Komponentenbasiertes Rapid Prototyping am Beispiel der Biomolekularen Sequenzanalyse

Alexander Georg Vilbig

Institut für Informatik der Technischen Universität München

Institut für Informatik der Technischen Universität München

Komponentenbasiertes Rapid Prototyping am Beispiel der Biomolekularen Sequenzanalyse

Alexander Georg Vilbig

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Dr. h.c. Jürgen Eickel

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Manfred Broy
2. Univ.-Prof. Dr. Johann Schlichter
3. Univ.-Prof. Dr. Walter Staudenbauer, Weihenstephan

Die Dissertation wurde am 25.6.2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 15.11.2001 angenommen.

KURZFASSUNG

Die Entwicklung umfangreicher und komplexer Software-Systeme ist eine überaus anspruchsvolle Aufgabe, deren Durchführung mit einem hohen Aufwand an Zeit und Kosten verbunden ist. Daher ist es wünschenswert, die Anforderungen an ein zukünftiges System möglichst genau und vollständig zu erfassen, um den Bedürfnissen des Anwenders tatsächlich gerecht zu werden und auftretende Mängel frühzeitig zu entdecken. Funktionale Prototypen des späteren Systems erleichtern diese Teilaufgabe der Systementwicklung erheblich, da somit eine tragfähige Grundlage für den Dialog zwischen Anwender und Entwickler geschaffen wird.

Software-Komponenten bieten sich in besonderer Weise zur Konstruktion funktionaler Prototypen an, weil diese bereits implementierte Funktionalität zusammenfassen und über ausgezeichnete Schnittstellen ihrer Umgebung zur Verfügung stellen. Durch Komposition entstehen so verhältnismäßig rasch übergeordnete Systeme mit umfangreicher Funktionalität. Dennoch verbleiben Suche, Auswahl und Verknüpfung geeigneter Komponenten als überwiegend manuelle Schritte der Konstruktion, die zudem durch ungenügende Beschreibung der angebotenen Funktionalität erheblich erschwert werden.

In dieser Arbeit wird daher ein fortgeschrittener Ansatz für komponentenbasiertes Rapid Prototyping entwickelt, der eine weitgehend automatisierte Erstellung funktionaler Prototypen an Hand vorgegebener Anforderungen und bereitgestellter Komponenten ermöglicht. Hierfür wird ein konzeptueller Rahmen vorgeschlagen, dessen klare, aufgabenbezogene Strukturierung der Problemstellung den Einsatz jeweils besonders geeigneter Modelle und Verfahren erleichtert. So kann Funktionalität auf anwendungsbezogener Ebene als Manipulation definierter Konzepte einer Ontologie verstanden werden, während deren Realisierung auf technischer Ebene durch typische Interaktionen beteiligter Schnittstellen beschrieben wird. Der tolerante Abgleich zwischen erwünschter und angebotener Funktionalität führt zu unterschiedlich zusammengesetzten Prototyp-Varianten, welche durch eine evolutionäre Heuristik schrittweise optimiert werden.

Auf diese Weise können auch zahlreiche alternative Lösungen mit vertretbarem Aufwand betrachtet werden. Darüber hinaus lassen sich potentielle Komponenten für Entwurf und Implementierung des späteren Systems frühzeitig hinsichtlich ihrer Eignung beurteilen. Die Effektivität und praktische Relevanz des vorgestellten Ansatzes wird durch eine Referenz-Implementierung sowie einen exemplarisch untersuchten Anwendungsbereich, die biomolekulare Sequenzanalyse, sichergestellt. Zudem belegen vielfältige Erweiterungen die Flexibilität und das zukünftige Potential der erarbeiteten Konzeption.

DANKSAGUNG

Mein besonderer Dank gilt zunächst Herrn Prof. Dr. Manfred Broy für seine wohlwollende, konstruktive und geduldige Betreuung dieser Arbeit. Zudem ist er maßgeblich verantwortlich für die Entscheidung, mich nach dem Studium der Biologie nunmehr der Informatik zuzuwenden. Diese Wahl hat meinen beruflichen Werdegang verändert und somit letztlich mein Leben und Denken wesentlich beeinflußt. Weiterhin möchte ich mich bei Herrn Prof. Dr. Walter Staudenbauer für die aufgewendete Mühe und Zeit bei Begutachtung dieser Arbeit herzlich bedanken. Er hat bereits 1992 meine erste Diplomarbeit im Bereich der Molekularbiologie engagiert betreut.

Überaus wertvolle Hinweise, Anregungen und Kritik an den fachlichen Inhalten der vorliegenden Arbeit sind Dr. Holger Giese zu verdanken. Mit ihm verbindet mich darüber hinaus ein freundschaftliches Verhältnis, das mir in schwierigen Zeiten wieder Kraft und Zuversicht gegeben hat.

Natürlich trägt auch die ausgesprochen produktive Atmosphäre am Lehrstuhl für Software & Systems Engineering wesentlich zum Erfolg dieser Arbeit bei. Ich möchte mich daher bei allen Kolleginnen und Kollegen sehr herzlich für die gute Zusammenarbeit und interessanten Diskussionen der letzten Jahre bedanken. Dies betrifft insbesondere Dr. Klaus Bergner, Andreas Rausch und Marc Sihling aus dem Teilprojekt „Methodik der bausteinorientierten Softwareentwicklung" des Bayerischen Forschungsverbundes Software Engineering. Unsere gemeinsamen Arbeiten und Veröffentlichungen haben mein Verständnis der komponentenbasierten Softwareentwicklung maßgeblich geprägt.

Viele dieser Ideen und Konzepte konnte ich in eigenen Projekten anwenden und erproben. Für deren praktische Umsetzung im Rahmen von Semester- oder Diplomarbeiten bin ich Wolfram Förster, José Leisibach und Andreas Sanft zu Dank verpflichtet.

Mein tiefer Dank gilt nicht zuletzt meiner Familie, ohne deren finanzielle und emotionale Unterstützung eine derart umfassende akademische Ausbildung unmöglich gewesen wäre. Besonders meiner Mutter, Dr. Margit Vilbig, möchte ich auf diesem Weg für die Liebe, Verständnis und tatkräftige Hilfestellung zu allen Zeiten danken.

Die vorliegende Arbeit ist meinen Freunden gewidmet. Ihre Zuneigung, kritische Auseinandersetzung und menschliche Unterstützung machen mein Leben erst lebenswert. You know who you are.

München, Juni 2001

INHALTSVERZEICHNIS

1. Einleitung . . . 1
- 1.1 Ausgangssituation . . . 1
- 1.2 Aufgabenstellung . . . 4
- 1.3 Aufbau und Beitrag der Arbeit . . . 7

2. Grundlagen . . . 11
- 2.1 Rapid Prototyping . . . 12
- 2.2 Komponentenbasierte Softwareentwicklung . . . 17
- 2.3 Biomolekulare Sequenzanalyse . . . 24
  - 2.3.1 Molekularbiologische Grundlagen . . . 25
  - 2.3.2 Ausgewählte Anwendungen der Sequenzanalyse . . . 31

3. Framework . . . 41
- 3.1 Anforderungen . . . 41
  - 3.1.1 Expressivität . . . 42
  - 3.1.2 Komplexität . . . 42
  - 3.1.3 Komponentenorientierung . . . 43
  - 3.1.4 Effektivität . . . 43
  - 3.1.5 Technische Umsetzung . . . 43
  - 3.1.6 Skalierbarkeit . . . 44
  - 3.1.7 Anwendbarkeit . . . 44
- 3.2 Überblick . . . 45
- 3.3 Ontologie . . . 53
- 3.4 Component Use Cases . . . 64
- 3.5 Funktionale Spezifikation . . . 75
- 3.6 Generierung funktionaler Prototypen . . . 80
  - 3.6.1 Anwendungsbeispiel . . . 84
  - 3.6.2 Komponenten-Auswahl . . . 89
  - 3.6.3 Repräsentationen und Adapter-Generierung . . . 95
  - 3.6.4 Varianten-Generierung . . . 105
  - 3.6.5 Varianten-Optimierung . . . 125
- 3.7 Methodische Umsetzung . . . 150

3.8 Zusammenfassung 156

4. Implementierung 159
4.1 Übersicht 159
4.2 Komponenten der Architektur 163
4.3 Zusammenfassung 171

5. Erweiterungen 173
5.1 Ontologie 174
5.1.1 Constraints 174
5.1.2 Integration von Manipulationen 177
5.1.3 Beschreibungstechnik und Werkzeugunterstützung . . . 178
5.2 Funktionale Spezifikation und Component Use Cases 183
5.2.1 Logische Anteile der Modellierung 184
5.2.2 Technische Anteile der Modellierung 185
5.2.3 Beschreibungstechnik und Werkzeugunterstützung . . . 187
5.3 Generierung funktionaler Prototypen 190
5.3.1 Komponenten-Auswahl 190
5.3.2 Adapter-Generierung 193
5.3.3 Varianten-Generierung 199
5.3.4 Varianten-Optimierung 204
5.4 Übergreifende Erweiterungen des Frameworks 210
5.5 Zusammenfassung 219

6. Diskussion 225
6.1 Erfüllung der gestellten Anforderungen 225
6.1.1 Expressivität 226
6.1.2 Komplexität 227
6.1.3 Komponentenorientierung 229
6.1.4 Effektivität 230
6.1.5 Technische Umsetzung 232
6.1.6 Skalierbarkeit 234
6.1.7 Anwendbarkeit 236
6.2 Vergleich mit bestehenden Ansätzen 239
6.2.1 Übergreifend vergleichbare Ansätze 239
6.2.2 Partiell vergleichbare Ansätze 245
6.3 Zusammenfassung und Bewertung 253

7. Zusammenfassung 261

Anhang 269

A. Technische Realisierung 271
A.1 Beschreibung der Ontologie 271
A.2 Beschreibung der Funktionalen Spezifikation 276
A.3 Beschreibung eines Component Use Case 277

Literaturverzeichnis 281

1. EINLEITUNG

1.1 Ausgangssituation

Die Erstellung umfangreicher, leistungsfähiger und qualitativ hochwertiger Software-Systeme repräsentiert nach wie vor eine überaus anspruchsvolle Aufgabe des Software Engineering. In der Praxis zeigen sich trotz verbesserter Sprachen, Werkzeuge und Vorgehensweisen anhaltende Probleme bei Durchführung entsprechender Entwicklungsprojekte. Ihre Ergebnisse werden in vielen Fällen verspätet ausgeliefert, übersteigen die ursprünglich veranschlagten Kosten oder werden den Bedürfnissen und Anforderungen der späteren Benutzer nur ungenügend gerecht.

Die möglichen Ursachen dieser Schwierigkeiten und auftretenden Mängel sind ebenso zahlreich wie vielschichtig. So nimmt zum einen die anwendungsbezogene Komplexität der erstellten Systeme fortwährend zu, da ständig neue, veränderte oder sogar widersprüchliche Anforderungen des Benutzers zu erfassen und angemessen zu berücksichtigen sind. Andererseits ist auch deren technische Komplexität aufgrund zunehmender Verteilung, geforderter Integration mit bestehenden Systemen sowie wachsenden Anforderungen an Zuverlässigkeit, Sicherheit und Performanz als durchaus erheblich einzuschätzen. Zuletzt erschweren gegenwärtig stark verkürzte Entwicklungszyklen und knappe Ressourcen, insbesondere hinsichtlich qualifiziertem Personal, die termingerechte Fertigstellung von Software mit gleichbleibend hoher Qualität.

Unter diesen Gesichtspunkten versprechen zwei grundlegende, sich gegenseitig ergänzende Ansätze des modernen Software Engineering weitreichende Verbesserungen bei der Entwicklung komplexer Software-Systeme. So ist zunächst das Verständnis eines solchen Systems als Komposition wohldefinierter Bestandteile, der sog. *Software-Komponenten*, von entscheidender Bedeutung. Diese Auffassung ermöglicht eine klare Strukturierung der anfallenden Entwicklungsaufgaben, vereinfachte Konstruktion umfangreicher Systeme sowie umfassende Wiederverwendung bereits implementierter und bewährter Funktionalität. Zudem stehen für ausgewählte Plattformen bereits seit längerer Zeit durchaus leistungsfähige und ausgereifte technische Infrastrukturen, wie JavaBeans [Sun00a], COM [Rog96] oder COR-

BA [Pop98], zur Verfügung, die neben entsprechenden Werkzeugen auch zu einem kommerziell bedeutenden Markt für hochwertige Komponenten geführt haben. Diese Basistechnologien werden durch aktuelle Arbeiten über Grundlagen und Methodik der komponentenbasierten Softwareentwicklung vervollständigt [Szy98, BBR+00, BRSV00b].

Weiterhin hat sich die Erstellung unterschiedlicher *Prototypen* des späteren Systems oder ausgewählter Ausschnitte in der praktischen Softwareentwicklung vielfach bewährt. Hierdurch kann unvermeidlichen Defiziten aufgrund langer Entwicklungsdauer und fehlender Gegenständlichkeit von Software wirksam begegnet werden. Je nach Typ und Umfang der erstellten Prototypen lassen sich mit ihrer Hilfe bestimmte Merkmale und Eigenschaften des Systems verhältnismäßig rasch ermitteln, beurteilen und bei den weiteren Schritten angemessen berücksichtigen. Dies erleichtert den Dialog zwischen Benutzer und Entwickler zur Klärung der gestellten Anforderungen, ermöglicht eine Überprüfung kritischer Bestandteile des Systems und minimiert somit wesentliche Entwicklungsrisiken. Aus diesen Gründen ist eine unter dem Begriff *Rapid Prototyping* zusammengefaßte, weitreichende und frühzeitige Einbeziehung verschiedener Prototypen ein bedeutender Bestandteil nahezu aller modernen Vorgehensmodelle des Software Engineering [Boe88, Kru98, DW99].

Allerdings können tatsächlich funktionale Prototypen häufig nur in begrenztem Umfang eingesetzt werden, da ihre Erstellung i.a. mit einem hohen Aufwand an Zeit und Kosten verbunden ist. Eine in der Praxis durchaus übliche Beschränkung auf einfache Prototypen der Benutzeroberfläche wird jedoch den oben aufgeführten Möglichkeiten und Vorteilen des Rapid Prototyping bei weitem nicht gerecht. Somit ist es naheliegend, beide genannten Ansätze zu kombinieren, also funktionale Prototypen verhältnismäßig rasch und einfach aus bereits vorhandenen Software-Komponenten zu konstruieren. Schließlich ist ein gewisser Anteil der gewünschten Funktionalität mit hoher Wahrscheinlichkeit bereits implementiert und auf eine Verknüpfung mit weiteren Bausteinen prinzipiell vorbereitet. Darüber hinaus lassen sich im Rahmen derartiger Prototypen auch vielversprechende Komponenten frühzeitig ermitteln und hinsichtlich ihrer Eignung als potentieller Bestandteil des späteren Systems beurteilen.

Eine tragfähige Umsetzung dieses überaus lohnenswerten Ansatzes scheitert jedoch gegenwärtig an den Unzulänglichkeiten der bekannten Technologien und Konzepte. So ist die angebotene Funktionalität einer Software-Komponente häufig nicht ausdrucksvoll, prägnant und übersichtlich genug beschrieben. Dies erschwert bereits die Suche und Auswahl möglicherweise geeigneter Bausteine erheblich. Zudem weisen gerade leistungsfähige Software-Komponenten besonders umfangreiche Schnittstellen sowie zahlreiche Inter-

aktionen mit ihrer Umgebung auf. Eine diesbezügliche Dokumentation ihres Verhaltens bzw. der Nutzung ihrer Funktionalität steht jedoch in aller Regel nur informell als Beschreibung in natürlicher Prosa zur Verfügung. Dies führt zu einem erhöhten Einarbeitungsaufwand und verhindert eine einfache, zumindest partiell automatisierbare Integration mit anderen Komponenten des Systems. Somit kann das eigentliche Potential des komponentenbasierten Rapid Prototyping nicht annähernd ausgeschöpft werden.

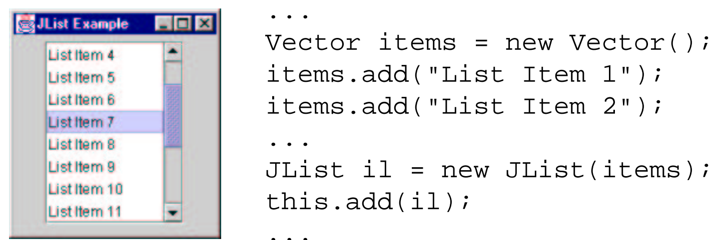


Abb. 1.1: *Benutzung der Komponente JList*

Abbildung 1.1 verdeutlicht diesen Sachverhalt am Beispiel der dargestellten Klasse JList, ein elementarer Bestandteil grafischer Benutzeroberflächen im Rahmen der Java-Plattform [AGH00]. Ungeachtet einer später eingeführten, genauen Definition des zentralen Begriffs „Software-Komponente“, läßt sich die implementierte Funktionalität der Klasse JList über ihre als öffentlich deklarierten Methoden in zahlreichen unterschiedlichen Anwendungen gewinnbringend nutzen; JList kann somit durchaus als eigenständige und umfassend wiederverwendbare Komponente verstanden werden. Allerdings beinhaltet die Schnittstelle dieser Komponente bereits etwa 60 eigene Methoden sowie über 200 indirekt, über den objekt-orientierten Mechanismus der Vererbung angebotene Methoden ihrer Oberklassen. Diese überaus umfangreiche Schnittstelle wird lediglich durch eine einfache Aufzählung aller Methoden mit weitgehend informeller Beschreibung in einem besonderen, als *JavaDoc* bezeichneten Format [Sun00b], dokumentiert. Hierdurch wird offensichtlich ein rascher Überblick über die insgesamt angebotene Funktionalität oder deren zielführende Benutzung erheblich erschwert.

Andererseits läßt sich ihre *grundlegende* Funktionalität, also die Anzeige einer Menge oder Liste von Elementen, bereits über eine sehr einfach gestaltete Interaktion in Anspruch nehmen, wie das nebenstehende Code-Fragment in Abbildung 1.1 belegt. Tatsächlich werden derartige Anwendungsbeispiele häufig zu Beginn der Dokumentation angeführt, um dem Entwickler das Verständnis der Komponente zu erleichtern. Darüber hin-

aus erlaubt die Kenntnis solcher Interaktionen bereits heute gängigen Entwicklungswerkzeugen, die visuelle Konstruktion umfangreicher grafischer Benutzeroberflächen effektiv und umfassend zu unterstützen. Hierbei plaziert der Entwickler gewünschte Elemente der Oberfläche am Bildschirm, während das Werkzeug entsprechenden Code für deren Integration automatisch generiert. Es stellt sich somit die Frage, inwieweit eine solche anwendungsbezogene Auffassung der angebotenen Funktionalität verallgemeinert und geeignet modelliert werden kann, um eine weitgehend automatisierte Verknüpfung funktionaler Komponenten zu übergeordneten Systemen zu erreichen. Diese besondere Zielsetzung liegt der gewählten Aufgabenstellung zugrunde, wie im folgenden näher erläutert wird.

1.2 Aufgabenstellung

Aufgrund der oben beschriebenen Ausgangssituation erscheint es überaus lohnenswert, die Techniken der komponentenbasierten Softwareentwicklung zur Konstruktion funktionaler Prototypen zu nutzen. Diese Modelle des späteren Systems lassen sich wesentlich schneller und somit günstiger erstellen, falls eine hinreichende Anzahl geeigneter, bereits implementierter Software-Komponenten angenommen werden kann. Im folgenden wird nunmehr die in dieser Arbeit untersuchte Aufgabenstellung im Überblick vorgestellt, wobei zunächst die hierbei vorausgesetzten Annahmen und Beschränkungen besonders hervorgehoben werden. Nach Betrachtung der erforderlichen technischen und fachlichen Grundlagen in Kapitel 2, werden später in Abschnitt 3.1 die maßgeblichen, allgemeinen Anforderungen an einen tragfähigen Ansatz für komponentenbasiertes Rapid Prototyping erarbeitet und ausführlich erläutert.

Die wesentlichen Aspekte der fundamentalen Problemstellung werden in Abbildung 1.2 im Rahmen einer schematischen Übersicht zusammengefaßt. Der obere Teil der Abbildung zeigt die Entwicklung eines Software-Systems als überwiegend manuellen Prozeß, in dessen Verlauf aus den gestellten Anforderungen des Anwenders – also eine Beschreibung der gewünschten Funktionalität sowie geltender Nebenbedingungen – eine möglichst umfassende Implementierung abgeleitet wird. Ohne besondere Kenntnisse über die Details dieser Implementierung kann angenommen werden, daß hierbei ein vollständig neues System gemäß den Bedürfnissen und Anforderungen im jeweiligen Anwendungsbereich erstellt wird, wie durch eine dunkle Färbung des verwendeten grafischen Symbols angedeutet ist. Demgegenüber kann mit Hilfe eines gegebenen Vorrats an funktionalen Software-Komponenten ein entsprechender Prototyp des späteren Systems verhältnismäßig rasch zusam-

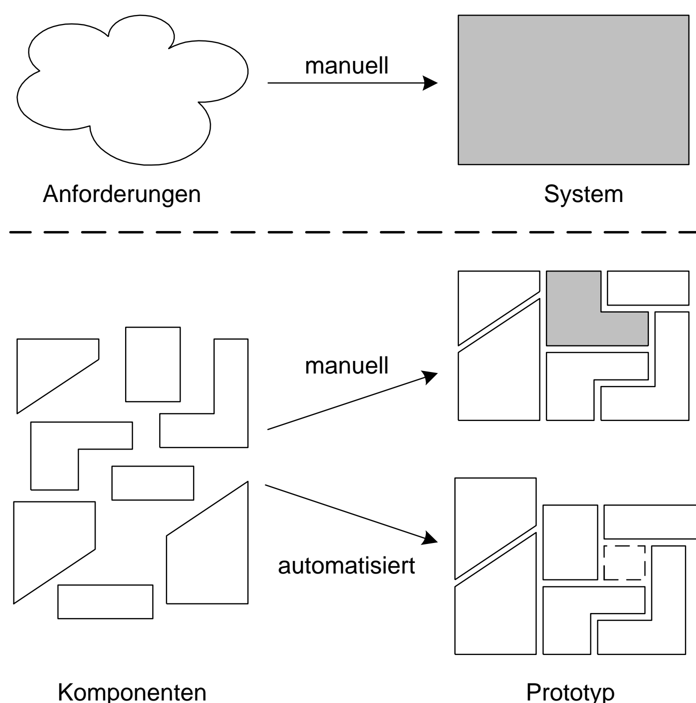


Abb. 1.2: *Schematische Übersicht der Problemstellung*

mengestellt werden, auch wenn ggf. bestimmte Ausschnitte des Systems aufgrund fehlender Funktionalität manuell zu ergänzen sind. Darüber hinaus verbleibt die Suche, Auswahl und Verknüpfung geeigneter Komponenten als durchaus aufwendige und anspruchsvolle Aufgabe des Entwicklers.

Das grundlegende Ziel der vorliegenden Arbeit ist daher eine möglichst weitgehende Automatisierung dieser bisher überwiegend manuell durchzuführenden Schritte, wie im unteren Teil von Abbildung 1.2 dargestellt ist. Hierdurch können auch zahlreiche, unterschiedlich zusammengesetzte Prototypen mit vertretbarem Aufwand betrachtet und in den Dialog mit dem Anwender einbezogen werden. Dies erleichtert einerseits die Erfassung und Konkretisierung der maßgeblichen funktionalen Anforderungen, ermöglicht andererseits aber auch eine effektive und effiziente Evaluierung potentiell geeigneter Bestandteile des späteren Systems.

Allerdings ist die allgemeine Lösung dieses Problems, also die vollständig automatisierte Konstruktion ausführbarer Prototypen aus einer beliebigen Menge an gestellten Anforderungen und vorhandenen Komponenten, offensichtlich eine überaus schwierige, wenn nicht sogar unlösbare Aufgabe. Daher werden im folgenden gewisse Annahmen und Einschränkungen getroffen, die

eine tatsächlich erfolgreiche Bearbeitung der genannten Problemstellung sicherstellen:

- Die für ein gegebenes Entwicklungsprojekt ebenfalls bedeutenden, *nicht-funktionalen* Anforderungen an das spätere System werden im vorgestellten Ansatz nicht berücksichtigt. Diese durchaus wünschenswerten Eigenschaften, wie Robustheit, Performanz oder Sicherheit, werden unter dem Begriff *Software-Qualität* zusammengefaßt und betreffen typischerweise übergreifend sämtliche Bestandteile des Systems. Ihre Vernachlässigung vereinfacht somit eine weitgehend lokale Betrachtung der ausgewählten Komponenten. Zudem kann auf ein umfassendes theoretisches Modells des Verhaltens einer Komponente oder eines zusammengesetzten Systems verzichtet werden.

- Bei Suche, Auswahl und Verknüpfung vorhandener Komponenten werden grundsätzlich auch suboptimale Ergebnisse akzeptiert, d.h. eine Integration möglicherweise wenig geeigneter Komponenten ist prinzipiell zulässig. Dieser, in Abbildung 1.2 durch entsprechende grafische Symbole der Komponenten angedeutete Sachverhalt erlaubt eine gewisse Toleranz beim Abgleich zwischen gewünschter und angebotener Funktionalität, wie später in Kapitel 3 ausführlich erläutert wird. Ein solches Vorgehen entspricht letztlich der Unbestimmtheit initialer funktionaler Anforderungen und repräsentiert somit keine fundamentale Einschränkung der Aufgabenstellung.

- Eine vollständige und umfassende Erfüllung sämtlicher funktionaler Anforderungen durch den erstellten Prototypen ist nicht zwingend erforderlich. Falls tatsächlich keine Komponente mit geeigneter Funktionalität ermittelt werden kann, wird zumindest ein entsprechender Rahmen bereitgestellt, der anschließend durch den Entwickler zu ergänzen ist. Diese Anteile der generierten Prototypen sind in Abbildung 1.2 durch eine unterbrochene Umrißlinie gekennzeichnet. Solche ggf. erforderlichen, manuellen Schritte verringern zwar die Effizienz eines ansonsten automatisierten Verfahrens, führen andererseits aber zu einer effektiven und pragmatischen Lösung der Problemstellung.

- Aus der Vielfalt an unterschiedlichen Typen von Software-Systemen werden in dieser Arbeit ausschließlich betriebliche und wissenschaftliche Informationssysteme betrachtet. Somit können einerseits Anwendungsbereiche ausgeschlossen werden, in denen nicht-funktionale Anforderungen eine besonders ausgezeichnete Rolle einnehmen, beispielsweise auf dem Gebiet der eingebetteten Systeme. Andererseits zeigen

Informationssysteme einen charakteristischen Bezug zu bekannten, gemeinsamen Konzepten des jeweiligen Anwendungsbereichs. Ein derartiger Bezug erleichtert dessen Modellierung sowie die Beschreibung zugehöriger Funktionalität erheblich. Diese Beschränkung schmälert zudem den Nutzen der vorgestellten Lösung nicht wesentlich, da Informationssysteme eine umfangreiche und wirtschaftlich bedeutende Klasse von Software-Systemen repräsentieren, wie nicht zuletzt durch den exemplarisch untersuchten Anwendungsbereich belegt wird.

Trotz der genannten Annahmen und Einschränkungen ergeben sich bei genauer Untersuchung der oben beschriebenen Zielsetzung zahlreiche grundlegende Fragestellungen: Was ist unter den zentralen Begriffen „Funktionalität" und „funktionale Anforderung" zu verstehen? Wie lassen sich diese Konzepte geeignet modellieren und beschreiben? Auf welche Weise können potentielle Komponenten gemäß ihrer angebotenen Funktionalität ausgewählt und weitgehend automatisiert verknüpft werden? Wie lassen sich technisch inkompatible Komponenten unabhängiger Hersteller gemeinsam berücksichtigen? Mit welchen Mitteln kann schließlich der kombinatorischen Vielfalt an unterschiedlich zusammengesetzten Prototypen begegnet werden?

Die vorliegende Arbeit beinhaltet mögliche Antworten auf diese Fragen, auch wenn eine vollständige, übergreifend gültige und abschließende Behandlung der Thematik offensichtlich nicht angenommen werden kann. Daher ist der vorgeschlagene Ansatz für komponentenbasiertes Rapid Prototyping grundsätzlich modular strukturiert, so daß sich ausgewählte Teile auch für andere Zwecke nutzen oder zukünftig durch besser geeignete Lösungen ersetzen lassen. Zudem wurden wesentliche Anteile der erarbeiteten Modelle und Verfahren im Zuge einer Referenz-Implementierung praktisch umgesetzt und an Hand eines durchgängigen Anwendungsbeispiels erprobt. Dies unterstreicht den pragmatischen und experimentellen Charakter dieser Arbeit, deren Aufbau und eigenständiger Beitrag im folgenden Überblick zusammengefaßt wird.

1.3 Aufbau und Beitrag der Arbeit

Bevor eine mögliche Lösung der in Abschnitt 1.2 beschriebenen Aufgabenstellung entwickelt wird, ist es zwingend erforderlich, die hierfür relevanten, technischen und fachlichen Grundlagen zu erläutern. Sie verdeutlichen die oben getroffenen Annahmen oder Beschränkungen und erlauben so die Ableitung allgemeiner Anforderungen an einen tragfähigen Ansatz für komponentenbasiertes Rapid Prototyping. Gemäß dem Titel dieser Arbeit werden daher in Kapitel 2 zunächst die unterschiedlichen Ziele und praktische Bedeutung

des Rapid Prototyping im übergeordneten Zusammenhang des Software Engineering diskutiert. Anschließend werden die grundlegenden Konzepte und gegenwärtig bedeutsamen Ansätze der komponentenbasierten Softwareentwicklung behandelt, wobei ein besonderer Schwerpunkt auf eine möglichst genaue Auffassung der zentralen Begriffe, wie „Software-Komponente“ oder „Schnittstelle“, gelegt wird. Zuletzt wird ein Überblick über die molekularbiologischen Grundlagen und ausgewählte Verfahren der biomolekularen Sequenzanalyse gegeben, um das Verständnis der später eingeführten Anwendungsbeispiele zu erleichtern.

Daraufhin können in Kapitel 3 die wesentlichen Ergebnisse der vorliegenden Arbeit vorgestellt werden. Nach Aufzählung der maßgeblichen, allgemeinen Anforderungen wird im folgenden ein konzeptueller Rahmen (engl. *framework*) zur Lösung der gestellten Aufgabe ausführlich erläutert. Dieses Framework strukturiert den vorgeschlagenen Ansatz in zusammengehörige Bereiche, so daß eine klare Trennung zwischen anwendungsbezogenen und technischen Anteilen der Problemstellung erreicht wird. Dies erlaubt den Einsatz jeweils besonders geeigneter Modelle und Verfahren, deren Entwicklung und Zusammenspiel den eigenständigen Beitrag dieser Arbeit beinhalten:

- Auf logischer Ebene des Frameworks wird gezeigt, wie sich die wesentlichen Konzepte des jeweiligen Anwendungsbereichs sowie ihre charakteristischen Beziehungen mit Hilfe einer geeignet modellierten *Ontologie* angemessen repräsentieren und übersichtlich beschreiben lassen. Hierbei ermöglicht eine zugrundeliegende Semantik der Ontologie die Ableitung weiterführender Zusammenhänge zwischen den so definierten Konzepten.

- Das erstellte Modell des Anwendungsbereichs erlaubt eine weitgehend deklarative Beschreibung grundlegender Funktionalität als Manipulation von Konzepten. Diese zentrale Abstraktion dient im Rahmen einer sog. *Funktionalen Spezifikation* zur Angabe der funktionalen Anforderungen an das System oder wird existierenden Komponenten in Form eines komponentenbezogenen Anwendungsfalls (*Component Use Case*) zur Dokumentation ihrer angebotenen Funktionalität beigefügt. Darüber hinaus beinhaltet ein derartiger Component Use Case auch technische Anteile, die typische Interaktionen bei Benutzung der betreffenden Komponente wiedergeben.

- Die erarbeiteten Modelle für Funktionale Spezifikation und komponentenbezogene Anwendungsfälle ermöglichen einen toleranten Abgleich zwischen erwünschter und angebotener Funktionalität auf logischer

Ebene des Frameworks. Die vorgeschlagenen Verfahren auf technischer Ebene erlauben anschließend eine weitgehend automatisierte Verknüpfung so ausgewählter Komponenten zu funktionalen Prototypen, wobei *Adapter* zwischen Komponenten unabhängiger Hersteller vermitteln. Somit werden auch zahlreiche, unterschiedlich zusammengesetzte Prototyp-Varianten effektiv unterstützt, deren Optimierung iterativ unter Einbeziehung des Entwicklers erfolgen kann. Zu diesem Zweck wird eine bekannte und bewährte Heuristik, ein sog. *Genetischer Algorithmus*, auf die vorliegende Problemstellung angepaßt.

Sämtliche oben aufgeführten Modelle und Verfahren werden durch entsprechende Beispiele aus dem exemplarisch untersuchten Anwendungsbereich, der Biomolekularen Sequenzanalyse, veranschaulicht und erläutert. Dies erleichtert einerseits das Verständnis der vorgestellten Ergebnisse, verdeutlicht andererseits aber auch die Effektivität und praktische Relevanz des vorgeschlagenen Ansatzes. Dieser übergeordneten Zielsetzung trägt ebenfalls die in Kapitel 4 beschriebene Referenz-Implementierung des Frameworks Rechnung. Sie zeigt eine mögliche Umsetzung der erarbeiteten Ergebnisse und demonstriert so deren Anwendbarkeit. Zudem kann die entwickelte Software-Architektur als Ausgangspunkt für eine zukünftige, praxisgerechte Implementierung herangezogen werden.

Ein wesentliches Merkmal des vorgestellten Ansatzes ist seine Flexibilität und Erweiterbarkeit im Hinblick auf verbesserte Lösungen einzelner Teilprobleme. Daher werden in Kapitel 5 entsprechende Erweiterungen oder alternative Modelle und Verfahren zusammengefaßt. Diese betreffen sowohl lokale Verbesserungen in ausgewählten Bereichen des Frameworks, als auch übergreifend bedeutende Weiterentwicklungen des gesamten Ansatzes. Hierbei wird angeführt, welcher Aufwand zu ihrer Umsetzung erforderlich ist, welche Vorteile gegenüber der bisherigen Lösung zu erwarten sind und auf welche Weise ihre Integration in das übergeordnete Framework erfolgen kann. Somit lassen sich bestehende Einschränkungen aufheben und vielversprechende Ansätze für zukünftige Arbeiten identifizieren.

Der erreichte Erfolg wird anschließend in Kapitel 6 ausführlich diskutiert. Hierfür werden zunächst die zuvor aufgestellten, allgemein gültigen Anforderungen an einen Ansatz für komponentenbasiertes Rapid Prototyping herangezogen. Dies erlaubt eine umfassende Einschätzung des vorgeschlagenen Frameworks hinsichtlich wesentlicher Kriterien, wie Effektivität, Skalierbarkeit oder Anwendbarkeit. Weiterhin ermöglicht der folgende Vergleich mit bekannten, ähnlichen oder verwandten Lösungen eine Beurteilung der erzielten Innovation gegenüber dem aktuellen Stand der Technik. Zudem ergeben sich durch eine solche Betrachtung wiederum lohnenswerte Ansätze für zukünf-

tige Weiterentwicklungen. Nach einer abschließenden, kritischen Bewertung der vorgestellten Ergebnisse werden die wesentlichen Inhalte dieser Arbeit in Kapitel 7 nochmals zusammengefaßt.

2. GRUNDLAGEN

Nach der Einführung in die untersuchte Problemstellung werden in diesem Kapitel die wesentlichen technischen und fachlichen Grundlagen vorgestellt. Hierfür ist zunächst eine Einordnung von Rapid Prototyping in den übergeordneten Kontext des Software Engineering hilfreich. Somit läßt sich die gewählte Zielsetzung und praktische Bedeutung der vorliegenden Arbeit genauer charakterisieren. Anschließend werden die besonderen Merkmale und Konzepte der komponentenbasierten Softwareentwicklung angeführt, um die später verwendeten Begriffe zu definieren sowie die grundsätzliche Eignung dieses Ansatzes zur Erstellung funktionaler Prototypen zu motivieren. Der zuletzt vorgestellte Überblick über die Molekularbiologie und ausgewählte Methoden der biomolekularen Sequenzanalyse ermöglicht ein grundlegendes Verständnis des exemplarisch untersuchten Anwendungsbereichs. Auf diese Weise lassen sich die später in dieser Arbeit aufgeführten Anwendungsbeispiele besser nachvollziehen, ohne die zur Analyse eingesetzten Verfahren im Detail zu behandeln.

Aufgrund des Umfangs und Komplexität der betrachteten Themenbereiche können offensichtlich nur die jeweils besonders bedeutsamen Konzepte und ihre Beziehungen näher erläutert werden. Daher ist für weiterführende Aspekte und ausführliche Behandlung der vorausgesetzten Grundlagen auf entsprechende Literatur zu verweisen. So orientiert sich Abschnitt 2.1 im wesentlichen an den in [MW91] zusammengefaßten Erkenntnissen über die Rolle des Rapid Prototyping im Softwareentwicklungsprozeß. Ein Überblick der komponentenbasierten Softwareentwicklung findet sich bei [Szy98], wobei in Abschnitt 2.2 auch aktuelle Forschungsarbeiten angemessen berücksichtigt werden [BBR+00, BRSV00b]. Die schließlich in Abschnitt 2.3 aufgeführten molekularbiologischen Grundlagen sind bekannten Lehrbüchern entnommen [Str88, Lew90], während die eingesetzten Modelle und Verfahren der biomolekularen Sequenzanalyse beispielsweise in [Wat95] und [LG91] umfassend beschrieben werden.

2.1 Rapid Prototyping

In nahezu allen Bereichen der produzierenden Industrie ist es gegenwärtig üblich, vor Beginn der Massenfertigung verschiedene Prototypen eines geplanten neuen Produkts zu erstellen. Dies erlaubt die Überprüfung ausgewählter Eigenschaften und Merkmale der Konstruktion, so daß etwaige Mängel oder Unzulänglichkeiten bereits frühzeitig erkannt und kostengünstig behoben werden können. Zudem läßt sich an Hand der erstellten Prototypen die Reaktion potentieller Kunden ermitteln und somit der spätere Markterfolg des Produkts sicherstellen. Auf diese Weise werden also bedeutende Risiken der Produktentwicklung durch den Einsatz von Prototypen minimiert sowie letztlich die Qualität des Produkts verbessert.

Aufgrund dieser Vorteile ist es naheliegend, diese erfolgreiche Strategie auch auf die Entwicklung komplexer Software-Systeme zu übertragen. Schließlich erfordert die Planung, Umsetzung und Einführung eines solchen Systems in der Regel ebenfalls einen erheblichen Aufwand an Mitteln und Zeit bei verhältnismäßig hoher Anzahl an möglichen Fehlerquellen und auftretenden Unsicherheiten. Daher bietet ein geeigneter Prototyp, der dem späteren System hinsichtlich charakteristischer Merkmale ähnelt, vielversprechende Möglichkeiten zur Lösung bedeutender, mit der Entwicklung verbundener Schwierigkeiten und Probleme, wie später in diesem Abschnitt ausführlich erläutert wird. Zunächst wird jedoch ein gemeinsames Verständnis der Begriffe *Prototyp* und *Rapid Prototyping* im Kontext der Softwareentwicklung festgelegt. Es bildet die Grundlage für die in dieser Arbeit vorausgesetzte Auffassung von Rapid Prototyping als bedeutendes Hilfsmittel des Software Engineering.

Definition 2.1: *Ein* Software Prototyp *ist ein Modell des Systemverhaltens, mit dessen Hilfe das zu entwickelnde System bzw. ausgewählte Aspekte besser verstanden sowie dessen zu erfüllende Anforderungen geklärt werden können.* Rapid Prototyping *bezeichnet die methodische, möglichst rasche Erstellung und Einsatz solcher Modelle zur Minimierung wesentlicher Entwicklungsrisiken und Verbesserung der insgesamt erzielten Qualität.*

Hierbei weist das erstellte Modell gewisse Eigenschaften und Merkmale des Systems auf, die je nach Zweck und Anwendungsbereich des Prototypen unterschiedliche Beurteilungen des zukünftigen Produkts erlauben. Beispielsweise führt ein Prototyp der Benutzeroberfläche auch bei einfachen Systemen mit vollständig erfaßten Anforderungen zu wertvollen Erkenntnissen über die bestmögliche Interaktion mit dem späteren Anwender. Hingegen liefert bei komplexen und umfangreichen Systemen ein entsprechend aufwendiger Prototyp weiterführende Informationen über kritische Aspekte des Entwurfs,

etwa die Leistungsfähigkeit einer geplanten Datenbank-Anbindung. In jedem Fall ergibt sich die Art des gewählten Modells sowie der erforderliche Aufwand für Prototyping aus der jeweils beabsichtigten Zielsetzung.

In diesem Zusammenhang ist die im Begriff „Rapid Prototyping“ angedeutete, *rasche* Erstellung entsprechender Prototypen grundsätzlich im Verhältnis zur benötigten Zeit für die Entwicklung des eigentlichen Systems zu verstehen. So ist auch eine verhältnismäßig lange Entwicklungsdauer für einen gegebenen Prototyp durchaus vertretbar, falls die hiermit gewonnenen Erkenntnisse über das spätere System hinreichend bedeutsam sind. Dennoch ist offensichtlich im allgemeinen eine vereinfachte und schnelle Konstruktion anzustreben, um die anfallenden Kosten möglichst gering zu halten.

Nach dieser grundlegenden Definition der zentralen Begriffe wird im folgenden die Bedeutung von Rapid Prototyping für den übergeordneten Prozeß der Softwareentwicklung genauer untersucht. Dies verdeutlicht die zu erwartenden Vorteile bei Einsatz funktionaler Prototypen und erlaubt eine Differenzierung der verschiedenen Ansätze. Somit kann im Anschluß die in dieser Arbeit vorgestellte Lösung hinsichtlich ihrer besonderen Zielsetzung genauer charakterisiert werden.

Die Entwicklung komplexer Software-Systeme ist aller Regel eine umfangreiche und schwierige Aufgabe, die sich zweckmäßig in verschiedene Teilaufgaben zerlegen läßt. Somit kann die Lösung dieser unterschiedlichen Teilprobleme durch geeignete Vorgehensweisen, Werkzeuge oder andere Hilfsmittel angemessen unterstützt werden. Hierbei wird Organisation, zeitlicher Ablauf und Abhängigkeit der im einzelnen erforderlichen Schritte und ihrer Ergebnisse durch den gewählten Softwareentwicklungsprozeß festgelegt. Für große Projekte werden vielfach bewährte, genau definierte Vorgehensmodelle, etwa das *V-Modell* [DW99] oder der *Rational Unified Process* [Kru98], eingesetzt, um eine möglichst erfolgreiche und mit gleichbleibender Qualität wiederholbare Durchführung zu gewährleisten.

Abbildung 2.1 zeigt eine schematische Übersicht der hierfür grundlegenden, im Zusammenhang mit Rapid Prototyping relevanten Schritte eines solchen Entwicklungsprozesses. So ist zu Beginn des Projektes zu klären, welche Aufgaben durch das zukünftige System aus Sicht des Anwenders zu erfüllen sind. Die gewünschte Funktionalität des Systems wird im Verlauf der *Anforderungsanalyse* im Dialog mit dem Anwender ermittelt und führt letztlich zu einem gemeinsamen Verständnis des betrachteten Anwendungsbereichs. Die so festgelegten, funktionalen Anforderungen werden unter Berücksichtigung ggf. bestehender Nebenbedingungen hinsichtlich Zuverlässigkeit, Performanz, Sicherheit u.ä., im Rahmen einer umfassenden *Systemspezifikation*

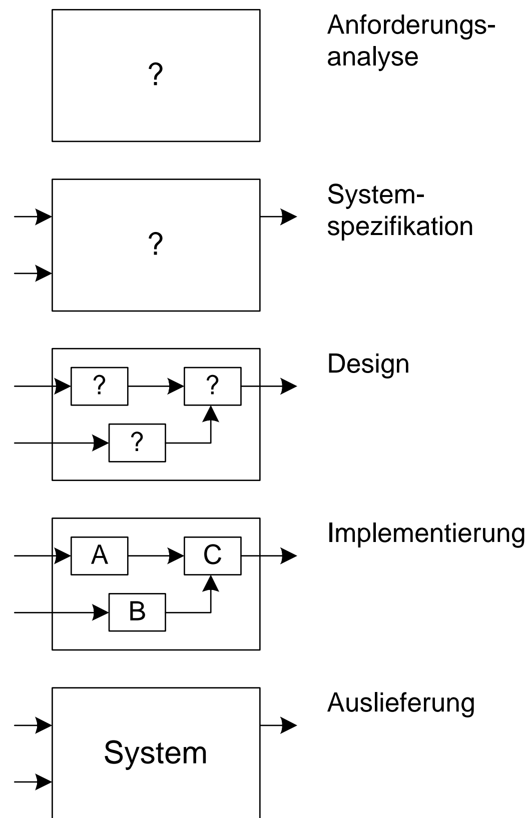


Abb. 2.1: *Grundlegende Schritte der Systementwicklung*

dokumentiert. Sie dient als Grundlage für das *Design*, in dem Struktur und Funktionsweise des Systems als Interaktion seiner Bestandteile entwickelt wird. Anschließend kann die eigentliche *Implementierung* der Komponenten erfolgen und das fertiggestellte System an den Anwender ausgeliefert werden. Es ist zu beachten, daß sämtliche genannten Schritte nicht notwendigerweise streng sequentiell und vollständig durchgeführt werden, sondern bei Anwendung eines modernen Ansatzes üblicherweise im Rahmen eines iterativen und inkrementellen Vorgehens organisiert sind. Zudem bedingt eine wirklich komponentenbasierte Softwareentwicklung die Einbeziehung bereits vorhandener Komponenten, wie später in Abschnitt 2.2 erläutert wird.

Dennoch erlaubt die in Abbildung 2.1 zusammengefaßte Übersicht des Entwicklungsprozesses eine Differenzierung der unterschiedlichen Ansatzpunkte für Rapid Prototyping. So verspricht gerade zu Beginn der Systementwicklung, also im Verlauf von Anforderungsanalyse und Systemspezifikation, der Einsatz funktionaler Prototypen bedeutende Vorteile.

Konsolidierung existierender Anforderungen: Die möglichst exakte, widerspruchsfreie Festlegung funktionaler Anforderungen ermöglicht ein angemessenes Design sowie vollständige Implementierung des Systems und führt somit zu einem für den Anwender befriedigenden Ergebnis der Entwicklung. Allerdings besitzt der Anwender in vielen Fällen eine zunächst nur ungefähre Vorstellung der erwünschten Funktionalität, die zudem bei unterschiedlichen Gruppen beteiligter Anwender gelegentlich auf inkonsistente Weise geäußert wird. Die Entdeckung solcher Widersprüche sowie eine insgesamt genauere Beschreibung der funktionalen Anforderungen wird durch den gemeinsamen Bezug auf einen vorhandenen Prototypen bzw. unterschiedlich implementierte Varianten der gleichen Funktionalität wesentlich erleichtert.

Ermittlung neuer Anforderungen: Häufig ist zu Beginn der Entwicklung die Gesamtheit aller gestellten Anforderungen nicht vollständig bekannt, etwa weil der Anwender nicht ausführlich befragt wurde oder bestimmte Funktionalität nur implizit vorausgesetzt wird. Wiederum ermöglicht die frühzeitige Betrachtung funktionaler Prototypen eine vereinfachte Ermittlung dieser bislang nicht berücksichtigten Anforderungen, da entsprechende Lücken der Spezifikation unmittelbar ersichtlich werden. Zudem können gerade im Vergleich mit existierenden Lösungen besonders ähnliche oder abweichende Anforderungen leichter entdeckt und ausgedrückt werden.

Verbesserte Kommunikation: Die Komplexität und fehlende Gegenständlichkeit eines Software-Systems erschwert ein gemeinsames Verständnis über das zu entwickelnde Produkt. Dies betrifft zunächst insbesondere die grundsätzlich erforderliche Kommunikation zwischen Anwender und Entwickler, da beide unterschiedliche Kompetenzen in ihrem jeweiligen Arbeitsbereich aufweisen. Zudem unterscheiden sich die jeweils gebräuchlichen Notationen und Beschreibungstechniken oftmals erheblich. Darüber hinaus ergeben sich gerade bei umfangreichen Entwicklungsprojekten auch häufig Probleme bei der Kommunikation zwischen den beteiligten Entwicklern. In allen genannten Fällen erleichtert ein vorhandener Prototyp die Diskussion über die zwischenzeitlich erzielten Resultate und das weitere Vorgehen im Projekt.

Die oben aufgeführten Vorzüge des Rapid Prototyping sind besonders bedeutsam, da auftretende Mängel oder Fehler zu Beginn des Entwicklungsprozesses bei späterer Entdeckung und Behebung unverhältnismäßig hohe Kosten verursachen. Schließlich beruhen alle weiteren Schritte der Entwicklung auf den erzielten Ergebnissen der Anforderungsanalyse und Systemspe-

zifikation. Dennoch bietet der Einsatz funktionaler Prototypen auch in den anschließenden Schritten wesentliche Vorteile.

Bewertung alternativer Lösungsvorschläge: Insbesondere bei anspruchsvoller Problemstellung oder neuartigen technischen Rahmenbedingungen ist die Wahl eines tatsächlich geeigneten Designs nicht unmittelbar ersichtlich. In diesem Fall erlaubt die Erstellung unterschiedlicher Prototypen eine umfassende Beurteilung möglicher Alternativen sowie den Nachweis der praktischen Umsetzung kritischer Ausschnitte des betrachteten Systems.

Ermittlung geeigneter Komponenten: Um den erforderlichen Aufwand an Zeit und Kosten für die Entwicklung zu reduzieren, bietet es sich an, bereits vorhandene Komponenten bei Design und Implementierung des Systems zu berücksichtigen. Hierbei erlauben unterschiedlich zusammengesetzte Prototypen eine rasche Einbindung potentiell geeigneter Komponenten, die somit frühzeitig hinsichtlich ihrer Qualität und angebotenen Funktionalität beurteilt werden können.

Verbesserung der Motivation: Design und Implementierung umfangreicher Software-Systeme erfordern zahlreiche Entwickler, die über einen langen Zeitraum hinweg jeweils unterschiedliche, klar abgegrenzte Bereiche des Systems bearbeiten. Unter diesen Umständen kann eine periodische Erstellung funktionaler Prototypen die Motivation des gesamten Teams erhöhen, da somit der erzielte Fortschritt des Projektes unmittelbar festzustellen ist.

Gerade der zuletzt genannte Gesichtspunkt einer kontinuierlichen und inkrementellen Weiterentwicklung ausführbarer Prototypen führt zum Ansatz des *evolutionären Prototyping*. Hierbei repräsentiert der betrachtete Prototyp selbst das eigentliche Produkt der Entwicklung, das in einem iterativ organisierten Prozeß nach Vorgaben oder Kommentaren des Anwenders schrittweise vervollständigt und verbessert wird, bis es schließlich als fertiggestelltes System ausgeliefert werden kann. Hingegen bezeichnet *exploratives Prototyping* eine ausschließliche Verwendung von Prototypen für ein verbessertes Verständnis der zugrundeliegenden Aufgabenstellung und möglicher Lösungsansätze. Dementsprechend werden explorative Prototypen nicht zur Konstruktion des endgültigen Systems eingesetzt. Dies erlaubt deutliche Abstriche bei Umfang, Qualität und Zuverlässigkeit der erstellten Prototypen und verringert somit den insgesamt erforderlichen Aufwand.

In dieser Hinsicht repräsentiert die vorliegende Arbeit einen Ansatz für exploratives Prototyping mit besonderem Nutzen für die frühen Schritte der

Softwareentwicklung. Die angestrebte, weitgehend automatisierte Konstruktion funktionaler Prototypen ermöglicht die Betrachtung zahlreicher unterschiedlicher Varianten des zukünftigen Produkts. Auf diese grundsätzlich experimentelle Weise wird vornehmlich der Dialog zwischen Anwender und Entwickler wirkungsvoll unterstützt. Allerdings bedeutet die vereinfachte Integration vorhandener Komponenten auch eine wertvolle Hilfestellung für Design und Implementierung, da somit potentielle Bestandteile des späteren Systems mit vertretbarem Aufwand ermittelt und beurteilt werden können. Die hierfür maßgeblichen Merkmale und Eigenschaften der komponentenbasierten Softwareentwicklung sind Gegenstand des folgenden Abschnitts.

2.2 Komponentenbasierte Softwareentwicklung

Das grundlegende Verständnis eines Systems als Kombination unterscheidbarer, klar definierter Bestandteile ist ein wesentliches Merkmal aller gereiften ingenieurwissenschaftlichen Disziplinen, etwa im Bereich des Automobil- oder Maschinenbaus. Mit seiner Hilfe können auch umfangreiche und komplexe Systeme zuverlässig aus bereits vorhandenen oder neu entwickelten Komponenten konstruiert werden. Dies reduziert den erforderlichen Aufwand für Entwicklung, Fertigung oder Wartung erheblich und ermöglicht somit letztlich kostengünstige, leistungsfähige und qualitativ hochwertige Produkte.

Eine Übertragung dieses weithin erfolgreichen Ansatzes auf den Bereich des Software Engineering erscheint daher naheliegend und vielversprechend. Schließlich repräsentiert heutige Software aufgrund stetig steigender Anforderungen und technischer Innovationen ebenfalls ein überaus komplexes Produkt, dessen Herstellung mit einem hohen Aufwand an Zeit und Kosten verbunden ist. Andererseits unterscheidet sich Software in einigen bedeutenden Punkten von herkömmlichen, materiellen Produkten. So entfällt ein Großteil der erforderlichen Ressourcen auf die Entwicklung und Wartung eines Software-Systems, weil dessen „Fertigung“ durch einfache Vervielfältigung eines Datenträgers erfolgen kann. Weiterhin erschwert die fehlende Gegenständlichkeit von Software die Bildung eines einheitlichen, gemeinsamen Verständnisses über das zu entwickelnde Produkt (vgl. Abschnitt 2.1). Zuletzt wird Software in zahlreichen verschiedenen Anwendungsbereichen eingesetzt, die jeweils spezifische Anforderungen an technische Umsetzung, Funktionalität und Qualität der geplanten Lösung stellen. Dies bedingt eine geringe Vergleichbarkeit unterschiedlicher Systeme, die zudem vielfach auch nachträglichen Veränderungen und Weiterentwicklungen unterworfen sind.

Aus diesen Gründen existiert gegenwärtig keine übergreifende, umfassende und weithin anerkannte Methodik der komponentenbasierten Soft-

wareentwicklung. Insbesondere wird das zentrale Konzept einer *Software-Komponente* in der Literatur durchaus unterschiedlich aufgefaßt. Daher wird zunächst der in dieser Arbeit verwendete Begriff genauer festgelegt. Die im folgenden vorgestellte Definition entspricht aktuellen Erkenntnissen [Szy98, BRSV00b] und läßt sich aufgrund ihrer Formulierung auf verschiedene technische Plattformen abbilden, wie später erläutert wird.

Definition 2.2: *Eine* Software-Komponente *repräsentiert die grundlegende Einheit der Funktion und Komposition als Bestandteil eines Software-Systems. Sie stellt ihre angebotene Funktionalität ausschließlich über wohldefinierte Schnittstellen bereit, die eine Verknüpfung mit anderen Komponenten des Systems ermöglichen. Die Entwicklung, Installation und Integration einer Software-Komponente kann durch unterschiedliche Parteien erfolgen, sofern jede Abhängigkeit von einem spezifischen Kontext explizit angegeben ist.*

Diese durchaus allgemein gehaltene Definition erlaubt dennoch bereits eine Ableitung weiterer charakteristischer Merkmale und Eigenschaften sowie eine Abgrenzung von bestehenden, bei erster Betrachtung ähnlichen Konzepten. So beinhaltet die Auffassung einer Komponente als Einheit der Funktion einen dynamischen, ablaufbezogenen Aspekt, der sie von anderen wiederverwendbaren Teilprodukten der Systementwicklung, wie beispielsweise Spezifikationen oder Entwurfsmuster [GHJV94], unterscheidet. Zudem läßt sich eine Komponente, etwa im Gegensatz zu objekt-orientierten Frameworks [Deu89], nur als vollständige und unveränderte Einheit nutzen.

Die Forderung nach *wohldefinierten Schnittstellen* entspricht dem Wunsch nach einer zuverlässigen und möglichst fehlerfreien Verknüpfung mit anderen Komponenten des Systems. Ungeachtet der spezifischen technischen Ausprägung einer solchen Schnittstelle wird in diesem Zusammenhang zunächst ein eindeutig festgelegter und konsistenter Zugriff auf deren angebotene Mittel zur Nutzung von Funktionalität vorausgesetzt. Dies betrifft beispielsweise die Benennung von Operationen, Typ und Reihenfolge erforderlicher Parameter oder Übergabe bereitgestellter Ergebnisse. Darüber hinaus bestimmt häufig ein definiertes *Protokoll* über den geeigneten dynamischen Ablauf, gültige Vor- und Nachbedingungen oder beobachtbare Zwischenzustände bei Benutzung einer gegebenen Schnittstelle. Diese Informationen werden bei weiterführenden Ansätzen als Teil der Spezifikation einer Komponente aufgefaßt und stehen somit für eine Integration mit anderen Komponenten zur Verfügung.

Schließlich repräsentiert die in Definition 2.2 zuletzt angeführte Unterscheidung zwischen Entwickler und Benutzer einer Komponente das grundlegende Merkmal der komponentenbasierten Softwareentwicklung und führt zu

weitreichenden Konsequenzen bei erfolgreicher praktischer Umsetzung eines solchen Ansatzes. So kann eine gegebene Komponente nur dann unabhängig entwickelt und eingesetzt werden, falls ihre angebotene Funktionalität im Kontext des jeweiligen Anwendungsbereichs klar ersichtlich und abgegrenzt ist, vorhandene Abhängigkeiten von diesem Kontext oder bereitgestellten Diensten der Umgebung explizit angegeben sind sowie eine hinreichende Flexibilität bei Kombination mit anderen Komponenten angenommen werden kann. Diese wesentlichen Eigenschaften sind allerdings nicht grundsätzlich technischer Natur und unterliegen somit einer gewissen Subjektivität bei Beurteilung einer tatsächlich „geeigneten" Komponente. Sie werden daher im folgenden auch an Hand charakteristischer Beispiele näher erläutert.

Die angebotene Funktionalität einer Komponente ist zunächst ausschlaggebend für deren Auswahl zur Konstruktion eines übergeordneten Systems. Falls diese Funktionalität und deren Nutzen im Hinblick auf den betrachteten Anwendungsbereich nicht klar ersichtlich oder hinreichend spezifisch erscheint, wird die entsprechende Komponente nicht eingesetzt bzw. voraussichtlich nicht erst aus der Menge an verfügbaren Komponenten ausgewählt. So kann beispielsweise das gesamte Betriebssystem eines Rechners in diesem Sinne schwerlich als eigenständige Komponente aufgefaßt werden, weil dieser Bestandteil des Systems eine ganze Reihe von höchst unterschiedlichen Diensten für verschiedenste Zwecke bereitstellt. Andererseits können ausgewählte Teile eines Betriebssystems, etwa die Benutzerverwaltung oder Elemente der grafischen Oberfläche, durchaus als Komponenten neuer Systeme verwendet werden, sofern ihr Einsatz in einem anderen Kontext technisch möglich ist. Der erkennbare Bezug zum jeweiligen Anwendungsbereich ist somit wesentlich für das gemeinsame Verständnis von Entwickler und Benutzer einer Komponente.

In aller Regel benötigt eine Komponente zur Erfüllung ihrer Aufgabe eine Reihe von weiteren Informationen oder Diensten aus ihrer Umgebung, beispielsweise den übergreifend verwendeten Zeichensatz oder die Möglichkeit, neue Ordner und Dateien anzulegen. Die benötigte Funktionalität wird üblicherweise von anderen Komponenten erbracht, die zum Zeitpunkt der Konstruktion oder Ausführung des Systems anzugeben sind[1]. Aus diesem Grund erfordert eine erfolgreiche Integration die genaue Bekanntgabe sämtlicher Abhängigkeiten im Rahmen der Dokumentation oder Spezifikation einer Komponente durch ihren Entwickler. Somit kann der Benutzer diese Komponente zuverlässig einsetzen, ohne die technischen Details ihrer Implementierung zu kennen. Diese auch als *Black-Box Wiederverwendung* bezeichne-

[1] Übergreifend benötigte Funktionalität wird häufig bereits von der eingesetzten technischen Infrastruktur bereitgestellt.

te, konsequente Beschränkung auf explizit angegebene Eigenschaften und Abhängigkeiten eines späteren Bestandteils des Systems unterscheidet komponentenbasierte und objekt-orientierte Ansätze der Softwareentwicklung. So stützt sich etwa die Implementierung einer Klasse in objekt-orientierten Programmiersprachen typischerweise auf vorhandene Funktionalität ihrer Oberklassen, deren Verfügbarkeit zur Laufzeit implizit vorausgesetzt wird.

Die Qualität einer gegebenen Komponente zeigt sich nicht zuletzt in ihrer Flexibilität bei Verknüpfung mit anderen Komponenten zur Konstruktion neuer Systeme. Besonders zahlreiche Abhängigkeiten von anderen Komponenten der Umgebung erschweren offensichtlich eine erfolgreiche Wiederverwendung in einem anderen Kontext. Andererseits führt der weitgehende Verzicht auf externe Abhängigkeiten zur Entwicklung von Komponenten mit entweder sehr generischer oder unnötig umfangreicher Funktionalität, welche den spezifischen Bedürfnissen des Benutzers nicht gerecht werden. Beispielsweise stellt nahezu jede technische Infrastruktur bestimmte Komponenten zur Verfügung, mit deren Hilfe sich Mengen von beliebigen anderen Komponenten effizient verwalten und manipulieren lassen. Diese ausgesprochen allgemein gehaltene Funktionalität leistet allerdings bei Konstruktion eines neuen Systems in einem gewählten Anwendungsbereich nur einen verhältnismäßig geringen Beitrag. Ein markantes Beispiel für besonders umfangreiche Komponenten liefern aktuelle Textverarbeitungssysteme, die üblicherweise auch Funktionalität zur Bearbeitung von Grafiken beinhalten. Die angebotenen Möglichkeiten sind jedoch häufig sehr einfach gehalten und somit den Leistungen speziell angepaßter Komponenten weit unterlegen. In der Praxis ist also bei der Entwicklung ein Kompromiß zwischen möglichst weitgehender Wiederverwendbarkeit und besonderer Abhängigkeit von einem spezifischen Kontext anzustreben, um den tatsächlichen Nutzen einer Komponente zu maximieren. Idealerweise ist die Implementierung einer Komponente auf verschiedene mögliche Umgebungen vorbereitet und paßt ihre angebotene Funktionalität den jeweils vorgefundenen Fähigkeiten und Beschränkungen in gewissen Grenzen an.

Die oben aufgeführten Prinzipien und weitergehenden Annahmen der komponentenbasierten Softwareentwicklung verdeutlichen, daß dieser Ansatz eine evolutionäre Weiterentwicklung bestehender, hinlänglich bewährter Methoden und Techniken repräsentiert. So bildet die Zusammenfassung von Zustand und Funktionalität hinsichtlich bedeutender Abstraktionen des Anwendungsbereichs eine wesentliche Motivation der objekt-orientierten Softwareentwicklung, während Modularität, Black-Box Wiederverwendung und das Konzept eindeutig spezifizierter Schnittstellen in zahlreichen unterschied-

lichen Ansätzen als Vorbedingung zur effektiven Entwicklung komplexer und umfangreicher Systeme propagiert wird. Die in Definition 2.2 getroffenen Aussagen fassen diese zentralen Aspekte zusammen, ohne hierbei eine besondere technische Realisierung zu favorisieren. Daher werden im folgenden verschiedene, gegenwärtig bedeutsame Technologien zur Entwicklung komponentenbasierter Systeme kurz vorgestellt. Dies veranschaulicht einerseits die flexible praktische Umsetzung der eingeführten abstrakten Konzepte, erlaubt andererseits aber auch eine erste Einschätzung ihrer wirtschaftlichen Relevanz.

So zeigt etwa das Beispiel der ursprünglich von Sun Microsystems entwickelten Java-Plattform [AGH00], daß eine gegebene Komponente durchaus als Klasse einer objekt-orientierten Programmiersprache implementiert werden kann. Das als *JavaBeans* [Sun00a] bezeichnete Komponentenmodell dieser technischen Plattform setzt hierfür zunächst lediglich die Existenz eines parameterlosen Konstruktors voraus, um eine Instanz der betreffenden Komponente bei Bedarf automatisch erzeugen zu können. Darüber hinaus läßt sich Zustand und Funktionalität einer solchen JavaBeans-Komponente über entsprechend benannte, als öffentlich deklarierte Methoden dieser Klasse manipulieren bzw. in Anspruch nehmen. Hierbei beinhaltet Java zusätzlich das Konstrukt einer expliziten Schnittstelle (engl. *interface*), welche die erwartete Signatur einer Implementierung festlegt. Mit Hilfe vorgegebener Schnittstellen und Protokolle können so auch weitergehende Dienste der zugrundeliegenden Infrastruktur, beispielsweise die Verteilung asynchroner Nachrichten oder die persistente Speicherung aktueller Konfigurationsdaten, für die Entwicklung einer Komponente genutzt werden. Ein besonderes Werkzeug der technischen Plattform ermöglicht schließlich die Zusammenführung aller Teile der Implementierung sowie deren zugehörige Dokumentation im Rahmen eines sog. *JAR-Archivs*, das somit als einheitliches Format zur Verteilung und Installation der betreffenden Komponente dient.

Im Gegensatz zu JavaBeans trifft das von Microsoft eingeführte *Component Object Model* (COM) [Rog96] und dessen aktuelle Weiterentwicklungen [BBC00, Mic01e] trotz seiner Namensgebung keine besonderen Annahmen über die zur Implementierung benutzte Programmiersprache. Es definiert vielmehr einen binären Standard zum Austausch von Parametern und Rückgabewerten bei Aufruf von Operationen aus einer spezifizierten Schnittstelle. Eine derartige Schnittstelle wird durch eine eigene Sprache, die sog. *Microsoft Interface Description Language* (MS-IDL), beschrieben und durch einen automatisch generierten Bezeichner global eindeutig identifiziert. Die jeweils angebotenen Schnittstellen werden im Zuge einer Installation der betreffenden Komponente bei der zugrundeliegenden Infrastruktur registriert und stehen fortan für andere Komponenten des Systems zur Verfügung.

Hierbei ermöglichen vordefinierte Schnittstellen und zugehörige Protokolle auch eine dynamische Zuordnung möglicher Implementierungen zur Laufzeit, die etwa im Rahmen geeigneter Entwicklungsumgebungen zur flexiblen Konstruktion ausgenutzt werden kann. Darüber hinaus bietet die eingesetzte Infrastruktur eine umfassende Unterstützung u.a. im Bereich der Versionierung, Verteilung und Sicherheit bei Entwicklung komponentenbasierter Systeme. Diese Merkmale haben in Verbindung mit einer weiten Verbreitung der Betriebssysteme und Entwicklungswerkzeuge von Microsoft zu einem umfangreichen kommerziellen Markt für COM-basierte Komponenten geführt, der letztlich auch das wirtschaftliche Potential dieses Ansatzes verdeutlicht.

Neben den oben aufgeführten Ansätzen steht mit der *Common Object Request Broker Architecture* (CORBA) [Pop98] auch ein unabhängiger, objekt-orientierter Standard zur Verknüpfung verteilter Komponenten zur Verfügung. Diese Entwicklung der Object Management Group (OMG) [OMG01b], ein Konsortium führender Hersteller der Computerindustrie, spezifiziert hierfür eine zentrale Komponente der Infrastruktur, den sog. *Object Request Broker* (ORB). Die Implementierung eines ORB ermöglicht den Aufruf von Operationen aus Schnittstellen beteiligter Komponenten des Systems unabhängig von deren Ort, Sprache ihrer Implementierung oder der zur Ausführung genutzten Plattform. Zu diesem Zweck werden die angebotenen Schnittstellen in einer eigenen Sprache, der sog. *OMG Interface Definition Language* (OMG-IDL) beschrieben und im Verlauf der Installation einer Komponente bei einem *Interface Repository* der zugrundeliegenden Infrastruktur registriert. Aufgrund definierter Abbildungen zwischen dieser gemeinsamen Beschreibung und den eingesetzten Programmiersprachen können die übergebenen Parameter und erhaltenen Rückgabewerte automatisch in das jeweils erwartete Format übersetzt und über das vorhandene Kommunikationsnetzwerk weitergeleitet werden. Hierbei erlaubt der ORB eine weitgehend dynamische und transparente Zuordnung zwischen Schnittstelle und Implementierung an Hand eindeutiger Referenzen für registrierte Komponenten des Systems. Somit können beispielsweise auch zur Laufzeit neue Komponenten in das System integriert oder verfügbare Funktionalität auf verschiedene Rechner verteilt werden.

Auf Basis dieses leistungsfähigen Standards zur Verknüpfung heterogener Komponenten auf technischer Ebene definiert die OMG mit der *Object Management Architecture* [Pop98] eine umfassende, weiterführende Software-Architektur zur vereinfachten Entwicklung komplexer verteilter Systeme auf anwendungsbezogener Ebene. Sie beinhaltet sowohl Spezifikationen für übergreifend genutzte Dienste der Infrastruktur, etwa im Bereich asynchroner Kommunikation, persistenter Speicherung von Objekten oder sicheren Transaktionen, als auch Vorgaben für spezifische Strukturen und Funktionalität in

bedeutsamen Anwendungsbereichen. Die letztgenannten Standards werden von unterschiedlich zusammengesetzten Arbeitsgruppen entwickelt und nach ihrer Verabschiedung durch verschiedene, oftmals spezialisierte Unternehmen implementiert. Allerdings zeigt die Erfahrung, daß divergierende Interessen der beteiligten Mitglieder sowie nicht ausreichend erfaßte, häufig wechselnde oder besonders spezifische Anforderungen eine rasche praktische Umsetzung der erzielten Ergebnisse verhindern. Zudem verursacht Komplexität und Umfang der vorgeschlagenen Lösung in vielen Fällen einen unangemessen hohen Aufwand für ihre Implementierung.

Dennoch führt dieser Ansatz der OMG zu wertvollen Erkenntnissen für zukünftige Arbeiten auf dem Gebiet der komponentenbasierten Softwareentwicklung. Schließlich repräsentiert die gegenwärtig erreichte, zuverlässige Verknüpfung unterschiedlicher Komponenten auf technischer Ebene nur eine notwendige, nicht aber hinreichende Bedingung für eine letztlich erfolgreiche Anwendung dieses Prinzips zur Konstruktion umfangreicher und komplexer Systeme. So stellt insbesondere die in Definition 2.2 getroffene Unterscheidung zwischen Entwickler und Benutzer einer Komponente in ihrer Folge weitergehende Anforderungen an logische Zusammengehörigkeit, Granularität und Flexibilität der angebotenen Funktionalität, wie oben ausführlich erläutert wird. Weiterhin ist zu untersuchen, wie das beobachtbare Verhalten einer Komponente bzw. eines zusammengesetzten Systems eindeutig beschrieben und getestet werden kann, welche Techniken und Werkzeuge eine derartige Spezifikation geeignet unterstützen und auf welche Weise der gesamte Entwicklungsprozeß eines komponentenbasierten Systems möglichst effektiv und effizient zu gestalten ist. Zuletzt sind auch wirtschaftliche Aspekte zu berücksichtigen, um die Ausbildung eines umfangreichen und stabilen Marktes für qualitativ hochwertige Komponenten zu gewährleisten. Diese Fragestellungen werden u.a. im Kontext moderner Softwareentwicklungsmethoden [DW98] sowie in aktuellen Arbeiten über Software-Architektur [GP95, AG97] und Grundlagen der komponentenbasierten Softwareentwicklung [BRSV98a, BRSV98b, BRSV99a, BRSV00b, BBR+00, BRSV99b, BJR+01] behandelt.

Für die Konstruktion funktionaler Prototypen ist eine umfassende und weitreichende Bearbeitung dieser Themenbereiche jedoch zunächst von untergeordneter Bedeutung, da hierbei durchaus erhebliche Abstriche bei Qualität und Zuverlässigkeit der erhaltenen Ergebnisse zugunsten einer schnellen und effektiven Lösung zu akzeptieren sind (vgl. Abschnitt 1.2 und 2.1). Immerhin versprechen die in Definition 2.2 zusammengefaßten Annahmen eine grundsätzliche Eignung von Softwarekomponenten zur Erstellung funktio-

naler Prototypen, da ihre abgegrenzte, klar definierte Funktionalität sowie die vorausgesetzte Flexibilität bei Komposition mit anderen Komponenten den raschen Aufbau umfangreicher Systeme wesentlich erleichtern. Zudem können leistungsfähige und ausgereifte Technologien für ihre Implementierung und spätere Verknüpfung herangezogen werden. Somit läßt sich die gestellte Aufgabe, also ein möglichst weitgehend automatisierter Übergang von funktionalen Anforderungen zu ausführbaren Prototypen, auf einer überwiegend anwendungsbezogenen Ebene lösen. Der in diesem Zusammenhang exemplarisch untersuchte Anwendungsbereich wird im folgenden Überblick vorgestellt.

2.3 Biomolekulare Sequenzanalyse

Die Molekularbiologie beschäftigt sich mit den Grundlagen des Lebens auf molekularer Ebene, d.h. sie versucht die beobachtbaren Vorgänge innerhalb und zwischen den Zellen lebender Organismen durch Interaktion der verschiedenen, übergreifend auftretenden Makromoleküle zu erklären. Es zeigt sich, daß bestimmte Klassen von Makromolekülen grundsätzlich unterschiedliche Aufgaben erfüllen, die in ihrer Gesamtheit und engen Wechselwirkung die Existenz und Weiterentwicklung biologischen Lebens erst ermöglichen. So besitzen *Proteine* eine Schlüsselrolle bei nahezu allen intra- und interzellulären Prozessen des Stoffwechsels, während *Nukleinsäuren*, wie Desoxyribonukleinsäure (DNA) oder Ribonukleinsäure (RNA), zur Speicherung und Weitergabe der grundlegenden Erbinformation jedes Organismus dienen.

Aufgrund der besonderen Struktur dieser bedeutenden Makromoleküle als weitgehend unverzweigte Kette definierter Einheiten, können wesentliche Erkenntnisse über ihre Funktion und Wechselwirkung allein aus der Abfolge ihrer Bestandteile abgeleitet werden. Eine solche *biomolekulare Sequenz* ist zumindest für Nukleinsäuren verhältnismäßig einfach und weitgehend automatisiert zu ermitteln, so daß gegenwärtig bereits die Erbinformation einiger ausgewählter Arten, auch die des Menschen, vollständig bestimmt werden konnte. Die gewonnenen Daten sind offensichtlich von enormer Bedeutung für wissenschaftliche, medizinische und wirtschaftliche Fragestellungen, etwa bei Entwicklung neuartiger Medikamente oder im Bereich gentechnisch veränderter Lebensmittel.

Allerdings sind für ein wirklich umfassendes Verständnis der Zusammenhänge und Abläufe weiterführende Analysen, Vergleiche mit bekannten Sequenzen sowie langwierige praktische Experimente unerläßlich. Hierbei leistet die Informatik einen überaus wertvollen Beitrag, da einerseits der beträchtliche Umfang an anfallenden Daten nur mit Hilfe leistungsfähiger Soft-

ware gespeichert, verwaltet und weltweit zugänglich gemacht werden kann. Andererseits lassen sich bereits bekannte, effiziente Algorithmen zur Analyse und Manipulation von Sequenzen, beispielsweise im Umgang mit Zeichenketten, auf die biologische Problemstellung übertragen. Zuletzt können theoretische Modelle für molekularbiologische Verhältnisse mit geeigneter Software simuliert und analysiert werden, um experimentelle Ergebnisse zu überprüfen oder zu ergänzen.

Somit repräsentiert die biomolekulare Sequenzanalyse einen bedeutenden Anwendungsbereich mit durchaus komplexen funktionalen Anforderungen an unterstützende Software-Systeme, aber überwiegend klar definierten Konzepten als unmittelbare Abbildung der realen Verhältnisse. Angesichts dieser Tatsachen erscheint ein komponentenbasierter Ansatz besonders vielversprechend, wie später in Kapitel 3 an Hand zahlreicher Anwendungsbeispiele belegt wird (vgl. auch Abschnitt 6.1.7). Zum besseren Verständnis dieser Beispiele werden im folgenden die wesentlichen, hierfür erforderlichen Grundlagen in vereinfachter Form zusammengefaßt.

2.3.1 Molekularbiologische Grundlagen

Proteine spielen eine entscheidende Rolle in nahezu allen biologischen Prozessen. Als Enzyme (Biokatalysatoren), Hormone, Rezeptoren und Antikörper, Struktur-, Transport- oder Speicherproteine erfüllen sie eine Vielzahl grundlegender Aufgaben, ohne die kein lebender Organismus existieren könnte. Gemäß ihrer besonderen molekularen Struktur handelt es sich bei Proteinen um lineare Polypeptide aus *Aminosäuren* als monomere Bestandteile. Jede Aminosäure besitzt als gemeinsames Merkmal sowohl eine *Aminogruppe* ($-NH_2$) als auch eine *Carboxylgruppe* ($-COOH$), über die sie mit anderen Aminosäuren unter Abspaltung von Wasser verknüpft werden kann. Durch eine wiederholte Anlagerung weiterer Aminosäuren entstehen so unverzweigte Ketten aus bis zu 3000 Bausteinen, deren Abfolge für das jeweilige Protein charakteristisch ist. Aufgrund der freien Amino- und Carboxylgruppe an den Enden der Kette kann dieser Sequenz eine eindeutig definierte Orientierung zugeordnet werden.

Abbildung 2.2 zeigt die verallgemeinerte chemische Strukturformel einer Aminosäure sowie die schematische Verknüpfung solcher Bausteine zu einem Polypeptid aus n Aminosäuren. Hierbei bezeichnet R bzw. R_1 bis R_n verschiedene chemische Restgruppen, durch die sich Aminosäuren voneinander unterscheiden. Diese differenzierenden Bestandteile der 20 natürlich vorkommenden Aminosäuren sind bezüglich ihrer Größe, Form und chemischen Eigenschaften durchaus heterogen und bestimmen so in ihrer Gesamtheit die Eigenschaften des jeweiligen Polypeptids maßgeblich.

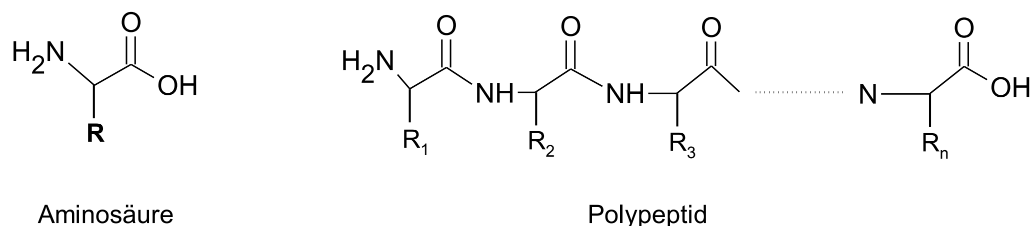


Abb. 2.2: *Bestandteile und Struktur von Proteinen*

Darüber hinaus legen die Seitengruppen eines Proteins auch dessen räumliche Struktur weitgehend fest. So lagern sich hydrophobe Gruppen in wäßriger Lösung zusammen und bewirken eine eher globuläre Form, während kovalente Bindungen oder polare Wechselwirkungen zwischen voneinander entfernten Gruppen auch Faltungen und Schleifen der Kette stabilisieren. Durch solche Effekte ergibt sich in der Regel eine sehr komplexe räumliche Struktur, wie in Abbildung 2.3 exemplarisch an Hand einer perspektivischen Darstellung von Streptavidin – ein mittelgroßes Protein aus 159 Aminosäuren (vgl. [Vil96]) – verdeutlicht wird. Hierbei sind lediglich die Bindungen der beteiligten Atome durch entsprechende Linien repräsentiert.

Es zeigt sich, daß die räumliche Struktur eines Proteins in den meisten Fällen für dessen biologische Funktion maßgeblich ist. Sie bestimmt etwa bei Enzymen die Form und Größe des umgesetzten Substrats, da sich dieses vor Ablauf der Reaktion an einen ausgezeichneten Ort des Enzyms, das sog. *aktive Zentrum*, anlagert. Erst an dieser Stelle, meist eine Spalte oder Mulde in der Oberfläche des Proteins, können die reaktiven Aminosäurereste mit dem Substrat in Kontakt treten und ihre katalytische Aktivität entfalten. Somit repräsentiert die Aufklärung der räumlichen Verhältnisse, beispielsweise über experimentelle oder rechnergestützte Verfahren, eine bedeutende Aufgabe der Molekularbiologie. Andererseits lassen sich bereits aus der Aminosäuresequenz eines Proteins wesentliche Erkenntnisse über dessen mögliche Funktion ermitteln, wie später in Abschnitt 2.3 erläutert wird.

Die vollständige Information über Aufbau und Zusammensetzung der verschiedenen Proteine einer Zelle wird als Nukleinsäure stofflich repräsentiert und in dieser Form an die Nachkommen eines Organismus weitergegeben. Hierbei handelt es sich ebenfalls um ein regelmäßig aufgebautes, lineares Makromolekül, dessen monomere Bestandteile als *Nucleotide* bezeichnet werden. Ein solches Nucleotid wiederum besteht aus einem Phosphatrest, einem Zucker sowie einer charakteristischen nitrogenen Base (siehe Abbildung 2.4).

