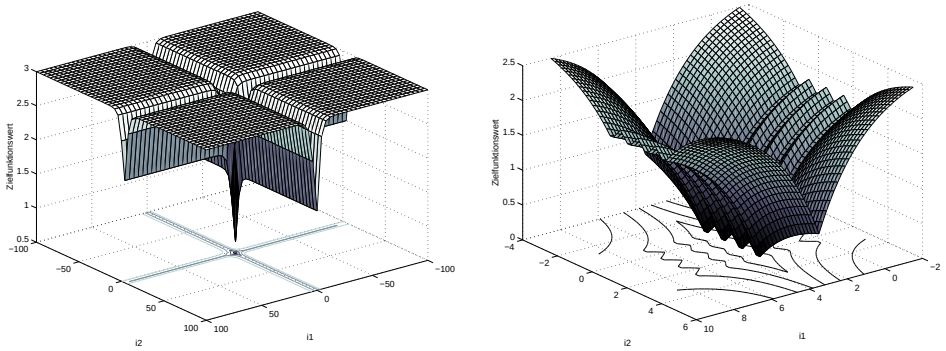

Abbildung 5.6: Die Zielfunktionslandschaft des Teilziels 20.0 des Testobjekts *BrakeAssistant1* für zwei Dimensionen

```
}
```

Listing 5.2: Struktur der Teilziele 20.0, 20.1, 21.0, 21.1, 23.0, 23.1, 68.0, 72.1 des Testobjekts BrakeAssistant1

Diese Teilziele generieren eine Zielfunktionslandschaft, welche der in Abbildung 5.6 dargestellten ähnelt. Sie stellt eine verhältnismäßig einfache Optimierungsfunktion dar, da sie von jedem Punkt im Datenraum aus durch unterschiedliche Fitnesswerte die Richtung zum gesuchten Optimum vorgibt. In der Nähe des Ursprungs befinden sich insgesamt vier lokale Optima, welche gleichzeitig dem globalen Optimum entsprechen und deren Auffindung zum Erreichen des Teilziels ausreicht.

Testobjekt ClassifyTriangle

Beide Suchverfahren konnten das Testobjekt *ClassifyTriangle* gleichermaßen überdecken: Beide erreichten eine Erfolgsrate von 100%. Beide konnten Testdaten ermitteln, welche die Zweige aller 26 Teilziele des korrespondierenden Kontrollflussgraphen zur Ausführung bringt.

Drei der vorhandenen Teilziele wurden von beiden Verfahren bereits während der Initialisierung erreicht, für 12 Teilziele konnten die GAs die Lösung mit einer geringeren Anzahl an Zielfunktionsevaluationen finden, während in den restlichen elf Fällen die PSO eine schnellere Konvergenz aufweisen konnte. Beide Verfahren besitzen demnach eine vergleichbare Effizienz für dieses Testobjekt, wobei die GAs gemittelt über alle Teilziele ungefähr 8% weniger Zielfunktionsaufrufe benötigten. Die Struktur eines Teilziels (21.0), für welches die GAs eine höhere Konvergenzgeschwindigkeit aufweisen konnten, ist in Listing 5.3 dargestellt, die daraus resultierende Zielfunktionslandschaft ist in Abbildung 5.7 visualisiert. Dafür wurde der Parameter c auf 0 gesetzt.

```
function ClassifyTriangle (double a, double b, double c)
{ ...
  if (a != b || b != c)
```

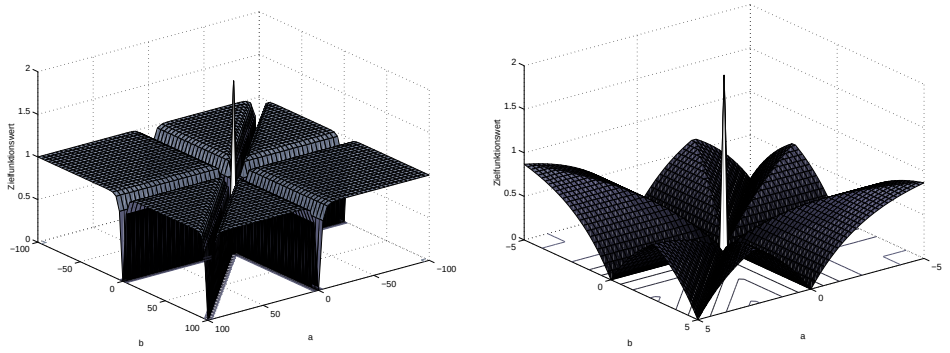

Abbildung 5.7: Die Zielfunktionslandschaft des Teilziels 21.0 des Testobjekts *ClassifyTriangle* für zwei Dimensionen

```
    if (a == c || b == c || a == b)
      // Teilziel 21.0
  ...
}
```

Listing 5.3: Struktur des Teilziels 21.0 des Testobjekts ClassifyTriangle

Die Optimierung dieser Funktion stellt keine hohen Ansprüche an das aktive Suchverfahren, da sie weder lokale Optima besitzt noch über Plateaus verfügt. Dies lässt die Abbildung zwar vermuten, allerdings unterscheiden sich in diesem Fall die Zielfunktionswerte zweier benachbarter Punkte aufgrund der Konstruktion der Zielfunktion voneinander.

Testobjekt De-icer1

Die Effektivität beider getesteter Suchverfahren war für das Testobjekt *De-icer1* gleich. Beide wiesen eine Erfolgsrate von 100% auf, sie konnten jedes der insgesamt 56 Teilziele in allen 30 Läufen erfolgreich optimieren.

Bezüglich der Effizienz kann eindeutig festgestellt werden, dass die PSO den GAs deutlich überlegen war. In den 34 Teilzielen, welche von beiden Techniken bereits während der Initialisierung erreicht werden konnten, benötigte die PSO wie oben beschrieben nur ein Sechstel der Zielfunktionsaufrufe, die von den GAs für die Initialisierung benötigt wurden. Für die restlichen Teilziele benötigte die PSO im Schnitt nur 54% der Zielfunktionsevaluationen der GAs. Es existiert nur ein Teilziel, welches von den GAs schneller gelöst werden konnte als von der PSO: Teilziel 22.0, welches die in Listing 5.4 dargestellte vereinfachte Struktur aufweist.

```
function De-icer1 (int i1, int i2)
{ ...
  if (i1 == 0)
    if (i2 >= 1000)
      // Teilziel 22.0
```

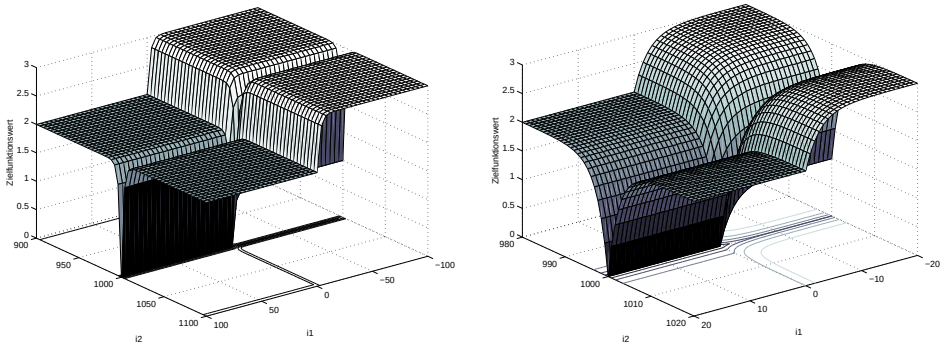

Abbildung 5.8: Die Zielfunktionslandschaft des Teilziels 22.0 des Testobjekts *De-icer1* für zwei Dimensionen

```
  ...
}
```

Listing 5.4: Struktur des Teilziels 22.0 des Testobjekts De-icer1

Die aus diesem Teilziel resultierende Zielfunktionslandschaft ist in Abbildung 5.8 für zwei Dimensionen visualisiert. Sie enthält keine lokalen Optima und ist daher und aufgrund des langen globalen Optimums, welches sich über die Hälfte des Definitionsbereichs erstreckt, sehr einfach zu lösen.

Testobjekt De-icer2

Wie bei Testobjekt *De-icer1* war die Effektivität des Testobjekts *De-icer2* für beide Verfahren gleich. Beide konnten eine Erfolgsrate von 100% erreichen, d.h. sie konnten die globalen Optima der 18 Teilziele in allen Testläufen bestimmen.

Beide Optimierungsverfahren waren gemittelt über alle Teilziele vergleichbar effizient, wobei die PSO dabei nur 95% der Zielfunktionsevaluationen der GAs benötigte. Neben den neun Teilzielen, für welche bereits während der Initialisierung optimale Lösungen gefunden werden konnten, existieren Teilziele, welche einerseits von den GAs und andererseits von der PSO effizienter gelöst werden konnten. Für die Teilziele 6.1, 15.0 und 17.0 konnte von den GAs eine höhere Effizienz erreicht werden. Ersteres hat die in Listing 5.5 dargestellte vereinfachte Struktur während die der beiden letzteren in Listing 5.6 dargestellt ist.

```
function De-icer2 (int i1, int i2)
{ ...
  if (i1 * i1 - i2 - 1000 != 0)
    ...
  else
    // Teilziel 6.1
  ...
```

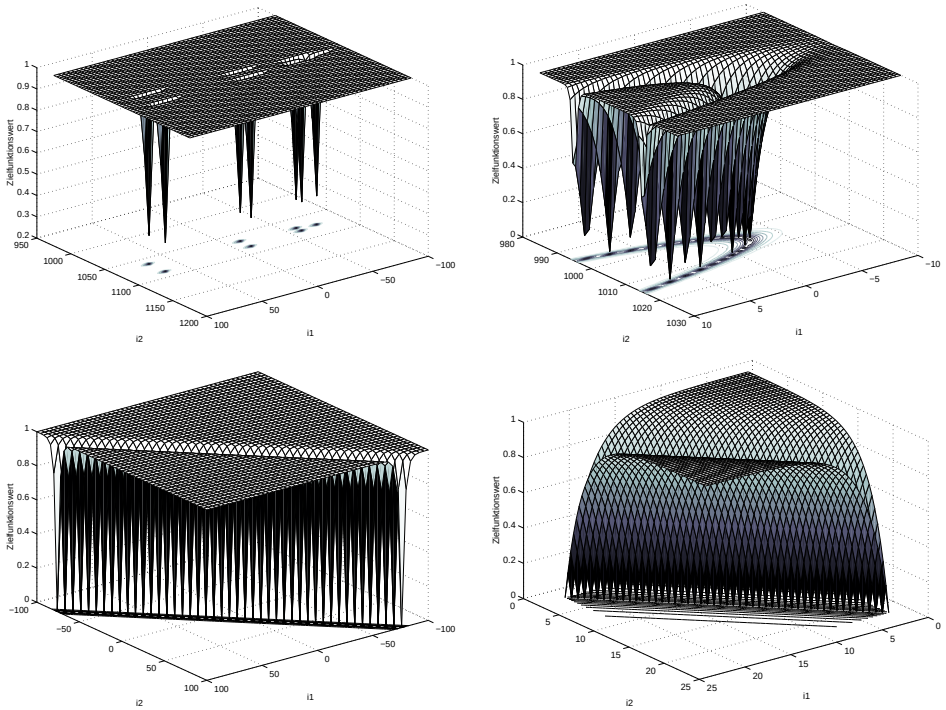

Abbildung 5.9: Die Zielfunktionslandschaft der Teilziele 6.1 und 15.0 des Testobjekts *De-icer2* für zwei Dimensionen, jeweils in zwei unterschiedlichen Auflösungsstufen

```
}
```

Listing 5.5: Struktur des Teilziels 6.1 des Testobjekts De-icer2

```
function De-icer2 (int i1, int i2)
{ ...
  if (i1 + i2 + 1000 == 1024)
    // Teilziel 15.0
  ...
}
```

Listing 5.6: Struktur des Teilziels 15.0 des Testobjekts De-icer2

Die Zielfunktionslandschaften, die aus beiden Strukturen hervorgehen, sind in Abbildung 5.9 dargestellt. Zu beachten ist, dass es dabei zu Darstellungsfehlern gekommen ist, da das verwendete Visualisierungswerkzeug nur die Funktionswerte für Punkte auf einem definierten Gitter berechnen kann. Dadurch werden einzelne globale Optima nicht dargestellt. Die Zielfunktionslandschaft des Teilziels 6.1 besitzt viele elliptische Löcher, welche jeweils eine Vielzahl an globalen Optima enthalten. Diese haben alle den Funktionswert 0 und nicht wie fälschlicherweise dargestellt 0.2 bis 0.5. Die Funktionen sind sehr einfach zu optimieren, da sie weder lokale Minima noch Plateaus besitzen.

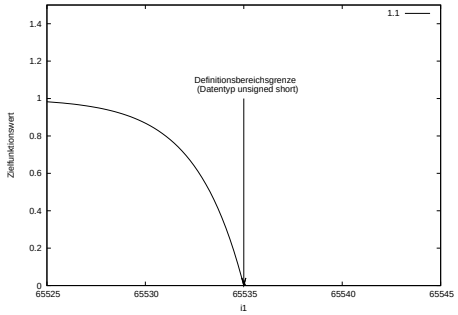

Abbildung 5.10: Die Zielfunktionslandschaft des Teilziels 1.1 des Testobjekts *EmergencyBrake* für zwei Dimensionen

Testobjekt EmergencyBrake

Das Testobjekt *EmergencyBrake* konnte von der PSO mit 61 von insgesamt 62 erreichten Teilzielen besser überdeckt werden als von den GAs, welche nur 53 Teilziele in allen 30 Läufen erreichen konnten. Die PSO war also mit einer Erfolgsrate von 98% für dieses Testobjekt effektiver als die GAs mit einer Erfolgsrate von 85%.

Bezüglich der Effizienz lieferten beide Optimierungskomponenten vergleichbare Ergebnisse. Die PSO war den GAs dabei allerdings leicht überlegen. 29 aller Teilziele wurden von beiden Verfahren während der Initialisierung erreicht, neun Teilziele konnten von den GAs und insgesamt 24 von der PSO effizienter erreicht werden. Gemittelt über alle Teilziele benötigte die PSO 14% weniger Zielfunktionsevaluationen als die GAs. Ein Strukturmerkmal des Teilziels 1.1, welches von den GAs schneller erreicht werden konnte, ist in Listing 5.7 dargestellt. Zur Erfüllung dieses Teilziels muss ein Parameter genau auf die Grenze des Definitionsbereichs gesetzt werden. Da die zu optimierende Variable vom Datentyp *unsigned short* ist, beträgt deren maximaler Wert 65.535.

```
function EmergencyBrake (short i1)
{ ...
  if (i1 < 65.535)
    ...
  else
    // Teilziel 1.1
  ...
}
```

Listing 5.7: Struktur des Teilziels 1.1 des Testobjekts EmergencyBrake

Dieses Teilziel ist in Abbildung 5.10 dargestellt. Es ist sehr einfach zu erreichen, da dessen Zielfunktion keinerlei Schwierigkeiten für die Optimierungskomponente verursacht. Es besitzt weder lokale Optima noch Plateaus, sodass von jedem Punkt des Datenraums aus die Richtung zum globalen Optimum ersichtlich ist. Ein Grund für die geringere Effizienz der PSO gegenüber den GAs für dieses Teilziel kann die Verwendung des Konzepts der

dämpfenden Wände als Definitionsbereichsgrenzen sein, da die Partikel so von diesen imaginären Wänden, welche hier dem gesuchten Optimum entsprechen, abprallen und sich dadurch wieder vom globalen Optimum entfernen können. Es wird vermutet, dass die Verwendung der absorbierenden Wände in diesem speziellen Fall beschleunigend wirken würde. Dies wurde im Rahmen dieser Arbeit jedoch nicht näher untersucht.

Testobjekt ICCDeterminer

Das Testobjekt *ICCDeterminer* konnte von den GAs und von der PSO mit gleicher Effektivität überdeckt werden. Beide Verfahren erreichten eine Erfolgsrate von 67%. Sie konnten beide insgesamt 20 der 30 Teilziele erreichen.

Die Effizienz der GAs und der PSO war dabei ebenfalls vergleichbar. So konnten beide insgesamt 17 Teilziele bereits während der Initialisierung erreichen, während die restlichen drei erfolgreich optimierten Teilziele von der PSO mit einer geringeren Anzahl von benötigten Zielfunktionsaufrufen erreicht werden konnte. Die PSO benötigte gemittelt über alle Teilziele 99% der Zielfunktionsaufrufe der GAs.

Testobjekt Ignition

Für das Testobjekt *Ignition* konnten beide Optimierungskomponenten ähnlich dem Testobjekt *BatteryModel* fast identische Resultate liefern. Sie erzielten beide eine Überdeckung von 19% und waren lediglich in der Lage sechs der insgesamt 32 Teilziele zu erreichen. Dieser geringe Überdeckungsgrad resultiert ebenfalls aus der Unerreichbarkeit dieser Teilziele aufgrund von Kalibrierungskonstanten.

Die sechs erfolgreich optimierten Teilziele konnten jeweils bereits während der Initialisierung erreicht werden, weshalb sich aufgrund der geringeren Individuenanzahl der PSO gemittelt über alle Teilziele bezüglich der Effizienz eine minimale Überlegenheit der PSO gegenüber den GAs ergibt.

Testobjekt LogicModule

Beide Suchverfahren konnten 47 der insgesamt 70 Teilziele des Testobjekts *LogicModule* erreichen. Zusätzlich war die PSO im Gegensatz zu den GAs in der Lage, ein weiteres Teilziel in allen 30 Läufen zu erreichen. Die restlichen Teilziele konnten weder von den GAs noch von der PSO in allen Optimierungsläufen erreicht werden. Die PSO erreichte eine Erfolgsrate von 71% während die GAs eine Erfolgsrate von 70% erreichten.

Es konnten insgesamt 22 Teilziele von beiden Verfahren bereits während der Initialisierung erreicht werden. Für eines der Teilziele benötigten die GAs weniger Zielfunktionsevaluationen als die PSO, welche demgegenüber für die restlichen 25 Teilziele eine

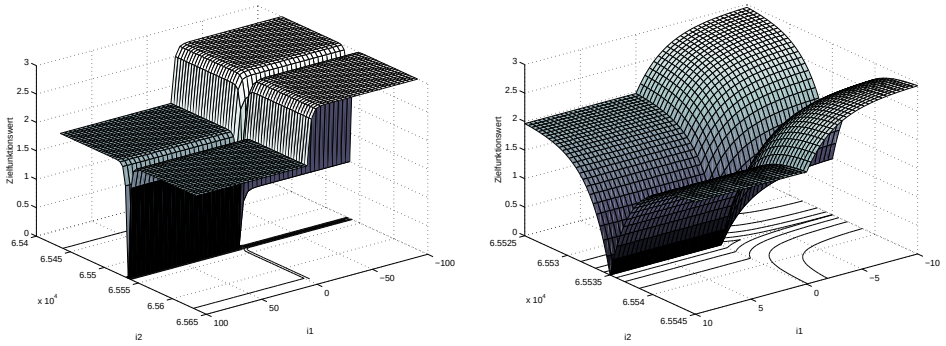

Abbildung 5.11: Die Zielfunktionslandschaft des Teilziels 18.1 des Testobjekts *LogicModule* für zwei Dimensionen

höhere Konvergenzgeschwindigkeit aufweisen konnte. Gemittelt über alle Teilziele benötigte die PSO gegenüber den GAs 5% weniger Zielfunktionsaufrufe. Die Struktur des Teilziels 18.1, für welches die GAs die Lösung schneller finden konnten, ist in Listing 5.8 dargestellt.

```
function LogicModule (short i1, short i2)
{ ...
  if (i1 == 0)
    if (i2 < 65.535)
      ...
    else
      // Teilziel 18.1
  ...
}
```

Listing 5.8: Struktur des Teilziels 18.1 des Testobjekts LogicModule

Die sich aus dieser Struktur ergebende Zielfunktionslandschaft ist in Abbildung 5.11 in zwei Auflösungsstufen dargestellt. Zu beachten ist dabei, dass die zu optimierenden Parameter des Datentyps *unsigned short* einen Maximalwert von 65.535 annehmen können. Das globale Optimum liegt also genau auf der Definitionsbereichsgrenze. Dies kann - wie auch für das Testobjekt EmergencyBrake - der Grund für die geringere Konvergenzgeschwindigkeit der PSO gegenüber den GAs sein.

Testobjekt Multimedia

Die GAs konnten 170 der 254 Teilziele des Testobjekts *Multimedia* erreichen, während die PSO im Gegensatz dazu 203 Teilziele erreichen konnte. Die GAs erreichten damit eine Erfolgsrate von 66% gegenüber der Erfolgsrate von 78% der PSO. Die PSO war demnach für dieses Testobjekt deutlich effektiver.

Insgesamt konnten 110 der 254 Teilziele von beiden Verfahren während der Initialisie-

rung gefunden werden. Die GAs konnten die gesuchte Lösung für drei Teilziele schneller bestimmen als die PSO, welche wiederum in den restlichen 114 erfolgreich optimierten Teilzielen das effizientere Suchverfahren darstellte. Die PSO benötigte gemittelt über alle Teilziele 22% weniger Zielfunktionsevaluationen als die GAs.

Die drei Teilziele, die von den GAs schneller erreicht werden konnten, haben die strukturelle Gemeinsamkeit, keinerlei logische Verknüpfungen aufzuweisen. Sie bestehen lediglich aus untereinander stehenden Bedingungen, welche das Setzen von als boolsche Werte interpretierte Ganzahlen auf 0 oder 1 verlangen. Die daraus resultierenden Zielfunktionen besitzen keine lokalen Optima und sind sehr leicht zu optimieren. Die restlichen Fälle, in denen die PSO eine schnellere Konvergenzgeschwindigkeit aufweisen konnte, besitzen jeweils mindestens eine Bedingung, welche aus logischen Verknüpfungen mehrerer Parameter besteht. Dies führt zu lokalen Optima und somit zu einer Erhöhung der Komplexität der resultierenden Zielfunktion.

Testobjekt Preprocessor

Die PSO erreichte für das Testobjekt *Preprocessor* insgesamt fünf Teilziele, welche die GAs jeweils in mindestens einem der 30 durchgeführten Optimierungsläufen nicht erreichen konnte. Die PSO wies mit einer Erfolgsrate von 87% gegenüber den GAs mit einer Erfolgsrate von 81%, eine höhere Effektivität auf.

Bezüglich der Effizienz war die PSO den GAs ebenfalls überlegen. Sie benötigte, über alle Teilziele gemittelt, 13% weniger Zielfunktionsaufrufe als die GAs. Insgesamt konnten 72 der 108 Teilziele bereits während der Initialisierung erreicht werden. Sieben weitere Teilziele konnten von der PSO mit einer deutlich geringeren Anzahl an Zielfunktionsaufrufen erreicht werden, während die GAs lediglich ein Teilziel effizienter erreichen konnten als die PSO. Die vereinfachte Struktur des zuletzt genannten Teilziels ist in Listing 5.9 dargestellt, die daraus resultierende Zielfunktionslandschaft in Abbildung 5.12.

```
function Preprocessor (int i1, int i2)
{ ...
  if (i1 + i2 > 10.485.760)
    // Teilziel 21.0
  ...
}
```

Listing 5.9: Struktur des Teilziels 21.0 des Testobjekts Preprocessor

Dieser Ausschnitt stellt eine einfach zu optimierende Funktion dar, da fast die Hälfte des Definitionsbereichs eine global optimale Lösung darstellt. Die restlichen Teilziele, für die die PSO eine schnellere Konvergenzgeschwindigkeit aufwies, hängen von Bedingungen mit diversen logischen Verknüpfungen mehrerer Parameter ab.

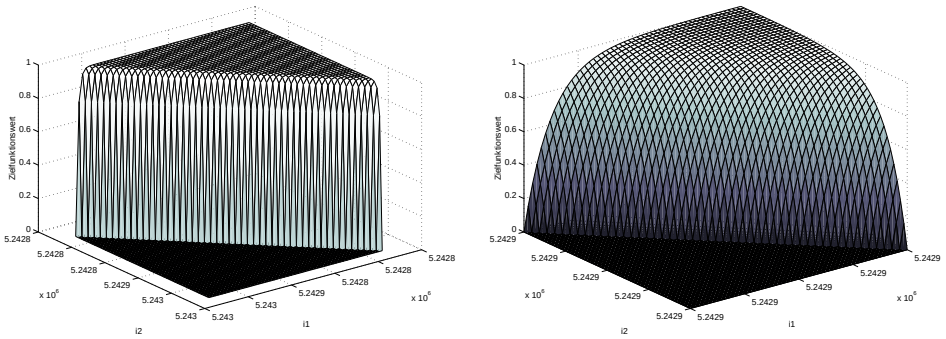

Abbildung 5.12: Die Zielfunktionslandschaft des Teilziels 21.0 des Testobjekts *Preprocessor* für zwei Dimensionen

Testobjekt PrototypingTool

Die GAs konnten für das Testobjekt *PrototypingTool* 40 und die PSO 44 von den insgesamt 46 Teilzielen erreichen. Die beiden restlichen Teilziele sind unerreichbar. Die GAs konnten demnach eine Überdeckung von 87% und die PSO eine Überdeckung von 96% erzielen. Die PSO führte die effektivere Suche durch.