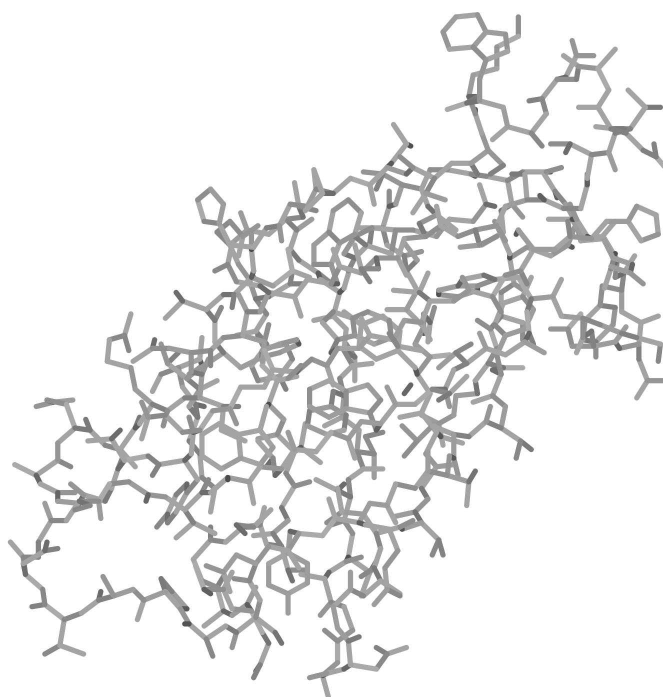


Abb. 2.3: *Räumliche Struktur des Proteins Streptavidin*

Aufgrund der chemischen Zusammensetzung kann zwischen *Desoxyribonukleinsäure* (DNA) und *Ribonukleinsäure* (RNA) unterschieden werden. So beinhaltet DNA ausschließlich *Desoxyribose* als Zucker sowie *Adenin* (A), *Guanin* (G), *Cytosin* (C) und *Thymin* (T) als mögliche Base eines Nucleotids, während bei RNA der Zucker *Ribose* vertreten ist und statt Thymin die Base *Uracil* (U) gefunden wird.

Die Kettenbildung erfolgt, indem der Zucker eines Nucleotids mit der Phosphatgruppe eines weiteren Nucleotids unter Abspaltung von Wasser reagiert. Entsprechend den numerierten Kohlenstoff-Atomen des Zuckers ergibt sich hierdurch auch bei Nukleinsäuren eine definierte Orientierung, wobei die Enden der Kette als 3' bzw. 5'-Terminus bezeichnet werden (vgl. Abbildung 2.4). Die so gebildete Kette eines gegebenen Nukleinsäuremoleküls unterscheidet sich lediglich durch die Abfolge der zugehörigen Basen, da Zucker und Phosphatrest bei allen Nucleotiden identisch sind. Adenin und Guanin werden aufgrund ihrer charakteristischen Ringstruktur unter dem Begriff *Purin-Basen* zusammengefaßt, während Cytosin, Thymin und Uracil in die Gruppe der *Pyrimidin-Basen* eingeordnet werden.

Eine wesentliche, für ihre biologische Funktion entscheidende Eigenschaft

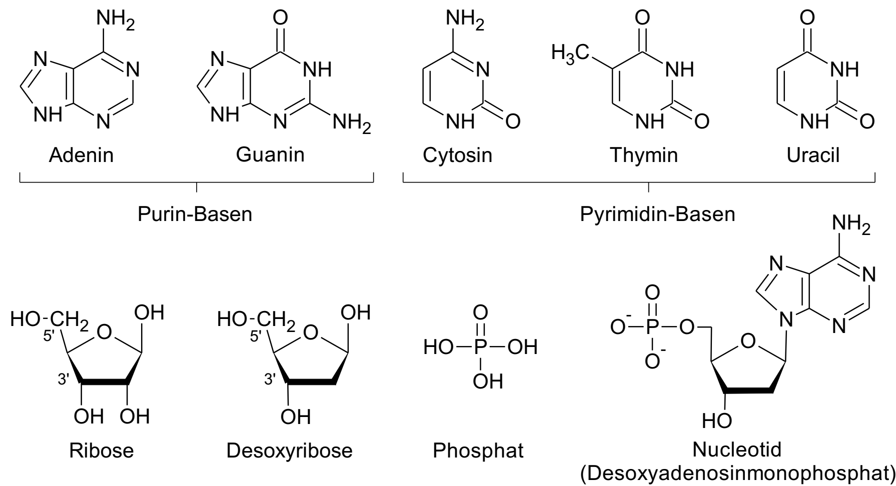


Abb. 2.4: *Bestandteile von Nukleinsäuren*

von Nukleinsäuren ist die mögliche *Basenpaarung* zwischen komplementären Purin- und Pyrimidin-Basen der verschiedenen Nucleotide. Hierbei bilden sich zwischen Adenin und Thymin bzw. Uracil sowie zwischen Guanin und Cytosin sog. *Wasserstoff-Brückenbindungen*, bei denen ein Wasserstoff-Atom von den beteiligten Molekülen gewissermaßen „geteilt“ wird, wie in Abbildung 2.5 durch unterbrochene Linien angedeutet ist. Der entstehende Komplex ist thermodynamisch bevorzugt und fördert somit die Stabilität der gepaarten Nucleotide.

DNA tritt in der Zelle üblicherweise als doppelsträngiges Molekül auf, wobei beide Ketten zueinander komplementär sind, also *wechselseitig* entsprechende Basenpaarungen eingehen. Somit ergibt sich die charakteristische räumliche Struktur einer Doppelhelix, deren Außenseite durch eine alternierende Folge von Phosphat- und Desoxyribose-Resten gebildet wird (siehe Abbildung 2.5). Demgegenüber findet sich RNA in der Regel als einzelsträngiges Molekül, bei dem Basenpaarungen zwischen komplementären Basen der *selben* Kette auftreten können. Dies führt zu einer weniger regelmäßigen räumlichen Struktur, die letztlich von der jeweiligen Zusammensetzung des RNA-Moleküls abhängig ist.

Die erforderliche Information zum Aufbau der zahlreichen verschiedenen Proteine einer Zelle ergibt sich aus der Abfolge der Nucleotide in den Nukleinsäuren. Hierbei dient DNA als dauerhafter Speicher der Erbinformation, der bei jeder Zellteilung repliziert und an die entstandenen Tochterzellen weitergegeben wird. Demgegenüber repräsentiert ein gegebenes RNA-Molekül in

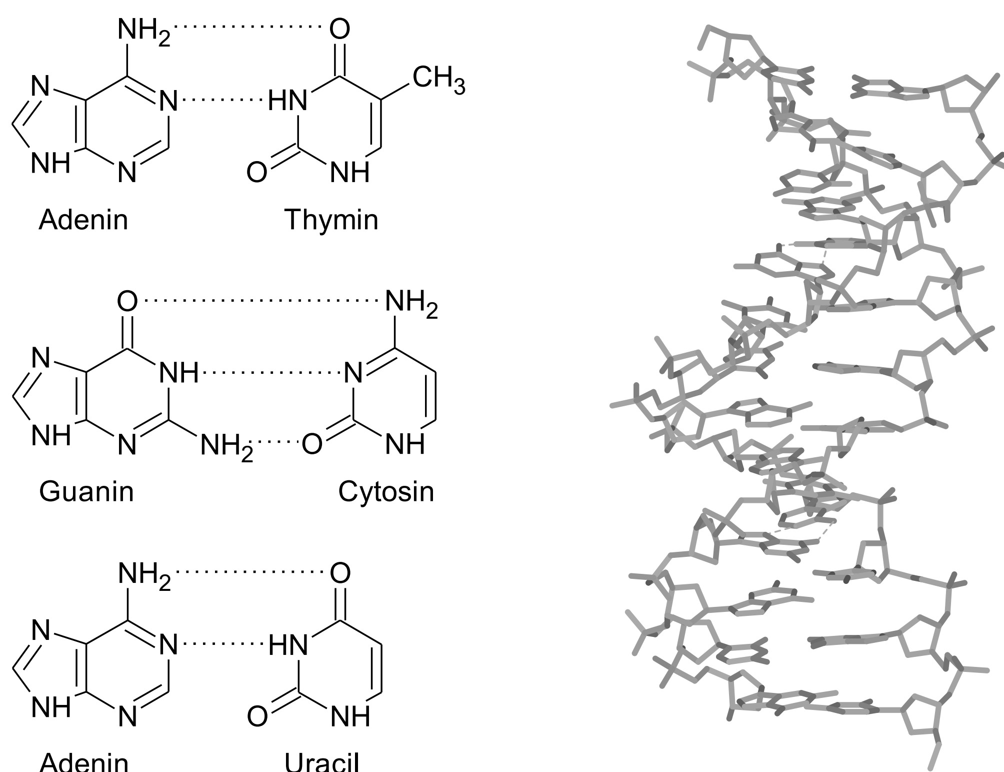


Abb. 2.5: *Basenpaare und räumliche Struktur von DNA*

der Regel die kurzlebige Kopie dieser Information zur lokalen Synthese eines bestimmten Proteins, welche durch entsprechende Enzyme in der Zelle rasch abgebaut wird[2].

Abbildung 2.6 faßt die wesentlichen Verhältnisse der *Expression* genetischer Information in schematischer Form zusammen. So wird zunächst ein zusammenhängender Bereich der DNA in einem als *Transkription* bezeichneten Vorgang kopiert. Aufgrund der komplementären Basenpaarung entspricht die entstandene *Messenger-RNA* (mRNA) einer exakten Kopie des abgebildeten, kodierenden Strangs der DNA, wobei Thymin durch Uracil ersetzt wird. Anschließend wird die Information der mRNA im Verlauf der *Translation* zur Synthese eines Proteins genutzt. Hierbei kodiert die Folge von drei Nucleotiden, ein sog. *Codon*, für eine eindeutig festgelegte Aminosäure des gebildeten Proteins. Beispielsweise entspricht das Codon `AUG` der Aminosäure *Methionin* (Met), `UCA` der Aminosäure *Serin* (Ser) oder `AGA` der Aminosäure

[2] Andere RNA-Moleküle erfüllen strukturelle Aufgaben oder entfalten selbst katalytische Aktivität.

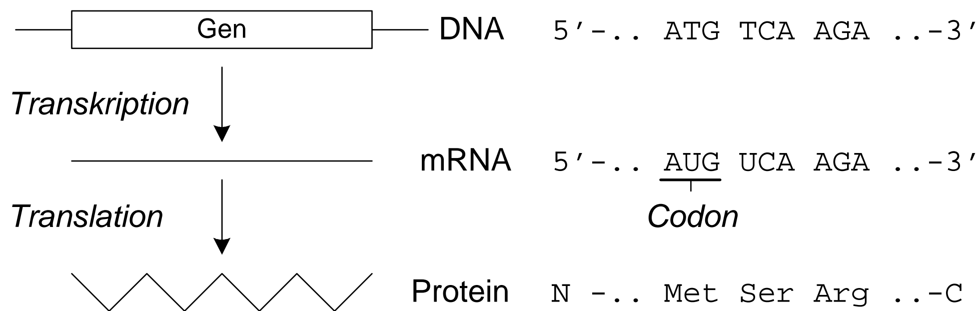


Abb. 2.6: *Schematischer Ablauf der genetischen Expression*

Arginin (Arg). Dieser *Genetische Code*, also die Abbildung zwischen Codon und kodierter Aminosäure, ist universell, d.h. bei nahezu allen Organismen identisch. Darüber hinaus repräsentieren besondere Codons das Ende der entstehenden Polypeptid-Kette. Somit ergibt sich insgesamt eine genau definierte Synthese des jeweiligen Proteins, dessen Orientierung vom N-Terminus zum C-Terminus letztlich der Orientierung des kodierenden Strangs der DNA vom 5'- zum 3'-Ende folgt.

Der für die Expression eines Proteins maßgebliche, zusammenhängende Ausschnitt der DNA wird als *(Struktur-)Gen* bezeichnet und repräsentiert die Einheit der Vererbung[3]. Somit können einzelne Gene nur vollständig an die Nachkommen vererbt werden, da partiell synthetisierte Proteine in der Regel ihre Funktion in der Zelle nicht erfüllen können. Die Gesamtheit aller Gene einer Spezies bildet das für sie charakteristische *Genom*. Dieses ist bei Eukaryonten, also Organismen, deren Zellen einen echten Zellkern aufweisen, typischerweise in zahlreiche verschiedene DNA-Doppelstränge, die sog. *Chromosomen* aufgeteilt. Beispielsweise besitzt der Mensch nach neuesten Schätzungen ca. $3 \cdot 10^4$ verschiedene Gene auf 23 Chromosomen mit insgesamt $3.6 \cdot 10^9$ Basenpaaren.

Aufgrund des fundamentalen Zusammenhangs zwischen der Nucleotid-Sequenz eines Gens und der Aminosäuresequenz des zugehörigen Proteins, ist es naheliegend, dessen biologische Funktion durch die Analyse seiner Sequenz aufzuklären oder zumindest erste Hinweise auf mögliche Funktionen und Beziehungen zu anderen Proteinen zu erhalten. Dies erscheint besonders vorteilhaft, da gerade DNA-Sequenzen durch fortschrittliche Laborverfahren verhältnismäßig einfach und weitgehend automatisiert ermittelt werden

[3] Es sind auch Gene bekannt, die nicht in ein Protein übersetzt werden. In diesem Fall besitzt entweder bereits die erstellte RNA selbst eine besondere biologische Funktion oder das betreffende Gen dient zur Regulation anderer Gene.

können. Eine Auswahl der gegenwärtig eingesetzten, grundlegenden Techniken der biomolekularen Sequenzanalyse wird in der folgenden Übersicht vorgestellt.

2.3.2 Ausgewählte Anwendungen der Sequenzanalyse

Die Aufklärung der molekularen Grundlagen des Lebens beinhaltet weitreichende Konsequenzen für Forschung und Anwendung im Bereich der modernen Biologie. Da letztlich nur in Form von DNA festgelegte Informationen von Generation zu Generation weitergegeben werden, läßt sich zumindest prinzipiell Aufbau, Funktionsweise und Entwicklung jedes Lebewesens allein aus der spezifischen Zusammensetzung seines Genoms vollständig erklären. Die aus der Analyse derartiger Sequenzen gewonnenen Erkenntnisse erweisen sich im Vergleich mit traditionellen, überwiegend an äußerlichen Merkmalen ausgerichteten Methoden als besonders verläßlich und weitgehend objektiv.

Mittels einer zentralen Abstraktion, die Nukleinsäure- bzw. Aminosäure-Sequenzen als Zeichenketten eines endlichen Alphabets auffaßt, können wesentliche Aufgaben einer weiterführenden Analyse durch entsprechende Verfahren und Algorithmen der Informatik übernommen werden. Dies erleichtert die Bearbeitung und Verwaltung experimentell gewonnener Sequenzdaten, ermöglicht den umfassenden Vergleich mit bereits bekannten Sequenzen oder erlaubt die Vorhersage abgeleiteter Zusammenhänge aus theoretischen Modellen. Aus der Fülle an gegenwärtig eingesetzten Methoden zur Sequenzanalyse wird im folgenden die Berechnung von sog. *Alignments* und *Phylogenetischen Bäumen* exemplarisch vorgestellt, weil die hierbei eingeführten Begriffe und Verfahren später in Kapitel 3 als Anwendungsbeispiel herangezogen werden.

Sequenz-Alignments

Der Vergleich von Protein- oder Nukleinsäure-Sequenzen, etwa bei Untersuchung eines bestimmten Gens mit identischer Funktion in verschiedenen Organismen, erfordert eine gewisse Toleranz hinsichtlich üblicherweise auftretender Unterschiede in der Abfolge der zugehörigen Bestandteile. Diese Unterschiede ergeben sich aus Fehlern der eingesetzten experimentellen Verfahren zur Bestimmung der Sequenz oder als Folge natürlicher Veränderungen des betreffenden Gens im Verlauf der evolutionären Entwicklung. Das übergeordnete Ziel der eingesetzten Verfahren zum paarweisen Vergleich von Sequenzen ist daher eine optimale Ausrichtung (engl. *alignment*), welche eine möglichst geringe Anzahl derartiger Veränderungen impliziert. Hierfür können Lücken (engl. *gap*) in die gegeneinander ausgerichteten Sequenzen

eingeführt werden, um fehlende bzw. neu hinzugekommene Bestandteile auszugleichen.

```
A:   GCTGATATAGCT
B:   GGGTGATTAGCT

A': -GCTGATATAGCT
B': GGGTGAT-TAGCT
```

Abb. 2.7: *Beispiel eines Alignments zweier DNA-Sequenzen*

Abbildung 2.7 verdeutlicht diesen Sachverhalt am Beispiel eines möglichen Alignments zweier DNA-Sequenzen A und B. Das unten abgebildete Alignment beinhaltet in den gegeneinander ausgerichteten Sequenzen A' und B' jeweils eine durch das Sonderzeichen '-' repräsentierte Lücke, um ansonsten eine möglichst weitgehende Übereinstimmung an den restlichen Positionen zu erzielen. Dennoch befindet sich an der dritten Stelle des Alignments ein Unterschied zwischen der Base Cytosin (C) in der Sequenz A' und Guanin (G) in der Sequenz B'. Somit ist im weiteren Verlauf der Analyse zu klären, ob dieser Unterschied auf einen Fehler bei Bestimmung der Sequenzen oder aber eine zufällig aufgetretene, evolutionäre Veränderung, eine sog. *Mutation*, zurückzuführen ist.

Zur effektiven Berechnung eines derartigen Alignments wird zunächst der Begriff selbst sowie die zugrundeliegende Interpretation von biomolekularen Sequenzen als Zeichenketten eines endlichen Alphabets definiert. Anschließend kann den ausgerichteten Zeichenketten ein geeignet definierter Wert für die Güte des Alignments zugeordnet werden, der eine Bestimmung des optimalen Alignments nach überwiegend biologisch motivierten Kriterien ermöglicht.

Definition 2.3: *Sei Σ ein endliches Alphabet ohne das Leerzeichen '-' und $\Sigma' = \Sigma \cup \{'-'\}$, sowie $A = a_1 \ldots a_n$ und $B = b_1 \ldots b_m$ mit $a_i, b_j \in \Sigma$ zwei Zeichenketten über diesem Alphabet, d.h. $A, B \subseteq \Sigma^*$. Ein* Alignment *von A und B bezeichnet somit zwei Zeichenketten $A' = a'_1 \ldots a'_k$ und $B' = b'_1 \ldots b'_k$ mit $a'_i, b'_j \in \Sigma'$ und $n, m \leq k \leq n + m$ mit der Eigenschaft, daß A' bzw. B' ohne Leerzeichen der ursprünglichen Zeichenkette A bzw. B entspricht.*

Sei $c(A', B') = \sum_{i=1}^{k} d(a'_i, b'_i)$ mit $d(a'_i, b'_i) > 0, \forall a'_i \neq b'_i$ der Wert bzw. die

Kosten eines Alignments. Dann bezeichnet $d(A,B) = \min_{\{A',B'\}} c(A',B')$ *die* Distanz *der Zeichenketten* A *und* B*.*

Das in Definition 2.3 vorausgesetzte Alphabet entspricht jeweils dem Typ der untersuchten Sequenzen, also etwa $\Sigma = \{A,C,G,T\}$ für DNA-Sequenzen oder $\Sigma = \{A,C,G,U\}$ für RNA-Sequenzen (vgl. Abschnitt 2.3.1). Der Wert bzw. die Kosten eines gegebenen Alignments repräsentiert die auftretenden Unterschiede an den einzelnen Positionen der ausgerichteten Zeichenketten A' und B'. Hierbei sind grundsätzlich drei Fälle zu unterscheiden:

Deletion - Aus Sequenz A wurde an der i-ten Stelle ein Zeichen entfernt, d.h. im Alignment befindet sich an dieser Position in B' ein Leerzeichen $'-'$.

Insertion - In Sequenz B wurde an der i-ten Stelle ein Zeichen eingefügt, d.h. im Alignment befindet sich an dieser Position in A' ein Leerzeichen $'-'$.

Substitution - In Sequenz B wurde an der i-ten Stelle ein Zeichen aus A ersetzt, d.h. im Alignment befinden sich an dieser Position in A' und B' unterschiedliche Zeichen.

Durch eine geeignete Wahl der Kosten für diese drei unterschiedlichen Veränderungen beider Sequenzen können spezifische biologische Gegebenheiten berücksichtigt werden, beispielsweise durch geringere Kosten für Deletion und Insertion gegenüber Substitution oder geringe Kosten für eine Substitution chemisch ähnlicher Aminosäuren in Protein-Sequenzen. Die Berechnung des Werts eines in dieser Hinsicht optimalen Alignments über die gesamte Länge der beteiligten Sequenzen läßt sich rekursiv auf die bereits ermittelten Werte für deren Präfixe zurückführen:

Satz 2.1: *Sei* $D_{i,j} = d(a_1 \dots a_i, b_1 \dots b_j)$ *die Distanz zweier Teilsequenzen* $a_1 \dots a_i, b_1 \dots b_j$ *der Zeichenketten* $A = a_1 \dots a_n$ *und* $B = b_1 \dots b_m$ *mit* $A, B \subseteq \Sigma^*$*. Weiterhin sei* $D_{0,0} = 0$, $D_{0,j} = \sum_{k=1}^{j} d(-, b_k)$, $D_{i,0} = \sum_{k=1}^{i} d(a_k, -)$.
Dann gilt: $D_{i,j} = \min\{D_{i-1,j} + d(a_i,-), D_{i-1,j-1} + d(a_i,b_j), D_{i,j-1} + d(-,b_j)\}$
Der Wert eines optimalen Alignments beträgt somit $D_{n,m}$*.*

Diese Tatsache ergibt sich aus den drei oben aufgeführten Möglichkeiten zur Verlängerung eines bereits optimalen Alignments für Präfixe aus A und B. Die eigentliche Berechnung der Rekursion nach Satz 2.1 basiert auf dem Prinzip der *dynamischen Programmierung*, indem bereits gelöste Teilprobleme für eine spätere Verwendung zwischengespeichert werden. Ein derartiger Algorithmus berechnet den Wert eines optimalen Alignments mit einem Aufwand von $O(mn)$ Einzelschritten, wobei durch Verfolgung des Rechenwegs anschließend auch das Alignment selbst ausgegeben werden kann.

Neben der so zusammengefaßten Ermittlung eines sog. *globalen* Alignments zweier Sequenzen mit vergleichbarer Länge, ermöglichen Varianten dieses Verfahrens auch die Berechnung *semi-globaler* oder *lokaler* Alignments zwischen Sequenzen deutlich unterschiedlicher Länge. Hierbei werden ausgedehnte Lücken in einer der beiden Sequenzen mit vergleichsweise geringen Kosten belegt, um zusammengehörige Teilsequenzen möglichst lückenlos gegeneinander auszurichten. Auf diese Weise können beispielsweise auch funktional ähnliche Abschnitte von ansonsten sehr unterschiedlichen Genen zuverlässig identifiziert und verglichen werden. Demgegenüber eignet sich ein globales Alignment u.a. zur Bestimmung der Verwandtschaftsverhältnisse auf Basis der Übereinstimmung zwischen weitgehend vergleichbaren Genen, wie im folgenden erläutert wird.

Phylogenetische Bäume

Die Vielfalt der gegenwärtig existierenden Lebensformen läßt sich auf einen gemeinsamen Ursprung zurückführen, wie nicht zuletzt die übergreifende Gültigkeit des Genetischen Codes belegt (siehe Abschnitt 2.3.1). Diese kleine Gruppe urtümlicher Organismen hat sich im Verlauf der Evolution über ca. vier Milliarden Jahre hinweg fortgepflanzt, verändert und hierbei in unterscheidbare Entwicklungslinien aufgeteilt. Somit sind alle jemals auf der Erde vertretenen Arten miteinander verwandt, wobei der Grad ihrer Verwandtschaft durch den Zeitpunkt ihrer Trennung festgelegt wird. Daher kann angenommen werden, daß eng verwandte Arten eine verhältnismäßig große Anzahl ähnlicher Merkmale aufweisen, während sich entfernt verwandte Arten aufgrund des frühen gemeinsamen Vorfahren in der Regel deutlich voneinander unterscheiden. Diese beobachtbaren Merkmale werden herangezogen, um die Entwicklung der Arten in einer hierarchischen Struktur, dem sog. *Phylogenetischen Baum*, zu beschreiben.

Abbildung 2.8 zeigt ein schematisches Beispiel eines derartigen Phylogenetischen Baums in einer möglichen grafischen Darstellung. Die mit A, B, C, D und E beschrifteten Blätter des Baums repräsentieren gegenwärtig existierende Arten, während die inneren Knoten F, G, H und I gemeinsame Vorfahren bezeichnen, die häufig bereits ausgestorben sind. Durch die Topologie des üblicherweise binären Baums wird die bekannte oder angenommene Verwandtschaft der beschriebenen Arten wiedergegeben. Beispielsweise sind C und D über den gemeinsamen Vorläufer G verhältnismäßig eng verwandt. Hingegen ist die Verwandschaft zwischen C und E eher gering einzuschätzen, da sich ihre Entwicklung nach I bereits sehr früh im Verlauf der Evolution getrennt hat. Die geschätzten Zeiträume der Entwicklung sind in Abbildung 2.8 durch entsprechende Beschriftungen der Kanten des Baums angegeben, wo-

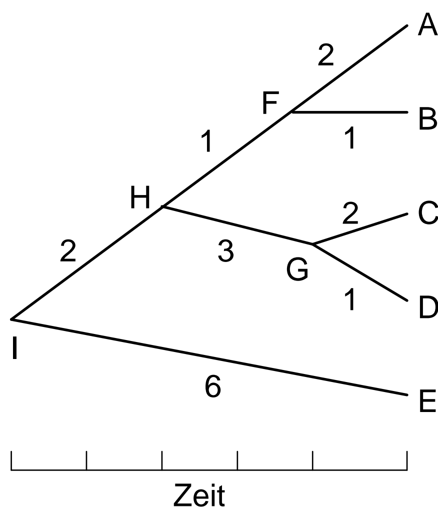


Abb. 2.8: *Schematisches Beispiel eines Phylogenetischen Baums*

bei in aller Regel nur relative Werte bestimmt werden können.

Neben dem wissenschaftlichen Interesse sind Kenntnisse über derartige Beziehungen gegenwärtig existierender Arten auch von großer medizinischer und wirtschaftlicher Bedeutung. Beispielsweise lassen sich neu entwickelte Medikamente zunächst an nahen Verwandten des Menschen testen, wobei die erhaltenen Ergebnisse über beabsichtigte Wirkung und mögliche Nebenwirkungen in vielen Fällen weitgehend übertragen werden können. Allerdings war die Ermittlung entsprechender Phylogenetischer Bäume vor Entdeckung der biomolekularen Zusammenhänge auf überwiegend morphologische Merkmale, d.h. unmittelbar ersichtliche makroskopische Eigenschaften, wie Form, Färbung, Anzahl der Extremitäten, u.ä., beschränkt. Die beigemessene Bedeutung solcher abgeleiteter Merkmale unterliegt aber zu großen Teilen der subjektiven Beurteilung des jeweiligen Forschers und führt somit zu sehr ungenauen, inkonsistenten oder sogar grundsätzlich fehlerhaften Phylogenetischen Bäumen.

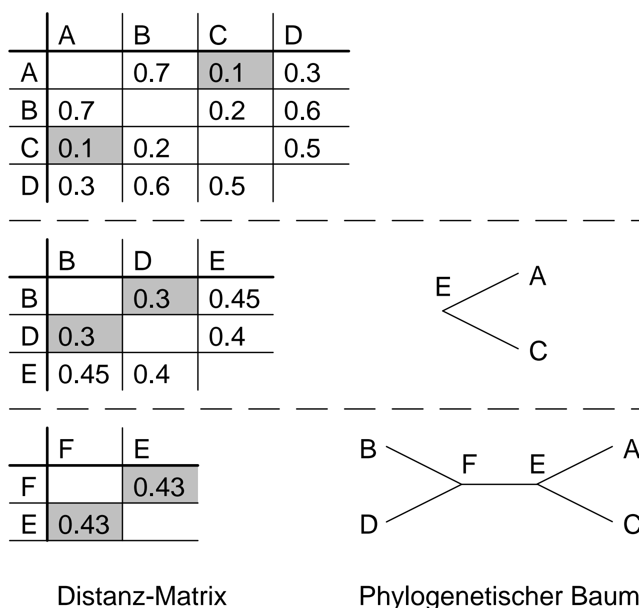
Hingegen ist das Genom eines Organismus bzw. seiner Spezies für sämtliche beobachtbaren Merkmale ursächlich und darüber hinaus durch entsprechende Verfahren weitgehend objektiv zu ermitteln. Daher ist es naheliegend, die Verwandtschaftsbeziehungen zwischen Arten durch Analyse ausgewählter Gene, also der Sequenz ihrer Nucleotide, zu bestimmen. Hierbei sind grundsätzlich *homologe*, d.h. mit vergleichbarer biologischer Funktion assoziierte Gene zu betrachten. Somit kann angenommen werden, daß sich diese aus einem gemeinsamen Vorläufer im Verlauf der Evolution entwickelt haben. Die

sich ergebenden Unterschiede in der Sequenz homologer Gene resultieren aus weitgehend zufälligen Veränderungen durch *Mutation* einzelner Positionen, etwa aufgrund von Einflüssen der Umwelt, sowie *Rekombination* mit bereits veränderten Genen, beispielsweise als Folge der sexuellen Fortpflanzung. Das Alignment homologer Sequenzen zeigt somit charakteristische Insertionen, Deletionen und Substitutionen, welche als Maß für die Ähnlichkeit bzw. Verwandschaft der zugehörigen Arten herangezogen werden können.

Die Klasse der *konstruktiven* Verfahren zur Rekonstruktion Phylogenetischer Bäume benutzt diese Informationen unmittelbar zum Aufbau hierarchischer Strukturen. So erfaßt der als *Neighbor Joining* bekannte Algorithmus zunächst alle wechselseitig ermittelten Unterschiede, beispielsweise den berechneten Wert des jeweiligen paarweisen Alignments, in einer symmetrischen Distanz-Matrix. Anschließend werden wiederholt die Sequenzen mit geringstem Abstand aus der Matrix entfernt und durch einen neu eingeführten Vertreter mit gemittelten Abstand zu den bisherigen Sequenzen ersetzt. Dies wird solange fortgeführt, bis nurmehr zwei Elemente als letzte Knoten in den entsprechend konstruierten Baum eingeführt werden. Dies führt, wie bei den meisten anderen Verfahren auch, zu einem Phylogenetischen Baum ohne ausgezeichnete Wurzel (engl. *unrooted tree*), da in aller Regel keine Informationen über die genaue evolutionäre Abfolge der beteiligten Arten verfügbar sind. Ein derartiger „Baum" beschreibt also lediglich die verwandtschaftlichen Beziehungen der untersuchten Sequenzen, nicht aber den zeitlichen Verlauf ihrer Entwicklung.

Der Ablauf eines solchen Verfahrens wird durch Abbildung 2.9 an Hand eines vereinfachten Beispiels verdeutlicht. Auf der linken Seite der Abbildung ist die sich jeweils ergebende, normierte Distanz-Matrix für die vier untersuchten Sequenzen A, B, C und D sowie deren abgeleitete Vorläufer dargestellt, während auf der rechten Seite die Konstruktion des Phylogenetischen Baums nachvollzogen wird. Innerhalb der Matrix ist darüber hinaus die jeweils geringste Distanz, also die größte Ähnlichkeit zweier Sequenzen, dunkel hinterlegt. Gemäß dem oben beschriebenen Algorithmus werden also im ersten Schritt die Sequenzen A und C durch einen neu eingeführten Vertreter E ersetzt, dessen Abstand zu B und D als arithmetisches Mittel der ursprünglichen Distanzen angenommen wird. Im nächsten Schritt werden nunmehr B und D über einen neuen Knoten F zusammengefaßt und das Verfahren terminiert. Der resultierende ungewurzelte Baum repräsentiert somit die verwandtschaftlichen Beziehungen der beteiligten Sequenzen auf Basis der ermittelten Distanzen.

Ein derartiges Verfahren liefert allerdings nur dann plausible Ergebnisse, falls die Anzahl der Veränderungen in allen Entwicklungslinien mit dem Zeitraum ihrer Trennung linear korreliert, die Evolution der untersuchten Gene



	A	B	C	D
A		0.7	0.1	0.3
B	0.7		0.2	0.6
C	0.1	0.2		0.5
D	0.3	0.6	0.5	

	B	D	E
B		0.3	0.45
D	0.3		0.4
E	0.45	0.4	

	F	E
F		0.43
E	0.43	

Abb. 2.9: *Exemplarischer Ablauf des* Neighbor Joining *Verfahrens*

also insgesamt gleichmäßig verlaufen ist. Zudem werden die zugrundeliegenden Sequenzdaten nur weitgehend undifferenziert in Form der berechneten Distanzwerte berücksichtigt, obwohl in vielen Fällen ausgewählte Positionen der Sequenz besonders charakteristisch für die evolutionäre Entwicklung des gesamten Gens sind. In der Praxis werden daher neben dem oben beschriebenen Verfahren auch andere Methoden eingesetzt, die weitergehende Modelle der biomolekularen Verhältnisse beinhalten. Hierbei handelt es sich in der Regel um *optimierende* Verfahren, die potentielle Phylogenetische Bäume bewerten und durch Anwendung geeigneter Heuristiken den bestmöglichen Baum ermitteln.

So berechnet etwa das *Maximum Parsimony* Verfahren die Anzahl der insgesamt erforderlichen Substitutionen, um die beobachteten Unterschiede der untersuchten Sequenzen an Hand des jeweiligen Baums zu erklären. Hierbei werden ausschließlich diejenigen Positionen des Alignments aller Sequenzen betrachtet, deren Zusammensetzung auch tatsächlich Rückschlüsse auf unterschiedliche Topologien der zugehörigen Bäume erlaubt. Die geringste Anzahl an Substitutionen kennzeichnet somit den gesuchten Phylogenetischen Baum, da eine entsprechende „Sparsamkeit“ (engl. *parsimony*) der natürlichen Evolution angenommen wird.

Abbildung 2.10 veranschaulicht diese Bewertung verschiedener Phylo-

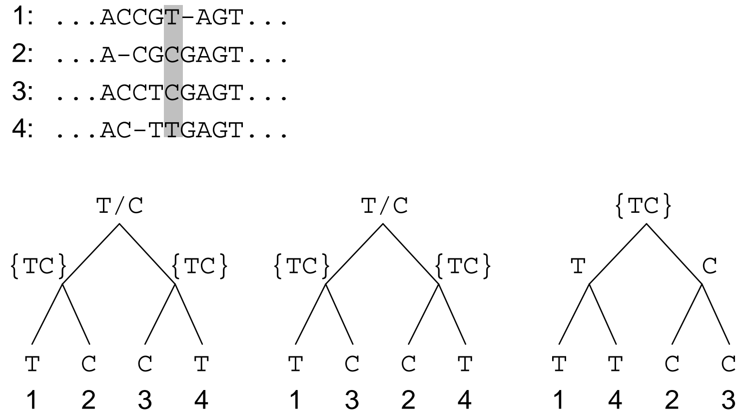


Abb. 2.10: *Exemplarische Anwendung des* Maximum Parsimony *Prinzips*

genetischer Bäume an Hand eines gegebenen Alignments aus vier DNA-Sequenzen. An der dunkel hervorgehobenen Position des Alignments unterscheiden sich die untersuchten Sequenzen: Während Sequenz 1 und 4 die Base Thymin (T) aufweisen, befindet sich bei Sequenz 2 und 3 an dieser Stelle die Base Cytosin (C). Die unterhalb des Alignments abgebildeten Phylogenetischen Bäume erklären diesen Unterschied durch eine mögliche Verwandtschaft der beteiligten Sequenzen. Hierbei beschreiben der linke und mittlere Baum eine Entwicklung, deren Ausgangspunkt als Base T oder C des gemeinsamen Vorfahren angenommen wird. In beiden Fällen sind jedoch in der Folge zwei Substitutionen erforderlich, um die entsprechende Base der Zwischenknoten in die beobachtete Base an den Blättern des Baums zu überführen. Demgegenüber impliziert die Topologie des rechten Phylogenetischen Baums lediglich *eine* Substitution zu Beginn der Entwicklung und kann somit als biologisch plausibel eingeschätzt werden.

Die Implementierung des Maximum Parsimony Verfahrens erfolgt durch einen rekursiven Algorithmus, der zunächst den betrachteten Baum von den Blättern zur Wurzel für eine gegebene Position des Alignments durchläuft. Hierbei wird jedem Knoten die Schnittmenge aus den vorliegenden Zeichen seiner Söhne zugeordnet, falls diese Schnittmenge nicht leer ist. Andernfalls wird die Vereinigungsmenge aller auftretenden Zeichen gebildet, da keine Aussage über die genaue Beschaffenheit des gemeinsamen Vorfahren getroffen werden kann. In einem zweiten Durchlauf durch den Baum wird nunmehr die Anzahl der Vereinigungen zweier Mengen gezählt; sie entspricht der minimal erforderlichen Anzahl an Substitutionen für die gegebene Topologie des Baums. Diese Berechnung wird für alle relevanten Positionen des Alignments

durchgeführt, die Ergebnisse aufsummiert und als maßgebliche Bewertung des Baums zurückgeliefert.

Aufgrund der exponentiell wachsenden Anzahl an Bäumen mit unterschiedlicher Topologie wird für die Untersuchung zahlreicher Sequenzen das Maximum Parsimony Verfahren mit einer geeigneten Heuristik kombiniert. Diese erlaubt es, nurmehr besonders aussichtsreiche Kandidaten zu berücksichtigen oder sich schrittweise an (sub-)optimale Lösungen anzunähern. Je nach Art des Problems und gewählter Implementierung werden hierfür in der Praxis unterschiedliche Heuristiken, wie etwa *Branch-and-Bound*, *Simulated Annealing* oder ein *Genetischer Algorithmus*, eingesetzt. Weiterhin sind neben Maximum Parsimony auch andere optimierende Verfahren zur Bestimmung Phylogenetischer Bäume gebräuchlich. Beispielsweise berücksichtigt das *Maximum Likelihood* Verfahren unterschiedliche Wahrscheinlichkeiten für das Auftreten einer Substitution im Verlauf der Entwicklung. Dementsprechend werden in diesem Zusammenhang diejenigen Bäume besser bewertet, deren Wahrscheinlichkeit aller implizierten Substitutionen besonders hoch ist. Allerdings ist das zugrundeliegende stochastische Modell durchaus komplex und daher dessen praktische Umsetzung relativ aufwendig.

Bereits die oben aufgeführten Beispiele zur Berechnung von Alignments und Phylogenetischen Bäumen verdeutlichen die vielfältigen Möglichkeiten und das enorme praktische Potential der biomolekularen Sequenzanalyse. Darüber hinaus werden noch zahlreiche weitere Anwendungen für wissenschaftliche, medizinische oder biotechnologische Fragestellungen eingesetzt oder entwickelt. Hierbei besteht in vielen Fällen weitgehende Einigkeit über die untersuchten Konzepte und ermittelten Ergebnisse, während die implementierten Algorithmen selbst als verschiedenartig und anspruchsvoll eingeschätzt werden können. Daher eignet sich dieser Anwendungsbereich in besonderer Weise für die Entwicklung komponentenbasierter Systeme, in denen entsprechende Softwarebausteine die grundlegende Funktionalität erbringen und über definierte Resultate oder Parameter verknüpft werden. Unter diesen besonderen Annahmen lassen sich auch umfangreiche, nicht-triviale Prototypen des späteren Systems weitgehend automatisiert erstellen, wie im folgenden Kapitel ausführlich erläutert wird.

3. EIN FRAMEWORK FÜR KOMPONENTENBASIERTES RAPID PROTOTYPING

Nachdem in den vorangegangenen Kapiteln die Entwicklung eines komponentenbasierten Ansatzes zur raschen Entwicklung funktionaler Software-Prototypen motiviert sowie die erforderlichen technischen und fachlichen Grundlagen hinreichend erläutert wurden, stellt das folgende Kapitel die erzielten Ergebnisse der Arbeit vor. Zunächst werden die grundlegenden Anforderungen an einen derartigen Ansatz aufgeführt. Diese bestimmen maßgeblich die Konzeption der erarbeiteten Lösung und werden später in Kapitel 6 zur Bewertung und Diskussion der erarbeiteten Ergebnisse herangezogen. Der anschließende Überblick erläutert die übergeordnete Struktur des vorgestellten Frameworks, führt die wesentlichen Themenfelder und Konzepte ein und setzt diese zueinander in Beziehung. Dies erleichtert die Orientierung in den folgenden Abschnitten, welche die Elemente des Frameworks im Detail beschreiben. Schließlich werden methodische Aspekte des Ansatzes untersucht und die zentralen Ergebnisse am Ende des Kapitels nochmals zusammengefaßt.

3.1 Anforderungen

Bevor eine geeignete Lösung entwickelt und vorgestellt wird, ist es zwingend erforderlich, die grundlegenden Anforderungen an einen Ansatz für komponentenbasiertes Rapid Prototyping zu definieren und zu priorisieren. Hierdurch wird der Charakter der gewählten Konzeption entscheidend geprägt und eine weitgehend objektive Bewertung der erzielten Ergebnisse ermöglicht. Darüber hinaus erlauben sie einen ausführlichen Vergleich mit existierenden Ansätzen, der in Kapitel 6 durchgeführt wird.

Die im folgenden aufgeführten Anforderungen resultieren aus der allgemeinen Beschreibung der Zielsetzung in Kapitel 1 sowie den maßgeblichen Eigenschaften und Merkmalen komponentenbasierter Softwareentwicklung, die in Abschnitt 2.2 erläutert werden. Sie repräsentieren daher sowohl eine Strukturierung als auch eine genauere Formulierung der gestellten Aufgabe. Mit ihrer Hilfe können die vorgeschlagenen Lösungen an die Komplexität

der Problemstellung angepaßt werden, wie später in Abschnitt 3.2 dargelegt wird.

3.1.1 Expressivität

Die Ausdruckskraft der angebotenen Mittel zur Beschreibung von Funktionalität ist offensichtlich von entscheidender Bedeutung für den Erfolg des gewählten Ansatzes. Sowohl die angebotene Funktionalität der verfügbaren Software-Komponenten als auch die erwünschte Funktionalität des zu erstellenden Gesamtsystems ist geeignet zu formulieren. Der Abgleich von angebotener mit erforderlicher Funktionalität ermöglicht somit im Anschluß die Konstruktion des gesuchten Prototypen.

Hierbei ist die zugrundeliegende Fragestellung von ausgesprochen fundamentaler Natur. Sie beinhaltet letztlich eine übergreifend gültige Interpretation des Begriffs *Funktion* in der komponentenbasierten Softwareentwicklung. Nur ein flexibles, verständliches und möglichst weitgehend anwendbares Modell zur Beschreibung von Funktionalität erlaubt die Suche und Auswahl geeigneter Softwarebausteine sowie deren zielgerichtete Verknüpfung zu übergeordneten Systemen mit gewünschter Funktionalität.

3.1.2 Komplexität

Neben der Expressivität bestimmt gerade auch die Komplexität des verwendeten Modells zur Beschreibung von Funktionalität den praktischen Erfolg des Ansatzes. Hierbei erstreckt sich der Bereich der denkbaren Lösungen von streng formalisierten und theoretisch fundierten Ansätzen, wie etwa Zustandsautomaten [HU79, Har87], Prozeßkalküle [ISO89], oder strombasierte Funktionen [BDD+92], über Klassifikation oder Indizierung mit festgelegtem Vokabular [PD87], bis hin zu überwiegend formlosen Beschreibungen in natürlicher Prosa [Sun00b].

Offensichtlich sollte der Grad der Formalisierung angemessen gewählt werden, so daß einerseits eine möglichst weitgehend automatisierte Auswahl und Verknüpfung der Komponenten erreicht werden kann, andererseits aber auch die Beherrschbarkeit und ein möglichst niedriger Einarbeitungsaufwand für den Anwender gewährleistet bleiben. Dieses Spannungsfeld zwischen Ausdruckskraft und Komplexität führt letztlich zu dem praxistauglichen Kompromiß des vorgestellten Ansatzes, wie später in Abschnitt 6.1 diskutiert wird.

3.1.3 Komponentenorientierung

Die in Abschnitt 2.2 angesprochenen Besonderheiten der komponentenbasierten Softwareentwicklung müssen in der Konzeption explizit berücksichtigt werden. Dies betrifft insbesondere den Einsatz von Komponenten unterschiedlicher Hersteller ohne Zugriff auf deren Quellcode. Somit ist eine möglichst weitgehende Unterstützung der Black-Box Wiederverwendung unerläßlich, weil die Implementierung derartiger Softwarebausteine voraussichtlich nur mit großem manuellen Aufwand geändert werden kann.

Unter idealen Voraussetzungen sind die Schnittstellen der verwendeten Komponenten standardisiert und auf Ebene der eingesetzten Programmiersprache zueinander technisch kompatibel bzw. typkonform. Jedoch zeigen praktische Erfahrungen, daß solche Bedingungen durch die Konkurrenz unabhängiger Hersteller, kurze Entwicklungszyklen, sowie langwierige Standardisierungsverfahren nicht zwingend vorausgesetzt werden können. Deshalb ist es wünschenswert, daß logisch zusammengehörige Komponenten zur Konstruktion eines Prototypen herangezogen werden können, selbst wenn deren Schnittstellen auf technischer Ebene inkompatibel sind.

3.1.4 Effektivität

Ein leistungsfähiges Framework zur Konstruktion funktionaler Prototypen sollte dem unbestimmten Charakter initialer Anforderungen Rechnung tragen. Gerade bei explorativem Prototyping ist zu Beginn weder der Umfang noch die genaue Ausprägung der geforderten Gesamtfunktionalität eindeutig festgelegt (vgl. Abschnitt 2.1). Daher müssen verschiedene Varianten funktionaler Prototypen aus einer gegebenen Beschreibung der Anforderungen abgeleitet werden. Dies ermöglicht einerseits dem Anwender die Konkretisierung der Anforderungen, erleichtert andererseits aber auch dem Entwickler die Wahl der am besten geeigneten Komponenten für das spätere Produkt.

Allerdings steigt mit Anzahl der Varianten auch der erforderliche Aufwand zur Konstruktion der Prototypen. Deshalb sollte der manuelle Aufwand für diesen Prozeß möglichst gering gehalten und dessen weitgehende Automatisierung angestrebt werden. Bei unvermeidlichen Eingriffen, etwa zur Verknüpfung von Komponenten mit technisch inkompatiblen Schnittstellen, ist darauf zu achten, daß diese möglichst einfach und an definierter Stelle vorgenommen werden können.

3.1.5 Technische Umsetzung

Wie in Abschnitt 2.2 angeführt, existiert gegenwärtig eine Reihe konkurrierender technischer Standards zur Entwicklung komponentenorientierter Sy-

steme, die zumindest in näherer Zukunft ihre praktische Bedeutung beibehalten werden [Szy98]. Daher sollte die grundlegende Konzeption eines Ansatzes für komponentenbasiertes Rapid Prototyping auf möglichst viele dieser technischen Infrastrukturen anwendbar sein. Dies betrifft sowohl das oben erwähnte Modell zur Beschreibung von Funktionalität, als auch die konkrete technische Umsetzung bei Generierung ausführbarer Prototypen.

Hierbei muß insbesondere sichergestellt werden, daß die logische Verknüpfung von Komponenten auf gängige technische Kommunikationsmechanismen, wie synchroner Aufruf einer Operation über eine Schnittstelle oder asynchroner Austausch von Nachrichten über sog. Callback-Schnittstellen, abgebildet werden kann. Dies erlaubt die Verwendung einer Vielzahl an verfügbaren Komponenten, ohne deren Implementierung aufwendig anzupassen, oder sogar eine eigene technische Infrastruktur zu etablieren.

3.1.6 Skalierbarkeit

Aufgrund der zumindest potentiell hohen Anzahl an existierenden Komponenten, die zur Konstruktion eines gegebenen Prototypen herangezogen werden können, sowie der daraus resultierenden, exponentiell wachsenden Anzahl an unterschiedlich zusammengesetzten Prototyp-Varianten, sind an die Effizienz eines praxisgerechten Verfahrens hohe Ansprüche zu stellen. Hierbei sind sowohl die eingesetzten Algorithmen zur Auswahl und Verknüpfung geeigneter Bausteine, als auch der erforderliche manuelle Aufwand, etwa bei Annotation oder Bewertung einzelner Komponenten, kritisch zu betrachten.

Benötigte Rechenleistung und Speicherplatzbedarf sollten deshalb höchstens linear mit der Anzahl betrachteter Komponenten wachsen. Darüber hinaus ist es wünschenswert, die eingesetzten Algorithmen den verfügbaren Ressourcen anpassen zu können, damit sich unterschiedlich leistungsfähige technische Plattformen nutzen lassen. Die erforderliche Interaktion mit dem Benutzer sollte flexibel gestaltet werden, so daß sich bei Bedarf die Qualität der konstruierten Prototypen durch zusätzlichen manuellen Aufwand steigern läßt.

3.1.7 Anwendbarkeit

Ein tragfähiger Ansatz für komponentenbasiertes Rapid Prototyping sollte sich bei erster Betrachtung für möglichst alle denkbaren Anwendungsbereiche und technische Infrastrukturen eignen. Tatsächlich unterscheiden sich die Charakteristika der verschiedenen Anwendungsbereiche sowie der dort eingesetzten Software-Systeme in der Praxis oftmals sehr deutlich. Daher ist eine Forderung nach universeller Anwendbarkeit schwerlich aufrecht zu erhalten

und kann den erzielten Ergebnissen sogar schaden.

Eine differenzierte Betrachtung der Anwendbarkeit erfordert somit eine kritische Diskussion, die deutlich darlegt, welche Eigenschaften und Merkmale eines Anwendungsbereichs durch den betreffenden Ansatz vorausgesetzt werden. Dies erlaubt letztlich eine Abschätzung des praktischen Nutzens, gerade falls noch keine umfassenden empirischen Erkenntnisse herangezogen werden können.

3.2 Überblick

Die oben aufgeführten Anforderungen sowie die in Kapitel 1 zusammengefaßte Einführung zur Aufgabenstellung bestimmen maßgeblich den Charakter des im folgenden vorgestellten Ansatzes. Aufgrund der Komplexität der Problemstellung und dem Anspruch, wesentliche Elemente der Konzeption durch eine Referenz-Implementierung praktisch zu validieren, ist allerdings eine vollständige, allgemein gültige und in sich geschlossene Lösung nicht zu erwarten. Diese Annahmen führen zur Entwicklung eines *Frameworks*, daß durch geeignete Modelle, Verfahren und ihre Zusammenhänge den konzeptuellen Rahmen für zukünftige, praxistaugliche Lösungen vorgibt.

In Anlehnung an die Konzeption eines Frameworks in der objektorientierten Systementwicklung wird hierbei angenommen, daß die Mehrheit der vorgegebenen Elemente für den späteren praktischen Einsatz tatsächlich angepaßt, erweitert oder sogar ersetzt werden muß. Trotzdem vermittelt das Framework ein grundlegendes Verständnis der zugehörigen Teilprobleme, möglicher Lösungsansätze und deren zielführende Verknüpfung. Darüber hinaus stellt es für jeden Teilbereich eine konkrete, bewußt einfach gehaltene Umsetzung dieser Ansätze bereit, die auch mit vertretbarem Aufwand zu implementieren ist (siehe Kapitel 4).

Durch diese Festlegung ergeben sich im folgenden bestimmte, offensichtliche Vereinfachungen und Einschränkungen, die jedoch in den wenigsten Fällen eine grundsätzliche Folge der gewählten Gesamtkonzeption sind, wie in Kapitel 6 diskutiert wird. Insbesondere sind die technischen Elemente des Frameworks so gestaltet, daß eine möglichst weitgehende Automatisierung des gesamten Verfahrens ermöglicht wird. Tatsächlich können bestimmte Teilergebnisse und Prozesse auch für eine eher manuelle Entwicklung funktionaler Prototypen herangezogen werden. Der erhöhte Grad an Interaktion mit dem Benutzer führt zu einer verbesserten Qualität der entwickelten Prototypen bei allerdings deutlich gestiegenem Aufwand. Aus diesen Gründen kann eine umfassende Beurteilung des Frameworks nur in Verbindung mit den in Kapitel 5 vorgestellten Erweiterungen getroffen werden. Sie verdeut-

lichen eine zukünftige Weiterentwicklung des Frameworks und lassen so die Grenzen, aber auch das Potential der erarbeiteten Ergebnisse erkennen.

Die Konzeption eines Frameworks erfordert eine klare, problembezogene Strukturierung in einzelne, thematisch zusammengehörige Elemente, die für eine Lösung des gestellten Problems nach den Regeln des Frameworks kombiniert werden. Im Kontext des komponentenbasierten Rapid Prototyping ergibt sich hieraus der Wunsch nach einer möglichst strikten Trennung zwischen der *logischen*, *anwendungsbezogenen Ebene* zur abstrakten Beschreibung von Anwendungsbereich und Funktionalität, sowie der *technischen*, *implementierungsbezogenen Ebene* zur konkreten Realisierung der gesuchten funktionalen Prototypen. Eine derartige Trennung führt zu einem insgesamt höheren Abstraktionsgrad und mithin zu einer besseren Beherrschbarkeit der zugrundeliegenden Komplexität. Darüber hinaus erlaubt ein hoher Abstraktionsgrad die anwendungsnahe Formulierung funktionaler Anforderungen und wird somit der besonderen Rolle des explorativen Prototyping im Software-Entwicklungsprozeß besser gerecht (vgl. Abschnitt 2.1). Schließlich ermöglicht die sorgfältige Abgrenzung zwischen anwendungsbezogener und implementierungsbezogener Ebene eine Abbildung der Konzeption auf verschiedene technische Infrastrukturen. Dies führt letztlich zu einer effektiven praktischen Umsetzung sowie einer dauerhaften Wiederverwendung der auf logischer Ebene erstellten Modelle.

Der in Abbildung 3.1 dargestellte schematische Überblick verdeutlicht die in dieser Arbeit vorgenommene Trennung zwischen den beiden oben erwähnten Bereichen. Auf logischer Ebene wird der Anwendungsbereich im Rahmen einer Ontologie durch *Konzepte* und sie verbindende *Relationen* definiert, während auf technischer Ebene eine geeignete Verknüpfung von Software-Komponenten über ihre Schnittstellen die gewünschte Funktionalität des Prototypen realisiert. Die notwendige Verbindung beider Ebenen wird durch Dokumentation der funktionalen Anforderungen (*Funktionale Spezifikation*) sowie dem zentralen Modell eines komponentenbezogenen Anwendungsfalls, dem sog. *Component Use Case* (CUC), gewährleistet. Letzterer beschreibt die grundlegende Funktionalität einer Komponente mit Bezug auf die Ontologie als typische Interaktionsmuster, die zur Verknüpfung auf technischer Ebene herangezogen werden können (siehe Abschnitt 3.4).

Konzepte der Ontologie repräsentieren in diesem Zusammenhang die wesentlichen, eindeutig identifizierbaren Entitäten eines Anwendungsbereichs. Hierbei handelt es sich um reale Objekte, etwa ein DNA-Molekül, oder auch abgeleitete, im strengen Sinn nicht-existente Konzepte, wie beispielsweise ein Alignment unterschiedlicher DNA-Sequenzen (vgl. Abschnitt 2.3). Rela-

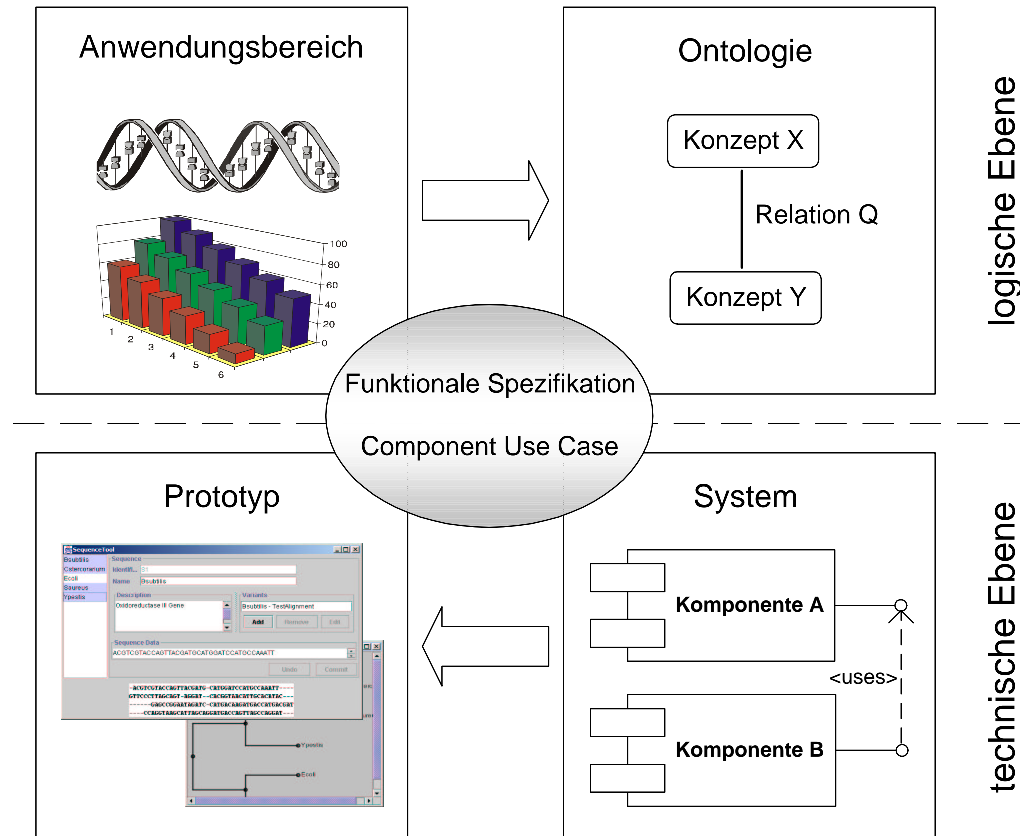


Abb. 3.1: *Schematische Übersicht über das Framework*

tionen hingegen beschreiben charakteristische Beziehungen zwischen diesen Entitäten, die typischerweise im Anwendungsbereich auftreten. Somit ist es z.B. möglich, die Zusammensetzung eines Proteins aus Aminosäuren durch eine entsprechende Relation zwischen den Konzepten Protein und Aminosäure zu modellieren. Die Einführung von Relationen mit vordefinierter Semantik, insbesondere der Generalisierungs- und Interpretationsbeziehung, erlaubt eine einfach anwendbare und dennoch ausdrucksvolle Formalisierung des Anwendungsbereichs, wie später in Abschnitt 3.3 ausführlich beschrieben wird.

Das Modell eines Software-Systems auf technischer Ebene entspricht dem in Abschnitt 2.2 beschriebenen Paradigma der komponentenbasierten Softwareentwicklung. Hierbei wird die tatsächlich grundlegende Funktionalität des Gesamtsystems ausschließlich durch wiederverwendbare Komponenten erbracht, die mittels möglichst einfacher programmiersprachlicher Konstrukte über ihre Schnittstellen verknüpft werden. Der im System erforderliche Kontroll- und Informationsfluß wird durch Aufrufe der über Schnittstellen

angebotenen Operationen realisiert. Dies ermöglicht sowohl synchrone als auch asynchrone Kommunikation zwischen den Komponenten. Ein derartiges Verständnis eines Software-Systems eignet sich besonders für die rasche Erstellung funktionaler Prototypen, wie in den folgenden Abschnitten erläutert wird.

Hierfür muß letztlich ein Modell für die Verbindung zwischen Anwendungsbereich und technischer Realisierung gefunden werden, daß den Abgleich zwischen vom Anwender geforderter und durch die Komponenten erbrachter Funktionalität ermöglicht. Für diese zentrale Aufgabe definiert die vorliegende Arbeit, wie oben bereits erwähnt, das Konzept eines komponentenbezogenen Anwendungsfalls. Auf logischer Ebene beschreibt ein solcher Anwendungsfall die grundlegende Funktionalität einer Komponente als *Manipulation von Konzepten des Anwendungsbereichs*, beispielsweise die Berechnung eines Alignments aus einer vorgegebenen Menge an DNA-Sequenzen als Calculate Alignment. Diese Kombination aus aktiven und passiven Elementen ähnelt der Verknüpfung von Verb und Objekt innerhalb eines Satzes der natürlichen Sprache. In Anlehnung an die typische Benennung eines herkömmlichen Anwendungsfalls in der Anforderungsanalyse der Softwareentwicklung [Jac92], erlaubt ein CUC somit eine weitgehend intuitive und anwendungsnahe Formulierung gewünschter oder erbrachter Funktionalität.

Hierbei ist das Subjekt des Satzes gewissermaßen implizit durch diejenige Komponente gegeben, der ein gegebener CUC zugeordnet ist. Sie erbringt die beschriebene Funktionalität durch eine *Interaktion* mit ihrer Umgebung, also anderen Komponenten des Systems oder dem Benutzer. Auf technischer Ebene führt eine solche Interaktion zu einer typischen Folge von Operationsaufrufen zwischen den Schnittstellen der beteiligten Komponenten. Dieses Interaktionsmuster wird durch einfache programmiersprachliche Konstrukte im Rahmen des betreffenden CUC beschrieben, um hieraus später den zur Verknüpfung notwendigen Quellcode zu generieren. Offensichtlich unterscheiden sich diese Interaktionsmuster je nach Komponente auch bei gleich benannten CUCs, während umgekehrt eine gegebene Komponente durchaus mehrere verschiedene CUCs erfüllen kann (siehe Abschnitt 3.4).

Abbildung 3.2 zeigt ein vereinfachtes Beispiel für den oben erwähnten Anwendungsfall Calculate Alignment. Er wird der Komponente `Clustal` zugeordnet und beschreibt, in welcher Reihenfolge die Operationen `setSequences` und `calcAlignment` der zugehörigen Schnittstelle `IfcClustal` aufgerufen werden müssen, um aus einer gegebenen Menge an DNA-Sequenzen ein mögliches Alignment zu berechnen. Darüber hinaus wird durch den CUC festgelegt, welche Konzepte des Anwendungsbereichs mit welcher Kardinalität als Parameter oder Rückgabewerte einer Operation erwartet werden. Im Beispiel betrifft dies eine Menge des Konzepts DNA als Parameter `s` der

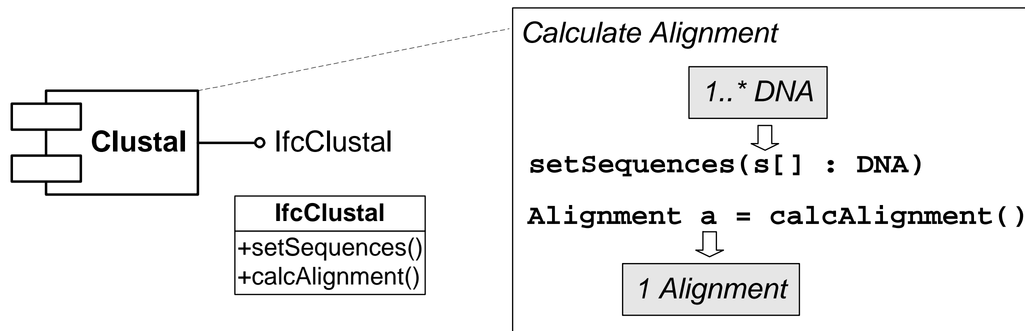


Abb. 3.2: *Beispiel eines Component Use Case*

Operation `setSequences` sowie ein einzelnes Konzept Alignment als Rückgabewert der Operation `calcAlignment`.

Hierbei ist zu beachten, daß die in der Schnittstelle bezeichneten Typen der Parameter oder Rückgabewerte üblicherweise nicht unmittelbar dem Namen der betreffenden Konzepte des Anwendungsbereichs entsprechen. Sie werden vielmehr als *technische Repräsentation* dieser Konzepte aufgefaßt, die bei Interaktionen zwischen Komponenten verschiedener Hersteller geeignet zu konvertieren sind. Es zeigt sich, daß die Repräsentation von Konzepten im Rahmen des Frameworks als Manipulation mit vordefinierter Semantik innerhalb eines CUC aufgefaßt werden kann. Hierdurch wird insbesondere unter bestimmten Voraussetzungen die automatische Generierung entsprechender *Adapter* ermöglicht, wie in Abschnitt 3.6 erläutert wird.

Sind alle verfügbaren Komponenten hinsichtlich ihrer grundlegenden Funktionalität in der oben skizzierten Weise beschrieben, so kann der Benutzer die geforderte Gesamtfunktionalität des Systems durch eine geeignete Kombination verschiedener Anwendungsfälle formulieren. Im vorgestellten Ansatz geschieht dies, wie oben bereits erwähnt, im Rahmen einer Funktionalen Spezifikation. Sie beinhaltet im einfachsten Fall eine Folge von Manipulationen, die über beteiligte Konzepte oder benannte Zwischenergebnisse verknüpft sind. Aufgrund der gewählten Modellierung ergibt sich somit wiederum eine weitgehend anwendungsnahe Beschreibung funktionaler Anforderungen. Das entwickelte Verfahren kann nunmehr für die aufgeführten Manipulationen und Konzepte passende Komponenten auswählen und an Hand der im jeweiligen CUC enthaltenen technischen Informationen verknüpfen, d.h. einen tatsächlich ausführbaren Prototypen generieren. Somit kann eine Funktionale Spezifikation gewissermaßen als „Rückgrat“ oder übergeordneter Bauplan des späteren Systems aufgefaßt werden (siehe Abschnitt 3.5).

Abbildung 3.3 illustriert diese Zusammenhänge an Hand einer beispiel-

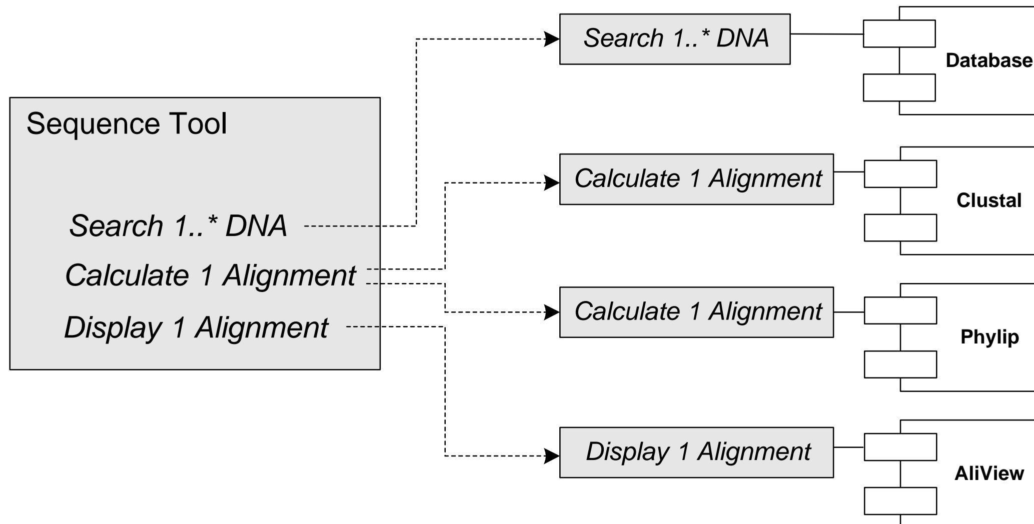


Abb. 3.3: *Beziehung zwischen Funktionaler Spezifikation und CUCs*

haften Spezifikation für ein einfaches Werkzeug zur Verarbeitung biomolekularer Sequenzdaten. Die auf der linken Seite dargestellte Spezifikation Sequence Tool beschreibt dessen geforderte Funktionalität als Suche von DNA-Sequenzen sowie Berechnung und Anzeige eines Alignments aus diesen Sequenzen. Die auf der rechten Seite abgebildeten Komponenten bieten entsprechende Anwendungsfälle an und können somit zur Konstruktion eines Prototypen herangezogen werden. Hierbei wird der CUC Calculate 1 Alignment sogar von zwei verschiedenen Komponenten `Clustal` und `Phylip` erfüllt, etwa aufgrund unterschiedlicher Algorithmen, welche durch diese Komponenten implementiert werden. An dieser Stelle führt der Abgleich zwischen erforderlicher und angebotener Funktionalität zu verschiedenen möglichen Prototyp-Varianten, die getrennt betrachtet werden sollten.

Auch wenn eine derartige, effektive Variantenbildung prinzipiell durchaus erwünscht ist, wie in Abschnitt 3.1.4 dargelegt, so ist in der Praxis doch eine unüberschaubar große Anzahl möglicher Kombinationen zu erwarten. Bei n gegebenen Positionen der Spezifikation, für die im Mittel k verschiedene Komponenten ausgewählt werden können, ergibt sich offensichtlich eine Anzahl von k^n möglichen, unterschiedlich zusammengesetzten Prototyp-Varianten. Darüber hinaus sind durch die bewußt einfach gehaltene Modellierung von Funktionalität auch zahlreiche, tatsächlich ungeeignete Varianten nicht auszuschließen.

Dieser grundlegenden Problematik begegnet der vorgestellte Ansatz durch ein iteratives Vorgehen bei der Konstruktion von Prototypen bzw. ihrer Varianten, sowie einer geeignet angepaßten Heuristik für ihre Weiterentwicklung, Bewertung und Optimierung. Im Rahmen eines sog. *Genetischen Algorithmus* [Gol89] wird die Konfiguration aus einer gegebenen Spezifikation und den sie erfüllenden Komponenten als *Individuum* einer gesamten *Population* möglicher Varianten aufgefaßt. Nach dessen Konstruktion wird jedem Individuum eine bestimmte *Fitneß* zugeordnet, die eine *Selektion* der „besten" Individuen erlaubt. Diese Auswahl unterliegt zu Beginn des nächsten Schritts der *Mutation*, also bestimmten zufälligen Veränderungen ihrer Konfiguration, sowie der *Rekombination* mit anderen selektierten Individuen. Diese veränderten oder neu entstandenen Varianten bilden die Grundlage der nächsten *Generation* einer Population. Der hierdurch skizzierte, evolutionäre Prozeß wird so lange durchgeführt, bis keine signifikante Verbesserung der Fitneß erzielt werden kann. Die im Genetischen Algorithmus gewählte Größe der Population, also der Umfang der betrachteten Prototyp-Varianten, kann hierbei flexibel den verfügbaren Ressourcen angepaßt werden. Dieses Vorgehen sichert somit die Skalierbarkeit des vorgestellten Ansatzes, wie in Abschnitt 3.6.5 diskutiert wird.

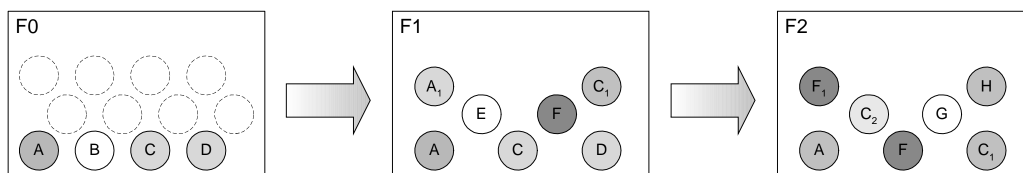


Abb. 3.4: *Schematischer Ablauf des Genetischen Algorithmus*

Die Grundidee einer solchen heuristischen Optimierung wird durch den schematischen Ablauf des Genetischen Algorithmus in Abbildung 3.4 erläutert. Zu Beginn werden aus der Gesamtheit aller möglichen Varianten zufällig bestimmte Individuen ausgewählt. Sie bilden die erste, in der Abbildung als F0 bezeichnete Generation. Eine Bewertung ihrer Fitneß erlaubt quantitative Vergleiche innerhalb der betrachteten Population. Hierbei entspricht die Farbe des Hintergrunds ihrer so bestimmten Fitneß, d.h. die jeweils dunkler hinterlegten Individuen besitzen im Beispiel die größere Fitneß. Somit werden die am besten bewerteten Individuen A, C und D für die nächste Generation F1 ausgewählt. Durch Mutation entstehen die Varianten A_1 und C_1, während die Rekombination zweier existierender Varianten die neuen Varianten E und F hervorbringt. Entsprechend den neu ermittelten Fitneßwerten führt die Selektion zur Auswahl von A, F und C_1 als erste

Individuen der nächsten Generation F2. Sie wird durch die über Mutation oder Rekombination erhaltenen Individuen F_1, C_2, G und H ergänzt. An dieser Stelle kann der evolutionäre Prozeß fortgeführt oder durch Ausgabe der besten Individuen F und F_1 beendet werden.

Der Erfolg einer solchen Heuristik wird offensichtlich maßgeblich durch die Wahl einer geeigneten Fitneß-Funktion bestimmt. Sie bestimmt das eigentliche Ziel der Optimierung durch eine Bewertung der Individuen nach Merkmalen der zugrundeliegen Problemstellung. Für die betrachteten Prototyp-Varianten schlägt die vorliegende Arbeit aus pragmatischen Gründen eine Kombination aus automatisierter und manueller Bewertung der Individuen vor. Die zuerst genannte Bewertung bezieht sich auf weitgehend technische Merkmale *aller* Varianten der gesamten Population, beispielsweise Anzahl der benötigten Komponenten und Adapter, Bereitstellung der erforderlichen Rollen und Parameter in Interaktionen u.ä., während in einem zweiten Schritt der Anwender die Tauglichkeit einer *deutlich kleineren Vorauswahl* nach eher anwendungsbezogenen, fachlich motivierten Gesichtspunkten vornimmt. Dies erlaubt dem Benutzer des Frameworks gewissermaßen die Evolution in Richtung der insgesamt besser geeigneten Prototypen zu beeinflussen. Schließlich kann letztlich nur er die Eignung der generierten Varianten hinsichtlich der gegebenen funktionalen Anforderungen abschließend beurteilen. Hierbei können die Gewichtung der verschiedenen Teilbewertungen sowie die grundlegenden Parameter der Heuristik, wie Populationsgröße oder Mutations- bzw. Rekombinationswahrscheinlichkeit, in weiten Grenzen den jeweiligen Gegebenheiten und Ressourcen angepaßt werden (siehe Abschnitt 3.6.5).

Falls die so erhaltenen Prototyp-Varianten dennoch nicht den Erwartungen des Anwenders entsprechen oder die initialen funktionalen Anforderungen nunmehr genauer formuliert werden können, führen entsprechende Veränderungen an der ursprünglichen Funktionalen Spezifikation zu einem neuen Zyklus des insgesamt iterativ organisierten, übergeordneten Prozesses. Eine mehrfache Wiederholung des oben skizzierten Ablaufs aus Abgleich der Funktionalität, Konstruktion der Varianten sowie deren Optimierung, erlaubt dem Benutzer somit eine schrittweise Annäherung an einen möglicherweise geeigneten Prototypen für das zu entwickelnde System. In jedem Fall erhält der Entwickler einen umfassenden Überblick über die verfügbaren funktionalen Komponenten.

Nach der vorangegangenen Übersicht werden in den folgenden Abschnitten die Elemente des erarbeiteten Frameworks im Detail vorgestellt. Hierfür werden überwiegend Beschreibungstechniken der *Unified Modeling Language* (UML) [RJB98] eingesetzt, weil diese in der Softwareentwicklung als be-

kannt vorausgesetzt werden können und so den unmittelbaren Bezug zur Referenz-Implementierung des Frameworks ermöglichen (siehe Kapitel 4). Die Beschreibung der erarbeiteten Ergebnisse folgt im wesentlichen der oben beschriebenen Aufteilung des Frameworks in logisch zusammengehörige Bereiche (vgl. Abbildung 3.1). Zunächst wird in Abschnitt 3.3 erläutert, wie sich ein gegebener Anwendungsbereich als Ontologie modellieren läßt und welche Zusammenhänge aus dem entstandenen Modell abgeleitet werden können. Anschließend wird in Abschnitt 3.4 das zentrale Konzept des komponentenbezogenen Anwendungsfalls als Vermittler zwischen Anwendungsbereich und technischer Realisierung entwickelt. Die Beschreibung *gewünschter* Funktionalität folgt der gewählten Modellierung von *angebotener* Funktionalität und wird durch die in Abschnitt 3.5 betrachtete Funktionale Spezifikation zusammengefaßt. Mit Hilfe dieser Modelle kann nunmehr eine weitgehend automatisierte Generierung funktionaler Prototypen erreicht werden. Die hierfür erarbeiteten Verfahren werden in Abschnitt 3.6 an Hand eines durchgängigen, vereinfachten Anwendungsbeispiels erläutert. Es bezieht sich, wie auch die zuvor zur Illustration herangezogenen Beispiele, auf den in Abschnitt 2.3 eingeführten Anwendungsbereich der biomolekularen Sequenzanalyse.

3.3 Ontologie

Informationssysteme werden überwiegend entwickelt und eingesetzt, um den Menschen in seinem beruflichen Umfeld und den dazugehörigen Aufgaben und Tätigkeiten zu unterstützen. Ein solcher Anwendungsbereich ist allgemein gekennzeichnet durch real existierende Objekte, abstrakte Konzepte oder Ideen, sowie zahlreichen Beziehungen zwischen diesen Bestandteilen. Beispielsweise bezeichnet der Begriff „Adenin“ in der Biochemie ein bestimmtes Molekül, der Begriff „Purin“ eine im strengen Sinn nicht existierende Kategorisierung von Adenin und Guanin, während das Konzept „DNA“ vereinfacht als sequentielle Komposition der Moleküle Adenin, Guanin, Cytosin und Guanin aufgefaßt werden kann[1].

Die eindeutige Benennung dieser Objekte, Konzepte und Beziehungen der realen Welt erlauben es dem Menschen, Informationen zu dokumentieren, auszutauschen und zusammen mit bekannten Interpretationen sowie anwendbaren Regeln neues Wissen zu erlangen. Auch wenn heutige Software-Systeme die menschliche Flexibilität und den intelligenten Umgang mit Informationen bei weitem nicht erreichen, so beinhalten sie doch explizit oder implizit eine bestimmte Sicht auf die reale Welt bzw. den Bereich, in dem sie

[1] Tatsächlich handelt es sich bei Nukleinsäuren um eine Folge von Nukleotiden, die aus Zucker, Phosphat und einer nitrogenen Base bestehen (siehe Abschnitt 2.3).

eingesetzt werden. Sie umfaßt u.a. die in der Software repräsentierten Konzepte und ihre Beziehungen, beispielsweise das zugrundeliegende Schema einer relationalen Datenbank oder das Klassenmodell einer objekt-orientierten Anwendung.

Eine derartige Aufteilung des Anwendungsbereichs wird in den Forschungsarbeiten zur Künstlichen Intelligenz als *Konzeptionalisierung* bezeichnet. Die explizite Repräsentation einer Konzeptionalisierung nennt man *Ontologie*, ein ursprünglich der Philosophie entlehnter Begriff für eine systematische Aufzeichnung des Existierenden [GU96, Gru93]. Sie beinhaltet notwendigerweise ein Vokabular aus benutzten Begriffen bzw. Namen für Konzepte und Relationen, sowie deren Definition, also eine gewisse Spezifikation ihrer Bedeutung. Somit ergibt sich letztlich ein gemeinsames Verständnis des Anwendungsbereichs, das je nach Ausprägung der Ontologie für menschliche Kommunikation, Interoperabilität zwischen Software-Systemen, Wiederverwendung oder Systemspezifikation genutzt werden kann [GU96].

Da in der Praxis zahlreiche ähnliche, aber doch unterschiedliche Auffassungen des Begriffs Ontologie gebräuchlich sind [GG95], wird im folgenden die in dieser Arbeit vorausgesetzte Bedeutung zunächst informell eingeführt:

Definition 3.1: *Eine* Ontologie *definiert die Verhältnisse und Zusammenhänge in einem oder mehreren zusammengehörigen Anwendungsbereichen bzw.* Domänen. *Zu diesem Zweck beinhaltet jede aufgeführte Domäne eine Menge an eindeutig bezeichneten* Konzepten *sowie* Relationen, *welche diese Konzepte zueinander in Beziehung setzen. Die beabsichtigte Bedeutung eines gegebenen Konzepts wird durch die zugehörige Domäne, seinen Namen sowie die Menge aller ihn betreffenden Relationen vollständig festgelegt. Die beabsichtigte Bedeutung einer gegebenen Relation wird entweder fest vorgegeben oder durch ihren Namen gekennzeichnet.*

Die praktische Anwendung einer so aufgefaßten Ontologie erfordert allerdings ein genauer festgelegtes, formal definiertes Modell, das sich letztlich auch in einem Software-System abbilden läßt. Hierbei wird der gewählte Grad der Formalisierung durch den Verwendungszweck bestimmt. Er reicht von natürlicher Sprache für die ausschließliche Kommunikation zwischen Menschen, über einfache formale Repräsentationen mit definierter Grammatik für wissensbasierte Systeme [Gru93, GF92, SWA+94] bis hin zu streng formalisierten Ansätzen mit genau definierter Semantik, Axiomen und Ableitungsregeln, die weiterführende Aussagen und Beweise erlauben [FG98]. Das in dieser Arbeit verfolgte Ziel einer anwendungsnahen Verknüpfung unterschiedlicher Softwarekomponenten erfordert in diesem Zusammenhang ein eher einfach gehaltenes Modell der zugrundeliegenden Ontologie. Es sollte

einerseits das in Anwendungsfällen benutze Vokabular hinreichend genau definieren, andererseits aber auch für durchschnittlich ausgebildete Entwickler beherrschbar sein (vgl. Abschnitte 3.1.2 und 3.1.4). Schließlich besitzt das unterstützte Prototyping selbst grundsätzlich explorativen Charakter mit der Aufgabe, die endgültigen funktionalen Anforderungen zu ermitteln oder zu konkretisieren, wie in Abschnitt 2.1 beschrieben ist. Somit wäre eine strenge und weitreichende Formalisierung, etwa im Sinne ausführbarer Spezifikationen [LAK+95], der Problemstellung nicht angemessen.

Aus diesen Gründen wird in der vorliegenden Arbeit eine Lösung gewählt, die gängigen objekt-orientierten Modellierungstechniken ähnelt und daher für die Mehrheit der Software-Entwickler intuitiv verständlich ist. Sie basiert ursprünglich auf einer umfassenden Ontologie der Molekularbiologie, die im Rahmen des TAMBIS-Projekts [BBB+99] erstellt wurde, um heterogene Datenbanken für biomolekulare Sequenzen zu integrieren. Allerdings wurde diese Ontologie nahezu vollständig überarbeitet, erweitert und für die vorliegende Problemstellung angepaßt. Insbesondere das im folgenden vorgestellte modulare, objekt-orientierte Modell einer Ontologie repräsentiert einen eigenständigen Beitrag dieser Arbeit. Da ein derartiges Modell Informationen beschreibt, welche selbst wiederum ein Modell der Wirklichkeit bilden, findet sich hierfür in der Literatur häufig zur besseren Unterscheidung auch die Bezeichnung *Metamodell* (vgl. [OMG00]).