Neben einem Teilziel, welches bereits während der Initialisierung erreicht werden konnte, fand die PSO die gesuchte Lösung in allen Teilzielen bei weniger Zielfunktionsevaluationen und führte so ebenfalls die effizientere Suche durch. Gemittelt über alle Teilziele benötigte die PSO weniger als 45% der Ressourcen, die die GAs für das gleiche Ergebnis benötigten.

Die Teilziele bestehen aus vielen Konjunktionen mehrerer Parameter und weisen ähnliche Strukturen auf wie die Teilziele des Testobjekts Reaction. So lässt sich auch hier vermuten, dass die PSO in der Lage ist, schwer zu optimierende Funktionslandschaften mit einer höheren Diversität zu erkunden.

Testobjekt Reaction

Alle 96 Teilziele des Testobjekts *Reaction* konnten von der PSO erreicht werden, sie erreichte also eine Erfolgsrate von 100%. Die GAs erreichten im Gegensatz dazu nur eine Erfolgsrate von 59%, da sie nur 57 der zu überdeckenden Zweige in allen 30 Testfällen zur Ausführung bringen konnten. Die PSO erwies sich für dieses Testobjekt als das deutlich effektivere Optimierungsverfahren.

Dies spiegelt sich auch im Vergleich der Effizienz beider Verfahren für dieses Testobjekt wider. Insgesamt konnten 57 Teilziele sowohl von den GAs als auch von der PSO bereits während der Initialisierung gelöst werden, was aufgrund der kleineren Individuenanzahl der PSO bereits zu einer höheren Effizienz der PSO gegenüber den GAs führt. Zusätzlich war die PSO in der Lage, die restlichen Teilziele deutlich effizienter zu lösen. Gemittelt

über alle Teilziele benötigte die PSO lediglich 2% der Zielfunktionsaufrufe der GAs.

Die Teilziele bestehen aus grundsätzlich sehr einfach zu optimierenden Strukturen. Da allerdings oft binäre Variablen und/oder diskrete Variablen in den Bedingungen verglichen werden, entstehen für die entsprechenden alternativen Zweige Zielfunktionen, welche durch das Vorhandensein von Plateaus unvorteilhaft für die Optimierung sind, da sie so dem Optimierungsverfahren keinerlei Informationen über die Richtung zur Position des gesuchten Optimums bietet. Es kann daher nur angenommen werden, dass dies ein Grund für die schnellere Konvergenz der PSO ist. Da in diesem Fall die Optimierung zu einer Zufallssuche degradiert, könnte die Natur der PSO für eine höhere Diversität der Suche und eines daraus resultierenden schnelleren Auffindens des gesuchten Optimums verantwortlich sein.

Testobjekt Warning

Alle 100 Teilziele des Testobjekts *Warning* konnten von der PSO erreicht werden. Die GAs konnten demgegenüber davon nur 80 Teilziele erreichen. Die Erfolgsrate der PSO und der GAs beträgt demnach 100 bzw. 80%. Die PSO stellte damit für dieses Testobjekt die deutlich effektivere Optimierungskomponente dar.

Insgesamt konnten 35 der 100 Teilziele bereits während der Initialisierung der PSO bzw. der GAs erreicht werden. Für vier Teilziele benötigten die GAs zur Auffindung der optimalen Lösung geringfügig weniger Zielfunktionsaufrufe während die PSO in den restlichen 61 Fällen eine höhere Effizienz aufweisen konnte. Die PSO benötigte im Schnitt nur etwa 32% der Zielfunktionsevaluationen der GAs und war diesen somit bezüglich der Effizienz deutlich überlegen. Das Teilziel 70.0 konnte von der PSO deutlich schneller erreicht werden, dessen Struktur ist in Listing 5.10 dargestellt.

```
function Warning (int i1, int i2, int i3, bool i4)
{ ...
  if (i1 == 1)
    if (i2 != 3 && i2 != 4 && i2 != 5)
      if (i3 >= 1 || i3 == 0)
        if (i4 < 2 || i4 == 0)
          // Teilziel 70.0
  ...
}
```

Listing 5.10: Teilstruktur des Teilziels 70.0 des Testobjekts Warning

Es besteht vereinfacht aus vier Bedingungen, die für die übergebenen Parameter erreicht werden müssen. Die aus den ersten beiden Bedingungen resultierende Zielfunktionslandschaft ist in Abbildung 5.13 dargestellt, die letzten beiden in Abbildung 5.14.

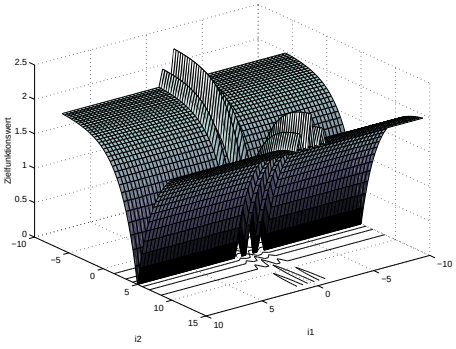

Abbildung 5.13: Ausschnitt der Zielfunktionslandschaft des Teilziels 70.0 des Testobjekts *Warning* für zwei Dimensionen

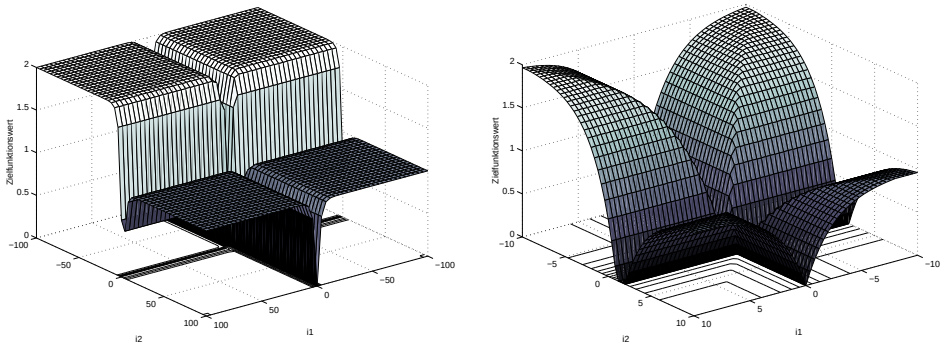

Abbildung 5.14: Ausschnitt der Zielfunktionslandschaft des Teilziels 70.0 des Testobjekts *Warning* für zwei Dimensionen

Zusammenfassung der Testobjekte

Abbildung 5.15 fasst die Überdeckungsgrade aller Teilziele der industriellen Testobjekte für beide Optimierungsverfahren zusammen. Es ist der prozentuale Anteil der von keinem, von beiden oder von nur einem der verwendeten Optimierungsverfahren überdeckten Teilziele dargestellt. Es existieren sechs Testobjekte, für die die GAs und die PSO eine gleiche Anzahl von Teilzielen erreichen konnte: sowohl die Testobjekte *BatteryModel*, *ICCDeterminer* und *Ignition*, für welche von beiden Verfahren aufgrund einer hohen Anzahl von code-bedingt unerreichbaren Zweigen eine sehr geringe Überdeckung erreicht werden konnte, als auch die Testobjekte *ClassifyTriangle*, *De-icer1* und *De-icer2*, für die von beiden Verfahren eine Überdeckung von 100% erreicht wurde. Für die restlichen neun Testobjekte wiesen beide Optimierungstechniken unterschiedliche Überdeckungsgrade auf. So konnten mindestens 60% der Teilziele dieser Testobjekte von beiden Verfahren erreicht werden. Es ist darüber hinaus zu sehen, dass die PSO für all diese Testobjekte in der Lage war, zwischen 1,4% und 40% mehr Teilziele zu erreichen, als die GAs. Umgekehrt existiert jedoch kein Teilziel, welches allein von den GAs in allen 30 Testläufen erreicht wurde. Die PSO wies demnach ausnahmslos eine höhere Effektivität für die Verwendung für den Evolutionären Strukturtest auf. Die Effektivität beider Verfahren ist für die einzelnen Testobjekte in Abbildung 5.16 vergleichend

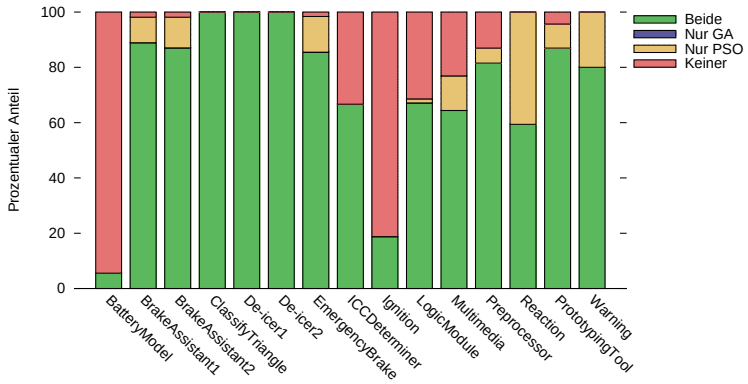

Abbildung 5.15: Überblick über die Teilziele der industriellen Testobjekte, welche von beiden oder keinem Optimierungsverfahren, oder nur von einem der beiden erreicht werden konnte

dargestellt. Die Erfolgsrate entspricht dabei dem prozentualen Anteil der von den GAs

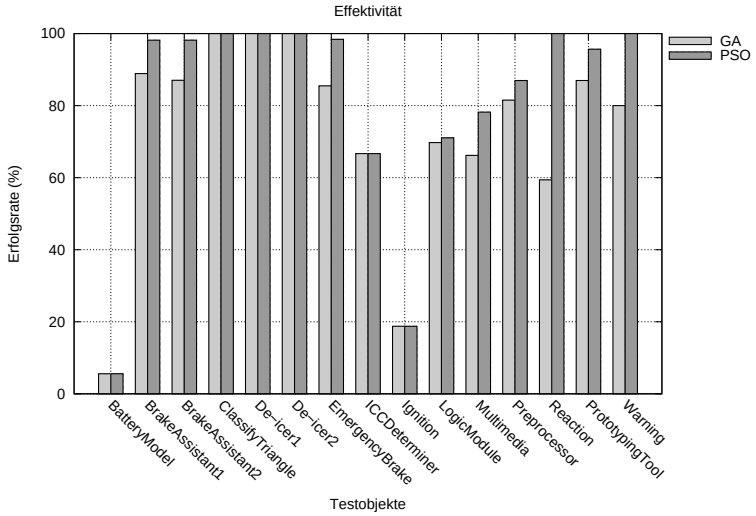

Abbildung 5.16: Experimentelle Ergebnisse für die Effektivität beider Optimierungsverfahren für die getesteten industriellen Testobjekte

bzw. der PSO in allen 30 Testläufen erreichten Teilziele des jeweiligen Testobjekts. Wie bereits mittels des Überdeckungsgrades festgestellt, weist die PSO im Vergleich mit den GAs für diese Testobjekte entweder eine höhere oder aber eine mindestens gleichwertige Effektivität auf.

Abbildung 5.17 fasst die für die untersuchten Testobjekte für beide Optimierungsverfahren gemessene Effizienz zusammen. Diese ist durch die gemittelte Anzahl von Zielfunktionsevaluationen gegeben: Je geringer die Anzahl der benötigten Evaluationen, desto effektiver ist das verwendete Optimierungsverfahren. Für die beiden Testobjekte *BatteryModel* und *Ignition* wurde die geringste Effizienz der GAs und der PSO gemessen, da diese eine Vielzahl an nicht erreichbaren Teilzielen besitzen, für welche beide

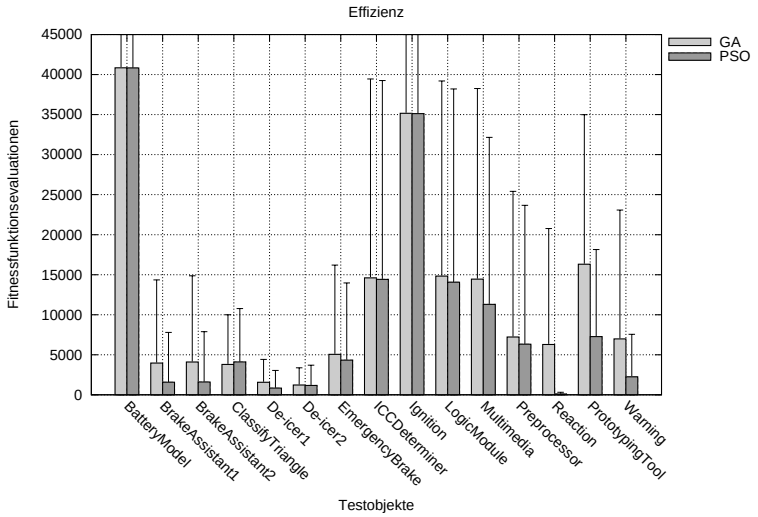

Abbildung 5.17: Experimentelle Ergebnisse für die Effizienz beider Optimierungsverfahren für die getesteten industriellen Testobjekte

Optimierungsverfahren bis zum Eintreten des garantierten Abbruchkriteriums erfolglos Berechnungen durchführen. Die hohen Standardabweichungen der berechneten Mittelwerte resultieren daraus, dass einige Teilziele sehr einfach unter Verwendung einer sehr geringen Anzahl an Zielfunktionsevaluationen und andere sehr schwer oder gar nicht erreicht werden können. Zu sehen ist, dass die GAs der PSO bezüglich der Effizienz in nur einem Testobjekt überlegen waren: für das Testobjekt *ClassifyTriangle*. Für die restlichen 14 Testobjekte konnte die PSO im Vergleich zu den GAs eine höhere Effizienz aufweisen.

Zusammenfassend sind in Tabelle 5.5 die Ergebnisse aller Testobjekte für die beiden Optimierungskomponenten dargestellt. Die Spalten *Testobjekt* und *Optimierung* beschreiben, welches Optimierungsverfahren (GA oder PSO) für welches Testobjekt die angegebenen Resultate erzielen konnte. Die übergreifenden Spalten *Teilziele* und *Testläufe* fassen Ergebnisse für die einzelnen Teilziele des jeweiligen Testobjektes und die dafür ausgeführten Testläufe zusammen. Die Spalte *Anzahl* gibt dabei die Anzahl der Teilziele bzw. Testläufe je Testobjekt an. Die Spalte *Fehlerhaft* zeigt die davon nicht vor Ablauf der gestatteten maximalen Anzahl von Zielfunktionsevaluationen erfolgreich beendeten Teilziele bzw. Testläufe und die *Erfolgsrate* ist der prozentuale Anteil der erfolgreich und nicht erfolgreich zu Ende geführten Optimierungen. Die Effizienz des Optimierungsverfahrens für die Testobjekte ist in einer über alle Teilziele gemittelten Anzahl von Zielfunktionsaufrufen (Spalte *Mittel*) mit der entsprechenden Standardabweichung (Spalte *Std.abw.*) dargestellt. Beispielhaft sollen hier kurz die Ergebnisse der GAs für das Testobjekt BrakeAssistant1 erläutert werden. Der Kontrollflussgraph dieses Testobjekts enthält insgesamt 108 zu erreichende Zweige, woraus sich die Anzahl der vorhandenen Teilziele ergibt. Davon konnten insgesamt 12 Teilziele nicht in allen 30 Testläufen erreicht werden, was einer Erfolgsrate von 88,89% entspricht. Da für jedes

		Teilziele			Testläufe			Effizienz	
Testobjekt	Optimierung	Anzahl	Erfolglos	Erfolgsrate	Anzahl	Erfolglos	Erfolgsrate	Mittel	Std.abw.
BatteryModel	GA	18	17	5,56%	540	510	5,56%	40836,00	40927,87
	PSO	18	17	5,56%	540	510	5,56%	40822,72	40927,23
BrakeAssistant1	GA	108	12	88,89%	3240	151	95,34%	3958,00	10390,42
	PSO	108	2	98,15%	3240	60	98,15%	1570,89	6227,01
BrakeAssistant2	GA	108	14	87,04%	3240	167	94,85%	4092,13	10769,99
	PSO	108	2	98,15%	3240	60	98,15%	1602,05	6280,84
ClassifyTriangle	GA	26	0	100,00%	780	0	100,00%	3794,86	6209,86
	PSO	26	0	100,00%	780	0	100,00%	4101,42	6669,29
De-icer1	GA	56	0	100,00%	1680	0	100,00%	1555,41	2865,92
	PSO	56	0	100,00%	1680	0	100,00%	837,05	2211,33
De-icer2	GA	18	0	100,00%	540	0	100,00%	1226,80	2148,58
	PSO	18	0	100,00%	540	0	100,00%	1163,27	2542,49
EmergencyBrake	GA	62	9	85,48%	1860	69	96,29%	5054,01	11143,68
	PSO	62	1	98,39%	1860	30	98,39%	4325,80	9644,33
ICCDeterminer	GA	30	10	66,67%	900	300	66,67%	14606,64	24838,58
	PSO	30	10	66,67%	900	300	66,67%	14420,22	24829,96
Ignition	GA	32	26	18,75%	960	780	18,75%	35164,50	38485,42
	PSO	32	26	18,75%	960	780	18,75%	35119,90	38483,82
LogicModule	GA	70	23	69,74%	2280	661	71,01%	14820,72	24386,82
	PSO	70	22	71,05%	2280	647	71,62%	14063,37	24140,48
Multimedia	GA	264	93	66,18%	8250	2190	73,45%	14451,06	23800,97
	PSO	264	60	78,18%	8250	1683	79,60%	11301,78	20850,97
Preprocessor	GA	92	17	81,52%	2760	388	85,94%	6520,28	16396,79
	PSO	92	12	86,96%	2760	360	86,96%	5701,55	15603,45
PrototypingTool	GA	46	6	86,96%	1380	83	93,99%	16311,03	18677,48
	PSO	46	2	95,65%	1380	60	95,65%	7269,63	10870,39
Reaction	GA	96	39	59,38%	2880	215	92,53%	6283,13	14491,56
	PSO	96	0	100,00%	2880	0	100,00%	107,71	204,64
Warning	GA	100	20	80,00%	3000	396	86,80%	6980,28	16097,82
	PSO	100	0	100,00%	3000	0	100,00%	2247,62	5316,01

Tabelle 5.5: Zusammenfassung der Ergebnisse der GAs und der PSO für die durchgeführten Experimente mit industriellen Testobjekten

Teilziel 30 Testläufe durchgeführt wurden, beläuft sich die Gesamtanzahl der ausgeführten Testläufe auf 3240, wovon insgesamt 151 nicht vor Eintreten des garantierten Abbruchkriteriums erfolgreich optimiert werden konnten, was wiederum bezüglich der Testläufe einer Erfolgsrate von 95,34% entspricht. Die mittlere Anzahl der zum Erreichen der Teilziele notwendigen Zielfunktionsaufrufe beträgt $3958,00$ mit einer Standardabweichung von $10.390,42$.

Wie bei den Experimenten mit künstlich erstellten Testobjekten, ließ sich auch hier zeigen, dass die PSO eine lohnenswerte Alternative zu den GAs für den Evolutionären Strukturtest ist. Die erwarteten Ergebnisse konnten im Groben bestätigt werden. Die PSO wies im Vergleich mit den GAs für alle Testobjekte eine mindestens gleichwertige Effektivität auf. Für einige Testobjekte war die erreichte Effektivitätssteigerung sehr deutlich. Demgegenüber existierte kein Fall, in dem die GAs der PSO bezüglich der Effektivität der Suche überlegen war. Ferner konnten die Erwartungen übertroffen werden, indem gezeigt wurde, dass die PSO für die untersuchten Testobjekte in 94% der Fälle eine höhere Effizienz aufweisen konnte. Wie bereits bei den künstlichen Testobjekten gezeigt werden konnte, war die Steigerung der Effizienz der PSO gegenüber den GAs problemabhängig: Der Unterschied war größer, wenn die zu optimierenden Probleme komplexer und damit schwieriger zu lösen waren und er war kleiner, wenn es sich um

Probleme geringerer Komplexität handelte. Zudem konnte beobachtet werden, dass die PSO bei Optimierungsproblemen, welche mittels der Zielfunktion keinerlei Informationen über die Lage der zu suchenden Optima bereitstellen, deutliche Vorteile gegenüber den GAs zu haben scheint, da sie vermutlich eine höhere Diversität der Suche aufweisen kann. Um diesbezüglich allerdings genaue Aussagen treffen zu können, bedarf es weiterer Untersuchungen. Ein Großteil der Effizienzunterschiede zwischen den GAs und der PSO für die einzelnen Teilziele der Testobjekte war statistisch signifikant. Ungefähr 80% davon wiesen lediglich eine Irrtumswahrscheinlichkeit von maximal 5% auf. Insgesamt waren 74% der Unterschiede sogar sehr signifikant, d.h. deren Irrtumswahrscheinlichkeit ist kleiner als 1%.

Eine der Intentionen dieser Arbeit war es auch, genaue Fälle, wie beispielsweise bestimmte Codestrukturen oder andere Eigenschaften der zu erreichenden Teilziele ausfindig zu machen, welche für die bessere Leistungsfähigkeit eines der beiden Suchverfahren verantwortlich sind, um diese als Entscheidungskriterium für die Auswahl des für jedes Teilziel am besten geeigneten Suchverfahrens zu verwenden. Leider konnten die durchgeführten Experimente diesbezüglich keine genauen Antworten liefern. So könnten eventuell Software-Maße Anhaltspunkte für diese Entscheidungen liefern, ebenso wie eine genaue Analyse des Charakters der Suchraumerkundung beider Suchverfahren. Dies war jedoch im Rahmen dieser Arbeit nicht mehr möglich.

Zusammenfassend ist die Particle Swarm Optimization sowohl aufgrund der erhöhten Effektivität und Effizienz der Suche als auch aufgrund der einfachen Implementierung für den Evolutionären Strukturtest als Ersatz für die Genetischen Algorithmen besonders zu empfehlen.

Kapitel 6

Zusammenfassung und Ausblick

Diese Arbeit beschäftigte sich mit der Anwendung der Particle Swarm Optimization für die automatisierte Testfallgenerierung des Evolutionären Strukturtests. Dazu wurde zuerst eine Recherche nach zu diesem Zweck geeigneten Varianten der PSO durchgeführt. Die vier ausgewählten Varianten daraufhin anhand von sieben Testfunktionen verglichen, die häufig als Benchmark-Funktionen für Optimierungsverfahren verwendet werden. Aufgrund dieser Untersuchung wurde der CL-PSO als Repräsentant der PSO für den folgenden Vergleich mit den GAs ausgewählt. Der Aufbau und die Funktionsweise des Testsystems wurden vorgestellt. Die vorgenommenen Erweiterungen und Anpassungen beruhen auf einer Architektur, die ein einfaches Austauschen bzw. Umschalten zwischen verschiedenen Optimierungsverfahren ermöglicht. Dieses System wurde anschließend dazu verwendet, vergleichende Experimente mit 25 künstlichen und insgesamt 15 komplexen industriellen Testobjekten durchzuführen, deren Teilziele zu unterschiedlich komplexen Zielfunktionen führten. Sowohl die Particle Swarm Optimization als auch die Genetischen Algorithmen wurden auf dieselben Testobjekte angewandt. Als Vergleichsmetrik wurde dabei jeweils die zur Erreichung des globalen Optimums notwendige Anzahl von Zielfunktionsaufrufen verwendet, da diese für den Strukturtest das übliche Vergleichsmaß darstellt.