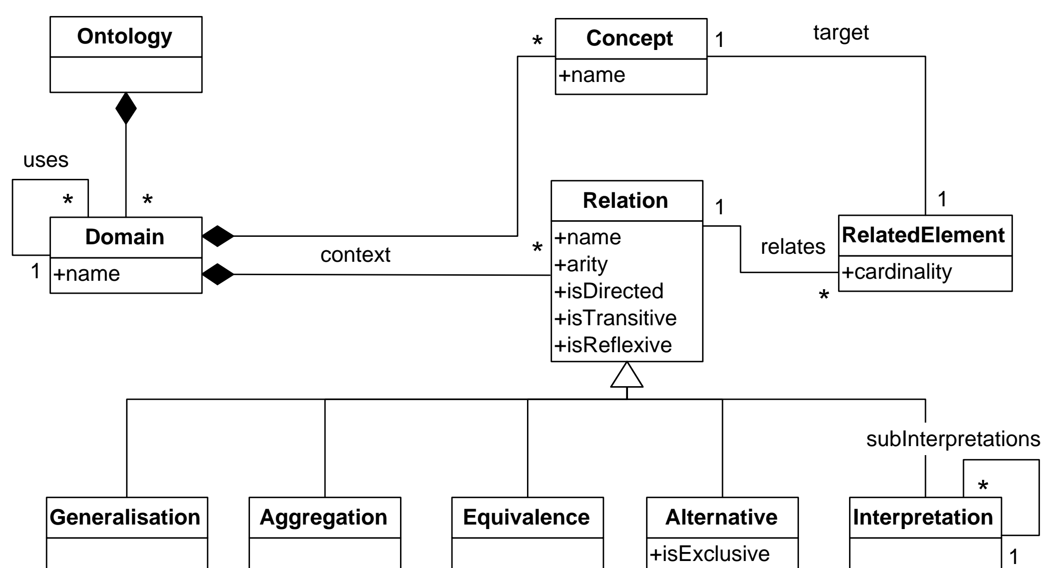


Abb. 3.5: *Metamodell der Ontologie*

Das eingeführte Metamodell entspricht einer formalen Beschreibung von Definition 3.1 und wird in Abbildung 3.5 als UML-Klassendiagramm mit englischen Bezeichnungen dargestellt. Demnach besteht eine Ontologie (Ontology) aus beliebig vielen Domänen (Domain), die verschiedene Anwendungsbereiche über ihren eindeutigen Namen repräsentieren, etwa den Bereich der Biochemie über den Namen `Biochemistry`. Jede Domäne beinhaltet in ihrem Kontext beliebig viele Konzepte (Concept) und Relationen (Relation), die wiederum über eindeutige Namen unterschieden werden. Hierbei repräsentieren Relationen charakteristische Beziehungen zwischen Konzepten, die im betreffenden Anwendungsbereich üblicherweise auftreten. Beispielsweise kann die Tatsache, daß ein gegebenes Molekül ein bestimmtes Molekulargewicht besitzt über eine Relation `hasWeight` zwischen den Konzepten `Molecule` und `MolecularWeight` im Kontext der Domäne `Chemistry` ausgedrückt werden. Darüber hinaus läßt sich die Kardinalität der assoziierten Konzepte (RelatedElement) über ein entsprechendes Attribut angegeben. Dies erweist sich als nützlich, um bei binären Relationen prinzipiell zwischen 1:1-, 1:n- und n:m-Beziehungen zu unterscheiden, kann aber auch der genaueren Modellierung eines Anwendungsbereichs dienen, etwa um zu beschreiben, daß ein tetrameres Protein aus genau 4 Untereinheiten besteht.

Neben den Konzepten und Relationen, die unmittelbar innerhalb einer Domäne definiert werden, besteht zusätzlich die Möglichkeit, andere Domänen bzw. deren Elemente zu importieren und an geeigneter Stelle zu referenzieren. Beispielsweise kann die Domäne `SequenceAnalysis` das Konzept `DNA` aus der Domäne `Biochemistry` importieren und mittels der lokalen Relation `hasIdentifier` einen eindeutigen Namen zuordnen. Diese Beziehung ist im Kontext der Sequenzanalyse durchaus sinnvoll, schließlich werden DNA-Sequenzen üblicherweise in Datenbanken gespeichert und über eindeutige Bezeichner verwaltet. Im Rahmen der Biochemie wäre eine solche Beziehung allerdings unsinnig, da reale chemische Moleküle offensichtlich keinen derartigen Bezeichner aufweisen. Die so importierten und durch eine neue, ungerichtete Beziehung veränderten Konzepte sind als Bestandteil der neuen Domäne aufzufassen und ggf. durch die Angabe des Kontexts von den ursprünglichen Konzepten zu unterscheiden. Auf diese Weise fördert der Import von Konzepten und Relationen die strukturierte, logisch motivierte Entwicklung der erstellten Modelle — eine wünschenswerte Eigenschaft, die darüber hinaus auch durch die Einführung von ausgezeichneten Relationen mit vordefinierter Bedeutung unterstützt wird.

Für die genauere Charakterisierung einer Relation werden zusätzliche Attribute verwendet, die ihre Stelligkeit (engl. *arity*), also die Anzahl der in Beziehung gesetzten Konzepte, sowie typische Eigenschaften wie Richtung, Transitivität und Reflexivität bestimmen. Die jeweilige Ausprägung dieser

Attribute kann für eine kompakte Modellierung des Anwendungsbereichs sowie weiterführende Ableitungen herangezogen werden. Zu diesem Zweck definiert das erarbeitete Metamodell eine Reihe von übergreifenden Relationen, die im folgenden näher erläutert werden.

Die Generalisierung (Generalisation) beschreibt eine binäre, gerichtete, irreflexive und transitive Relation, die eine Verallgemeinerung oder Gruppierung von Konzepten modelliert. Hierdurch kann beispielsweise ausgedrückt werden, daß `DNA` eine Spezialisierung des Konzepts `NucleicAcid` innerhalb der Domäne `Biochemistry` darstellt. Die zugrundeliegende Semantik der Generalisierung erlaubt es, alle Relationen des allgemeinen Konzepts auf das spezielle Konzept zu übertragen. Für ein importiertes und durch weitere Relationen verändertes, allgemeines Konzept gilt diese Aussage ebenfalls, d.h. auch sämtliche neuen Relationen übertragen sich auf dessen Spezialisierungen, die somit implizit in die neue Domäne übernommen werden. Zur Unterscheidung bei späterem Bezug auf die betreffenden Konzepte muß wiederum der jeweils erwünschte Kontext angegeben werden. Falls etwa die Domäne `SequenceAnalysis` das Konzept `NucleicAcid` aus der Domäne `Biochemistry` importiert und durch Definition einer Relation `hasIdentifier` erweitert, so überträgt sich diese Beziehung auch auf das speziellere Konzept `DNA`. Dieses wird daher ebenfalls zu einem Bestandteil der Domäne `SequenceAnalysis`, wie auch alle anderen, möglicherweise vorhandenen Spezialisierungen von `NucleicAcid`.

Es ist zu beachten, daß ein gegebenes Konzept durchaus als Spezialisierung *verschiedener* übergeordneter Konzepte aufgefaßt werden kann, das Konzept `DNA` z.B. auch als `Entry` einer Sequenz-Datenbank innerhalb der Domäne `SequenceAnalysis`. Die aus objekt-orientierten Ansätzen bekannten Probleme mit der sog. Mehrfach-Vererbung spielen hierbei keine Rolle, da eine Ontologie kein Verhalten der Konzepte spezifiziert. Offensichtlich können möglicherweise auftretende Namenskonflikte bei der Übertragung von Relationen durch eine geeignete Konvention vermieden werden.

Eine der Generalisierung ähnliche Semantik besitzt die Äquivalenz-Relation (Equivalence). Diese binäre, ungerichtete, reflexive und transitive Beziehung beschreibt, daß die durch sie assoziierten Konzepte innerhalb einer gegebenen Domäne als gleich anzusehen sind. Deshalb können alle bestehenden Relationen wechselseitig übertragen und die äquivalenten Konzepte im gleichen Kontext synonym benutzt werden. Dies erleichtert eine anwendungsgerechte Modellierung und erlaubt die Verwendung gängiger Abkürzungen, etwa die Bezeichnung `DNA` für das Konzept `DesoxyRibonucleicAcid` innerhalb der Domäne `Biochemistry`.

Die Aggregation ist eine binäre, gerichtete, irreflexive und nicht-transitive Relation, die den Aufbau eines Konzepts aus anderen Konzepten model-

liert. So kann beispielsweise in der Domäne `SequenceAnalysis` die Tatsache beschrieben werden, daß ein `Alignment` aus gegeneinander ausgerichteten Nukleinsäure-Sequenzen (`AlignedNucleicAcid`) besteht. Diese Art von Beziehung findet sich in vielen unterschiedlichen Anwendungsbereichen, so daß eine übergreifende Modellierung sinnvoll erscheint. Die besonders starke Assoziation zwischen den beteiligten Konzepten motiviert eine zugrundeliegende Semantik, die das zusammengesetzte Konzept nur in Verbindung mit seinen Teilkonzepten als eigenständiges Element der Ontologie versteht.

Gelegentlich können bestimmte Eigenschaften eines Konzepts aus einer kleinen Auswahl vorgegebener Möglichkeiten abgeleitet werden. So besitzt etwa die Ladung (`Charge`) eines Ions als Konzept der Domäne `Chemistry` entweder positives oder negatives Vorzeichen. Für einen solchen Fall definiert das Metamodell der Ontologie die n-stellige, gerichtete, irreflexive und nicht transitive Relation Alternative. Sie erlaubt die kompakte Repräsentation einer derartigen Aufzählung, wobei deren Exklusivität, d.h. die Tatsache daß jeweils nur eine bestimmte Auswahlmöglichkeit getroffen werden darf, durch ein besonderes Attribut der Beziehung gekennzeichnet ist.

Neben der oben erwähnten Möglichkeit, Konzepte direkt aus anderen Domänen zu importieren und mit ihnen neue Beziehungen zu assoziieren, erlaubt der vorgestellte Ansatz ein weiteres, verfeinertes Vorgehen zur Verbindung unterschiedlicher Domänen. Die Interpretationsbeziehung (Interpretation) als binäre, gerichtete, irreflexive und nicht-transitive Relation definiert, daß ein gegebenes Konzept innerhalb einer bestimmten Domäne als Konzept einer anderen Domäne aufgefaßt werden kann. Im Unterschied zur einfachen Spezialisierung eines importierten Konzepts werden jedoch die ursprünglichen Relationen nicht einfach ohne weiter Struktur übertragen. Vielmehr besteht die Möglichkeit, über explizit angegebene Sub-Interpretationen eine kontextsensitive Mehrfach-Übersetzung zu spezifizieren.

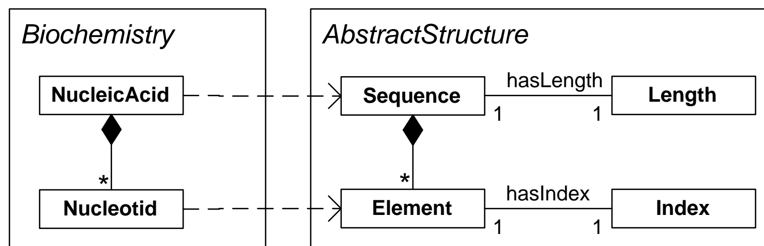


Abb. 3.6: *Interpretationsbeziehung für das Konzept* `NucleicAcid`

Das in Abbildung 3.6 dargestellte Beispiel erläutert diese Zusammenhänge an Hand einer Verbindung zwischen den Domänen `Biochemistry` und

AbstractStructure. Hierbei wird in Anlehnung an die UML-Beschreibungstechnik des Klassendiagramms ein Konzept durch ein beschriftetes Rechteck, eine domänenspezifische Relation durch eine durchgezogene Linie, eine Aggregation durch eine Linie mit ausgefüllter Raute und eine Interpretation durch einen unterbrochenen Pfeil visualisiert. Die Angabe der Kardinalität erfolgt ebenfalls nach Vorgaben der UML, d.h. ein $*$ repräsentiert beliebig viele Teilnehmer an der gekennzeichneten Relation. Somit modelliert der gezeigte Ausschnitt der Ontologie, daß innerhalb der Domäne `Biochemistry` eine Nukleinsäure (`NucleicAcid`) aus beliebig vielen Nukleotiden (`Nucleotid`) besteht. Innerhalb der Domäne `AbstractStructure` wird das Konzept einer Sequenz (`Sequence`) mit einer bestimmten Länge (`Length`) definiert. Die Sequenz besteht aus beliebig vielen Elementen, die wiederum einen bestimmten Index aufweisen. Die Interpretation des Konzepts `NucleicAcid` als `Sequence` in Verbindung mit der zugehörigen Sub-Interpretation von `Nucleotide` als `Element` erlaubt nunmehr die kontextsensitive Übertragung ausgewählter Relationen. Beispielsweise ist es somit möglich, auf den Index eines Nukleotids der Nukleinsäure zu referenzieren — eine Beziehung die außerhalb der betrachteten Interpretation offensichtlich nicht sinnvoll ist.

Die Interpretation repräsentiert somit eine häufig angewandte Technik, die Bedeutung eines gegebenen Konzepts durch den Vergleich mit anderen, bereits bekannten Konzepten zu erläutern. Im Unterschied zur Generalisierung ist dieser Vergleich allerdings nur für den beschriebenen Ausschnitt gültig, d.h. darüber hinaus gehende Beziehungen oder Ableitungen können nicht getroffen werden. Ihr Nutzen liegt v.a. in der modularen, weitgehend redundanzfreien Modellierung der verschiedenen Anwendungsbereiche innerhalb der Ontologie. Die jeweiligen Konzepte und Relationen werden in logisch zusammengehörigen Domänen definiert, wobei wichtige Querbezüge durch Import, Generalisierung oder Interpretation lokal in den betreffenden Domänen angegeben werden können. Diese klare Strukturierung erleichtert Erstellung, Benutzung und Pflege der Ontologie.

Es liegt nahe, die innerhalb der Ontologie definierten Informationen über Konzepte und ihre Beziehungen zur Klärung zentraler Fragestellungen im Zusammenhang mit der übergeordneten Zielsetzung des vorgestellten Ansatzes heranzuziehen. So ist es von entscheidender Bedeutung zu klären, inwieweit ein Bezug auf ein gegebenes Konzept durch andere Konzepte substituiert werden kann, ohne die Bedeutung und den Kontext der ursprünglichen Aussage völlig zu verlieren. Im Rahmen des Frameworks wird Funktionalität u.a. durch Bezug auf Konzepte der Ontologie beschrieben, wie später in Abschnitt 3.4 erläutert wird. Somit ermöglicht die Substitution dieser Bezüge

beispielsweise die Auswahl, Verwendung und Verknüpfung von Komponenten, die zumindest *wahrscheinlich* erwünschte Funktionalität bereitstellen, ohne jedoch exakt das gleiche Konzept zu referenzieren.

Im folgenden werden daher schrittweise geeignete Begriffe von *semantischer Kompatibilität* zwischen Konzepten erarbeitet. Sie basieren auf vorhandene Übereinstimmungen zwischen den Beziehungen der betrachteten Konzepte, denn schließlich ist die Gesamtheit aller existierenden Beziehungen, neben dem eigentlichen Namen, für die Definition eines Konzepts maßgeblich. So ist etwa innerhalb der Domäne `Biochemistry` eine Spezialisierung des Konzepts `Biopolymer`, die in einer Aggregationsbeziehung zu beliebig vielen Elementen des Konzepts `Nucleotide` steht, semantisch sehr ähnlich zum Konzept `NucleicAcid`, auch wenn dessen Name nicht übereinstimmt. Mit dieser Vorüberlegung wird zunächst die Gleichheit von Konzepten definiert:

Definition 3.2: *Zwei Konzepte A und B einer gegebenen Ontologie sind* äquivalent *bzw.* vollständig semantisch kompatibel, *d.h. jeder Bezug auf A kann durch B ersetzt werden (und umgekehrt), falls*

- *ihre zugehörige Domäne sowie ihr jeweiliger Name übereinstimmt,*
- *oder ihre zugehörige Domäne übereinstimmt sowie A und B in einer (möglicherweise transitiven) Äquivalenzbeziehung zueinander stehen.*

Im ersten Fall der obigen Definition sind die betrachteten Konzepte tatsächlich identisch, da eine eindeutige Namensgebung innerhalb einer Domäne vorausgesetzt werden kann. Der zweite Fall hingegen entspricht genau der oben beschriebenen Semantik der Äquivalenz-Beziehung, d.h. *alle* definierten Beziehungen des einen Konzepts sind in gleicher Weise auch für das andere Konzept gültig. Somit können Bezüge auf die beteiligten Konzepte innerhalb des betreffenden Kontexts gegeneinander ausgetauscht werden. Die Transitivität der Äquivalenz-Beziehung überträgt diese Eigenschaft offensichtlich auf alle durch sie verbundenen Konzepte[2].

Neben dieser strengen Definition von vollständiger Kompatibilität ergibt sich allerdings auch ein weiter gefaßter Begriff, welcher die Ersetzung eines allgemeinen durch ein spezielles Konzept innerhalb einer Aussage erlaubt. Er

[2] Sie bilden somit eine Äquivalenzklasse, bei der jedes Mitglied alle anderen Mitglieder gleichberechtigt repräsentiert. Daher gelten die weiteren Definitionen für unterschiedliche Formen von semantischer Kompatibilität zwischen Konzepten in gleicher Weise für alle Mitglieder der jeweiligen Äquivalenzklasse, auch wenn diese Tatsache aus Gründen der besseren Anschaulichkeit im folgenden nicht explizit berücksichtigt wird.

stützt sich auf die oben beschriebene Generalisierungsbeziehung, die sämtliche Relationen des übergeordneten Konzepts auf das untergeordnete Konzept überträgt.

Definition 3.3: *Zwei Konzepte* A *und* B *einer gegebenen Ontologie sind* zugesichert semantisch kompatibel*, d.h. jeder Bezug auf* A *kann durch* B *ersetzt werden, falls ihre zugehörige Domäne übereinstimmt, sowie* A *in einer (möglicherweise transitiven) Generalisierungsbeziehung zu* B *steht, also ein allgemeineres Konzept als* B *repräsentiert.*

Definition 3.3 entspricht der durchaus intuitiven Vorstellung, daß Aussagen über ein allgemeines Konzept in gleicher Weise für seine Spezialisierungen gelten. Dies beruht auf der Tatsache, daß derartige Aussagen nur die vorhandenen Beziehungen des allgemeinen Konzepts referenzieren können, welche sich offensichtlich nach Definition der Generalisierung bei jedem speziellen Konzept wiederfinden. Wiederum führt die Transitivität der Generalisierungsbeziehung zu einer Übertragung der semantischen Kompatibilität auf noch speziellere Konzepte. Der so definierte Kompatibilitätsbegriff ist im übrigen vergleichbar mit den Erkenntnissen über objekt-orientierte Typsysteme [CW85]. Hierbei können Zuweisungen, etwa bei Belegung der formalen Parameter einer Operation, nur von Subtypen (bzw. deren Instanzen) auf den übergeordneten Typ erfolgen. Der Compiler einer Programmiersprache mit einem solchen Typsystem kann diese Regel bereits bei der Übersetzung prüfen.

Die Bedeutung der oben beschriebenen Interpretationsbeziehung erlaubt darüber hinaus eine weitere, eingeschränkte Auffassung von semantischer Kompatibilität. Sie umfaßt die selektive Übertragung von Beziehungen zwischen Konzepten unterschiedlicher Domänen.

Definition 3.4: *Zwei Konzepte* A *und* B *einer gegebenen Ontologie sind* eingeschränkt semantisch kompatibel*, d.h. jeder Bezug auf* A *kann durch* B *ersetzt werden, falls* A *in einer Interpretationsbeziehung zu* B *steht, also* A *als* B *aufgefaßt werden kann, und alle möglicherweise zugehörigen Sub-Interpretationen beachtet werden, also deren referenzierte Konzepte gleichfalls ersetzt werden.*

Der so definierte Kompatibilitätsbegriff repräsentiert unmittelbar die ursprüngliche Intention für die Einführung der Interpretationsbeziehung. Unter Beachtung des Kontexts aller evtl. spezifizierten, untergeordneten Interpretationen können gegebene Konzepte als vergleichbar angesehen werden, d.h. Aussagen über das ursprüngliche Konzept gelten mit Einschränkungen

auch für das interpretierte Konzept. Dies resultiert aus der Übertragung ausgewählter Beziehungen, wobei die eigentlichen Partner jeder Beziehung erst durch die vorgegebenen Sub-Interpretationen festgelegt werden.

Eine letzte vorgestellte Auffassung von semantischer Kompatibilität behandelt schließlich die Umkehrung von Definition 3.3. In bestimmten Fällen kann also ein Bezug auf ein spezielles Konzept auch durch ein allgemeineres Konzept ersetzt werden.

Definition 3.5: *Zwei Konzepte A und B einer gegebenen Ontologie sind* potentiell semantisch kompatibel, *d.h. jeder Bezug auf A kann durch B ersetzt werden, falls ihre zugehörige Domäne übereinstimmt, sowie B in einer (möglicherweise transitiven) Generalisierungsbeziehung zu A steht, also ein allgemeineres Konzept als A repräsentiert.*

Diese Form der Substitution ist immer dann möglich, wenn eine Aussage über das spezielle Konzept ausschließlich Beziehungen nutzt, die so bereits für das allgemeine Konzept definiert sind. In diesem Fall bleibt die Aussage gültig, obwohl sie offensichtlich auch unmittelbar auf das allgemeine Konzept Bezug nehmen könnte. Beispielsweise läßt sich das Molekulargewicht eines Zuckermoleküls angeben, falls dessen chemische Summenformel bekannt ist. Diese besondere Aussage gilt allerdings auch für jedes andere Molekül. Somit kann in diesem Fall der vorliegende Bezug auf das Konzept Sugar durch dessen übergeordnetes Konzept Molecule ersetzt werden.

Tatsächlich kann eine Entscheidung über die Gültigkeit einer solchen Ersetzung erst bei genauer Kenntnis der jeweiligen Aussage getroffen werden. Daher verbieten gängige Typsysteme für objekt-orientierte Programmiersprachen eine derartige Substitution bei entsprechenden Zuweisungen. Trotzdem ist die Berücksichtigung dieser Definition von Kompatibilität zumindest auf konzeptioneller Ebene wünschenswert, um den Kreis potentiell geeigneter Komponenten zu erweitern (siehe Abschnitt 3.6.2). Außerdem können auf diese Weise evtl. auftretende Ungenauigkeiten bei Beschreibung von Anwendungsbereich oder Funktionalität ausgeglichen werden.

Die oben eingeführten Begriffe von Kompatibilität lassen allerdings zunächst nur eine diskrete Beurteilung der betrachteten Konzepte zu, d.h. an Hand der vorliegenden Ontologie kann ausschließlich festgestellt werden, ob gegebene Konzepte zueinander semantisch kompatibel sind oder eben nicht. In der Praxis ist aber eine quantitative Einschätzung der Kompatibilität wünschenswert, um den Grad der *Ähnlichkeit* zwischen den betrachteten Konzepten zu ermitteln. Schließlich kann beispielsweise eine Aussage über ein sehr allgemeines Konzept A wenig Bedeutung für ein sehr spezielles Konzept B besitzen, auch wenn A nach Definition 3.3 zugesichert semantisch

kompatibel zu B ist. Demgegenüber ist eine Aussage über ein potentiell semantisch kompatibles Konzept C für die Betrachtung von B möglicherweise sehr bedeutsam, falls es nur wenig spezieller als dieses ist.

Aus diesem Grund führt der vorgestellte Ansatz eine geeignete Metrik für die Ontologie ein, über die sich die semantische Nähe ihrer Konzepte bestimmen läßt. Hierfür wird zunächst jeder Generalisierungs- oder Interpretationsbeziehung r_{AB} zwischen zwei durch sie verbundenen Konzepten A und B eine *konzeptuelle Distanz* $d(r_{AB}) \in [0,1]$ zugeordnet, die ohne weitere Angaben als fester Wert 0.5 angenommen werden kann. Ein bei Erstellung der Ontologie explizit angegebener Wert für diese Distanz modelliert, wie semantisch ähnlich die betrachteten Konzepte zu verstehen sind. Hierbei repräsentieren Werte am unteren Ende des Intervalls große Ähnlichkeit, während Werte am oberen Ende eine eher entfernte Ähnlichkeit spezifizieren. Auf diese Weise können Anwendungsbereiche genauer charakterisiert werden, deren Modellierung eine heterogene Granularität aufweist, etwa weil diese noch nicht fertiggestellt ist oder bestimmte Teilbereiche sehr detailliert erfaßt sind.

Vergleichbar mit bekannten Arbeiten über *Semantische Netze* [Woo75, Fin79], wird eine Ontologie somit als gewichteter Graph aufgefaßt, wobei Konzepte die Knoten und Generalisierungs- sowie Interpretationsbeziehungen die gerichteten Kanten des Graphs repräsentieren. Weiterhin wird eine Hilfsfunktion $p(\mathsf{X},\mathsf{Y})$ vorausgesetzt, welche den nach der Gesamtdistanz kürzesten Pfad $\vec{p}_{XY}$ zwischen den gegebenen Konzepten X und Y berechnet, falls dieser existiert[3]. Ein solcher Pfad beinhaltet die Folge der Knoten zwischen X und Y (einschließlich X und Y selbst) sowie die zugehörigen Distanzen der sie verbindenden Relationen. Somit kann die semantische Kompatibilität $k(\mathsf{X},\mathsf{Y}) \in [0,1]$ als Funktion in Abhängigkeit des kürzesten Pfads angegeben werden. Um weiterhin zwischen den verschiedenen, in Definition 3.2 bis 3.5 festgelegten Formen der semantischen Kompatibilität zu unterscheiden, wird $k(\mathsf{X},\mathsf{Y})$ über die jeweils spezifischen Bewertungsfunktionen k_e, k_l, k_p definiert als

$$k(\mathsf{X},\mathsf{Y}) = \begin{cases} 1 & \text{falls X vollständig semantisch kompatibel zu Y} \\ k_e(\vec{p}_{XY}) & \text{falls X zugesichert semantisch kompatibel zu Y} \\ k_l(\vec{p}_{XY}) & \text{falls X beschränkt semantisch kompatibel zu Y} \\ k_p(\vec{p}_{XY}) & \text{falls X potentiell semantisch kompatibel zu Y} \\ 0 & \text{sonst} \end{cases} \tag{3.1}$$

[3] Unter der üblicherweise angenommenen Voraussetzung, daß keine zyklischen Generalisierungs- oder Interpretationsbeziehungen in der Ontologie zugelassen sind, ist die Funktion p wohldefiniert und kann offensichtlich mit linearem Aufwand in der Anzahl der Konzepte berechnet werden.

Hierbei repräsentiert ein Wert am unteren Ende des Intervalls $[0, 1]$ eine geringe semantische Kompatibilität, während ein Wert am oberen Ende des Intervalls eine hohe konzeptionelle Übereinstimmung signalisiert. Die genaue Definition der Funktionen k_e, k_l und k_p ist in diesem Zusammenhang von untergeordneter Bedeutung, sofern eine effektive und effiziente Berechnung gewährleistet ist. Sie können im wesentlichen frei gewählt werden, um je nach gewünschtem Grad der Toleranz einen flexiblen Abgleich der semantisch kompatiblen Konzepte zu erreichen, wie später in Abschnitt 3.6.2 ausführlich erläutert wird.

Typischerweise sind die eingesetzten Bewertungsfunktionen zwar monoton, aber weder linear noch stetig, d.h. eine hohe konzeptionelle Distanz führt zu einer überproportional geringen Kompatibilität oder nimmt bei Überschreiten eines festgelegten Schwellwerts unmittelbar den Wert 0 an. In Abschnitt 3.6.2 werden exemplarische Definitionen der oben eingeführten Bewertungsfunktionen vorgestellt, die zur toleranten Auswahl von Komponenten hinsichtlich ihrer Funktionalität herangezogen werden. Darüber hinaus tragen sie auch zur späteren, automatisierten Teilbewertung der generierten Prototypen bei (siehe Abschnitt 3.6.5). Somit repräsentieren die Bewertungsfunktionen k_e, k_l und k_p ein bedeutendes Element des übergeordneten Frameworks, das nach weiteren praktischen Erfahrungen der jeweiligen Problemstellung angepaßt werden kann.

Mit den oben beschriebenen Mitteln ist es möglich, den Anwendungsbereich eines Software-Systems hinsichtlich seiner wesentlichen Konzepte und Beziehungen für die Zwecke dieser Arbeit angemessen zu modellieren. Das so definierte Vokabular erlaubt es, die geforderte Funktionalität des Gesamtsystems sowie die angebotene Funktionalität der vorhandenen Komponenten anwendungsbezogen zu formulieren. Der vorgestellte Begriff der semantischen Kompatibilität zwischen Konzepten des Anwendungsbereichs ist hinreichend genau definiert, um den konkreten Bezug auf ein gegebenes Konzept unter bestimmten Voraussetzungen durch ein semantisch kompatibles Konzept zu ersetzen. Eine derartige Substitution ermöglicht eine gewisse Toleranz beim Abgleich zwischen gewünschter und angebotener Funktionalität. Dies erleichtert die Auswahl, Verknüpfung und Bewertung der beteiligten Komponenten wesentlich, wie später in Abschnitt 3.6 gezeigt wird.

3.4 Component Use Cases

Zu Beginn der Systementwicklung stellt sich grundsätzlich die Frage nach den gestellten funktionalen Anforderungen. Sie beschreiben, welche Funktio-

nalität das zu entwickelnde System dem späteren Anwender anbieten soll. Diese Anforderungen werden üblicherweise vom Entwickler bzw. einem geschulten Software-Analysten in enger Zusammenarbeit mit dem Anwender geklärt, modelliert und geeignet dokumentiert. Anschließend bilden die so erstellten Dokumente und Modelle die Grundlage der weiteren Entwicklungsschritte, wie Design oder Implementierung (vgl. Abbildung 2.1).

In Umkehrung dieser Verhältnisse stellt sich bei komponentenbasierter Softwareentwicklung zusätzlich die Frage nach den *erfüllten* funktionalen Anforderungen, d.h. welche Funktionalität von den vorhandenen, bereits implementierten Komponenten tatsächlich angeboten wird. Erst eine explizite Modellierung und Beschreibung dieser durch Komponenten erbrachten Funktionalität ermöglicht die Auswahl und Verknüpfung geeigneter Software-Bausteine zu einem Gesamtsystem.

Die offensichtlich verwandte Problemstellung legt eine Betrachtung der gängigen Techniken zur Analyse und Beschreibung funktionaler Anforderungen nahe. Moderne, häufig angewandte Modelle des Software-Entwicklungsprozesses, wie der *Unified Process* [BJR98] oder der *Rational Unified Process* [Kru98], empfehlen hierfür das zentrale Konzept des Anwendungsfalls (engl. *use case*). Dieses ursprünglich von Jacobson eingeführte Modell [Jac92] beschreibt eine Sequenz von Aktionen, die ein System durchführt, um dem externen Interaktionspartner ein für ihn nützliches Ergebnis zu liefern. Die Benennung eines Anwendungsfalls sollte diesen Zweck möglichst intuitiv wiedergeben. Typische Anwendungsfälle im Kontext eines Geldautomaten sind beispielsweise „Auszahlung von Bargeld“, „Überweisung auf ein Konto“ oder „Anzeige des Kontostands“. Sie bezeichnen klar definierte Interaktionen mit dem System, die dem Kunden einen beobachtbaren Nutzen erbringen.

In der vorliegenden Arbeit wird dieser bewährte Ansatz der Anforderungsanalyse auf die Spezifikation von Komponenten übertragen. Hierfür sind allerdings die zentralen Begriffe, wie *Anwendungsfall* oder *Interaktion*, genauer zu modellieren und in Bezug zu technischen Elementen der komponentenbasierten Softwareentwicklung zu setzen. Dies ist die maßgebliche Voraussetzung, um später ausführbare Prototypen des zukünftigen Systems zu generieren. Aus Gründen der besseren Übersicht werden zunächst die logischen, anwendungsbezogenen Teile des Modells vorgestellt. Seine anschließend aufgeführten, technischen Elemente sind jedoch als integraler Bestandteil aufzufassen, entsprechend dem Charakter eines komponentenbezogenen Anwendungsfalls als Bindeglied zwischen logischer und technischer Ebene des vorgeschlagenen Frameworks (vgl. Abbildung 3.1).

Abbildung 3.7 zeigt die anwendungsbezogene Ebene des gewählten Ansatzes als UML-Klassendiagramm, wobei die Richtung der benannten Assoziationen ihre beabsichtigte Interpretation vorgibt. Das Modell definiert

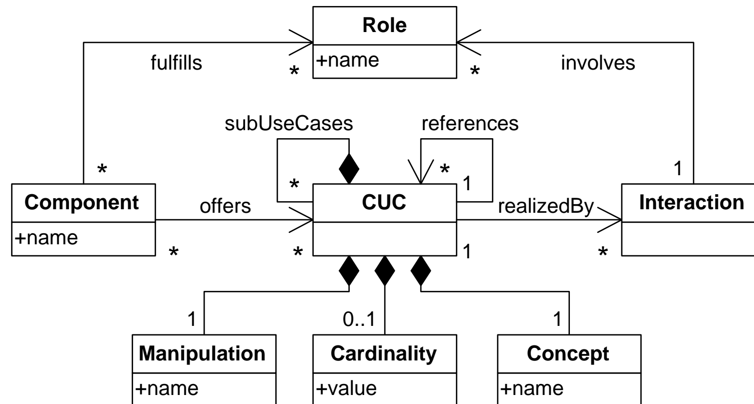


Abb. 3.7: *Logische Anteile eines Component Use Case*

demnach einen komponentenbezogenen Anwendungsfall (ComponentUseCase bzw. CUC) als Manipulation von Konzepten des Anwendungsbereichs, welche über eine Verknüpfung mit dem Element Concept durch eine geeignete Ontologie beschrieben sind. Optional kann eine explizit durch das Element Cardinality repräsentierte Kardinalität der manipulierten Konzepte spezifiziert werden. Die Manipulation selbst wird durch ihren Namen charakterisiert, der nach einer üblichen Konvention in Form eines Verbs angegeben wird. Somit lassen sich intuitiv verständliche Beschreibungen der angebotenen Funktionalität formulieren, etwa Search * DNA für die Suche nach einer Menge von DNA-Sequenzen in einer Datenbank oder Display 1 Alignment für die Anzeige eines Sequenz-Alignments auf dem Bildschirm.

Hierbei entwickelt das vorliegende Modell den Bezug zum Anwendungsbereich im wesentlichen über das in der Ontologie definierte Konzept, während Manipulationen zunächst nicht genauer interpretiert werden (vgl. aber Abschnitt 5.2.1). Ausnahmen dieser Regel stellen die als `Represent` oder `Adapt` bezeichnete Manipulationen dar, deren vordefinierte Semantik als Repräsentation von Konzepten zur automatischen Adapter-Generierung ausgenützt wird, wie später in Abschnitt 3.6.3 ausführlich beschrieben ist. Darüber hinaus werden bestimmte Manipulationen mit ausgezeichneten Namen wie `Select`, `Initialize` oder `Configure` zur Unterstützung der Code-Generierung herangezogen (siehe Abschnitt 3.6.4).

Die eigentliche Funktionalität eines derartigen Anwendungsfalls wird naturgemäß von Software-Komponenten (Component) erbracht, wobei ein gegebener CUC auf logischer Ebene durchaus von verschiedenen Komponenten erfüllt werden kann. Letztlich trifft erst der später beschriebene, techni-

sche Anteil eines CUC die Festlegung auf eine bestimmte Komponente. Aus pragmatischen Gründen kann diese Zuordnung statisch durch entsprechende Meta-Informationen erfolgen. Diese werden dem ausführbaren Format einer Komponente als entsprechendes Dokument hinzugefügt, etwa innerhalb des JAR-Archivs einer JavaBeans-Komponente (vgl. Abschnitt 2.2). Umgekehrt kann eine gegebene Komponente durchaus mehrere verschiedene CUCs anbieten, beispielsweise sowohl Display 1 Alignment als auch Edit 1 Alignment, falls neben der Anzeige eines Alignments auch dessen Bearbeitung ermöglicht wird. Somit ergibt sich insgesamt eine n:m-Beziehung zwischen den Elementen Component und CUC, die in Abbildung 3.7 mit offers bezeichnet ist.

Aufgrund ihrer Komplexität oder logischen Zusammengehörigkeit kann es sinnvoll erscheinen, eine Menge von Anwendungsfällen bezüglich *einer, fest zugeordneten* Komponente hierarchisch zu strukturieren. So kann etwa eine Komponente zur Benutzerverwaltung den übergeordneten Anwendungsfall Manage * User anbieten, welcher die untergeordneten CUCs Add 1 User, Delete 1 User und Edit 1 User umfaßt. Zu diesem Zweck definiert das Modell die rekursive Aggregation subUseCases, welche zur Beschreibung einer entsprechenden Struktur dient. Im Unterschied hierzu modelliert die rekursive Assoziation references eine bestehende, nicht durch Struktur motivierte Beziehung zwischen den Anwendungsfällen *verschiedener* Komponenten. Somit ist es beispielsweise möglich, die erwartete Repräsentation eines manipulierten Konzepts zu spezifizieren, wie später in Abschnitt 3.6.3 erläutert wird.

Im Übergang zum technischen Teil des Modells wird durch die Assoziation realizedBy zwischen den Elementen CUC und Interaction ausgedrückt, daß ein gegebener Anwendungsfall durch eine oder mehrere alternative Interaktionen zwischen Komponenten realisiert wird. Hierbei beschreibt der Begriff *Interaktion* eine typische Folge von Kommunikationsschritten zwischen den beteiligten Partnern, die möglichst einfach auf gängige technische Infrastrukturen für komponentenbasierte Systeme abgebildet werden kann. Im Unterschied zur konventionellen Auffassung eines Anwendungfalls, werden sämtliche Kommunikationspartner durch Software-Komponenten repräsentiert, die indirekt über ihre Rollen (Role) in der betreffenden Interaktion bestimmt sind. Es wird vorausgesetzt, daß die erforderliche Interaktion mit dem Benutzer durch eigene Komponenten mittels entsprechender CUCs behandelt wird, etwa die Auswahl einer bestimmten DNA-Sequenz über eine am Bildschirm präsentierte Liste durch den angebotenen CUC Select 1 DNA.

Die durch die Assoziationen involves und fulfills modellierte, indirekte Beziehung der beteiligten Interaktionspartner über Rollen repräsentiert eine Abstraktion von konkreten, eher technisch motivierten Elementen der eingesetzten Infrastruktur. Somit kann die bei der Generierung von Prototypen erfolgte Zuordnung zwischen Komponenten und Rollen nach anwendungsbezoge-

nen Kriterien erfolgen, gerade wenn technische Kompatibilität auf Basis der verwendeten Typen nicht vorausgesetzt werden kann (vgl. Abschnitt 3.1.3). Aus pragmatischen Gründen beinhaltet das vorgestellte Modell eine Reihe von vordefinierten Rollen mit festgelegten Namen. So bezeichnet `Self` den eigentlichen Anbieter eines Anwendungsfalls, also diejenige Komponente, der ein gegebener CUC statisch zugeordnet wurde, `Client` den wesentlichen Nutzer eines Anwendungsfalls, während `Listener` den Partner einer ereignisbasierten, asynchronen Kommunikation bestimmt (siehe Abschnitt 3.5).

Um die auf logischer Ebene spezifizierten Interaktionen auch auf technischer Ebene umsetzen zu können, ist offensichtlich ein entsprechendes Modell einer Komponente bzw. eines komponentenbasierten Systems erforderlich. Tatsächlich beschreibt ein derartiges Modell ein Informationssystem, welches selbst wiederum als Modell der Wirklichkeit aufgefaßt werden kann. Daher kann das übergeordnete Modell, wie im Fall der Ontologie, ebenfalls zur besseren Unterscheidung als Metamodell bezeichnet werden. Um die praktische Umsetzung der erarbeiteten Ergebnisse zu erleichtern, wird im folgenden ein bewußt einfach gehaltenes Metamodell eines komponentenbasierten Systems vorgestellt, das ohne Schwierigkeiten auf verschiedene existierende technische Infrastrukturen übertragbar ist, wie später in Kapitel 6 diskutiert wird.

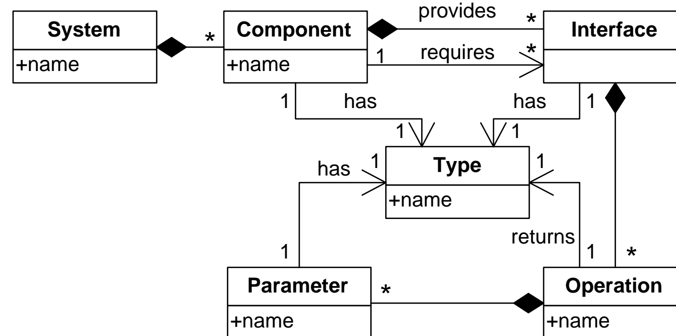


Abb. 3.8: *Metamodell eines komponentenbasierten Systems*

Abbildung 3.8 zeigt das zugrundeliegende Metamodell als UML-Klassendiagramm. Das System setzt sich demnach aus einer Menge von Komponenten zusammen. In Übereinstimmung mit Definition 2.2 wird eine gegebene Komponente im wesentlichen durch die von ihr bereitgestellten Schnittstellen (Interface) charakterisiert. Eine Schnittstelle beinhaltet Operationen mit Parametern und Rückgabewerten, über deren Aufruf die eigentliche Funktionalität einer Komponente genutzt werden kann. Benötigt eine Komponente Informationen oder Funktionalität aus ihrer Umgebung, so werden diese

Abhängigkeiten explizit über von ihr importierte Schnittstellen spezifiziert. Die technisch korrekte Zuordnung der aufgeführten Elemente bzw. ihrer Instanzen wird gewöhnlich über den Namen ihres Typs (Type) durch die Regeln und Werkzeuge der verwendeten Programmiersprache sichergestellt.

Gerade bei objekt-orientierten Ansätzen wird das Konzept eines Typs üblicherweise auch zur Überprüfung der logischen, anwendungsbezogenen Kompatibilität herangezogen, falls die Struktur der Vererbung zwischen Klassen des Systems entsprechend gestaltet ist. Die vorliegende Arbeit vermeidet diese durchaus problematische Vermischung beider Aspekte (vgl. Abschnitt 6.2.2) und verwendet Typen sowie deren Beziehungen ausschließlich auf technischer Ebene, beispielsweise zur Ermittlung eindeutiger Namen für Komponenten oder zur Kontrolle von Zuweisungen im generierten Code (siehe Abschnitt 3.6).

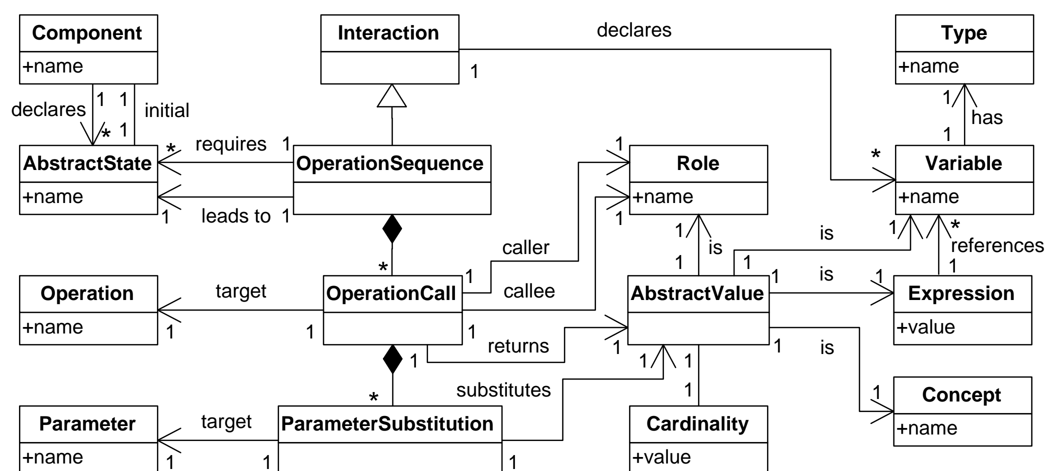


Abb. 3.9: *Technische Anteile eines Component Use Case*

Mit Hilfe dieses einfachen Metamodells für komponentenbasierte Systeme ist es nunmehr möglich, die auf logischer Ebene spezifizierten Interaktionen auf technischer Ebene zu realisieren. Abbildung 3.9 illustriert den in dieser Arbeit verfolgten Ansatz, eine Interaktion als Sequenz von Operationsaufrufen zu modellieren. Offensichtlich sind hierfür auch zahlreiche andere, fortgeschrittene Modelle zur Beschreibung von Verhalten denkbar, falls sie sich prinzipiell zur Generierung ausführbarer Prototypen eignen. In Kapitel 5 werden später entsprechende Erweiterungen vorgestellt und diskutiert.

Das gezeigte UML-Klassendiagramm definiert eine Sequenz von Operationsaufrufen (OperationSequence) als Spezialisierung einer Interaktion, die einen bestimmten Zustand (AbstractState) der zugeordneten Komponente

voraussetzt und nach ihrem Ablauf die Komponente in einen möglicherweise anderen Zustand versetzt. Diese von einer Komponente explizit zu deklarierenden Zustände geben hierbei in Übereinstimmung mit dem Prinzip der Black-Box Wiederverwendung keine Details der Implementierung bekannt, sondern beschreiben vielmehr die von außen beobachtbaren Unterschiede in ihrem Verhalten. Beispielsweise kann so die korrekte Initialisierung und Konfiguration einer Komponente beschrieben werden, etwa falls vor einer Sequenz-Analyse bestimmte Basisparameter des Verfahrens gesetzt werden müssen. Aus pragmatischen Gründen wird ein mit `Default` bezeichneter Zustand allen Komponenten des Systems implizit zugeordnet.

Ein einzelner Operationsaufruf (OperationCall) innerhalb der Sequenz spezifiziert die aufgerufene Operation einer Schnittstelle, die Rollen der aufrufenden bzw. aufgerufenen Komponente über die Assoziationen caller und callee, sowie die als zu ersetzende Parameter (ParameterSubstitution) übergebenen bzw. als Rückgabewert erhaltenen *abstrakten Werte* (AbstractValue). Ein solcher abstrakter Wert vermittelt zwischen den oben eingeführten, logischen Elementen und den durch Typen identifizierten, technischen Elementen des Modells. So kann ein gegebener formaler Parameter einer Operation einen Kommunikationspartner festlegen, benötigte anwendungsbezogene Daten übergeben oder vorher festgelegte Konstanten und Hilfsinformationen übermitteln. Dementsprechend verweist ein abstrakter Wert auf eine innerhalb der Interaktion spezifizierte Rolle, etwa dem `Listener` in der später beschriebenen, ereignisbasierten Kommunikation, auf ein definiertes Konzept der Ontologie, auf eine vorher deklarierte Variable oder auf einen zur Laufzeit auswertbaren Ausdruck (Expression) aus Konstanten und Variablen. Die eigentliche Belegung eines abstrakten Werts mit konkreten Instanzen der technischen Elemente erfolgt erst zum Zeitpunkt der Generierung, sobald die jeweilige Konfiguration des Prototypen festgelegt ist (siehe Abschnitt 3.6.4). Hierbei gilt die offensichtliche Einschränkung, daß jeweils nur eine der vier vorgesehenen Möglichkeiten realisiert wird. Zudem bezieht sich die angegebene Kardinalität eines abstrakten Werts nur auf Konzepte und Rollen.

Mit dieser auf synchrone Kommunikation ausgelegten Modellierung einer Interaktion lassen sich bereits zahlreiche komponentenbezogene Anwendungsfälle beschreiben, wie in dem folgenden Beispiel erläutert wird. Hierbei wird aus Gründen der Anschaulichkeit eine einfache, textuelle Notation eingeführt, obwohl die in Kapitel 4 vorgestellte Referenz-Implementierung ein komfortables Werkzeug mit grafischer Benutzeroberfläche zur vollständigen Spezifikation eines CUC bereitstellt. Die verwendeten Bezeichner für technische Elemente der Beispiele entsprechen den Java-Konventionen, ohne jedoch hierdurch eine bestimmte technische Infrastruktur zwingend festzulegen.

Abbildung 3.10 zeigt eine ausführliche Beschreibung des bereits zu-

```
Component: casa.analysis.Clustal
    Offered Interfaces:  casa.analysis.AlignmentAnalysis
    Required Interfaces: casa.data.Alignment, casa.data.DNASequence
    Signature:           casa.analysis.AlignmentAnalysis {
                            void setSequences(java.util.List s);
                            casa.data.Alignment calcAlignment(float gapPenalty);
                         }
    States:                <Default>
    Initial State:         <Default>
    Provided Use Cases:    Calculate 1 Alignment
          Referenced Use Cases: casa.data.SeqImpl/Represent 1 DNA
          Interactions:    Standard from <Default> to <Default>
                           Self.setSequences(1..* DNA)
                           1 Alignment = Self.calcAlignment(0.75)
```

Abb. 3.10: *CUC-Beschreibung zur Komponente* `Clustal`

vor in Abschnitt 3.1 angeführten Beispiels zur Berechnung eines globalen Alignments aus einer Menge gegebener DNA-Sequenzen. Die beschriebene Komponente `casa.analysis.Clustal` bietet hierfür die Schnittstelle `casa.analysis.AlignmentAnalysis` an, während sie selbst die Schnittstelle `casa.data.DNASequence` benötigt. Die Signatur der angebotenen Schnittstelle spezifiziert die Operationen `setSequences` und `calcAlignment` mit ihren formalen Parametern und Rückgabewerten, sowie deren jeweilige Typen. Hierbei erfordert die Operation `setSequences` einen Parameter des Typs `java.util.List` aus der Java-Klassenbibliothek zur Angabe einer Menge von DNA-Sequenzen, während der Parameter `gapPenalty` der Operation `calcAlignment` die Ergebnisse des implementierten Algorithmus über einen Gewichtungsfaktor beeinflußt (vgl. Abschnitt 2.3). Weiterhin wird beschrieben, daß die angeführte Komponente keine besonderen, von außen beobachtbaren Zustände aufweist und somit grundsätzlich im impliziten Zustand `<Default>` verbleibt.

Die angebotene Funktionalität der Komponente wird durch den entsprechenden Anwendungsfall Calculate 1 Alignment bezeichnet, also die Kombination aus der Manipulation `Calculate`, dem manipulierten Konzept `Alignment` sowie dessen Kardinalität. Darüber hinaus verweist die Beschreibung des Anwendungsfalls auf den CUC Represent 1 DNA einer anderen Komponente `casa.data.SeqImpl`, welcher die erwartete technische Repräsentation einer DNA-Sequenz spezifiziert. Diese explizite Referenz ist erforderlich, weil die Operationen der Schnittstelle `java.util.List` nur auf generischen

Objekten des Typs `java.lang.Object` definiert sind. Somit kann die Überprüfung der technischen Kompatibilität erst zur Laufzeit erfolgen. Zu diesem Zeitpunkt sind auftretende Fehler jedoch deutlich schwerer zu lokalisieren und zu beheben.

Die für die Berechnung eines Alignments benötigen Kommunikationsschritte werden in der Interaktion `Standard` zusammengefaßt. Sie beschreibt eine Sequenz von zwei Operationsaufrufen, welche den Zustand der Komponente nicht verändern. Im ersten Schritt wird bei der Komponente selbst – bezeichnet durch die vordefinierte Rolle `Self` – die Operation `setSequences` aufgerufen, wobei der formale Parameter `s` durch eine nicht-leere Menge des Konzepts `DNA` ersetzt wird. Anschließend kann das Konzept `Alignment` als Ergebnis der Berechnung über den Aufruf der Operation `calcAlignment` erhalten werden. Hierbei wird der Ausdruck `0.75` als typischer, konstanter Wert für den formalen Parameter `gapPenalty` substituiert. Die Zuordnung der verwendeten Repräsentation für das Konzept `Alignment` ergibt sich implizit über den Typ des Rückgabewerts der Operation `calcAlignment`.

Die oben beschriebene Interaktion beschränkt sich auf eine *synchrone* Kommunikation zwischen den angegebenen Partnern, d.h. alle Operationsaufrufe werden bei Erfüllung des Anwendungsfalls nacheinander ausgeführt und die beteiligten Komponenten solange blockiert, bis die Ergebnisse vorliegen. Darüber hinaus ist aus praktischen Überlegungen, etwa zur Realisierung einer grafischen Benutzerschnittstelle, zusätzlich die Unterstützung von *asynchroner* Kommunikation wünschenswert. Hierbei kann die Ausführung einer Interaktion zu vorher nicht festgelegten Zeitpunkten erfolgen, erzwingt kein Warten auf Bestätigungen oder bereitgestellte Ergebnisse und erlaubt somit eine weitgehend lose Kopplung der beteiligten Partner. In der Praxis wird dieser Typ der Kommunikation häufig über sog. Ereignisse (engl. *event*) und begleitende, vordefinierte Interaktionsmuster realisiert. Typischerweise registrieren sich Interessenten für ein bestimmtes Ereignis zunächst bei der entsprechenden Komponente. Sobald das betreffende Ereignis eintritt, etwa durch eine bestimmte Aktion des Benutzers, werden die registrierten Komponenten über den Aufruf einer ausgezeichneten Operation aus einer Callback-Schnittstelle benachrichtigt. Die auslösende Komponente kann anschließend ihre Ausführung fortsetzen, während die am Ereignis interessierten Kommunikationspartner geeignet reagieren, beispielsweise durch Ermittlung des aktuellen Zustands der auslösenden Komponente.

Die vorliegende Arbeit integriert eine derartige, ereignisbasierte Kommunikation mit dem in Abbildung 3.11 vorgestellten Modell. Es erlaubt eine gegebene Interaktion mit einem sog. Ereignis-Muster (EventPattern) zu assoziieren. Letzteres beinhaltet das eigentliche Ereignis (Event) als Kombination aus Manipulation und betroffenem Konzept, sowie die benötigten Operati-

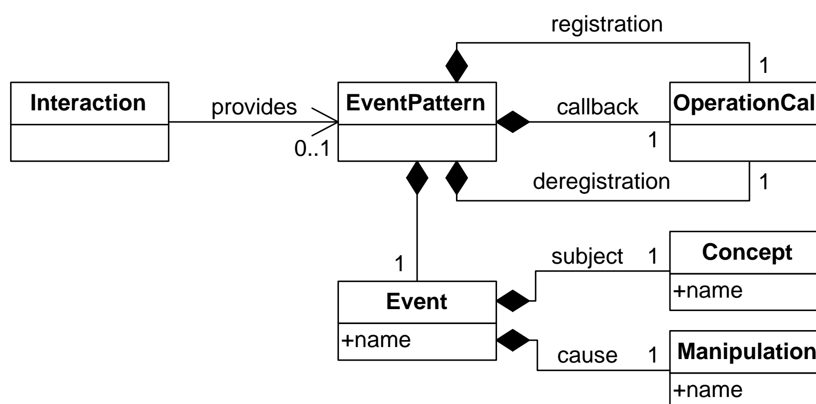


Abb. 3.11: *Modell der ereignisbasierten Kommunikation*

onsaufrufe für Anmeldung, Abmeldung und Rückruf bei Eintreten des Ereignisses. Hierbei werden durch das Ereignis selbst keine weiterführenden Informationen übermittelt – etwaige Ergebnisse des übergeordneten Anwendungsfalls werden, wie oben beschrieben, durch eine gesondert spezifizierte Interaktion bereitgestellt.

Abbildung 3.12 illustriert das oben beschriebene Modell an Hand der beiden Anwendungsfälle Display 1..* NucleicAcid und Select 1..* NucleicAcid, welche von der Komponente `casa.gui.SeqListView` angeboten werden. Sie ermöglicht die Darstellung einer Liste von Nukleinsäure-Sequenzen, aus denen der Benutzer interaktiv eine bestimmte Sequenz auswählen kann. Hierfür müssen zuvor die betreffenden Sequenzen mittels Aufruf der Operation `setSequences` übergeben werden. Dies führt die Komponente in den von außen beobachtbaren Zustand `<SeqDisplayed>`, wie in der zugehörigen Interaktion beschrieben ist. Somit sind die Voraussetzungen für die Durchführung des Anwendungsfalls Select 1..* NucleicAcid erfüllt, dessen Ereignis-Muster das Ereignis NucleicAcid Selected als Kombination der Manipulation `Select` und des Konzepts `NucleicAcid` spezifiziert. Hierbei tritt die vordefinierte Rolle `Listener` als Argument bei An- bzw. Abmeldung sowie als aufgerufener Partner bei Auslösung des Ereignisses auf. Die möglichen Ausprägungen dieser Rolle durch Komponenten des Prototyps werden zunächst über den in der Signatur spezifizierten Typ `casa.gui.SelectionListener` bestimmt. Jedoch können die im Ereignis-Muster angegebenen Informationen offensichtlich auch zur Generierung einfacher Ereignis-Adapter herangezogen werden (siehe Abschnitt 3.6.4). Das eigentliche Ergebnis des Anwendungsfalls, also die vom Benutzer ausgewählten Nukleinsäure-Sequenzen, kann anschließend

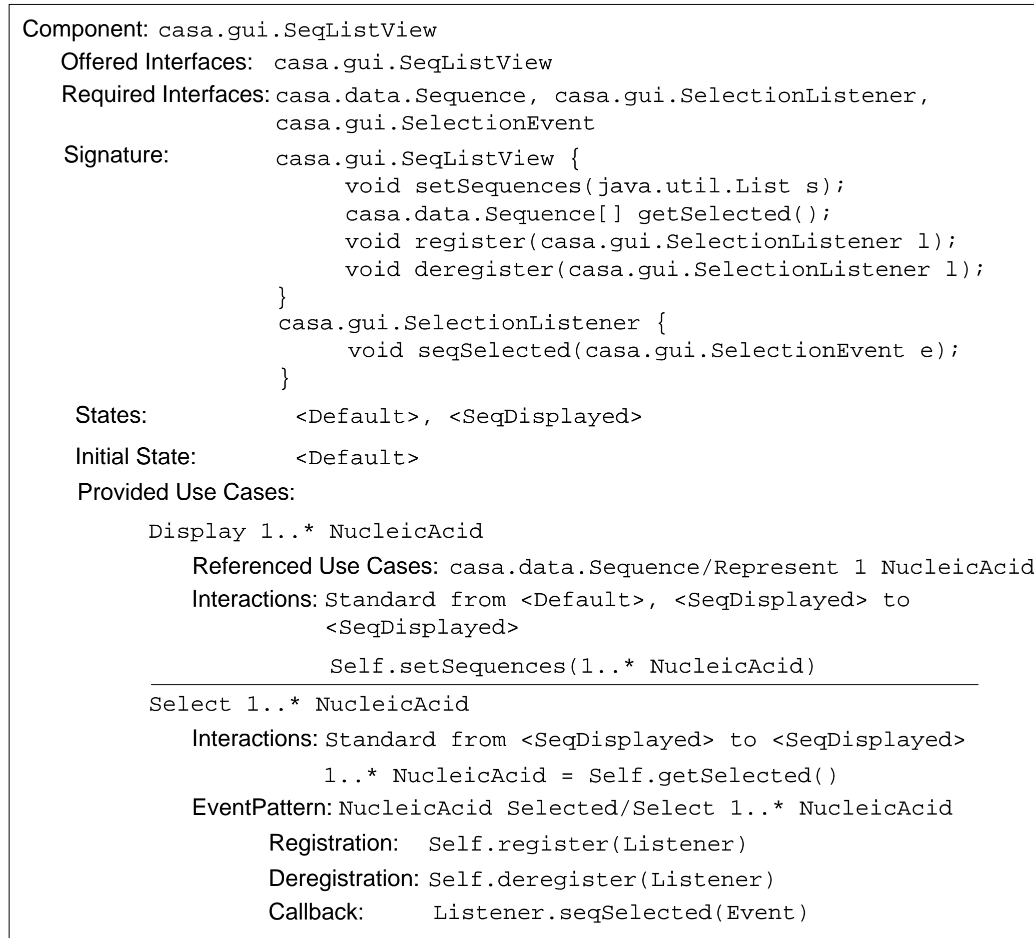


Abb. 3.12: *CUC-Beschreibung zur Komponente* `SeqListView`

durch Aufruf der Operation `getSelected` im Rahmen der zugehörigen Interaktion erhalten werden. Aufgrund des komplexen dynamischen Ablaufs eines Ereignis-Musters bietet es sich offensichtlich an, diese Verhältnisse durch eine geeignete grafische Beschreibungstechnik, beispielsweise als entsprechendes Sequenzdiagramm [Krü00, RJB98], zu spezifizieren, wie später in Abschnitt 5.2.3 ausführlich erläutert wird.

Die in diesem Abschnitt vorgestellte Modellierung eines komponentenbezogenen Anwendungsfalls erlaubt eine prägnante und anwendungsnahe Beschreibung der von einer Komponente angebotenen, grundlegenden Funktionalität. Die technischen Anteile eines CUC ermöglichen hierbei die einfache Abbildung des Modells auf verschiedene komponentenbasierte Infrastruktu-

ren durch eine entsprechende Umsetzung der zugehörigen Interaktionen. Wie im nächsten Abschnitt gezeigt wird, eignet sich die gewählte Modellierung auf logischer Ebene, also die Manipulation von Konzepten des Anwendungsbereichs, auch für die Spezifikation funktionaler Anforderungen.

3.5 Funktionale Spezifikation

Vor der Konstruktion von Prototypen sind offensichtlich die funktionalen Anforderungen an das spätere System zu spezifizieren. Sie legen fest, welche Funktionalität tatsächlich vom System aus Sicht des Anwenders erwartet wird. Weil der in dieser Arbeit vorgestellte Prototyping-Ansatz im wesentlichen zur Ermittlung und Konkretisierung der endgültigen funktionalen Anforderungen dient (vgl. Abschnitte 1.2 und 2.1), kann hierbei keine exakte, streng formalisierte Spezifikation vorausgesetzt werden. Deshalb wird im folgenden eine bewußt einfach gehaltene, anwendungsbezogene und weitgehend deklarative Beschreibung der erwünschten Funktionalität vorgeschlagen, die sich am zuvor eingeführten Modell eines komponentenbezogenen Anwendungsfalls orientiert. Diese grundlegende Beziehung erlaubt in Verbindung mit den zusätzlichen, operativen Elementen einer Funktionalen Spezifikation die effektive Generierung ausführbarer Prototypen, wie später in Abschnitt 3.6.4 erläutert wird.

In Übereinstimmung mit der ursprünglichen Intention eines Anwendungsfalls zur Beschreibung funktionaler Anforderungen, liegt es nahe, die zuvor in Abschnitt 3.4 für einen CUC gewählte Modellierung auch auf die vorliegende Aufgabe zu übertragen. Dementsprechend wird im Rahmen der Spezifikation gewünschte Funktionalität ebenfalls als *Manipulation von Konzepten des Anwendungsbereichs* aufgefaßt. Die sich hieraus ergebende Kombination aus aktiven und passiven Elementen entspricht in ihrer Formulierung der typischen Benennung von Anwendungsfällen in der Anforderungsanalyse der Systementwicklung.

Die geforderte Gesamtfunktionalität des Systems resultiert demzufolge zunächst aus der Menge aller spezifizierten Anwendungsfälle. Allerdings muß darüber hinaus zwischen diesen Anwendungsfällen ein logischer Zusammenhang hergestellt werden, um die effektive Generierung funktionaler Prototypen zu ermöglichen. Bei dem vorgestellten Ansatz geschieht dies durch Zwischenergebnisse der Ausführung eines Anwendungsfalls, die als Parameter für andere Anwendungsfälle übergeben werden können. Weiterhin ist es möglich, Anwendungsfälle zu gruppieren und deren Ausführung in Abhängigkeit von eingetretenen Ereignissen oder Benutzereingaben zu steuern. Diese einfachen operativen Elemente des vorgestellten Modells erlauben bereits die Spezifi-

kation und Generierung zahlreicher nicht-trivialer Prototypen. Sie können zudem durch weitere zweckmäßige Konstrukte auf unmittelbar ersichtliche Weise ergänzt werden, wie in Abschnitt 5.2.2 dargelegt wird.

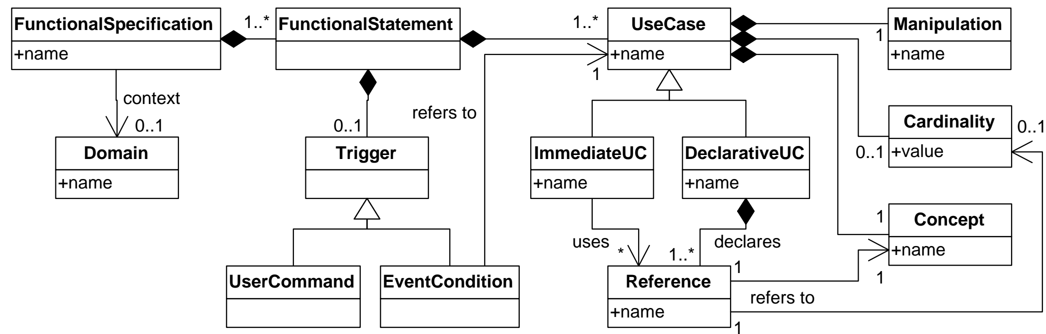


Abb. 3.13: *Modell der Funktionalen Spezifikation*

Abbildung 3.13 zeigt ein UML-Klassendiagramm des oben beschriebenen Modells zur Spezifikation funktionaler Anforderungen. Demnach wird eine derartige Spezifikation (FunctionalSpecification) durch ihren Namen und einen optional zu verwendenden Kontext der referenzierten Konzepte gekennzeichnet. Sie beinhaltet darüber hinaus eine Menge von sogenannten *funktionalen Aussagen* (FunctionalStatement), also eine Folge von Anwendungsfällen (UseCase), die gemeinsam ausgeführt werden sollen. Gemäß der Modellierung eines CUC (vgl. Abschnitt 3.4) wird ein einzelner Anwendungsfall als Manipulation eines in der Ontologie definierten Konzepts aufgefaßt. Darüber hinaus läßt sich optional die Kardinalität des manipulierten Konzepts angegeben. Die bei Ausführung eines Anwendungsfalls erhaltenen Ergebnisse oder benötigten Argumente können durch eine benannte Referenz (Reference) auf das entsprechende Konzept explizit gekennzeichnet werden. Hierbei muß die betreffende Referenz zuerst durch einen *deklarativen Anwendungsfall* (DeclarativeUC) eingeführt werden, bevor sie in einem später aufgeführten Anwendungsfall benutzt werden kann. Diese Festlegung unterscheidet ihn vom *unmittelbaren Anwendungsfall* (ImmediateUC), der selbst keine neuen Referenzen deklariert.

Das in Abbildung 3.14 dargestellte Beispiel einer Funktionalen Spezifikation für die erste Version eines Werkzeugs zur Analyse von DNA-Sequenzen erläutert das oben eingeführte Modell. Der gesuchte Prototyp `SequenceTool1` soll dem Anwender ermöglichen, eine Menge von DNA-Sequenzen zu suchen, etwa innerhalb einer entsprechenden Datenbank, und die erhaltenen Ergebnisse geeignet auf dem Bildschirm anzuzeigen. Hierfür ist zunächst die Angabe der Domäne `SequenceAnalysis` als Kontext erforder-

```
Prototype: SequenceTool1
Context:  SequenceAnalysis
   Search 1..* DNA "result"
   Display "result"
```

Abb. 3.14: *Funktionale Spezifikation für SequenceTool1*

lich, damit das später referenzierte Konzept `DNA` eindeutig zugeordnet werden kann. Nunmehr wird die im Beispiel zuerst geforderte Funktionalität durch den deklarativen Anwendungsfall `Search 1..* DNA` ausgedrückt, wobei die Ergebnisse der Suche mit der Referenz `result` verknüpft sind. Der folgende unmittelbare Anwendungsfall `Display 'result'` spezifiziert die gewünschte Anzeige der Ergebnisse durch Verwendung dieser zuvor deklarierten Referenz. Hierbei ergibt sich die Ausprägung und Kardinalität des manipulierten Konzepts implizit über die Deklaration der Referenz, so daß eine verkürzte, annähernd natürliche Formulierung der funktionalen Anforderung ermöglicht wird.

Bereits dieses vereinfachte Beispiel offenbart die charakteristischen Merkmale des vorgestellten Frameworks für komponentenbasiertes Rapid Prototyping. Er beschränkt sich bewußt auf die *grundlegende* Funktionalität des betrachteten Systems, um einerseits den praktischen Einsatz der Spezifikationstechniken zu erleichtern, und andererseits die Generierung unterschiedlicher Prototyp-Varianten nicht unnötig einzuschränken. Die erwünschte, *spezifische* Ausprägung der Funktionalität, etwa den Namen der durchsuchten Datenbank oder die angebotenen Suchkriterien, wird erst später bei Beurteilung der generierten Prototypen berücksichtigt (vgl. Abschnitt 3.6.5). Darüber hinaus läßt sich zum Zeitpunkt der Ausführung eines Prototypen die erbrachte Funktionalität typischerweise auch durch Konfiguration der beteiligten Komponenten modifizieren. Beispielsweise kann die Unterscheidung zwischen grafischer oder textueller Darstellung der gefundenen DNA-Sequenzen auf diese Weise getroffen werden, falls die verwendete Komponente eine entsprechende Konfiguration anbietet.

Eine weitere, im Modell vorgenommene Vereinfachung stellt die implizite Verknüpfung der für spätere Schritte benötigten Zwischenergebnisse dar. Abbildung 3.15 erläutert dieses Vorgehen an Hand einer Funktionalen Spezifikation für ein zweites Werkzeug `SequenceTool2`, das aus den gefundenen DNA-Sequenzen ein Alignment berechnen und dieses am Bildschirm anzeigen soll. Der für die Berechnung spezifizierte, deklarative Anwendungsfall `Calculate 1 Alignment 'alignment'` benötigt hierfür eine Men-

```
Prototype: SequenceTool2
Context:  SequenceAnalysis
   Search 1..* DNA
   Calculate 1 Alignment "alignment"
   Display "alignment"
```

Abb. 3.15: *Funktionale Spezifikation für SequenceTool2*

ge von DNA-Sequenzen, welche durch den vorangegangenen Anwendungsfall `Search 1..* DNA` bereitgestellt wird. Die Auswahl einer bestimmten Strategie bei Generierung der Prototypen stellt letztlich diese implizite Zuordnung sicher (siehe Abschnitt 3.6.4).

Neben der bisher erwähnten, streng sequentiellen Komposition verschiedener Anwendungsfälle ist es in vielen Fällen hilfreich, die Ausführung zur Laufzeit abhängig von bestimmten Ereignissen oder expliziten Befehlen des Benutzers zu spezifizieren. Zu diesem Zweck führt das in Abbildung 3.13 dargestellte Modell mit dem sogenannten *Auslöser* (Trigger) ein entsprechendes Element ein. Es kann mit einer gegebenen funktionalen Aussage assoziiert werden, um zu beschreiben, daß die so zusammengefaßte Funktionalität nur auf Wunsch des Benutzers (UserCommand) oder bei Eintreten eines bestimmten Ereignisses (EventCondition) erbracht werden soll. Während für den ersten Fall ein einfaches Element der Benutzeroberfläche generiert werden kann, etwa eine entsprechende Schaltfläche, muß für den zweiten Fall die Ursache oder Herkunft des Ereignisses angegeben werden. Deshalb verweist das Element EventCondition auf einen Anwendungsfall, der ein solches Ereignis erwartungsgemäß auslösen kann.

Die in Abbildung 3.16 dargestellte Funktionale Spezifikation für ein drittes Analyse-Werkzeug `SequenceTool3` erläutert beispielhaft die oben beschriebenen Verhältnisse. Sie beinhaltet drei funktionale Aussagen mit unterschiedlichen Auslösern. So erfolgt die Suche nach DNA-Sequenzen und deren Anzeige auf dem Bildschirm nur auf expliziten Wunsch des Benutzers, gekennzeichnet durch die Schlüsselwörter `On Command`. Weiterhin soll bei Auswahl einer bestimmten DNA-Sequenz diese durch den Benutzer bearbeitet werden können. Deshalb verknüpft die Spezifikation den Auslöser für den zugehörigen Anwendungsfall `Edit 1 DNA` mit dem ereignisbasierten Anwendungsfall `Select 1 DNA` über die Referenz `selected`, welche das Zwischenergebnis repräsentiert. Schließlich soll bei Auswahl mehrerer DNA-Sequenzen ein Alignment berechnet und angezeigt werden. Dieses Alignment bildet die Basis für eine Bestimmung der Verwandtschaftsverhältnisse, also

```
Prototype: SequenceTool3
Context:   SequenceAnalysis
   On Command :
          Search 1..* DNA "result"
          Display "result"

   On Select 1 DNA "selected" :
          Edit "selected"

   On Select 1..* DNA :
          Calculate 1 Alignment "alignment"
          Display "alignment"
          Calculate 1 Phylogeny "phylogeny"
          Display "phylogeny"
```

Abb. 3.16: *Funktionale Spezifikation für SequenceTool3*

des zugehörigen Phylogenetischen Baums (siehe Abschnitt 2.3.2), der ebenfalls auf dem Bildschirm dargestellt werden soll. Dementsprechend beinhaltet die Spezifikation einen ereignisbasierten Auslöser `Select 1..* DNA`, welcher die Ausführung der Anwendungsfälle `Calculate 1 Alignment`, `Display 1 Alignment`, `Calculate 1 Phylogeny` und `Display 1 Phylogeny` kontrolliert. Hierbei erfolgt die Zuordnung benötigter Zwischenergebnisse wiederum über entsprechend benannte Referenzen.

Wie das vorangegangene Beispiel zeigt, können mit dem vorgestellten Modell durchaus komplexe funktionale Anforderungen auf verhältnismäßig einfache und anwendungsbezogene Weise spezifiziert werden. Hierbei liegt der Schwerpunkt auf einer möglichst deklarativen Spezifikation der geforderten, grundlegenden Funktionalität, während der operative Anteil, also die Steuerung des Ablaufs zur Laufzeit, offensichtlich durch Integration zusätzlicher Elemente erweitert werden kann, wie in Abschnitt 5.2.2 dargelegt wird. Letztlich ermöglicht gerade die Übersichtlichkeit der gewählten Modellierung eine effektive und effiziente Generierung ausführbarer, funktionaler Prototypen. Dieses entscheidende Verfahren des vorgestellten Frameworks wird im folgenden Abschnitt ausführlich an Hand eines durchgängigen Anwendungsbeispiels erläutert.

3.6 Generierung funktionaler Prototypen

Die in den vorangegangenen Abschnitten aufgeführten Modelle zur Beschreibung des Anwendungsbereichs sowie der erbrachten und geforderten Funktionalität bilden die Grundlage für eine weitgehend automatisierte Generierung ausführbarer Prototypen. Hierfür werden zunächst die deklarativen, anwendungsbezogenen Elemente der Funktionalen Spezifikation mit den Anwendungsfällen der vorhandenen Komponenten über ihren gemeinsamen Bezug zur Ontologie abgeglichen. Anschließend kann mit Hilfe der operativen Elemente der Spezifikation in Verbindung mit den technischen Anteilen der ausgewählten CUCs das Grundgerüst des avisierten Prototypen erstellt werden. Hierbei erfordert die Kombination technisch inkompatibler Komponenten ggf. die Integration entsprechender Adapter. Diese sind im allgemeinen vom Benutzer bereitzustellen, können unter bestimmten Voraussetzungen aber auch zumindest in Teilen automatisch generiert werden. Die Auswahl verschiedener, prinzipiell geeigneter Komponenten zur Erfüllung der gleichen funktionalen Anforderung führt zu zahlreichen unterschiedlichen Prototyp-Varianten, die nach technischen und anwendungsbezogenen Merkmalen hinsichtlich ihrer Eignung bewertet werden. Diese Beurteilung beeinflußt maßgeblich den nächsten Zyklus eines insgesamt iterativ organisierten Prozesses.

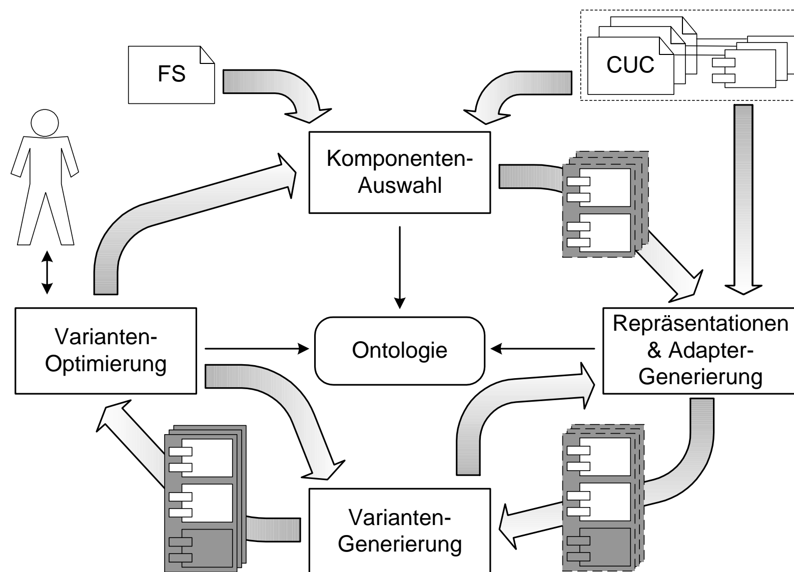


Abb. 3.17: *Schematischer Ablauf der Prototyp-Generierung*

Abbildung 3.17 verdeutlicht den so skizzierten Ablauf an Hand einer schematischen Darstellung der Zusammenhänge. Sie zeigt die grundlegenden Auf-

gaben und Teilschritte der Prototyp-Generierung als beschriftete Rechtecke, den Fluß von Informationen oder Zwischenergebnissen als breite, umrandete Pfeile sowie die erforderliche, dynamische Interaktion mit dem Benutzer oder aktiven Elementen des Frameworks als einfache, durchgezogene Pfeile. Die gegebene Funktionale Spezifikation sowie die vorhandenen Softwarekomponenten werden durch eine an die UML angelehnte Notation repräsentiert. Hierbei kennzeichnet eine dunkel hervorgehobene Darstellung die im Verlauf des Verfahrens generierten Anteile der Prototypen, also beispielsweise die notwendige Verknüpfung eingesetzter Komponenten oder neu erstellte Adapter.

Der Ausgangspunkt des vorgestellten Ansatzes ist der Abgleich zwischen erwünschter und angebotener Funktionalität, also die Auswahl geeigneter komponentenbezogener Anwendungsfälle hinsichtlich der gegebenen Funktionalen Spezifikation. In Abbildung 3.17 sind die entsprechenden Beschreibungen durch ein mit CUC bzw. FS beschriftetes Symbol repräsentiert, wobei die enge Beziehung zwischen CUCs und vorhandenen Softwarekomponenten durch ein sie umgebendes, unterbrochenes Rechteck angedeutet ist. Da in beiden Beschreibungen auf Manipulationen und Konzepte der Ontologie Bezug genommen wird (vgl. Abschnitte 3.4 und 3.5), kann die erforderliche Auswahl nach anwendungsbezogenen Kriterien und Regeln erfolgen, wie später in Abschnitt 3.6.2 ausführlich erläutert wird. Zu diesem Zweck wird auf Informationen der gemeinsamen Ontologie zurückgegriffen, wobei der in Abschnitt 3.3 eingeführte Begriff der semantischen Kompatibilität eine grundsätzlich tolerante Auswahl ermöglicht. Deshalb ist die in der Abbildung durch ein abgerundetes Rechteck repräsentierte Ontologie nicht als einfaches, passives Datenmodell, sondern vielmehr als aktiver, dynamischer Bestandteil des Frameworks aufzufassen (siehe Abschnitt 4.2).

Die hierdurch realisierte, flexible Zuordnung zwischen geforderter Funktionalität und angebotenen Anwendungsfällen führt in der Regel zu mehreren, unterschiedlich zusammengesetzten Kombinationen aus Komponenten für die Umsetzung einer gegebenen Funktionalen Spezifikation (vgl. Abbildung 3.3). Diese Konfigurationen beinhalten eine unmittelbare Beziehung zwischen funktionalen Aussagen der Spezifikation und ausgewählten Softwarekomponenten. Sie repräsentieren somit gewissermaßen die möglichen „Konstruktionspläne“ für die später zu generierenden, ausführbaren Prototyp-Varianten und sind daher in Abbildung 3.17 durch eine unterbrochene Umrißlinie gekennzeichnet.

Die zusammengestellten Konfigurationen dienen im nächsten grundlegenden Schritt des Ablaufs zur Festlegung der erforderlichen *Repräsentationen* für manipulierte Konzepte des Anwendungsbereichs sowie ggf. benötigter Zwischenergebnisse. Diese Zuordnung wird durch die technischen Elemente

der zugehörigen CUC-Beschreibungen ermöglicht. Sie beinhalten eine Abbildung von Konzepten auf Parameter bzw. Rückgabewerte von Operationen der beteiligten Schnittstellen (vgl. Abschnitt 3.4 sowie Abbildung 3.9). Die eingesetzte technische Infrastruktur wiederum bestimmt die zugehörigen Typen der verwendeten Programmiersprache sowie deren Kompatibilität. Somit kann beispielsweise festgestellt werden, daß der Java-Typ `String` eine mögliche Repräsentation des Konzepts Name einer DNA-Sequenz darstellt, während die Sequenz selbst, also das Konzept DNA, durch eine Klasse mit der Schnittstelle `casa.data.DNASequence` implementiert wird (vgl. Abbildung 3.10).

Aufgrund der in der Praxis zu erwartenden, technischen Probleme bei Verwendung von Komponenten unterschiedlicher Herkunft (siehe Abschnitt 3.1.3), ist jedoch für die geeignete Konvertierung eigentlich *logisch kompatibler* Repräsentationen des gleichen Konzepts Sorge zu tragen. So kann etwa eine zweite Analyse-Komponente mit Zwischenergebnissen des Typs `clustal.analysis.Sequence` arbeiten, falls ein entsprechender *Adapter* für die oben genannte Repräsentation des Konzepts DNA zur Verfügung steht. Eine weitere zentrale Aufgabe ist somit die Ermittlung und Integration geeigneter Repräsentationen und Adapter. Unter bestimmten Voraussetzungen ist es möglich, einen derartigen Adapter mit Hilfe der Ontologie automatisch zu generieren, wie später in Abschnitt 3.6.3 gezeigt wird.

Nach Ermittlung der benötigten Repräsentationen und Adapter wird im nächsten Schritt der Iteration die eigentliche Generierung der ausführbaren Prototyp-Varianten durchgeführt. Hierfür muß zunächst die tatsächlich benutzte Zusammensetzung aus Komponenten, Repräsentationen und benötigter Adapter endgültig festgelegt und verknüpft werden. Neben der Zusammensetzung einer Variante bestimmt deren vollständiger Konstruktionsplan auch über die Auswahl und Umsetzung der zu einem CUC gehörigen Interaktion. Dies betrifft im wesentlichen die Belegung von Parametern sowie die Zuordnung zwischen Rollen und Instanzen der beteiligten Komponenten, wie in Abschnitt 3.6.4 erläutert wird. Aufgrund der exponentiell steigenden Anzahl an unterschiedlich zusammengesetzten Konfigurationen, wird nur ein Teil der möglichen Varianten auch tatsächlich realisiert. Die hierfür erforderliche Auswahl geschieht auf Basis einer geeigneten Heuristik, die im Zusammenspiel mit dem nächsten Schritt des Verfahrens zur Varianten-Optimierung eingesetzt wird (siehe Abschnitt 3.6.5).

Die anschließende Umsetzung des detaillierten Konstruktionsplans einer gegebenen Variante in fehlerfrei übersetzbaren Quellcode erfordert zahlreiche, detaillierte Einzelschritte, die offensichtlich maßgeblich durch die verwendete technische Infrastruktur und deren Besonderheiten bestimmt werden. Aus diesem Grund wird in den folgenden Abschnitten dieses Kapitels auf eine

vollständige und umfassende Erläuterung verzichtet. Allerdings beinhaltet Abschnitt 3.6.4 einige exemplarische Ausschnitte des generierten Codes, welche das Prinzip einer derartigen Umsetzung am Beispiel der Java-Plattform verdeutlichen.

Nach vollständiger Generierung und Übersetzung der entwickelten Prototyp-Varianten können diese nunmehr ausgeführt und hinsichtlich ausgewählter Merkmale bewertet und optimiert werden. Hierbei lassen sich bestimmte Merkmale durchaus automatisch beurteilen, etwa die Anzahl der insgesamt erforderlichen Komponenten und Adapter. Diese Informationen werden unmittelbar und ohne Zutun des Benutzers für die Generierung neuer, möglicherweise besser geeigneter Varianten herangezogen, wie durch einen entsprechenden Pfeil in Abbildung 3.17 verdeutlicht wird. Nach einer solchen, automatisch durchgeführten Optimierung werden dem Benutzer eine überschaubare Anzahl an Varianten zur Beurteilung vorgelegt. Nur durch diese manuelle Bewertung kann letztlich die Tauglichkeit der generierten Prototyp-Varianten hinsichtlich der gestellten Anforderungen zuverlässig ermittelt werden.

Der hierfür vorgeschlagene Ansatz erlaubt dem Benutzer eine quantitative Bewertung der gesamten Variante sowie einzelner, ausgewählter Komponenten. Beispielsweise erfüllt der Prototyp eines Werkzeuges zur Sequenzanalyse die gestellten Erwartungen im allgemeinen nicht, obwohl seine Komponente zur Berechnung eines Alignments den diesbezüglichen funktionalen Anforderungen sehr gut gerecht wird. Diese spezifischen Informationen können in die Generierung neuer Prototypen einbezogen werden, etwa durch Bevorzugung der als gut bewerteten Komponente beim nächsten Abgleich mit der entsprechenden funktionalen Aussage oder durch Modifikation der ursprünglichen Funktionalen Spezifikation.

Neben diesem direkten Übergang in eine weitere Iteration des gesamten Prozesses besteht allerdings auch die Möglichkeit, eine lokale Optimierung der erzeugten Varianten durchzuführen. Hierbei werden die im Verlauf der Bewertung gewonnenen Informationen zur Bildung neuer Kombinationen aus den bestehenden Prototyp-Varianten herangezogen. Auf diese Weise können die individuell als gut bewerteten Teile unterschiedlicher Varianten in eine gemeinsame, voraussichtlich besser geeignete Variante überführt werden. Aufgrund der zu erwartenden hohen Anzahl möglicher Varianten, schlägt diese Arbeit eine teilweise automatisierte, heuristische Optimierung durch einen *Genetischen Algorithmus* vor. Er betrachtet Prototyp-Varianten als Individuen einer gesamten Population, die sich durch Mutation, Rekombination und Selektion in einem evolutionären, zufallsgesteuerten Prozeß weiterentwickelt (vgl. Abbildung 3.4). Die hierfür erforderlichen *Genetischen Operatoren* sowie die für eine Selektion ausschlaggebende Fitneß-Funktion werden später in

Abschnitt 3.6.5 im Detail vorgestellt. Weil die Erzeugung neuer Varianten im Verlauf der Optimierung möglicherweise die Integration zusätzlicher Adapter erfordert, ist in Abbildung 3.17 neben der unmittelbaren Rückkopplung zwischen Bewertung und Generierung ein weiterer, entsprechend rückwärts gerichteter Informationsfluß dargestellt.

Nach diesem Überblick über den Ablauf und die grundsätzlichen Aufgaben der Prototyp-Generierung, werden in den nächsten Abschnitten die erarbeiteten Ergebnisse ausführlich vorgestellt. Hierbei kann der in Abbildung 3.17 dargestellte Prozeß zur Orientierung dienen. Aus Gründen der Anschaulichkeit wird der gewählte Ansatz für komponentenbasiertes Rapid Prototyping an Hand eines durchgängigen, vereinfachten Anwendungsbeispiels erläutert. Es beschreibt die Entwicklung eines Prototypen zur Bearbeitung und Analyse biomolekularer Sequenzen, basierend auf der in Abbildung 3.16 gezeigten Funktionalen Spezifikation. Ein Teil der hierfür vorausgesetzten Beschreibungen, Komponenten und Modelle, etwa die zugrundeliegende Ontologie des Anwendungsbereichs, wird zunächst in einem gesonderten Abschnitt eingeführt. In den darauf folgenden Abschnitten werden diese Elemente durch die jeweils benötigten Anteile schrittweise ergänzt. Das vollständige Beispiel sowie die erforderlichen Daten, Werkzeuge und Komponenten finden sich unter [Vil01b]. Die eingesetzten funktionalen Komponenten basieren ursprünglich auf Semester- und Diplomarbeiten im Rahmen des Projektes *CASA - A Component-oriented Architecture for Biomolecular Sequence Analysis* [Vil01a], wurden jedoch anschließend für die Zwecke der vorliegenden Arbeit umfassend überarbeitet und ergänzt.

3.6.1 Anwendungsbeispiel

Das in diesem Abschnitt vorgestellte, vereinfachte Anwendungsbeispiel beinhaltet die Entwicklung eines Werkzeugs zur Bearbeitung und Analyse von DNA-Sequenzen. Die initiale, informelle Beschreibung der gewünschten Funktionalität kann wie folgt zusammengefaßt werden:

- Das System soll es erlauben, nach vorhandenen DNA-Sequenzen innerhalb einer Datenbank zu suchen und das Ergebnis am Bildschirm darzustellen.

- Der Benutzer kann einzelne, ausgewählte Sequenzen bearbeiten, beispielsweise die Abfolge der enthaltenen Nucleotide verändern oder neue Teilsequenzen einfügen.

- Entsprechend veränderte Sequenzen sollen wieder in einer Datenbank gespeichert werden.

- Nach Auswahl mehrerer Sequenzen soll deren Alignment berechnet und am Bildschirm dargestellt werden. Dieses Alignment dient als Grundlage zur Ermittlung der Verwandtschaftsbeziehungen zwischen den beteiligten Sequenzen bzw. den zugehörigen Spezies. Der ermittelte Phylogenetische Baum soll wiederum am Bildschirm dargestellt werden.

Die so beschriebenen funktionalen Anforderungen an das zu entwickelnde System beinhalten durchaus typische Verarbeitungsschritte und beziehen sich auf wohlbekannte Begriffe des Anwendungsbereichs. Um sie genauer in Form einer Funktionalen Spezifikation zu beschreiben, ist jedoch zunächst die Modellierung einer entsprechenden Ontologie erforderlich (siehe Abschnitt 3.3). Im Rahmen des Anwendungsbeispiels sei hierfür der in Abbildung 3.18 dargestellte Ausschnitt der zugrundeliegenden Ontologie gegeben.

Sie beschreibt in einer an Klassendiagramme der UML angelehnten, grafischen Notation die für das Beispiel wesentlichen Konzepte und Relationen innerhalb der Domänen Biochemistry, AbstractStructure und SequenceAnalysis. In der Abbildung repräsentiert ein beschrifteter Kasten die jeweils beinhalteten Konzepte. Ein dunkel hinterlegter Kasten kennzeichnet diejenigen Konzepte, welche ursprünglich nicht in der sie umfassenden Domäne definiert wurden, etwa allgemein gebräuchliche Konzepte wie Name oder Length, die aus einer entsprechend vorausgesetzten Domäne importiert werden können. Hierbei ist zu beachten, daß nach den zugrundeliegenden Regeln der Ontologie importierte Konzepte möglicherweise durch neue Relationen verändert und somit Teil der neuen Domäne werden (siehe Abschnitt 3.3). Durch Angabe des zugehörigen Kontexts kann beispielsweise zwischen dem Konzept NucleicAcid in den Domänen Biochemistry und SequenceAnalysis unterschieden werden. Weiterhin werden alle Spezialisierungen des veränderten Konzepts ebenfalls Teil der importierenden Domäne. Dies betrifft im Beispiel die Konzepte DNA und RNA, welche nunmehr im Kontext der Domäne SequenceAnalysis mit einem Namen assoziiert werden. Aus Gründen der Übersichtlichkeit sind diese übertragenen Konzepte und Relationen nicht in Abbildung 3.18 dargestellt.

Darüber hinaus sind die Relationen zwischen Konzepten durch unterschiedliche Linien und Symbole entsprechend der dunkel hervorgehobenen Legende repräsentiert (vgl. Abbildung 3.6). Ergänzend ist hinzuzufügen, daß die Pfeilspitze der Generalisierungsbeziehung auf das allgemeinere Konzept hinweist. Beispielsweise kann Abbildung 3.18 entnommen werden, daß innerhalb der Domäne Biochemistry das Konzept NucleicAcid eine Generalisierung

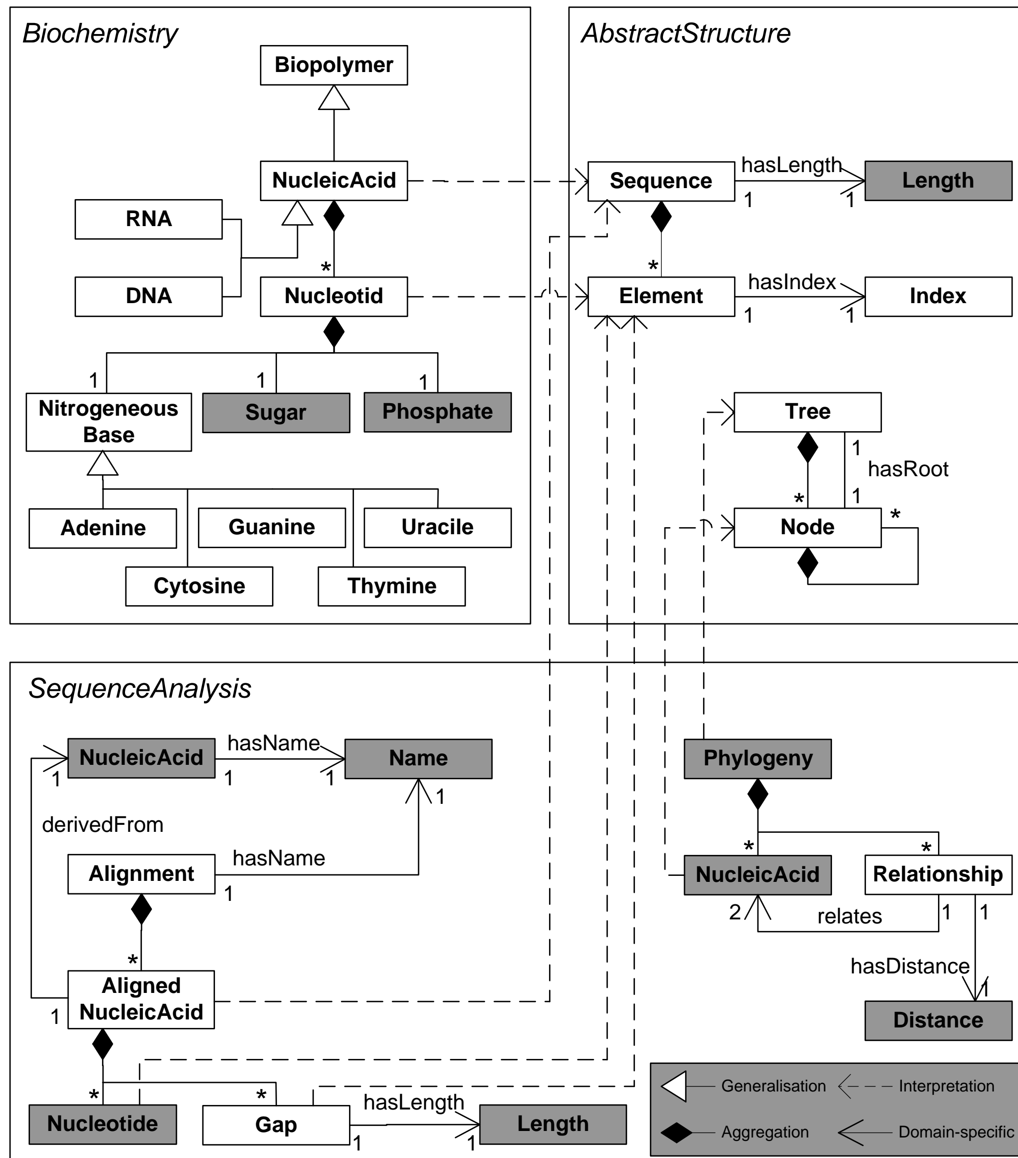


Abb. 3.18: *Ausschnitt der gemeinsamen Ontologie*

der Konzepte DNA und RNA umfaßt. In vergleichbarer Weise gibt die Pfeilspitze einer domänenspezifischen Relation deren optional spezifizierte Richtung an. So wird innerhalb der Domäne SequenceAnalysis festgelegt, daß die Konzepte NucleicAcid und Alignment über die asymmetrische Relation hasName mit dem Konzept Name in Beziehung stehen, d.h. ausgehend von Name besteht *keine* Beziehung zu diesen Konzepten. Deshalb wird das importier-

te Konzept Name im Kontext der Domäne SequenceAnalysis nicht verändert und ist somit auch nicht als deren Bestandteil zu verstehen.

Allerdings kann die in Abbildung 3.18 verwendete grafische Notation nicht sämtliche vorhandenen Zusammenhänge der Ontologie beschreiben. Daher ist bei Darstellung der aufgeführten Interpretationsbeziehungen deren inhärente Struktur zu ergänzen (siehe Abschnitt 3.3). So läßt sich innerhalb der Domäne Biochemistry das Konzept NucleicAcid nur dann als Konzept Sequence der Domäne AbstractStructure auffassen, falls das Konzept Nucleotide als Element interpretiert wird. Eine ähnliche Struktur besitzt die Interpretation des Konzepts AlignedNucleicAcid innerhalb der Domäne SequenceAnalysis als Sequence der Domäne AbstractStructure. In diesem Zusammenhang sind die Konzepte Nucleotide und Gap als Element zu interpretieren. Schließlich kann das Konzept Phylogeny innerhalb der Domäne SequenceAnalysis als Tree der Domäne AbstractStructure aufgefaßt werden, wobei NucleicAcid als Node angesehen wird. Die im Rahmen der Referenz-Implementierung tatsächlich eingesetzte, textuelle Notation zur Beschreibung einer Ontologie unterliegt offensichtlich den oben genannten Beschränkungen nicht (siehe Kapitel 4 und Anhang A.1).

```
Prototype: SequenceTool
Context:   SequenceAnalysis
   On Command :
           Search 1..* DNA "result"
           Display "result"

   On Select 1 DNA "selected" :
           Edit "selected"

   On Change "selected" :
           Store "selected"

   On Select 1..* DNA :
           Calculate 1 Alignment "alignment"
           Display "alignment"
           Calculate 1 Phylogeny "phylogeny"
           Display "phylogeny"
```

Abb. 3.19: *Funktionale Spezifikation des Anwendungsbeispiels*

Durch die in Abbildung 3.18 festgelegten Konzepte und ihre Relationen ist es nunmehr möglich, die initiale, informell gehaltene Beschreibung des

Anwendungsbeispiels in eine für die weiteren Schritte geeignete Funktionale Spezifikation zu überführen. Diese Umsetzung basiert auf der in Abbildung 3.16 gezeigten Spezifikation und erweitert diese entsprechend der gewünschten Funktionalität um den Anwendungsfall `Store 'selected'`, wie in Abbildung 3.19 dargestellt ist. Hierbei gibt der neu eingeführte, ereignisbasierte Auslöser `On Change` an, daß die folgende funktionale Aussage nur in Abhängigkeit einer Zustandsänderung des referenzierten Konzepts ausgeführt wird. Eine derartige Zustandsänderung kann explizit durch die gewählte Repräsentation über ein entsprechendes Ereignismuster bekannt gegeben oder im Verlauf der Generierung automatisch berücksichtigt werden, wie später in Abschnitt 3.6.3 erläutert wird. Insgesamt ergibt sich hierdurch eine anschauliche und intuitive Übersetzung der ursprünglich gestellten Anforderungen an das zu entwickelnde System.

Komponente	***Schnittstellen***	***Component Use Cases***
casa.storage.Repository	casa.storage.Repository	Search * DNA Search * RNA Store 1 DNA Store 1 RNA
casa.gui.SeqListView	casa.gui.SequenceView	Display 1..* NucleicAcid Select 1..* NucleicAcid
casa.edit.SeqEditor	casa.gui.SequenceEditor	Edit 1 NucleicAcid
casa.data.SeqImpl	casa.data.DNASequence casa.data.Sequence	Represent 1 DNA
casa.data.AlignmentImpl	casa.analysis.Alignment	Represent 1 Alignment
casa.analysis.Clustal	casa.analysis.AlignmentAnalysis	Calculate 1 Alignment
other.Phylip	other.Phylip	Calculate 1 Alignment Calculate 1 Phylogeny
other.PhylipSequence	other.PhylipSequence	Represent 1 NucleicAcid
other.PhylipAlignment	other.PhylipAlignment	Represent 1 Alignment
other.PhylipPhylogeny	other.PhylipPhylogeny	Represent 1 Phylogeny
other.PhylipView	other.PhylipView	Display 1 Phylogeny Display 1 Alignment
javax.swing.JTree	javax.swing.JTree ...	Display 1 Tree Select 1..* Node
javax.swing.tree.DefaultTreeModel	javax.swing.tree.TreeModel	Represent 1 Tree
javax.swing.tree.DefaultMutableTreeNode	javax.swing.tree.TreeNode	Represent 1 Node

Abb. 3.20: *Komponenten des Anwendungsbeispiels*

Um mit ihrer Hilfe auch tatsächlich ausführbare Prototypen zu generieren, ist allerdings zusätzlich ein Vorrat gegebener Software-Komponenten

mit einer Beschreibung der durch sie angebotenen Funktionalität erforderlich. Im Rahmen des vereinfachten Anwendungsbeispiels sei hierfür die in Abbildung 3.20 tabellarisch aufgelistete Menge an Komponenten mit den durch sie erbrachten Anwendungsfällen gegeben. Hierbei sind aus Gründen der Übersichtlichkeit zunächst nur die übergeordneten, logischen Anteile des jeweiligen CUC angeführt. Die komponentenspezifischen, technischen Anteile werden erst in den späteren Abschnitten bei Bedarf eingeführt.

Durch die Einhaltung der Java-Konvention bei Benennung der aufgeführten Komponenten wird deutlich, daß diese von durchaus unterschiedlichen Autoren erstellt wurden. Trotzdem läßt die in Abschnitt 3.4 eingeführte Modellierung zur Beschreibung ihrer wesentlichen Funktionalität bereits den gemeinsamen Bezug zum Anwendungsbereich erkennen. In den folgenden Abschnitten wird gezeigt, wie das in dieser Arbeit vorgeschlagene Framework die Generierung funktionaler Prototypen für das so beschriebene Anwendungsbeispiel ermöglicht.

3.6.2 Komponenten-Auswahl

Nach Angabe der Funktionalen Spezifikation sowie der vorhandenen Komponenten und ihrer zugehörigen CUC-Beschreibungen, besteht die erste grundlegenden Aufgabe der Prototyp-Generierung in der Auswahl möglicherweise geeigneter Komponenten (vgl. Abbildung 3.17). Zu diesem Zweck werden die im Rahmen der Funktionalen Spezifikation aufgeführten Anwendungsfälle mit den angebotenen komponentenbezogenen Anwendungsfällen abgeglichen. Da beide Elemente des Frameworks auf der logischen, anwendungsbezogenen Ebene gleichartig modelliert sind (siehe Abschnitt 3.4 und 3.5), kann dieser Abgleich verhältnismäßig einfach und automatisiert an Hand der beteiligten Manipulationen, Konzepte und ihrer Kardinalitäten erfolgen.

Durch die Verwendung einer gemeinsamen Ontologie sind hierbei insbesondere die referenzierten Konzepte des Anwendungsbereich eindeutig definiert. Ihre in der Ontologie festgelegten Beziehungen sowie der sich hieraus ergebende Begriff von semantischer Kompatibilität erlauben darüber hinaus eine der Problemstellung angepaßte Flexibilität und Toleranz bei der Zuordnung zwischen gewünschten und erbrachten Anwendungsfällen (vgl. Abschnitt 3.3). Aus diesen Gründen stützt sich der im folgenden vorgestellte Ansatz zur Ermittlung geeigneter Komponenten im wesentlichen auf die in den Anwendungsfällen referenzierten Konzepte.

Das eigentliche Ziel der Komponenten-Auswahl ist die Abbildung jedes in der Spezifikation enthaltenen Anwendungsfalls auf eine Menge von logisch kompatiblen, komponentenbezogenen Anwendungsfällen. Durch Verwendung des in den Definitionen 3.2 bis 3.5 festgelegten Begriffs der semantischen

Kompatibilität zwischen Konzepten kann die logische Kompatibilität zwischen Anwendungsfällen auf einfache Weise wie folgt definiert werden:

Definition 3.6: *Ein Anwendungsfall der Funktionalen Spezifikation ist* logisch kompatibel *mit einem gegebenen Component Use Case, falls*

- *der Name ihrer jeweiligen Manipulation übereinstimmt,*
- *und die Kardinalität des im Anwendungsfall referenzierten Konzepts kleiner oder gleich der im CUC spezifizierten Kardinalität ist,*
- *sowie die jeweils referenzierten Konzepte* semantisch kompatibel *sind bzw. der Grad ihrer semantischen Kompatibilität einen vorher festgelegten Schwellwert nicht unterschreitet. Hierbei ist die Richtung der Substitution beim Abgleich referenzierter Konzepte zu beachten.*

Die erste Bedingung der obigen Definition fordert eine vollständige Übereinstimmung der beteiligten Manipulationen. Falls innerhalb des übergeordneten Frameworks auch Beziehungen zwischen Manipulationen eingeführt werden, wie in Abschnitt 5.1 diskutiert wird, so ist an dieser Stelle offensichtlich ein weiter gefaßter Begriff der logischen Kompatibilität möglich.

Die zweite Bedingung entspricht der plausiblen Annahme, daß eine Komponente, die eine größere Anzahl an semantisch kompatiblen Konzepten manipuliert, jederzeit auch mit einer kleineren Anzahl dieser Konzepte benutzt werden kann. Allerdings ist ggf. bei späterer Generierung aus technischen Gründen eine schematisch durchzuführende Anpassung der beteiligten Kardinalitäten erforderlich, wie in Abschnitt 3.6.4 erläutert wird.

Die dritte Bedingung aus Definition 3.6 erfordert die Festlegung auf eine konkrete Realisierung der Bewertungsfunktionen für semantische Kompatibilität sowie eines passenden Schwellwerts $\epsilon_k \geq 0$, welcher die untere Grenze der tolerierten Kompatibilität angibt. Wie bereits in Abschnitt 3.3 beschrieben, ist jeder Generalisierungs- und Interpretationsbeziehung zwischen zwei benachbarten Konzepten i und j der Ontologie eine konzeptuelle Distanz $d_{ij} \in [0, 1]$ zugeordnet, die ohne explizite Angabe als 0.5 angenommen werden kann. Für zwei gegebene, nicht äquivalente Konzepte X und Y bezeichne $\vec{p}_{XY}$ den kürzesten Pfad zwischen X und Y als Folge von Konzepten mit den zugehörigen Distanzen ihrer Beziehungen[4]. Weiterhin wird die Länge dieser Folge, also die Anzahl der enthaltenen Konzepte, durch $|\vec{p}_{XY}|$ bezeichnet[5],

[4] Ein solcher Pfad kann durch eine Hilfsfunktion $p(\mathsf{X}, \mathsf{Y})$ auf einfache Weise berechnet werden, falls dieser existiert (vgl. Abschnitt 3.3).

[5] Es gilt offensichtlich $|\vec{p}_{XY}| \geq 2$, da X und Y selbst Teil des Pfads sind (vgl. Abschnitt 3.3).

während $||\vec{p}_{XY}||$ die absolute Länge des Pfads als Summe aller konzeptuellen Distanzen zwischen X und Y angibt. Somit können folgende, beispielhafte Definitionen der Bewertungsfunktionen k_e, k_l und k_p für zugesicherte, beschränkte und potentielle semantische Kompatibilität festgelegt werden:

$$k_e(\vec{p}_{XY}) = \left(\frac{1}{e^{\frac{||\vec{p}_{XY}||}{|\vec{p}_{XY}|-1}}} \right)^p \tag{3.2}$$

$$k_l(\vec{p}_{XY}) = 1 - \frac{||\vec{p}_{XY}||}{|\vec{p}_{XY}| - 1} \tag{3.3}$$

$$k_p(\vec{p}_{XY}) = \begin{cases} 0 & \text{falls } |\vec{p}_{XY}| > 2 \\ k_e(\vec{p}_{XY}) & \text{sonst} \end{cases} \tag{3.4}$$

Gleichung 3.2 definiert also die zugesicherte semantische Kompatibilität $k_e(\vec{p}_{XY}) \in [0,1]$ als nichtlineare Funktion der normierten Gesamtdistanz $||\vec{p}_{XY}||/(|\vec{p}_{XY}| - 1)$, wobei der frei wählbare Parameter p den exponentiellen Abfall der zugehörigen Kurve beeinflußt, wie Abbildung 3.21 zeigt. Die so festgelegte Funktion beschreibt eine Kompatibilität, die mit zunehmender Distanz zwischen X und Y zunächst überproportional sinkt, um gegen Ende einem festen Grenzwert entgegenzustreben, der noch deutlich über 0 liegt. Somit werden semantisch ähnliche Konzepte, die nur wenig spezieller als X sind bei einer möglichen Substitution bevorzugt, während bei weit entfernten Konzepten keine ausgeprägte Differenzierung stattfindet (vgl. Definition 3.3).

Demgegenüber beschreibt Gleichung 3.3 einen einfachen linearen Zusammenhang zwischen beschränkter semantischer Kompatibilität $k_e(\vec{p}_{XY}) \in [0,1]$ und der normierten Gesamtdistanz $||\vec{p}_{XY}||/(|\vec{p}_{XY}| - 1)$. Diese Definition erscheint ausreichend, weil die hierfür maßgebliche Interpretationsbeziehung ohnehin nicht transitiv ist, also $|\vec{p}_{XY}| = 2$ angenommen werden kann (vgl. Abschnitt 3.3 sowie Definition 3.4). Die bei Erstellung der Ontologie angegebene Distanz für eine betrachtete Interpretation repräsentiert somit unmittelbar die eingeschätzte semantische Kompatibilität der beteiligten Konzepte.

Die Definition der potentiellen semantischen Kompatibilität $k_p(\vec{p}_{XY}) \in [0,1]$ in Gleichung 3.4 illustriert die Verwendung einer nicht stetigen Bewertungsfunktion, die von der absoluten Anzahl an transitiven Generalisierungsschritten abhängig ist. So erscheint es in vielen Fällen plausibel, die potentielle semantische Kompatibilität als 0 anzunehmen, sofern sich mehr als ein Konzept über entsprechende Generalisierungsbeziehungen zwischen den Konzepten X und Y befindet, also $|\vec{p}_{XY}| > 2$ gilt. Andernfalls kann auf die Definition der zugesicherten semantischen Kompatibilität k_e zurückgegriffen werden. Eine solche Festlegung von k_p verringert die Gefahr, daß allgemeine

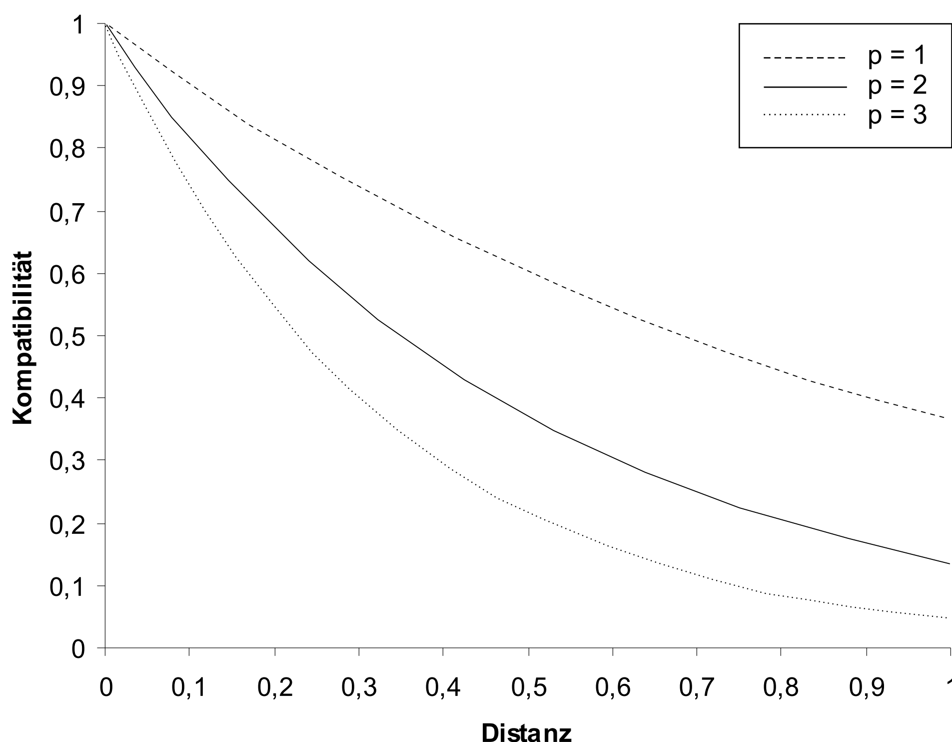


Abb. 3.21: *Zugesicherte semantische Kompatibilität k_e als Funktion der normierten Distanz*

Konzepte in einem sehr speziellen Kontext benutzt werden, der tatsächlich viele spezifische Beziehungen voraussetzt (vgl. Definition 3.5).

Bei der zuletzt erforderlichen Wahl des in Gleichung 3.2 eingeführten Parameters p sowie der unteren Grenze ϵ_k für akzeptable semantische Kompatibilität ist die Vorgabe der semantischen Distanz als fester Wert 0.5 zu berücksichtigen. Daher erscheint es sinnvoll, die Suche nach geeigneten Anwendungsfällen zunächst mit beispielsweise $p = 2$ und $\epsilon_k = 0.2$ zu beginnen, um referenzierte Konzepte mit dieser mittleren Distanz nicht auszuschließen. Anschließend kann je nach Umfang der so gefundenen Ergebnisse eine andere Belegung gewählt werden, die eine gleichermaßen akzeptable wie beherrschbare Anzahl an logisch kompatiblen Anwendungsfällen liefert.

Wie durch die oben aufgeführten Bewertungsfunktionen verdeutlicht wird, ist die Richtung der erforderlichen Substitution bei Untersuchung von Anwendungsfällen auf logische Kompatibilität von entscheidender Bedeutung (vgl. Definition 3.6). Schließlich ist die so realisierte, übergeordnete Funktion $k(X, Y)$ zur Bewertung der semantischen Kompatibilität zweier Konzepte X und Y nicht symmetrisch bezüglich ihrer Argumente, d.h. im allgemeinen

gilt $k(X,Y) \neq k(Y,X)$. Dieser Sachverhalt resultiert letztlich aus den Übereinstimmungen der Menge aller Beziehungen, die innerhalb der Ontologie für die betreffenden Konzepte definiert sind, wie in Abschnitt 3.3 ausführlich erläutert wird. Im Hinblick auf die hierfür ausschlaggebenden Generalisierungsbeziehungen ergibt sich somit die Frage nach der *Gültigkeit von Aussagen* über spezifische Konzepte in einem allgemeineren Kontext oder umgekehrt.

In Verbindung mit komponentenbezogenen Anwendungsfällen sind solche Aussagen als *Manipulation von Konzepten* zu verstehen, d.h. es stellt sich die Frage, ob eine gegebene Komponente auch mit spezielleren oder allgemeineren Konzepten als den ursprünglich Spezifizierten umgehen kann. Daher ist nach den Definitionen 3.2 bis 3.5 das im CUC spezifizierte Konzept in der Regel als *erstes* Argument der durch Gleichung 3.1 definierten Bewertungsfunktion $k(X,Y)$ anzugeben. Es wird also überprüft, ob das im CUC angegebene Konzept durch das in der Funktionalen Spezifikation referenzierte Konzept ersetzt werden kann bzw. welcher Grad der semantischen Kompatibilität bei dieser Ersetzung zu erwarten ist.

Eine Ausnahme dieser Regel sind Anwendungsfälle, deren eigentliches Ergebnis durch das referenzierte primäre Konzept repräsentiert wird. Hierbei wird das betreffende Konzept bzw. dessen Repräsentation ja erst als Folge der Ausführung des Anwendungsfalls erstellt und in weiteren Schritten gemäß der Funktionalen Spezifikation behandelt. In diesem Fall ist zu prüfen, inwieweit das in der Spezifikation erwartete Konzept durch dieses Ergebnis ersetzt werden kann. Das im CUC referenzierte Konzept muß also als *zweites* Argument der Bewertungsfunktion $k(X,Y)$ übergeben werden.

Die so festgelegte Richtung der Substitution entspricht daher im wesentlichen der Anwendung von Regeln des Typsystems einer objekt-orientierten Programmiersprache bei Zuweisungen von Variablen oder Konstanten unterschiedlichen Typs. Allerdings geht der hier vorgestellte Ansatz, wie in Abschnitt 3.3 beschrieben, über den dort festgelegten, eng gefaßten Begriff der technischen Kompatibilität hinaus, da auch beschränkt oder potentiell semantisch kompatible Konzepte bei der Substitution zugelassen sind.

Abbildung 3.22 erläutert die Umsetzung der auf diese Weise ermittelten logischen Kompatibilität von Anwendungsfällen für das betrachtete Beispiel. Auf Basis der in Abbildung 3.18 dargestellten Ontologie werden die meisten Anwendungsfälle der Spezifikation unmittelbar auf entsprechende CUCs der aufgeführten Komponenten abgebildet, weil diese semantisch äquivalente Konzepte behandeln. Somit erreicht der Grad ihrer semantischen Kompatibilität den Wert 1 (vgl. Gleichung 3.1) und liegt daher offensichtlich über dem zuvor festgelegten Schwellwert von $\epsilon_k = 0.2$.

Daneben können die Komponenten `SeqListView` und `SeqEditor` für

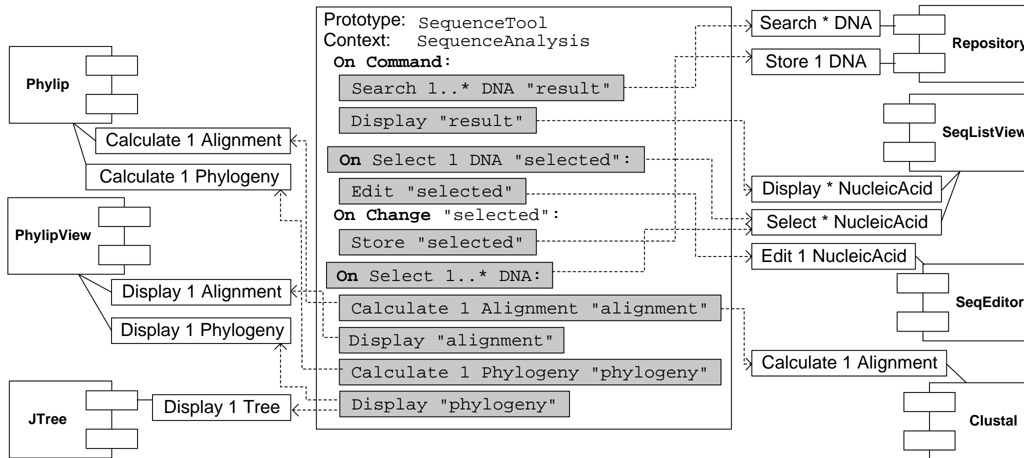


Abb. 3.22: *Komponenten-Auswahl des Anwendungsbeispiels*

die Darstellung, Auswahl und Bearbeitung des Konzepts DNA herangezogen werden, weil das von ihnen manipulierte Konzept NucleicAcid mit DNA zugesichert semantisch kompatibel ist. Auf Basis der vorausgesetzten Distanz von 0.5 zwischen den Konzepten NucleicAcid und DNA ergibt sich somit im Beispiel eine semantische Kompatibilität $k(\mathsf{NucleicAcid}, \mathsf{DNA}) = k_e(p(\mathsf{NucleicAcid}, \mathsf{DNA})) \approx 0.37$, die wiederum größer als ϵ_k ist.

Demgegenüber dient die Komponente `JTree` eigentlich der Darstellung des Konzepts Tree einer unterschiedlichen Domäne AbstractStructure. Trotzdem kann sie zur Erfüllung des Anwendungsfalls `Display 'phylogeny'` ausgewählt werden, da in der Ontologie eine entsprechende Interpretation des Konzepts Phylogeny existiert. Somit sind die Konzepte Phylogeny und Tree eingeschränkt semantisch kompatibel, wobei der Grad ihrer semantischen Kompatibilität $k(\mathsf{Phylogeny}, \mathsf{Tree}) = k_l(p(\mathsf{Phylogeny}, \mathsf{Tree})) = 0.5$ deutlich über ϵ_k liegt[6].

Zur Illustration der potentiellen semantischen Kompatibilität sei im Rahmen des Beispiels angenommen, daß die Funktionale Spezifikation die Berechnung eines lokalen Sequenz-Alignments LocalAlignment erfordert, das als direkte Spezialisierung des Konzepts Alignment in der Ontologie definiert ist. In diesem Fall wäre LocalAlignment mit dem Konzept Alignment des ausgewählten CUC Calculate 1 Alignment potentiell semantisch kompatibel, wobei der Grad ihrer Kompatibilität $k(\mathsf{LocalAlignment}, \mathsf{Alignment}) =$

[6] Die Richtung der Substitution ist bei Prüfung der eingeschränkten semantischen Kompatibilität eindeutig durch die Richtung der zugrundeliegenden Interpretationsbeziehung gegeben.

$k_p(p(\mathsf{LocalAlignment}, \mathsf{Alignment})) \approx 0.37$ beträgt[7]. Somit wäre auch bei dieser hypothetischen Annahme die gleiche Komponente `Clustal` zur Erfüllung des Anwendungsfalls `Calculate 1 LocalAlignment` ausgewählt worden. Ob diese Auswahl im weiteren Verlauf Schwierigkeiten bereitet, d.h. tatsächlich spezifische Beziehungen des Konzepts LocalAlignment verwendet werden, kann erst später zur Laufzeit des generierten Prototypen festgestellt werden. Trotzdem ist es in vielen Fällen sinnvoll, derartige Komponenten nicht unmittelbar zu Beginn auszuschließen, gerade wenn keine geeignet spezifische Komponente zur Verfügung steht.

Wie das Anwendungsbeispiel zeigt, erlaubt der in Definition 3.6 eingeführte Begriff von logischer Kompatibilität einen flexiblen, anwendungsbezogenen Abgleich zwischen Funktionaler Spezifikation und komponentenbezogenen Anwendungsfällen. Er stützt sich im wesentlichen auf die semantische Kompatibilität der beteiligten Konzepte, wobei eine geeignete Festlegung der zugrundeliegenden Bewertungsfunktionen sowie des akzeptablen Grads der Kompatibilität eine tolerante, den jeweiligen Gegebenheiten anpaßbare Komponenten-Auswahl ermöglicht.

Nachdem alle zur Erfüllung der geforderten Funktionalität potentiell verwendbaren Komponenten mit Hilfe der Ontologie ermittelt sind, müssen in einem nächsten Schritt die manipulierten Konzepte selbst auf Bestandteile der entstehenden Prototyp-Varianten abgebildet werden. Dies betrifft insbesondere die im Rahmen der Funktionalen Spezifikation referenzierten Zwischenergebnisse, welche zur Verknüpfung unterschiedlicher Anwendungsfälle dienen (vgl. Abschnitt 3.5). Obwohl die hierfür erforderlichen Begriffe von semantischer Kompatibilität mit den oben beschriebenen Verhältnissen identisch sind, werden die grundlegenden Fragestellungen zur Repräsentation von Konzepten und deren Anpassung an den gegebenen Kontext im folgenden Abschnitt gesondert zusammengefaßt.

3.6.3 Repräsentationen und Adapter-Generierung

Im ersten Schritt der Prototyp-Generierung werden die potentiell geeigneten Komponenten an Hand der durch sie manipulierten Konzepte des Anwendungsbereichs ausgewählt. Diese Konzepte umfassen die in der Funktionalen Spezifikation explizit aufgeführten Zwischenergebnisse, aber auch Parameter und Rückgabewerte der zugehörigen Interaktionen (vgl. Abschnitt 3.4). Es ist daher von entscheidender Bedeutung, wie Konzepte auf technischer

[7] Die Richtung der Substitution ergibt sich aus der Tatsache, daß `Alignment` erst als Ergebnis der zugehörigen Interaktion vorliegt, wie durch eine Untersuchung der technischen Anteile des CUC leicht festgestellt werden kann.

Ebene des Frameworks abgebildet werden und auf welche Weise zwischen Komponenten unterschiedlicher Hersteller vermittelt werden kann.

Zu diesem Zweck führt der vorgestellte Ansatz den Begriff der *Repräsentation* von Konzepten ein. Er bezeichnet die Abbildung von in der Ontologie definierten Konzepten auf einfache Basistypen der verwendeten technischen Plattform oder eigenständige, durchaus komplexe Komponenten mit ausgezeichneten Anwendungsfällen. Beispielsweise kann das Konzept Name durch den grundlegenden Typ `java.lang.String` der Java-Plattform repräsentiert werden, während das Konzept DNA durch eine spezielle Klasse des Typs `casa.data.SeqImpl` implementiert wird. Im letztgenannten Fall wird die entsprechende Klasse als eigene Komponente des Systems aufgefaßt, welche den besonderen CUC Represent 1 DNA anbietet. Wie später erläutert wird, können hierbei aufgrund der festgelegten Definition eines Konzepts als Element der Ontologie weiterführende Annahmen über eine derartige Repräsentation getroffen werden.

Die oben beschriebene Modellierung erleichtert den einheitlichen Umgang mit allen Komponenten im Verlauf der Generierung und ermöglicht die Anwendung der gleichen Regeln bei Ermittlung geeigneter Repräsentationen. Somit kann eine Komponente ein gegebenes Konzept durch einen entsprechenden Anwendungsfall repräsentieren, falls

- die Domäne der referenzierten Konzepte übereinstimmt,
- die betreffende Manipulation einer Repräsentation entspricht, also den ausgezeichneten Namen `Represent` besitzt,
- und die beteiligten Konzepte semantisch kompatibel sind bzw. der Grad ihrer Kompatibilität einen vorher festgelegten Schwellwert nicht unterschreitet (vgl. Abschnitt 3.6.2).

Hierbei kommt der Kardinalität der beteiligten Konzepte keine besondere Bedeutung zu, da jede gängige technische Infrastruktur über entsprechende Mechanismen zum Ausgleich unterschiedlicher Kardinalitäten verfügt. So kann etwa bei späterer Generierung von Java-Code eine geeignete Implementierung der Schnittstelle `java.util.Set` herangezogen werden, um eine Menge des Konzepts DNA durch zahlreiche Instanzen der Komponente `casa.data.SeqImpl` zu repräsentieren.

Die Anwendung der oben aufgeführten Regeln führt im Rahmen des Anwendungsbeispiels zu der in Abbildung 3.23 gezeigten Zuordnung zwischen Konzepten der Funktionalen Spezifikation und verfügbaren Komponenten bzw. den von ihnen angebotenen Anwendungsfällen. Diese zur besseren Unterscheidung auch *primäre Konzepte* genannten Elemente der Spezifikation

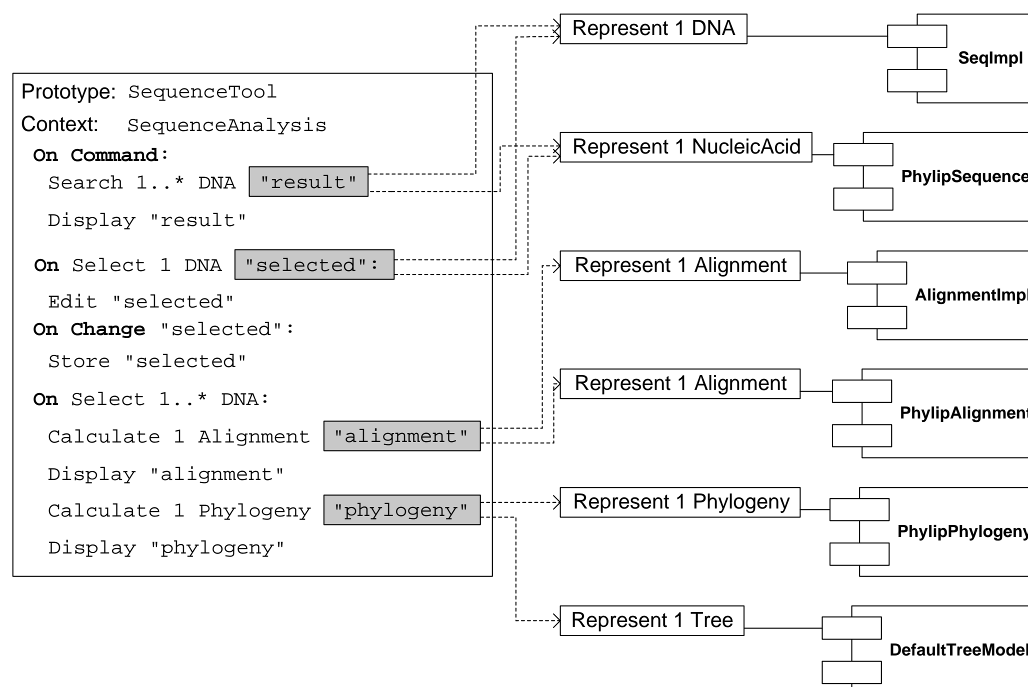


Abb. 3.23: *Repräsentationen des Anwendungsbeispiels*

sind in der Abbildung dunkel hinterlegt und über unterbrochene Pfeile mit ihren möglichen Repräsentationen verbunden. Wiederum kann entsprechend der in Abbildung 3.18 eingeführten Ontologie das Konzept DNA auf eine Repräsentation des übergeordneten Konzepts NucleicAcid abgebildet werden, während das Konzept Phylogeny auch als Tree aufgefaßt werden kann (vgl. Abschnitt 3.6.2).

Neben den primären Konzepten der gegebenen Spezifikation sind allerdings auch geeignete Repräsentationen für die in den Interaktionen referenzierten, zusätzlichen Konzepte zu ermitteln. Diese auch als *sekundäre Konzepte* bezeichneten Referenzen können im technischen Teil eines CUC als Parameter oder Rückgabewerte von Operationsaufrufen angegeben werden, wie das in Abbildung 3.9 vorgestellte Modell verdeutlicht. In diesem Fall ist jedoch die notwendige Repräsentation in den meisten Fällen bereits durch den jeweils definierten technischen Typ festgelegt. Somit kann unmittelbar die Abbildung auf entsprechende Instanzen in den generierten Prototyp-Varianten vorgenommen werden.

Das in Abbildung 3.24 dargestellte Beispiel erläutert diese Verhältnisse an Hand einer CUC-Beschreibung zur Komponente `casa.data.Repository`. Im

```
Component: casa.storage.Repository
    Offered Interfaces:  casa.storage.Repository
    Required Interfaces: casa.data.DNASequence
    Signature:           casa.storage.Repository {
                             ...
                             void setExternalDB(java.net.URL db);
                             java.util.Set searchByName(String name);
                             ...
                         }
    States:              <Default>
    Initial State:       <Default>
    Provided Use Cases:
        Search * DNA
            Referenced Use Cases: casa.data.SeqImpl/Represent 1 DNA
            Interactions: Standard from <Default> to <Default>
                          Self.setExternalDB(1 Database)
                          * DNA = Self.searchByName(Name)
        ...
```

Abb. 3.24: *CUC-Beschreibung zur Komponente* `Repository`

gezeigten Ausschnitt des Anwendungsfalls Search * DNA sind die in der entsprechenden Interaktion referenzierten, primären und sekundären Konzepte dunkelgrau bzw. hellgrau hinterlegt. Mit Hilfe der Signatur der zugehörigen Schnittstelle wird deutlich, daß das Konzept Name durch den Basistyp `java.lang.String` repräsentiert wird, während eine externe Datenbank (Database) durch eine Netzwerk-Adresse des Typs `java.net.URL` angegeben wird. Diese Repräsentationen sind somit festgelegt und müssen bei Inanspruchnahme der entsprechenden Interaktion geeignet berücksichtigt werden. So können beispielsweise technisch kompatible Instanzen für sekundäre Konzepte erstellt und mit konkreten Belegungen initialisiert werden, wie später in Abschnitt 3.6.4 erläutert wird.

Demgegenüber läßt sich die verwendete Repräsentation des primären Konzepts DNA nicht unmittelbar aus der Signatur ableiten, weil dessen mengenwertige Kardinalität durch den Einsatz des generischen Typs `java.util.Set` realisiert ist. Für diesen Fall erlaubt die Modellierung eines CUC den Verweis auf zusätzliche Anwendungsfälle, welche die eingesetzten Repräsentationen eindeutig festlegen (vgl. Abschnitt 3.4). Im gezeigten Beispiel wird deshalb der Anwendungsfall Represent 1 DNA einer anderen Kom-

ponente `casa.data.SeqImpl` referenziert und somit die einzelnen Elemente der zurückgelieferten Menge genauer charakterisiert.

Nachdem die erforderlichen Repräsentationen für primäre und sekundäre Konzepte auf die oben vorgestellte Weise ermittelt sind[8], stellt sich unmittelbar die Frage nach ihrer technischen Kompatibilität. Schließlich ist gerade bei Verknüpfung von Komponenten unterschiedlicher Hersteller mit technisch nicht-kompatiblen Repräsentation des gleichen Konzepts zu rechnen (vgl. Abschnitt 3.1.3). Zur Lösung dieses grundlegenden Problems definiert das vorgestellte Framework sogenannte *Adapter*, die zwischen unterschiedlichen Repräsentationen auf technischer Ebene vermitteln. Ihre Aufgabe ist es, die vom jeweiligen Interaktionspartner erwartete Repräsentation über eine geeignete Schnittstelle zur Verfügung zu stellen und evtl. auftretende Zustandsänderungen des repräsentierten Konzepts zur Laufzeit abzugleichen. Aus Gründen der vereinfachten Handhabung während den späteren Schritten der Generierung wird hierbei eine einheitliche technische Realisierung jedes Adapters mit entsprechend ausgezeichneten Schnittstellen entwickelt.

Abbildung 3.25 verdeutlicht den gewählten Ansatz an Hand möglicher Repräsentationen des dunkel hervorgehobenen Konzepts Alignment. Um die Komponenten `Clustal` und `PhylipView` zur Berechnung bzw. Anzeige eines Sequenz-Alignments zu kombinieren, ist es erforderlich, zwischen den unterschiedlichen Repräsentationen `AlignmentImpl` und `PhylipAlignment` zu vermitteln. Aufgrund ihrer Herkunft und Signatur sind die beteiligten Schnittstellen `casa.data.Alignment` und `other.PhylipAlignment` technisch gesehen nicht kompatibel, d.h. das Typsystem der verwendeten Plattform erlaubt keine direkte Zuweisung zwischen Instanzen der zugehörigen Komponenten. Da beide Komponenten aber auf logischer Ebene das gleiche Konzept repräsentieren, kann ein geeigneter Adapter `AlignmentAdapter` konstruiert werden, der beide ursprünglich vorausgesetzten Schnittstellen bereitstellt. Dies erlaubt den Einsatz von Instanzen des Adapters in Interaktionen mit beiden oben genannten funktionalen Komponenten.

Hierbei wird eine gleichnamige Schnittstelle `AlignmentAdapter` eingesetzt, um zur Laufzeit die tatsächlich benutzten Instanzen der ursprünglichen Repräsentationen zu übergeben oder abzufragen. Diese Vorgabe erleichtert die Abbildung auf objekt-orientierte technische Infrastrukturen, die keine expliziten Schnittstellen oder Mehrfachvererbung unterstützen. In diesem Fall ist es möglich, daß die beteiligten Repräsentationen durch einfache Klassen

[8] Aus Gründen der Effizienz erfolgt die eigentliche Festlegung auf bestimmte Repräsentationen erst in der folgenden Phase der Generierung, sobald die genaue Zusammensetzung einer Prototyp-Variante bekannt ist.

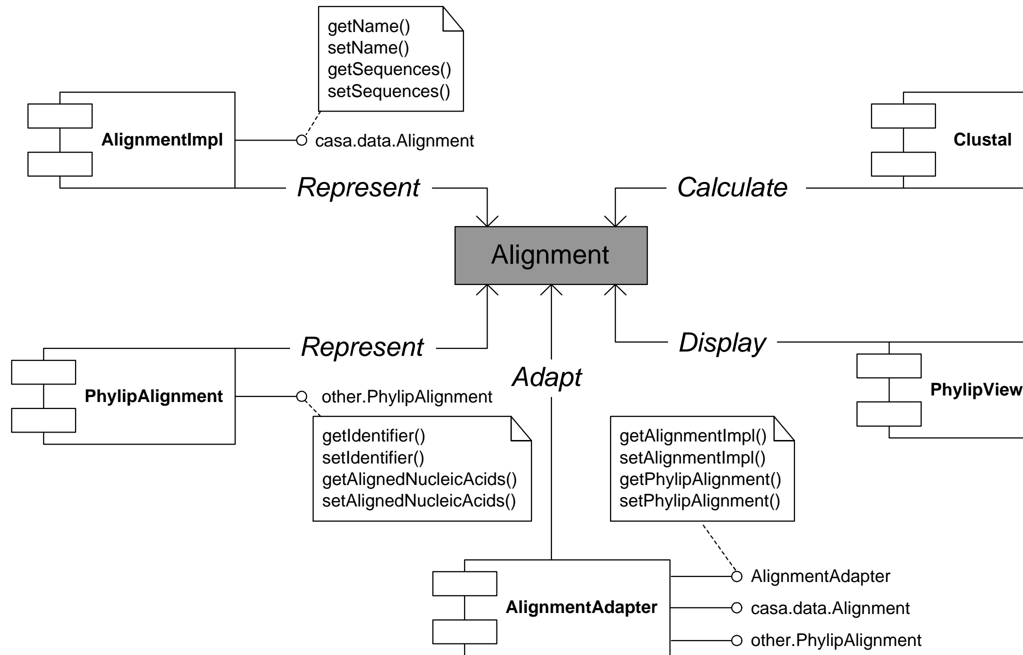


Abb. 3.25: *Beispiel eines Adapters für das Konzept Alignment*

ohne besondere Schnittstelle implementiert sind, d.h. deren als öffentlich deklarierte Methoden sind als eigentliche Schnittstelle im Sinne des Frameworks aufzufassen. Deshalb erlaubt es die vorgestellte, verallgemeinerte Variante des bekannten Design Patterns *Adapter* [GHJV94] die beteiligten Repräsentationen je nach Kontext der Interaktion auch unmittelbar selbst zu verwenden. Allerdings muß hierbei darauf geachtet werden, daß mögliche Zustandsänderungen einer so eingesetzten Repräsentation auch konsistent mit allen anderen beinhalteten Repräsentationen des betreffenden Adapters abgeglichen werden.

Im allgemeinen muß dieser Abgleich manuell vorgenommen werden, da hierfür eine genaue Kenntnis der beteiligten Repräsentationen und fachlicher Konsistenzbedingungen notwendig ist. Der Entwickler implementiert den betreffenden Adapter also selbst und stellt ihn dem übergeordneten Verfahren zur Verfügung. Allenfalls ein entsprechender Rahmen, also die Deklaration des Adapters mit dessen erforderlichen Schnittstellen und ihren Operationen, kann als Vorlage für eine spätere Implementierung automatisch generiert werden. Danach kann der betreffende Adapter offensichtlich in allen weiteren Abläufen des iterativen Prozesses wiederverwendet werden.

Unter bestimmten Voraussetzungen ist es jedoch möglich, den jeweils

erforderlichen Adapter auch vollständig automatisiert zu erstellen. Hierfür können wiederum die in der Ontologie festgelegten Informationen über Konzepte und ihre Relationen herangezogen werden. Schließlich bildet jede vorhandene Repräsentation ein Konzept des Anwendungsbereichs auf ein Element der technischen Ebene ab. Da Konzepte neben ihrem Namen und der übergeordneten Domäne gerade auch durch ihre Beziehungen zu anderen Konzepten charakterisiert sind, findet sich in der Implementierung einer Repräsentation typischerweise Funktionalität, um diese Beziehungen zu etablieren oder zu ermitteln.

Das in Abbildung 3.25 dargestellte Beispiel verdeutlicht diese plausible Annahme an Hand der verschiedenen Repräsentationen des Konzepts Alignment. Durch die zugrundeliegende Ontologie wird festgelegt, daß Alignment in der anwendungsspezifischen Beziehung hasName mit dem Konzept Name steht, also einen eigenen Namen trägt, und darüber hinaus aus einer Menge des Konzepts AlignedNucleicAcid zusammengesetzt ist (vgl. Abbildung 3.18). Tatsächlich bieten beide Schnittstellen `casa.data.Alignment` und `other.PhylipAlignment` der beteiligten Repräsentationen entsprechende Operationen zur Behandlung eben dieser Relationen an, auch wenn Namensgebung und Signatur nicht einheitlich gewählt sind. Falls nun die Parameter bzw. Rückgabewerte dieser Operationen einen technisch kompatiblen Typ besitzen, etwa `java.lang.String` für Name und Sequenzdaten, oder durch bereits vorhandene Adapter verknüpft werden können, so ist die automatische Generierung des Adapters `AlignmentAdapter` prinzipiell möglich.

Um den so skizzierten, grundlegenden Ansatz genauer zu beschreiben, definiert das vorgestellte Framework eine besondere Struktur der repräsentationsbezogenen Anwendungsfälle. Gemäß der hierarchischen Modellierung eines CUC (vgl. Abbildung 3.7), kann somit eine gegebene Repräsentation untergeordnete Anwendungsfälle beinhalten, die mittels der ausgezeichneten Namen `Set` und `Get` ihrer zugehörigen Manipulationen die Etablierung bzw. Abfrage möglicher Beziehungen des repräsentierten Konzepts erlauben. Hierbei geben die jeweiligen Interaktionen wie üblich die konkrete technische Realisierung dieser Anwendungsfälle an. Typischerweise finden sich die in Beziehung stehenden Konzepte als Parameter oder Rückgabewerte von entsprechenden Operationen der zugehörigen Schnittstelle.

Es ist zu beachten, daß die Modellierung eines CUC durchaus mehrere verschiedene Interaktionen zur Erfüllung des gleichen Anwendungsfalls zuläßt (siehe Abschnitt 3.4 sowie Abbildung 3.9). Somit können die mittels Get- und Set-Manipulationen behandelten Konzepte auch selbst über unterschiedliche Repräsentationen bereitgestellt werden. Dies führt u.a. zu einer höheren Flexibilität bei der späteren Suche nach technisch kompatiblen Repräsentationen der betrachteten Konzepte.

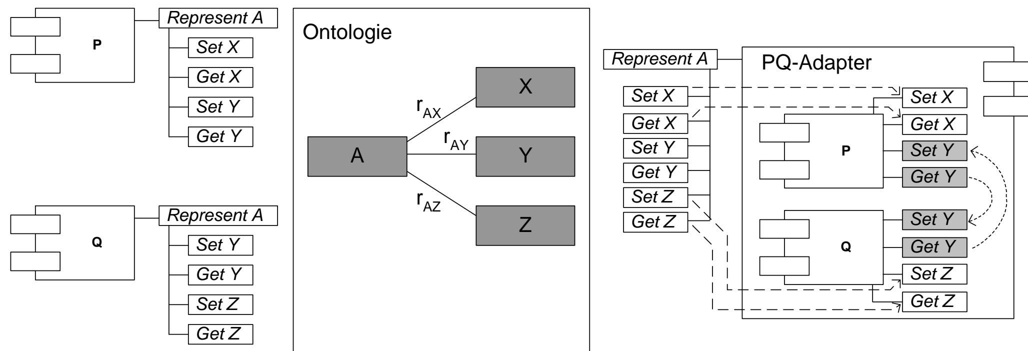


Abb. 3.26: *Schematisches Beispiel der Adapter-Konstruktion*

Abbildung 3.26 verdeutlicht das Prinzip der Adapter-Konstruktion sowie das resultierende, dynamische Verhalten durch ein schematisches Beispiel. Das in der Ontologie definierte Konzept A wird durch die beiden Komponenten `P` und `Q` repräsentiert, d.h. beide Komponenten bieten den CUC Represent A an. Von den möglichen Relationen zwischen A und den Konzepten X, Y und Z unterstützen `P` und `Q` jeweils die ersten bzw. letzten beiden Relationen über entsprechende, untergeordnete Anwendungsfälle. Hierbei wird deutlich, daß eine Ontologie nur die Menge der *möglichen* Beziehungen zwischen Konzepten definiert – eine gegebene Repräsentation kann durchaus nur eine Teilmenge dieser Beziehungen implementieren, etwa weil ursprünglich vorgesehene Nutzer der Repräsentation nur einen Ausschnitt des Modells bearbeiten.

Der automatisch generierte Adapter `PQ-Adapter` beinhaltet beide ursprünglichen Komponenten und delegiert spezifische Funktionalität an die jeweils betreffende Repräsentation. Soll eine Beziehung hergestellt oder abgefragt werden, die wechselseitig repräsentiert ist, im Beispiel also bei Ausführung der Anwendungsfälle Set Y und Get Y, so entscheidet der innerhalb der Interaktion benutzte Typ über die am besten geeignete Subkomponente. Alternativ können Interaktionspartner, die explizit eine bestimmte Repräsentation des betreffenden Konzepts erwarten, eine Referenz auf die entsprechende Subkomponente erhalten. In diesem Fall werden im weiteren Verlauf offensichtlich nur die jeweils spezifischen Anwendungsfälle der ausgewählten Komponente benutzt.

Jedoch ist in allen Fällen auf die *Konsistenz* beider Repräsentationen des Adapters achten. Sobald sich der Zustand einer Subkomponente ändert, muß der äquivalente Teilzustand der jeweils anderen Subkomponente entsprechend angepaßt werden. In diesem Zusammenhang ist der Zustand einer Repräsen-

tation als *Zusammenfassung aller etablierten Beziehungen* zu verstehen, d.h bei Änderung einer Beziehung durch die Ausführung einer mit `Set` bezeichneten Manipulation ändert sich auch der übergeordnete Zustand der betreffenden Repräsentation. In diesem Fall wird durch den Adapter zur Laufzeit ein Abgleich des aktuellen Zustands über die Kombination der komplementären Get- und Set-Manipulationen vorgenommen. Dieser Sachverhalt ist in Abbildung 3.26 durch gekrümmte, unterbrochene Pfeile zwischen den Anwendungsfällen Set Y und Get Y innerhalb des Adapters angedeutet. Das hierfür benötigte Protokoll kann, wie auch bei funktionalen Komponenten, der jeweils zugeordneten Interaktion entnommen werden (vgl. Abschnitt 3.6.4).

Der so durchgeführte Abgleich erfordert u.U. die Einbindung weiterer Adapter, falls die beteiligten Konzepte wiederum durch technisch nichtkompatible Komponenten repräsentiert sind bzw. keine entsprechende Interaktion der komplementären Get- und Set-Manipulationen gefunden werden kann. Insgesamt ergibt sich so eine hierarchische Organisation der Adapter-Suche und -Generierung, auf deren unterster Ebene ausschließlich technisch kompatible Repräsentationen ausgetauscht werden. Diese Form der Kompatibilität ergibt sich letztlich aus dem Typsystem der eingesetzten technischen Plattform oder den manuell vom Benutzer erstellten Adaptern. Letztere werden als vollständig kompatible Vermittler zwischen verschiedenen Repräsentationen angenommen. Sie können in das Framework über besondere Anwendungsfälle mit einer als `Adapt` gekennzeichneten Manipulation integriert werden (vgl. Abbildung 3.25).

Die Einbindung manuell erstellter Adapter ist letztlich notwendig, weil die oben erläuterte Strategie nur bei einfachen Beziehungsgeflechten der beteiligten Konzepte eingesetzt werden kann. So erfordert bereits eine rekursive Struktur der betrachteten Beziehungen, beispielsweise bei Definition des Konzepts Tree in Abbildung 3.18, eine andere, deutlich aufwendigere Vorgehensweise. Auch wenn später in Abschnitt 5.3 entsprechende Erweiterungen des Frameworks vorgestellt werden, sind im allgemeinen Fall, gerade bei komplexen Definitionen und Konsistenzbedingungen, manuelle Schritte unerläßlich. Dennoch kann der vorgestellte Ansatz zumindest den Rahmen und einfache Umsetzungen für die spätere, vollständige Implementierung vorgeben.

Dies gilt auch für die häufig genutzte Funktionalität, auf Änderungen des Zustands einer Repräsentation geeignet zu reagieren. Hierfür wird in der Praxis oftmals das Design Pattern *Observer* [GHJV94] oder vergleichbare Lösungen eingesetzt. Es erlaubt anderen Komponenten, sich bei einer gegebenen Repräsentation zu registrieren, die in der Folge auftretende Änderungen ihres Zustands über asynchrone, ereignisbasierte Kommunikation bekannt gibt. Die so benachrichtigten Komponenten können im Anschluß den aktuellen Zu-

stand der betreffenden Repräsentation ermitteln und für ihre Zwecke nutzen, etwa zur Aktualisierung der Bildschirmdarstellung.

Die in Abschnitt 3.5 vorgeschlagene Modellierung der Funktionalen Spezifikation unterstützt diese Form der Entkopplung explizit über den vordefinierten, ereignisbasierten Auslöser On Change (vgl. Abbildung 3.13). Bei Eintritt einer Zustandsänderung des referenzierten Konzepts werden die folgenden Anwendungsfälle gemeinsam ausgeführt, wie auch am gewählten Anwendungsbeispiel deutlich wird (vgl. Abbildung 3.19). Falls jedoch die gewählte Repräsentation des Konzepts kein derartiges Ereignismuster implementiert, kann die verfügbare Information über Get- und Set-Manipulationen zu dessen Generierung herangezogen werden. Hierbei wird die ursprüngliche Repräsentation Bestandteil eines sog. *Wrappers*, der selbst zwar sämtliche Anwendungsfälle an diese Subkomponente delegiert, aber bei Ausführung einer Set-Manipulation darüber hinaus ein entsprechendes Ereignis auslöst. Auf diese Weise ermöglichen die innerhalb eines CUC bereitgestellten Informationen eine weitere Reduktion des insgesamt erforderlichen manuellen Aufwands bei der Konstruktion funktionaler Prototypen.

Zusammenfassend läßt sich feststellen, daß die Modellierung einer Repräsentation von Konzepten des Anwendungsbereichs eine zentrale Rolle innerhalb des Frameworks einnimmt. Sie verdeutlicht den Übergang von logischer zu technischer Ebene des vorgestellten Ansatzes und bestimmt letztlich die möglichen Verknüpfungen unabhängig entwickelter Komponenten. Hierbei erlaubt eine vordefinierte Struktur mit untergeordneten Anwendungsfällen zur Ermittlung und Änderung des Zustands eine zumindest teilweise automatisierte Generierung von Adaptern, die zwischen technisch nicht-kompatiblen Repräsentationen des gleichen Konzepts vermitteln. Trotzdem kann im allgemeinen Fall nicht auf eigens bereitgestellte Funktionalität zur Vermittlung verzichtet werden. Deshalb ergibt sich insgesamt folgende Präferenz bei Auswahl geeigneter Repräsentationen und ihrer Verknüpfung im Rahmen von Interaktionen:

- Zunächst werden bevorzugt technisch kompatible Repräsentationen ausgewählt, d.h. die beteiligten Instanzen der Komponenten können aufgrund ihres Typs einander zugewiesen werden. Hierfür sollte das Typsystem der verwendeten technischen Infrastruktur entsprechende Informationen zur Laufzeit anbieten.

- Falls keine derartigen Repräsentationen existieren, werden vom Benutzer explizit bereitgestellte Adapter herangezogen. Diese bieten verschiedene, technisch kompatible Schnittstellen für die von ihnen vermittelten Repräsentationen an.

- Falls kein geeigneter, manuell erstellter Adapter vorhanden ist, wird versucht, eine entsprechende Komponente automatisch mittels der oben beschriebenen Strategie zu generieren. In der Regel führt dies zur Suche nach untergeordneten Repräsentationen, die selbst wiederum der gleichen Präferenz folgt.

- Falls die Struktur der Ontologie keine Anwendung der oben beschriebenen Strategie erlaubt, wird zumindest ein entsprechender Rahmen generiert, der durch den Benutzer geeignet zu ergänzen ist.

Diese variable Abstufung bei Auswahl und Verknüpfung von Repräsentationen erlaubt eine spätere, automatisierte Teilbewertung der generierten Prototyp-Varianten hinsichtlich ihrer Qualität, wie in Abschnitt 3.6.5 erläutert wird. Schließlich ist die Auswahl geeigneter Repräsentationen sowie die Integration erforderlicher Adapter kein einmaliger Vorgang, sondern durch die erzielten Ergebnisse der Varianten-Generierung unmittelbar in den iterativen Gesamtprozeß eingebunden (vgl. Abbildung 3.17). Dieser nächste grundlegende Schritt des vorgestellten Verfahrens ist Gegenstand des folgenden Abschnitts.

3.6.4 Varianten-Generierung

Nachdem in den vorangegangenen Schritten eine Auswahl potentiell geeigneter funktionaler Komponenten sowie möglicher Repräsentationen für Konzepte ermittelt wurde, sind nun konkrete Festlegungen zu treffen, die eine Generierung übersetzbarer und ausführbarer Prototypen erlauben. Hierbei läßt sich das eingesetzte Verfahren bei genauerer Betrachtung in mehrere Teilschritte untergliedern, die in der folgenden Übersicht zusammengefaßt sind:

1. Zunächst wird die genaue Zusammensetzung der initialen Prototyp-Varianten bestimmt, d.h. aus der Menge an möglichen Komponenten und Repräsentationen werden für jede Variante bestimmte, jeweils unterschiedliche Vertreter ausgewählt und den Anwendungsfällen bzw. Zwischenergebnissen der Spezifikation zugeordnet. Der hierdurch festgelegte *Konstruktionsplan* beinhaltet entsprechende Instanzen, also eindeutig benannte Referenzen mit zugehörigem Typ, für jede so ausgewählte Komponente oder Repräsentation.

2. Nunmehr können die operativen Elemente der Funktionalen Spezifikation zur Erstellung weiterer Strukturen herangezogen werden. Hierbei werden die jeweils festgelegten CUCs entsprechend den funktionalen Aussagen gruppiert und ggf. unter die Kontrolle des angegebenen

Auslösers gestellt. Im Falle eines ereignisbasierten Auslösers ist das zugehörige Ereignismuster der bereitstellenden Komponente geeignet zu initialisieren sowie eine möglicherweise zugeordnete Interaktion als erster Bestandteil der zusammengefaßten Anwendungsfälle umzusetzen.

3. Danach werden die im Konstruktionsplan aufgeführten Komponenten über ihre in den Anwendungsfällen beschriebenen Interaktionen verknüpft. Hierfür werden die Rollen einer Interaktion den Instanzen der beteiligten Komponenten zugeordnet, sowie die referenzierten primären und sekundären Konzepte mit Instanzen der zugehörigen Repräsentationen belegt. Darüber hinaus sind evtl. zusätzliche Interaktionen erforderlich, um eine gegebene Komponente in den jeweils vorausgesetzten, von außen beobachtbaren Zustand zu überführen.

4. Anschließend werden die innerhalb der Interaktionen referenzierten Repräsentationen hinsichtlich ihrer technischen Kompatibilität überprüft. Diese Überprüfung führt ggf. zur Einbindung entsprechender Adapter, die möglicherweise eigens generiert werden müssen, wie im vorherigen Abschnitt beschrieben ist. Wiederum werden bei Bedarf eigene Instanzen für die verwendeten Adapter erstellt, initialisiert und in den Konstruktionsplan der betreffenden Variante integriert.

5. Schließlich kann für den so fertiggestellten Konstruktionsplan Quellcode generiert werden, der sich übersetzen und ausführen läßt. Dieser Teilschritt beinhaltet zahlreiche technische Details, die maßgeblich durch die Vorgaben der eingesetzten technischen Plattform bestimmt werden. Aus diesem Grund werden hierfür nur exemplarische Ausschnitte der im Rahmen dieser Arbeit betrachteten Java-Implementierung vorgestellt.

Diese so zusammengefaßten Teilschritte der Varianten-Generierung werden im folgenden an Hand des in Abschnitt 3.6.1 eingeführten Anwendungsbeispiels näher erläutert. In diesem Zusammenhang ist die detaillierte Darstellung des oben erwähnten Konstruktionsplans einer Prototyp-Variante allerdings wenig geeignet, die vorgestellte Konzeption anschaulich zu vermitteln. Ein derartiger Konstruktionsplan stellt vielmehr ein besonderes Zwischenformat dar, daß im Verlauf der Varianten-Generierung bearbeitet, verändert und zunehmend erweitert wird, bis er schließlich in Quellcode umgesetzt werden kann.

Aus diesem Grund werden bei Erläuterung der oben aufgeführten Teilschritte die jeweils erzielten Ergebnisse im Vorgriff als Java-Quellcode der generierten Variante dargestellt. Dies erleichtert das Verständnis im Hinblick auf das erreichte Endergebnis und belegt nicht zuletzt auch die erfolgreiche

praktische Umsetzung des erarbeiteten Frameworks. Das vollständige Anwendungsbeispiel sowie der Quellcode sämtlicher generierter Prototyp-Varianten findet sich unter [Vil01b].

Teilschritt 1

Zu Beginn des Verfahrens werden die verschiedenen Prototyp-Varianten durch Kombination der zuvor ausgewählten Komponenten und Repräsentationen festgelegt. Die Anzahl aller unterschiedlichen Varianten N_V ist bei einer gegebenen Funktionalen Spezifikation mit n Anwendungsfällen und m referenzierten, primären Konzepten offensichtlich durch das Produkt der Anzahl an jeder Position möglicher Variationen gegeben. Dieser Sachverhalt wird durch Gleichung 3.5 wiedergegeben, wobei c_i die Anzahl der zugeordneten Komponenten für den Anwendungsfall i und r_j die Anzahl der zugeordneten Repräsentationen für das primäre Konzept j beschreibt.

$$N_V = \prod_{i=1}^{n} c_i \cdot \prod_{j=1}^{m} r_j \tag{3.5}$$

Somit führt die in den Abschnitten 3.6.2 und 3.6.3 erläuterte Auswahl bereits für das einfache Anwendungsbeispiel zu insgesamt $(2{\cdot}2){\cdot}(2{\cdot}2{\cdot}2{\cdot}2) = 64$ möglichen, unterschiedlich zusammengesetzten Varianten (vgl. Abbildung 3.22 und 3.23). Dies verdeutlicht die Notwendigkeit, die so entstehende kombinatorische Vielfalt durch eine geeignete Heuristik zu begrenzen. Die hierfür in dieser Arbeit entwickelte Lösung wird später in Abschnitt 3.6.5 im Detail vorgestellt. An dieser Stelle genügt es festzustellen, daß die Auswahl einer beherrschbaren Teilmenge aller Kombinationen durch ein zufallsgesteuertes Verfahren erfolgt.

Abbildung 3.27 zeigt das Ergebnis der Festlegung in Teilschritt 1 für eine bestimmte Kombination aus Komponenten und Repräsentationen des Anwendungsbeispiels. Der hier dargestellte, initiale Konstruktionsplan für die Variante `SequenceTool_1` ordnet jedem Anwendungsfall der Spezifikation genau *eine* Instanz der mit ihnen durch eine durchgezogene Linie verbundenen Komponente zu. Demgegenüber werden die dunkel hervorgehobenen, primären Konzepte der Spezifikation ggf. durch *verschiedene* Instanzen der mit ihnen durch eine unterbrochene Linie verbundenen Komponenten repräsentiert. Im Beispiel betrifft dies die Konzepte `1..* DNA 'result'` und `1 DNA 'selected'`, die zwar beide durch die gleiche Komponente `SeqImpl` repräsentiert sind, aber offensichtlich in unterschiedlichen Anwendungsfällen referenziert werden.

Im erstgenannten Fall muß darüber hinaus eine Anpassung der Kardinalität vorgenommen werden, da `SeqImpl` tatsächlich nur eine einzige DNA-

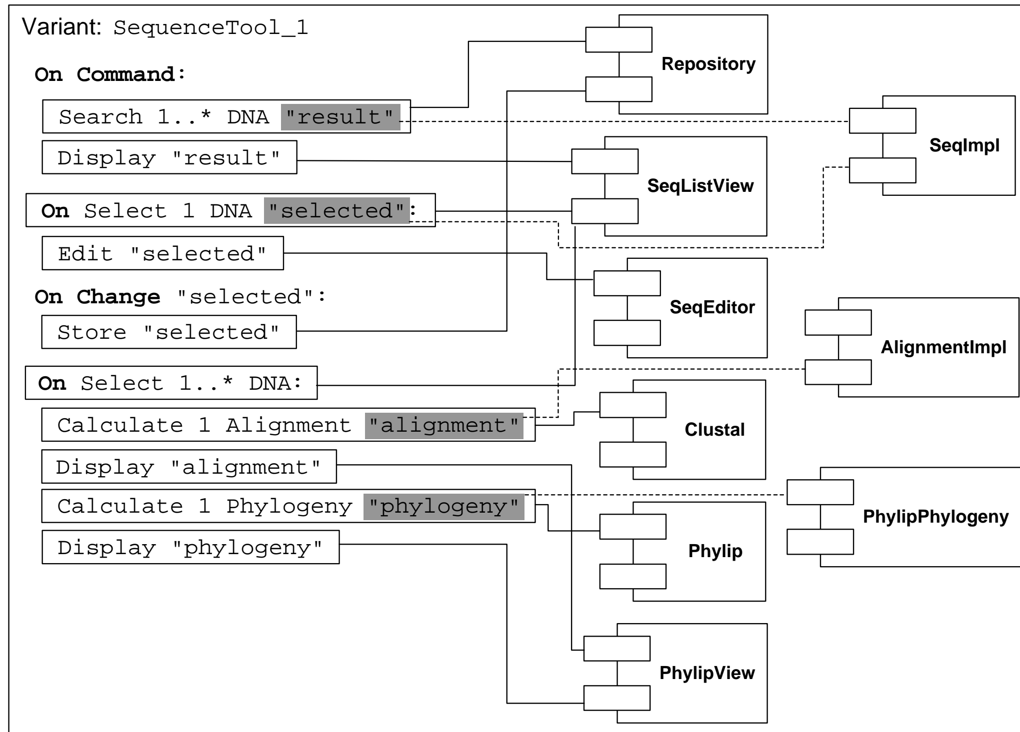


Abb. 3.27: *Eine mögliche Variante des Anwendungsbeispiels*

Sequenz repräsentiert (vgl. Abbildung 3.20). Aus diesem Grund wird zusätzlich eine nicht dargestellte, generische Hilfskomponente `Collection` eingeführt, die eine Menge von Instanzen der gleichen Komponente verwaltet. Sie wird durch das Framework bereitgestellt, das somit auch bei Bedarf auf die individuellen Repräsentationen zurückgreifen kann. Die Namensgebung der Instanzen funktionaler Komponenten ist beliebig (wenn auch eindeutig), während die gewählten Namen für Repräsentationen typischerweise den in der Spezifikation angeführten Referenzen folgen.

Um dieses Vorgehen zu illustrieren, ist im folgenden ein Ausschnitt des später in Teilschritt 5 für diese Teilaufgabe generierten Quellcodes dargestellt. Er ergibt sich aus der im Rahmen dieser Arbeit entwickelten Referenz-Implementierung für die Java-Plattform und besitzt somit ausschließlich exemplarischen Charakter. Wie oben bereits erwähnt, lassen sich offensichtlich auch andere technische Plattformen oder Code-Generatoren in das Framework integrieren.

```
class SequenceTool_1 {
  ...
  // Unique main instance of current variant
  static SequenceTool_1 thisVariant;

  // Primary concepts
  SeqImpl selected = new SeqImpl();
  Collection result = new Collection("casa.data.SeqImpl");
  ...
  // Functional components
  Repository repository = new Repository();
  SeqListView seqListView = new SeqListView();
  ...
}
```

Durch den gezeigten Ausschnitt wird deutlich, daß sich die getroffene Auswahl funktionaler Komponenten und Repräsentationen auf einfache Weise im generierten Code wiedergeben läßt[9]. Die Prototyp-Variante selbst ist als eigene Klasse realisiert, deren einzige Instanz zur Laufzeit geeignet initialisiert wird. Die verwendeten Repräsentationen und funktionalen Komponenten werden zu globalen Attributen dieser Klasse, deren Erstellung in der Regel durch Aufruf eines parameterlosen Konstruktors geschehen kann. Lediglich die oben erwähnte Hilfskomponente `Collection` benötigt zusätzlich die Angabe der durch sie verwalteten Repräsentation.