Die Ergebnisse der Experimente zeigen, dass die Leistungsfähigkeit der Particle Swarm Optimization vergleichbar mit der Leistungsfähigkeit der Genetischen Algorithmen ist und diese für komplexe Testobjekte sogar übertrifft. Obwohl die Genetischen Algorithmen einige Teilziele für einige Testobjekte schneller erfüllen konnten als die PSO, wies das zuletztgenannte Suchverfahren für den Großteil aller Teilziele eine höhere Konvergenzgeschwindigkeit auf. Allgemein lies sich dabei beobachten, dass die GAs vor allem bei Teilzielen, welche zu einer sehr einfach zu optimierenden Zielfunktion führten, das gesuchte Optimum mit einer geringeren Anzahl an benötigten Zielfunktionsaufrufen auffinden konnten. Demgegenüber war die PSO in der Lage, für jene Teilziele, welche zu einer komplexeren und somit schwieriger zu lösenden Zielfunktion führten, eine höhere Konvergenzgeschwindigkeit aufzuweisen. Das globale Optimum von Zielfunktionen, für welche die Optimierung zu einer Zufallssuche degradierte, konnte von der PSO im

Vergleich zu den GAs ebenfalls nach einer geringeren Anzahl von Zielfunktionsaufrufen gefunden werden. Abbildung 6.1 zeigt das Verhältnis der Leistungsfähigkeit der PSO zur Leistungsfähigkeit der GAs für die durchgeführten industriellen Testobjekte. Zum einen wird der Quotient aus der gemittelten Anzahl der von der PSO und den GAs benötigten Zielfunktionsaufrufe und zum anderen der Quotient aus der gemittelten Erfolgsrate beider dargestellt. Zu sehen ist, dass einerseits das Verhältnis der Erfolgsraten beider

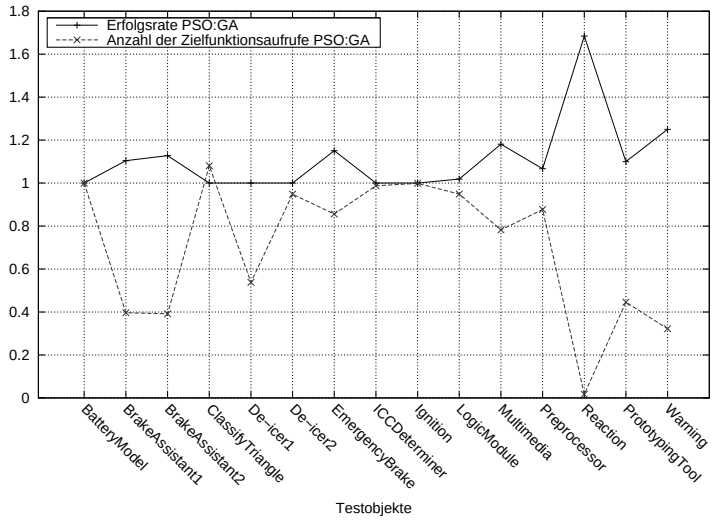

Abbildung 6.1: Die gemittelte Anzahl der benötigten Zielfunktionsaufrufe bzw. die gemittelte Erfolgsrate der PSO im Verhältnis zu denen der GAs.

Verfahren (PSO/GA) für alle Testobjekte mindestens 1 beträgt, d.h. für all diese Testobjekte konnte die PSO im Vergleich zu den GAs eine gleiche oder höhere Erfolgsrate erreichen. Andererseits liegt das Verhältnis der benötigten Zielfunktionsaufrufe bis auf einen Fall fast immer unter 1. Dies bedeutet, dass die PSO für 14 der 15 Testobjekte im Vergleich zu den GAs eine maximal gleich hohe oder geringere Anzahl von Zielfunktionsaufrufen benötigte. Für die durchgeführten Experimente mit industriellen Testobjekten konnte die PSO im Schnitt eine Effektivitätssteigerung um 8% erreichen. Für 60% dieser Testobjekte wies sie eine höhere Effektivität der Suche auf, für die restlichen 40% glich diese der Effektivität der GAs. Bezüglich der Effizienz der Suche lässt sich zusammenfassen, dass die PSO für die untersuchten industriellen Testobjekte im Schnitt nur etwa 71% der für die GAs notwendigen Zielfunktionsaufrufe benötigte. Diese Ergebnisse verdeutlichen, dass die PSO eine attraktive Alternative zu den GAs darstellt, da deren Leistung in Hinblick auf die Effektivität und Effizienz der Suche mindestens genauso gut oder sogar besser ist als die der GAs. Zusätzlich verwendet die PSO einen viel einfacheren Algorithmus mit einer deutlich geringeren Anzahl von Parametern, welche vom Benutzer konfiguriert werden müssen.

Die PSO kann somit aufgrund der erziehlten Ergebnisse als primäres Suchverfahren für den Evolutionären Strukturtest empfohlen werden.

Nichtsdestotrotz sind für allgemeingültigere Aussagen bezüglich der relativen Leistungs-

fähigkeit der Particle Swarm Optimization und der Genetischen Algorithmen für die Verwendung im Kontext des Evolutionären Strukturtests weitere Experimente mit weiteren Testobjekten aus verschiedenen Anwendungsbereichen durchzuführen.

Um einen detaillierten Überblick über die Eignung beider Suchverfahren für die Anwendung in diesem Kontext zu erhalten, müsste ebenso als ein alternativer Vergleichsansatz eine theoretische Analyse beider Suchverfahren durchgeführt werden. Dies würde einen Überblick über die Charakteristika der durchgeführten Suche beider Verfahren geben und könnte bei der Entscheidung, welches Verfahren für welche Art von Zielfunktionen besonders gut geeignet wäre, eine wesentliche Hilfe darstellen. Zusätzlich wären dadurch vermutlich Aussagen über die schnellere Lösungsfindung der PSO bei zur Zufallssuche degradierten Optimierungsproblemen ableitbar.

In der Verwendung der PSO für den Strukturtest ist vermutlicherweise noch ein großes Potential für Verbesserungen auszuschöpfen. So wurden in den Experimenten dieser Arbeit Standardparameter für den CL-PSO verwendet, welche von den Autoren mittels empirischer Untersuchungen für eine Vielzahl von Optimierungsproblemen als optimal bestimmt wurden. Diese müssen allerdings nicht zwangsläufig auch optimale Werte für die Optimierung der für den Strukturtest typischen Zielfunktionen darstellen. So würde die Bestimmung von einer für dieses Anwendungsgebiet optimalen Konfiguration der PSO mutmaßlich in einer zusätzlichen Verbesserung deren Leistungsfähigkeit resultieren. Ferner sollte eine Verwendung des G-PSO näher untersucht werden, da dieser eine signifikant höhere Konvergenzgeschwindigkeit aufweisen konnte als der CL-PSO. Für den Evolutionären Strukturtest wurde er allerdings nicht verwendet, da er zu oft in lokale Optima konvergierte. Eventuell ließe sich dies jedoch durch eine parallele Ausführung mehrerer G-PSO-Schwärme, ähnlich dem Konzept des regionalen Populationsmodells der GAs, verhindern. Alternativ könnte auch die Verwendung eines PSO-Algorithmus' untersucht werden, welcher mehrere Unterschwärme besitzt, die die Suche gemäß unterschiedlicher PSO-Varianten ausführen (z.B. Unterschwarm 1 lernt gemäß G-PSO, Unterschwarm 2 verwendet den CL-PSO, usw.).

Für die industrielle Verwendung der PSO sollten die in dieser Arbeit verwendeten Verfahren und Techniken jedoch erneut, optimiert implementiert werden. Für die im Rahmen dieser Arbeit durchgeführten Vergleichstests stand der möglichst generische und übersichtliche Aufbau der Implementierung im Vordergrund, weshalb dazu die Programmiersprache Java verwendet wurde. Für den industriellen Einsatz wird jedoch viel Wert auf die Geschwindigkeit gelegt. Aus diesem Grund sollte dafür eine maschinennähere Programmiersprache, wie beispielsweise C/C++ verwendet werden.

Die Entwicklung einer Meta-Heuristik, welches für jedes Teilziel separat entscheidet, welche das für dieses Teilziel optimale Suchverfahren ist, ist besonders interessant. Dies könnte durch Klassifikationsalgorithmen unter Verwendung von Software-Maßen unterstützt werden. Dazu sollten allerdings nicht nur die PSO und GAs als relevante Suchverfahren in Betracht gezogen werden, vielmehr sollte die Auswahl an Suchverfahren

entsprechend erweitert werden, um ein größeres Spektrum an Suchcharakteristiken abdecken zu können. So sollten beispielsweise Verfahren wie *hill climbing*, *simulated annealing* oder auch die Zufallssuche mit einbezogen werden.

Anhang A
Quellcode

A.1 Quellcode der künstlichen Testobjekte

```
int fun_b_01v(bool a)
{
        if (a)
                return 1;
        return 0;
}
```

Listing A.1: Testobjekt f01

```
int fun_i_01v(int a)
{
        if (a == 1)
                return 1;
        return 0;
}
```

Listing A.2: Testobjekt f02

```
int fun_d_01v(double a)
{
        if (a == 1.5)
                return 1;
        return 0;
}
```

Listing A.3: Testobjekt f03

```
int fun_1lO_b_01v(bool a)
{
        if ((a || !a) && a)
                return 1;
        return 0;
```

```
}
```

Listing A.4: Testobjekt f04

```
int fun_1lO_i_01v(int a)
{
        if ((a == 0 || a == 1) && a == 0)
                return 1;
        return 0;
}
```

Listing A.5: Testobjekt f05

```
int fun_1lO_d_01v(double a)
{
        if ((a == 0.5 || a == 1.5) && a == 0.5)
                return 1;
        return 0;
}
```

Listing A.6: Testobjekt f06

```
int fun_4lO_i_01v(int a)
{
        if ((a == -2 || a == -1 || a == 0 ||
             a == 1 ||  a == 2) && a == 0)
                return 1;
        return 0;
}
```

Listing A.7: Testobjekt f07

```
int fun_4lO_d_01v(double a)
{
        if ((a == -2.5 || a == -1.5 || a == 0 ||
             a == 1.5 ||  a == 2.5) && a == 0)
                return 1;
        return 0;
}
```

Listing A.8: Testobjekt f08

```
int fun_b_02v(bool a, bool b)
{
        if (a && b)
                return 1;
        return 0;
}
```

Listing A.9: Testobjekt f09

```
int fun_i_02v(int a, int b)
{
        if ((a == 1) && (b == 1))
                return 1;
        return 0;
}
```

Listing A.10: Testobjekt f10

```
int fun_d_02v(double a, double b)
{
        if ((a == 1.5) &&
            (b == 1.5))
                return 1;
        return 0;
}
```

Listing A.11: Testobjekt f11

```
int fun_24lO_i_02v(int a, double b)
{
        if (((a == -2 || a == -1 || a == 0 ||
              a == 1 || a == 2) && a == 0) &&
            ((b == -2.5 || b == -1.5 || b == 0 ||
              b == 1.5 || b == 2.5) && b == 0))
                return 1;
        return 0;
}
```

Listing A.12: Testobjekt f12

```
int fun_b_03v(bool a, bool b, bool c)
{
        if (a && b && c)
                return 1;
        return 0;
}
```

Listing A.13: Testobjekt f13

```
int fun_i_03v(int a, int b, int c)
{
        if ((a == 1) && (b == 1) && (c == 1))
                return 1;
        return 0;
}
```

Listing A.14: Testobjekt f14

```
int fun_d_03v(double a, double b, double c)
{
        if ((a == 1.5) &&
            (b == 1.5) &&
            (c == 1.5))
                return 1;
        return 0;
}
```

Listing A.15: Testobjekt f15

```
int fun_bid_03v(bool a, int b, double c)
{
        if ((a) &&
            (b == 1) &&
            (c == 1.5))
                return 1;
        return 0;
}
```

Listing A.16: Testobjekt f16

```
int fun_vlO_i_03v(bool a, int b, double c)
{
        if (((a || !a) && a) &&
       ((b == -2 || b == -1 || b == 0 ||
         b == 1 || b == 2) && b == 0) &&
            ((c == -2.5 || c == -1.5 || c == 0 ||
              c == 1.5 || c == 2.5) && c == 0))
                return 1;
        return 0;
}
```

Listing A.17: Testobjekt f17

```
int fun_b_06v(bool a, bool b, bool c, bool d, bool e, bool f)
{
        if (a && b && c && d && e && f)
                return 1;
        return 0;
}
```

Listing A.18: Testobjekt f18

```
int fun_i_06v(int a, int b, int c, int d, int e, int f)
{
        if ((a == 1) && (b == 1) && (c == 1) &&
            (d == 1) && (e == 1) && (f == 1))
```

```
                return 1;
        return 0;
}
```

Listing A.19: Testobjekt f19

```
int fun_d_06v(double a, double b, double c,
              double d, double e, double f)
{
        if ((a == 1.5) && (b == 1.5) && (c == 1.5) &&
            (d == 1.5) && (e == 1.5) && (f == 1.5))
                return 1;
        return 0;
}
```

Listing A.20: Testobjekt f20

```
int fun_bid_06v(bool a, int b, double c, bool d, int e, double f)
{
        if ((a) && (b == 1) && (c == 1.5) &&
            (d) && (e == 1) && (f == 1.5))
                return 1;
        return 0;
}
```

Listing A.21: Testobjekt f21

```
int fun_b_12v(bool a, bool b, bool c, bool d, bool e, bool f,
              bool g, bool h, bool i, bool j, bool k, bool l)
{
        if (a && b && c && d && e && f &&
            g && h && i && j && k && l)
                return 1;
        return 0;
}
```

Listing A.22: Testobjekt f22

```
int fun_i_12v(int a, int b, int c, int d, int e, int f,
              int g, int h, int i, int j, int k, int l)
{
        if ((a == 1) && (b == 1) && (c == 1) && (d == 1) &&
            (e == 1) && (f == 1) && (g == 1) && (h == 1) &&
            (i == 1) && (j == 1) && (k == 1) && (l == 1))
                return 1;
        return 0;
}
```

Listing A.23: Testobjekt f23

```
int fun_d_12v(double a, double b, double c, double d,
              double e, double f, double g, double h,
              double i, double j, double k, double l)
{
        if ((a == 1.5) && (b == 1.5) && (c == 1.5) && (d == 1.5) &&
            (e == 1.5) && (f == 1.5) && (g == 1.5) && (h == 1.5) &&
            (i == 1.5) && (j == 1.5) && (k == 1.5) && (l == 1.5))
                return 1;
        return 0;
}
```

Listing A.24: Testobjekt f24

```
int fun_bid_12v(bool a, int b, double c, bool d, int e, double f,
                bool g, int h, double i, bool j, int k, double l)
{
        if ((a) && (b == 1) && (c == 1.5) &&
            (d) && (e == 1) && (f == 1.5) &&
            (g) && (h == 1) && (i == 1.5) &&
            (j) && (k == 1) && (l == 1.5))
                return 1;
        return 0;
}
```

Listing A.25: Testobjekt f25

Anhang B
Ergebnisse der Experimente

B.1 Industrielle Testobjekte

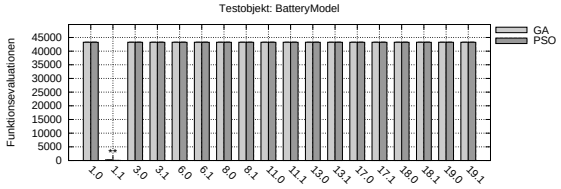

Abbildung B.1: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *BatteryModel*

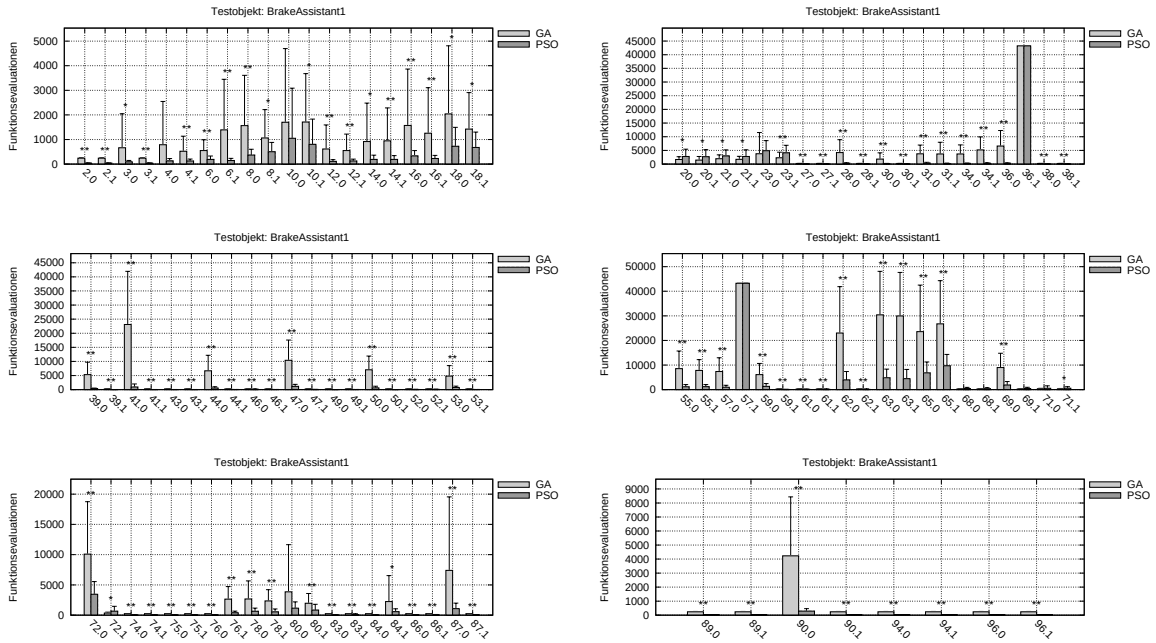

Abbildung B.2: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *BrakeAssistant1*

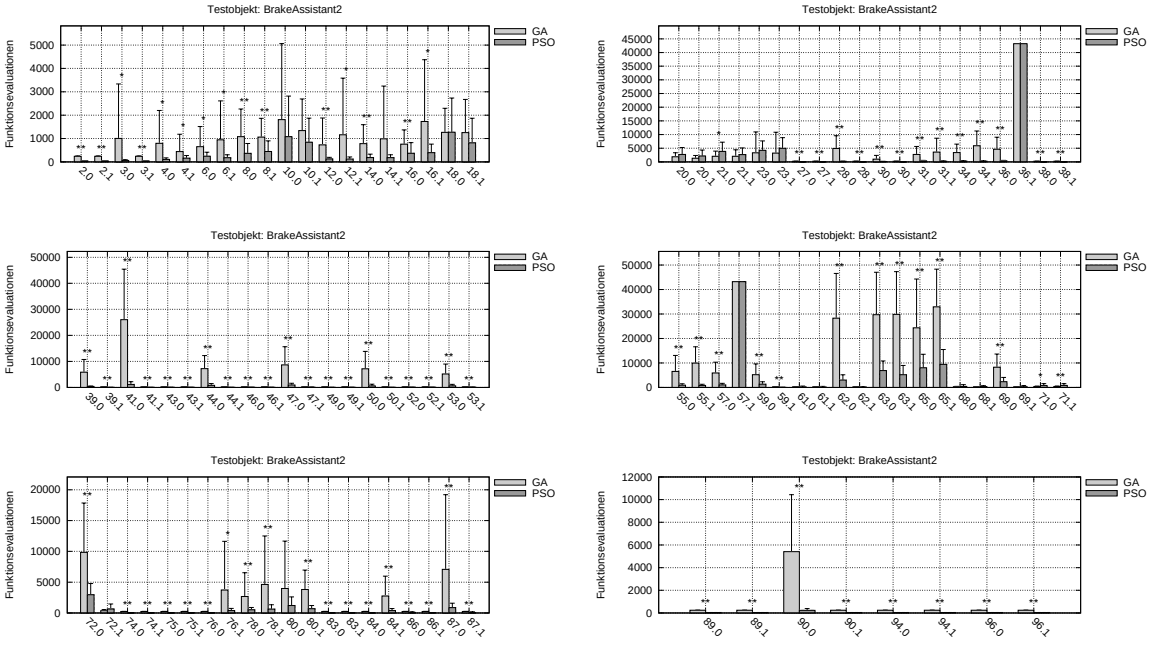

Abbildung B.3: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *BrakeAssistant2*

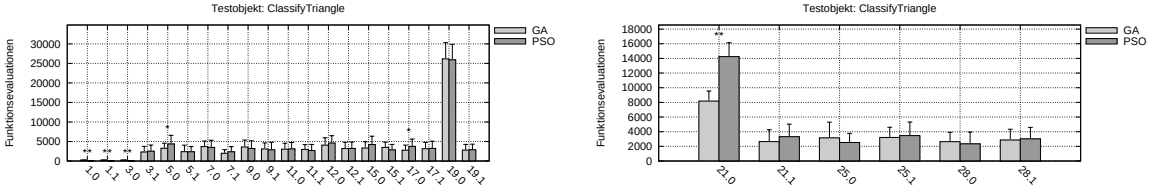

Abbildung B.4: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *ClassifyTriangle*

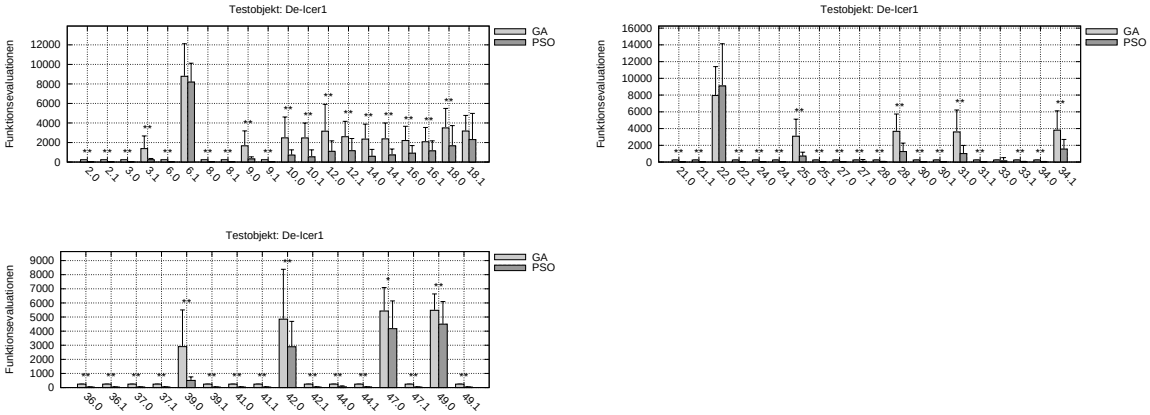

Abbildung B.5: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *De-icer1*

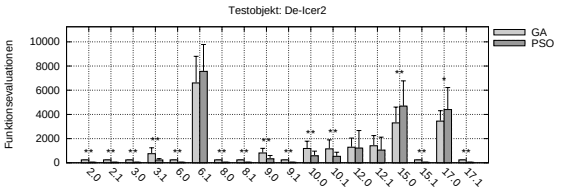

Abbildung B.6: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *De-icer2*

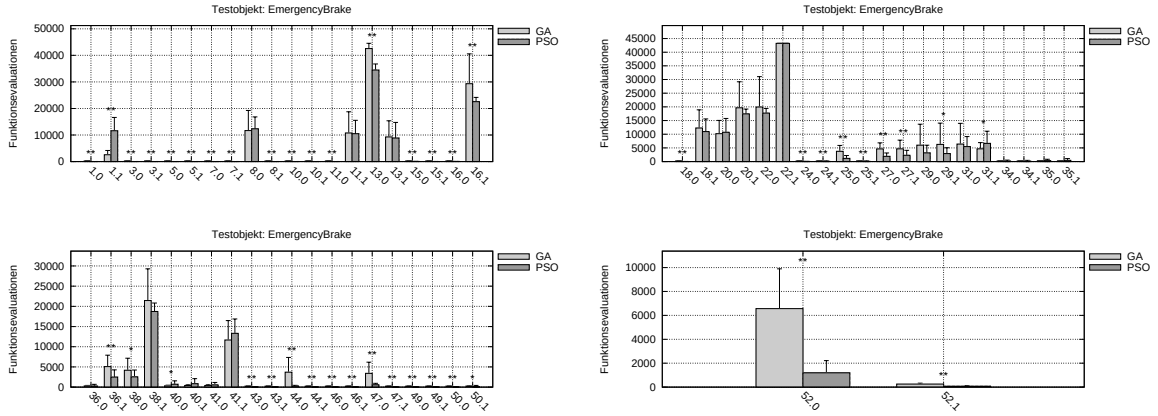

Abbildung B.7: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *EmergencyBrake*

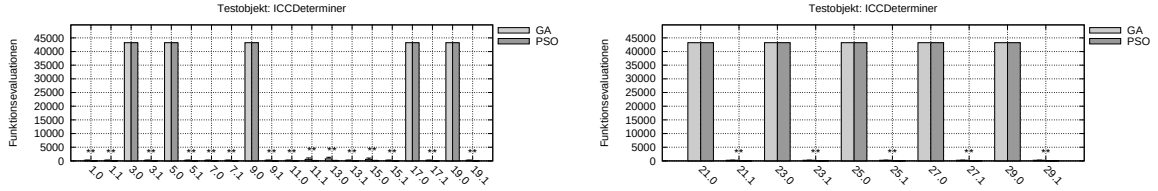

Abbildung B.8: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *ICCDeterminer*