Teilschritt 2

In diesem Teilschritt des Verfahrens wird die Struktur sowie der grundlegende Kontrollfluß festgelegt. Diese Zusammenhänge ergeben sich im wesentlichen aus den Vorgaben der Funktionalen Spezifikation hinsichtlich funktionaler Aussagen und den für sie angegebenen Auslösern (vgl. Abschnitt 3.5). Durch die bewußt einfach gehaltene Modellierung einer Funktionalen Spezifikation ist diese Umsetzung mit geringem Aufwand verbunden. Während für die unmittelbar durch den Benutzer kontrollierten Auslöser geeignete Elemente der grafischen Oberfläche, etwa Menüs oder Schaltflächen, eingebunden werden, erfordern ereignisbasierte Auslöser die Behandlung des entsprechenden Ereignis-Musters der bereitstellenden Komponente (siehe Abschnitt 3.4 sowie Abbildung 3.11). Um eine übersichtliche Struktur des generierten Quellco-

[9] Aus Gründen der Übersichtlichkeit wird vorausgesetzt, daß die verwendeten Bezeichner für Klassen eindeutig gewählt und durch entsprechende Import-Anweisungen zur Verfügung gestellt werden.

des zu erreichen, ist hierfür die Verwendung eines automatisch generierten Ereignis-Adapters vorgesehen, der zwischen Quelle und Verarbeitung des Ereignisses vermittelt. Dies entspricht dem üblichen Vorgehen bei Realisierung ereignisbasierter Kommunikation, etwa gemäß den Konventionen der Java-Plattform.

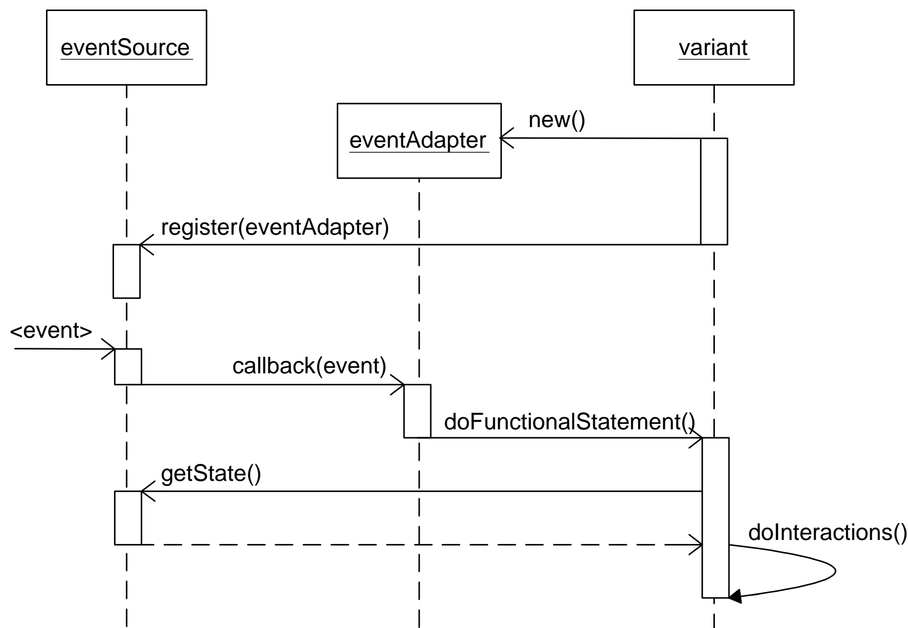


Abb. 3.28: *Schematischer Ablauf der ereignisbasierten Kommunikation*

Abbildung 3.28 verdeutlicht den gewählten Ansatz durch eine schematische Darstellung des resultierenden dynamischen Ablaufs als UML-Sequenzdiagramm. Im Verlauf ihrer Initialisierung erstellt die Variante `variant` eine neue Instanz des Ereignis-Adapters `eventAdapter` und registriert diese bei der Instanz `eventSource` derjenigen Komponente, die zuvor in Teilschritt 1 als Quelle des Ereignisses festgelegt wurde. Hierbei ist die erwartete Schnittstelle des Adapters sowie die in der Abbildung mit `register` bezeichnete Operation durch das jeweilige Ereignis-Muster bestimmt. Der generierte Ereignis-Adapter wird also mit der vordefinierten Rolle Listener assoziiert. Tritt nun im weiteren Verlauf das mit `<event>` bezeichnete Ereignis ein, so ruft die betreffende Komponente eine im Ereignis-Muster als callback spezifizierte Operation des Adapters auf. Der Ereignis-Adapter sei-

nerseits benachrichtigt daraufhin eine zuvor bestimmte, im Beispiel mit `doFunctionalStatement` bezeichnete Operation der übergeordneten Variante. Der tatsächliche Name ist frei (aber eindeutig) wählbar, weil dieser erst bei Generierung des Ereignis-Adapters festgelegt wird.

Bevor innerhalb dieser Operation eine durch den Auslöser kontrollierte, funktionale Aussage behandelt werden kann, ist zunächst eine evtl. im Ereignis-Muster spezifizierte Interaktion mit der betreffenden Komponente durchzuführen. Diese, in der Abbildung durch den Aufruf der Operation `getState` vereinfacht dargestellte Interaktion dient der Ermittlung möglicher Ergebnisse, welche als Folge des Ereignisses übermittelt werden. Die Zuordnung dieses Ergebnisses zu primären oder sekundären Konzepten der Variante erfolgt entweder unmittelbar über die verwendete Referenz im ereignisbasierten Auslöser oder indirekt über möglicherweise benötigte Parameter für weitere Interaktionen, wie später beschrieben wird.

Wiederum illustriert ein Ausschnitt des später in Teilschritt 5 generierten Quellcodes die praktische Anwendung des oben beschriebenen Verfahrens. Hierfür sei die in Abbildung 3.12 gezeigte CUC-Beschreibung zur Komponente `SeqListView` gegeben. Sie spezifiziert das erforderliche Ereignis-Muster, um die von ihr angebotene Funktionalität Select 1..* NucleicAcid auf technischer Ebene umzusetzen. Die Realisierung des ereignisbasierten Auslösers `On Select 1 DNA 'selected'` der gegebenen Funktionalen Spezifikation (vgl. Abbildung 3.19), erfordert darüber hinaus eine Berücksichtigung der unterschiedlichen Kardinalität, wie später erläutert wird.

```
class SequenceTool_1 {
  // Generated event adapter as inner class
  class SelectionAdapter1
  implements casa.gui.SelectionListener {
    public void seqSelected(casa.gui.SelectionEvent e) {
      thisVariant.selected_DNA_Selected();
    }
  ...
  // Initialisation
  void init() {
    // Create and register event adapters
    SelectionAdapter1 sa = new SelectionAdapter1();
    seqListView.register(sa);
    ...
  }
  ...
```

```
  void selected_DNA_Selected() {
    // Retrieve selected concept 1 DNA
    casa.data.Sequence[] tmp  = seqListView.getSelected();
    selected = tmp[0];
    // Further interactions within given statement
    ...
  }
}
```

Das oben stehende Code-Fragment der generierten Prototyp-Variante verdeutlicht die automatische Umsetzung der in Abbildung 3.28 gezeigten Aufteilung. Der Ereignis-Adapter `SelectionAdapter1` wird als untergeordnete Subklasse der Prototyp-Variante umgesetzt. Hierbei können die erforderlichen Schnittstellen, Operationen und deren Signatur unmittelbar aus den Informationen der CUC-Beschreibung bzw. dem Typsystem der technischen Infrastruktur entnommen werden. Hierfür muß die verwendete Plattform derartige Informationen allerdings zur Laufzeit bereitstellen, wie dies etwa bei Java über den sog. Reflection-Mechanismus geschieht. In der Implementierung der Methode `init` wird nun eine Instanz `sa` des Adapters erstellt und bei der Komponente `SeqListView` registriert. Dies entspricht einer Zuordnung der Instanz `sa` zur Rolle Listener im zugehörigen Ereignis-Muster.

Schließlich dient die Methode `selected_DNA_Selected` zur Gruppierung der durch den Auslöser kontrollierten Anwendungsfälle. Die Namensgebung ist zwar grundsätzlich beliebig, wird im Beispiel aber aus dem Namen von Referenz und Ereignis gebildet, um somit einen eindeutigen Bezeichner zu erhalten. Zu Beginn der aufgeführten Implementierung wird das mit dem Ereignis verbundene Ergebnis, also die Menge der durch den Benutzer ausgewählten Nukleinsäure-Sequenzen, in einer Zwischenvariable `tmp` gespeichert. Dies entspricht einer Realisierung der im zugehörigen CUC spezifizierten Interaktion, wobei die vordefinierte Rolle Self mit der Instanz `seqListView` assoziiert wird[10]. Die anschließende Zuweisung des ersten Elements der Zwischenvariablen `tmp` zur Instanz `selected` als Repräsentation des primären Konzepts 1 DNA verwirklicht den notwendigen Abgleich der Kardinalität, da ja in der Spezifikation nur eine einzige selektierte Sequenz erwünscht ist. Es ist zu beachten, daß an dieser Stelle kein Adapter für die beteiligten Repräsentationen erforderlich ist, weil die Komponente `SeqImpl` ebenfalls die Schnittstelle `casa.data.Sequence` implementiert (vgl. Abbildung 3.20).

[10] Der im CUC für die Ausführung dieser Interaktion eigentlich vorausgesetzte Zustand `<SeqDisplayed>` wird innerhalb der gleichen Variante durch vorherige Inanspruchnahme des Anwendungsfalls Display 1..* NucleicAcid erreicht und muß daher an dieser Stelle nicht berücksichtigt werden.

Teilschritt 3

Anschließend werden die von den Komponenten angebotenen Anwendungsfälle verknüpft sowie die betreffenden Interaktionen in die zuvor erstellte Struktur der Prototyp-Variante integriert. Hierfür können aus den spezifizierten Interaktionen diejenigen ausgewählt werden, deren beteiligte Rollen sowie referenzierte primäre und sekundäre Konzepte möglichst vollständig im Kontext der betrachteten Variante belegt werden können. Für diesen Teilschritt bestehen innerhalb des Frameworks prinzipiell verschiedene Lösungsmöglichkeiten und Strategien, die je nach Grad der Automatisierung und erwünschter Qualität des Verfahrens herangezogen werden können. Um eine möglichst weitreichende Automatisierung zu erreichen, werden in der prototypischen Referenz-Implementierung (siehe Kapitel 4) folgende Vereinfachungen vorausgesetzt:

- Die in den bereitgestellten CUC-Beschreibungen enthaltenen Interaktionen beschränken sich auf die vordefinierten Rollen Client und Self.

- Sekundäre Konzepte als Parameter von Operationen besitzen Repräsentationen, die über vorhandene Komponenten oder Basistypen der technischen Plattform initialisiert werden können.

Während die letztgenannte Voraussetzung tatsächlich nur eine technisch motivierte Vereinfachung darstellt, ist die zuvor genannte Beschränkung auf binäre Interaktionen mit vordefinierten Rollen für die möglichst allgemeine Beschreibung von Funktionalität offensichtlich zu restriktiv. Deshalb werden später in Abschnitt 5.3 entsprechende Anpassungen und Erweiterungen des Frameworks für diesen Bereich diskutiert. Die angenommenen Voraussetzungen erlauben jedoch bereits eine automatische Generierung zahlreicher, nicht-trivialer Prototypen, wie am vorgestellten Anwendungsbeispiel deutlich wird.

So kann etwa die in Abbildung 3.10 gezeigte Beschreibung des CUC Calculate 1 Alignment zur Komponente `Clustal` herangezogen werden, um den entsprechenden Anwendungsfall der Spezifikation zu implementieren. Hierbei wird das primäre Konzept Alignment in der zugehörigen Interaktion mit der Instanz `alignment` der gewählten Repräsentation `AlignmentImpl` belegt (vgl. Abbildung 3.22). Schließlich ist Alignment auch als primäres Konzept der übergeordneten Funktionalen Aussage definiert (vgl. Abbildung 3.27). Hingegen ist eine mögliche Belegung für das im Rahmen der Interaktion als Parameter benötigte, sekundäre Konzept 1..* DNA nicht unmittelbar ersichtlich. In einem solchen Fall wird in der gegenwärtigen Implementierung zunächst nach logisch kompatiblen Rückgabewerten aus vorangegangenen Interaktionen gesucht. Diese plausible Heuristik führt im Beispiel

zum Ergebnis des Anwendungsfalls Select 1..* DNA, der als ereignisbasierter Auslöser durch die Spezifikation angegeben ist. Somit sind alle benötigten Informationen verfügbar und die betreffende Interaktion kann später in Teilschritt 5 umgesetzt werden, wie der folgende Ausschnitt des generierten Codes verdeutlicht.

```
class SequenceTool_1 {
  AlignmentImpl alignment = new AlignmentImpl();
  Clustal clustal = new Clustal();
  ...
  // Generated event adapter as inner class
  class SelectionAdapter2
  implements casa.gui.SelectionListener {
    public void seqSelected(casa.gui.SelectionEvent e) {
      thisVariant.multiple_DNA_Selected();
    }
  ...
  // Initialisation
  void init() {
    SelectionAdapter2 sa2 = new SelectionAdapter2();
    seqListView.register(sa2);
    ...
  }
  ...
  void multiple_DNA_Selected() {
    // Retrieve selected concept 1..* DNA
    casa.data.Sequence[] multi  = seqListView.getSelected();
    // Interaction Calculate 1 Alignment
    Collection tmp = new Collection("casa.data.SeqImpl");
    tmp.fromArray(multi);
    clustal.setSequences(tmp.toList());
    alignment = clustal.calcAlignment(0.75);
    // Further interactions within given statement
    ...
  }
}
```

In dem oben stehenden Code-Fragment sind zunächst aus Gründen der Anschaulichkeit die jeweiligen Ergebnisse der vorangegangenen Teilschritte, also Instanzen für Repräsentationen und funktionale Komponenten sowie Strukturen für Ereignis-Muster, nochmals aufgeführt. In der Methode

multiple_DNA_Selected sind die neu hinzugekommenen Bestandteile zusammengefaßt. Entsprechend der in Abbildung 3.10 spezifizierten Interaktion ist zunächst die Übergabe der betreffenden DNA-Sequenzen durch Aufruf der Operation setSequences erforderlich. Aus technischen Gründen muß hierbei zwischen der Repräsentation multi und der aktuellen Parameterbelegung vermittelt werden, weil die vorangegangene Interaktion mit der Komponente SeqListView eine Reihung zurückliefert, während der formale Parameter s der Operation setSequences den Typ java.util.List besitzt. Eine derartige Umsetzung wird durch die generische Hilfskomponente Collection realisiert, die vom Framework bereitgestellt wird. Aufgrund ihrer Konstruktion kann sie zwischen gängigen Implementierungen der verwendeten technischen Plattform für mengenwertige Konzepte vermitteln. Anschließend wird das gewünschte Sequenz-Alignment durch Aufruf der Operation calcAlignment berechnet und der entsprechenden Repräsentation zugewiesen. In diesem Fall ist der Wert 0.75 für den Parameter gapPenalty durch einen konstanten Ausdruck innerhalb der CUC-Beschreibung festgelegt (vgl. Abbildung 3.10). Die so beschriebene Umsetzung der Interaktion entspricht also einer Abbildung der Rolle Self auf die Instanz clustal der gleichnamigen Komponente sowie implizit der Rolle Client auf die Instanz der betrachteten Variante selbst. Eine derartig schematische Zuordnung ist allerdings nur unter den oben genannten Voraussetzungen möglich.

Während im oben aufgeführten Beispiel das als Parameter benötigte sekundäre Konzept durch Suche nach Rückgabewerten vorheriger Interaktionen gefunden wird, ist diese einfache Strategie im allgemeinen Fall offensichtlich nicht anwendbar. So verlangt die in Abbildung 3.24 dargestellte Beschreibung des CUC Search 1..* DNA zur Komponente Repository die Angabe einer Datenbank als Parameter der Operation setExternalDB, bevor durch Aufruf der Operation searchByName die eigentliche Suche erfolgen kann. Die beiden beteiligten sekundären Konzepte Database und Name können jedoch im Kontext der betrachteten Prototyp-Variante nicht unmittelbar zugeordnet werden. Somit muß eine Lösung gefunden werden, die zumindest eine Generierung übersetzbaren und ausführbaren Quellcodes erlaubt. Hierbei sind tatsächlich nur Parameterbelegungen zu berücksichtigen, da ggf. anfallende Rückgabewerte einfach entsprechenden Variablen eines durch die Signatur gegebenen Typs zugewiesen werden können.

Für die Ermittlung nicht zugeordneter, sekundärer Konzepte als Parameter bei der Umsetzung einer gegebenen Interaktion sind je nach Typ des Konzepts, verfügbaren Metainformationen und gewünschtem Grad der Automatisierung verschiedene Strategien möglich:

- Auswahl einer evtl. angegebenen, alternativen Interaktion für den gleichen CUC, deren benötigte sekundäre Konzepte problemlos zugeordnet werden können.

- Generierung einfacher Elemente der grafischen Oberfläche für sekundäre Konzepte, welche durch Basistypen der technischen Plattform repräsentiert sind.

- Unmittelbare Belegung mit neu erstellten Instanzen des in der Signatur gegebenen Typs.

- Suche nach technisch kompatiblen Repräsentationen des betreffenden Konzepts, die eine automatische Initialisierung erlauben, beispielsweise durch vorhandene funktionale Komponenten mit ausgezeichneten Manipulationen wie `Select` oder `Initialize`.

- Generierung des Rahmens einer neuen Komponente, die zur Erstellung und Initialisierung des benötigten Konzepts dient. Diese Komponente ist später vom Benutzer manuell zu implementieren.

- Explizite Zuordnung zwischen Instanzen und sekundären Konzepten durch den Benutzer im Verlauf eines interaktiven Verfahrens.

In der vorliegenden Arbeit wird das Ziel verfolgt, eine möglichst weitreichende Automatisierung der Prototyp-Generierung zu erreichen. Deshalb werden durch die gegenwärtige Implementierung v.a. die zuerst aufgeführten Strategien verfolgt. Die Entscheidung, ob das Verhalten der auf diese Weise generierten Varianten auch tatsächlich den Erwartungen des Anwenders entspricht, muß später im Verlauf der Varianten-Bewertung letztlich im Dialog mit dem Entwickler getroffen werden (siehe Abschnitt 3.6.5).

Unter der oben genannten Voraussetzung, daß für jedes sekundäre Konzept eine zugehörige Komponente existiert, die zu ihrer Initialisierung herangezogen werden kann, läßt sich die Umsetzung des Anwendungsfalls Search 1..* DNA im Anwendungsbeispiel am folgenden Ausschnitt des generierten Codes nachvollziehen.

```
class SequenceTool_1 {
  Collection result = new Collection("casa.data.SeqImpl");
  Repository repository = new Repository();
  SeqListView seqListView = new SeqListView();
  ...
  void doCommand_1() {
    // Component for initialisation of concept Database
```

```
    casa.storage.DBSelector sel=new casa.storage.DBSelector();
    // Assign secondary concepts
    URL db = sel.selectDatabase();
    String name = nameTextField.getText();
    // Interaction for "Search 1..* DNA 'result'"
    repository.setExternalDB(db);
    Set tmp = repository.searchByName(name);
    result.fromSet(tmp);
    // Interaction for "Display 'result'"
    seqListView.setSequences(result.toList());
  }
  ...
}
```

Wiederum sind zunächst die Ergebnisse der vorangegangenen Teilschritte, also die Festlegung der Repräsentation für das primäre Konzept `result` sowie Instanzen der funktionalen Komponenten `Repository` und `SeqListView`, erneut aufgeführt. Die Implementierung der Methode `doCommand_1` beinhaltet die Realisierung der spezifizierten Anwendungsfälle `Search 1..* DNA 'result'` und `Display 'result'`, wie in Abbildung 3.27 dargestellt ist.

Hierbei wird vorausgesetzt, daß eine Komponente `DBSelector` gefunden werden kann, die den Anwendungsfall Select 1 Database über eine entsprechende Interaktion realisiert. Sie dient der Initialisierung des sekundären Konzepts `db`, beispielsweise durch Anzeige einer Auswahlliste verschiedener Datenbanken im Rahmen eines interaktiven Dialogs. Der zusätzlich benötigte Parameter `name` besitzt den Basistyp `java.lang.String` der technischen Plattform und kann somit über den Inhalt eines generischen Eingabefelds `nameTextField` bestimmt werden. Die hierfür nötige Initialisierung und Einbindung in die grafische Benutzeroberfläche der Variante ist aus Gründen der Übersichtlichkeit nicht dargestellt. Anschließend kann die betreffende Interaktion mit der Instanz `repository` ausgeführt (vgl. Abbildung 3.24) und deren Ergebnis der Komponente `SeqListView` zur Anzeige übergeben werden. Hierfür ist eine Vermittlung zwischen den zugehörigen Typen `java.util.Set` und `java.util.List` erforderlich, die wiederum durch die Hilfskomponente `Collection` vorgenommen wird.

Eine letzte, durchaus bedeutende Aufgabe dieses Teilschritts der Varianten-Generierung ist die geeignete Berücksichtigung unterscheidbarer Zustände der beteiligten Komponenten. Wie in Abschnitt 3.4 erläutert, können diese bei Beschreibung eines CUC angegeben werden, um die Voraussetzung für die Ausführung einer Interaktion genauer zu spezifizieren.

```
Component: casa.storage.Repository
    Offered Interfaces:  casa.storage.Repository
    Required Interfaces: casa.data.DNASequence
    Signature:           casa.storage.Repository {
                             ...
                             void setExternalDB(java.net.URL db);
                             java.util.Set searchByName(String name);
                             ...
                         }
    States:               <Default>, <DBSelected>
    Initial State:        <Default>
    Provided Use Cases:

        Initialise 1 Database
             Interactions: Standard from <Default>, <DBSelected> to
                           <DBSelected>
                               Self.setExternalDB(1 Database)
        Search * DNA
             Interactions: Standard from <DBSelected> to <DBSelected>
                               * DNA = Self.searchByName(Name)
```

Abb. 3.29: *Alternative CUC-Beschreibung zur Komponente* `Repository`

Beispielsweise könnte die Funktionalität der Komponente `Repository` prinzipiell auch durch die in Abbildung 3.29 dargestellte CUC-Beschreibung modelliert werden. Im Gegensatz zu der in Abbildung 3.24 gezeigten Beschreibung unterscheidet sie zwischen Initialisierung und eigentlicher Suche nach Sequenzen als eigenständige Anwendungsfälle. Die für ihre Interaktionen angegebenen Zustandsübergänge stellen sicher, daß vor Ausführung des Anwendungsfalls Search 1..* DNA mindestens einmal die Initialisierung durchgeführt wird. Diese Modellierung erleichtert letztlich die Benutzung der Komponente, da nach einmaliger Initialisierung wiederholt in der betreffenden Datenbank gesucht werden kann.

Bei Konstruktion der Prototyp-Varianten erfordert diese Form der Beschreibung eine Verfolgung des von außen beobachtbaren Zustands der betreffenden Komponente, die idealerweise zur Laufzeit der Variante erfolgt, wie in Abschnitt 5.3 diskutiert wird. Im Rahmen der Referenz-Implementierung wird jedoch ein vereinfachtes Verfahren gewählt, daß evtl. vorausgesetzte Zustandsübergänge bereits zum Zeitpunkt der Generierung berücksichtigt.

Hierbei wird der implizierte, endliche Zustandsautomat nachgebildet und bei Inanspruchnahme einer Interaktion entsprechend des eindeutig spezifizierten Übergangs simuliert. Falls sich die betreffende Komponente nicht im vorausgesetzten Zustand befindet, so wird eine endliche Folge von Interaktionen gesucht, die sie in den erforderlichen Zustand überführt. Aufgrund des vorgegebenen, initialen Zustands kann eine solche Folge durch einen effizienten Algorithmus gefunden werden. Anschließend werden die so gefundenen Interaktionen *vor* der ursprünglichen Interaktion in den Code integriert.

Abb. 3.30: *Zustandsdiagramm zur Komponente* `Repository`

Abbildung 3.30 verdeutlicht dieses Verfahren an Hand des Zustandsdiagramms zur Komponente `Repository`, das sich aus der in Abbildung 3.29 gezeigten CUC-Beschreibung ableiten läßt. Hierbei setzt sich der Name einer Transition aus der Bezeichnung des Anwendungsfalls sowie dem Namen der zugehörigen Interaktion zusammen, da ja im allgemeinen verschiedene Interaktionen mit unterschiedlichen Zustandsübergängen für den gleichen Anwendungsfall angegeben werden können. Falls zunächst nur die Interaktion zum Anwendungsfall Search 1..* DNA ausgewählt wird, so findet das oben beschriebene Verfahren die zuvor notwendige Interaktion zum Anwendungsfall Initialize 1 Database, um die Komponente `Repository` aus dem Initialzustand `<Default>` in den erforderlichen Zustand `<DBSelected>` zu überführen.

Im Anwendungsbeispiel unterscheidet sich der später generierte Code tatsächlich nicht vom zuletzt gezeigten Ausschnitt, weshalb auf eine entsprechende Darstellung verzichtet werden kann. Falls die betrachtete Funktionale Aussage allerdings erneut Bezug auf den Anwendungsfall Search 1..* DNA nimmt, oder eine Erweiterung der Funktionalen Spezifikation auch Iterationen zuläßt (vgl. Abschnitt 5.2.2), so wird in der Folge eine mehrfache Initialisierung der Komponente verhindert.

Teilschritt 4

In diesem Teilschritt der Varianten-Generierung werden die in den vorangegangenen Schritten verknüpften Instanzen auf technische Kompatibilität geprüft und ggf. über den Einsatz entsprechender Adapter vermittelt. Schließlich sollte sich der später generierte Code zumindest fehlerfrei übersetzen und ausführen lassen. Hierfür werden alle Zuweisungen, die sich aus den zuvor eingebundenen Interaktionen ergeben, den Regeln des verwendeten Typsystems unterworfen. Die eingesetzte technische Plattform sollte diese Informationen zweckmäßigerweise zum Zeitpunkt der Generierung bereitstellen, wie dies etwa im Fall der Java-Plattform durch den sog. Reflection-Mechanismus ermöglicht wird.

Falls keine technische Kompatibilität gewährleistet ist, wird zunächst eine alternative Interaktion des gleichen CUC gesucht, deren Umsetzung eine technisch kompatible Verknüpfung erlaubt. Andernfalls ist ein geeigneter Adapter für die betreffende Repräsentation zu ermitteln oder zu generieren, wie in Abschnitt 3.6.3 ausführlich erläutert wird. Anschließend wird eine entsprechende Instanz des Adapter erstellt und mit einer festgelegten Repräsentation des vermittelten Konzepts initialisiert. Die Auswahl der jeweils geeigneten Repräsentation ergibt sich hierbei eindeutig aus der Richtung der Zuweisung, d.h. die vor der Verknüpfung bereits existierende Repräsentation dient als Ausgangspunkt für den internen Abgleich des Zustands.

Im oben aufgeführten Anwendungsbeispiel der Variante `SequenceTool_1` (vgl. Abbildung 3.27) erfordert die zugehörige Interaktion des Anwendungsfalls Display 1 Alignment den Einsatz eines solchen Adapters, falls die in Abbildung 3.31 dargestellte CUC-Beschreibung zur ausgewählten funktionalen Komponente `PhylipView` angenommen wird. Die Repräsentation des primären Konzepts `alignment` ist im Kontext der betrachteten Variante auf die Komponente `AlignmentImpl` festgelegt, welche jedoch neben `Alignment` nicht zusätzlich die Schnittstelle `PhylipAlignment` implementiert (vgl. Abbildung 3.20). Somit ist die vorliegende Repräsentation technisch gesehen mit dem entsprechenden Parameter `a` der Operation `setAlignment` aus der Schnittstelle `PhylipView` nicht kompatibel. Daher muß ein geeigneter Adapter vor Ausführung der Interaktion erstellt und initialisiert werden, wie der nachfolgende Ausschnitt aus dem später generierten Code verdeutlicht.

```
class SequenceTool_1 {
  AlignmentImpl alignment = new AlignmentImpl();
  PhylipView phylipView = new PhylipView();
  ...
```

```
Component: other.PhylipView
    Offered Interfaces:  other.PhylipView
    Required Interfaces: other.PhylipAlignment, other.PhylipPhylogeny
    Signature:           other.PhylipView {
                             ...
                             void setAlignment(other.PhylipAlignment a);
                             void setPhylogeny(other.PhylipPhylogeny p);
                             ...
                         }
    States:               <Default>
    Initial State:        <Default>
    Provided Use Cases:
        Display 1 Alignment
            Interactions: Standard from <Default> to <Default>
                              Self.setAlignment(1 Alignment)
        Display 1 Phylogeny
            Interactions: Standard from <Default> to <Default>
                              Self.setPhylogeny(1 Phylogeny)
```

Abb. 3.31: *CUC-Beschreibung zur Komponente* ***`PhylipView`***

```
  void multiple_DNA_Selected() {
    // Create and initialise necessary adapter
    AlignmentAdapter alignmentAdapter = new AlignmentAdapter();
    alignmentAdapter.setAlignmentImpl(alignment);
    // Interaction for "Display 'alignment'"
    phylipView.setAlignment(alignmentAdapter);
    // Further interactions within functional statement
    ...
  }
}
```

Die Implementierung der Methode `multiple_DNA_Selected` beinhaltet die in diesem Teilschritt neu erstellten Anteile. Vor Ausführung der Interaktion für den Anwendungsfall `Display 'alignment'` wird der entsprechende Adapter `AlignmentAdapter` erstellt und über dessen wohlbekannte Operation `setAlignmentImpl` mit der existierenden Repräsentation `alignment` initialisiert. Der hiermit verbundene Abgleich des Zustands zwischen den beteiligten Repräsentationen unterliegt der alleinigen Verantwortung des Ad-

apters. Anschließend kann der betreffende Adapter im folgenden Aufruf der Operation `setAlignment` als Parameter übergeben werden, da er aufgrund seiner Konstruktion ebenfalls die Schnittstelle `PhylipAlignment` implementiert (vgl. Abbildung 3.25).

Eine weitere grundlegende Aufgabe dieses Teilschritts der Varianten-Generierung ist der Abgleich unterschiedlicher Kardinalität sowie die Vermittlung zwischen bekannten, diesbezüglichen Implementierungen der verwendeten technischen Plattform. So stellt beispielsweise Java eine ganze Reihe verschiedener Schnittstellen wie `java.util.Set`, `java.util.List` oder `java.util.Vector` bereit, deren Implementierung jeweils eine Menge von Konzepten repräsentiert. In diesem Fall bietet es sich an, eine entsprechende Hilfskomponente einzuführen, die bei Bedarf zwischen diesen häufig eingesetzten Schnittstellen vermittelt. Im Rahmen der Referenz-Implementierung geschieht dies durch die oben bereits erwähnte Komponente `Collection`. Ihre Anwendung wird somit durch einige der zuvor aufgeführten Beispiele ausreichend erläutert.

Teilschritt 5

Schließlich kann im letzten Teilschritt der Varianten-Generierung aus dem fertiggestellten Konstruktionsplan einer Variante Quellcode generiert werden, der sich übersetzen und ausführen läßt. Naturgemäß beinhaltet dieser Teilschritt zahlreiche technische Details und Besonderheiten, wie etwa die Behandlung von Ausdrücken und Laufzeit-Fehlern (engl. *exception*), die maßgeblich durch die Vorgaben der eingesetzten technischen Plattform bestimmt werden. Aus diesem Grund sei neben den vorangegangenen Code-Fragmenten zur weiteren Erläuterung auf die im Rahmen dieser Arbeit erstellte Java-Implementierung verwiesen (siehe Kapitel 4). Dennoch lassen sich die so erzielten Ergebnisse prinzipiell auch auf andere technische Plattformen übertragen, weil die oben beschriebenen Modelle und Verfahren nicht grundsätzlich an eine bestimmte Infrastruktur gebunden sind.

Darüber hinaus ist die Umsetzung in Quellcode durch die geleisteten Vorarbeiten der bisherigen Teilschritte mit verhältnismäßig geringem Aufwand zu realisieren. Neben der oben verfolgten Strategie, Strukturen der Funktionalen Spezifikation möglichst übersichtlich in den Quellcode zu übernehmen, sollte zusätzlich ein geeignetes Rahmenprogramm für jede Variante erstellt werden. Es erlaubt später die einfache Ausführung der jeweiligen Prototyp-Variante, beispielsweise innerhalb einer integrierten Entwicklungsumgebung. Zuletzt sollte eine grafische Benutzeroberfläche generiert werden, die visuelle Komponenten geeignet darstellt und entsprechende Bestandteile, wie etwa Menüs oder Schaltflächen, für angegebene Auslöser der Funktionalen Spe-

zifikation vorsieht. Diese Teilaufgabe läßt sich vorteilhaft mit bestehenden Ansätzen und Werkzeugen zur Erstellung von GUI-Prototypen lösen, wie in Abschnitt 5.3 diskutiert wird.

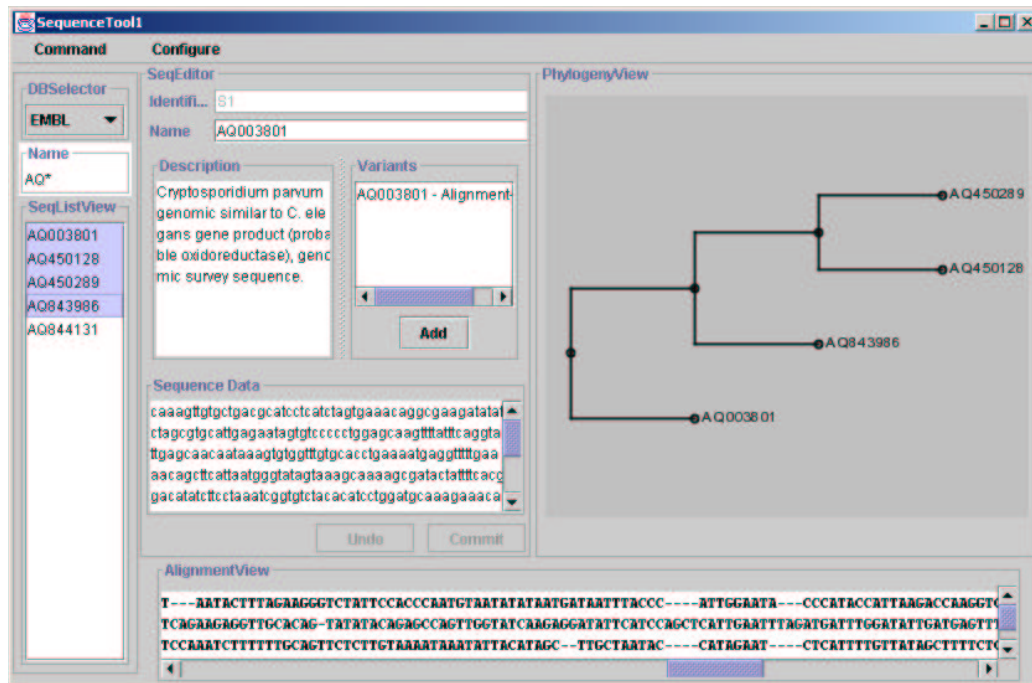


Abb. 3.32: *Bildschirmdarstellung der Variante* `SequenceTool_1`

Abbildung 3.32 zeigt die Bildschirmdarstellung der so generierten und übersetzten Prototyp-Variante `SequenceTool_1` zur Laufzeit[11]. Die visuellen Komponenten wie `SeqListView`, `SeqEditor` oder `AlignmentView` sind durch entsprechend beschriftete Rahmen gekennzeichnet. Darüber hinaus sind im Menü `Command` Einträge zur Auslösung der vom Benutzer kontrollierten Funktionalen Aussagen zusammengefaßt (vgl. Abbildung 3.19). Demgegenüber beinhaltet das Menü `Configure` Einträge zur Konfiguration ausgewählter Komponenten, falls diese entsprechende Funktionalität über ausgezeichnete Anwendungsfälle bereitstellen. Auf diese Weise kann beispielsweise die grafische Darstellung der Phylogenie nach den Bedürfnissen des Benutzers angepaßt werden.

Weiterhin ist in Abbildung 3.32 der dynamische Ablauf der funktionalen Prototyp-Variante angedeutet. Nach Angabe der externen Datenbank `EMBL`

[11] Das Layout der Bildschirmdarstellung wurde aus Gründen der Übersichtlichkeit manuell optimiert.

über die links oben dargestellte Komponente `DBSelector`, wurde über das darunter liegende Eingabefeld `Name` nach DNA-Sequenzen mit dem Präfix `AQ` gesucht. Die Anzeige der gefundenen Sequenzen durch die ebenfalls links angeordnete Komponente `SeqListView` erlaubt zusätzlich die Auswahl einer Teilmenge des Ergebnisses durch eine entsprechende Interaktion mit dem Benutzer. Im Beispiel führt die Selektion der ersten vier Sequenzen zu einer Berechnung ihres Alignments, das durch die unten angeordnete Komponente `AlignmentView` dargestellt wird. Darüber hinaus kann die zuerst ausgewählte Sequenz durch Interaktion mit der in der Mitte befindlichen Komponente `SeqEditor` bearbeitet werden. Schließlich wird aus dem Sequenz-Alignment eine mögliche Verwandtschaftsbeziehung berechnet, welche durch die rechts oben angeordnete Komponente `PhylipView` grafisch repräsentiert ist.

Somit entspricht die Funktionalität der betrachteten Prototyp-Variante im wesentlichen den in Abschnitt 3.6.1 aufgeführten, ursprünglichen funktionalen Anforderungen an das zu erstellende System. Inwieweit diese Anforderungen allerdings tatsächlich erfüllt sind oder welche Komponenten in das spätere Produkt übernommen werden, kann letztlich nur der Entwickler in Zusammenarbeit mit dem späteren Anwender an Hand des generierten Prototypen entscheiden. Möglicherweise ist eine andere, unterschiedlich zusammengesetzte Prototyp-Variante hinsichtlich der gestellten Anforderungen besser geeignet. Zuletzt kann die ursprüngliche Funktionale Spezifikation entsprechend den neu gewonnenen Erkenntnissen geändert werden. Dies führt zu einer weiteren Iteration des in Abbildung 3.17 gezeigten, übergeordneten Prozesses.

Zusammenfassend läßt sich feststellen, daß die oben eingeführten Modelle und Verfahren unter bestimmten Voraussetzungen eine effektive und weitgehend automatisierte Generierung ausführbarer Prototyp-Varianten ermöglichen. Die hierfür erforderlichen Daten und eingesetzten Algorithmen sind im Hinblick auf die Generierung *einer* ausgewählten Prototyp-Variante beherrschbar, wie das oben aufgeführte Beispiel verdeutlicht. Tatsächlich führt die Vielzahl an zumindest potentiell geeigneten Komponenten zu einer exponentiell wachsenden Anzahl unterschiedlich zusammengesetzter Varianten (vgl. Gleichung 3.5), die unmöglich in ihrer Gesamtheit untersucht werden können. Die sich hieraus ergebenden Fragestellungen und Lösungsansätze zur Ermittlung bestmöglicher Kombinationen sind Gegenstand des folgenden Abschnitts.

3.6.5 Varianten-Optimierung

Die in Abschnitt 3.1 zusammengefaßten Anforderungen an ein Framework für komponentenbasiertes Rapid Prototyping führen unmittelbar zu einer sehr grundlegenden Problemstellung. Einerseits sollten möglichst viele verschiedene Prototyp-Varianten generiert werden, da die endgültigen funktionalen Anforderungen an das spätere System sowie die hierfür geeigneten Software-Komponenten zu Beginn nicht festgelegt sind. Andererseits führt die Berücksichtigung zahlreicher verschiedener Komponenten zu einer exponentiell steigenden Anzahl zu betrachtender Prototyp-Varianten, wodurch die Skalierbarkeit des gewählten Ansatzes gefährdet ist.

Auch wenn Konstruktion, Generierung und Bewertung einer einzelnen Variante mit vertretbarem Aufwand zu bewerkstelligen ist, wie die vorangegangenen Abschnitte zeigen, so ist eine derartige Behandlung *aller* möglichen Varianten bei einer realistischen Problemgröße nicht zu erreichen. Aus diesem Grund muß eine geeignete Auswahl einer Teilmenge aller Varianten erfolgen, deren Umfang eine beherrschbare Anwendung des gesamten Verfahrens ermöglicht. Die Grundlagen für eine solche Auswahl und deren weitere Optimierung, also die vorausgesetzten Annahmen, verwendeten Regeln und Zielgrößen, werden durch eine entsprechende Heuristik bestimmt.

Obwohl prinzipiell verschiedene Heuristiken zur Verfügung stehen, schlägt der vorgestellte Ansatz hierfür die Anwendung eines *Genetischen Algorithmus* vor [Gol89]. Dies ergibt sich im wesentlichen aus der Natur des zugrundeliegenden Problems. So kann bereits der Austausch einer einzigen Komponente innerhalb einer gegebenen Variante zu einem tatsächlich ungeeigneten Prototyp führen. In diesem Fall ergibt sich zunächst kein Anhaltspunkt, in welche Richtung die betreffende Variante weiterentwickelt oder verändert werden muß, um die gestellten Anforderungen besser zu erfüllen. Aus diesem Grund versprechen etwa gradientenbasierte Verfahren wie das sog. *Hill-Climbing* [Win89] wenig Aussicht auf Erfolg.

Hingegen sind zufallsbasierte Verfahren, wie eben ein Genetischer Algorithmus, für diesen Problemtyp seht gut geeignet, da sie keine Stetigkeit der zu optimierenden Zielgröße voraussetzen und der Fortschritt der erzielten Verbesserungen nicht durch lokale Optima behindert wird bzw. verschiedene lokale Optima als gleichberechtigte Lösungen zugelassen sind [Gol89]. Darüber hinaus kann diese Heuristik über verschiedene Parameter sehr genau an die vorliegende Problemstellung sowie die verfügbaren Ressourcen angepaßt werden. Diese Flexibilität ist von großer Bedeutung, sobald umfangreiche praktische Erfahrungen mit dem vorgestellten Ansatz vorliegen. Ein letzter, eher technisch motivierter Vorteil liegt der vereinfachten Behandlung möglicher Alternativen bei der in Abschnitt 3.6.4 erläuterten Varianten-

Generierung. So können verschiedene mögliche Adapter oder Interaktionen weitgehend zufällig und ohne besondere Strategie ausgewählt werden, sofern eine entsprechende Bewertung in die übergeordnete Optimierung einbezogen wird.

Wie bereits in Abschnitt 3.2 angesprochen, betrachtet ein Genetischer Algorithmus eine festgelegte Anzahl an Individuen einer gesamten Population. Jedes Individuum besitzt einen eindeutigen Genotyp, der eine mögliche Lösung des gestellten Problems repräsentiert. Aufgrund der Qualität dieser Lösung kann jedem solchen Individuum bzw. dessen Genotyp eine definierte Fitneß zugeordnet werden, die eine Selektion der besten Individuen erlaubt. Sie bilden die Grundlage einer nächsten Generation, die über Reproduktion sowie zufällige Veränderungen durch Mutation und Rekombination bestehender Individuen ergänzt wird. Darüber hinaus können neue, ebenfalls zufällig gebildete Individuen der Population hinzugefügt werden, um die Diversifikation der Population zu erhalten oder zu steigern. Die fortgesetzte Anwendung dieser Schritte beschreibt einen evolutionären Prozeß, an dessen Ende keine signifikante Verbesserung der durchschnittlichen Fitneß innerhalb der Population erzielt werden kann.

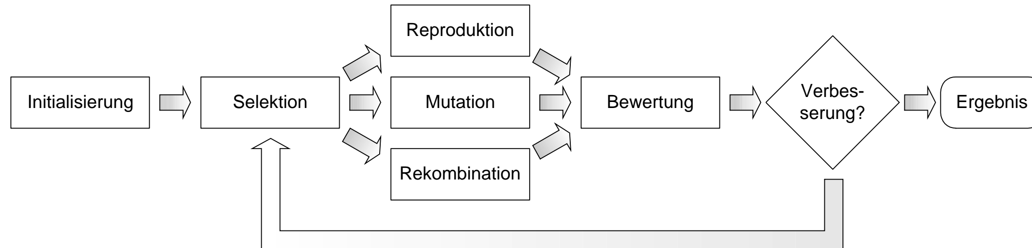


Abb. 3.33: *Schematischer Ablauf eines Genetischen Algorithmus*

Abbildung 3.33 illustriert diesen prinzipiellen Ablauf eines Genetischen Algorithmus an Hand eines schematischen Flußdiagramms. Während der Initialisierung wird eine festgelegte Anzahl an Individuen zufällig erstellt und für die erste Generation der Population ausgewählt. Durch Anwendung der Genetischen Operatoren Reproduktion, Rekombination und Mutation entstehen neue Individuen bis hin zu einer festgelegten Obergrenze für die gesamte Population. Anschließend wird eine Bewertung jedes Individuums hinsichtlich seiner Fitneß durchgeführt sowie die durchschnittliche Fitneß der besten Individuen mit der vorherigen Generation verglichen. Wird in diesem Schritt eine Verbesserung erzielt oder ist eine vorgegebene, maximale Anzahl an Generationen noch nicht erreicht, so führt eine Selektion der besten Individuen zu einer neuen Generation. Andernfalls terminiert das Verfahren mit der

zuletzt betrachteten Generation. Es ist zu beachten, daß Reproduktion, Rekombination und Mutation als stochastische Operation, d.h. mit einer gewissen Wahrscheinlichkeit, für ein gegebenes Individuum durchgeführt werden. Hierbei ist typischerweise die Wahrscheinlichkeit einer Mutation fest vorgegeben, während Reproduktion und Rekombination bei Individuen mit größerer Fitneß auch mit entsprechend höherer Wahrscheinlichkeit ausgeführt werden. Dies entspricht einer grundlegenden Annahme der eingesetzten Heuristik, daß die Kombination guter Lösungen in der Regel zu noch besseren Lösungen führt.

Die Anwendung des so zusammengefaßten Verfahrens zur heuristischen Optimierung auf die vorliegende Problemstellung im Rahmen des übergeordneten Frameworks erfordert im wesentlichen vier grundlegende Schritte:

- Festlegung des Zusammenhangs zwischen einer Prototyp-Variante und dem im Verlauf des Genetischen Algorithmus betrachteten Genotyp eines Individuums der Population.

- Definition der Genetischen Operatoren Reproduktion, Mutation und Rekombination auf Basis der oben getroffenen Festlegung.

- Angabe einer geeigneten Fitneß-Funktion $f(x)$, die zur Bewertung und Selektion eines gegebenen Individuums x herangezogen werden kann. Sie bestimmt die Richtung der Evolution und somit das Ziel der durchgeführten Optimierung.

- Festlegung der Basisparameter des Genetischen Algorithmus, d.h. Größe der Population, Anteil der durch Selektion berücksichtigten Individuen, Grenzwert für die Verbesserung der durchschnittlichen Fitneß sowie die Wahrscheinlichkeit für Mutation, Reproduktion und Rekombination eines gegebenen Individuums in Abhängigkeit seiner Fitneß.

Das zunächst erforderliche Verständnis einer Prototyp-Variante als Individuum einer Population ergibt sich intuitiv aus der jeweiligen Kombination an ausgewählten Komponenten für Anwendungsfälle der Funktionalen Spezifikation. Diese in Abschnitt 3.6.2 beschriebene Auswahl führt zu eindeutig bestimmten Varianten, die sich letztlich gerade in ihrer Zusammensetzung aus verschiedenen Komponenten unterscheiden. Somit können einzelne Anwendungsfälle der Spezifikation im Kontext des Genetischen Algorithmus als *Gene* aufgefaßt werden, deren Gesamtheit auch als *Genom* bezeichnet wird. In diesem Zusammenhang entspricht eine durch die Funktionale Spezifikation vorgegebene Struktur aus Anwendungsfällen und Funktionalen Aussagen einer Unterteilung des Genoms in unterschiedliche *Chromosomen*. Diese Organisation des Genoms kann bei Definition des Genetischen Operators

Rekombination ausgenützt werden, wie später erläutert wird. Die konkrete Ausprägung eines betrachteten Gens wird durch die jeweils ausgewählte Komponente einer gegebenen Variante festgelegt. In der Terminologie des Genetischen Algorithmus werden diese verschiedenen möglichen Ausprägungen eines Gens im folgenden auch als *Allele* bezeichnet. Die Gesamtheit aller bestimmten Allele legt somit den *Genotyp* eines Individuums eindeutig fest.

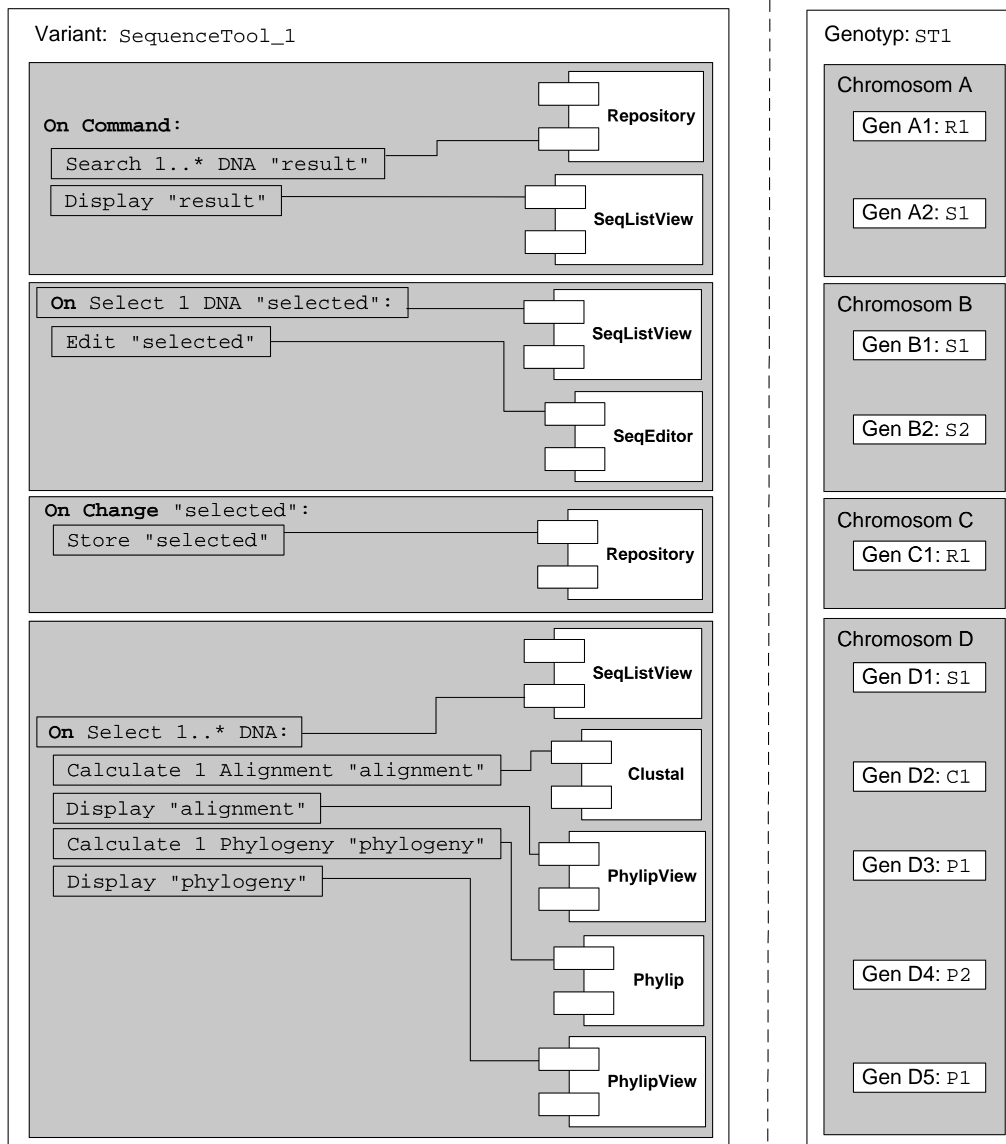


Abb. 3.34: *Genotyp der Prototyp-Variante* `SequenceTool_1`

Abbildung 3.34 illustriert die so beschriebene Abbildung einer Prototyp-Variante auf den Genotyp eines Individuums der Population. Auf der linken Seite ist die für Variante `SequenceTool_1` des Anwendungsbeispiels getroffene Auswahl an funktionalen Komponenten dargestellt (vgl. Abbildung 3.27), während die rechte Seite den hieraus resultierenden Genotyp zeigt[12]. Die Funktionalen Aussagen der Spezifikation entsprechen den Chromosomen A bis D, wobei die untergeordneten Anwendungsfälle als entsprechend bezeichnete Gene A1 bis D4 aufgefaßt werden. Die jeweils für einen gegebenen Anwendungsfall ausgewählte Komponente entspricht einem Allel des betreffenden Gens, das durch eine abgekürzte Bezeichnung identifiziert wird. Die Gesamtheit aller so festgelegten Allele bestimmt den Genotyp `ST1` des betrachteten Individuums. Er ist somit unter allen anderen möglichen Varianten des Anwendungsbeispiels eindeutig festgelegt.

Genetische Operatoren

Auf Basis dieser einfachen Abbildung von Prototyp-Varianten auf Genotypen der Population können nunmehr die Genetischen Operatoren Reproduktion, Mutation und Rekombination geeignet definiert werden. Hierbei wird die Reproduktion als einfache Duplikation des betreffenden Genotyps festgelegt, d.h. die Anwendung dieses unären Operators auf ein Individuum liefert eine exakte Kopie dieses Individuums hinsichtlich seiner Zusammensetzung aus verschiedenen Allelen. Somit entspricht die Reproduktion im wesentlichen einer asexuellen Fortpflanzung, wie sie in der Biologie typischerweise bei Prokaryonten zu beobachten ist.

Hingegen verändert die Mutation ein zufällig ausgewähltes Gen des betreffenden Individuums. Hierbei wird in der Regel das zugehörige Allel durch eines des anderen möglichen Allele für dieses Gen ersetzt. Darüber hinaus kann der Genetische Operator Mutation für die betrachtete Problemstellung erweitert werden, so daß nicht zwangsläufig das bestehende Allel ersetzt wird, sondern ggf. nur eine vorhandene, alternative Interaktion des entsprechenden CUC ausgewählt wird (vgl. Abschnitt 3.4 und 3.6.4). Die Einführung einer solchen lokalen Mutation ist hilfreich, um Interaktionen zu berücksichtigen, deren erforderliche Rollen und Parameter im Verlauf der Generierung möglicherweise besser zugeordnet werden können. Allerdings ist hierfür bei Realisierung des Genetischen Operators ein entsprechender Bezug auf den ursprünglichen Konstruktionsplan der jeweiligen Prototyp-Variante erforderlich. Außerdem ist die verwendete Fitneß-Funktion geeignet anzupassen, wie

[12] Aus Gründen der Übersichtlichkeit wurde auf die Darstellung der ausgewählten Repräsentationen verzichtet, auch wenn diese offensichtlich ebenfalls als Teil des Genotyps aufzufassen sind.

später erläutert wird. In jedem Fall liefert die Anwendung des Operators Mutation ein neues Individuum mit modifiziertem Genotyp.

Der binäre Genetische Operator Rekombination entspricht im wesentlichen der sexuellen Fortpflanzung zweier Individuen, wie sie in der Biologie üblicherweise bei höher entwickelten Eukaryonten zu beobachten ist. In Anlehnung an das biologische Vorbild werden die Chromosomen der beteiligten Individuen gegeneinander ausgerichtet und für jedes Chromosom eine Position zwischen zwei benachbarten Genen zufällig bestimmt. Anschließend werden die ober- bzw. unterhalb dieser Position befindlichen Allele zwischen den beteiligten Individuen wechselseitig ausgetauscht. Durch diese Kreuzung (engl. *cross-over*) entstehen zwei neue Individuen, deren Genotyp eine jeweils unterschiedliche Kombination der beiden ursprünglichen Individuen repräsentiert.

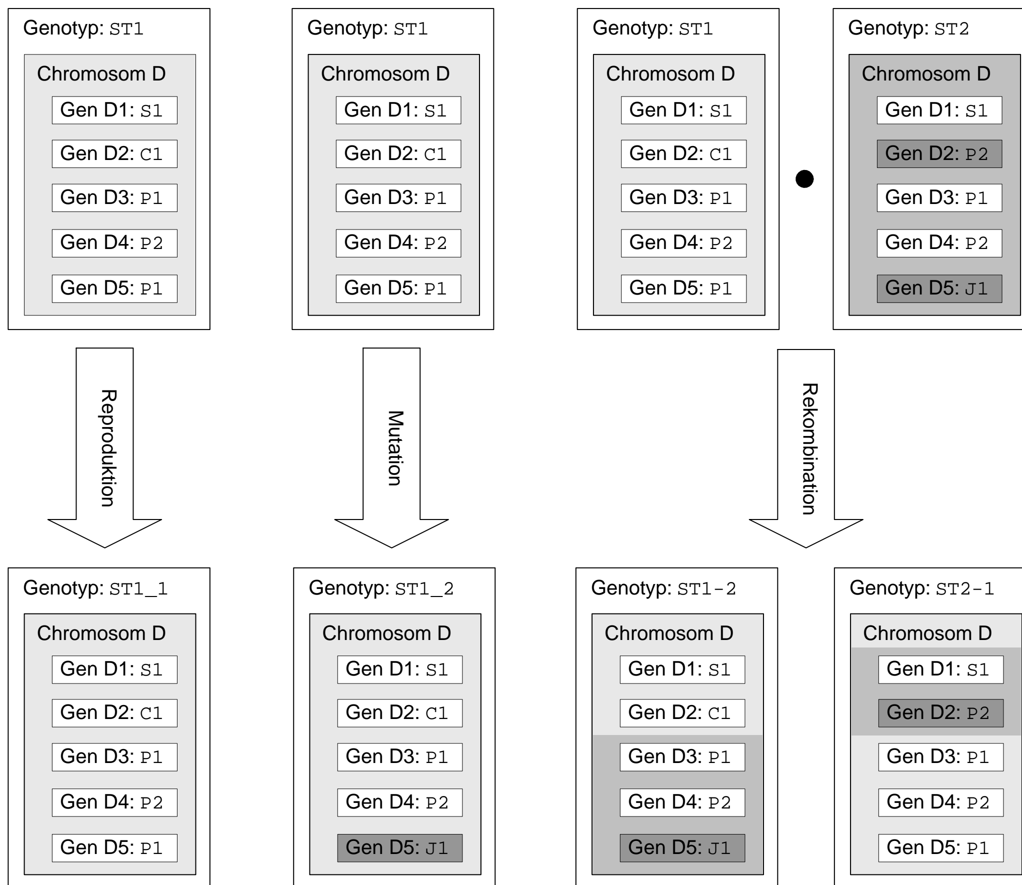


Abb. 3.35: *Beispielhafte Anwendung der Genetischen Operatoren*

Abbildung 3.35 verdeutlicht die Anwendung der so definierten Genetischen Operatoren an einem Ausschnitt des Anwendungsbeispiels. Hierbei wird aus Gründen der Übersichtlichkeit nur das Chromosom D des in Abbildung 3.34 eingeführten Genotyps ST1 betrachtet. Die oben beschriebene Reproduktion liefert eine identische Kopie des betreffenden Genotyps, der im Beispiel zur besseren Unterscheidung mit ST1_1 bezeichnet ist. Tatsächlich befinden sich nach Anwendung des Operators *zwei verschiedene* Individuen mit dem *gleichen* Genotyp ST1 in der Population. Hingegen verändert die dargestellte Mutation im Beispiel das dunkel hervorgehobene Gen D5. Anstelle des im Genotyp ST1 befindlichen Allels P1 findet sich im resultierenden Genotyp ST1_2 das Allel J1. In diesem Fall wird also bei der entsprechenden Prototyp-Variante die Komponente PhylipView durch die an dieser Stelle ebenfalls mögliche Komponente JTree ersetzt (vgl. Abbildung 3.22).

Die in Abbildung 3.35 auf der rechten Seite dargestellte Rekombination führt zu einer Kreuzung der Genotypen ST1 und ST2. Hierbei unterscheidet sich der Genotyp ST2 in den dunkel hervorgehobenen Genen D2 und D5 durch Verwendung der Allele P2 bzw. J1. Die zugehörige Prototyp-Variante benutzt also die Komponente Phylip zur Erfüllung des Anwendungsfalls Calculate 1 Alignment sowie die Komponente JTree zur Anzeige der später berechneten Phylogenie (vgl. Abbildung 3.22 und 3.34). Die im Beispiel durch einen Punkt bezeichnete, zufällig bestimmte Position des Cross-Overs für Chromosom D führt nach Anwendung der Rekombination zu den beiden neuen Genotypen ST1-2 und ST2-1. Im Genotyp ST1-2 ergeben sich die Allele für die oberhalb der Cross-Over Position befindlichen Gene D1 und D2 aus den entsprechenden Allelen des ursprünglichen Genotyps ST1, während sich die unterhalb befindlichen Gene D3, D4 und D5 aus dem Genotyp ST2 ableiten. Für den resultierenden Genotyp ST2-1 gelten offensichtlich inverse Verhältnisse.

Die auf diese Weise festgelegten Genetischen Operatoren erlauben die Generierung neuer Genotypen durch einfache, weitgehend zufällige Veränderungen bestehender Individuen. Während Mutation jedoch mit gleicher Wahrscheinlichkeit auf jedes Individuum angewandt wird, entscheidet bei Reproduktion und Rekombination die jeweilige Fitneß über die Anwendung des Genetischen Operators. Dies entspricht der grundlegenden Annahme der eingesetzten Heuristik, daß der Genotyp „erfolgreicher“ Individuen überdurchschnittlich häufig in der betrachteten Population vertreten ist, während die Kombination solcher Individuen mit einer höheren Wahrscheinlichkeit zu noch besseren Genotypen führt. Für den Erfolg dieser Heuristik ist allerdings die Definition einer geeigneten Fitneß-Funktion von entscheidender Bedeutung, wie im folgenden erläutert wird.

Fitneß-Funktion

Die im Verlauf eines Genetischen Algorithmus eingesetzte Fitneß-Funktion repräsentiert eine Bewertung jedes Individuums der Population hinsichtlich ausgewählter Kriterien der untersuchten Problemstellung. Sie erlaubt einen quantitativen Vergleich aller Individuen, der letztlich über die oben beschriebenen Teilschritte der Heuristik, wie Selektion, Reproduktion und Rekombination, die Richtung der Optimierung bestimmt. Aus diesem Grund bezieht sich die Bestimmung des Fitneß-Werts eines Individuums im wesentlichen auf dessen ursprüngliche Interpretation im Anwendungsbereich, also im vorliegenden Fall auf generierte Prototyp-Varianten bzw. deren Konstruktionspläne (vgl. Abschnitt 3.6.4). Auch wenn hierbei prinzipiell beliebig komplexe Zusammenhänge ausgewertet werden können, so ist doch auf eine effiziente Realisierung der Fitneß-Funktion zu achten. Schließlich muß diese im Verlauf des Verfahrens für jedes Individuum der Population in jeder Generation neu berechnet werden.

Unter diesen Voraussetzungen schlägt die vorliegende Arbeit eine entsprechend geeignete, aus Teilbewertungen zusammengesetzte Fitneß-Funktion vor, die wesentliche Merkmale der generierten Prototyp-Varianten berücksichtigt. Die im folgenden definierten Bewertungsfunktionen betrachten logisch zusammengehörige Aspekte einer gegebenen Variante zunächst weitgehend isoliert, um anschließend über frei wählbare Gewichtungsfaktoren zu einer Gesamtbewertung beizutragen. Dieser Ansatz erlaubt eine Anpassung an verschiedene Anwendungsbereiche und unterschiedliche Vorgehensweisen bei der eigentlichen Optimierung. Nicht zuletzt bestimmen auch zukünftige praktische Erfahrungen über eine geeignete Wahl der Bewertungsfunktionen und Gewichtungsfaktoren, wie später in Abschnitt 3.7 diskutiert wird.

Abbildung 3.36 zeigt eine schematischen Übersicht der verwendeten Teilbewertungen, die durch abgerundete Rechtecke grafisch repräsentiert sind. Die Auswahl der getrennt bewerteten Bereiche einer Prototyp-Variante X folgt im wesentlichen den Ergebnissen der in Abschnitt 3.6.2, 3.6.3 und 3.6.4 beschriebenen, grundlegenden Schritten der Prototyp-Generierung. So wird die im Verlauf der Komponenten-Auswahl getroffene Festlegung auf Repräsentationen $\mathsf{R_1}$ bis $\mathsf{R_m}$ und funktionale Komponenten $\mathsf{C_1}$ bis $\mathsf{C_n}$ durch die Bewertungsfunktionen f_{tr} und f_{tc} beurteilt. Sie stützen sich auf den hierfür durchgeführten Abgleich mit den primären Konzepten $\mathsf{K_1}$ bis $\mathsf{K_m}$ und Anwendungsfällen $\mathsf{UC_1}$ bis $\mathsf{UC_n}$ der Funktionalen Spezifikation. Es ist zu beachten, daß eine gegebene funktionale Komponente der Variante X durchaus mehrere verschiedene Anwendungsfälle der Spezifikation erfüllen kann und daher mehrfach an den entsprechenden Positionen der Variante referenziert wird (vgl. Abbildung 3.34). Die Folgerung $i \neq j \Rightarrow \mathsf{C_i} \neq \mathsf{C_j}$ ist somit im allge-

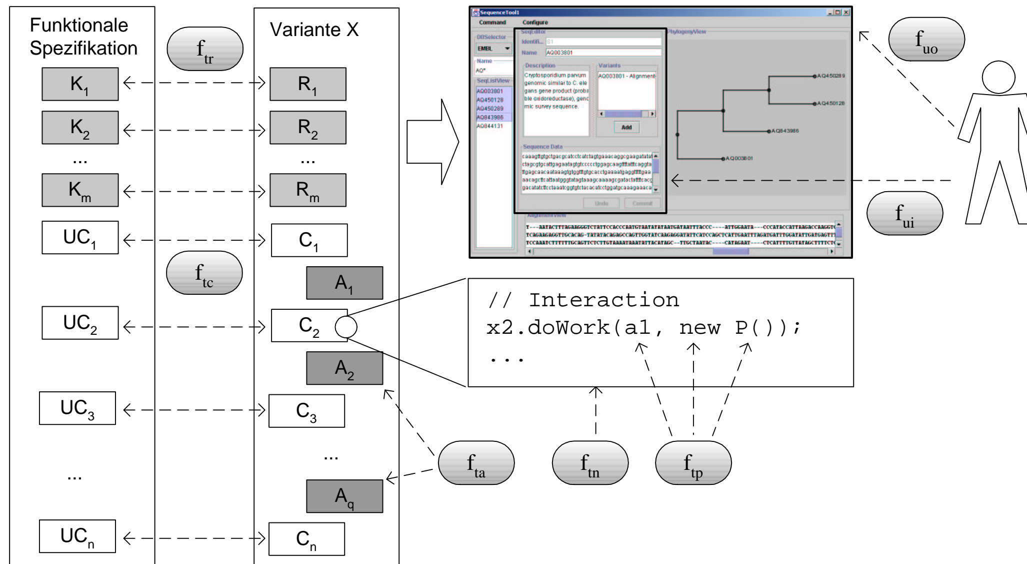


Abb. 3.36: *Schematische Übersicht der Varianten-Bewertung*

meinen *nicht* gültig. Eine entsprechende Aussage trifft auch auf festgelegte Repräsentationen zu, die ebenfalls möglicherweise mehrere primäre Konzepte der Spezifikation wiedergeben.

Die Anzahl und Qualität der zur Verknüpfung unterschiedlicher Komponenten ggf. erforderlichen Adapter A_1 bis A_q wird durch f_{ta} bewertet, während die im Verlauf der Varianten-Generierung bestimmten Interaktionen durch f_{tn} behandelt werden. Hierbei ist wegen der in Abschnitt 3.6.4 eingeführten Vereinfachungen eine durch f_{tp} repräsentierte Beschränkung auf die gewählten Belegungen für Parameter möglich. Im allgemeinen Fall sind allerdings auch andere Merkmale der betreffenden Interaktion, wie etwa die Zuordnung zwischen Instanzen und Rollen, zu berücksichtigen.

Neben den oben aufgeführten, automatisch durchzuführenden Teilbewertungen kann die Eignung einer generierten Prototyp-Variante hinsichtlich der gewünschten Funktionalität letztlich nur durch den Entwickler beurteilt werden. Diese Beurteilung erfolgt in der Regel interaktiv an Hand der ausführbaren Variante und findet als f_{uo} bzw. f_{ui} Eingang in die Gesamtbewertung. Hierbei entspricht f_{uo} einer Bewertung der gesamten betrachteten Variante, während sich f_{ui} auf einzelne, ausgewählte Komponenten bezieht. Beispielsweise kann eine gegebene Variante als insgesamt schlecht beurteilt werden, während einige ihrer Komponenten bestimmte Teilaufgaben sehr gut erfüllen. Obwohl diese Informationen den weiteren Verlauf der Optimierung maßgeblich beeinflussen, sollte der erforderliche manuelle Aufwand möglichst gering

gehalten werden. Hierfür werden später pragmatische Lösungsansätze vorgestellt.

Die in Abbildung 3.36 dargestellten Teilbewertungen werden im folgenden genauer definiert und schrittweise zu einer übergeordneten, für die Anwendung des Genetischen Algorithmus geeigneten Bewertungsfunktion kombiniert. Die so bestimmte Fitneß-Funktion f ordnet jedem Individuum $\vec{x}$ der Population einen Wert $f(\vec{x}) \in [0,1]$ zu. Hierbei entspricht ein Wert am unteren Ende des Intervalls einer niedrigen Fitneß des betrachteten Individuums, während ein Wert am oberen Ende des Intervalls eine hohe Fitneß repräsentiert. Diese besondere Interpretation des Wertebereichs $[0,1]$ einer Bewertungsfunktion wird auch in den folgenden Definitionen dieses Abschnitts angenommen.

Die gewählte Notation symbolisiert den sequentiellen Aufbau der zu $\vec{x}$ gehörigen Prototyp-Variante hinsichtlich ihrer Zusammensetzung aus der Menge ihrer Repräsentationen R_x, Adapter A_x und funktionalen Komponenten C_x (vgl. Abbildung 3.36). Dementsprechend werden die jeweiligen Teilsequenzen in $\vec{x}$ im folgenden mit $\vec{r}$, $\vec{a}$ und $\vec{c}$ bezeichnet, während r_i, a_i bzw. c_i das i-te Element einer solchen Sequenz repräsentiert[13]. Wie oben bereits erwähnt, kann eine solche Sequenz im allgemeinen das gleiche Element an verschiedenen Positionen beinhalten. Ihre Länge, also die Anzahl der enthaltenen Elemente, wird mit $|\vec{r}|$, $|\vec{a}|$ bzw. $|\vec{c}|$ bezeichnet. Hierbei kann ohne Beschränkung der Allgemeinheit angenommen werden, daß $|\vec{r}|$ und $|\vec{c}|$ der Anzahl an unterschiedlichen primären Konzepten bzw. Anwendungsfällen der Funktionalen Spezifikation entspricht. Dies kann beispielsweise durch die Einführung von nicht-funktionalen „Platzhalter-Komponenten“ erreicht werden, die entsprechende Positionen der Variante einnehmen, falls keine geeignete Komponenten-Auswahl getroffen werden konnte.

Eine durch die übergeordnete Fitneß-Funktion f modellierte Eignung einer Prototyp-Variante wird durch zwei grundlegende Anteile bestimmt. Einerseits können zahlreiche Merkmale der betrachteten Variante, wie etwa Anzahl der benötigten Komponenten und Adapter, automatisch bewertet werden. Andererseits obliegt das maßgebliche Urteil letztlich dem Entwickler, der ggf. in Zusammenarbeit mit dem Anwender entscheidet, inwieweit der generierte Prototyp oder einzelne seiner Komponenten die gestellten funktionalen Anforderungen tatsächlich erfüllt. Daher wird $f(\vec{x})$ aus den Teilbewertungen $f_u(\vec{x}), f_t(\vec{x}) \in [0,1]$ ermittelt, welche die manuelle bzw. automatische Bewertung der Variante $\vec{x}$ repräsentieren. Somit gilt

$$f(\vec{x}) = w_u \cdot f_u(\vec{x}) + w_t \cdot f_t(\vec{x}) \quad \text{mit } w_u + w_t = 1 \tag{3.6}$$

[13] Aus Gründen der Übersichtlichkeit wird hierbei ein impliziter Bezug auf die jeweils betrachtete Variante x vorausgesetzt.

wobei die weitgehend frei wählbaren Gewichtungsfaktoren $w_u, w_t \in [0,1]$ die *zugemessene Bedeutung* der jeweiligen Teilbewertungen wiedergeben. Später wird erläutert, wie sich diese und weitere Gewichtungsfaktoren zur Reduzierung des erforderlichen manuellen Aufwands bei der Optimierung einsetzen lassen.

Die vom Benutzer vorgenommene Bewertung f_u bezieht sich, wie aus Abbildung 3.36 ersichtlich, sowohl auf die gesamte Prototyp-Variante als auch auf einzelne ihrer Komponenten. Aus diesem Grund entspricht f_u der Kombination aus einer übergreifenden Bewertungsfunktion $f_{uo}(\vec{x}) \in [0,1]$ sowie einer normierten Bewertung $f_{ui}(c_i) \in [0,1]$ der funktionalen Komponenten c_i[14]:

$$f_u(\vec{x}) = w_{uo} \cdot f_{uo}(\vec{x}) + w_{ui} \cdot \frac{\sum f_{ui}(c_i)}{|\vec{c}|} \quad \text{mit } w_{uo} + w_{ui} = 1 \tag{3.7}$$

Hierbei legen die frei wählbaren Gewichtungsfaktoren $w_{uo}, w_{ui} \in [0,1]$ wiederum die zugemessene Bedeutung der Teilbewertungen fest. Hingegen werden die tatsächlichen Werte der Funktionen $f_{uo}, f_{ui} \in [0,1]$ in der Regel durch eine geeignete Interaktion mit dem Benutzer ermittelt, wie später in Kapitel 4 an Hand der Referenz-Implementierung beispielhaft erläutert wird. Aus pragmatischen Gründen ist es daher sinnvoll, diese interaktive Bewertung nur für einen beherrschbaren Ausschnitt der gesamten Population durchzuführen und ansonsten einen festen Wert $f_u(\vec{x}) = k_u \in [0,1]$ anzunehmen. Dieser wird zu Beginn vorgegeben, etwa mit $k_u = 0.5$, und im weiteren Verlauf der Optimierung an die durchschnittlichen Bewertungen des Benutzers angepaßt, wie später beschrieben wird.

Eine derartige Beschränkung auf einen Ausschnitt der Population betrifft die automatisch durchgeführte Bewertung f_t offensichtlich nicht. Sie kann unmittelbar nach den logisch zusammengehörigen Aspekten der betrachteten Varianten untergliedert werden (vgl. Abbildung 3.36). Entsprechend den in Abschnitt 3.6.2, 3.6.3 und 3.6.4 beschriebenen, grundlegenden Aufgaben und Lösungsansätzen zur Komponenten-Auswahl, Vermittlung zwischen unterschiedlichen Repräsentationen sowie Generierung ausführbarer Prototyp-Varianten wird $f_t(\vec{x})$ definiert als

$$\begin{aligned} f_t(\vec{x}) = w_{tc} \cdot f_{tc}(\vec{x}) + w_{tr} \cdot f_{tr}(\vec{x}) + w_{ta} \cdot f_{ta}(\vec{x}) + w_{tn} \cdot f_{tn}(\vec{x}) \\ \text{mit } w_{tc} + w_{tr} + w_{ta} + w_{tn} = 1 \wedge w_{tc}, w_{tr}, w_{ta}, w_{tn} \in [0,1] \end{aligned} \tag{3.8}$$

In diesem Zusammenhang beschreibt $f_{tc}(\vec{x}) \in [0,1]$ die Bewertung der für die Variante $\vec{x}$ getroffenen Auswahl an funktionalen Komponenten. Sie ergibt

[14] Die eingesetzten Repräsentationen r_i sind für den Benutzer nicht unmittelbar ersichtlich und werden daher in der vorgeschlagenen Bewertung nicht berücksichtigt.

sich zunächst unmittelbar aus der logischen Kompatibilität zwischen Anwendungsfällen der Funktionalen Spezifikation und komponentenbezogenen Anwendungsfällen der ausgewählten Komponenten (vgl. Abschnitt 3.6.2).

Hierfür kann die gemäß Gleichung 3.1 bestimmte, semantische Kompatibilität der manipulierten Konzepte herangezogen werden. Darüber hinaus ist es möglich, die insgesamt vorhandene Anzahl $|C_x|$ an *unterschiedlichen* funktionalen Komponenten zu berücksichtigen. Varianten mit geringer Anzahl $|C_x|$ werden besser bewertet, da in diesem Fall eine homogene technische Zusammensetzung und somit vereinfachte Generierung angenommen werden kann. Dieser Sachverhalt führt zu einer Definition von $f_{tc}(\vec{x})$ als

$$f_{tc}(\vec{x}) = w_{sc} \cdot \frac{\sum s_c(c_i)}{|\vec{c}|} + w_{nc} \cdot \frac{|\vec{c}| - |\mathsf{C_x}|}{|\vec{c}|} \quad \text{mit } w_{sc} + w_{nc} = 1 \wedge w_{sc}, w_{nc} \in [0,1] \tag{3.9}$$

Hierbei bezeichnet $s_c(c_i) \in [0,1]$ eine einfache, aus Gründen der Übersichtlichkeit nicht näher definierte Hilfsfunktion, die für eine referenzierte funktionale Komponente c_i den zugehörigen CUC bestimmt, die Richtung der Substitution manipulierter Konzepte ermittelt und nach Gleichung 3.2, 3.3 bzw. 3.4 den Grad ihrer semantischen Kompatibilität berechnet[15]. Das im zweiten Term der Gleichung 3.9 berechnete Verhältnis zwischen tatsächlich benötigten und maximal erforderlichen funktionalen Komponenten kann für eine genauere Bewertung auch durch einen Vergleich mit der durchschnittlich benötigten Anzahl an funktionalen Komponenten ersetzt werden. Die vorgestellte Lösung bevorzugt eine lokale Betrachtung der gegebenen Variante, um nicht zuvor den Durchschnitt *aller* Varianten der Population berechnen zu müssen.

In vergleichbarer Weise wird die Bewertungsfunktion f_r für Repräsentationen festgelegt durch

$$f_{tr}(\vec{x}) = w_{sr} \cdot \frac{\sum s_r(r_i)}{|\vec{r}|} + w_{nr} \cdot \frac{|\vec{r}| - |\mathsf{R_x}|}{|\vec{r}|} \quad \text{mit } w_{sr} + w_{nr} = 1 \wedge w_{sr}, w_{nr} \in [0,1] \tag{3.10}$$

Wiederum bezeichnet $s_r(r_i) \in [0,1]$ eine geeignete Hilfsfunktion, welche die semantische Kompatibilität der repräsentierten Konzepte nach Gleichung 3.2, 3.3 oder 3.4 berechnet. In diesem Fall ist die Richtung der Substitution offensichtlich eindeutig vorgegeben, d.h. das in der Funktionalen Spezifikation referenzierte Konzept wird grundsätzlich durch das von r_i repräsentierte Konzept ersetzt. Der zweite Term in Gleichung 3.10 bevorzugt Varianten mit einer geringen Anzahl an unterschiedlichen Repräsentationen für primäre

[15] Aus Gründen der besseren Effizienz kann dieser Wert offensichtlich bereits bei der Komponenten-Auswahl berechnet und für spätere Schritte bereitgehalten werden.

Konzepte der Spezifikation. Dieser Anteil an $f_{tr}(\vec{x})$ ist in der Regel höher zu gewichten als der entsprechende Term in Gleichung 3.9, da die Funktionale Spezifikation mit hoher Wahrscheinlichkeit unterschiedliche Referenzen auf das gleiche Konzept beinhaltet (vgl. Abbildung 3.19). In diesem Fall führt die Auswahl gleicher Repräsentationen zu insgesamt einfacher strukturierten Varianten, die voraussichtlich eine geringere Anzahl an Adapter benötigen.

Dennoch ist im allgemeinen die Verwendung unterschiedlicher Adapter zwischen den verschiedenen erwarteten Repräsentationen primärer Konzepte unvermeidlich (siehe Abschnitt 3.6.3). Daher bewertet die Funktion f_{ta} aus Gleichung 3.8 diese Anteile einer Variante nach Qualität und Anzahl der erforderlichen Adapter[16]. Unter Berücksichtigung der Tatsache, daß sich die *maximale* Anzahl benötigter Adapter a_{max} in der Regel aus der Anzahl unterschiedlicher funktionaler Komponenten $|C_x|$ abschätzen läßt, kann $f_{ta}(\vec{x})$ in Anlehnung an Gleichung 3.9 und 3.10 definiert werden als

$$f_{ta}(\vec{x}) = w_{qa} \cdot \frac{\sum f_a(a_i)}{|\vec{a}|} + w_{na} \cdot \frac{a_{max} - |\vec{a}|}{a_{max}}$$
$$\text{mit } a_{max} \approx |C_x| - 1, |\vec{a}| > 0, w_{qa} + w_{na} = 1 \wedge w_{qa}, w_{na} \in [0,1] \qquad (3.11)$$

Die Funktion $f_a(a_i) \in [0,1]$ liefert ein Maß für die geschätzte Qualität eines gegebenen Adapters. Aus Gründen der vereinfachten Berechnung können hierfür vorgegebene Werte $k_{au}, k_{af}, k_{ap} \in [0,1]$ je nach Herkunft des betreffenden Adapters zugeordnet werden, d.h.

$$f_a(a_i) = \begin{cases} k_{au} & \text{falls } a_i \text{ durch den Benutzer erstellt} \\ k_{af} & \text{falls } a_i \text{ vollständig generiert} \\ k_{ap} & \text{falls } a_i \text{ partiell generiert} \end{cases} \qquad (3.12)$$

In der Regel gilt $k_{au} >> k_{af} >> k_{ap}$, also beispielsweise $k_{au} = 1$, $k_{af} = 0.5$ sowie $k_{ap} = 0$, da vom Benutzer erstellte und explizit bereitgestellte Adapter erwartungsgemäß eine höhere Qualität aufweisen als automatisch generierte Adapter. Demgegenüber sind partiell generierte Adapter nicht funktionstüchtig und müssen erst in einem weiteren Schritt durch den Benutzer implementiert werden (siehe Abschnitt 3.6.3). Für eine genauere Einschätzung der Qualität können zur Festlegung von f_a bei Bedarf allerdings auch weitere Merkmale der verwendeten Adapter, wie Anzahl und semantische Kompatibilität der in Beziehung gesetzten Konzepte, herangezogen werden. Trotzdem sind insgesamt Varianten zu bevorzugen, die mit einer möglichst geringen Anzahl an erforderlichen Adaptern konstruiert werden können. Dementsprechend führt der zweite Term in Gleichung 3.11 zu einer besseren Bewertung dieser Varianten.

[16] Sollten keine Adapter benötigt werden, also $|\vec{a}| = 0$, so kann offensichtlich ein fester Wert $f_{ta}(\vec{x}) = 1$ angenommen werden.

Gemäß Gleichung 3.8 ist zuletzt auch die technische Umsetzung der ausgewählten Interaktionen zu berücksichtigen. Schließlich erfordert die Festlegung auf eine bestimmte Interaktion des CUC einer funktionalen Komponente im allgemeinen Fall die konkrete Belegung benötigter Parameter, Zuordnung von Instanzen zu Rollen oder Verarbeitung der anfallenden Zwischenergebnisse, wie in Abschnitt 3.6.4 ausführlich erläutert wird. Die entsprechende Bewertungsfunktion $f_{tn}(\vec{x}) \in [0,1]$ beurteilt eine gegebene Variante $\vec{x}$ hinsichtlich der so beschriebenen Umsetzung ihrer ausgewählten Interaktionen (vgl. Abbildung 3.36). Die in Kapitel 4 vorgestellte Referenz-Implementierung beschränkt sich jedoch auf einfache Interaktionen mit vordefinierten Rollen, die im Verlauf der Varianten-Generierung problemlos zugeordnet werden können (vgl. Abschnitt 3.6.4). Daher stützt sich f_{tn} ausschließlich auf eine Bewertung $f_{tp}(c_i) \in [0,1]$ der ermittelten Belegungen für Parameter. Diese bestimmen letztlich die Wahrscheinlichkeit einer erfolgreichen Ausführung der für eine funktionale Komponente c_i ausgewählten Interaktion. Somit kann $f_{tn}(\vec{x})$ definiert werden als

$$f_{tn}(\vec{x}) = \frac{\sum f_{tp}(c_i)}{|\vec{c}|} \quad \text{mit} \quad f_{tp}(c_i) = \frac{\sum\limits_j f_p(p_{ij})}{|\vec{p_i}|} \in [0,1], |\vec{p_i}| > 0 \tag{3.13}$$

Die hierfür zusätzlich eingeführte Bewertungsfunktion f_p beurteilt jede Parameterbelegung p_{ij} aus der Sequenz aller Belegungen $\vec{p_i}$ der ausgewählten Interaktion für eine fest vorgegebene Komponente c_i[17]. Diese Beurteilung ergibt sich letztlich aus der Herkunft des Wertes für die betrachtete Parameterbelegung, die ja nach unterschiedlichen Strategien ermittelt werden kann, wie in Abschnitt 3.6.4 beschrieben ist. Daher wird $f_p(p_j) \in [0,1]$ einer gegebenen Parameterbelegung p_j definiert als

$$f_p(p_j) = \begin{cases} k_{pe} & \text{falls } p_j \text{ mittels existierender Interaktion belegt} \\ k_{pb} & \text{falls } p_j \text{ durch neue Instanz eines Basistyps belegt} \\ k_{pa} & \text{falls } p_j \text{ mittels zusätzlicher Interaktion belegt} \\ k_{pd} & \text{falls } p_j \text{ durch neu erstellte Instanz belegt} \end{cases} \tag{3.14}$$

Die festgelegten Konstanten $k_{pe}, k_{pb}, k_{pa}, k_{pd} \in [0,1]$ repräsentieren die eingeschätzte Qualität der ermittelten Belegung in Hinblick auf eine erfolgreiche Ausführung der zugehörigen Interaktion. So deutet die Zuordnung von Zwischenergebnissen aus einer existierenden, möglicherweise der gleichen zu c_i gehörenden Interaktion auf eine wahrscheinlich geeignete, kontextbezogene Parameterbelegung hin, während für eine neu benötigte Instanz eines Basistyps der technischen Infrastruktur immerhin noch einfache Möglichkeiten

[17] Falls keine Parameterbelegungen erforderlich sind, also $|\vec{p_i}| = 0$, kann offensichtlich ein fester Wert $f_{tp}(c_i) = 1$ angenommen werden.

zu dessen Eingabe generiert werden können (vgl. Abschnitt 3.6.4). Demgegenüber verursacht eine zusätzlich eingeführte Komponente, die über besonders ausgezeichnete Manipulationen wie `Select` oder `Initialize` zur Initialisierung der erforderlichen Parameterbelegung herangezogen wird, möglicherweise neue Probleme bei anschließender Generierung und Ausführung der Prototyp-Variante. Schließlich führt die zuletzt in Betracht gezogene Erstellung einer Instanz des erforderlichen Typs, ohne sie anschließend geeignet zu initialisieren, mit hoher Wahrscheinlichkeit zu Fehlern beim späteren Ablauf der Interaktion. Dementsprechend sind die Werte der in Gleichung 3.14 verwendeten Konstanten so zu wählen, daß $k_{pe} > k_{pb} >> k_{pa} >> k_{pd}$ erfüllt ist, beispielsweise durch $k_{pe} = 1$, $k_{pb} = 0.8$, $k_{pa} = 0.4$ sowie $k_{pd} = 0$.

Die durch Gleichung 3.6 bis 3.14 eindeutig definierte Fitneß-Funktion f des Genetischen Algorithmus erlaubt somit eine rasche, weitgehend unkomplizierte Einschätzung der Qualität einer gegebenen Prototyp-Variante $\vec{x}$ hinsichtlich ihrer wesentlichen Merkmale. Die eingeführten Teilbewertungen beziehen sich ausschließlich auf die jeweils betrachtete Variante und sind daher, mit Ausnahme der vom Benutzer durchgeführten Beurteilung, effizient auch für eine große Anzahl an Prototyp-Varianten zu berechnen. Hierbei verfolgt der vorgestellte Ansatz das Ziel, die betrachteten Merkmale isoliert zu beurteilen und über geeignet gewählte Gewichtungsfaktoren w_i zu einer übergeordneten Gesamtbewertung zu verknüpfen. Dies erlaubt einerseits die Anpassung der Bewertung an unterschiedliche Anwendungsbereiche und praktische Erfahrungen, kann andererseits aber auch für pragmatische Modifikationen des dynamischen Ablaufs der Optimierung eingesetzt werden, wie später erläutert wird.

Offensichtlich sind die mittels f_t automatisch ermittelten Anteile der Fitneß-Funktion kein absolutes Maß für die erreichte Qualität einer Prototyp-Variante, d.h. auch tatsächlich ungeeigneten Varianten kann ein vergleichsweise hoher Wert für f_t zugeordnet werden. Im Rahmen des übergeordneten Frameworks und dessen eigentlicher Zielsetzung ist eine genauere und somit aufwendigere Berechnung allerdings nicht zwingend erforderlich oder möglicherweise sogar hinderlich. Schließlich sind die exakten funktionalen Anforderungen sowie die eigentlich erwünschten Komponenten des vollständigen Systems zu Beginn des Verfahren nicht bekannt. Deshalb sollte grundsätzlich der Benutzer die Tauglichkeit der initialen, automatisch generierten Vorschläge an Hand der ausführbaren Varianten abschließend beurteilen. Diese Informationen bestimmen maßgeblich den weiteren Verlauf der Optimierung, wie im folgenden beschrieben wird.

Parameter und dynamischer Ablauf

Der über $f(\vec{x})$ ermittelte Fitneß-Wert einer Variante $\vec{x}$ ermöglicht den quantitativen Vergleich mit anderen Individuen der Population. Dieser bildet die Grundlage für weitere Elemente der eingesetzten Heuristik, wie Selektion, Reproduktion oder Rekombination ausgewählter Individuen. Hierfür sind in einem letzten Schritt die grundlegenden Parameter des Genetischen Algorithmus zu bestimmen und dessen dynamischer Ablauf zu regeln. Die an dieser Stelle vorgeschlagenen Werte beruhen auf ersten praktischen Erfahrungen mit der Referenz-Implementierung im Rahmen des in dieser Arbeit untersuchten Anwendungsbereichs. Es ist daher zu erwarten, daß weitergehende Erfahrungen mit anderen Anwendungsbereichen und realistischen Problemgrößen zu insgesamt besser geeigneten Vorschlägen führen. Aufgrund der im folgenden näher vorgestellten Parameter lassen sich diese Anpassungen einfach in das übergeordnete Framework integrieren.

Dies betrifft zunächst die Größe der Population N_P, d.h. die Anzahl der in jeder Generation betrachteten Individuen. Die geeignete Wahl von N_P ist abhängig von den verfügbaren Ressourcen sowie der Anzahl an prinzipiell möglichen, unterschiedlich zusammengesetzten Prototyp-Varianten. Hierbei wird der insgesamt erforderliche Rechenaufwand und Speicherplatzbedarf für eine gegebene Variante zwar im allgemeinen stark von der Komplexität der Funktionalen Spezifikation bestimmt, jedoch sind alle Schritte des in Abschnitt 3.6.2, 3.6.3, 3.6.4 und 3.6.5 beschriebenen Verfahrens so gewählt, daß der Aufwand im wesentlichen linear mit der Anzahl an unterschiedlichen Anwendungsfällen bzw. der Anzahl an ausgewählten Komponenten wächst. Somit können auch realistische Problemgrößen und umfangreiche Populationen betrachtet werden, insbesondere weil die Teilschritte des Genetischen Algorithmus weitgehend lokal für einzelne Varianten durchgeführt werden. Deshalb ist eine Parallelisierung der Implementierung und anschließende Verteilung der Last auf verschiedene Rechner ohne Schwierigkeiten erreichbar.

Somit richtet sich die Wahl von N_P im wesentlichen nach der Anzahl an unterschiedlichen Prototyp-Varianten N_V, die nach erfolgter Komponenten-Auswahl gemäß Gleichung 3.5 ermittelt wird (vgl. Abschnitt 3.6.2). Für eine ausreichende Überdeckung des so dimensionierten Lösungsraums sollten in der Regel mindestens 10% aller möglichen Varianten betrachtet werden, d.h. $N_P \gtrsim 0.1 \cdot N_V$. Dies kann für große Werte von N_V durch eine entsprechend stringente Komponenten-Auswahl erreicht werden. In diesem Fall wird die geforderte semantische Kompatibilität der manipulierten Konzepte beim Abgleich der Anwendungsfälle entsprechend erhöht (vgl. Abschnitt 3.3). Hierbei ist durch Interaktion mit dem Benutzer auch eine Differenzierung nach einzelnen Anwendungsfällen der Funktionalen Spezifikation möglich, d.h. dieje-

nigen Anwendungsfälle, deren Abgleich zu einer besonders hohen Anzahl an zugeordneten Komponenten führt, werden mit einem entsprechend höheren Schwellwert ϵ_k erneut abgeglichen, so daß letztlich nur eine beherrschbare Menge an möglichen Komponenten zu betrachten ist.

Ein weiterer maßgeblicher Parameters des Genetischen Algorithmus ist der Anteil der durch Selektion berücksichtigten Individuen $r_s \in [0, 1]$ an der gesamten Population N_P. Er bestimmt, welche Individuen aus der letzten Generation übernommen und durch Anwendung der Genetischen Operatoren zur Bildung einer neuen Generation herangezogen werden (siehe Abbildung 3.33). Hierbei können je nach genetischer Zusammensetzung der Population oder gewünschter Variation in der Zusammensetzung generierter Prototyp-Varianten unterschiedliche Werte für r_s gewählt werden. In der Regel kann zunächst $r_s \in [0.1, 0.2]$ angenommen werden, d.h. zwischen 10 und 20% der nach ihrer Fitneß besten Individuen werden in die nächste Generation übernommen. Falls sich jedoch herausstellt, daß auf diese Weise ein auffällig hoher Anteil an tatsächlich ungeeigneten oder sogar fehlerhaften Varianten in der Population verbleibt, kann eine stärkere Selektion erfolgen, indem ein entsprechend kleinerer Wert für r_s gewählt wird.

In anderen Fällen erscheint es sinnvoll, die Vielfalt an möglichen Lösungen zu erhöhen und daher eine weniger strenge Selektion durchzuführen, also einen entsprechend höheren Wert für r_s zu wählen. Diese Maßnahme erhöht indirekt die Vielzahl an unterschiedlichen Genotypen in der Population, die trotz einer zunächst geringeren Fitneß möglicherweise interessante Varianten repräsentieren. Aus diesem Grund kann ebenfalls eine einfache Modifikation des ursprünglichen Genetischen Algorithmus eingeführt werden, bei der im Zuge der Selektion neben den besten Individuen auch eine gewisse Anzahl zufällig ausgewählter Individuen ungeachtet ihrer Fitneß aus der bestehenden Population in die nächste Generation übernommen wird. Dies gewährleistet eine grundlegende Diversifikation der Population und verringert somit die Gefahr, daß der evolutionäre Prozeß frühzeitig mit einem nicht erwünschten, lokalen Optimum endet.

Nach der Selektion führt, wie oben bereits erwähnt, die wiederholte Anwendung der Genetischen Operatoren Mutation, Reproduktion und Rekombination zu neuen oder veränderten Varianten, bis die ursprüngliche Größe der Population wieder erreicht ist. Hierbei wird jede selektierte Variante $\vec{x}$ mit einer Wahrscheinlichkeit $p_o(\vec{x}) \in [0, 1]$ und $p_r(\vec{x}) \in [0, 1]$ für Anwendung von Reproduktion bzw. Rekombination ausgewählt. Diese Wahrscheinlichkeit ist in Anlehnung an das biologische Vorbild direkt proportional zu dem für $\vec{x}$ ermittelten Fitneß-Wert, d.h. $p_o(\vec{x}) = f(\vec{x})$ und $p_r(\vec{x}) = f(\vec{x})$. Das Verhältnis zwischen durch Reproduktion oder Rekombination erhaltenen, neuen Varianten kann frei gewählt werden, wobei in der Regel die Rekombination deutlich

zu bevorzugen ist, da ihre Anwendung tatsächlich neue Genotypen hervorbringt[18].

Nachdem die neue Generation mittels Reproduktion und Rekombination den ursprünglichen Umfang erreicht hat, wird mit einer festen Wahrscheinlichkeit $p_m \in [0,1]$ jedes Individuum einer Mutation ausgesetzt. Diese modifiziert den Genotyp des betroffenen Individuums auf zufällige Weise und führt so zu veränderten oder sogar gänzlich neuen Allelen, die bisher nicht in der Population vertreten waren. Allerdings kann hierdurch möglicherweise auch eine Verringerung der Fitneß eintreten, gerade bei Individuen, die zuvor bereits verhältnismäßig gut bewertet wurden. Daher ist die vorgegebene Mutationsrate in der Regel klein, etwa $p_m = 0.05$, auch wenn diese zwischenzeitlich erhöht werden kann, um die Diversifikation der Population zu steigern.

Die wiederholte Anwendung von Selektion, Mutation, Reproduktion und Rekombination führt zu einer Folge verschiedener Generationen, deren Individuen im allgemeinen eine kontinuierlich steigende Fitneß aufweisen. Das Kriterium für ein Ende der Optimierung ist die vorgegebene untere Grenze ϵ_f für die erzielte Verbesserung der durchschnittlichen Fitneß. Hierfür werden zweckmäßigerweise nur die durch Selektion ausgewählten Individuen berücksichtigt, um starke Schwankungen des ermittelten Durchschnittswerts zu vermeiden. Die Anzahl N_S der so ausgewählten Individuen ist bei fester Populationsgröße N_P und Anteil der Selektion r_s offensichtlich gegeben durch $N_S = \lfloor r_s \cdot N_P \rfloor$. Der Vergleich zweier aufeinander folgender Generationen nach t bzw. $t+1$ Iterationen des Verfahrens führt also zu einem Abbruch des Genetischen Algorithmus falls

$$\frac{\bar{f}_{t+1} - \bar{f}_t}{\bar{f}_t} \leq \epsilon_f \quad \text{mit} \quad \bar{f} = \frac{\sum\limits_{i=1}^{N_S} f(\vec{x}_i)}{N_S} \tag{3.15}$$

wobei $\bar{f}_t$ und $\bar{f}_{t+1}$ die vorherige bzw. aktuelle durchschnittliche Fitneß der jeweils selektierten Individuen bezeichnen. Typischerweise wird ein Wert für ϵ_f zwischen 0.05 und 0.1 vorgegeben, obwohl das Verfahren auch jederzeit nach einer manuellen Bewertung durch den Benutzer abgebrochen werden kann. Erfahrungsgemäß schwanken die durchschnittlichen Fitneß-Werte zu Beginn der Optimierung verhältnismäßig stark, so daß zusätzlich eine untere Grenze für die Anzahl der mindestens durchzuführenden Iterationen angegeben werden sollte. Dies verringert die Gefahr einer vorzeitigen Terminierung, bevor sich die Änderung der durchschnittlichen Fitneß stabilisiert.

[18] Offensichtlich sind für die Rekombination Individuen auszuwählen, die einen unterschiedlichen Genotyp aufweisen.

Zuletzt sind die Gewichtungsfaktoren $w_i \in [0,1]$ der in Gleichung 3.6 bis 3.14 definierten Bewertungsfunktionen geeignet festzulegen, um die Bedeutung der jeweiligen Anteile an der Gesamtbewertung $f(\vec{x})$ zu bestimmen. Hierbei kann zunächst eine gleiche Bedeutung aller Merkmale angenommen werden, d.h. die Gewichtungsfaktoren werden so gewählt, daß ihr jeweils gleicher Wert in der Summe der zugehörigen Gleichung den Wert 1 ergibt. Falls anschließend im Verlauf der Optimierung jedoch besonders starke Schwankungen oder extreme Größen der Teilbewertungen auftreten, kann für weitere Optimierungen der entsprechende Gewichtungsfaktor verringert werden. Dies erhöht die Unempfindlichkeit des Verfahrens gegenüber möglicherweise nur unzuverlässig zu bewertenden Merkmalen einer Prototyp-Variante.

Umgekehrt erlaubt die relative Erhöhung eines bestimmten Gewichtungsfaktors die stärkere Berücksichtigung des jeweils bewerteten Merkmals im Rahmen der übergeordneten Optimierung. So kann beispielsweise in Gleichung 3.9 der Faktor w_{sc} deutlich größer als w_{nc} gewählt werden, um Varianten zu bevorzugen, deren funktionale Komponenten eine besonders hohe semantische Kompatibilität mit den Anwendungsfällen der Spezifikation aufweisen. Die absolute Anzahl der benötigten Komponenten sei in diesem Szenario dagegen von untergeordneter Bedeutung. In dieser Weise eingesetzt, ermöglicht die Festlegung der Gewichtungsfaktoren eine deklarative, den jeweiligen Gegebenheiten anpaßbare Steuerung der evolutionären Optimierung.

Neben der oben beschriebenen, statischen Festlegung der Gewichtungsfaktoren *vor* der eigentlichen Optimierung, ist es darüber hinaus möglich, diese auch *während* des Ablaufs, also dynamisch, zu verändern. Dies betrifft insbesondere w_u und w_t in Gleichung 3.6, welche die Bedeutung der manuellen bzw. automatisch durchgeführten Bewertung einer Variante bestimmen. Die vom Benutzer interaktiv vorzunehmende Beurteilung ist durchaus aufwendig und daher bei einer großen Population aus pragmatischen Gründen nicht für jedes Individuum zu ermitteln. Daher ist es sinnvoll, zunächst eine ausschließlich automatische Optimierung mit $w_u = 0, w_t = 1$ über einige Generationen hinweg durchzuführen. Anschließend wird für einen Teil der so optimierten Varianten übersetzbarer Code generiert, dieser ausgeführt und dem Entwickler zur interaktiven Bewertung vorgelegt. Die so erhaltenen Ergebnisse können im weiteren Verlauf besonders stark berücksichtigt werden, etwa durch eine Wahl der Gewichtungsfaktoren als $w_u = 0.8$ und $w_t = 0.2$. Hierbei ist für die tatsächlich nicht durch den Benutzer bewerteten Individuen $\vec{x}_k$ der interaktiv ermittelte Durchschnittswert als Ergebnis der Bewertung $f_u(\vec{x}_k)$ in Gleichung 3.6 anzunehmen, um die anschließende Selektion nicht ungewollt zu beeinflussen.

Bei Verwendung dynamisch veränderlicher Gewichtungsfaktoren sind die

absoluten Fitneß-Werte der Individuen offensichtlich nicht über alle Generationen hinweg vergleichbar. Dies ist für die eingesetzte Heuristik allerdings nicht zwingend erforderlich, weil die vorgeschlagene Umsetzung des Genetische Algorithmus nur vergleichbare *relative* Fitneß-Werte innerhalb der selben Generation voraussetzt, wie oben erläutert wird. Somit muß in diesem Fall lediglich die Bedingung für die Terminierung des Verfahrens geeignet angepaßt werden, etwa durch Vorgabe einer mindestens durchzuführenden Anzahl an Iterationen nach jeder Änderung eines Gewichtungsfaktors.

Auf diese Weise wird insgesamt eine flexible, den jeweiligen Gegebenheiten angepaßte Strategie der evolutionären Optimierung ermöglicht. So kann beispielsweise durch die Festlegung von $w_{tc} = 0.7, w_{tr} = 0.1, w_{ta} = 0.1, w_{tn} = 0.1$ zunächst in den ersten Generationen eine Optimierung nach bestmöglichen funktionalen Komponenten vorgenommen werden, um erst später im weiteren Verlauf mit $w_{tc} = 0.1, w_{tr} = 0.3, w_{ta} = 0.3, w_{tn} = 0.3$ die hieraus folgenden, eher technischen Anteile der Prototyp-Varianten angemessen zu bewerten (siehe Gleichung 3.8). Es ist zu erwarten, daß umfangreiche praktische Erfahrungen mit dem vorgestellten Ansatz zur Ermittlung besonders aussichtsreicher Strategien herangezogen werden können.

Wie bereits erwähnt, kann die oben beschriebene, dynamische Veränderung der Gewichtungsfaktoren eingesetzt werden, um den erforderlichen Aufwand bei manueller Bewertung der Prototyp-Varianten so gering wie möglich zu halten. Allerdings ist diese vom Benutzer vorgenommene Beurteilung maßgeblich für den Erfolg des in dieser Arbeit vorgeschlagenen, experimentellen Ansatzes für komponentenbasiertes Rapid Prototyping. Sie bestimmt letztlich die Tauglichkeit der generierten Varianten sowie ihrer Komponenten hinsichtlich der gegebenen funktionalen Anforderungen an das zu entwickelnde System. Diese entscheidende Information beeinflußt die Richtung der Optimierung auch bei hoher Gewichtung w_u lediglich indirekt und undifferenziert als Anteil des übergeordneten Fitneß-Werts.

Aus diesem Grund kann eine weitere Modifikation des dynamischen Ablaufs der Optimierung eingeführt werden, welche einen direkten und unmittelbaren Einfluß des Benutzers ermöglicht. Hierfür werden im Rahmen des Genetischen Algorithmus die Extremwerte der auf einzelne Komponenten c_i bezogenen Bewertung $f_{ui}(c_i) \in [0, 1]$ auf besondere Weise interpretiert (vgl. Gleichung 3.7). So wird ein Wert $f_{ui}(c_i) = 0$ als explizite Entscheidung des Benutzers aufgefaßt, die betreffende Komponente zukünftig nicht mehr in generierten Prototyp-Varianten zu berücksichtigen. Auf diese Weise können definitiv als ungeeignet erachtete Komponenten von der Betrachtung ausgeschlossen werden. Diese Entscheidung wird vermerkt und im weiteren Verlauf bei Komponenten-Auswahl sowie Anwendung des Genetischen Operators Mutation umgesetzt. Darüber hinaus werden alle Individuen der Population,

die ein entsprechendes Allel in ihrem Genotyp aufweisen, einer zufälligen Mutation an dieser Position ausgesetzt.

Umgekehrt repräsentiert ein Wert von $f_{ui}(c_i) = 1$ als bestmögliche Beurteilung die explizite Entscheidung des Benutzers, die betreffende Komponente grundsätzlich bei allen zukünftigen Prototyp-Varianten zu berücksichtigen. Wiederum beeinflußt diese Entscheidung die weitere Ausführung der Komponenten-Auswahl und Mutation, so daß ausschließlich die jeweilige Komponente gewählt und nicht mehr ersetzt wird. Zusätzlich wird bei allen Individuen mit unterschiedlichem Genotyp das zugehörige Allel entsprechend verändert.

Auf diese Weise ergibt sich insgesamt ein dynamischer Ablauf der Varianten-Optimierung, der dem Benutzer eine weitgehend freie Erkundung des Lösungsraums ermöglicht, ohne ihn jedoch mit einer übergroßen Anzahl möglicher Lösungen zu überfordern. Die getroffenen Bewertungen beurteilen ausgewählte Merkmale der Varianten, bestimmen die Selektion der besten Individuen und beeinflussen somit die grundsätzlich zufallsgesteuerte Optimierung der betrachteten Varianten, wobei durch Mutation und Rekombination auch gänzlich neue Genotypen entstehen. Sie entsprechen neuen Kombinationen aus funktionalen Komponenten und Repräsentationen, zu deren Verknüpfung möglicherweise andere Adapter und Interaktionen erforderlich sind. Aus diesem Grund sind für solche Varianten die in Abschnitt 3.6.3 und 3.6.4 erläuterten Teilschritte des übergeordneten Prozesses erneut durchzuführen (vgl. Abbildung 3.17). Anschließend führen Bewertung und Selektion zu einer nächsten Generation, deren durchschnittliche Fitneß in der Regel höher einzuschätzen ist. Diese iterative und evolutionäre Optimierung wird solange fortgesetzt, bis keine signifikante Verbesserung zu erreichen ist oder der Benutzer das Verfahren explizit beendet. Hierbei kann die eingesetzte Heuristik über zahlreiche Parameter und mögliche Strategien sehr flexibel an den jeweiligen Anwendungsbereich sowie die verfügbaren Ressourcen angepaßt werden.

Schematisches Anwendungsbeispiel

Um die Plausibilität dieses evolutionären Vorgehens zu motivieren, wird im folgenden ein exemplarischer Ablauf der Varianten-Optimierung genauer erläutert. Aus Gründen der Anschaulichkeit und Übersichtlichkeit wird hierbei ein überwiegend schematisches Anwendungsbeispiel betrachtet, das sich auf wesentliche Aspekte des oben beschriebenen Ansatzes beschränkt. Dementsprechend wird eine stark vereinfachte Bewertungsfunktion vorausgesetzt, die ausschließlich zur Beurteilung der technischen Anteile einer Variante herangezogen wird.

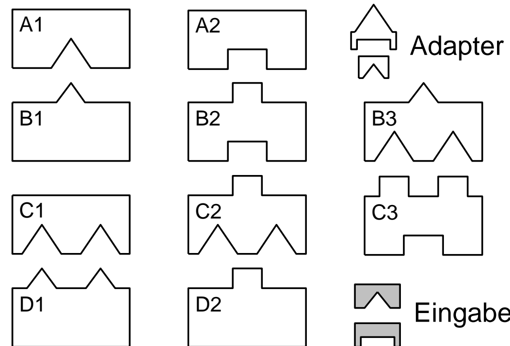


Abb. 3.37: *Komponenten des schematischen Anwendungsbeispiels*

Abbildung 3.37 zeigt die angenommenen Komponenten, also unterschiedliche Allele, des schematischen Anwendungsbeispiels für die vier Gene A bis D des einzigen betrachteten Chromosoms. Im Rahmen der gewählten grafischen Notation symbolisieren geometrische Merkmale an der Oberseite einer Komponente die für eine Interaktion benötigten Parameter, während entsprechend komplementäre Merkmale an deren Unterseite die ggf. zurückgelieferten Ergebnisse repräsentieren. Aufgrund der besonderen Form dieser Merkmale kann im Beispiel auf zwei unterschiedliche Repräsentationen für Parameter oder Ergebnisse geschlossen werden, zwischen denen zwei vorhandene, ebenfalls in Abbildung 3.37 dargestellte Adapter vermitteln. Darüber hinaus kann der Zustand beider Repräsentation durch geeignete Möglichkeiten zur Eingabe, etwa über einfache Initialisierung oder besondere Bestandteile der Benutzeroberfläche (vgl. Abschnitt 3.6.4), ermittelt werden. Diese weitgehend generische Belegung der erforderlichen Parameter ist in der Abbildung durch dunkel hinterlegte Symbole hervorgehoben.

Trotz dieser durchaus übersichtlichen Verhältnisse können im Beispiel offensichtlich bereits $2 \cdot 3 \cdot 3 \cdot 2 = 36$ unterschiedlich zusammengesetzte Varianten konstruiert werden. Für die im folgenden vorgestellte Optimierung ist jedoch nur die Betrachtung eines zufällig ausgewählten Ausschnitts erforderlich, der sich als Population über mehrere Generationen hinweg gemäß den Regeln und Vorgaben des Genetischen Algorithmus weiterentwickelt. Hierfür sei die Fitneß f eines gegebenen Individuums $\vec{x}$ vereinfacht angenommen als $f(\vec{x}) = 0.7^a \cdot 0.4^b$, wobei a und b die Anzahl der benötigten Adapter bzw. Eingaben bezeichnet. Die so festgelegte Gewichtung entspricht der Einschätzung, daß die Belegung eines Parameters über Vermittlung einer bereits existierenden Repräsentation grundsätzlich der manuellen Eingabe oder einfachen Initialisierung vorzuziehen ist.

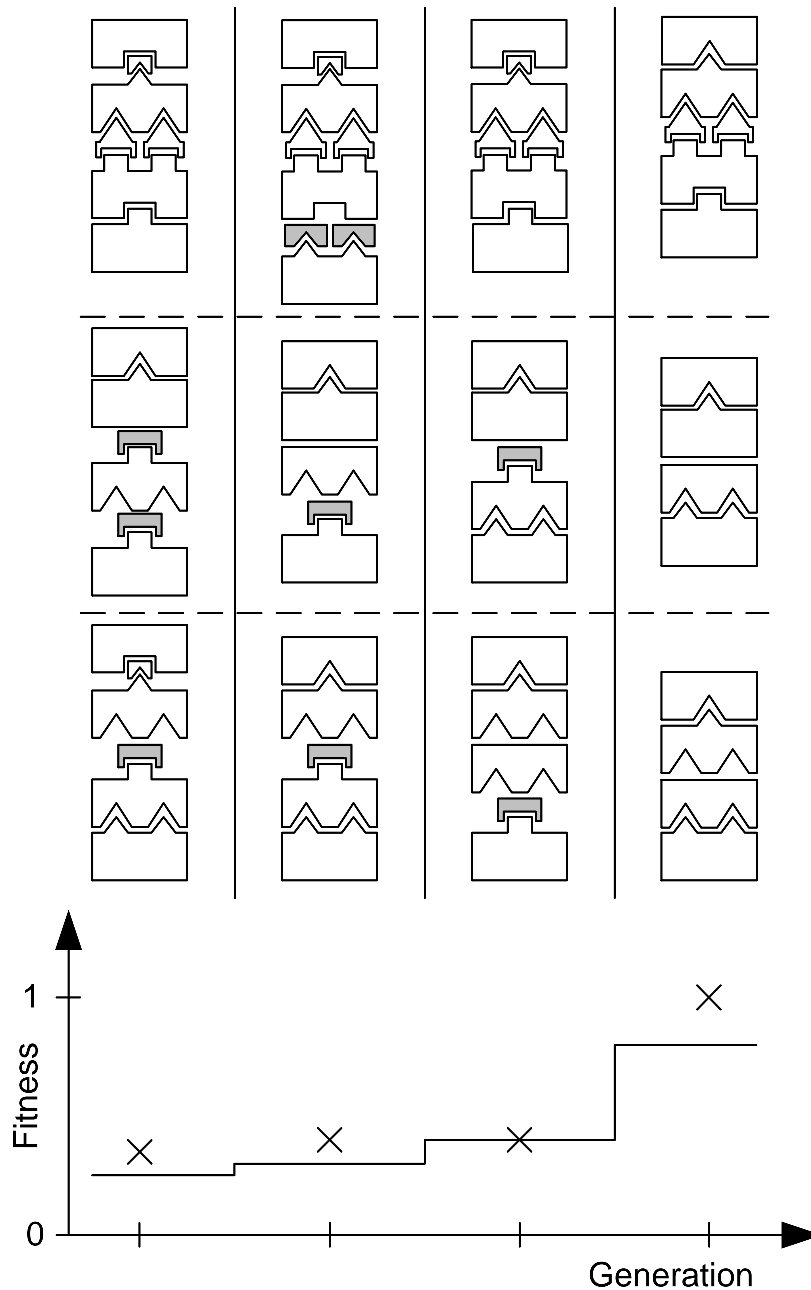


Abb. 3.38: *Optimierung des schematischen Anwendungsbeispiels*

Ein sich aus diesen Annahmen möglicherweise ergebender Verlauf der evolutionären Optimierung ist in Abbildung 3.38 über vier Generationen hinweg dargestellt. Hierbei werden pro Generation jeweils drei unterschiedliche Vari-

anten als selektierte Individuen mit größter Fitneß der Population aufgeführt. Aus Gründen der Übersichtlichkeit wird auf eine Bezeichnung der Individuen oder ihres Genotyps verzichtet. Diese Verhältnisse ergeben sich aus der abgebildeten Matrix bzw. dem Vergleich mit den entsprechenden grafischen Symbolen in Abbildung 3.37. Beispielsweise setzt sich Variante $\vec{x}_{31}$ als erstes Individuum der dritten Generation (erste Zeile, dritte Spalte der Matrix) aus den Komponenten A2, B3, C3 und D2 zusammen. Ihre Fitneß berechnet sich zu $f(\vec{x}_{31}) = 0.7^3 \cdot 1 = 0.342$, da insgesamt zwar drei Adapter aber immerhin keine generische Eingabe benötigt werden.

Der untere Teil der Abbildung zeigt den Verlauf der durchschnittlichen Fitneß aller selektierten Individuen als durchgezogene Linie, während der jeweils beste erreichte Fitneß-Wert innerhalb einer Generation durch ein Kreuz gekennzeichnet ist. Somit wird ersichtlich, daß zu Beginn des Verfahrens die zufällig zusammengestellte Ausgangsgeneration mit dem Wert 0.25 eine verhältnismäßig geringe durchschnittliche Fitneß aufweist. Jedoch entsteht etwa durch Mutation des Individuums $\vec{x}_{21}$ im Gen C vom Allel C2 nach C1 eine neue Variante $\vec{x}_{22}$ der nächsten Generation, die bereits eine deutlich höhere Fitneß (0.4) besitzt. Eine vergleichbare Aussage betrifft Variante $\vec{x}_{13}$, da wegen ihrer Mutation im Gen A vom Allel A2 nach A1 in Variante $\vec{x}_{23}$ nunmehr kein Adapter benötigt wird. Demgegenüber führt eine ungünstige Mutation im Gen D des Individuums $\vec{x}_{11}$ zur Variante $\vec{x}_{21}$ mit zwei zusätzlich erforderlichen Eingabemöglichkeiten und einer entsprechend geringen Fitneß (0.128). Dennoch ist die durchschnittliche Fitneß der zweiten Generation mit ca. 0.3 bereits etwas höher als zuvor.

Bei Bildung der nächsten Generation führt die Rekombination der Varianten $\vec{x}_{22}$ und $\vec{x}_{23}$ zwischen den Genen A und B sowie C und D zu den Individuen $\vec{x}_{32}$ und $\vec{x}_{33}$, die allerdings gegenüber ihren Vorgängern keine verbesserte Fitneß aufweisen. Hingegen ist Variante $\vec{x}_{31}$ als Rückmutation zum Genotyp $\vec{x}_{11}$ aufzufassen, so daß sich insgesamt eine wiederum leicht verbesserte durchschnittliche Fitneß von ca. 0.4 ergibt. Schließlich bewirkt eine Mutation im Gen A der Variante $\vec{x}_{31}$ während des letzten betrachteten Schritts eine weitere geringe Verbesserung der entstandenen Variante $\vec{x}_{41}$. Jedoch führt eine Mutation im Gen C bzw. D des Individuums $\vec{x}_{32}$ und $\vec{x}_{33}$ zu den überaus günstigen Varianten $\vec{x}_{42}$ und $\vec{x}_{43}$ mit ihrer im Beispiel maximalen Fitneß von 1. Dementsprechend repräsentiert auch die durchschnittliche Fitneß der letzten Generation mit einem Wert von ca. 0.8 das Maximum aller aufgeführten Generationen.

Offensichtlich kann das so zusammengefaßte, schematische und daher stark vereinfachte Anwendungsbeispiel keinen endgültigen Beweis für Tauglichkeit des vorgestellten Verfahrens liefern. Dennoch vermittelt es einen Eindruck vom besonderen Charakter und Ablauf einer evolutionären Optimie-

rung. Zudem ergeben sich aus der Betrachtung wertvolle Hinweise für die Gestaltung des Genetischen Algorithmus und besonders günstige Voraussetzungen seiner Anwendung. So wird zunächst die grundlegende Annahme bestätigt, daß einfach gestaltete Interaktionen mit einer geringen Anzahl an benötigten Parametern zu einer erheblich vereinfachten Verknüpfung unterschiedlicher Komponenten führen. Dieser Tatsache ist bei Spezifikation der übergeordneten, komponentenbezogenen Anwendungsfälle Rechnung zu tragen, um eine möglichst weitgehende Automatisierung zu erleichtern. Eine zweite wesentliche Erkenntnis des Anwendungsbeispiels betrifft die ausgezeichnete Bedeutung des Genetischen Operators Rekombination für den Erfolg der heuristischen Optimierung. Es zeigt sich, daß hierdurch besonders vielversprechende Teilkombinationen aus Allelen, im Beispiel also A1 und B1 sowie C2 und D1, gemeinsam vererbt werden und daher mit einer höheren Wahrscheinlichkeit zu überdurchschnittlich erfolgreichen Varianten beitragen.

Allerdings können derartige Hinweise erst durch umfangreiche praktische Erfahrungen mit dem vorgestellten Ansatz abschließend beurteilt werden. Dies gilt in gleicher Weise auch für den insgesamt erzielbaren Erfolg eines evolutionären Verfahrens zur Optimierung funktionaler Prototypen. Immerhin belegt die Literatur eine grundsätzliche Eignung Genetischer Algorithmen zur Lösung kombinatorischer Probleme [Gol89], wie später in Abschnitt 6.2.2 diskutiert wird. Zudem erlaubt das erarbeitete Framework auch den Einsatz alternativer Verfahren zur heuristischen Optimierung (siehe Abschnitt 5.3.4), sofern mit ihrer Hilfe effektiv und effizient vielversprechende Prototyp-Varianten abgeleitet werden können.

Falls die so erhaltenen Vorschläge für Prototyp-Varianten nicht den Erwartungen des Entwicklers oder Anwenders entsprechen, kann die gegebene Funktionale Spezifikation entsprechend verändert und anschließend eine weitere Iteration des übergeordneten Prozesses durchgeführt werden, wie in Abbildung 3.17 dargestellt ist. Der so beschriebene Ablauf verdeutlicht den grundsätzlich explorativen und experimentellen Charakter des erarbeiteten Frameworks für komponentenbasiertes Rapid Prototyping. Dieser Ansatz setzt nur wenige, verhältnismäßig einfache Informationen über den Anwendungsbereich und vorhandene Komponenten bzw. die von ihnen angebotene Funktionalität voraus. Die in der Folge zu erwartenden, zahlreichen tatsächlich ungeeigneten oder sogar fehlerhaften Prototyp-Varianten werden schrittweise im Verlauf der heuristischen Optimierung eliminiert. Eine derartige Lösung der übergeordneten Problemstellung erfüllt somit im wesentlichen die in Abschnitt 3.1 aufgeführten Anforderungen, wie später in Ab-

schnitt 6.1 ausführlich diskutiert wird.

Ein grundsätzlicher Anspruch der vorgestellten Arbeit ist jedoch die Validierung der erzielten Ergebnisse durch eine in Kapitel 4 beschriebene Referenz-Implementierung. Aus diesem Grund werden im Rahmen der oben aufgeführten Modellen und Verfahren zahlreiche Festlegungen getroffen, die offensichtlich zu gewissen Einschränkungen bei späterer Anwendung in der Praxis führen. Darüber hinaus ist anzunehmen, daß umfangreiche praktische Erfahrungen mit realistischen Problemgrößen und anderen Anwendungsbereichen für weitere Verbesserungen des grundlegenden Ansatzes herangezogen werden können. Daher ist die vorliegende Lösung als Framework konzipiert, dessen klare Strukturierung eine entsprechende Anpassung und Erweiterung seiner Elemente mit vertretbarem Aufwand ermöglicht. Dieses Merkmal der Konzeption und weitere Aspekte einer methodischen Umsetzung werden im folgenden Abschnitt genauer untersucht.

3.7 Methodische Umsetzung

Die in den vorangegangen Abschnitten vorgestellten Modelle und Verfahren sind als Bestandteile eines übergeordneten Frameworks für komponentenbasiertes Rapid Prototyping zu verstehen. Hierbei regelt das Framework das Zusammenspiel seiner Elemente mit dem Ziel, möglichst einfach und weitgehend automatisiert funktionale Prototypen aus vorhandenen Komponenten zu generieren. Obwohl in den betreffenden Abschnitten bereits Teile dieser Wechselwirkung aus Gründen der Motivation und Anschaulichkeit beschrieben sind, faßt der folgende Abschnitt die wesentlichen Erkenntnisse zusammen und stellt sie in einen übergeordneten, methodischen Zusammenhang. Dies verdeutlicht die praktische Umsetzung der erarbeiteten Ergebnisse und liefert somit die Grundlage für deren Beurteilung hinsichtlich der gestellten Anforderungen, wie in Kapitel 6 ausgeführt wird.

Für eine erfolgreiche Übertragung und Anwendung des vorgeschlagenen Frameworks sind unterschiedliche Schritte erforderlich, die sich nach ihrer zeitlichen Abfolge sowie dem voraussichtlich benötigten Aufwand differenzieren lassen. Sie werden im folgenden Überblick aufgeführt und später ausführlich beschrieben:

- Zu Beginn sollte das Framework selbst nach den jeweiligen Gegebenheiten, wie etwa verfügbare Ressourcen, Umfang und Qualität vorhandener Komponenten oder besondere praktische Erfahrungen, angepaßt und erweitert werden. Dieser auch als *Tailoring* bezeichnete Schritt legt die genaue Ausprägung der eingesetzten Elemente des Frameworks

fest. Je nach Umfang der gewünschten Veränderung können beispielsweise einzelne Parameter geeignet gewählt, Teile der Modelle an der eingesetzten technischen Plattform ausgerichtet oder sogar wesentliche Elemente des Frameworks, wie die Generierung und heuristische Optimierung der Prototyp-Varianten, vollständig durch eigene Lösungen ersetzt werden (vgl. Kapitel 5).

- Anschließend wird der betrachtete Anwendungsbereich auf logischer Ebene im Rahmen der Ontologie modelliert (siehe Abschnitt 3.3). Die so erstellten Modelle sind unabhängig von der verwendeten technischen Infrastruktur und können daher auch für andere Zwecke, etwa die spätere Systementwicklung, wiederverwendet werden.

- Begleitend kann die grundlegende Funktionalität der vorhandenen Komponenten durch entsprechende, ihnen zugeordnete Anwendungsfälle beschrieben werden (siehe Abschnitt 3.4). Die technischen Anteile der erstellten CUC-Beschreibungen entsprechen hierbei der jeweils eingesetzten Plattform sowie dem tatsächlich verwendeten Modell einer Interaktion zwischen Komponenten. Hingegen kann die Erfassung der logischen Anteile zur Erstellung und Verbesserung der Ontologie herangezogen werden, falls sich hierbei genauere Erkenntnisse über den Anwendungsbereich ergeben.

- Nach diesen erforderlichen Vorarbeiten erfolgt die eigentliche Generierung funktionaler Prototypen. Dies ist ein iterativ organisierter Prozeß aus Erstellung und Modifikation der gegebenen Funktionalen Spezifikation, sowie Generierung und Bewertung der erhaltenen Prototypen (siehe Abschnitt 3.5 und 3.6). Je nach Komplexität der funktionalen Anforderungen, Vollständigkeit der Ontologie, Umfang der vorhandenen Komponenten und Adapter sowie gewünschtem Grad der Automatisierung ist hierbei mit einer unterschiedlichen Qualität der konstruierten Prototypen zu rechnen. Die so gewonnenen Erfahrungen können zu weiteren Anpassungen des Frameworks herangezogen werden.

Auf diese Weise ergibt sich durch den methodischen Einsatz des Frameworks eine kontinuierliche Verbesserung der erzielten Ergebnisse, auch wenn grundlegende Veränderungen seiner Elemente möglicherweise mit erheblichem Aufwand verbunden sind. Allerdings verspricht die vorgenommene, strikte Trennung zwischen logischer und technischer Ebene innerhalb des Ansatzes (siehe Abschnitt 3.2) eine größere Beständigkeit der anwendungsbezogenen Elemente. Die mit ihrer Hilfe erstellten Modelle des Anwendungsbereichs sowie erbrachter bzw. gewünschter Funktionalität sind daher auch

für andere Aufgaben der Systementwicklung zu nutzen, wie in Abschnitt 6.3 diskutiert wird.

Dennoch kann im allgemeinen nicht auf eine Anpassung oder Erweiterung des Frameworks verzichtet werden. Im einfachsten Fall werden zu diesem Zweck vorgegebene Parameter, Schwellwerte oder Funktionen geeignet gewählt bzw. definiert. Hierauf sind bestimmte Teile des vorgestellten Ansatzes entsprechend vorbereitet, wie in den betreffenden Abschnitten erläutert wird. So erfolgt die Auswahl potentieller Komponenten auf Basis der semantischen Kompatibilität zwischen manipulierten Konzepten des Anwendungsbereichs (siehe Abschnitt 3.3 und 3.6.2). Der Grad ihrer Kompatibilität wird nach Gleichung 3.1 durch eine geeignete Definition der spezifischen Funktionen k_e, k_l und k_p festgelegt. Die in Gleichung 3.2, 3.3 und 3.4 vorgestellten Vorschläge können bei Bedarf nach den jeweiligen Gegebenheiten und praktischen Erfahrungen durch eigene Definitionen ersetzt werden. Die so bestimmten Alternativen lassen sich darüber hinaus auch später, bei Durchführung der iterativen Prototyp-Generierung heranziehen, um eine beherrschbare Anzahl ausgewählter Komponenten zu erhalten.

Mit vergleichbar geringem Aufwand können die Bewertungsfunktionen und zugehörigen Gewichtungsfaktoren des in Abschnitt 3.6.5 beschriebenen Genetischen Algorithmus geändert werden. Sie bestimmen die Einschätzung der generierten Prototyp-Varianten hinsichtlich ihrer Qualität und somit die übergeordnete Zielsetzung der heuristischen Optimierung. Weiterhin sind durch eine geeignete dynamische Veränderung der Gewichtungsfaktoren zahlreiche, flexibel einsetzbare Strategien bei Ablauf der Optimierung möglich. Zuletzt erlauben die frei wählbaren Basisparameter des Genetischen Algorithmus, wie Größe der Population oder Anteil der Selektion, eine weitere Anpassung des Verfahrens an praktische Erfahrungen und verfügbare Ressourcen.

Ein höherer Aufwand ist jedoch mit Änderungen verbunden, die Teile der zugrundeliegenden Modellierung betreffen. So ist beispielsweise die technische Realisierung einer Interaktion innerhalb eines komponentenbezogenen Anwendungsfalls als Sequenz von Operationsaufrufen modelliert (siehe Abschnitt 3.4 sowie Abbildung 3.9). Tatsächlich kann die Umsetzung eines Anwendungsfalls auch durch andere, ausdrucksvolle Modelle von Verhalten, wie etwa Zustandsautomaten, beschrieben werden. Hierfür ist das vorgeschlagene Klassenmodell entsprechend zu ergänzen, die bisher erstellten CUC-Beschreibungen zu aktualisieren, und die in Abschnitt 3.6.4 beschriebene Varianten-Generierung geeignet zu erweitern. Darüber hinaus sollte die im Rahmen der Optimierung eingesetzte Bewertungsfunktion f_{tn} modifiziert

werden (siehe Gleichung 3.13), um die neu hinzugekommenen Anteile einer Interaktion angemessen zu berücksichtigen. Dieses Beispiel verdeutlicht, daß die klare Strukturierung des vorgestellten Frameworks auch weitergehende Anpassungen vereinfacht. Letztlich ist die Umsetzung der Varianten-Generierung ohnehin für jede unterstützte technische Plattform erneut zu implementieren, so daß diesbezügliche Erweiterungen mit nur geringem zusätzlichen Aufwand zu integrieren sind.

In bestimmten Fällen ist sogar die vollständige Ersetzung eines Elements des Frameworks mit vertretbarem Aufwand durchzuführen. Beispielsweise können durchaus fortgeschrittene Verfahren zur automatischen Adapter-Generierung eingesetzt werden (vgl. Abschnitt 5.3), falls die so erstellten Adapter die vom Framework erwarteten Schnittstellen implementieren (siehe Abschnitt 3.6.3). In der Regel muß jedoch mit deutlich höherem Aufwand gerechnet werden, da gerade auf technischer Ebene ein enges Zusammenspiel der einzelnen Elemente erforderlich ist und somit zahlreiche Abhängigkeiten zwischen ihnen existieren. Trotzdem kann sich eine derartige Veränderung des Frameworks lohnen, etwa wenn der Grad der Automatisierung zugunsten einer höheren Qualität der generierten Prototypen verringert wird. In diesem Fall sind nahezu alle Teilschritte der in Abschnitt 3.6 beschriebenen Prototyp-Generierung entsprechend anzupassen, d.h. auf einen weitgehend interaktiven Betrieb auszurichten (vgl. Abschnitt 5.4). Auf diese Weise können die beteiligten Komponenten wesentlich zuverlässiger über ihre angegebenen Interaktionen verknüpft werden. Dennoch sind die Modelle des Anwendungsbereichs sowie die erstellten CUC-Beschreibungen weiterhin zur automatisierten Suche und Auswahl vielversprechender Komponenten geeignet.

Bevor das so angepaßte Framework zur Generierung funktionaler Prototypen genutzt werden kann, sind offensichtlich eine Ontologie des jeweiligen Anwendungsbereichs sowie entsprechende CUC-Beschreibungen für vorhandene Komponenten erforderlich (siehe Abschnitt 3.3 und 3.4). Durch die zunehmende Integration von Geschäftsprozessen über weltweite Datennetze sind zumindest mittelfristig standardisierte, deklarative Beschreibungen bestimmter, wirtschaftlich bedeutender Anwendungsbereiche zu erwarten [Mic01a]. Diese können als Grundlage einer entsprechenden Ontologie dienen, wobei die in Abschnitt 3.3 gewählte, objekt-orientierte Modellierung eine weitgehend intuitive, leicht verständliche Umsetzung erlaubt. Darüber hinaus sind bereits zahlreiche existierende Ontologien verfügbar [Sta01], die mit geeigneter Unterstützung durch Werkzeuge wenigstens teilweise in das in dieser Arbeit vorausgesetzte Format übersetzt werden können. Anschließend führt

eine manuelle Bearbeitung der so erhaltenen Ergebnisse zu einer möglichst vollständigen, konsistenten und klar strukturierten Ontologie im Sinne des vorgestellten Frameworks.

Für die Erstellung, Ergänzung oder Modifikation einer Ontologie kann in der Regel auf bekannte Mittel und Verfahren der objekt-orientierten Analyse, wie Gespräche mit dem Anwender oder Untersuchung der relevanten Fachliteratur, zurückgegriffen werden [Som87, MW91, Wie99]. Hierbei ist zunächst eine Aufteilung nach deutlich abgegrenzten Domänen vorzunehmen, um durch Import oder Interpretation eine modulare Strukturierung, vereinfachte Pflege und spätere Wiederverwendung ausgewählter Domänen zu erreichen. Nunmehr werden die zuerst identifizierten, grundlegenden Konzepte jeder Domäne durch Einführung entsprechender Generalisierungen in verschiedenen Hierarchien angeordnet, die gewissermaßen das „Rückgrat" der Ontologie bilden. Anschließend können durch Verwendung von Relationen mit vordefinierter Semantik, wie Aggregation, Äquivalenz oder alternative Auswahl, zusätzliche Konzepte und maßgebliche Verhältnisse im Anwendungsbereich repräsentiert werden. Zuletzt werden spezifische Beziehungen mit eindeutigen Merkmalen, wie Name oder Richtung, definiert und zur Beschreibung entsprechender Sachverhalte in der betrachteten Domäne eingesetzt. Offensichtlich führt die Diskussion der so erhaltenen Ergebnisse mit dem Anwender zu weiteren Korrekturen und Ergänzungen, die letztlich in einer besseren Abbildung des Anwendungsbereichs resultieren. Darüber hinaus können Bereiche mit unterschiedlich genauer Modellierung durch Angabe einer konzeptuellen Distanz für Generalisierungs- und Interpretationsbeziehungen wiedergegeben werden, wie in Abschnitt 3.3 erläutert ist.

Tatsächlich führt auch die Annotation vorhandener Komponenten durch entsprechende CUC-Beschreibungen im allgemeinen zu Verbesserungen oder Erweiterungen der betrachteten Ontologie. Schließlich liefert die in Abschnitt 3.4 eingeführte Modellierung der grundlegenden Funktionalität als Manipulation von Konzepten ebenfalls wertvolle Erkenntnisse über den Anwendungsbereich. Wie am Beispiel der Repräsentation ersichtlich ist, gibt die betrachtete Funktionalität in vielen Fällen bedeutende, logisch motivierte Beziehungen zwischen Konzepten unmittelbar wieder (vgl. Abschnitt 3.6.3). Dennoch liegt der Schwerpunkt bei Erstellung einer CUC-Beschreibung auf der möglichst prägnanten, anwendungsbezogenen Formulierung angebotener Funktionalität. Hierfür ist neben dem Bezug auf die Ontologie auch die Benennung der zugehörigen Manipulation von Bedeutung. Falls nicht ohnehin entsprechende Vorgaben getroffen sind, wie später in Abschnitt 5.2.1 vorgeschlagen wird, ist im Zweifel auch eine mehrfache, also redundante Beschreibung der Funktionalität auf logischer Ebene möglich. So kann beispielsweise der in Abbildung 3.10 dargestellte Anwendungsfall Calculate 1 Alignment zur

Komponente `Clustal` offensichtlich in gleicher Weise auch mit Align 1..* DNA bezeichnet werden. In diesem Fall erlaubt eine einfache Kopie der zugehörigen technischen Anteile, die so beschriebene Funktionalität möglichst flexibel auszuwählen und bei der Generierung in Anspruch zu nehmen.

In vergleichbarer Weise können auf technischer Ebene verschiedene Realisierungen des gleichen CUC angegeben werden, um die Funktionalität einer Komponente über unterschiedliche, alternativ angebotene Interaktionen zu nutzen (siehe Abschnitt 3.4). Diese unterscheiden sich im allgemeinen in ihrer Komplexität, d.h. der Anzahl beteiligter Rollen und sekundärer Konzepte als Parameter oder Rückgabewerte von Operationsaufrufen, sowie dem vorausgesetzten dynamischen Ablauf der Interaktion. Hierbei ist es sinnvoll, zumindest eine ausgewählte Interaktion mit möglichst geringer Komplexität zu beschreiben, um die spätere automatische Verknüpfung mit anderen Komponenten zu erleichtern. Zu diesem Zweck können beispielsweise erforderliche Parameter mit festgelegten Konstanten oder zur Laufzeit auswertbaren Ausdrücken belegt werden. Darüber hinaus sollten unterschiedliche Interaktionen angegeben werden, falls die betreffende Komponente mehrere Repräsentationen des gleichen primären oder sekundären Konzepts unterstützt. Hierdurch besteht die Möglichkeit, die Anzahl evtl. benötigter Adapter zu minimieren und somit den erforderlichen manuellen Aufwand zu verringern (siehe Abschnitt 3.6.3). In jedem Fall kann die geeignete Umsetzung einer zielführenden Interaktion der beiliegenden Dokumentation einer Komponente entnommen werden. Sie beinhaltet mindestens eine textuelle Beschreibung der zugehörigen Schnittstellen und ihrer Benutzung. Häufig werden zusätzlich exemplarische Code-Fragmente angegeben, die typische Abläufe und Szenarien illustrieren (vgl. Abschnitt 1.1). Diese lassen sich in der Regel als Ausgangspunkt zur Beschreibung der jeweiligen Interaktion im Sinne des vorgestellten Frameworks heranziehen.

Die vielfältigen Möglichkeiten zur Beeinflussung des eigentlichen Ablaufs der Prototyp-Generierung werden in Abschnitt 3.6 ausführlich behandelt. Tatsächlich sind gerade in diesem Bereich des Frameworks je nach Grad der gewünschten Automatisierung sowie Komplexität und Qualität der erhaltenen Prototyp-Varianten zahlreiche Anpassungen und Erweiterungen zu erwarten, die später in Abschnitt 5.3 aufgeführt werden. Aus diesem Grund werden an dieser Stelle keine weiterführenden Aussagen getroffen. Dennoch liefert die Validierung der erzielten Ergebnisse durch die in Kapitel 4 beschriebene Referenz-Implementierung bereits wertvolle Erkenntnisse über deren Umsetzung in die Praxis.

3.8 Zusammenfassung

Wesentliche Anforderungen an einen Ansatz für komponentenbasiertes Rapid Prototyping sind angemessene Expressivität und Komplexität der angebotenen Mittel zur Beschreibung von Funktionalität. Sie ermöglichen den grundsätzlich erforderlichen Abgleich zwischen gewünschter und erbrachter Funktionalität. Das hierfür eingesetzte Verfahren sollte eine effektive, möglichst weitgehend automatisierte Generierung funktionaler Prototypen auch bei zunächst nicht exakt bestimmten, initialen funktionalen Anforderungen erlauben. Hierbei ist zu berücksichtigen, daß verwendete Komponenten in der Regel von unterschiedlichen Herstellern bezogen werden und in ihrer Implementierung nicht verändert werden können. Darüber hinaus ist die Skalierbarkeit auch für eine hohe Anzahl potentiell geeigneter Komponenten zu gewährleisten. Zudem sollte die praktische Umsetzung der erzielten Ergebnisse für gegenwärtig bedeutende technische Plattformen sichergestellt werden. Zuletzt ist eine möglichst umfassende Unterstützung unterschiedlicher Anwendungsbereiche wünschenswert.

Die vorliegende Arbeit begegnet diesen Anforderungen durch den Entwurf eines konzeptuellen Frameworks, das eine klare Trennung zwischen logischer, anwendungsbezogener Ebene und technischer, implementierungsbezogener Ebene vorgibt. Der betrachtete Anwendungsbereich wird auf logischer Ebene durch eine Ontologie aus verschiedenen Domänen modelliert. Jede Domäne beinhaltet Konzepte und deren Beziehungen, wobei Relationen mit vordefinierter Semantik eine kompakte, modulare und ausdrucksvolle Beschreibung der Verhältnisse im Anwendungsbereich ermöglichen. Darüber hinaus wird ein Begriff für semantische Kompatibilität zwischen Konzepten vorgestellt, der eine Abschätzung erlaubt, inwieweit eine Aussage über ein gegebenes Konzept auch für ein anderes Konzept der Ontologie gültig ist.

Mit Hilfe der Ontologie kann angebotene bzw. erwünschte Funktionalität als Manipulation von wohldefinierten Konzepten des Anwendungsbereichs aufgefaßt werden. Vorhandene Komponenten erbringen ihre so beschriebene Funktionalität durch Interaktion mit anderen Komponenten des Systems. Eine solche Interaktion kann auf technischer Ebene als typische Sequenz von Operationsaufrufen der beteiligten Schnittstellen modelliert werden. Hierbei treten Konzepte als Parameter oder beobachtbares Ergebnis der Interaktion auf. Die so geschaffene Verbindung zwischen logischer und technischer Ebene wird durch das zentrale Konzept eines komponentenbezogenen Anwendungsfalls (CUC) zusammengefaßt. Verschiedene CUCs werden einer gegebenen Komponente zugeordnet und später zu ihrer Verknüpfung herangezogen.

Die funktionalen Anforderungen an das zu entwickelnde System werden in gleicher Weise als Manipulation von Konzepten des Anwendungsbereichs

im Rahmen einer Funktionalen Spezifikation angegeben. Dies ermöglicht unter Einbeziehung der semantischen Kompatibilität zwischen Konzepten einen flexiblen und toleranten Abgleich mit Informationen der vorhandenen CUC-Beschreibungen. Die so getroffene Auswahl führt zu zahlreichen, unterschiedlich zusammengesetzten Prototyp-Varianten, deren Komponenten über in der Spezifikation deklarierte Konzepte als Parameter oder Ergebnisse ihrer zugehörigen Interaktionen verknüpft sind. Hierbei erfordert die Kombination von Komponenten unterschiedlicher Hersteller in der Regel die Vermittlung zwischen verschiedenen technischen Repräsentationen des gleichen logischen Konzepts. Diese Aufgabe wird von Adaptern erfüllt, die entweder explizit durch den Benutzer oder, bei einfach zusammengesetztem Zustand einer Repräsentation, auch automatisch durch das Framework bereitgestellt werden.

Für die auf diese Weise zusammengestellten Prototyp-Varianten kann in der Folge Quellcode der verwendeten technischen Plattform generiert werden, der sich übersetzen und ausführen läßt. Hierfür müssen im wesentlichen die Teilnehmer jeder Interaktion sowie die erforderlichen Parameter durch Instanzen der beteiligten Komponenten belegt werden. Je nach gewünschtem Grad der Automatisierung ist nach diesem Schritt mit einer hohen Anzahl an tatsächlich ungeeigneten oder sogar fehlerhaften Prototyp-Varianten zu rechnen. Aus diesem Grund führt der vorgestellte Ansatz eine heuristische Optimierung ein, welche diese ungünstigen Varianten schrittweise eliminiert. Der hierfür eingesetzte Genetische Algorithmus betrachtet Prototyp-Varianten als Individuen einer Population, die sich durch fortgesetzte Anwendung der Genetischen Operatoren Selektion, Mutation, Reproduktion und Rekombination in einem zufallsgesteuerten, evolutionären Prozeß weiterentwickelt. Hierbei bestimmt die Definition der für jedes Individuum ausgewerteten Fitneß-Funktion maßgeblich das Ziel der Optimierung. Sie setzt sich aus einer automatisch durchgeführten Beurteilung der technischen Merkmale, sowie einer durch den Benutzer vorgenommenen, interaktiven Beurteilung der anwendungsbezogenen Merkmale einer Variante zusammen. Die erarbeitete Umsetzung des Genetischen Algorithmus auf die vorliegende Problemstellung erlaubt eine weitgehende, flexible Anpassung an die verfügbaren Ressourcen und den erwünschten Grad der Automatisierung. Darüber hinaus kann das Verfahren jederzeit unterbrochen und durch Änderung der Spezifikation ein erneuter Abgleich der Funktionalität herbeigeführt werden. Somit ergibt sich ein übergeordneter, iterativ organisierter Prozeß zur Erstellung funktionaler Prototypen aus vorhandenen Komponenten.

Die in diesem Kapitel beschriebenen Modelle und Verfahren des erarbeiteten Frameworks sind zu großen Teilen auf Basis einer ausgewählten techni-

schen Plattform umgesetzt. Diese im folgenden Kapitel erläuterte Referenz-Implementierung ermöglicht eine praktische Validierung der erzielten Ergebnisse an Hand des exemplarisch untersuchten Anwendungsbereichs. Für eine praxisgerechte Umsetzung des Frameworks sowie eine Verbesserung der erzielten Ergebnisse sind jedoch voraussichtlich Anpassungen und Erweiterungen erforderlich, die anschließend in Kapitel 5 angesprochen werden. Sie erlauben eine umfassende Beurteilung der vorgeschlagenen Lösung hinsichtlich der in Abschnitt 3.1 aufgeführten Anforderungen, wie später in Kapitel 6 diskutiert wird.

4. IMPLEMENTIERUNG

Nach ausführlicher Beschreibung des erarbeiteten, konzeptuellen Frameworks für komponentenbasiertes Rapid Prototyping, wird im folgenden Kapitel dessen technische Realisierung für die Java-Plattform vorgestellt. Diese Referenz-Implementierung besitzt allerdings selbst prototypischen Charakter und kann daher nicht in der Systementwicklung eingesetzt werden. Dennoch verdeutlicht sie die praktische Umsetzung der in Kapitel 3 aufgeführten Modelle und Verfahren und ermöglicht nicht zuletzt eine erste empirische Überprüfung der erzielten Ergebnisse. Die so gewonnenen Erkenntnisse tragen später in Kapitel 6 zu einer Diskussion des erreichten Erfolgs bei. Darüber hinaus kann die in diesem Kapitel beschriebene Software-Architektur als Grundlage einer zukünftigen, praxisgerechten Implementierung des Frameworks dienen.

Zunächst wird eine Übersicht dieser Architektur hinsichtlich ihrer Strukturierung in Komponenten[1], verarbeiteten Informationen und dem Zusammenspiel ihrer Elemente beschrieben. Anschließend werden die zentralen Bestandteile der Architektur genauer erläutert, wobei der Schwerpunkt auf deren grundsätzliche Aufgaben und Interaktionen mit anderen Komponenten gelegt wird. Technische Details, wie etwa Syntax der eingesetzten textuellen Notationen oder Signatur der beteiligten Schnittstellen, können dem Anhang sowie dem vollständigen Quellcode der Implementierung [Vil01b] entnommen werden. Zuletzt werden die wesentlichen Inhalte am Ende des Kapitels zusammengefaßt.

4.1 Übersicht

Auch wenn prinzipiell verschiedene technische Infrastrukturen durch das vorgestellte Framework unterstützt werden, wie später in Abschnitt 6.1.5 diskutiert wird, so eignet sich doch die Java-Plattform [AGH00] in besonderem Maße für eine prototypische Umsetzung der erarbeiteten Ergebnisse. Java selbst ist eine moderne, objekt-orientierte Programmiersprache mit klar

[1] Zur besseren Unterscheidung werden die funktionalen Komponenten des Anwendungsbereichs in diesem Kapitel ausschließlich als *Softwarekomponenten* bezeichnet.

definierten Konzepten, umfangreicher Klassenbibliothek und leistungsfähigen Entwicklungswerkzeugen. Darüber hinaus ermöglicht Java über den sog. Reflection-Mechanismus eine dynamische Auswertung von Typinformationen zur Laufzeit. Dies erleichtert die Generierung von Quellcode für funktionale Prototyp-Varianten erheblich (vgl. Abschnitt 3.6.4). Außerdem verfügt die Java-Plattform mit *JavaBeans* [Sun00a] über ein ausgereiftes technisches Komponentenmodell, welches die Erstellung und Verwendung ausführbarer Softwarekomponenten vereinfacht (siehe Abschnitt 2.2).

Aus diesen Gründen bezieht sich die im folgenden beschriebene Referenz-Implementierung auf die Gegebenheiten dieser Plattform, obwohl sich die vorgestellte Software-Architektur auch auf andere technische Infrastrukturen übertragen läßt. Schließlich definiert eine solche Architektur die grundlegende Strukturierung des übergeordneten Systems hinsichtlich seiner Komponenten und ihres Zusammenspiels [SG96]. Hierbei folgt die praktische Umsetzung des entwickelten Frameworks im wesentlichen der in Abschnitt 3.6 erläuterten Aufgabenteilung bei Generierung funktionaler Prototypen.

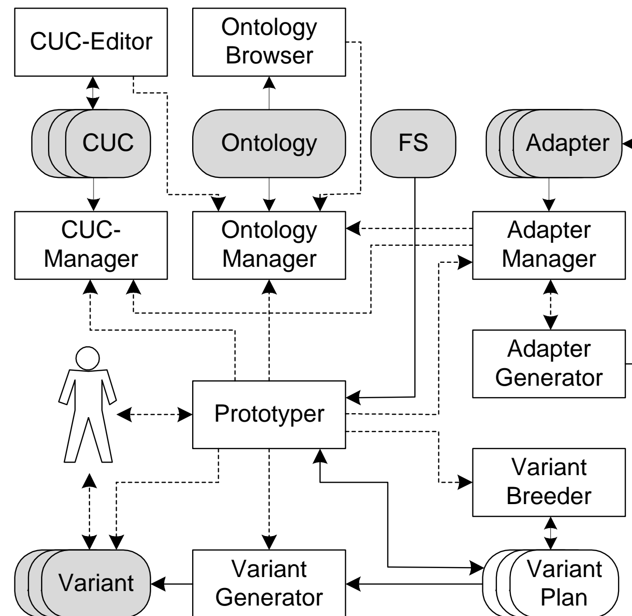


Abb. 4.1: *Grobarchitektur der Referenz-Implementierung*

Dementsprechend ergibt sich die in Abbildung 4.1 dargestellte Grobarchitektur der Referenz-Implementierung. Sie zeigt die funktionalen Hauptkomponenten als beschriftete Rechtecke, sowie den wesentlichen Informationsfluß und vorhandene Interaktionen zwischen Komponenten als durchgezoge-

ne bzw. unterbrochene Pfeile. Hierbei repräsentiert die angegebene Richtung einer Interaktion die Inanspruchnahme von Funktionalität der betreffenden Komponente, also eine Benutzung ihrer zugehörigen Schnittstelle. Darüber hinaus sind die verarbeiteten Beschreibungen, Dokumente oder generierten Zwischenergebnisse als abgerundete Rechtecke symbolisiert, wobei persistent gespeicherte Daten durch eine dunkle Färbung gekennzeichnet sind.

Somit wird aus Abbildung 4.1 ersichtlich, daß der zur interaktiven Bearbeitung von CUC-Beschreibungen eingesetzte CUC-Editor entsprechende Funktionalität der Komponente Ontology Manager nutzt, um die zur Verfügung stehenden Konzepte der Ontologie zu ermitteln. Die zugrundeliegende Ontologie selbst wird durch eine textuelle Notation beschrieben (siehe Anhang A.1), welche durch den Ontology Manager eingelesen und in eine geeignete, interne Repräsentation übersetzt wird (vgl. Abbildung 3.5). Darüber hinaus implementiert diese zentrale Komponente auch die in Abschnitt 3.3 eingeführten Definitionen und Regeln für Konzepte der Ontologie und deren Beziehungen. Mit ihrer Hilfe kann beispielsweise festgestellt werden, welche Konzepte in einer (möglicherweise transitiven) Generalisierungsbeziehung zueinander stehen, welche Subinterpretationen für eine bestimmte Interpretation zu beachten sind, oder wie hoch der Grad der semantischen Kompatibilität zwischen gegebenen Konzepten einzuschätzen ist.

Die interne Repräsentation einer Ontologie wird auch von der Komponente Ontology Browser interpretiert, um dem Benutzer eine übersichtliche Darstellung der Ontologie, also ihrer beinhalteten Domänen, Konzepte und Relationen, anzubieten. Darüber hinaus kann der Ontology Browser unter Benutzung des Ontology Managers auch abgeleitete Informationen, wie transitive Beziehungen, konzeptuelle Distanzen oder den Grad der semantischen Kompatibilität, geeignet anzeigen. Dies erleichtert die Erstellung, Überprüfung und Pflege der betrachteten Ontologie, gerade bei umfangreichen Anwendungsbereichen. Beispielsweise umfaßt die im Rahmen dieser Arbeit eingesetzte Ontologie 9 Domänen mit insgesamt 823 Konzepten und 871 Relationen[2].

Die mit Hilfe des CUC-Editors erstellten CUC-Beschreibungen (siehe Anhang A.3) werden den vorhandenen Softwarekomponenten zugeordnet und in ihrer Gesamtheit durch den CUC-Manager in einer internen Repräsentation verwaltet (vgl. Abbildung 3.7 und 3.9). Dies erlaubt den geordneten Zugriff auf die enthaltenen Informationen über angebotene Anwendungsfälle, Schnittstellen, Interaktionen, usw. Sie werden von der zentralen Komponente Prototyper benötigt, um die logische Kompatibilität zwischen Anwen-

[2] Wie bereits in Abschnitt 3.3 erwähnt, basiert die verwendete Ontologie auf Vorarbeiten des TAMBIS-Projekts [BBB+99].

dungsfällen der Funktionalen Spezifikation FS und den existierenden CUCs zu ermitteln (vgl. Abschnitt 3.6.2). Hierfür liest Prototyper zunächst die in einer textuellen Notation angegebene Spezifikation ein (siehe Anhang A.2) und setzt diese in eine interne Repräsentation um (vgl. Abbildung 3.13). Anschließend können die vom Ontology Manager bereitgestellten Informationen über semantische Kompatibilität genutzt werden, um möglicherweise geeignete Softwarekomponenten bzw. deren zugehörige Anwendungsfälle auszuwählen.

Die so ermittelte Auswahl bestimmt den möglichen Lösungsraum aller unterschiedlich zusammengesetzter Prototyp-Varianten. Die Komponente Variant Breeder beschränkt diese potentiell hohe Anzahl auf eine beherrschbare, zufällig bestimmte Teilmenge nach den Vorgaben des durch sie implementierten Genetischen Algorithmus (siehe Abschnitt 3.6.5). Je nach Zusammensetzung der so festgelegten Konstruktionspläne für Varianten (Variant Plan) sind evtl. zusätzliche Adapter für unterschiedliche Repräsentationen von Konzepten des Anwendungsbereichs erforderlich (siehe Abschnitt 3.6.3). Die vorhandenen, zu Beginn durch den Benutzer bereitgestellten Adapter werden durch die Komponente Adapter Manager verwaltet und bei Bedarf durch die Komponente Prototyper in den jeweiligen Variant Plan eingebunden. Falls jedoch im Verlauf der Prototyp-Generierung ein bisher nicht existierender Adapter benötigt wird, ist es die Aufgabe des Adapter Managers eine entsprechende Softwarekomponente mit Hilfe des Adapter Generators zu erstellen. Zu diesem Zweck kann auf Informationen der Ontologie sowie vorhandene Repräsentationen und ihre angebotene Anwendungsfälle zurückgegriffen werden.

Der oben beschriebene Ablauf wiederholt sich für alle Varianten bzw. alle Generationen des Genetischen Algorithmus. Je nach gewählter Strategie im Rahmen der heuristischen Optimierung sind ausgewählte Varianten durch den Benutzer interaktiv zu bewerten (vgl. Abschnitt 3.6.5). Die durch Variant Breeder erhaltenen Konstruktionspläne dieser Varianten werden an die Komponente Variant Generator übergeben, die sie gemäß den Vorgaben aus Abschnitt 3.6.4 in Java-Quellcode umsetzt. Die so generierten, funktionalen Prototyp-Varianten werden übersetzt, ausgeführt und nach Interaktion mit dem Benutzer bewertet. Hierfür bietet Prototyper dem Benutzer eine entsprechende Oberfläche an und leitet dessen Beurteilung an die Komponente Variant Breeder zur Ermittlung des übergeordneten Fitneß-Werts weiter. Er bestimmt maßgeblich den weiteren Verlauf der iterativen Optimierung, wie in Abschnitt 3.6.5 ausführlich erläutert wird. Sobald sich auf diese Weise keine weitere Verbesserung der durchschnittlichen Fitneß erzielen läßt oder der Benutzer das Verfahren explizit beendet, sind die zwischenzeitlich generierten Varianten zunächst als erzieltes Ergebnis des Verfahrens aufzufassen. Darüber hinaus kann die gegebene Funktionale Spezifikation verändert und

damit eine erneute Iteration des übergeordneten Prozesses begonnen werden (vgl. Abbildung 3.17).

Die so beschriebene Übersicht der Referenz-Implementierung verdeutlicht, daß die klare Strukturierung des konzeptuellen Frameworks in der vorgestellten Software-Architektur wiedergegeben wird. Die logisch motivierten, grundlegenden Aufgaben der übergeordneten Problemstellung werden auf eigene Komponenten der Architektur abgebildet, die somit weitgehend lokal und nur mit den tatsächlich erforderlichen Abhängigkeiten implementiert werden können. Dies erleichtert Erstellung und Weiterentwicklung einer geeigneten Implementierung des Frameworks erheblich. Im folgenden Abschnitt werden nunmehr die wesentlichen Komponenten der Architektur genauer erläutert und, wo möglich, mit exemplarischen Abläufen und Bildschirmdarstellungen veranschaulicht. Die so vermittelten Erkenntnisse über die Referenz-Implementierung sind allerdings nicht als festgelegte Vorgaben, sondern vielmehr als Anhaltspunkte für eine zukünftige, praxisgerechte Umsetzung des Frameworks zu verstehen.

4.2 Komponenten der Architektur

Wie bereits in Abschnitt 3.4 erwähnt, ist es durchaus zweckmäßig, die Zuordnung zwischen erstellten CUC-Modellen und vorhandenen Softwarekomponenten statisch vorzunehmen. Auf diese Weise bilden ausführbare Komponenten und sie beschreibende Informationen eine in sich geschlossene Einheit, die gemeinsam archiviert, verteilt und schließlich genutzt werden kann. Daher definiert die Referenz-Implementierung eine textuelle Notation für CUC-Beschreibungen, deren Syntax in der sog. *Extensible Markup Languange* (XML) [XML01] festgelegt ist (siehe Anhang A.3). XML ist ein weltweit anerkannter Standard zum Austausch strukturierter Informationen, der gegenwärtig durch zahlreiche Werkzeuge auf nahezu allen verfügbaren Plattformen unterstützt wird. Somit können entsprechende Dokumente dem binären Format einer Softwarekomponente beigefügt und später auf möglichst flexible Weise genutzt werden. Im Rahmen der Java-Plattform kann dies beispielsweise durch Aufnahme in das zugehörige JAR-Archiv einer JavaBeans-Komponente geschehen (vgl. Abschnitt 2.2).

Auch wenn prinzipiell jeder verfügbare, generische XML-Editor zur Bearbeitung der so festgelegten CUC-Beschreibungen herangezogen werden kann, so ist doch aufgrund der Komplexität der beschriebenen Verhältnisse ein spezifisches Werkzeug für diese Aufgabe zu bevorzugen. Schließlich beinhaltet das in Abbildung 3.7, 3.8 und 3.9 dargestellte Modell für angebotene

Funktionalität zahlreiche unterschiedliche Elemente sowie deren logische Zusammenhänge und Konsistenzbedingungen (siehe Abschnitt 3.4). Aus diesem Grund ermöglicht die Komponente CUC-Editor der in Abbildung 4.1 vorgestellten Architektur eine komfortable, interaktive Erstellung und Bearbeitung von CUC-Beschreibungen.

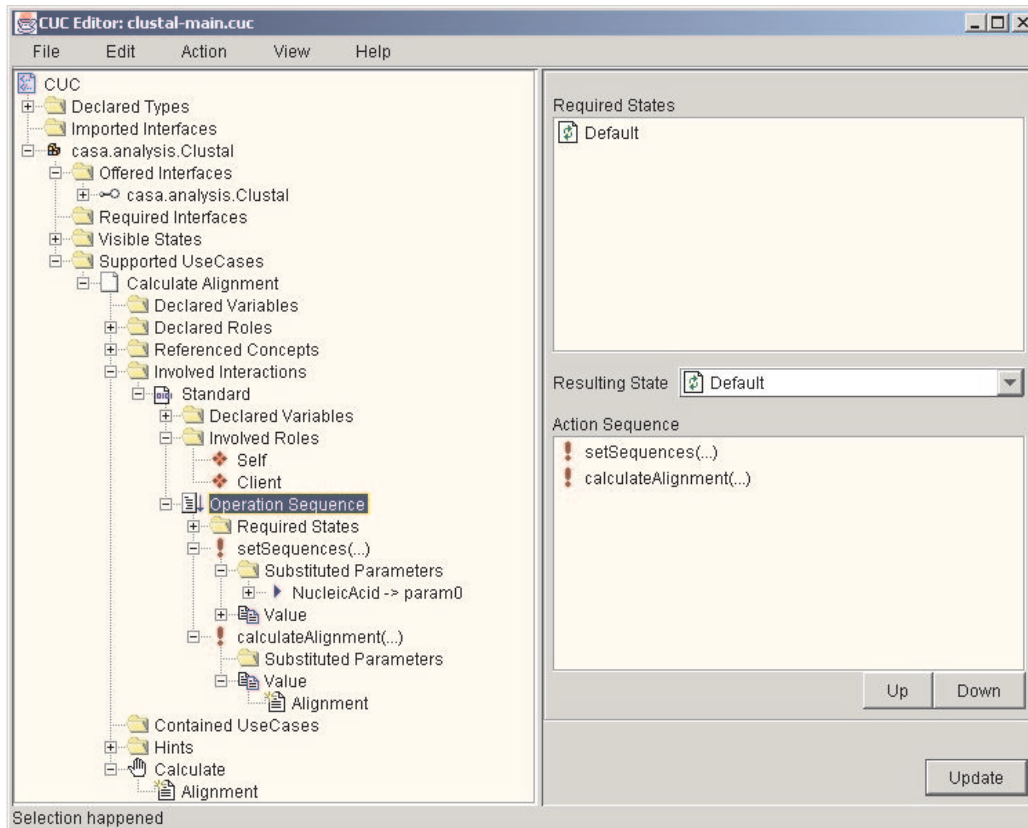


Abb. 4.2: *Bildschirmdarstellung der Komponente CUC-Editor*

Abbildung 4.2 zeigt die gewählte grafische Benutzeroberfläche des Werkzeugs bei Bearbeitung der CUC-Beschreibung zur Softwarekomponente `casa.analysis.Clustal` des Anwendungsbeispiels (siehe Abschnitt 3.6.1). Auf der linken Seite ist die Struktur des zugrundeliegenden Modells dargestellt, dessen zugehörige Elemente, wie etwa angebotene Schnittstellen und Anwendungsfälle, Interaktionen oder Folgen von Operationsaufrufen, durch den Benutzer selektiert werden können. Daraufhin werden auf der rechten Seite entsprechende Eingabefelder zur Erstellung oder Bearbeitung des jeweils selektierten Elements angezeigt. Im Beispiel ist die zum CUC Calculate Alignment gehörige Umsetzung der Interaktion `Standard` ausgewählt.

Dementsprechend kann der durch die Interaktion vorausgesetzte bzw. erreichte Zustand der Komponente sowie die Abfolge der Operationsaufrufe festgelegt werden (vgl. Abbildung 3.10).