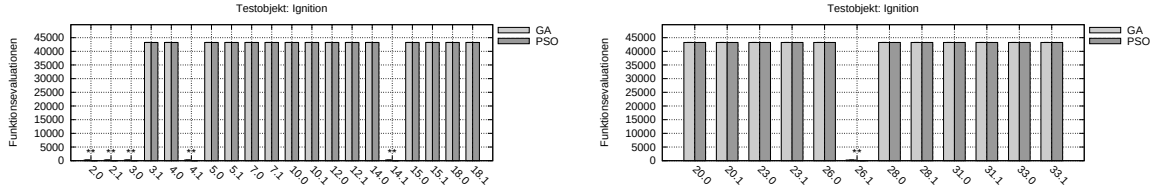

Abbildung B.9: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *Ignition*

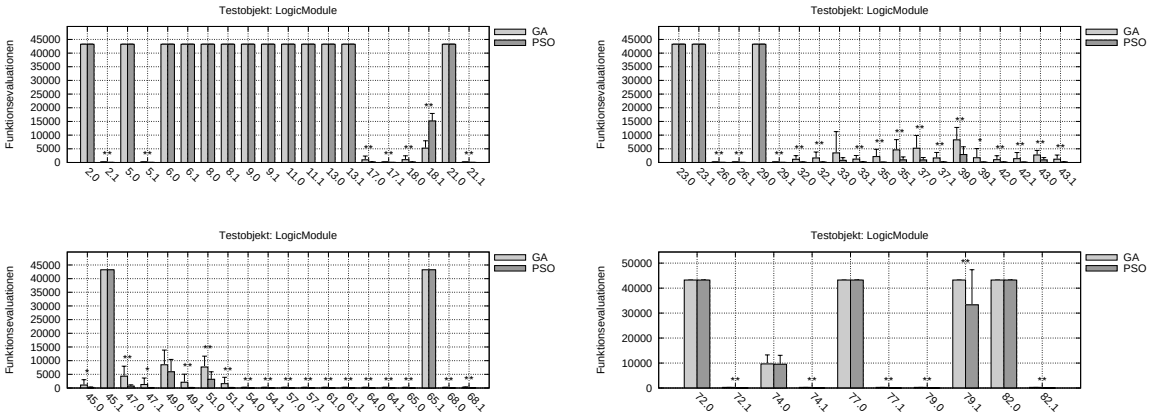

Abbildung B.10: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *LogicModule*

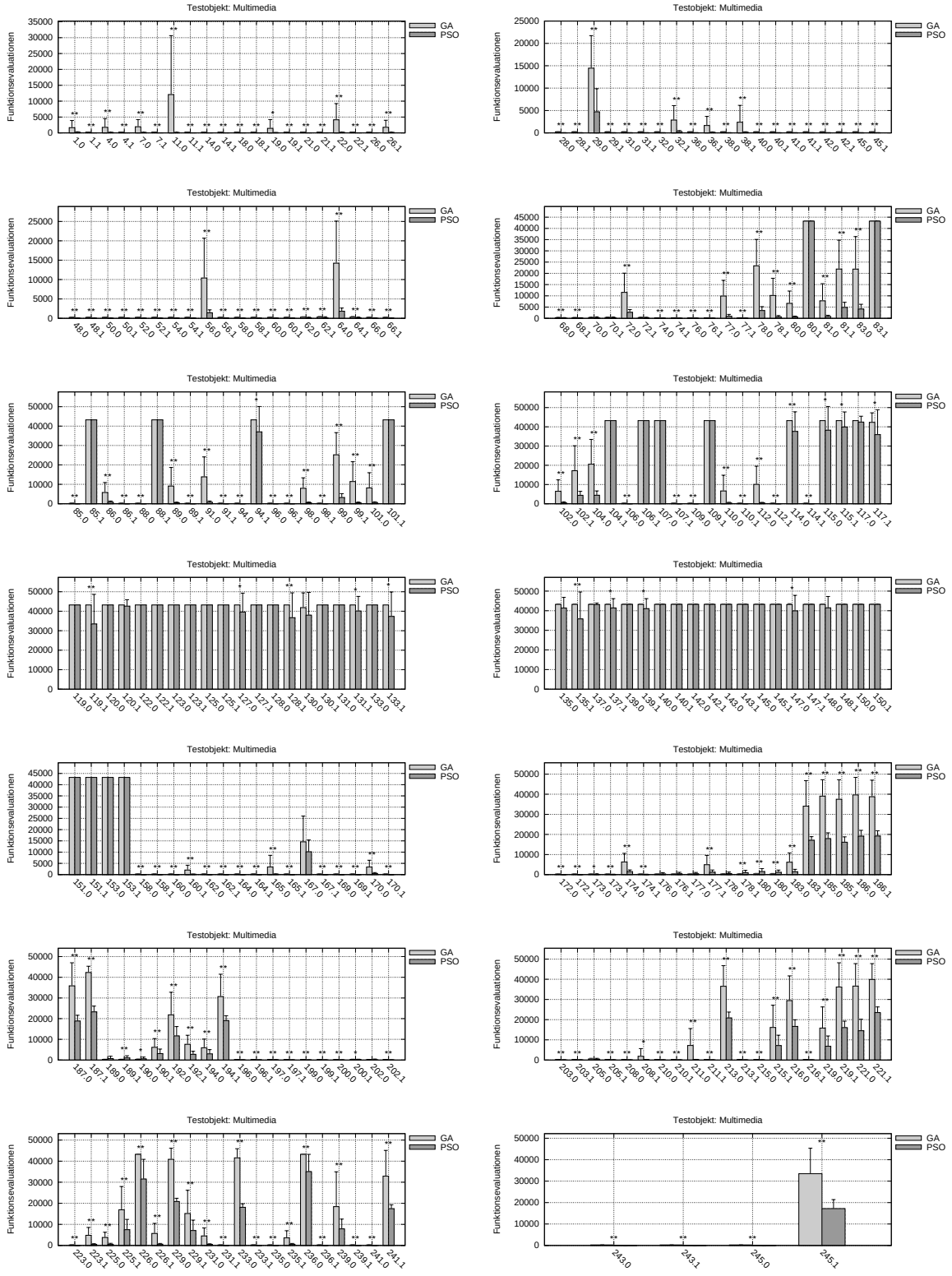

Abbildung B.11: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *Multimedia*

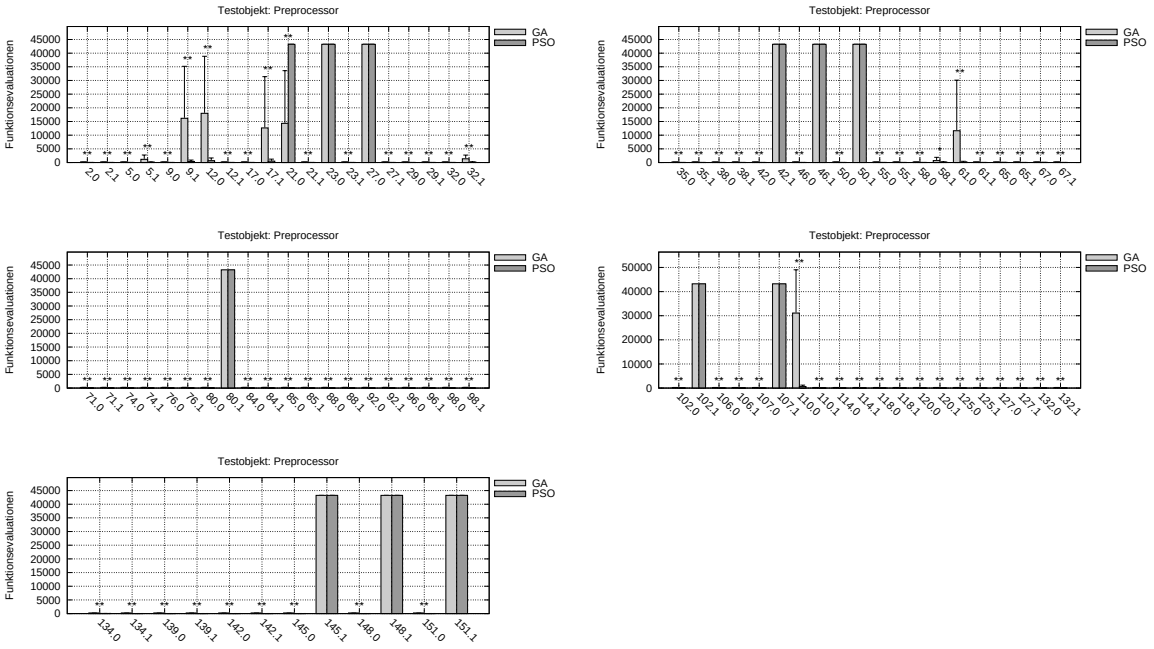

Abbildung B.12: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *Preprocessor*

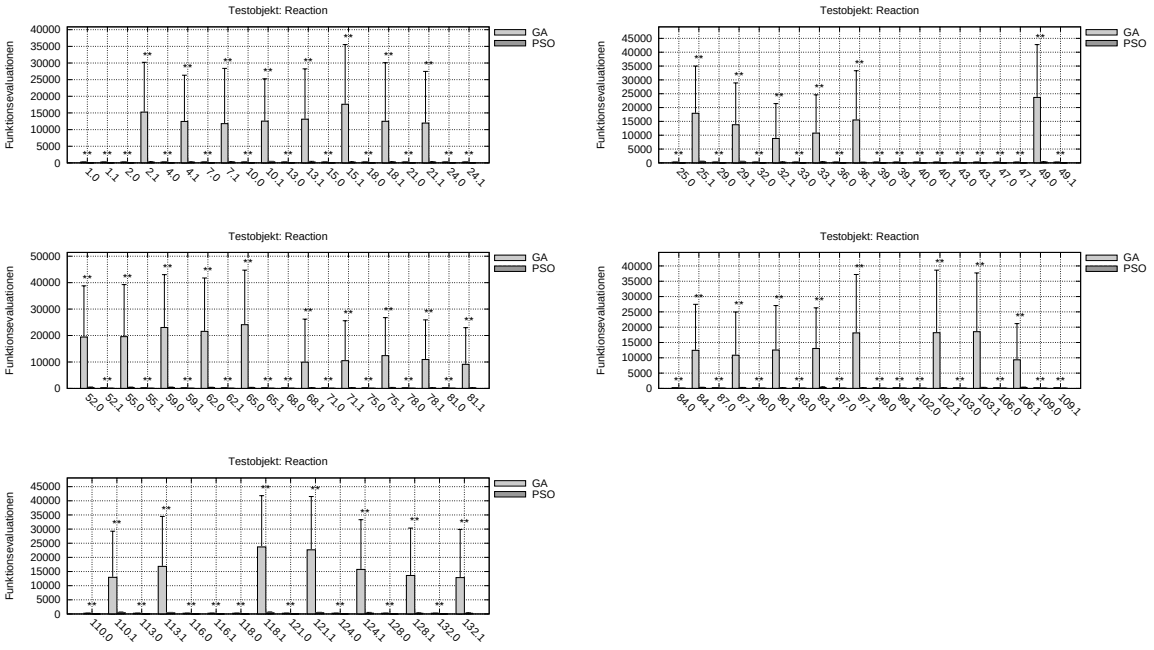

Abbildung B.13: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *Reaction*

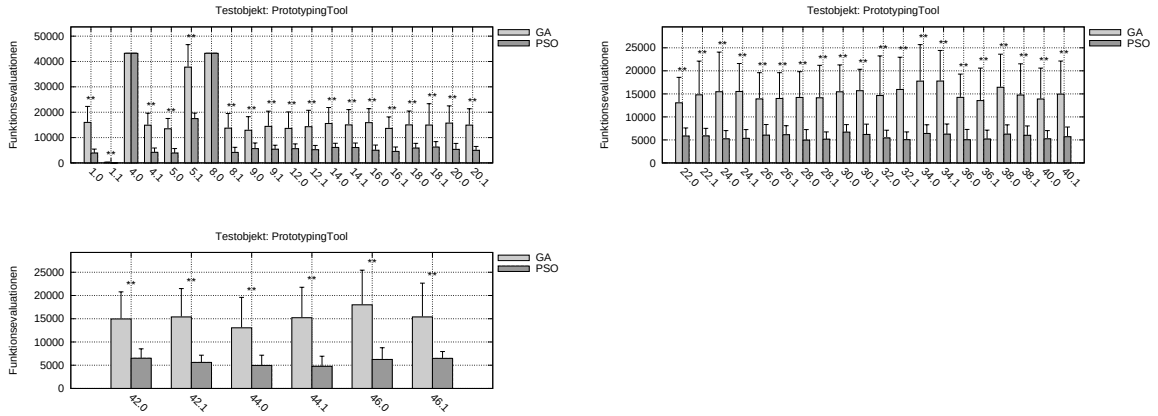

Abbildung B.14: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *PrototypingTool*