Durch eine derartige, kontextsensitive Benutzerführung können auch umfangreiche und komplexe CUC-Beschreibungen mit vertretbarem Aufwand erstellt werden. Hierbei überprüft der Editor die Konsistenz des Modells bereits während der Eingabe. So können beispielsweise nur Aufrufe von Operationen angegeben werden, die auch tatsächlich in angebotenen oder importierten Schnittstellen der beschriebenen Softwarekomponente aufgeführt sind. Darüber hinaus wird durch Zusammenarbeit mit dem Ontology Manager festgestellt, welche Konzepte der Ontologie zur Beschreibung von Funktionalität verwendet werden können (vgl. Abbildung 4.1). Weitere Funktionen des Editors, wie Kopieren und Einfügen von Elementen der Struktur oder automatische Ermittlung der Signatur von Java-Klassen und Komponenten, erlauben eine komfortable, möglichst schnelle Einbindung vorhandener Softwarekomponenten in das übergeordnete Verfahren.

Im abschließenden Schritt der Bearbeitung wird die betreffende CUC-Beschreibung als XML-Datei gespeichert (siehe Anhang A.3), die später durch den CUC-Manager eingelesen und den anderen Komponenten der Architektur zugänglich gemacht wird. Hierbei teilen sich CUC-Editor und CUC-Manager in ihrer Implementierung gemeinsame Subkomponenten zur Verarbeitung des XML-Formats und dessen Umsetzung in die intern benutzte Repräsentation.

In vergleichbarer Weise wird auch die betrachtete Ontologie mit Hilfe einer textuellen Notation angegeben. Aufgrund der Verwendung einer bereits existierenden Ontologie des gewählten Anwendungsbereichs (siehe Abschnitt 3.3) wird hierfür allerdings eine eigene, nicht standardisierte Definition der Syntax eingesetzt (siehe Anhang A.1). Sie erlaubt eine kompakte, einfach zu bearbeitende Beschreibung der Ontologie hinsichtlich ihrer beinhalteten Domänen, Konzepte und Relationen. Hierfür kann prinzipiell jeder verfügbare Texteditor benutzt werden. Dennoch erscheint mittelfristig eine ebenfalls XML-basierte Repräsentation sinnvoll, um den späteren Austausch von allgemein anerkannten Ontologien zu erleichtern. Darüber hinaus erlaubt die vorgestellte, objekt-orientierte Modellierung einer Ontologie (siehe Abbildung 3.5) deren komfortable Bearbeitung durch grafische Editoren, etwa im Rahmen eines übergreifenden CASE-Werkzeugs, wie später in Abschnitt 5.1 diskutiert wird.

In jedem Fall erfordern die durchaus komplexen Zusammenhänge innerhalb der Ontologie eine geeignete, übersichtliche Aufbereitung durch entspre-

chende, interaktiv zu benutzende Komponenten. Dies erleichtert die Erstellung, Überprüfung und Pflege der eingesetzten Ontologie. Schließlich führen die in Abschnitt 3.3 eingeführten Definitionen und Regeln zu abgeleiteten, weiterführenden Erkenntnissen über den Anwendungsbereich, die nicht unmittelbar aus der textuellen Beschreibung ersichtlich sind.

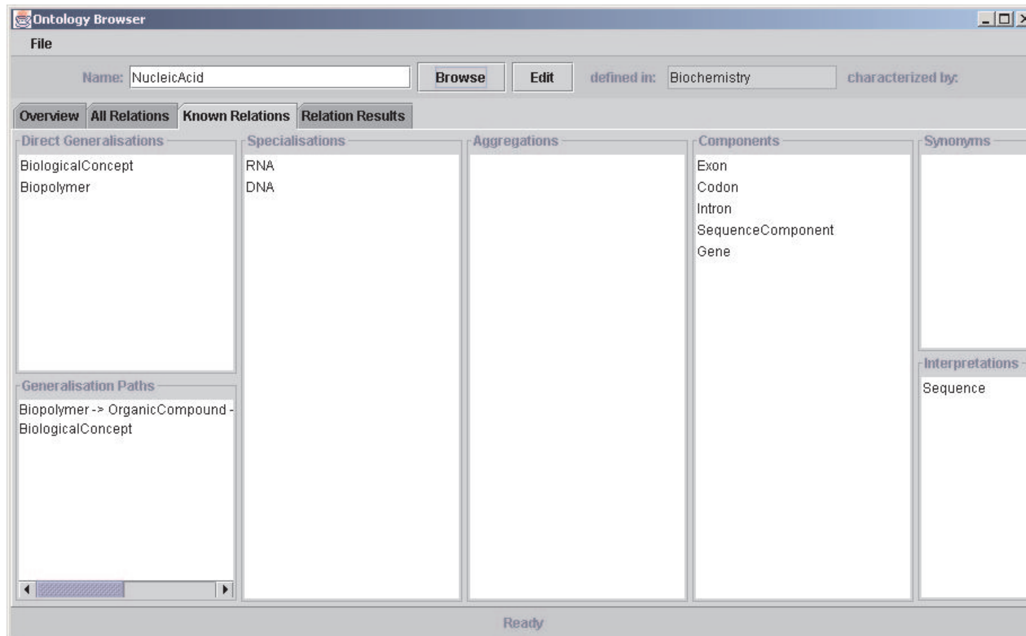


Abb. 4.3: *Bildschirmdarstellung der Komponente* Ontology Browser

Zu diesem Zweck stellt die Referenz-Implementierung die Komponente Ontology Browser bereit, deren exemplarische Bildschirmdarstellung in Abbildung 4.3 gezeigt ist. Sie ermöglicht eine strukturierte Darstellung aller in der Ontologie verfügbaren Informationen über Domänen, Konzepte und ihre Beziehungen. So werden etwa alle definierten Generalisierungen, Spezialisierungen oder Interpretationen eines gegebenen Konzepts schnell und übersichtlich aufgeführt, wie in Abbildung 4.3 am Beispiel des Konzepts NucleicAcid innerhalb der Domäne Biochemistry deutlich wird. Hierbei kann der Benutzer angezeigte Konzepte oder Relationen am Bildschirm auswählen und somit eine weitere, diesbezügliche Abfrage durchführen. Auf diese Weise ergibt sich eine interaktiv geführte Präsentation des erstellten Modells, welche die Orientierung im betrachteten Anwendungsbereich wesentlich erleichtert.

Darüber hinaus werden durch eine entsprechende Interaktion von Ontology Browser mit der Komponente Ontology Manager auch dynamisch ermittelte, abgeleitete Informationen wie transitive Beziehungen oder Grad der

semantischen Kompatibilität zwischen ausgewählten Konzepten dargestellt (vgl. Abbildung 4.1). Diese Zusammenhänge werden nicht zuletzt auch von anderen Komponenten wie Prototyper oder Adapter Manager benötigt, um logisch kompatible Anwendungsfälle auszuwählen oder Adapter für Repräsentationen zu generieren (vgl. Abschnitt 3.6.2 und 3.6.3). Die hierfür erforderliche Implementierung der in Abschnitt 3.3 eingeführten Regeln und Definitionen wird durch die Komponente Ontology Manager zusammengefaßt und über eine geeignete Schnittstelle den anderen Komponenten der Architektur zur Verfügung gestellt. Somit ergibt sich eine klare Lokalisierung der den Anwendungsbereich betreffenden Funktionalität, welche die vorgenommene Trennung zwischen logischer und technischer Ebene des vorgestellten Frameworks wirkungsvoll unterstützt.

Wie aus Abbildung 4.1 ersichtlich, nimmt die Komponente Prototyper eine zentrale Stellung innerhalb der erarbeiteten Referenz-Implementierung ein. Sie verarbeitet die in einer einfachen textuellen Notation verfaßte, vom Benutzer vorgegebene Funktionale Spezifikation (siehe Abschnitt 3.5 und Anhang A.2), setzt diese in die verwendete, interne Repräsentation um (vgl. Abbildung 3.13) und trifft anschließend eine geeignete Auswahl an Softwarekomponenten für deren Anwendungsfälle und primäre Konzepte (siehe Abschnitt 3.6.2). Hierfür wird entsprechende Funktionalität der oben beschriebenen Komponenten CUC-Manager und Ontology Manager genutzt. Im weiteren Verlauf koordiniert Prototyper die erforderlichen Interaktionen zwischen Benutzer, Adapter Manager, Variant Breeder und Variant Generator, um das in Abschnitt 3.6.4 erläuterte Verfahren zur Generierung funktionaler Prototypen weitgehend automatisiert durchzuführen.

So wird zunächst der durch Variant Breeder implementierte Genetische Algorithmus initialisiert, d.h. eine vorgegebene Anzahl an unterschiedlich zusammengesetzten Prototyp-Varianten bzw. deren zugehörige Konstruktionspläne auf zufälliger Basis erstellt (siehe Abschnitt 3.6.5). Je nach Konfiguration der auf diese Weise bestimmten Varianten ist zusätzlich die Integration entsprechender Adapter für verschiedene technische Repräsentationen primärer Konzepte erforderlich. Die Ermittlung oder Generierung dieser Adapter ist Aufgabe der Komponente Adapter Manager, wie später erläutert wird. Nunmehr können einige Iterationen der heuristischen Optimierung durch Variant Breeder durchgeführt werden, wobei eine ausschließlich automatische Bewertung der generierten Varianten vorgenommen wird. Hierbei ist in der Folge u.U. die Integration weiterer Adapter in den zugehörigen Konstruktionsplänen erforderlich, die wiederum durch die Komponente Prototyper in Zusammenarbeit mit Adapter Manager geleistet wird. Schließlich

wird eine Vorauswahl der so optimierten Varianten mit Hilfe der Komponente Variant Generator in Java-Code übersetzt (vgl. Abschnitt 3.6.4), auf Wunsch des Benutzers ausgeführt und anschließend interaktiv bewertet.

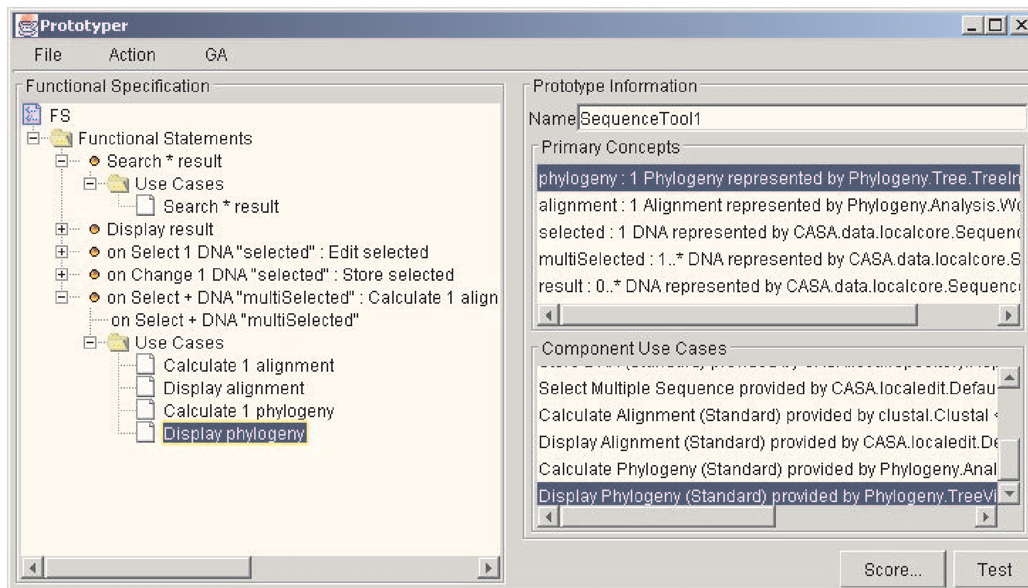


Abb. 4.4: *Bildschirmdarstellung der Komponente Prototyper*

Zu diesem Zweck besitzt Prototyper eine grafische Benutzeroberfläche, die in Abbildung 4.4 exemplarisch nach Auswahl einer generierten Prototyp-Variante für das in Abschnitt 3.6.1 eingeführte Anwendungsbeispiel dargestellt ist. Auf der linken Seite ist die Struktur der Funktionalen Spezifikation aus zusammengefaßten Anwendungsfällen und ihren Auslösern abgebildet, während die rechte Seite der Benutzeroberfläche die jeweilige Zusammensetzung der betrachteten Prototyp-Variante anzeigt. Die Selektion einzelner Elemente der Spezifikation durch den Benutzer führt zur Hervorhebung der entsprechenden Softwarekomponenten und Repräsentationen, die in der betreffenden Variante verwendet werden. Auf diese Weise kann der Zusammenhang zwischen gegebener Spezifikation und erhaltenen Ergebnissen nachvollzogen werden, wie in Abbildung 4.4 durch die Auswahl des Anwendungsfalls `Display 'phylogeny'` illustriert wird (vgl. Abbildung 3.22 und 3.23). Darüber hinaus wird über eine derartige Selektion auch die vom Benutzer durchzuführende Bewertung der gesamten Variante oder ihrer Bestandteile vorgenommen (siehe Abschnitt 3.6.5). Hierfür stellt die Komponente Prototyper entsprechende Mittel zur Ausführung und Beurteilung der angezeigten Variante bereit.

Diese Beurteilung findet als Wert der Funktionen f_{uo} und f_{ui} Eingang in die Gesamtbewertung jeder Variante, die ansonsten eigenständig durch die Komponente Variant Breeder gemäß den in Abschnitt 3.6.5 angegebenen Gleichungen ermittelt wird. Die zur Durchführung der heuristischen Optimierung zusätzlich benötigten Informationen, wie Basisparameter des Genetischen Algorithmus oder getroffene Festlegung der Gewichtungsfaktoren, werden zunächst im Rahmen einer Konfigurationsdatei angegeben, können später im Verlauf des Verfahrens aber auch dynamisch über eine entsprechende Schnittstelle von Variant Breeder verändert werden. Darüber hinaus liefert Variant Breeder zahlreiche statistische Daten über den Ablauf der Optimierung, wie etwa Anzahl der Generationen, durchschnittliche Fitneß-Werte oder Ausmaß und Anteil der Teilbewertungen, so daß Rückschlüsse auf den erzielten Erfolg ermöglicht werden. Dies erlaubt in der Folge eine teilweise manuelle Steuerung der Optimierung sowie den Einsatz unterschiedlicher Strategien zur Ermittlung bestmöglicher Prototyp-Varianten, wie in Abschnitt 3.6.5 ausführlich erläutert wird.

Die benötigten Adapter werden, wie oben bereits erwähnt, durch die Komponente Adapter Manager verwaltet und bei Bedarf bereitgestellt. Hierbei beschränkt sich die Referenz-Implementierung auf binäre Adapter zwischen zwei unterschiedlichen Repräsentationen des gleichen Konzepts des Ontologie. Für diese Repräsentationen ist die jeweils zugehörige CUC-Beschreibung mit einer besonderen, als `Represent` bezeichneten Manipulation anzugeben (siehe Abschnitt 3.6.3). Dies erlaubt der Komponente Adapter Manager durch Interaktion mit der Komponente CUC-Manager eine Abbildung auf den technischen Typ der betreffenden Repräsentation. Außerdem läßt sich hierdurch die im Rahmen der Repräsentation behandelten, zum repräsentierten Konzept in Beziehung stehenden Konzepte des Anwendungsbereichs ermitteln. Daraufhin kann Adapter-Manager überprüfen, ob bereits Adapter mit entsprechender `Adapt`-Manipulation existieren, deren vom Framework vorgegebene Schnittstelle dem jeweils erforderlichen Typ entspricht. In diesem Fall wird der so ermittelte Adapter unmittelbar als Ergebnis der ursprünglich gestellten Anfrage zurückgeliefert.

Ansonsten wird versucht, in Zusammenarbeit mit der Komponente Adapter Generator einen entsprechenden Adapter automatisch zu generieren (vgl. Abbildung 4.1). Hierfür wird im Rahmen der durch Adapter Generator implementierten Strategie auf Zusammenhänge der Ontologie sowie untergeordnete `Get`- und `Set`-Manipulationen der beteiligten Repräsentationen zurückgegriffen, wie in Abschnitt 3.6.3 ausführlich erläutert wird. Möglicherweise sind im Verlauf des Verfahrens weitere Adapter erforderlich, die

wiederum in der oben beschriebenen Weise durch die Komponente Adapter Manager ermittelt werden. Somit ergibt sich ein insgesamt rekursiv organisierter Prozeß, der unter bestimmten Voraussetzungen die automatische Generierung funktionaler Adapter ermöglicht. Andernfalls wird zumindest ein entsprechender Rahmen für den Adapter erstellt, der später vom Benutzer geeignet zu ergänzen ist. Diese Information wird durch die Komponente Adapter Manager übermittelt und von der Komponente Prototyper an den Benutzer weitergegeben bzw. unmittelbar in dem betreffenden Konstruktionsplan vermerkt. Dies erlaubt später eine geeignete Beurteilung durch die Komponente Variant Breeder im Verlauf der heuristischen Optimierung (siehe Gleichung 3.12).

Die zuletzt erforderliche Umsetzung des Konstruktionsplans einer gegebenen Prototyp-Variante in fehlerfrei übersetzbaren Java-Code wird durch die Komponente Variant Generator gemäß den in Abschnitt 3.6.4 aufgeführten Vorgaben geleistet. Hierbei wird vorausgesetzt, daß der betreffende Konstruktionsplan bereits vollständig festgelegt ist, also evtl. zusätzlich benötigte Interaktionen ermittelt sowie sämtliche in Interaktionen referenzierte Rollen und Parameter durch entsprechende Instanzen belegt sind. Die hierfür erforderlichen Anteile des Verfahrens werden durch die Komponente Prototyper implementiert, so daß Variant Generator selbst keine anderen Komponenten der Architektur in Anspruch nimmt (vgl. Abbildung 4.1).

Daher ergibt sich in der Realisierung der Komponente Variant Generator eine weitgehend schematische Umsetzung des in Abschnitt 3.6.4 erläuterten Verfahrens. Dennoch sind zahlreiche, technisch motivierte Detailaufgaben, wie Generierung von Ereignis-Adaptern, Vermittlung unterschiedlicher Kardinalität oder Behandlung von Exceptions in Operationsaufrufen, zu lösen. Darüber hinaus erstellt Variant Generator ein geeignetes Rahmenprogramm und eine einfache grafische Benutzeroberfläche für die jeweilige Variante, so daß sich diese später komfortabel mit Hilfe der Komponente Prototyper ausführen läßt.

Zusammenfassend ist festzustellen, daß Funktionalität und Zusammenspiel der beschriebenen Komponenten weitgehend der durch das Framework vorgegebenen Strukturierung folgen. Daher ergeben sich größtenteils klar abgegrenzte Aufgabenbereiche mit eindeutig definierten Abhängigkeiten zwischen den Elementen der in Abbildung 4.1 dargestellten Architektur. Hierbei wird bereits implementierte Funktionalität, etwa die Komponenten zur Interpretation der eingesetzten textbasierten Formate, so weit wie möglich wiederverwendet. Diese Merkmale erleichtern Implementierung und Weiterentwicklung der vorgestellten Software-Architektur erheblich.

4.3 Zusammenfassung

Die beschriebene Referenz-Implementierung verdeutlicht die praktische Umsetzung des in Kapitel 3 vorgestellten, konzeptuellen Frameworks für komponentenbasiertes Rapid Prototyping. Darüber hinaus erlaubt sie eine empirische Überprüfung der erzielten Ergebnisse. Auch wenn die Qualität der Implementierung nicht ausreichend für einen Einsatz in der Systementwicklung ist, so kann zumindest die erarbeitete Software-Architektur als Basis einer zukünftigen, praxisgerechten Umsetzung des Frameworks dienen.

Sie definiert eine grundlegende Strukturierung der Implementierung hinsichtlich ihrer wesentlichen Komponenten und deren Interaktionen. Die gewählte Architektur folgt den Vorgaben des Frameworks und ermöglicht somit eine klare Trennung der unterschiedlichen Aufgabenbereiche und resultierenden Abhängigkeiten zwischen ihren Bestandteilen. Auf diese Weise werden Realisierung und Weiterentwicklung der Software-Architektur merklich erleichtert.

Die implementierten Komponenten selbst ermöglichen eine durchaus komfortable Bearbeitung der erforderlichen Modelle und Beschreibungen. Darüber hinaus werden alle wesentlichen Verfahren des vorgestellten Frameworks wirkungsvoll unterstützt und im Rahmen eines interaktiven, iterativ organisierten Prozesses umgesetzt. Der hierdurch festgelegte Ablauf sowie die vorgeschlagene Benutzerführung kann ebenfalls als Ausgangspunkt einer zukünftigen Implementierung herangezogen werden.

Die im folgenden Kapitel aufgeführten Erweiterungen und Anpassungen des Frameworks lassen sich größtenteils ohne prinzipielle Schwierigkeiten in die zuvor beschriebene Software-Architektur integrieren, wie später erläutert wird. Diese Eigenschaft ist ein wesentliches Qualitätsmerkmal der vorgestellten Lösung und somit Gegenstand der anschließenden Diskussion.

5. ERWEITERUNGEN

In den vorangegangenen Kapiteln wurde die Problemstellung erläutert, die erforderlichen Grundlagen aufgeführt und als zentrales Ergebnis der vorliegenden Arbeit ein konzeptionelles Framework für komponentenbasiertes Rapid Prototyping vorgestellt. Ein wesentliches Merkmal eines solchen Frameworks ist dessen Erweiterbarkeit und Anpassungsfähigkeit hinsichtlich höherer Qualität der erstellten Prototypen, Integration bestehender Beschreibungstechniken und Werkzeuge, sowie einer insgesamt praxisgerechten Anwendung der erarbeiteten Modelle und Verfahren.

Aus diesem Grund werden im folgenden Kapitel zahlreiche vielversprechende Erweiterungen und Verbesserungen des Frameworks vorgeschlagen, die bestimmte, teilweise maßgeblich durch die Referenz-Implementierung vorgegebene Einschränkungen aufheben und somit das Potential der erarbeiteten Lösung verdeutlichen. Viele dieser Erweiterungen sind aufgrund der geleisteten Vorarbeiten unmittelbar und mit vertretbarem Aufwand in das Framework bzw. dessen Implementierung zu integrieren. Andere Vorschläge oder angesprochene Lösungsansätze betreffen übergreifende Anteile der Konzeption und erfordern daher einen voraussichtlich deutlich höheren Aufwand für ihre Umsetzung, wie später diskutiert wird.

In jedem Fall gibt das in Kapitel 3 erläuterte Framework eine klare Strukturierung der grundlegenden Aufgabenbereiche vor, die deshalb auch im folgenden zur Gliederung der aufgeführten Vorschläge herangezogen wird. So werden zunächst mögliche Erweiterungen im Bereich der Ontologie zur Modellierung des Anwendungsbereichs vorgestellt. Anschließend werden komponentenbezogene Anwendungsfälle und Funktionale Spezifikation gemeinsam betrachtet, da ihre Beschreibung von angebotener oder erwünschter Funktionalität in einem engen logischen Zusammenhang zu verstehen ist. Schließlich werden Verbesserungen bei der eigentlichen Generierung funktionaler Prototypen vorgeschlagen, die zur Steigerung der Qualität der erhaltenen Ergebnisse beitragen. Zuletzt werden weitreichende, übergreifende Weiterentwicklungen des Frameworks diskutiert und die wesentlichen Erkenntnisse am Ende des Kapitels zusammengefaßt.

5.1 Ontologie

Die grundlegende Zielsetzung der Ontologie ist eine eindeutige, der übergeordneten Problemstellung angemessene Modellierung des Anwendungsbereichs, wie in Abschnitt 3.3 ausführlich erläutert wird. Das erstellte Modell dient als Referenz zur anwendungsbezogenen Beschreibung von Funktionalität als Manipulation von Konzepten der Ontologie (siehe Abschnitt 3.4 und 3.5). Darüber hinaus können mit ihrer Hilfe unter bestimmten Voraussetzungen benötigte Adapter für unterschiedliche Repräsentationen automatisch generiert werden (siehe Abschnitt 3.6.3). Zuletzt stellt die Ontologie selbst ein bedeutendes Produkt der Softwareentwicklung dar, daß auch zur Lösung anderer Aufgaben, wie Anforderungsanalyse oder Dokumentation des späteren Systems, eingesetzt werden kann.

Dementsprechend lassen sich die möglichen Erweiterungen in diesem Bereich des Frameworks nach Verbesserung der Expressivität, Ableitung weiterführender Informationen, sowie Erleichterungen bei Erstellung, Pflege und Benutzung der Ontologie unterscheiden. So wird zunächst die Einführung von geeignet formulierten Beschränkungen (engl. *constraint*) auf den erstellten Modellen des Anwendungsbereichs vorgeschlagen, um deren Genauigkeit zu verbessern. Anschließend wird die Integration und systematische Behandlung von Manipulationen als eigenständiges Element des Metamodells erläutert. Weiterhin sind gerade bei Bearbeitung und Austausch der Ontologie wesentliche Verbesserungen der Werkzeugunterstützung naheliegend. Schließlich werden weiterführende Fragestellungen im Zusammenhang mit unabhängig entwickelten Ontologien untersucht.

5.1.1 Constraints

Die in Abschnitt 3.3 eingeführte, objekt-orientierte Modellierung einer Ontologie ermöglicht in der Regel eine kompakte und intuitiv verständliche Abbildung des betrachteten Anwendungsbereichs. Gerade die Semantik der Generalisierungsbeziehung zwischen Konzepten erleichtert die übersichtliche Repräsentation von durchaus komplexen Verhältnissen im Anwendungsbereich, da sämtliche Relationen des übergeordneten, allgemeinen Konzepts implizit auf dessen Spezialisierungen übertragen werden. In bestimmten Fällen führt diese einfache Übertragung jedoch zu unerwünschten oder sogar falschen Ergebnissen, die in der Folge auch tatsächlich unsinnige Aussagen über Zusammenhänge der Ontologie erlauben.

Abbildung 5.1 verdeutlicht dieses Problem am Beispiel einer möglichen Beziehung hasSite zwischen den Konzepten Protein und Site in der Domäne Biochemistry. Sie modelliert den zunächst durchaus plausiblen Sachverhalt,

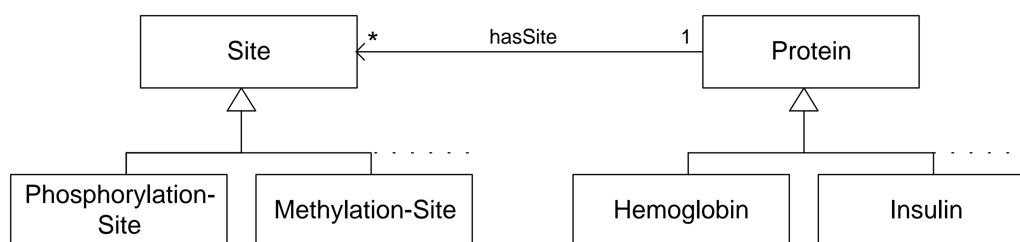


Abb. 5.1: *Problematische Generalisierung in der Domäne* Biochemistry

daß ein gegebenes Protein im allgemeinen bestimmte, ausgezeichnete Positionen aufweist, an denen nach dessen Biosynthese in der Zelle eine enzymatisch bedingte, biochemische Modifikation stattfindet [Str88]. Diese Veränderungen sind üblicherweise von großer Bedeutung für dessen biologische Funktion. Daher repräsentieren die im Modell als Site bezeichneten Positionen maßgebliche Informationen über das betreffende Protein.

Aufgrund der Übertragung dieser Beziehung auf alle, möglicherweise transitiven Spezialisierungen des Konzepts Protein, sind nunmehr jedoch *alle möglichen Kombinationen* mit Spezialisierungen des Konzepts Site prinzipiell zulässig. So ist im Rahmen des in Abbildung 5.1 gezeigten Beispiels implizit auch eine entsprechende Beziehung zwischen den Konzepten Hemoglobin und Methylation-Site definiert, obwohl das Protein Hämoglobin in der Zelle tatsächlich keine derartige Modifikation aufweist.

Eine ad-hoc Lösung dieses Problems betrachtet die im Beispiel eingeführte Beziehung zwischen Protein und Site als einen Fehler der Modellierung, welcher durch ihre Entfernung aus dem Modell sowie Einführung neuer Beziehungen zwischen den jeweiligen Spezialisierungen zu korrigieren ist. In der Praxis führt dieses Vorgehen jedoch zu zahlreichen, manuell zu erstellenden Beziehungen, die den ursprünglich modellierten Sachverhalt nicht prägnant wiedergeben. Aus diesem Grund führt das Metamodell der im TAMBIS-Projekt [BBB+99] erstellten, als Grundlage dieser Arbeit verwendeten Ontologie das Element der sog. *Sanction* ein [RBG+97]. Eine explizit angegebene Sanction für Relationen zwischen Konzepten modelliert, daß *mögliche* Beziehungen ihrer übergeordneten Konzepte auch tatsächlich für die betrachteten Spezialisierungen *gültig* sind. Im oben aufgeführten Beispiel werden also zusätzliche Sanctions zwischen denjenigen Spezialisierungen von Protein und Site eingeführt, bei denen in der Realität entsprechende Positionen im jeweiligen Protein auftreten. Zwischen den Konzepten Hemoglobin und Methylation-Site wird somit *keine* entsprechende Sanction definiert.

Diese zweistufige Vorgehen bei Definition von Relationen innerhalb der

Ontologie ist jedoch aufwendig und führt gerade bei umfangreichen Modellen zu zahlreichen erforderlichen Sanctions für enthaltene Beziehungen zwischen Konzepten. Die vorliegende Arbeit schlägt deshalb eine verallgemeinerte Lösung des oben beschriebenen Problems vor, bei der lediglich *Ausnahmen* von übertragenen Relationen durch entsprechend formulierte Beschränkungen angegeben werden. Ein solches Constraint ist allgemein als Prädikat über Konzepte und Relationen der Ontologie zu verstehen, das mit ausgewählten Konzepten assoziiert und bei Ermittlung abgeleiteter Informationen ausgewertet wird. Im Rahmen des in Abbildung 5.1 gezeigten Beispiels kann ein geeigneter Constraint mit dem Konzept Hemoglobin verbunden werden, um auszudrücken, daß in diesem Fall eine Beziehung hasSite mit dem Konzept Methylation-Site nicht zulässig ist.

Ein derartiger Ansatz erfordert voraussichtlich erheblich weniger Aufwand zur Modellierung des Anwendungsbereichs, da Beziehungen von allgemeinen Konzepten üblicherweise eingeführt werden, um *gemeinsame Merkmale* ihrer Spezialisierungen kompakt zu repräsentieren. Somit sind diesbezügliche Ausnahmen nur bei einer verhältnismäßig geringen Anzahl von untergeordneten Konzepten zu erwarten. Darüber hinaus können Constraints als allgemeine Prädikate über Konzepte und Relationen auch zur Beschreibung anderer Konsistenzbedingungen im Anwendungsbereich verwendet werden. Beispielsweise läßt sich mit ihrer Hilfe ausdrücken, daß jede Ausprägung des Konzepts NucleicAcid innerhalb der Domäne SequenceAnalysis über die Relation hasName durch genau einen, eindeutig bestimmten Bezeichner identifiziert wird (vgl. Abbildung 3.18). Solche Informationen können im weiteren Verlauf des Verfahrens, etwa bei automatischer Generierung von Adaptern für unterschiedliche Repräsentationen, angemessen berücksichtigt werden.

Zur systematischen Integration von Constraints in das vorgestellte Metamodell einer Ontologie (siehe Abbildung 3.5) ist eine entsprechende Syntax ihrer Beschreibung sowie eine zugrundeliegende Semantik ihrer Auswertung zu definieren. Aufgrund der gewählten, objekt-orientierten Modellierung liegt es nahe, hierfür bestehende Ansätze, etwa die *Object Constraint Language* [OMG99] der UML, zu verwenden bzw. geeignet anzupassen. Dies verringert den erforderlichen Aufwand, erleichtert die Einarbeitung bei Erstellung einer Ontologie und sichert die Unterstützung existierender Werkzeuge.

Andererseits ist der Zugewinn an Expressivität und Genauigkeit durch die Einführung von Constraints grundsätzlich auch mit einer Erhöhung der Komplexität des gesamten Ansatzes verbunden. Dies betrifft sowohl die Erstellung der Ontologie, als auch deren Pflege und Benutzung, gerade falls zahlreiche Constraints zur Laufzeit auszuwerten und hinsichtlich ihrer Wechselwirkung zu untersuchen sind. Daher sind zumindest methodische Hinweise für den angemessenen Einsatz von Constraints zu entwickeln.

5.1.2 Integration von Manipulationen

Eine weitere bedeutende Aufgabe im Bereich der Ontologie ist die methodische Integration der in Anwendungsfällen referenzierten Manipulationen. Sie repräsentieren in Kombination mit Konzepten der Ontologie das zentrale Modell zur Beschreibung von angebotener oder erwünschter Funktionalität (siehe Abschnitt 3.4 und 3.5). Tatsächlich setzt das Framework eine bekannte Semantik für wenige, ausgezeichnete Manipulationen, mit Namen wie `Represent`, `Set`, `Get` oder `Adapt`, voraus. Diese werden dementsprechend in besonderer Weise zur Generierung funktionaler Prototypen herangezogen, wie in Abschnitt 3.6 erläutert wird. Dennoch ist im allgemeinen eine durchgängige und systematische Behandlung sämtlicher Manipulationen anzustreben.

Hierfür ist es naheliegend, Manipulationen als eigene Bestandteile der Domänen einer Ontologie zu definieren. Diese können bestehenden Konzepten einer gegebenen Domäne zugeordnet werden, um die jeweils *erlaubten* Manipulationen des betreffenden Konzepts festzulegen. In der Folge werden bei Erstellung einer CUC-Beschreibung oder Anwendungsfällen der Funktionalen Spezifikation nurmehr die so bestimmten Kombinationen aus Manipulation und Konzept zugelassen. Diese Einschränkung des verfügbaren Vokabulars führt zu einer verbesserten Konsistenz bei Beschreibung von Funktionalität sowie einer vereinfachten Zuordnung bei späterer Komponenten-Auswahl (siehe Abschnitt 3.6.2).

Ein solches Vorgehen entspricht in gewisser Hinsicht dem objekt-orientierten Paradigma, zusammengehörige Struktur und Verhalten nach logischen Gesichtspunkten gemeinsam als Einheit zu modellieren [Mey97]. Allerdings wird im vorliegenden Ansatz das tatsächlich beobachtbare Verhalten durch die später für eine gegebene Manipulation ausgewählte Softwarekomponente bestimmt. Diese Komponente ist möglicherweise identisch mit der Repräsentation des manipulierten Konzepts, kann also als *Objekt* im Sinne der Objekt-Orientierung aufgefaßt werden, jedoch wird dieser Sachverhalt nicht zwingend durch die so modellierte Ontologie vorgegeben. Daher erlaubt eine derartige Integration von Manipulationen in das Metamodell den fließenden Übergang zwischen objekt-orientierter und komponentenbasierter Softwareentwicklung.

Aus pragmatischen Gründen werden bereits definierte Manipulationen durch andere Domänen importiert, falls der Charakter der betreffenden Manipulation durch den neuen Kontext nicht entscheidend verändert wird. Beispielsweise können Manipulationen mit Bezeichnungen wie `Edit`, `Display` oder `Select` in einer eigenen Domäne Graphical User Interface aufgeführt werden. Diese lassen sich anschließend in anderen Domänen referenzieren, um

eine interaktive Darstellung oder Bearbeitung ihrer beinhalteten Konzepte zu beschreiben.

Schließlich können auch Beziehungen zwischen Manipulationen in das Metamodell der Ontologie aufgenommen werden, falls ihre zugrundeliegende, vordefinierte Bedeutung weiterführende Ableitungen über die beschriebene Funktionalität ermöglicht. So läßt sich etwa eine gerichtete Beziehung implies zwischen `Edit`- und `Display`-Manipulation definieren, um auszudrücken, daß eine in Kombination mit `Edit` spezifizierte Funktionalität in jedem Fall auch eine entsprechende, `Display`-bezogene Funktionalität impliziert. Dies führt in der Folge zu einer Verbesserung der Toleranz bei Komponenten-Auswahl und Verknüpfung, wobei die bestehende Auffassung von logischer Kompatibilität zwischen Anwendungsfällen geeignet zu erweitern ist (siehe Definition 3.6 und Abschnitt 5.3.1).

Die oben aufgeführten Beispiele verdeutlichen, daß eine systematische Integration von Manipulationen in das Metamodell der Ontologie zahlreiche Vorteile bei der Generierung funktionaler Prototypen verspricht. Demgegenüber ist der sich ergebende, zusätzliche Aufwand bei Erstellung und Pflege der Ontologie zu vernachlässigen. Die Integration selbst kann durch einfache Erweiterungen des in Abbildung 3.5 dargestellten Metamodells erreicht werden, wobei die in Abschnitt 3.6 beschriebenen Verfahren zur Prototyp-Generierung entsprechend anzupassen sind.

5.1.3 Beschreibungstechnik und Werkzeugunterstützung

Der Umfang und die zu erwartende Komplexität einer aussagekräftigen Ontologie für den jeweils betrachteten Anwendungsbereich erfordern zwangsläufig leistungsfähige Beschreibungstechniken und Werkzeuge zur Erstellung, Bearbeitung, Validierung und Pflege eines solchen Modells. Die Vielfalt an unterschiedlichen Konzepten, abgeleiteten Beziehungen, anwendbaren Interpretationen, sowie die mögliche, zukünftige Integration von Constraints und Manipulationen (siehe Abschnitt 5.1.1 und 5.1.2) sind ohne geeignete Beschreibungstechniken und eine auf sie abgestimmte Werkzeugunterstützung kaum zu beherrschen.

UML und CASE-Werkzeuge

Zunächst sollte die in dieser Arbeit vorgestellte, textuelle Notation zur Beschreibung einer Ontologie (siehe Anhang A.1) im Sinne einer praxisgerechten Umsetzung des Frameworks durch eine überwiegend grafische Notation ersetzt werden. Dies erlaubt ein weitgehend intuitives Verständnis der grundlegenden Zusammenhänge innerhalb der Ontologie, das durch den zweckmäßi-

gen Einsatz von Symbolen, Typographie oder Farbgebung noch zusätzlich unterstützt werden kann.

Aufgrund der gewählten, objekt-orientierten Modellierung einer Ontologie ist es naheliegend, hierfür eine an das Klassendiagramm der UML angelehnte Notation einzusetzen bzw. diese Beschreibungstechnik mit Mitteln der UML geeignet anzupassen [RJB98]. Schließlich können Konzepte der Ontologie durchaus als Klassen aufgefaßt werden, während bestimmte, vorgegebene Relationen der Ontologie, wie Generalisierung, Aggregation oder einfache Assoziationen in vergleichbarer Weise auch im Rahmen der UML definiert sind. So erinnert die in dieser Arbeit zur Illustration eingesetzte Darstellung beispielhafter Ausschnitte einer Ontologie bereits deutlich an entsprechende UML-Diagramme (vgl. Abbildung 3.18). Lediglich die im Metamodell der Ontologie neu hinzugekommenen Relationen, wie Äquivalenz, Alternative und Interpretation, sind durch eigene grafische Symbole im Klassendiagramm zu ergänzen. Demgegenüber können die nicht grafisch repräsentierten Informationen über zugehörige Subinterpretationen oder Constraints als textuelle Annotation von Elementen des Diagramms angegeben werden.

Durch dieses Vorgehen ist zumindest die eingesetzte Notation für durchschnittlich ausgebildete Softwareentwickler vertraut, auch wenn eigentlicher Inhalt und Bedeutung einer Ontologie sich erst bei genauer Kenntnis des Anwendungsbereichs erschließen. Darüber hinaus können in der Folge vorhandene, ausgereifte Werkzeuge zur Erstellung von UML-Modellen auch zur Bearbeitung einer Ontologie herangezogen werden. Derartige CASE-Werkzeuge werden ohnehin bereits in vielen Bereichen der Softwareentwicklung eingesetzt [Tog01, Rat01b], so daß ein insgesamt geringer Einarbeitungsaufwand sowie eine in den übergeordneten Entwicklungsprozeß integrierte Behandlung der erstellen Modelle gewährleistet ist.

Allerdings muß die Funktionalität der so verwendeten CASE-Werkzeuge erweitert oder durch neu erstellte Komponenten ergänzt werden, um den dynamischen Aspekten einer Ontologie gerecht zu werden. Schließlich ergeben sich durch Anwendung der in Abschnitt 3.3 aufgeführten Regeln und Definitionen sowie der Auswertung zukünftig eingeführter Constraints (siehe Abschnitt 5.1.1) zahlreiche, nicht unmittelbar aus der Struktur ersichtliche Zusammenhänge. Die hierdurch erhaltenen Informationen über transitive Beziehungen, zugehörige Subinterpretationen, semantische Kompatibilität zwischen Konzepten oder verletzte Konsistenzbedingungen sind geeignet zu ermitteln und dem Benutzer übersichtlich zusammengefaßt darzustellen. Dies erleichtert Erstellung und Pflege der Ontologie, ermöglicht letztlich aber auch die Validierung der vorgenommenen Modellierung des Anwendungsbereichs. Somit können auftretende Fehler oder Ungenauigkeiten frühzeitig erkannt und korrigiert werden.

Integration unabhängig entwickelter Ontologien

Ein letzter bedeutender Aufgabenbereich für Werkzeugunterstützung betrifft Austausch und Integration von Ontologien unterschiedlicher Herkunft. Dies ermöglicht die Wiederverwendung der erstellten Modelle des Anwendungsbereichs und trägt somit wesentlich zum praktischen Erfolg des vorgestellten Ansatzes bei. Darüber hinaus ist es aus methodischen Gründen in bestimmten Fällen vorteilhaft, eine vorausgesetzte Ontologie dem ausführbaren Format einer Softwarekomponente beizufügen. Hierdurch kann die vorhandene CUC-Beschreibung der betreffenden Komponente mit dem zugehörigen Kontext im Anwendungsbereich bereits eindeutig assoziiert werden. Somit entfällt die Voraussetzung einer zentralen, universell anerkannten Ontologie zugunsten eines dezentralen, lose gekoppelten Verfahrens mit lokalen Modellen des jeweiligen Anwendungsbereichs. Diesen potentiellen methodischen Vorteilen stehen allerdings bedeutende Probleme bei der praktischen Umsetzung gegenüber, wie später erläutert wird.

Der eigentliche Austausch einer einmal erstellten Ontologie kann ohne Schwierigkeiten über eine entsprechend definierte, beispielsweise XML-basierte Syntax [Hol00] erfolgen. Dieser Ansatz wird im Rahmen der Referenz-Implementierung bereits erfolgreich eingesetzt, um erstellte CUC-Beschreibungen den vorhandenen Softwarekomponenten zuzuordnen (siehe Abschnitt 4.2). Die Definition der verwendeten Syntax im Rahmen einer *Document Type Definition* [XML01] folgt hierbei zweckmäßigerweise der Struktur des in Abbildung 3.5 dargestellten Metamodells einer Ontologie. Somit werden Instanzen der Klassen des Metamodells, also beispielsweise in der Ontologie aufgeführte Konzepte und Relationen, auf XML-Elemente abgebildet, während Assoziationen zwischen diesen Instanzen durch entsprechende Attribute der XML-Elemente repräsentiert sind (vgl. Anhang A.3). Die Umsetzung einer gegebenen Ontologie in die so definierte Syntax ist weitgehend schematisch und kann durch ein geeignetes Werkzeug implementiert werden[1].

In jedem Fall ist ein gemeinsames Verständnis der Syntax und Semantik einer derartigen Beschreibung zwingend erforderlich, um die so ausgetauschten Ontologien auch tatsächlich nutzen zu können. Hierfür muß das erhaltene XML-Dokument eingelesen, interpretiert und wieder in das ursprünglich verwendete Metamodell übersetzt werden. Je nach Umfang der Ontologie und möglichen Überschneidungen mit dem bereits vorhandenen Modell des Anwendungsbereichs ergeben sich hierbei typische Probleme

[1] Falls die vollständige Integration des Metamodells in die UML erreicht wird, wie oben beschrieben ist, so kann offensichtlich auf existierende Werkzeuge zum Export von UML-Modellen in das von der OMG vorgeschlagene XMI-Format [OMG01d] zurückgegriffen werden.

mit der Konsistenz der unabhängig erstellten Teilmodelle, etwa verschiedene Namen für gleiche Konzepte oder unterschiedlich definierte Beziehungen. Diese Probleme lassen sich auf technischer Ebene in der Regel jedoch verhältnismäßig einfach durch den Einsatz geeigneter Versionsmanagement-Werkzeuge lösen [Con01, Rat01a]. Darüber hinaus erlaubt das in Abschnitt 3.3 vorgestellte Metamodell eine Strukturierung der Ontologie in Domänen als modulare Einheiten, die auch getrennt bearbeitet und genutzt werden können.

Dennoch ist im allgemeinen Fall bei Integration unabhängig erstellter Ontologien des gleichen Anwendungsbereichs mit einem erheblichen manuellen Aufwand zu rechnen. Dies gilt insbesondere für den Übergang von einer zentral genutzten, standardisierten Ontologie hin zu einer vollständig verteilten, lokal durchgeführten Spezifikation angebotener Funktionalität. Ein derartiger Ansatz verspricht einige methodische Vorteile für die praktische Anwendung des vorgestellten Frameworks, gerade im Hinblick auf das zentrale Paradigma der komponentenorientierten Softwareentwicklung (siehe Abschnitt 2.2 und 3.1.3). So führt der Bezug auf eine lokal entwickelte und bei Verteilung der Softwarekomponente mitgelieferten Ontologie zu einer in sich geschlossenen Form der Wiederverwendung von Funktionalität. Hierbei erlauben die beigefügten Informationen über Signatur *und* Semantik ein umfassendes Verständnis der bereitgestellten Komponente.

Abbildung 5.2 verdeutlicht den so beschriebenen Übergang an Hand eines schematischen Beispiels der Zusammenhänge. Im oberen Teil der Abbildung sind die gegenwärtig durch das Framework vorausgesetzten Verhältnisse dargestellt. Die vorhandenen Komponenten zweier unterschiedlicher Hersteller A und B verfügen über begleitende CUC-Beschreibungen, deren logische, anwendungsbezogene Anteile sich auf Elemente einer gemeinsamen Ontologie beziehen (siehe Abschnitt 3.4). Diese Annahme ist durchaus plausibel, weil mittelfristig geeignete CUC-Beschreibungen für Komponenten ohnehin vorerst ausschließlich durch den Benutzer des Frameworks erstellt werden müssen (vgl. Abschnitt 3.7). Dieses Vorgehen sichert die Verwendung einer Vielzahl an existierenden Komponenten und gewährleistet die Konsistenz der erstellten Beschreibungen.

Im unteren Teil der Abbildung sind die entsprechenden Verhältnisse bei unterschiedlichen, jeweils lokal genutzten Ontologien dargestellt. Hierbei beziehen sich die CUC-Beschreibungen auf die weitgehend unabhängig entwickelten Ontologien des betreffenden Herstellers. Ein späterer Benutzer der so beschriebenen Komponenten erhält neben dem ausführbaren Format sowie den zugehörigen CUC-Beschreibungen auch die jeweils eingesetzte Ontologie des betrachteten Anwendungsbereichs. Die Teilmodelle werden anschließend zu einer gemeinsamen, übergeordnet gültigen Ontologie integriert, um die

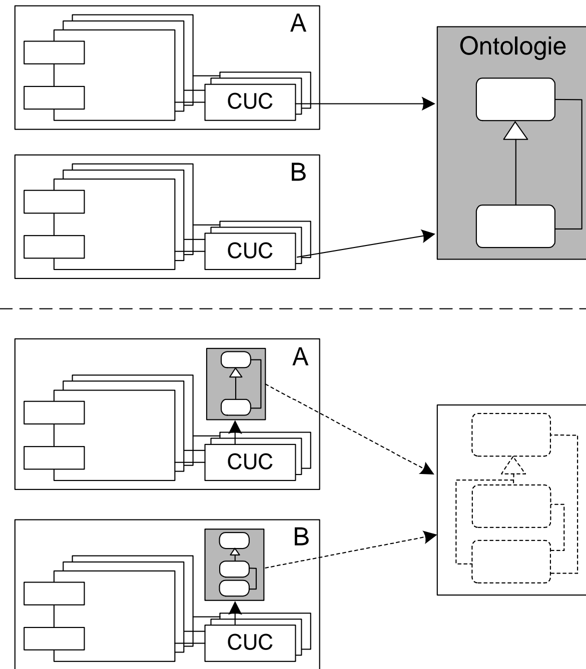


Abb. 5.2: *Gemeinsame und lokal definierte Ontologie*

jeweiligen Komponenten mit den Verfahren des vorgestellten Frameworks zu verknüpfen. Dies erfordert letztlich eine entsprechende Abbildung zwischen den betreffenden Ontologien, die semantisch äquivalente Konzepte, Relationen und Manipulationen zuverlässig identifiziert, wie in Abbildung 5.2 durch unterbrochene Pfeile und Umrißlinien angedeutet ist.

Eine derartige Abbildung ist in der Regel schwierig zu bestimmen, insbesondere wenn eine möglichst weitgehende Automatisierung des gesamten Verfahrens angestrebt wird. Schließlich sind in der Praxis zahlreiche Unterschiede und Besonderheiten der Modellierung zu erwarten, beispielsweise verschiedene Bezeichner für identische Konzepte und Relationen, divergente Struktur des Beziehungsgeflechts, oder eine heterogene Granularität der erstellten Modelle. Andererseits beschreiben alle beteiligten Ontologien den selben Anwendungsbereich mit seinen aus den realen Verhältnissen wohlbekannten Konzepten und deren Beziehungen. Außerdem ist das vorgestellte Metamodell einer Ontologie, etwa im Vergleich zu objekt-orientierten Klassenmodellen, verhältnismäßig einfach und übersichtlich einzuschätzen. Schließlich erlaubt der Verzicht auf technische Anteile eine Bestimmung semantisch ähnlicher Konzepte allein über für sie definierte Beziehungen, wie in Abschnitt 3.3 erläutert wird. Aus diesen Gründen erscheint eine zumindest

partielle Lösung dieser Problemstellung möglich, gerade falls für besonders kritische, initiale Schritte eine interaktive Hilfestellung durch den Benutzer angenommen werden kann.

Die Integration unterschiedlicher Modelle des gleichen Anwendungsbereichs ist letztlich auch für andere Bereiche der Softwareentwicklung von Bedeutung. Ein derartiges Modell der Realität beschreibt eine bestimmte Sicht auf die Umgebung, in der ein Softwaresystem dem Anwender Unterstützung bei alltäglichen Aufgaben und Tätigkeiten anbietet. Gerade zu Beginn der Softwareentwicklung sind während der Analyse zahlreiche unterschiedliche Sichten zu berücksichtigen, um die gestellten Anforderungen an das System möglichst vollständig zu ermitteln und zu dokumentieren [Wie99]. Diese grundlegende Aufgabe des Requirement Engineering wird durch geeignet formalisierte Beschreibungen des Anwendungsbereichs erleichtert. In diesem Zusammenhang kann das entwickelte Metamodell einer Ontologie in Verbindung mit einem Ansatz zur Integration unterschiedlicher Modelle einen wertvollen eigenständigen Beitrag leisten.

5.2 Funktionale Spezifikation und Component Use Cases

Die in Abschnitt 3.4 und 3.5 eingeführte Modellierung von angebotener bzw. erwünschter Funktionalität repräsentiert einen zentralen Bestandteil der anwendungsbezogenen Ebene des vorgestellten Frameworks. Die Beschreibung von Funktionalität als Manipulation von Konzepten des Anwendungsbereich ermöglicht den toleranten, flexibel anpaßbaren Abgleich zwischen Anwendungsfällen der Spezifikation und komponentenbezogenen Anwendungsfällen der verfügbaren Softwarekomponenten (siehe Abschnitt 3.6.2). Demgegenüber erlauben die technischen Anteile eines CUC in Verbindung mit den operativen Elementen der Spezifikation eine effektive Verknüpfung von Komponenten unterschiedlicher Herkunft, wie in Abschnitt 3.6 erläutert wird.

Allerdings ist die gegenwärtig vorgeschlagene Modellierung bewußt auf deren wesentliche Elemente beschränkt, um eine vereinfachte praktische Validierung der erzielten Ergebnisse durch die Referenz-Implementierung zu erreichen. Daher sind zunächst Erweiterungen wünschenswert, die zu einer Verbesserung der Expressivität bei Beschreibung von Funktionalität auf anwendungsbezogener Ebene führen. Hierbei ist die Zielsetzung in erster Näherung durch die Möglichkeiten der natürlichen Sprache bei Formulierung initialer funktionaler Anforderungen vorgegeben. Andererseits ist bei Einführung entsprechender Erweiterungen auf die praktische Umsetzung im Rahmen eines weitgehend automatisierten Verfahrens zu achten.

Auf technischer Ebene des Frameworks ist die Realisierung bzw. Verknüpfung von Anwendungsfällen auf eine weitgehend sequentielle Abfolge beschränkt. Diese Einschränkung kann offensichtlich durch Einführung weiterer operativer Elemente, wie Iteration, Fallunterscheidung, u.ä., ohne konzeptuelle Schwierigkeiten aufgehoben werden. Trotzdem ist auch in diesem Bereich auf die praktische Umsetzung sowie die übergeordnete Zielsetzung des vorgestellten Ansatzes zu achten. Schließlich besitzt das unterstützte Prototyping grundsätzlich explorativen Charakter und dient somit ausschließlich der Ermittlung und Konkretisierung endgültiger funktionaler Anforderungen an das zu entwickelnde System (siehe Abschnitt 1.2 und 2.2).

Zuletzt können die in Abschnitt 3.4 und 3.5 eingesetzten Beschreibungstechniken für komponentenbezogene Anwendungsfälle und Funktionale Spezifikation in Teilen durch eine besser geeignete, intuitiv verständliche grafische Notation ersetzt werden. Hierbei lassen sich bestehende, durch die UML bereitgestellte Lösungen verwenden oder anpassen, so daß ausgereifte Werkzeuge zu ihrer Erstellung und Bearbeitung herangezogen werden können. Dies erleichtert die praktische Anwendung des vorgestellten Ansatzes und ermöglicht eine vereinfachte Integration in den übergeordneten Prozeß der Softwareentwicklung.

5.2.1 *Logische Anteile der Modellierung*

Eine wichtige Zielsetzung der Modellierung von Anwendungsfällen auf logischer Ebene des Frameworks ist die möglichst verständliche und ausdrucksvolle Formulierung von erwünschter oder angebotener Funktionalität. Hierfür wird im vorliegenden Ansatz eine Kombination aus aktiven und passiven Elementen eingeführt, die einem einfachen Hauptsatz der natürlichen Sprache nachempfunden ist. In diesem Zusammenhang repräsentiert eine Manipulation das Verb, während das manipulierte Konzept als direktes Objekt des Satzes aufgefaßt werden kann. Hingegen ist das Subjekt des Satzes implizit durch die jeweils beschriebene Komponente gegeben, wie in Abschnitt 3.4 erläutert wird.

Im Rahmen dieser grammatikalischen Interpretation eines Anwendungsfalls besteht eine offensichtliche Erweiterung in der Einführung zusätzlicher, primärer Konzepte, die als ergänzende Objekte über ausgewählte Präpositionen mit dem unmittelbaren Objekt des Satzes kombiniert werden. Auf diese Weise kann Funktionalität genauer charakterisiert werden, beispielsweise durch Anwendungsfälle wie Search DNA by Name oder Calculate Alignment from NucleicAcid. Allerdings muß hierfür die vorausgesetzte Semantik der möglichen Präpositionen definiert und während der Generierung funktionaler Prototypen geeignet berücksichtigt werden. Um die entstehende Komplexität

zu begrenzen, erscheint es sinnvoll, nur jeweils eine Teilmenge der bekannten Präpositionen in Abhängigkeit der benutzten Manipulation sowie des manipulierten Konzepts tatsächlich zu verwenden. Diese Zuordnung kann im Zuge der Integration von Constraints und Manipulationen in das Metamodell der Ontologie erfolgen (vgl. Abschnitt 5.1.1 und 5.1.2).

Durch die Einführung zusätzlicher primärer Konzepte in der Modellierung eines Anwendungsfalls ergibt sich die Notwendigkeit, zwischen mehreren Bezügen auf das gleiche Konzept zu unterscheiden. Hierfür ist es zweckmäßig, qualifizierende Namen als optionaler Bestandteil der Beschreibung zu definieren. Somit können Anwendungsfälle wie Compare Phylogeny reference with Phylogeny target formuliert werden, die unterschiedliche Rollen der beteiligten Konzepte beinhalten. Diese Festlegung entspricht auf technischer Ebene den Bezeichnern für formale Parameter einer Operation, auch wenn ein Anwendungsfall tatsächlich durch eine Interaktion zwischen mehreren Komponenten realisiert wird (siehe Abschnitt 3.4).

Die Interpretation eines Anwendungsfalls als Hauptsatz einer natürlichen Sprache führt offensichtlich zu weiteren denkbaren Ergänzungen, wie charakterisierende Adverbien oder Adjektive, die zu einer genaueren Beschreibung von Funktionalität herangezogen werden können. Gegenüber den oben aufgeführten Erweiterungen erfordern derartige Anpassungen jedoch einen deutlich höheren Aufwand bei Integration in das übergeordnete Framework. Beispielsweise fehlen gegenwärtig entsprechende Regeln und Definitionen für die Bestimmung der semantischen Kompatibilität entsprechend attributierter Konzepte und Manipulationen. Außerdem ist die resultierende Komplexität bei praktischer Umsetzung des Verfahrens nicht zu unterschätzen. Dennoch liefert der grundlegende Ansatz wertvolle Anregungen für zukünftige Erweiterungen und Verbesserungen.

5.2.2 Technische Anteile der Modellierung

Auf technischer Ebene des Frameworks ist die Realisierung angebotener Funktionalität im Rahmen von CUC-Beschreibungen auf eine sequentielle Folge von Operationsaufrufen beschränkt (siehe Abschnitt 3.4 sowie Abbildung 3.9). Diese vereinfachte Auffassung einer Interaktion zwischen Komponenten erlaubt bereits die effektive Generierung zahlreicher, nicht-trivialer Prototypen, wie nicht zuletzt das durchgängige Anwendungsbeispiel belegt.

Dennoch ist im allgemeinen die Unterstützung weiterer, komplexer Interaktionsmuster wünschenswert. Daher bietet sich zunächst die Einführung von Elementen wie *Iteration* und *Fallunterscheidung* an, die in Abhängigkeit einer auswertbaren *Bedingung* den Ablauf der Interaktion bestimmen. Diese operativen Elemente können offensichtlich im Rahmen jeder technischen Platt-

form auf einfache Weise umgesetzt werden. Allerdings ist zu untersuchen, wie der jeweils aktuelle Zustand der zugehörigen Bedingung tatsächlich ermittelt wird. Neben expliziten Eingaben des Benutzers zur Laufzeit des Prototypen können hierfür v.a. Ergebnisse der Ausführung von Anwendungsfällen herangezogen werden. Beispielsweise kann die Suche nach DNA-Sequenzen in verschiedenen Datenbanken solange wiederholt werden, bis die Homologie mit einer bekannten Sequenz einen vorgegebenen Wert überschreitet (vgl. Abschnitt 2.3).

In jedem Fall ist das in Abbildung 3.9 dargestellte Modell der technischen Anteile eines CUC um die neu hinzugekommenen Elemente zu erweitern. So ist dieser Bereich des Frameworks auf entsprechende Erweiterungen zumindest formal als Spezialisierung der Klasse Interaction vorbereitet. Auf diese Weise können auch andere, ausdrucksvolle Beschreibungen für Verhalten, wie beispielsweise Zustandsautomaten [HU79, Har87], in den vorgestellten Ansatz integriert werden, sofern eine effektive Generierung und Bewertung ausführbarer Prototyp-Varianten gewährleistet ist. Je nach Komplexität der eingesetzten Teilmodelle zur Beschreibung von Verhalten sowie dem gewünschten Grad der Automatisierung ist jedoch ein erheblicher, zusätzlicher Aufwand bei Integration und praktischer Umsetzung zu erwarten. Daher sollte bei besonders weitgehenden Änderungen die bestehende Auffassung von Interaktion zwischen Komponenten grundsätzlich überarbeitet werden, wie später in Abschnitt 5.4 erläutert wird.

Workflow-Modellierung

Die zuvor angesprochene Einführung von operativen Elementen, wie *Iteration*, *Fallunterscheidung* oder *Bedingung*, ist prinzipiell ebenfalls geeignet, um die Expressivität der Funktionalen Spezifikation zu verbessern. Auf diese Weise lassen sich auch komplexe, übergeordnete Abläufe des gewünschten Systems spezifizieren, die unmittelbar bei späterer Generierung von Prototyp-Varianten berücksichtigt werden können. Andererseits verändert sich durch derartige Erweiterungen das Verhältnis von deklarativen zu operativen Anteilen der Spezifikation beträchtlich. Somit besteht die nicht zu unterschätzende Gefahr, wesentliche Funktionalität des Systems als „Algorithmus“ im Rahmen der Spezifikation zu beschreiben, anstatt die hierfür eigentlich vorgesehenen Softwarekomponenten zu nutzen.

Daher sind mittelfristig überwiegend grafische Beschreibungstechniken zu bevorzugen, mit deren Hilfe sich zwar grundsätzlich beliebige dynamische Verhältnisse angeben lassen, die aber ansonsten in der Regel zu einem höheren Abstraktionsgrad der Funktionalen Spezifikation führen. Beispielsweise beinhaltet ein Flußdiagramm Möglichkeiten zur Beschreibung von Iteration

und bedingter Verzweigungen, ohne jedoch zur detaillierten Repräsentation eines tatsächlich komplexen Verfahrens zu verleiten. Eine entsprechende Erweiterung der Frameworks in diesem Bereich definiert eine Abbildung zwischen Symbolen des Diagramms und Elementen des Metamodells einer Funktionalen Spezifikation. Darüber hinaus ist die beabsichtigte Semantik der grafischen Beschreibungstechnik anzugeben und bei Generierung funktionaler Prototypen zu berücksichtigen. Diese Umsetzung ist im Rahmen gängiger technischer Plattformen für die oben genannten operativen Elemente offensichtlich leicht möglich.

Langfristig erscheint eine grundlegende Weiterentwicklung des Frameworks vielversprechend, welche die Funktionale Spezifikation als automatisierbaren Teil eines übergeordneten, anwendungsbezogenen Geschäftsprozesses (engl. *business process*) versteht. Ein solcher, auch als *Workflow* bezeichneter Ablauf beinhaltet elementare und zusammengesetzte Arbeitsschritte oder Aktivitäten, die über Zwischenergebnisse sowie temporale und kausale Abhängigkeiten verknüpft sind [Jab95]. In diesem Zusammenhang entspricht ein Anwendungsfall der Funktionalen Spezifikation einem elementaren Arbeitsschritt des Workflows, während etwa eine Funktionale Aussage als besondere, sequentiell ausgeführte Folge von Aktivitäten aufgefaßt werden kann. Weiterführende Abhängigkeiten sowie komplexe dynamische Abläufe werden im Rahmen einer solchen Interpretation durch entsprechende Elemente der eingesetzten Workflow-Modellierung ausgedrückt.

Auf diese Weise ergibt sich ein überwiegend deklarativer, anwendungsbezogener Charakter der Funktionalen Spezifikation, wobei die gewünschte Funktionalität des zu entwickelnden Systems im Zusammenhang des übergeordneten Geschäftsprozesses verstanden wird. Bestehende Ansätze zur Workflow-Modellierung bieten darüber hinaus ausdrucksvolle, vielfach grafische Beschreibungstechniken, die mit vertretbarem Aufwand für die Zwecke des Frameworks angepaßt werden können. Schließlich sind zahlreiche, existierende Werkzeuge verfügbar, die Spezifikation und Ausführung eines gegebenen Workflows unterstützen, wie im folgenden Abschnitt angeführt wird.

5.2.3 Beschreibungstechnik und Werkzeugunterstützung

Der Einsatz geeigneter Beschreibungstechniken und Werkzeugen trägt wesentlich zu einer erfolgreichen praktischen Umsetzung des vorgestellten Ansatzes bei. Wie im Bereich der Ontologie (vgl. Abschnitt 5.1.3) läßt sich hierbei auch die zugehörige Interaktion eines komponentenbezogenen Anwendungsfalls mit Mitteln der UML beschreiben und somit über entsprechende Werkzeuge erstellen oder bearbeiten.

UML und CASE-Werkzeuge

Offensichtlich richtet sich die Wahl der geeigneten Beschreibungstechnik nach dem verwendeten Teilmodell zur Spezifikation von Verhalten. So kann die hierfür in dieser Arbeit vorgeschlagene Folge von Operationsaufrufen (siehe Abschnitt 3.4) auf unmittelbar ersichtliche Weise durch ein Sequenzdiagramm [Krü00] oder ein semantisch äquivalentes Kollaborationsdiagramm [RJB98] beschrieben werden. Hierfür werden referenzierte Rollen der Interaktion auf Akteure des Sequenzdiagramms abgebildet, während ein angegebener Operationsaufruf als durch einen Pfeil repräsentierter Nachrichtenaustausch aufgefaßt wird. Darüber hinaus ist die Zuordnung zwischen Parametern oder Rückgabewerten und Konzepten, Rollen oder Ausdrücken zu ergänzen (vgl. Abbildung 3.9).

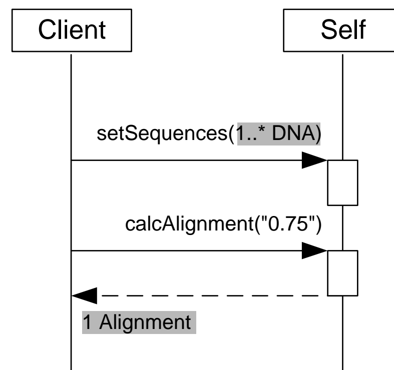


Abb. 5.3: *Sequenzdiagramm für den CUC Calculate Alignment*

Abbildung 5.3 zeigt ein einfaches Beispiel eines so eingesetzten Sequenzdiagramms zur Beschreibung der zum Anwendungsfall Calculate Alignment gehörigen Interaktion (vgl. Abbildung 3.10). Sie verdeutlicht, daß im Unterschied zur herkömmlichen Interpretation eines UML-Sequenzdiagramms tatsächlich nur die behandelten Konzepte, Rollen oder Ausdrücke im Kontext der ausgetauschten Nachrichten angegeben werden. Der auf technischer Ebene relevante Typ eines Parameters oder Rückgabewerts einer Operation wird erst später im Verlauf der Prototyp-Generierung berücksichtigt (siehe Abschnitt 3.6). Dies erlaubt eine weitgehende Abstraktion von technischen Details der Implementierung und erleichtert die anwendungsbezogene Beschreibung einer Interaktion.

Somit können auch umfangreiche Folgen von Operationsaufrufen mit mehreren Interaktionspartnern anschaulich spezifiziert werden. Zukünftig lassen sich auch komplexe dynamische Abläufe mit Sequenzdiagrammen re-

präsentieren, sobald vorgeschlagene Erweiterungen dieser Beschreibungstechnik, wie Iteration, Verzweigung oder hierarchische Strukturierung [Krü00], in die bestehenden Werkzeuge integriert werden. In diesem Fall ist die beabsichtigte Semantik der grafischen Notation genau zu definieren und bei späterer Generierung funktionaler Prototypen entsprechend zu berücksichtigen. Für zusätzlich eingeführte Modelle zur Beschreibung von Verhalten (siehe Abschnitt 5.2.2) sind hingegen andere Mittel der UML möglicherweise besser geeignet. So lassen sich beispielsweise Zustandsautomaten offensichtlich durch UML-Zustandsdiagramme [RJB98] repräsentieren. In jedem Fall verspricht der Einsatz bekannter, weitgehend standardisierter grafischer Beschreibungstechniken eine vereinfachte Anwendung des vorgestellten Ansatzes sowie eine umfassende Unterstützung verfügbarer Werkzeuge.

Workflow-Modellierung

Die in Abschnitt 5.2.2 erläuterte, weiterführende Auffassung der Funktionalen Spezifikation als Workflow eines übergeordneten Geschäftsprozesses eröffnet vielfältige Möglichkeiten zum Einsatz entsprechender Beschreibungstechniken und Werkzeuge. So definiert beispielsweise die *Workflow Management Coalition* [WMC01], ein Zusammenschluß zahlreicher namhafter Hersteller von Workflow Management Systemen, einen Standard für Dokumentation und Austausch von Workflow-Modellen [WMC99], der bereits durch ausgewählte, vorhandene Werkzeuge unterstützt wird [Fuj01, Ver01]. Sie erlauben die Modellierung von Workflows mittels überwiegend grafischer Beschreibungstechniken, die somit auch zur Erstellung und Bearbeitung der Funktionalen Spezifikation eingesetzt werden können.

Ein weiterer Vorteil einer derartigen Spezifikation von erwünschter Funktionalität liegt in der potentiellen Simulation und Ausführung von Workflow-Modellen. So beinhaltet ein Workflow Management System typischerweise eine sog. *Workflow Engine*, die einen geeignet beschriebenen Arbeitsablauf interpretiert und durch Integration externer Komponenten automatisiert [WMC95]. Hierbei kann unter bestimmten Voraussetzungen die Auswahl und erforderliche Verknüpfung entsprechender Softwarekomponenten mit Hilfe des in dieser Arbeit vorgestellten Frameworks vorgenommen werden, falls sich die eingesetzte Workflow Engine mit vertretbarem Aufwand anpassen läßt. Auf diese Weise ermöglicht eine solche, auch als *Embedded Workflow System* bezeichnete Infrastruktur den Übergang von generierten hin zu simulierten funktionalen Prototypen, wie später in Abschnitt 5.4 diskutiert wird.

Die praktische Umsetzung der aufgeführten Erweiterungen für diesen Bereich des Frameworks ist allerdings mit durchaus erheblichem Aufwand ver-

bunden. So muß zunächst untersucht werden, inwieweit sich die Funktionale Spezifikation im Hinblick auf Abstraktionsgrad und Expressivität tatsächlich auf die vorgegebene Workflow-Modellierung abbilden läßt. Weiterhin ist die Semantik der getroffenen Festlegung genau zu definieren und bei Generierung oder Simulation funktionaler Prototypen zu berücksichtigen. Zuletzt sind die eingesetzten Werkzeuge entsprechend anzupassen und um zusätzlich erforderliche Funktionalität zu ergänzen. Dennoch erscheint die angesprochene Integration des Frameworks mit Embedded Workflow Systemen aufgrund der oben genannten Vorteile als vielversprechender Ausgangspunkt für zukünftige Arbeiten.

5.3 Generierung funktionaler Prototypen

Der in Abschnitt 3.6 beschriebene Ansatz nutzt die im Rahmen von Ontologie, CUC-Beschreibungen und Funktionaler Spezifikation bereitgestellten Informationen zur möglichst weitgehend automatisierten Generierung funktionaler Prototypen. Hierbei strukturiert das Framework das übergeordnete Verfahren in die grundlegenden, logisch zusammengehörige Aufgabenbereiche Komponenten-Auswahl, Adapter-Generierung, Varianten-Optimierung und Varianten-Generierung (vgl. Abbildung 3.17), für die im folgenden jeweils unterschiedliche Erweiterungen vorgeschlagen werden. Sie können in der Regel auch getrennt betrachtet und unabhängig voneinander, je nach erforderlichem Aufwand, umgesetzt werden. Diese Eigenschaft verdeutlicht somit in besonderer Weise die Flexibilität der vorgestellten Lösung.

5.3.1 Komponenten-Auswahl

Zu Beginn des Verfahrens werden geeignete Komponenten für gegebene Anwendungsfälle und primäre Konzepte der Funktionalen Spezifikation ausgewählt (siehe Abschnitt 3.6.2). Hierfür ist die Definition der logischen Kompatibilität zwischen Anwendungsfällen ausschlaggebend, die selbst wiederum maßgeblich durch den eingeführten Begriff der semantischen Kompatibilität zwischen Konzepten der Ontologie bestimmt wird (siehe Definition 3.6).

Neben einer offensichtlichen Anpassung durch unterschiedlich definierte Bewertungsfunktionen zur Berechnung der semantischen Kompatibilität (siehe Gleichung 3.2, 3.3 und 3.4), ist für eine verbesserte Komponenten-Auswahl die weiterführende Interpretation der in der Ontologie festgelegten Zusammenhänge von entscheidender Bedeutung. Schließlich repräsentiert eine Ontologie die grundlegende Modellierung des Anwendungsbereichs, in dessen Umfeld die Funktionalität des zu entwickelnden Systems benötigt wird. Aufgrund des unbestimmten Charakters initialer funktionaler Anforderungen ist

die tolerante Auswahl möglichst zahlreicher, *potentiell geeigneter* Komponenten anzustreben, um die erreichbare Vielfalt generierter Prototyp-Varianten zu steigern.

Erweiterte semantische Kompatibilität

Mit dieser übergeordneten Zielsetzung kann zunächst die Auffassung von semantischer Kompatibilität zwischen Konzepten erweitert werden. Die bisherige Definition der verschiedenen Formen semantischer Kompatibilität stützt sich auf Relationen mit vordefinierter Semantik, wie Äquivalenz, Generalisierung oder Interpretation, und somit letztlich auf eine möglichst weitgehende Übereinstimmung der im Rahmen der Ontologie festgelegten Beziehungen für die jeweils betrachteten Konzepte (siehe Definition 3.2, 3.3, 3.4 und 3.5). Es ist daher naheliegend, dieses grundlegende Merkmal direkt und unmittelbar zur Bestimmung semantischer Kompatibilität heranzuziehen. Somit können gegebene Konzepte der Ontologie auch ohne Berücksichtigung ausgezeichneter Relationen als *erweitert semantisch kompatibel* aufgefaßt werden, falls hinreichend viele Beziehungen zu anderen Konzepten übereinstimmen. Neben der Anzahl an Übereinstimmungen kann die Bedeutung ausgewählter Beziehungen durch die Einführung entsprechender Gewichtungsfaktoren bei Erstellung der Ontologie berücksichtigt werden. Vergleichbar mit der Angabe einer konzeptuellen Distanz für Generalisierungs- und Interpretationsbeziehungen (siehe Abschnitt 3.3), erlaubt ein derartiger Gewichtungsfaktor die Modellierung besonders charakteristischer Beziehungen im Anwendungsbereich.

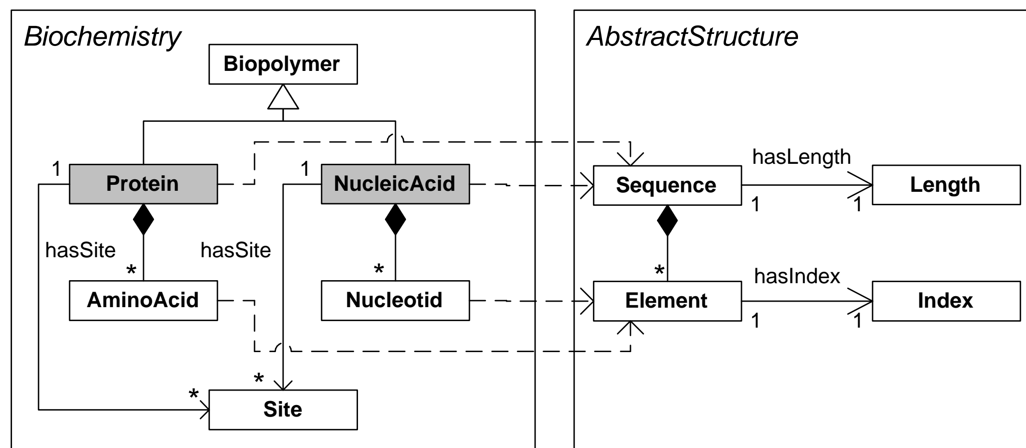


Abb. 5.4: *Erweiterte sem. Kompatibilität zwischen* Protein *und* NucleicAcid

Abbildung 5.4 verdeutlicht diesen Ansatz am Beispiel der dunkel hervorgehobenen Konzepte Protein und NucleicAcid im gezeigten Ausschnitt der gemeinsamen Ontologie. Mit Hilfe der in Abschnitt 3.6.1 eingeführten grafischen Notation wird ersichtlich, daß beide Konzepte der Domäne Biochemistry als Spezialisierungen des Konzepts Biopolymer definiert sind, die mit dem Konzept Site in Beziehung stehen (vgl. Abbildung 5.1) und als Sequence der Domäne AbstractStructure interpretiert werden können. Aufgrund dieser Übereinstimmungen sind Protein und NucleicAcid als erweitert semantisch kompatibel anzunehmen, obwohl keine Äquivalenz oder Generalisierungsbeziehung zwischen ihnen besteht. Dementsprechend lassen sich beispielsweise Komponenten mit dem angebotenen CUC Display NucleicAcid möglicherweise zur Erfüllung des Anwendungsfalls Display Protein heranziehen.

Andererseits ist die Zusammensetzung aus Aminosäuren (AminoAcid) und Nukleotiden für das Verständnis eines Proteins bzw. einer Nukleinsäure von weitaus größerer Bedeutung. Im Rahmen des gezeigten Ausschnitts der Ontologie unterschieden sich die Konzepte Protein und NucleicAcid tatsächlich nur in der jeweiligen Aggregationsbeziehung. Daher sind diese Beziehungen mit einem deutlich höheren Gewichtungsfaktor bei Modellierung des Anwendungsbereichs zu versehen. Die geeignete Definition einer Bewertungsfunktion zur Berechnung der erweiterten semantischen Kompatibilität berücksichtigt diese Unterschiede, so daß sich ein insgesamt geringer Grad der Kompatibilität zwischen Protein und NucleicAcid ergibt. Letztlich bestimmt in der Folge die Festlegung des Schwellwerts ϵ_k als untere Grenze der akzeptablen semantischen Kompatibilität über die getroffene Auswahl an Komponenten für Prototyp-Varianten (siehe Abschnitt 3.6.2).

Berücksichtigung von Manipulationen

Neben der so beschriebenen, mit geringem Aufwand zu realisierenden Erweiterung der semantischen Kompatibilität zwischen Konzepten, erlaubt die in Abschnitt 5.1.2 vorgeschlagene Integration von Manipulationen in das Metamodell der Ontologie eine weiterführende Auffassung der logischen Kompatibilität zwischen Anwendungsfällen. Beispielsweise ermöglicht eine im Rahmen der Ontologie definierte Beziehung implies zwischen den Manipulationen Edit und Display die Auswahl von Komponenten zur Bearbeitung von Konzepten für Anwendungsfälle der Funktionalen Spezifikation, die lediglich deren Anzeige erfordern. In vergleichbarer Weise kann eine Äquivalenzrelation für Manipulationen eingeführt werden, die im weiteren Verlauf zu einer gleichberechtigten Auswahl von entsprechend spezifizierten Anwendungsfällen führt.

Zur praktischen Umsetzung dieses Ansatzes ist zunächst die beabsichtigte Semantik derartiger Beziehungen zwischen Manipulationen der Ontologie

festzulegen. Anschließend ist die Definition von logischer Kompatibilität zwischen Anwendungsfällen geeignet zu erweitern (siehe Definition 3.6), wobei eine unterschiedliche Bedeutung der eingeführten Relationen durch entsprechende Gewichtungsfaktoren berücksichtigt werden kann. So führt beispielsweise die Äquivalenz beteiligter Manipulationen zu einer höheren logischen Kompatibilität bei Abgleich von gewünschter und erbrachter Funktionalität als eine vergleichbare Implikation. Diese Unterscheidung wird zweckmäßigerweise durch entsprechend definierte Teilbewertungsfunktionen umgesetzt.

Weiterhin ist neben der getrennten Betrachtung von Manipulationen und Konzepten bei Ermittlung von logischer Kompatibilität zwischen Anwendungsfällen auch eine explizite Bewertung ihrer gemeinsamen Kombination möglich. So kann bei der in Abschnitt 5.1.2 vorgeschlagenen Zuordnung zwischen Manipulationen und Konzepten festgelegt werden, daß bestimmte Kombinationen als semantisch äquivalent aufzufassen sind. Beispielsweise ist innerhalb der Domäne SequenceAnalysis eine mit Calculate 1 Alignment bezeichnete Funktionalität gleichbedeutend auch als Align 1..* NucleicAcid auszudrücken (vgl. Abschnitt 3.7). Derartige Beziehungen zwischen Anwendungsfällen auf Ebene der Ontologie sind im weiteren Verlauf des Verfahrens bei Bestimmung ihrer logischen Kompatibilität entsprechend zu berücksichtigen. Dieses Vorgehen erlaubt somit mehr Flexibilität und Toleranz bei Spezifikation, Auswahl und Verknüpfung von Anwendungsfällen.

Mit Hilfe der oben beschriebenen Erweiterungen können zusätzlich verfügbare Informationen über Verhältnisse im Anwendungsbereich für eine verbesserte Auswahl potentiell geeigneter Komponenten herangezogen werden. Der hierfür erforderliche Aufwand bei Erstellung, Validierung und Pflege der Ontologie erscheint gegenüber den zu erwartenden Vorteilen gerechtfertigt, insbesondere falls diese Aufgaben zukünftig durch leistungsfähige Werkzeuge umfassend unterstützt werden (siehe Abschnitt 5.1.3). Darüber hinaus ist die praktische Umsetzung der vorgeschlagenen Ansätze überwiegend mit geringem Aufwand durchzuführen, weil diese die grundlegende Konzeption des Frameworks nicht verändern.

5.3.2 Adapter-Generierung

Nach Festlegung der Kombination aus funktionalen Komponenten und Repräsentationen für primäre Konzepte der Spezifikation, wird im nächsten Schritt des übergeordneten Verfahrens versucht, zwischen technisch nicht-kompatiblen Repräsentationen des gleichen Konzepts über entsprechende Adapter zu vermitteln (siehe Abschnitt 3.6.3). Dies ermöglicht die Verknüpfung von Komponenten unterschiedlicher Hersteller über Parameter und

Ergebnisse der ausgewählten komponentenbezogenen Anwendungsfälle. Hierbei ist neben der einfachen Integration manuell erstellter Adapter v.a. deren möglichst weitgehend automatisierte Generierung von entscheidender Bedeutung für Effektivität und Effizienz des gesamten Verfahrens. Diese grundlegende Zielsetzung betrifft auch die Einbindung sog. *Interaktionsadaptern* zur Vermittlung zwischen Teilnehmern der im Rahmen von CUC-Beschreibungen angegebenen Interaktionen, wie später erläutert wird.