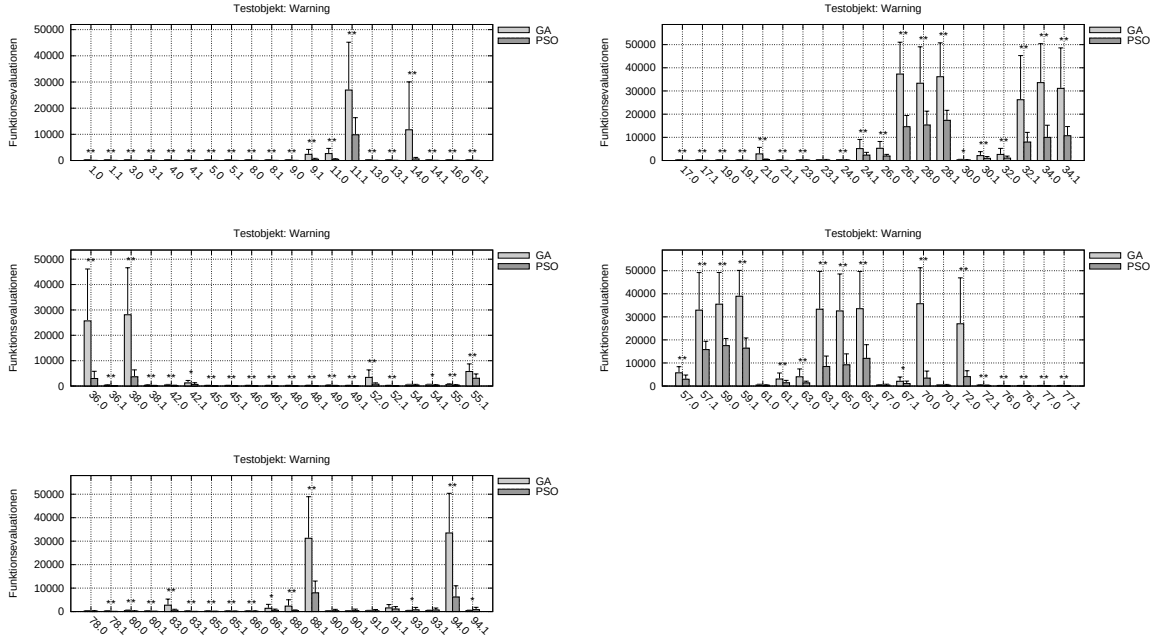

Abbildung B.15: Durchschnittliche Anzahl der benötigten Zielfunktionsevaluationen aller Teilziele des Testobjekts *Warning*

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Literaturverzeichnis

[Bak87] Baker, James E.: Reducing bias and inefficiency in the selection algorithm. In: *Proceedings of the Second International Conference on Genetic Algorithms and their application*. Mahwah, NJ, USA : Lawrence Erlbaum Associates, Inc., 1987. – ISBN 0–8058–0158–8, S. 14–21

[Bar00] Baresel, André: *Automatisierung von Strukturtests mit evolutionären Algorithmen*. Berlin, Humboldt Universitaet, Diplomarbeit, Juli 2000

[Bey01] Beyer, Hans-Georg: *The theory of Evolution Strategies*. Berlin : Springer, 2001

[BS03] Baresel, A. ; Sthamer, H.: Evolutionary Testing of Flag Conditions. In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2003, S. 2442–2454. – July 2003

[CK02] Clerc, Maurice ; Kennedy, James: The particle swarm - explosion, stability, and convergence in a multidimensional complex space. In: *IEEE Trans. Evolutionary Computation* 6 (2002), Nr. 1, S. 58–73

[CW04] Clow, Brian ; White, Tony: An Evolutionary Race: A Comparison of Genetic Algorithms and Particle Swarm Optimization for Training Neural Networks. In: *Proceedings of the International Conference on Artificial Intelligence, IC-AI* Bd. 2, 2004, S. 582–588

[DKC05] Das, Swagatam ; Konar, Amit ; Chakraborty, Uday K.: Improving particle swarm optimization with differentially perturbed velocity. In: *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, 2005, S. 177–184

[EK95] Eberhart, R. C. ; Kennedy, J.: A new optimizer using particle swarm theory. In: *Proceedings of the 6th International Symposium on Micromachine Human Science*, 1995, S. 39–43

[ES98] Eberhart, Russell C. ; Shi, Yuhui: Comparison between Genetic Algorithms and Particle Swarm Optimization. In: *EP '98: Proceedings of the*

7th International Conference on Evolutionary Programming VII. London, UK : Springer-Verlag, 1998. – ISBN 3–540–64891–7, S. 611–616

[ES03] Eiben, A. E. ; Smith, J. E.: *Introduction to Evolutionary Computing (Natural Computing Series)*. Springer, 2003. – ISBN 3–540–40184–9

[FOW66] Fogel, L. J. ; Owens, A. J. ; Walsh, M. J.: *Artificial Intelligence through Simulated Evolution*. New York, USA : John Wiley, 1966

[GD90] Goldberg, David E. ; Deb, Kalyanmoy: A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In: *FOGA*, 1990, S. 69–93

[HCW05] Hassan, R. ; Cohanim, B. ; Weck, O. de: A Comparison of Particle Swarm Optimization and the Genetic Algorithm. In: *Proceedings of the 46th Structural Dynamics and Materials Conference*, 2005

[HE02] Hu, X. ; Eberhart, R.: Multiobjective Optimization Using Dynamic Neighborhood Particle Swarm Optimization. In: *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, 2002, S. 1677–1681

[HM05] Huang, Tony ; Mohan, Ananda S.: A hybrid boundary condition for robust particle swarm optimization. In: *Antennas and Wireless Propagation Letters* 4 (2005), S. 112–117

[Hod02] Hodgson, R. J. W.: Partical Swarm Optimization Applied To The Atomic Cluster Optimization Problem. In: *GECCO*, 2002, S. 68–73

[Hol62] Holland, John H.: Outline for a Logical Theory of Adaptive Systems. In: *Journal of the Association for Computing Machinery* 9 (1962), Nr. 3, S. 297–314

[Hol75] Holland, J.H.: *Adaption in Natural and Artificial Systems*. Ann Arbor, Michigan : University of Michigan Press, 1975

[JC06] Jian, Ming chung ; Chen, Ying ping: Introducing recombination with dynamic linkage discovery to particle swarm optimization. In: *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 2006, S. 85–86

[JHR06] J. Horák, L. O. P. Chmela C. P. Chmela ; Raida, Z.: Global Optimization of the Dual-Band Planar Antenna: PSO versus GA. In: *Radioelektronika*, 2006

[JJLB06] Jing J. Liang, Ponnuthurai N. S. A. Kai Qin Q. A. Kai Qin ; Baskar, S.: Comprehensive learning particle swarm optimizer for global optimization of multimodal functions. In: *IEEE Trans. Evolutionary Computation* 10 (2006), Nr. 3, S. 281–295

[Jon05] Jones, K. O.: Comparison of Genetic Algorithm and Particle Swarm Optimization. In: *Proceedings of the International Conference on Computer Systems and Technologies*, 2005

[JSE96] Jones, B. F. ; Sthamer, H. ; Eyres, D. E.: Automatic Test Data Generation using Genetic Algorithms. In: *Software Engineering Journal* 11 (1996), September, Nr. 5, S. 299–306

[JTR05] Jason Tillett, Ferat S. T. M. Rao R. T. M. Rao ; Rao, Raghuveer M.: Darwinian Particle Swarm Optimization. In: *Proceedings of the 2nd Indian International Conference on Artificial Intelligence*, 2005, S. 1474–1487

[KE95] Kennedy, J. ; Eberhart, R.: Particle swarm optimization. In: *Neural Networks, 1995. Proceedings., IEEE International Conference on* Bd. 4, 1995, 1942–1948 vol.4

[KE01] Kennedy, James ; Eberhart, Russell C.: *Swarm intelligence*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001. – ISBN 1–55860–595–9

[Ken99] Kennedy, James: Small worlds and mega-minds: effects of neighborhood topology. In: *1999 Congress on Evolutionary Computation*, 1999, S. 1931–1938

[KM02] Kennedy, J. ; Mendes, R.: Population Structure and Particle Swarm Performance. In: *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, IEEE Press, 2002, S. 1671–1676

[Koz92] Koza, John R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, 1992. – ISBN 0262111705

[Lam05] Lammermann, Frank: *Dimensionierung quantitativer Abbruchkriterien für den evolutionären Strukturtest mit Hilfe von Software-Maßen*, Humboldt-Universität Berlin, Diss., 2005

[Lig90] Liggesmeyer, Peter: *Modultest und Modulverifikation - State of the art*. Mannheim [u.a.] : BI-Wiss.-Verl., 1990 (Angewandte Informatik; 4). – ISBN 3–411–14361–4

[MSV93] Mühlenbein, Heinz ; Schlierkamp-Voosen, Dirk: Predictive models for the breeder genetic algorithm, I.: continuous parameter optimization. In: *Evol. Comput.* 1 (1993), Nr. 1, S. 25–49. – ISSN 1063–6560

[PB06] Pasupuleti, Srinivas ; Battiti, Roberto: The gregarious particle swarm optimizer (G-PSO). In: *GECCO '06: Proceedings of the 8th annual con-*

ference on Genetic and evolutionary computation. New York, NY, USA : ACM Press, 2006. – ISBN 1–59593–186–4, S. 67–74

[PHP99] Pargas, R. P. ; Harrold, M. J. ; Peck, R. R.: Test-data Generation Using Genetic Algorithms. In: *Journal of Software Testing, Verification and Reliability* 9 (1999), Nr. 4, S. 263–282

[Poh00] Pohlheim, Hartmut: *Evolutionäre Algorithmen : Verfahren, Operatoren und Hinweise für die Praxis*. Berlin ; Heidelberg [u.a.] : Springer, 2000. – XIV, 317 S. : Ill., graph. Darst. + 1 CD–ROM (12 cm). – ISBN 3–540–66413–0

[RRS04] Robinson, Jacob ; Rahmat-Samii, Yahya: Particle Swarm Optimization in Electromagnetics. In: *IEEE Transactions on Antennas and Propagation* 52 (February 2004), Nr. 2, S. 397–407

[Sch65] Schwefel, Hans-Paul: *Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik*, Technische Universität Berlin, Hermann Föttinger–Institut für Strömungstechnik, Diplomarbeit, März 1965

[Sch97] Schwehm, Markus: *Globale optimierung mit massiv parallelen genetischen Algorithmen*, Universität Erlangen-Nürnberg, Institut für Matematische Maschinen und Datenverarbeitung (Informatik), Diss., 1997

[SE98a] Shi, Yuhui ; Eberhart, Russell C.: A modified particle swarm optimizer. In: *IEEE Congress on Evolutionary Computation*, 1998, S. 69–73

[SE98b] Shi, Yuhui ; Eberhart, Russell C.: Parameter Selection in Particle Swarm Optimization. In: *EP '98: Proceedings of the 7th International Conference on Evolutionary Programming VII*. London, UK : Springer-Verlag, 1998. – ISBN 3–540–64891–7, S. 591–600

[Sug98] Suganthan, P. N.: Particle swarm optimizer with neighborhood operator. In: *Proceedings of the Congress on Evolutionary Computation*, 1998, S. 1958–1962

[WBS01] Wegener, J. ; Baresel, A. ; Sthamer, H.: Evolutionary Test Environment for Automatic Structural Testing. In: *Information and Software Technology* 43 (2001), Nr. 1, S. 841–854

[WRDM96] Whitley, Darrell ; Rana, Soraya ; Dzubera, John ; Mathias, Keith E.: Evaluating evolutionary algorithms. In: *Artif. Intell.* 85 (1996), Nr. 1-2, S. 245–276

[WW06] Wappler, S. ; Wegener, J.: Evolutionary Unit Testing of Object-Oriented Software Using a Hybrid Evolutionary Algorithm. In: *Proceedings*

of the IEEE World Congress on Computational Intelligence (WCCI-2006). Vancouver, Canada : IEEE Press, Juli 2006, S. 3193–3200

Eidesstattliche Erklärung

Die selbständige und eigenhändige Anfertigung versichere ich an Eides Statt.

Berlin, den 3. April 2007

_____

Andreas Windisch