Repräsentationsadapter

Die Erstellung von Adaptern für Repräsentationen eines Konzepts erfordert im wesentlichen einen geeigneten Abgleich des jeweils unterschiedlich repräsentierten Zustands als Zusammenfassung aller etablierten Beziehungen zu anderen Konzepten der Ontologie. Während im allgemeinen aufgrund beliebig komplexer Verhältnisse im Anwendungsbereich nicht auf manuell erstellte Adapter verzichtet werden kann, so lassen sich doch in vielen Fällen bekannte Strukturen der Ontologie zur automatisierten Generierung entsprechender Funktionalität ausnutzen. Beispielsweise erfordert eine rekursive, hierarchisch organisierte Struktur der Konzepte und ihrer Beziehungen in der Regel ein ebenfalls rekursives Verfahren, um zwischen verschiedenen möglichen Repräsentationen dieser Struktur zu vermitteln. Hierbei werden grundlegende Schritte zum Abgleich des Zustands in der immer gleichen Weise wiederholt, etwa das Durchlaufen einer Baumstruktur in einer definierten Ordnung, um den betreffenden Baum in eine andere Repräsentation zu überführen.

Daher ist es naheliegend, diese Abläufe zu automatisieren, falls in der gemeinsamen Ontologie eine entsprechende Interpretation von Konzepten als besonders ausgezeichnete Struktur explizit festgelegt ist (vgl. Abbildung 3.18). Solche Interpretationen erlauben eine sehr generische, weitreichend anwendbare Umsetzung des eingesetzten Verfahrens zum Abgleich des repräsentierten Zustands. Aus diesem Grund wird im folgenden die Einführung von sog. *Strategien* als mögliche Erweiterung des Frameworks im Bereich der Adapter-Generierung vorgeschlagen. Eine derartige Strategie bezieht sich auf Konzepte und Manipulationen der Ontologie, um in Verbindung mit operativen Elementen, wie Iteration, Fallunterscheidung oder rekursiver Aufruf, einen *abstrakten Algorithmus* zur Konvertierung von Repräsentationen zu beschreiben. Die Implementierung des Frameworks kann die bereitgestellten Strategien nutzen, um benötigte Adapter automatisch zu erstellen. Hierbei wird die elementare Funktionalität der Strategie durch entsprechende Anwendungsfälle der beteiligten Repräsentationen implementiert.

```
Strategy: Sequence
    Domain: AbstractStructure
    Conversion: 1 Sequence seqA -> 1 Sequence seqB
    Component Use Cases:  Sequence : Get Length
                          Sequence : Get Element(Index)
                          Sequence : Append Element(Element )
    Algorithm:   Initialize seqB;
                 IterateFor seqA : Get Length {
                      1 Element x = seqA : Get Element( IterationCounter );
                      seqB : Append Element(x);
                 }
```

Abb. 5.5: *Strategie zur Konvertierung des Konzepts Sequence*

Abbildung 5.5 verdeutlicht den so skizzierten Ansatz an Hand einer exemplarischen Strategie zur Konvertierung des Konzepts Sequence aus der Domäne AbstractStructure (siehe Abbildung 3.18). Im Rahmen der gewählten textuellen Notation werden zunächst Richtung und beteiligte Partner des Abgleichs festgelegt. Anschließend wird die vorausgesetzte Funktionalität der betroffenen Repräsentationen aufgeführt. Sie entspricht in der Regel typischen Anwendungsfällen der zugehörigen Softwarekomponenten, wobei möglicherweise benötigte Parameter als Konzepte der Ontologie angegeben sind. Schließlich wird das eigentliche Verfahren beschrieben, nachdem diese Anwendungsfälle mit Hilfe vorgegebener, in der Abbildung entsprechend hervorgehobener operativer Elemente verknüpft werden. Diese abstrakte Algorithmus kann später, bei Kenntnis der beteiligten Repräsentationen für das Konzept Sequence, sowie der jeweils zugeordneten Anwendungsfälle und ihrer Implementierung in ausführbaren Quellcode der verwendeten technischen Plattform umgesetzt werden.

Die so festgelegte Strategie erlaubt einen unidirektionalen Abgleich des Zustands zwischen beliebigen Repräsentationen des Konzepts Sequence, sofern die vorausgesetzte Funktionalität über entsprechende Anwendungsfälle bereitgestellt wird[2]. Die tatsächliche Realisierung des abstrakten Algorithmus ergibt sich somit erst durch eine zugehörige Interaktion der beteiligten komponentenbezogenen Anwendungsfälle. Auf diese Weise wird eine unerwünschte Abhängigkeit von konkreten technischen Elementen, wie Schnitt-

[2] Offensichtlich können mehrere alternative Strategien mit unterschiedlichen Voraussetzungen angegeben werden, aus denen die jeweils am besten Geeignete nach den vorliegenden Verhältnissen ausgewählt wird.

stellen oder deren Signatur, vermieden. Dies ermöglicht eine weitreichende Anwendbarkeit der angegebenen Strategie, auch wenn die Korrektheit des umgesetzten Verfahrens nicht grundsätzlich zugesichert werden kann. Daher ist eine entsprechende Bewertung des so erstellten Adapters im Verlauf der Varianten-Optimierung zu berücksichtigen (siehe Abschnitt 3.6.5 und 5.3.4).

Weiterhin können mit Hilfe der in Abbildung 5.5 exemplarisch vorgestellten Strategie nunmehr alle Konzepte, für die eine Interpretation als Sequence innerhalb der Ontologie definiert ist, in vergleichbarer Weise behandelt werden. Somit erlaubt der abstrakte Algorithmus beispielsweise einen Abgleich des Zustands zwischen unterschiedlichen Repräsentationen des Konzepts Protein (vgl. Abbildung 5.4), wobei die zugehörigen Subinterpretationen zu beachten sind. Dies erfordert in der Folge möglicherweise eine geeignete Vermittlung zwischen den beteiligten Repräsentationen von Element und AminoAcid. Hierfür können wiederum andere verfügbare Strategien oder manuell erstellte Adapter herangezogen werden, so daß sich ein insgesamt rekursives Verfahren der Adapter-Generierung ergibt (vgl. Abschnitt 3.6.3).

Zusammenfassend läßt sich feststellen, daß die vorgeschlagene Erweiterung des Frameworks eine wesentliche Verbesserung von Umfang und Qualität automatisch generierter Adapter verspricht. Der im Rahmen einer Strategie beschriebene Algorithmus erlaubt die flexible Handhabung komplexer Verhältnisse im Anwendungsbereich, insbesondere falls zukünftig Constraints in das Metamodell einer Ontologie integriert werden (siehe Abschnitt 5.1.1). Die erreichte Unabhängigkeit von technischen Details der Implementierung ermöglicht eine weitgehende Wiederverwendung und Übertragbarkeit der entwickelten Strategien. Die praktische Umsetzung des vorgeschlagenen Ansatzes ist mit geringem Aufwand verbunden, da hierfür keine grundsätzlichen Änderungen des Frameworks erforderlich sind. Somit ist lediglich die Implementierung der Komponenten Adapter Manager und Adapter Generator entsprechend zu erweitern und eine geeignete Anpassung der Varianten-Bewertung im Verlauf der Optimierung vorzunehmen (vgl. Abschnitt 4.2 und 3.6.5).

Interaktionsadapter

Im Gegensatz zu Adaptern für Repräsentationen, kann die möglicherweise erforderliche Vermittlung zwischen unterschiedlichen Partnern einer Interaktion zwischen Komponenten im allgemeinen nicht schematisch behandelt werden. Hierbei kennzeichnen Rollen sowie deren jeweils vorausgesetzte Schnittstelle die angegebenen Teilnehmer einer solchen Interaktion (siehe Abschnitt 3.4). Aufgrund der in Abschnitt 3.6.4 erläuterten Beschränkung auf binäre Interaktionen mit vordefinierten Rollen wird bei der gegenwärti-

gen, prototypischen Referenz-Implementierung des Frameworks die Vermittlung zwischen technisch inkompatiblen Teilnehmern einer Interaktion nicht berücksichtigt. Tatsächlich sind jedoch in der Praxis Interaktionen zu erwarten, deren zugeordnete Komponenten zwar grundsätzlich die erwünschte, anwendungsbezogene Funktionalität erbringen, diese aber über eine Schnittstelle zur Verfügung stellen, deren Typ und Signatur nicht mit den jeweils spezifizierten Voraussetzungen übereinstimmt.

```
Component: casa.analysis.SequenceComparator
   Offered Interfaces:  casa.analysis.HomologyAnalysis
   Required Interfaces: casa.analysis.AlignmentAnalysis, casa.data.Alignment,
                        casa.data.DNASequence
   Signature:           casa.analysis.HomologyAnalysis {
                             void setSequences(java.util.List s);
                             void setAlignment(casa.data.Alignment);
                             double calcHomology();
                        }
   States:              <Default>
   Initial State:       <Default>
   Provided Use Cases:  Calculate 1 Homology
      Referenced Use Cases: casa.data.SeqImpl/Represent 1 DNA,
                            casa.analysis.Clustal/Calculate 1 Alignment
      Interactions: Standard from <Default> to <Default>
         Declared Roles: AliCalculator
         Operation Sequence: Self.setSequences(1..* DNA)
                             AliCalculator.setSequences(1..* DNA)
                             1 Alignment = AliCalculator.calcAlignment(0.6)
                             Self.setAlignment(1 Alignment)
                             1 Homology = Self.calcHomology()
```

Abb. 5.6: *CUC-Beschreibung zur Komponente* `SequenceComparator`

Diese Problematik wird beispielhaft durch die in Abbildung 5.6 dargestellte CUC-Beschreibung zur Komponente `SequenceComparator` verdeutlicht. Der von ihr angebotene Anwendungsfall Calculate 1 Homology, also die Berechnung eines Werts für die Homologie der übergebenen DNA-Sequenzen (vgl. Abschnitt 2.3), wird über eine Interaktion mit der zuvor deklarierten Rolle `AliCalculator` realisiert. Die später dieser Rolle zugeordnete Komponente erstellt zunächst ein entsprechendes Sequenz-Alignment, das mit einem Aufruf der Operation `setAlignment` an `SequenceComparator` übermittelt wird und anschließend als Grundlage der mittels `calcHomology` durchgeführten Berechnung dient. Hierbei wird die vorausgesetzte Schnittstelle des In-

teraktionspartners als `AlignmentAnalysis` zu Beginn der CUC-Beschreibung explizit aufgeführt. Darüber hinaus beinhaltet die in Abbildung 5.6 gezeigte Beschreibung einen Bezug zum Anwendungsfall Calculate 1 Alignment der Komponente `Clustal`. Tatsächlich implementiert `Clustal` die oben genannte Schnittstelle und kann somit als Teilnehmer der betreffenden Interaktion ausgewählt werden (vgl. Abbildung 3.10).

Andererseits ist es vorstellbar, daß zum Zeitpunkt der Prototyp-Generierung die Komponente `Clustal` nicht zur Verfügung steht, etwa weil diese nicht in Verbindung mit `SequenceComparator` installiert wurde, oder eine andere, bereits ausgewählte Komponente eine grundsätzlich vergleichbare Funktionalität erbringt und daher für diese Interaktion erneut verwendet wird. So bietet die Komponente `Phylip` des Anwendungsbeispiels ebenfalls den CUC Calculate 1 Alignment an, obwohl deren Schnittstelle als inkompatibel zu `AlignmentAnalysis` anzunehmen ist (vgl. Abbildung 3.20). Aus diesem Grund sollte in der Folge ein entsprechender *Interaktionsadapter* eingebunden werden, der zwischen den beteiligten Schnittstellen vermittelt und somit mehr Flexibilität bei Kombination von Komponenten unterschiedlicher Hersteller verspricht.

Ein derartiger Interaktionsadapter implementiert sämtliche beteiligten Schnittstellen, im Beispiel also sowohl `AlignmentAnalysis` als auch `Phylip`, wobei die angebotene Funktionalität einer gegebenen Schnittstelle mit den Möglichkeiten der jeweils anderen Schnittstellen umgesetzt wird. Dies kann in aller Regel erreicht werden, weil der zugrundeliegende Anwendungsbereich allen Komponenten gemein ist und diese ausschließlich über logisch kompatible Anwendungsfälle ausgewählt werden (vgl. Abschnitt 3.6.2). Die hierfür ebenfalls erforderlichen Adapter zwischen unterschiedlichen Repräsentationen für Parameter und Rückgabewerte zugehöriger Operationen können gemäß den in Abschnitt 3.6.3 und 5.3.2 erläuterten Verfahren generiert werden.

Für eine vereinfachte Implementierung und Integration ist es hilfreich, sich zunächst auf binäre, gerichtete Interaktionsadapter zu beschränken. Diese basieren auf einer fest vorgegebenen Komponente und bieten zusätzlich eine zweite, technisch unterschiedliche Schnittstelle an, deren Funktionalität durch den Adapter entsprechend übersetzt wird. Für das oben aufgeführte Beispiel ergibt sich auf diese Weise ein sog. *Wrapper* für die Komponente `Phylip`, der zusätzlich die Schnittstelle `AlignmentAnalysis` implementiert und Aufrufe der Operationen `setSequences` oder `calcAlignment` auf entsprechende Funktionalität von `Phylip` abbildet. Die Instanz eines solchen Interaktionsadapters kann somit später ohne Schwierigkeiten der Rolle `AliCalculator` zugeordnet werden.

Allerdings ist eine vollständig automatisierte Erstellung solcher Interaktionsadapter im allgemeinen nicht zu erreichen, weil das Framework keine wei-

terführenden Informationen über die erforderliche Abbildung zwischen den beteiligten Schnittstellen besitzt. Somit kann zunächst lediglich ein Rahmen generiert werden, der in einem zweiten Schritt vom Benutzer geeignet zu ergänzen ist. Anschließend läßt sich der manuell erstellte Adapter immerhin zur Vermittlung weiterer, gleichartiger Interaktionen erneut verwenden.

Bei genauerer Betrachtung kann ein Interaktionsadapter jedoch als Komponente verstanden werden, die simultan *mehrere, unterschiedliche Protokolle* erfüllt bzw. zwischen diesen Protokollen vermittelt. Hierbei beschreibt ein gegebenes Protokoll die vorausgesetzten Regeln und Konsistenzbedingungen bei Benutzung einer von der Komponente angebotenen Schnittstelle. Die so beschriebene Teilfunktionalität entspricht einem Ausschnitt oder *Aspekt* der gesamten Funktionalität im Sinne der aspektorientierten Systementwicklung [KLM+97, CHOT99]. Dieser Forschungsbereich bemüht sich um eine möglichst weitgehend automatisierte Synthese einer Komponente aus der Gesamtheit ihrer geeignet spezifizierten Aspekte. Daher können bestehende oder zukünftige Ergebnisse voraussichtlich auf die Generierung von Interaktionsadaptern übertragen werden.

So beschreibt beispielsweise [GV00] einen Ansatz für aspektorientiertes Softwaredesign, der eine konstruktive Integration mehrerer verhaltensbezogener Aspekte im Rahmen *einer* betrachteten Komponente erlaubt. Hierbei entspricht jeder Aspekt einer angebotenen Schnittstelle, deren Benutzung über ein als nicht-deterministischer Zustandsautomat spezifiziertes Protokoll geregelt wird. Das angegebene Verfahren konstruiert das Produkt aller beteiligten Zustandsautomaten und eliminiert anschließend die als unerwünscht gekennzeichneten Zustandskombinationen. Somit umfaßt der resultierende Produktautomat alle ursprünglich unterstützten Protokolle und kann als Ausgangspunkt für Simulation oder Implementierung der betroffenen Komponente dienen. Dieser vielversprechende Ansatz läßt sich grundsätzlich mit verhältnismäßig geringem Aufwand auf die Generierung von Interaktionsadaptern übertragen, insbesondere falls zukünftig auch Zustandsautomaten zur Spezifikation von Interaktionen in das übergeordnete Framework integriert werden (siehe Abschnitt 5.2.2).

5.3.3 Varianten-Generierung

Im Verlauf der Varianten-Generierung wird der genaue Konstruktionsplan einer Prototyp-Variante festgelegt, geeignete Interaktionen zur Umsetzung der angebotenen Funktionalität ausgewählt, die erforderliche Zuordnung zwischen Instanzen und Rollen sowie primären und sekundären Konzepten getroffen, um schließlich mit Hilfe der operativen Elemente einer Funktionalen Spezifikation ausführbaren Quellcode zu generieren, wie in Abschnitt 3.6.4

erläutert wird. Hierbei bestimmt das Verfahren zur Ermittlung und Initialisierung von zusätzlich benötigten Konzepten als Parameter von Operationsaufrufen maßgeblich über den erreichten Grad der Automatisierung und die Qualität der erhaltenen Ergebnisse. Daher werden im folgenden zunächst mögliche Verbesserungen für diese allgemein bedeutsame Aufgabe des Frameworks diskutiert.

Demgegenüber ist die eigentliche Generierung von übersetzbarem und ausführbarem Quellcode offensichtlich durch die eingesetzte technische Plattform bestimmt. Neben einer Unterstützung weiterer komponentenbasierter Infrastrukturen, ist hierbei v.a. die Integration von Werkzeugen zur vereinfachten Erstellung einer grafischen Benutzeroberfläche (engl. *graphical user interface*, GUI) von großer Bedeutung für die praktische Anwendung des vorgestellten Ansatzes. Weitere Verbesserungen der Varianten-Generierung sind nicht zuletzt von den in Abschnitt 5.2.2 und 5.2.3 vorgeschlagenen Erweiterungen für CUC-Beschreibung und Funktionalen Spezifikation abhängig. Sie werden daher im folgenden nicht näher betrachtet, auch wenn beispielsweise neu eingeführte, operative Elemente, wie Iteration oder Fallunterscheidung, auf unmittelbar ersichtliche Weise in Quellcode übersetzt werden können.

Adaptive Umsetzung von Interaktionen

Für die vollständige und möglichst fehlerfreie Umsetzung einer ausgewählten Interaktion müssen deren Teilnehmer sowie primäre oder sekundäre Konzepte als benötigte Parameter von Operationsaufrufen auf Instanzen von Komponenten der betrachteten Prototyp-Variante abgebildet werden. Um hierbei eine möglichst weitgehende Automatisierung des gesamten Verfahrens zu erreichen, verfolgt das vorgestellte Framework eine Reihe von Strategien, die zumindest bei überschaubaren Verhältnissen mit hoher Wahrscheinlichkeit erfolgreich angewendet werden können, wie in Abschnitt 3.6.4 erläutert wird. Allerdings sind die dort aufgeführten Strategien bisher lediglich als fest vorgegebener, impliziter Bestandteil des Frameworks aufzufassen und somit nicht in der entsprechenden Software-Architektur der Referenz-Implementierung berücksichtigt (vgl. Abschnitt 4.1). Zudem ist die gegenwärtige Implementierung auf binäre Interaktionen mit vordefinierten Rollen beschränkt, die in einem vollständig automatisierten Verfahren umgesetzt werden. Jedoch ist zu erwarten, daß insbesondere bei komplexen Verhältnissen ein optional durchgeführter, interaktiver Dialog mit dem Benutzer zu einer erheblich besseren Zuordnung und damit höherer Qualität der generierten Prototypen führt.

Aus diesen Gründen ist es wünschenswert, die Anwendung einer Strategie zur Umsetzung einer Interaktion als eigenen Aufgabenbereich innerhalb des Frameworks zu definieren. Somit können entsprechende Komponenten des

Frameworks entwickelt werden, die geeignete Lösungsansätze implementieren und ihre erzielten Ergebnisse dem übergeordneten Verfahren der Varianten-Generierung zur Verfügung stellen. Eine derartige Strukturierung erleichtert die Einordnung und praktische Umsetzung der in Abschnitt 3.6.4 aufgeführten Strategien erheblich. Auf diese Weise lassen sich aber auch grundsätzlich neue Ansätze, etwa interaktiv in Zusammenarbeit mit dem Benutzer ermittelte Belegungen für Parameter oder Rollen, ohne Schwierigkeiten in das bestehende Framework integrieren.

Die Implementierung kann so die jeweils am besten geeignete Strategie auswählen und deren ermittelte Belegungen für Parameter, Rückgabewerte oder Rollen einer Interaktion in den Konstruktionsplan der betrachteten Prototyp-Variante übernehmen. Darüber hinaus ist es nunmehr möglich, die tatsächlich eingesetzte Strategie im Zusammenhang mit der betreffenden Interaktion zu vermerken, um bei zukünftiger Behandlung des gleichen CUC bereits erfolgreiche Kombinationen erneut zu verwenden bzw. eine im späteren Verlauf als ungeeignet beurteilte Umsetzung zu vermeiden. Dieser Ansatz läßt sich auch auf die eigentlichen Ergebnisse der Anwendung einer Strategie übertragen, etwa um zunächst interaktiv ermittelte Zuordnungen für weitere Abläufe des Verfahrens automatisch bereitzustellen. Beispielsweise kann der Benutzer die erforderliche Belegung für das sekundäre Konzept Database bei Verwendung des von der Komponente `Repository` angebotenen CUC Search * DNA einmalig durch eine konstante URL angegeben (vgl. Abbildung 3.29). Diese konkrete Belegung steht daraufhin bei allen zukünftigen Varianten mit der betreffenden Komponente als mögliche Vorgabe für die Umsetzung der Interaktion zur Verfügung.

Mit dem so skizzierten, adaptiven Verfahren läßt sich die Qualität der generierten Prototyp-Varianten schrittweise verbessern, ohne den erforderlichen manuellen Aufwand übermäßig zu erhöhen. Die Integration in das vorgestellte Framework fällt leicht, da hiermit keine konzeptuellen Veränderungen verbunden sind, sondern im Gegenteil bestehende Aufgabenbereiche und Lösungsansätze genauer definiert werden. Andererseits ist die tatsächliche Implementierung einer Strategie selbst durchaus aufwendig, insbesondere falls zahlreiche interaktive Elemente für den Dialog mit dem Benutzer zu berücksichtigen sind. Daher sollten zunächst die vollständig automatisierten Strategien eingebunden werden, um den so erreichbaren Erfolg an Hand zukünftiger praktischer Erfahrungen zu beurteilen.

GUI-Prototyping

Nachdem der Konstruktionsplan einer gegebenen Prototyp-Variante durch die oben zusammengefaßte Umsetzung aller ausgewählten Interaktionen fer-

tiggestellt ist, kann auf verhältnismäßig einfache Weise fehlerfrei übersetzbarer Quellcode generiert werden, wie in Abschnitt 3.6.4 erläutert wird. Hierbei sind offensichtlich die allgemeinen Vorgaben sowie zahlreiche Details der eingesetzten technischen Plattform zu beachten. Aus diesem Grund beschränkt sich die in Kapitel 4 vorgestellte Referenz-Implementierung auf die Verwendung der Java-Plattform, um die grundlegenden Verfahren des Frameworks mit vertretbarem Aufwand zu realisieren.

In der Praxis ist jedoch die Unterstützung weiterer technischer Plattformen wünschenswert. Schließlich existieren gegenwärtig verschiedene Infrastrukturen, die als Grundlage einer komponentenbasierten Softwareentwicklung herangezogen werden können (siehe Abschnitt 2.2). Eine derartige Erweiterung des Frameworks erfordert keine konzeptionellen Änderungen, da sich die technischen Anteile der vorgestellten Modellierung ausschließlich auf übergreifende Merkmale aller gegenwärtig bedeutsamen Infrastrukturen beziehen, wie später in Abschnitt 6.1.5 diskutiert wird. Somit muß im Rahmen des Frameworks letztlich nur die Implementierung des Code-Generators für Adapter- und Varianten-Generierung entsprechend angepaßt oder erweitert werden, um den vollständigen Konstruktionsplan einer Variante auch für andere Plattformen umzusetzen. Hierbei ist zu überprüfen, ob sich die für den jeweiligen Prototyp ausgewählten Softwarekomponenten selbst ebenfalls unter mehreren technischen Plattformen einsetzen lassen. Unter bestimmten Voraussetzungen kann eine entsprechende Integration durch den Einsatz einer sog. *Bridge* als Vermittler zwischen unterschiedlichen komponentenbasierten Infrastrukturen erreicht werden [Szy98]. Auf diese Weise kann beispielsweise eine vorliegende COM-Komponente auch im Rahmen der Java-Plattform verwendet werden [Sun98, Int01].

Allerdings ist eine derartige Vermittlung in der Regel nicht für visuelle Komponenten möglich, die bei Ausführung einer Prototyp-Variante als Teil ihrer grafischen Benutzeroberfläche in Erscheinung treten. Daher ist zumindest die Generierung der Benutzeroberfläche auf eine bestimmte technische Plattform festgelegt, falls nicht tatsächlich unterschiedliche Versionen jeder visuellen Komponente zur Verfügung stehen. Die diesbezüglichen Anteile der Referenz-Implementierung sind bewußt einfach gehalten, da aus methodischen Gründen eine zukünftige Integration mit verfügbaren, spezialisierten Werkzeugen zur Erstellung eines GUI anzustreben ist. Diese erlauben i.a. eine rasche, intuitive und komfortable Beschreibung der gewünschten Bildschirmdarstellung hinsichtlich Größe und Layout der angezeigten Elemente.

Abbildung 5.7 illustriert den Einsatz eines solchen Werkzeugs zur Beschreibung einer möglichen grafischen Benutzeroberfläche des Prototypen `SequenceTool` (siehe Abschnitt 3.6.1). Das im gezeigten Beispiel verwendete Werkzeug [Mic01d] erlaubt die Auswahl, Plazierung und Kombination

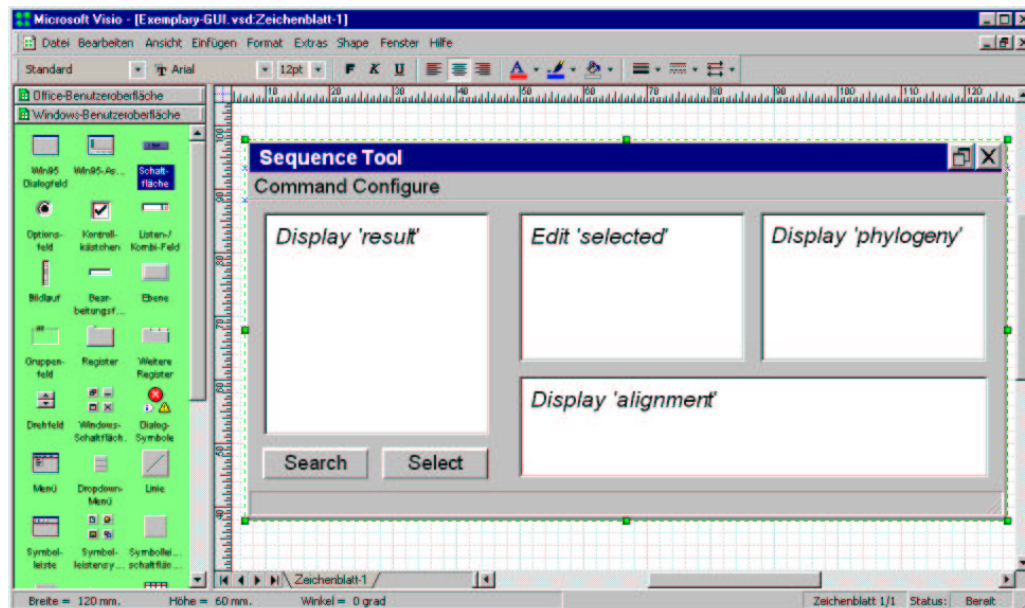


Abb. 5.7: *Bildschirmdarstellung eines Werkzeugs zur GUI-Erstellung*

vorgegebener GUI-Elemente aus einer auf der linken Seite dargestellten Palette. Anschließend werden die Elemente eindeutig benannt, wobei die vorgesehenen Bereiche für visuelle Komponenten einen entsprechenden Bezug auf die Anwendungsfälle der zugrundeliegenden Funktionalen Spezifikation aufweisen (vgl. Abbildung 3.19). Die Beschreibung eines so erstellten GUI-Prototypen kann der Funktionalen Spezifikation zugeordnet und im Verlauf der Varianten-Generierung unmittelbar in eine tatsächlich funktionale Oberfläche umgesetzt werden, wie etwa durch die manuell optimierte Bildschirmdarstellung in Abbildung 3.32 verdeutlicht wird.

Hierbei ist das verwendete Format zur Beschreibung der grafischen Benutzeroberfläche zunächst durch das jeweils eingesetzte Werkzeug bestimmt. Allerdings ist es wünschenswert, eine eigene Beschreibungstechnik einzuführen, mit deren Hilfe dieser Teil des erstellten Prototypen auf möglichst deklarative und technisch neutrale Weise spezifiziert werden kann. Somit ist es prinzipiell möglich, verschiedene GUI-Werkzeuge zu integrieren, deren spezifische Ergebnisse in das gemeinsam genutzte Format überführt werden. Darüber hinaus läßt sich eine derartige Spezifikation für verschiedene technische Plattformen auf jeweils unterschiedliche Weise umsetzen, ohne daß mehrfache, weitgehend redundante Beschreibungen der grundsätzlich gleichen Oberfläche angegeben werden müssen. Aufgrund dieser Vorteile wurden bereits mehrere, überwiegend XML-basierte Beschreibungstechniken für gra-

fische Benutzeroberflächen entwickelt [Kor01, Swi01, Rog01], die als Grundlage für diesbezügliche Erweiterungen des vorgestellten Frameworks dienen können. Der hierfür erforderliche Aufwand ist als verhältnismäßig gering einzuschätzen, insbesondere falls sich die Implementierung auf wenige, besonders bedeutsame Elemente eines GUI beschränkt.

In ihrer Gesamtheit ermöglichen die in diesem Abschnitt vorgeschlagenen Verbesserungen der Varianten-Generierung eine erhöhte Qualität der erstellten Prototypen, die vereinfachte Integration existierender Werkzeuge zur Bearbeitung von GUI-Prototypen sowie eine leichtere Übertragbarkeit der erzielten Ergebnisse auf unterschiedliche technische Plattformen. Diese Aspekte sind gerade im Hinblick auf Effektivität und praktische Umsetzung des vorgestellten Ansatzes von großer Bedeutung, wie später in Abschnitt 6.1.5 diskutiert wird.

5.3.4 Varianten-Optimierung

Die zu erwartende Vielzahl an möglichen Lösungen erfordert eine geeignete Heuristik zur Auswahl und Weiterentwicklung einer beherrschbaren Teilmenge aller Prototyp-Varianten. Hierfür definiert das vorgestellte Framework einen Genetischen Algorithmus, der solche Varianten als Individuen einer gesamten Population auffaßt. Jedem Individuum wird ein eindeutig bestimmter Fitneß-Wert zugeordnet, der eine Selektion der besten Individuen erlaubt. Diese bilden die Grundlage einer jeden Generation der Population, wobei durch Anwendung der weitgehend zufallsbasierten Genetischen Operatoren, wie Mutation, Reproduktion oder Rekombination, auch zusätzlich neue, unterschiedlich zusammengesetzte Individuen entstehen. Die fortgesetzte Durchführung dieser grundlegenden Schritte des Genetischen Algorithmus repräsentiert einen evolutionären Prozeß, in dessen Verlauf sich die durchschnittliche Fitneß jeder Generation in der Regel verbessert (siehe Abschnitt 3.6.5).

Dieses zur Varianten-Optimierung eingesetzte Verfahren kann durch zahlreiche Parameter und Gewichtungsfaktoren bereits sehr flexibel an die jeweiligen Gegebenheiten angepaßt werden. Gerade deren dynamische Veränderung im Verlauf der Optimierung erlaubt den Einsatz durchaus unterschiedlicher Strategien zur Ermittlung bestmöglicher Prototyp-Varianten, wie in Abschnitt 3.6.5 ausführlich erläutert wird. Darüber hinaus können jedoch auch Modifikationen des Genetischen Algorithmus sowie ausgewählter Genetischer Operatoren eingeführt werden, die zu weitergehenden Änderungen der betrachteten Varianten führen. Weiterhin ist es wünschenswert, den erforderlichen manuellen Aufwand zur Bewertung von Prototyp-Varianten zu senken, um eine entsprechend höhere Anzahl unterschiedlicher Lösungen berücksich-

tigen zu können. Zuletzt kann die eingesetzte Heuristik selbst ersetzt bzw. eine Kombination verschiedener Verfahren angewendet werden, um die Qualität der erzielten Ergebnisse zu verbessern.

Weiterentwicklung des Genetischen Algorithmus

Das in dieser Arbeit vorgeschlagene Verfahren zur Varianten-Optimierung eröffnet vielfältige Möglichkeiten für Verbesserungen, die mit geringem Aufwand zu implementieren sind. Hierbei erlaubt der Genetische Algorithmus auch durchaus innovative oder experimentelle Ansätze zur Konstruktion neuer Varianten, da letztlich ausschließlich der individuell ermittelte Fitneß-Wert über deren Übernahme in die nächste Generation der Population entscheidet. So läßt sich beispielsweise der Genetische Operator Reproduktion dahingehend verändern, statt einer einfachen Übernahme des betrachteten Genotyps einer Variante auch zufällig ausgewählte Bestandteile, wie einzelne Gene oder ein gesamtes Chromosom, in mehrfacher Kopie zu erstellen. Diese redundanten Elemente können anschließend einer Mutation unterzogen werden, ohne die ursprünglich vorhandene genetische Information zu beeinträchtigen. Somit werden kumulative Veränderungen und Weiterentwicklungen der jeweiligen Variante ermöglicht, die einzeln betrachtet zu einer deutlichen Verminderung ihrer Fitneß geführt hätten[3].

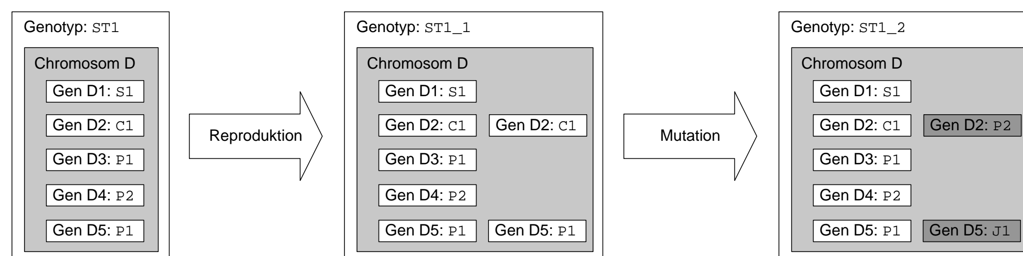


Abb. 5.8: *Modifikation des Genetischen Operators* Reproduktion

Abbildung 5.8 verdeutlicht diese Erweiterung des Verfahrens an Hand eines Ausschnitts des Genotyps ST1 (siehe Abschnitt 3.6.1 und Abbildung 3.34). Die exemplarische Anwendung des Genetischen Operators Reproduktion führt zu einer Duplikation der Gene D2 und D5 innerhalb des Chromosoms D, deren Ausprägung anschließend durch Mutation an den dunkel hervorgehobenen Allelen geändert wird (vgl. Abbildung 3.35). Für eine

[3] Tatsächlich läßt sich diese besondere Form der evolutionären Entwicklung auch in der Biologie am Genom zahlreicher Spezies nachvollziehen [Lew90].

angemessene Bewertung des so entstandenen Genotyps ST1_2 muß offensichtlich die Definition der Fitneß-Funktion entsprechend angepaßt werden (siehe Gleichung 3.6), etwa indem jeweils nur das Maximum der möglichen Teilbewertungen berücksichtigt wird. Außerdem ist für eine interaktive Beurteilung die geeignete Generierung der so zusammengesetzten Prototyp-Variante zu gewährleisten. Beispielsweise können tatsächlich zwei verschiedene Varianten generiert werden, die jeweils getrennt zu bewerten sind und auf Wunsch des Benutzers in der Folge weiterhin als eigenständige Individuen behandelt werden.

Neben der so skizzierten Erweiterung des Genetischen Operators Reproduktion sind offensichtlich ähnliche Modifikationen der Operatoren Mutation und Rekombination vorstellbar, die ebenfalls statt einzelner Gene ganze Gruppen zusammengehöriger Allelen berücksichtigen. Auf diese Weise können in der Vergangenheit als besonders gut beurteilte Kombinationen von Komponenten gemeinsam eingeführt oder ausgetauscht werden, falls es der durch die Funktionale Spezifikation gegebene, aktuelle Kontext zuläßt. Eine derartige, adaptive Interpretation dieser Genetischen Operatoren entspricht zwar nicht dem weitgehend zufälligen Charakter der biologischen Evolution, ist aber gerade im Hinblick auf eine zukünftige Erweiterung des Frameworks um vorgegebene, standardisierte Interaktionen von möglicherweise großer Bedeutung (siehe Abschnitt 5.4).

Anwendungsbezogene Bewertungskomponenten

Verbesserungen der anwendungsbezogenen Teilbewertung einer Prototyp-Variante sind für jedes eingesetzte Verfahren zur heuristischen Optimierung vorteilhaft. Schließlich dienen die generierten Prototypen ausschließlich der Ermittlung und Konkretisierung endgültiger funktionaler Anforderungen an das zu entwickelnde System, wie in Abschnitt 2.1 erläutert wird. Je genauer also diesbezügliche Merkmale einer gegebenen Variante bewertet werden können, desto wahrscheinlicher führt die Auswahl und Weiterentwicklung generierter Lösungen zu plausiblen und tatsächlich verwertbaren Ergebnissen. Darüber hinaus ermöglicht eine weitergehende Automatisierung der gegenwärtig ausschließlich vom Benutzer durchgeführten Beurteilung auch eine gesteigerte Effizienz des übergeordneten Verfahrens, so daß insgesamt eine deutlich höhere Anzahl an unterschiedlichen Prototyp-Varianten betrachtet werden kann.

Aus diesen Gründen verspricht die Integration sog. *Bewertungskomponenten* in das vorgestellte Framework eine wesentliche Verbesserung des erreichten Erfolgs. Eine solche Bewertungskomponente implementiert eine Anzahl von *Testfällen* bezüglich eines gegebenen Anwendungsfalls, die zur Bewer-

tung der jeweils ausgewählten Komponente herangezogen werden. Hierbei beinhaltet ein Testfall typischerweise festgelegte Eingangsdaten oder Parameterbelegungen sowie erwartete Ergebnisse nach Ausführung des zugehörigen Anwendungsfalls. Der Vergleich zwischen erwarteten und tatsächlich erhaltenen Ergebnissen nach Benutzung der getesteten Komponente liefert die Grundlage zu deren anwendungsbezogener Beurteilung. Diese findet im Anschluß Eingang in die Gesamtbewertung der jeweils betrachteten Variante (vgl. Abbildung 3.36).

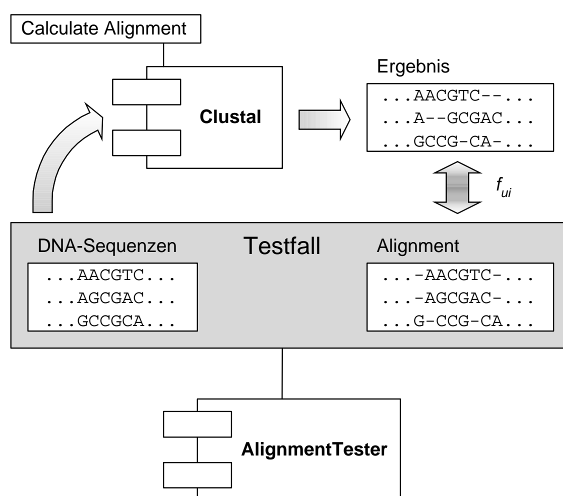


Abb. 5.9: *Bewertungskomponente für den CUC Calculate Alignment*

Das Prinzip einer solchen automatisierten Teilbewertung wird in Abbildung 5.9 durch den Einsatz der exemplarischen Bewertungskomponente `AlignmentTester` für den Anwendungsfall Calculate Alignment verdeutlicht. Sie stellt einen dunkel hervorgehobenen Testfall mit vorgegebenen DNA-Sequenzen bereit, aus denen die im Beispiel getestete Komponente `Clustal` ein entsprechendes Alignment berechnet (vgl. Abbildung 3.10). Der Vergleich dieses Ergebnisses mit dem im Rahmen des Testfalls vorgegebenen Alignment führt zu einer anwendungsbezogenen Bewertung f_{ui} der untersuchten Komponente. Der erhaltene Wert kann gemäß Gleichung 3.7 als Anteil einer anwendungsbezogenen Beurteilung der übergeordneten Variante aufgefaßt werden.

In diesem Zusammenhang erfolgt die eigentliche Berechnung der Teilbewertung f_{ui} durch die eingesetzte Bewertungskomponente, da hierfür Informationen über den Anwendungsbereich bzw. die tatsächlich erwünschten Ergebnisse erforderlich sind. Die Bewertungskomponente selbst ist also zunächst

vom Benutzer zu erstellen, kann anschließend aber im weiteren Verlauf der aktuellen Optimierung sowie für zukünftige Generierung anderer Prototypen erneut verwendet werden. Dies erfordert letztlich nur eine Anpassung an die jeweils getestete Komponente hinsichtlich ihrer Schnittstelle und dem Protokoll ihrer Benutzung. Zu diesem Zweck stellt das Framework jedoch bereits entsprechende Funktionalität zur Vermittlung zwischen unterschiedlichen Repräsentationen oder Interaktionen über geeignete Adapter bereit (siehe Abschnitt 3.6.3 und 5.3.2).

Auch wenn offensichtlich nicht alle Anwendungsfälle der Funktionalen Spezifikation auf diese Weise getestet werden können, so verspricht die Einführung von Bewertungskomponenten doch in den meisten Fällen eine grundlegende Verbesserung des übergeordneten Verfahrens zur Varianten-Optimierung. Beispielsweise läßt sich eine fortschrittliche Strategie umsetzen, die nach einer vorangegangen Optimierung der technischen Merkmale zunächst ausschließlich die automatisiert zu bewertenden, anwendungsbezogenen Merkmale der betreffenden Prototyp-Varianten berücksichtigt. Anschließend muß lediglich die so getroffene Vorauswahl vom Benutzer interaktiv beurteilt werden. Somit verringert sich der insgesamt erforderliche manuelle Aufwand erheblich, ohne die Qualität der erzielten Ergebnisse merklich zu beeinträchtigen. Die hierdurch erreichte Steigerung der Effizienz erleichtert die praxisgerechte Anwendung des vorgestellten Frameworks.

Alternative Heuristiken

Eine derartige Verbesserung der anwendungsbezogenen Bewertung ist grundsätzlich auch für andere heuristische Verfahren zur Varianten-Optimierung vorteilhaft. Schließlich setzt jede Heuristik eine geeignet definierte Zielgröße voraus, die Ablauf und Richtung der Optimierung bestimmt. Tatsächlich existieren neben dem in dieser Arbeit vorgeschlagenen Genetischen Algorithmus zahlreiche weitere Verfahren, die voraussichtlich auf die gegebene Problemstellung anwendbar sind. Daher wird im folgenden untersucht, inwieweit sich derartige Lösungsansätze in das Framework einbinden lassen.

Beispielsweise steht mit *Simulated Annealing* [KGV83] eine Heuristik zur Verfügung, die bereits bei vielfältigen kombinatorischen Optimierungsproblemen erfolgreich eingesetzt wird. Sie basiert ursprünglich auf einem physikalischen Modell des Verhaltens von Flüssigkeiten bei langsamer Abkühlung. Dementsprechend ermittelt Simulated Annealing neue *Konfigurationen* als mögliche Lösungen des gestellten Problems durch Veränderung einzelner Bestandteile der aktuellen Konfiguration, wobei die Wahrscheinlichkeit und Richtung einer solchen *Transition* neben der zugeordneten *Energie*, also letztlich dem zugehörigen Wert der zu minimierenden Zielgröße, auch durch die

aktuelle *Temperatur* als globaler Parameter bestimmt wird. Zu Beginn des Verfahrens werden bei hoher Temperatur auch weitreichende, im Sinne der Optimierung möglicherweise ungünstige Veränderungen toleriert, während später, bei sinkender Temperatur, nurmehr lokale Verbesserungen der betrachteten Lösung durchgeführt werden. Es zeigt sich, daß diese stochastische Heuristik bei vielen kombinatorischen Problemen qualitativ gute Lösungen ermittelt, sofern die Abkühlung hinreichend langsam erfolgt, also entsprechend viele Iterationen durchgeführt werden [Haj88].

Das so beschriebene Verfahren kann mit geringem Aufwand auf die Optimierung generierter Prototyp-Varianten übertragen werden. Hierfür wird der gegebene Konstruktionsplan einer Variante $\vec{x}$ als Konfiguration aufgefaßt, deren Energie umgekehrt proportional zu ihrer in Gleichung 3.6 definierten Gesamtbewertung $f(\vec{x})$ festgelegt wird. Eine Transition im Sinne des Simulated Annealing entspricht dem Austausch einer funktionalen Komponente oder der Repräsentation eines primären Konzepts innerhalb des Konstruktionsplans. Hierbei werden zu Beginn der Optimierung bei niedriger Gesamtbewertung und hoher Temperatur auch Veränderungen zugelassen, die zu einer potentiell schlechteren Lösung führen, etwa weil Komponenten mit geringer logischer Kompatibilität ihrer Anwendungsfälle eingeführt werden. Später, bei verhältnismäßig hoher Gesamtbewertung und niedriger Temperatur werden nur noch lokale Transitionen durchgeführt. Hierfür ist eine geeignete Nachbarschaftsrelation zwischen Komponenten zu definieren, beispielsweise aufgrund semantischer Kompatibilität der manipulierten Konzepte oder gleicher Herkunft der verknüpften Komponenten. Dementsprechend werden nur eng benachbarte Komponenten durch eine Transition gegeneinander ausgetauscht. Zuletzt ist ein dynamischer Ablauf des Simulated Annealing festzulegen, der zumindest zu Beginn eine möglichst geringe Interaktion mit dem Benutzer erfordert, beispielsweise durch ausschließliche Betrachtung technischer Merkmale einer Variante oder den Einsatz geeigneter Bewertungskomponenten.

Wie diese exemplarisch vorgestellte Umsetzung verdeutlicht, erlaubt das erarbeitete Framework eine weitgehend einfach durchzuführende Integration alternativer Heuristiken zur Varianten-Optimierung. Die hierfür benötigten konzeptuellen Elemente, wie Zusammensetzung und Struktur einer Variante oder eine genaue Auffassung ihrer Bewertung, sind klar definiert und im Rahmen der Referenz-Implementierung über entsprechende Schnittstellen zugänglich. Bei Bedarf können darüber hinaus auch weiterführende Informationen, etwa Zusammenhänge der Ontologie oder logische Kompatibilität von Anwendungsfällen, zur Optimierung herangezogen werden. Dies erleichtert die zukünftige Entwicklung und Umsetzung von innovativen, leistungsfähigen Verfahren mit deren Hilfe sich die Qualität der erzielten Ergebnisse weiter

verbessern läßt. So erscheint gerade die Kombination unterschiedlicher Heuristiken vielversprechend, um die Schwächen einzelner Ansätze auszugleichen. Beispielsweise beschreibt [Wen95] eine kooperative Integration von Genetischen Algorithmen und Simulated Annealing, die qualitativ gute Lösungen kombinatorischer Optimierungprobleme in verhältnismäßig geringer Zeit ermittelt.

Zusammenfassend läßt sich feststellen, daß die Varianten-Optimierung einen grundlegenden Aufgabenbereich des Frameworks repräsentiert, der maßgeblich durch umfangreiche praktische Erfahrungen mit realistischen Problemgrößen bestimmt wird. Die aufgeführten Änderungen der bestehenden Heuristik sind daher als initiale Vorschläge aufzufassen, die nach zukünftigen Erkenntnissen geeignet zu überarbeiten sind. Dennoch verdeutlichen sie die Flexibilität und Anpassungsfähigkeit des vorgestellten Ansatzes. Hierbei ist der benötigte Aufwand zur Integration von Bewertungskomponenten oder Modifikation des Genetischen Algorithmus als gering einzuschätzen, da keine grundsätzlichen Änderungen des Frameworks vorzunehmen sind.

Die praktische Umsetzung alternativer Heuristiken, wie Simulated Annealing oder dessen Weiterentwicklungen, erfordert hingegen zwar einen grundsätzlich höheren Aufwand, liefert andererseits aber auch wertvolle Erkenntnisse über den Charakter des zugrundeliegenden Optimierungsproblems. Aufgrund der ausgeprägten logischen Zusammengehörigkeit lokaler Bereiche jeder Prototyp-Variante sowie der inhärenten Unbestimmtheit der Zielgröße, also der tatsächlich erzielten Qualität einer gefundenen Lösung, unterscheidet sich die gestellte Aufgabe doch wesentlich von herkömmlichen Problemen der kombinatorischen Optimierung. Daher beinhaltet dieser Aufgabenbereich einige interessante Fragestellungen für zukünftige Arbeiten.

5.4 Übergreifende Erweiterungen des Frameworks

Während die bisher aufgeführten Erweiterungen jeweils eindeutig bestimmten Bereichen des erarbeiteten Frameworks zugeordnet sind, werden in diesem Abschnitt übergreifende, langfristig bedeutsame Weiterentwicklungen vorgestellt. So verspricht zunächst die Einbeziehung und explizite Modellierung zusätzlich verfügbarer Informationen über Software-Architektur eine wesentliche Verbesserung der erzielten Ergebnisse hinsichtlich ihrer Qualität. Die konsequente Verfolgung dieses Ansatzes führt zu anwendungsbezogenen Interaktionen, die weitgehend unabhängig von technischen Gegebenheiten formuliert und somit als eigenständiges Produkt der Systementwicklung wiederverwendet werden können. Zuletzt wird eine alternative praktische Um-

setzung des Frameworks diskutiert, die eine Zusammenführung von Entwurf und Ausführung der betrachteten Prototypen vorsieht. Hierdurch wird eine weitgehend interaktive und inkrementelle Erstellung funktionaler Prototypen ermöglicht, die zu einer insgesamt engeren Zusammenarbeit zwischen Anwender und Entwickler beiträgt.

Berücksichtigung von Software-Architektur

Das Element *Interaktion* des in Abbildung 3.7 und 3.9 dargestellten Modells eines CUC beschreibt die Realisierung des zugehörigen komponentenbezogenen Anwendungsfalls auf technischer Ebene, wie in Abschnitt 3.4 ausführlich erläutert wird. Hierbei kennzeichnen Rollen die Teilnehmer an einer solchen Interaktion, während ihr Ablauf durch eine angegebene Folge von Operationsaufrufen oder andere, zur Code-Generierung geeignete Beschreibungen festgelegt wird (vgl. Abschnitt 5.2.2). Im Verlauf der Prototyp-Generierung werden Komponenten bzw. deren Instanzen den so definierten Rollen zugeordnet, sowie evtl. erforderliche Parameter durch entsprechende Repräsentationen belegt, um die jeweilige Interaktion schließlich in ausführbaren Quellcode zu übersetzen (siehe Abschnitt 3.6).

Allerdings zeigen erste praktische Erfahrungen mit der Referenz-Implementierung des Frameworks, daß gerade die Zuordnung von Komponenten bei komplexen Interaktionen mit zahlreichen Rollen zu Problemen bei der Varianten-Generierung führt. Für diese grundlegende Aufgabe kann das Framework neben der technischen Kompatibilität beteiligter Schnittstellen lediglich Informationen über möglicherweise im Rahmen der Interaktion referenzierte, externe komponentenbezogene Anwendungsfälle heranziehen. Aus diesem Grund beschränkt sich die gegenwärtige Implementierung auf binäre Interaktionen mit vordefinierten Rollen, die eine merklich vereinfachte Umsetzung erlauben (siehe Abschnitt 3.6.4). Hierdurch folgt die Verknüpfung von Komponenten weitestgehend dem *Client-Server Prinzip*, wobei die als Server auftretende Komponente in der Regel nur wenige Annahmen über den jeweiligen Client voraussetzt. Diese einfache Auffassung einer Interaktion erlaubt bereits eine Generierung zahlreicher, nicht-trivialer Prototypen, wie nicht zuletzt das durchgängige Anwendungsbeispiel belegt.

Dennoch ist es offensichtlich wünschenswert, zukünftig auch komplexe Interaktionen mit mehreren Teilnehmern und weitergehenden Abhängigkeiten möglichst umfassend und zuverlässig zu unterstützen. Neben einer allgemeinen Verbesserung des gegenwärtigen Verfahrens zur Umsetzung einer Interaktion (vgl. Abschnitt 5.3.3), Einsatz leistungsfähiger Interaktionsadapter (vgl. Abschnitt 5.3.2) und Weiterentwicklung der Varianten-Optimierung (vgl. Abschnitt 5.3.4), erscheint hierfür v.a. die Einführung vordefinierter Interaktio-

nen mit übergreifend definierter Semantik vielversprechend. Diese werden im Rahmen einer CUC-Beschreibung zur Umsetzung von Anwendungsfällen referenziert, wobei lediglich die Zuordnung der betreffenden Komponente zu einer bestimmten Rolle der betrachteten Interaktion anzugeben ist. Nunmehr können potentiell geeignete Komponenten über ihren gemeinsamen Bezug zu einer derartigen Interaktion ausgewählt und durch eine Abbildung der beteiligten Rollen auf einfache Weise verknüpft werden.

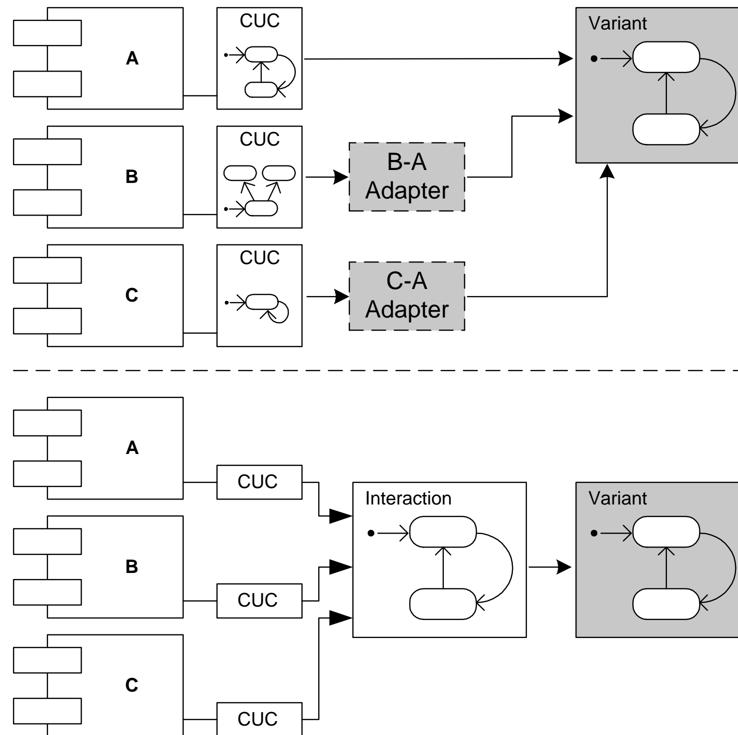


Abb. 5.10: *Lokale und gemeinsam genutzte Interaktionen*

Diese Unterscheidung zwischen lokal definierten Interaktionen und Bezug auf eine gemeinsame Interaktion wird in Abbildung 5.10 an Hand eines schematischen Beispiels erläutert. Im oberen Teil der Abbildung werden drei Komponenten `A`, `B` und `C` über eine als Zustandsautomat spezifizierte Interaktion verknüpft. Diese ist im Beispiel dem von `A` angebotenen CUC entnommen, so daß ihre Umsetzung im allgemeinen entsprechende Interaktionsadapter für die Teilnehmer `B` und `C` erfordert. Demgegenüber beziehen sich im unteren Teil der Abbildung dargestellten Fall sämtliche angebotenen CUCs auf eine gemeinsame, allen Teilnehmern bekannte Interaktion. Diese kann daher für die betrachtete Variante auf einfache und unmittelbare Weise um-

gesetzt werden, sofern der jeweilige Bezug eine entsprechende Zuordnung zu einer Rolle der gemeinsamen Interaktion beinhaltet. Dementsprechend ist die Qualität der später generierten Variante als besonders hoch einzuschätzen, da weder eine geeignete Heuristik zur Ermittlung der korrekten Zuordnung noch zusätzliche Adapter für ihre Realisierung erforderlich sind.

Eine derart eindeutige Spezifikation unterstützter Interaktionen entspricht dem gegenwärtig akzeptierten Verständnis von *Software-Architektur* als möglichst exakte Beschreibung eines Systems hinsichtlich seiner Strukturierung in Komponenten und ihrem Zusammenspiel [SG96]. Hierbei zeigt sich in der Praxis, daß bestimmte Architektur-Varianten oder -Stile häufig zur Lösung gleichartiger Aufgabenstellungen eingesetzt werden, beispielsweise Schichtenarchitekturen zur Konstruktion verteilte Systeme in heterogenen Rechnernetzen [Tan96]. Es erscheint daher vielversprechend, dieses vorhandene Wissen über geeignete Architekturen oder Architekturstile zur Konstruktion funktionaler Prototypen zu nutzen, etwa in Form weitgehend generischer Architektur-Muster [BMR+96] oder spezifischer Architekturen für ausgewählte Anwendungsbereiche [SEI90]. Die hierdurch vorgegebene Spezifikation von Struktur und Interaktion erleichtert die Zuordnung von Komponenten und spätere Generierung ausführbarer Prototypen erheblich, sofern entsprechende Beschreibungen in das Framework integriert werden.

Die praktische Umsetzung einer solchen Erweiterung kann weitgehend evolutionär durchgeführt werden. Die zusätzlich eingeführten Beschreibungen von Architektur sowie der jeweils gültigen Zuordnung funktionaler Komponenten ergänzen hierbei die zuvor in dieser Arbeit vorgestellten Modelle. Auf diese Weise läßt sich die Qualität der erzielten Ergebnisse verbessern, falls zukünftig tatsächlich allen beteiligten Komponenten entsprechende Informationen beigefügt sind oder sogar bestimmte Architekturen als übergreifender Standard eines gegebenen Anwendungsbereichs akzeptiert werden. Andernfalls kann weiterhin das in Abschnitt 3.6 erläuterte Verfahren gleichberechtigt angewendet werden.

Abstrakte Interaktionen

Die oben vorgeschlagene Festlegung auf gemeinsame, vordefinierte Interaktionen bzw. die Einordnung in eine vorgegebene Software-Architektur beschränkt jedoch die möglichst anwendungsbezogene und praxisgerechte Verknüpfung von Komponenten unabhängiger Hersteller. Schließlich kann im allgemeinen Fall gerade *nicht* vorausgesetzt werden, daß diese ausschließlich kompatible Schnittstellen oder gemeinsame Protokolle unterstützen (siehe Abschnitt 3.1.3). Daher ist es überaus lohnenswert, eine weiterführende Auffassung von Interaktion zwischen Komponenten zu entwickeln, die

zunächst eine Abstraktion von konkreten technischen Elementen erlaubt und erst später, gemäß den tatsächlich vorliegenden Verhältnissen, zur Erstellung des gewünschten Prototypen umgesetzt wird.

Zur Spezifikation einer derartigen, *abstrakten Interaktion* können die anwendungsbezogenen Elemente der in Kapitel 3 vorgestellten Modelle herangezogen werden. Die so definierten Konzepte und Anwendungsfälle werden über operative Elemente, wie sequentielle Komposition, Iteration oder Fallunterscheidung, verknüpft, um den dynamischen Ablauf der betrachteten Interaktion eindeutig festzulegen. Auf diese Weise kann die erforderliche Umsetzung der beschriebenen Funktionalität erst später, durch Abbildung auf angebotene Anwendungsfälle tatsächlich verfügbarer Komponenten erfolgen. Die zugehörigen, *technischen Interaktionen* zur Realisierung dieser Anwendungsfälle sind dementsprechend einfach zu gestalten, damit eine weitgehend problemlose Generierung gewährleistet ist.

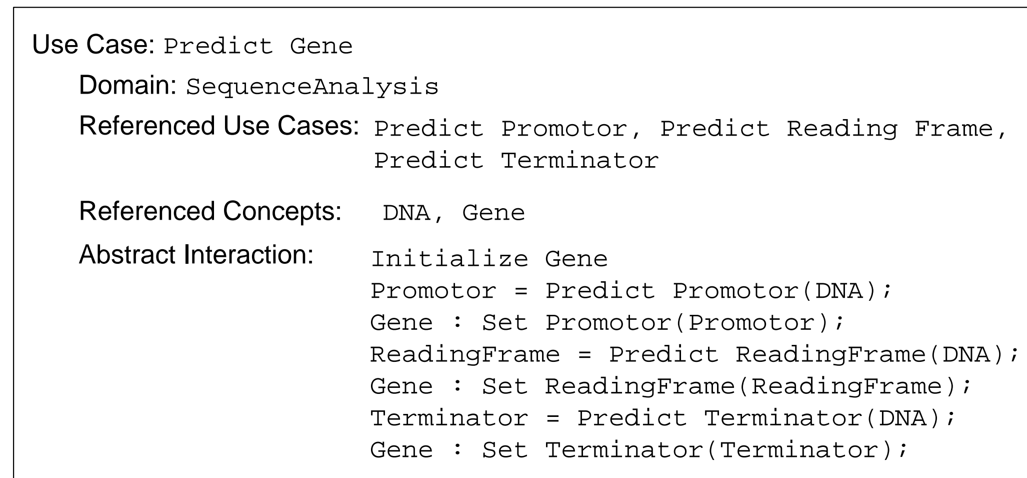

Use Case: `Predict Gene`
Domain: `SequenceAnalysis`
Referenced Use Cases: `Predict Promotor, Predict Reading Frame, Predict Terminator`
Referenced Concepts: `DNA, Gene`
Abstract Interaction:
```
Initialize Gene
Promotor = Predict Promotor(DNA);
Gene : Set Promotor(Promotor);
ReadingFrame = Predict ReadingFrame(DNA);
Gene : Set ReadingFrame(ReadingFrame);
Terminator = Predict Terminator(DNA);
Gene : Set Terminator(Terminator);
```

Abb. 5.11: *Abstrakte Interaktion zum Anwendungsfall* Predict Gene

Abbildung 5.11 verdeutlicht diesen Ansatz an Hand einer exemplarischen abstrakten Interaktion bei struktureller Analyse einer gegebenen DNA-Sequenz. Sie beschreibt die sequentielle Komposition der Anwendungsfälle Predict Promotor, Predict ReadingFrame und Predict Terminator zur Ermittlung der entsprechenden, für die Transkription eines Gens maßgeblichen genetischen Elemente (vgl. Abschnitt 2.3.1). Die Position und Ausdehnung dieser Elemente kann aus der DNA-Sequenz eines Gens vorhergesagt werden, wobei in der Regel weiterführende Informationen über den jeweiligen Organismus und andere, bereits bekannte Gene herangezogen werden [Wat95]. Daher ist

diese wesentliche Funktionalität von geeigneten Softwarekomponenten zu erbringen, die im weiteren Verlauf für die betreffenden Anwendungsfälle ausgewählt werden. Hingegen beschreibt Abbildung 5.11 eine zwar einfache, aber dennoch im betrachteten Anwendungsbereich durchaus sinnvolle und logisch zusammengehörige Interaktion, welche selbst wiederum als eigener Anwendungsfall Predict Gene gekennzeichnet werden kann. Dieser läßt sich zukünftig gleichberechtigt bei Spezifikation funktionaler Anforderungen angegeben, sofern das eingesetzte Verfahren zur Prototyp-Generierung entsprechend angepaßt wird (vgl. Abschnitt 3.6).

Somit wird insgesamt eine konzeptuell deutliche Unterscheidung zwischen technischer und anwendungsbezogener Interaktion erreicht, die einen höheren Grad der Wiederverwendung auf logischer Ebene des Frameworks verspricht. Darüber hinaus besteht durch geeignet eingeführte, operative Elemente die Möglichkeit, komplexe dynamische Abläufe weitgehend unabhängig von konkreten technischen Details der Implementierung anzugeben. Dies erleichtert die Umsetzung einer so spezifizierten abstrakten Interaktion erheblich, falls die technische Realisierung referenzierter Anwendungsfälle nunmehr auf einfache Abläufe beschränkt werden kann. Diese Einschätzung wird durch die in Abschnitt 5.3.2 vorgeschlagenen Strategien zur Adapter-Generierung gestützt, da solche Strategien ebenfalls als abstrakte Interaktionen aufzufassen sind.

Die Integration einer derartigen, langfristig bedeutsamen Erweiterung in das vorgestellte Framework kann wiederum evolutionär erfolgen, wobei abstrakte Interaktionen zunächst lediglich als Ergänzung der bestehenden Modelle eingeführt werden. So kann beispielsweise der in Abbildung 5.11 dargestellte Anwendungsfall Predict Gene in vergleichbarer Weise auch durch eine eigene Softwarekomponente angeboten werden, deren Implementierung eine entsprechende Verknüpfung der benötigten Funktionalität gewährleistet. Letztlich ist es Aufgabe der Prototyp-Generierung, die verschiedenen Alternativen zur Umsetzung eines gegebenen Anwendungsfalls gegeneinander abzuwägen. Dies wird nicht zuletzt durch eine gleichartige Definition der eingeführten operativen Elemente für Funktionale Spezifikation, Beschreibung von Interaktion sowie Strategien der Adapter-Generierung wesentlich erleichtert. Somit lassen sich auch entsprechende Verbesserungen der Beschreibungstechnik und Werkzeugunterstützung übergreifend nutzen (siehe Abschnitt 5.2.2 und 5.3.2).

Simulation und interaktive Erstellung funktionaler Prototypen

Das in Abschnitt 3.6 erläuterte Verfahren zur Erstellung funktionaler Prototypen beinhaltet eine weitgehend sequentielle Abfolge von Spezifikation

der gewünschten Funktionalität, Auswahl geeigneter Komponenten sowie Generierung und Bewertung entsprechender Prototyp-Varianten (vgl. Abbildung 3.17). Zwar sind der übergeordnete Prozeß und ausgewählte Teile, wie etwa die Varianten-Optimierung, grundsätzlich iterativ organisiert, jedoch führen Änderungen der Funktionalen Spezifikation in der gegenwärtigen Implementierung zu einem erneuten, vollständig durchgeführten Ablauf des gesamten Verfahrens. Diese Festlegung wird getroffen, um einerseits die wesentlichen Aufgabenbereiche, Zwischenergebnisse und entwickelten Lösungsansätze möglichst anschaulich zu vermitteln, und andererseits die erarbeitete praktische Umsetzung des vorgestellten Frameworks zu vereinfachen.

Eine vielversprechende, übergreifende Weiterentwicklung des Ansatzes verschränkt jedoch die vorgesehenen Teilabläufe und deren Ergebnisse, um die strikte Trennung zwischen Spezifikation, Erstellung und Bewertung möglicher Lösungen aufzuheben. Dies erlaubt eine weitgehend interaktive und inkrementelle Konstruktion funktionaler Prototypen, welche dem besonderen Charakter des explorativen Rapid Prototyping besser gerecht wird. Insbesondere können somit Anforderungen oder Beurteilungen des späteren Anwenders frühzeitig berücksichtigt werden, falls der übergeordnete Prozeß in enger Zusammenarbeit mit dem Entwickler durchgeführt wird.

Abbildung 5.12 verdeutlicht eine derartige Erweiterung des Frameworks durch einen schematischen Vergleich der unterschiedlichen Vorgehensweisen. Im oberen Teil der Abbildung ist der bisherige Ablauf des Verfahrens im Überblick dargestellt (vgl. Abbildung 3.17). Die vom Benutzer erstellte Funktionale Spezifikation dient als Eingabe für den aus Komponenten-Auswahl, Adapter-Generierung, Varianten-Generierung und Varianten-Optimierung zusammengesetzten Prozeß. Hierbei wird bereits eine in Abschnitt 5.2.2 erläuterte Auffassung der Funktionalen Spezifikation als besonderer Workflow mit unterscheidbaren Zuständen, Zwischenergebnissen und durch Anwendungsfälle gegebenen Arbeitsschritten angenommen. Dies wird in der Abbildung durch entsprechende grafische Symbole angedeutet. Das Ergebnis des Verfahrens ist eine Menge funktionaler Prototypen, welche die gewünschte Funktionalität möglichst weitgehend erfüllen. Der Benutzer kann dieses Ergebnis lediglich durch eine interaktiv vorgenommene Bewertung während der Varianten-Optimierung indirekt beeinflussen. Umfassende Änderungen erfordern eine Anpassung der Funktionalen Spezifikation sowie eine erneute Iteration des gesamten Ablaufs.

Demgegenüber erlaubt die im unteren Teil der Abbildung dargestellte, inkrementelle Konstruktion der betrachteten Prototypen ein wesentlich höheres Maß an Interaktion mit dem Benutzer. Hierbei lassen sich ausgewählte Ausschnitte der Funktionalen Spezifikation, etwa logisch zusammengehörige Bereiche der erwünschten Funktionalität, auch zunächst getrennt behandeln.

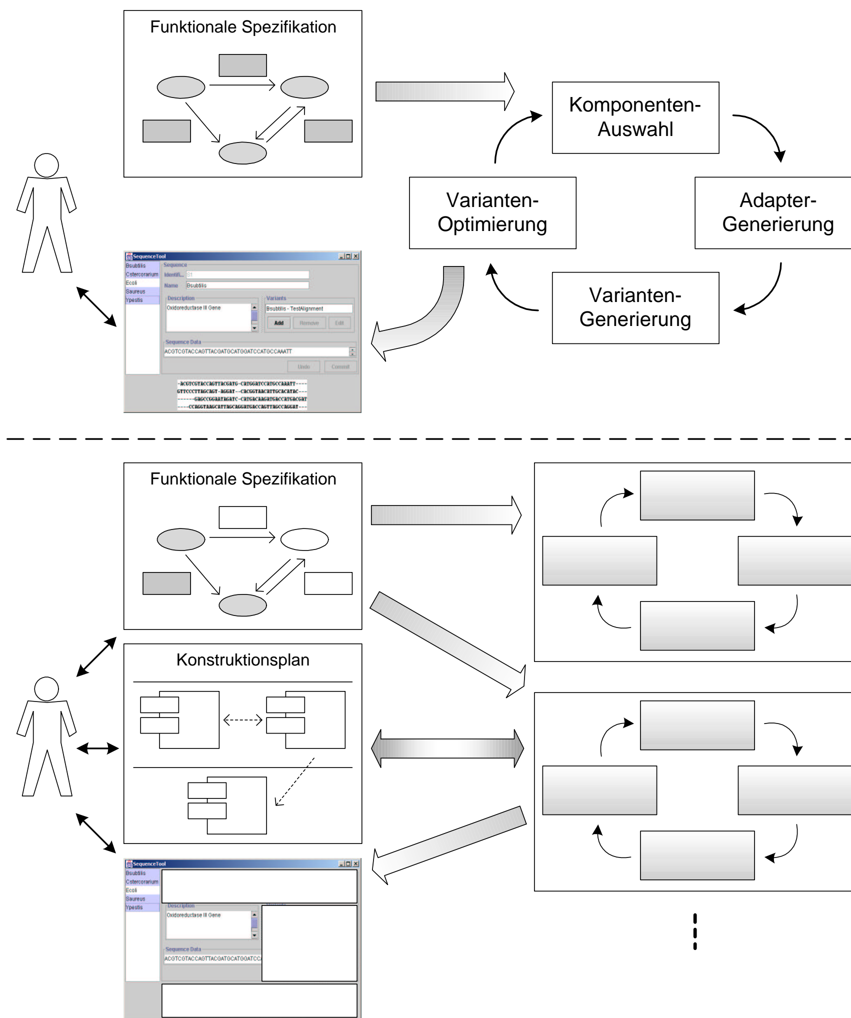


Abb. 5.12: *Sequentielle und interaktive Erstellung funktionaler Prototypen*

Dies führt in der Folge zu partiellen Prototypen, deren Konstruktionsplan durch den Benutzer interaktiv verändert werden kann. Auf diese Weise werden vorgeschlagene Komponenten, deren Zusammenspiel, sowie evtl. herangezogene Architekturmuster ergänzt oder durch besser geeignete Elemente ersetzt. Die so erhaltenen Informationen dienen neben der anwendungsbezogenen Beurteilung zur Verbesserung der erzielten Ergebnisse im Verlauf weiterer lokaler Iterationen des Verfahrens. Begleitend kann die Funktionale Spezifikation vervollständigt oder gemäß den gewonnenen Erkenntnissen modifiziert werden.

Eine derartige Erweiterung verspricht entscheidende Vorteile für die praxisgerechte Anwendung des vorgestellten Frameworks. So wird die Qualität der generierten Prototypen deutlich verbessert, da möglicherweise auftretende technische Schwierigkeiten, wie fehlerhafte Zuordnung von Interaktionspartnern oder fehlende Belegung von Parametern, bei Bedarf durch gezielten Eingriff des Benutzers korrigiert werden können. Weiterhin erlaubt die inkrementelle Konstruktion eine effiziente Ausnutzung der verfügbaren Ressourcen, etwa falls weitgehend unabhängige Teilbereiche parallel behandelt werden. Zuletzt entspricht der interaktive Ablauf dem grundsätzlich experimentellen Charakter des erarbeiteten Ansatzes für komponentenbasiertes Rapid Prototyping. Die bereits vorhandenen Komponenten werden auf möglichst einfache und anwendungsbezogene Weise zur Erfüllung vorgegebener Funktionalität verknüpft, wobei der erforderliche manuelle Aufwand je nach Aufgabenstellung bzw. Genauigkeit der funktionalen Anforderungen flexibel angepaßt werden kann.

Allerdings ist die praktische Umsetzung dieser übergreifenden Erweiterung mit durchaus erheblichem Aufwand verbunden, da geeignete Lösungen für eine möglichst effektive Interaktion mit dem Benutzer entwickelt und implementiert werden müssen. Für diese Aufgabe können jedoch Erfahrungen mit bestehenden visuellen Werkzeugen zur interaktiven Verknüpfung von Komponenten herangezogen werden [IBM01b, NAG01]. Darüber hinaus ist ein Übergang von weitgehend statischer Generierung hin zur dynamischen Simulation funktionaler Prototypen erforderlich, damit auch partielle Prototyp-Varianten ausgeführt und beurteilt werden können. Hierfür ist eine geeignete Simulationsumgebung zu realisieren, die alle benötigten Elemente, wie Komponenten, Adapter oder Testfälle, flexibel zur Laufzeit des Prototypen integriert und gemäß den Ergebnissen des übergeordneten Verfahrens verknüpft. Auf diese Weise lassen sich auch gegenwärtige technische Mängel der Varianten-Generierung, etwa die adequate Behandlung extern beobachtbarer Zustände einer Komponente oder die umfassende Unterstützung asynchroner Kommunikation über Ereignisse (vgl. Abschnitt 3.6.4), mit geringem zusätzlichen Aufwand lösen. Daher repräsentiert der Übergang zur Simulation von Prototypen eine besonders interessante und lohnenswerte Aufgabe für zukünftige Weiterentwicklungen des vorliegenden Ansatzes.

Die in diesem Abschnitt zusammengefaßten, übergreifenden Erweiterungen betreffen auf längere Sicht nahezu alle Bereiche auf technischer Ebene des Frameworks. Sie erlauben eine wesentliche Verbesserung der erzielten Ergebnisse, auch wenn deren praktische Umsetzung teilweise mit erheblichem Aufwand verbunden ist. Während die Berücksichtigung von Informatio-

nen über Software-Architektur sowie die Integration abstrakter Interaktionen noch weitgehend evolutionär durchgeführt werden kann, erfordert der Übergang zur interaktiven und inkrementellen Entwicklung funktionaler Prototypen eine vollständige Änderung der in dieser Arbeit vorgestellten Verfahren zu deren Generierung.

Dennoch lassen sich die eingeführten Modelle und Verfahren auf logischer Ebene des Frameworks weiterhin zur anwendungsbezogenen Beschreibung, Ermittlung und Verknüpfung angebotener oder erwünschter Funktionalität einsetzen. Diese Tatsache resultiert aus der grundlegenden Konzeption des Frameworks mit klar definierten Aufgaben sowie problembezogenen Abhängigkeiten zwischen den jeweils erhaltenen Teilergebnissen. Somit verdeutlichen die vorgeschlagenen Erweiterungen auf besondere Weise die Modularität des erarbeiteten Ansatzes.

5.5 *Zusammenfassung*

Erweiterbarkeit und Anpassungsfähigkeit repräsentieren wesentliche Merkmale des vorgestellten Frameworks für komponentenbasiertes Rapid Prototyping. Die aufgeführten Erweiterungen ermöglichen eine höhere Qualität der erstellten Prototypen, die Integration bestehender Beschreibungstechniken und Werkzeuge, sowie eine insgesamt praxisgerechte Anwendung der erarbeiteten Modelle und Verfahren. Hierbei erlaubt die vorgegebene Strukturierung des Frameworks eine klare Zuordnung der vorgeschlagenen Ansätze zu maßgeblichen Teilaufgaben der übergeordneten Lösung.

Im Bereich der Ontologie können Constraints als Prädikate über Konzepte und Relationen eingeführt werden, um komplexe Verhältnisse im Anwendungsbereich angemessen zu modellieren. Ihre Auswertung beeinflußt die Ableitung weiterführender Beziehungen und ermöglicht die Spezifikation übergreifender Konsistenzbedingungen. Somit wird die Expressivität der Ontologie wesentlich verbessert, auch wenn die Komplexität der resultierenden Modelle deutlich höher einzuschätzen ist.

Die vorgeschlagene Integration von Manipulationen in die Ontologie führt zu einer insgesamt durchgängigen und systematischen Behandlung aller wesentlichen Elemente zur Beschreibung von Funktionalität. Die hierdurch erreichte, flexible Zuordnung zwischen Manipulationen und Konzepten erlaubt eine anwendungsbezogene und weitgehend redundanzfreie Modellierung zusammengehöriger Funktionalität. Darüber hinaus lassen sich festgelegte Relationen zwischen Manipulationen für eine erhöhte Toleranz bei späterer Auswahl von Komponenten nutzen.

Die Erstellung und Pflege einer Ontologie wird durch geeignete Beschreibungstechniken und Werkzeuge erleichtert. Hierbei bietet sich ein UML-Klassendiagramm als bekannte, überwiegend grafische Notation an, da somit existierende CASE-Werkzeuge zu ihrer Bearbeitung eingesetzt werden können. Diese Werkzeuge müssen allerdings in ihrer Funktionalität angepaßt und erweitert werden, um die dynamisch ermittelten Zusammenhänge einer Ontologie zu berücksichtigen.

Demgegenüber wird Austausch und Integration unabhängig erstellter Ontologien durch eine textuelle, beispielsweise XML-basierte Notation zu deren Beschreibung vereinfacht. Somit können Teile des erstellten Modells wiederverwendet oder als lokal vorausgesetzte Annahmen über den Anwendungsbereich dem ausführbaren Format einer Komponente beigefügt werden. Die so erreichte Unabhängigkeit von einer zentral festgelegten Ontologie verspricht zwar mehr Flexibilität bei Kombination verfügbarer Komponenten, erfordert andererseits aber weiterführende Verfahren zur Integration unterschiedlicher Modelle des gleichen Anwendungsbereichs.

Die Expressivität der Funktionalen Spezifikation und komponentenbezogener Anwendungsfälle kann durch Einführung zusätzlicher Konzepte der Ontologie bei Beschreibung von erwünschter oder angebotener Funktionalität verbessert werden. Diese werden über festgelegte Präpositionen mit Manipulationen und primären Konzepten verknüpft, um die Möglichkeiten der natürlichen Sprache nachzuahmen. Allerdings wird hierdurch je nach Umfang der Erweiterung die spätere Generierung funktionaler Prototypen erschwert. Daher ist bei entsprechenden Lösungsansätzen eine Abwägung zwischen gesteigerter Expressivität und erhöhter Komplexität des Verfahrens erforderlich.

Auf technischer Ebene des Frameworks lassen sich auch komplexe dynamische Abläufe durch zusätzlich eingeführte operative Elemente, wie Iteration, Fallunterscheidung oder Bedingung, beschreiben. Sie können im Rahmen jeder gängigen technischen Plattform unmittelbar umgesetzt werden. Darüber hinaus erlaubt die Modellierung eines CUC auch die Integration alternativer Beschreibungen von Interaktion, beispielsweise die Angabe von Zustandsautomaten. Hierfür sind die Verfahren zur Prototyp-Generierung geeignet anzupassen. In jedem Fall ist die Verwendung bekannter grafischer Beschreibungstechniken, wie Sequenz- oder Zustandsdiagramme der UML, zur vereinfachten Erstellung und Bearbeitung der entsprechenden Modelle anzustreben.

Obwohl die genannten operativen Elemente auch zur Erweiterung der Funktionalen Spezifikation eingesetzt werden können, erscheint langfristig

eine weiterführende Auffassung als anwendungsbezogener Workflow besonders vielversprechend. Hierbei entsprechen Anwendungsfälle der Spezifikation grundlegenden Arbeitsschritten des Workflows, die über Zwischenergebnisse sowie temporale und kausale Abhängigkeiten verknüpft sind. Mit Hilfe dieser Interpretation lassen sich bestehende Beschreibungstechniken und Werkzeuge der Workflow-Modellierung auch für den vorgestellten Ansatz nutzen. Dies betrifft insbesondere die Integration von Workflow Management Systemen für den späteren Übergang zur Simulation funktionaler Prototypen.

Das übergeordnete Verfahren zur Generierung funktionaler Prototypen gliedert sich in die grundlegenden Aufgabenbereiche Komponenten-Auswahl, Adapter-Generierung, Varianten-Generierung und Varianten-Optimierung. Hierbei besteht zunächst die Zielsetzung in einer möglichst flexiblen und toleranten Auswahl potentiell geeigneter Komponenten für gegebene Anwendungsfälle der Funktionalen Spezifikation. Zu diesem Zweck wird ein Begriff der erweiterten semantischen Kompatibilität zwischen Konzepten der Ontologie eingeführt, der auf einer Übereinstimmung ihrer festgelegten Beziehungen basiert. Somit können logisch kompatible Anwendungsfälle ermittelt werden, auch wenn die jeweils manipulierten Konzepte nicht über Generalisierung oder Interpretation in Beziehung gesetzt sind. Darüber hinaus lassen sich im Rahmen der Ontologie definierte Relationen zwischen Manipulationen, beispielsweise die Implikation, für eine weiterführende Bestimmung der logischen Kompatibilität einsetzen.

Die anschließende Verknüpfung von Komponenten unterschiedlicher Hersteller erfordert im allgemeinen die Integration entsprechender Adapter für Repräsentationen eines Konzepts oder Teilnehmer einer Interaktion. In diesem Zusammenhang kann eine weitergehende Automatisierung durch die Einführung von Strategien erreicht werden, die mit Hilfe geeigneter operativer Elemente einen abstrakten Algorithmus zur Konvertierung von Repräsentationen bzw. deren Zustand beschreiben. Sie beziehen sich ausschließlich auf Konzepte und Manipulationen der Ontologie, die erst später auf die tatsächlich beteiligten Komponenten und deren angebotene Funktionalität abgebildet werden. Hierdurch wird eine weitreichende Anwendbarkeit und Wiederverwendung der betreffenden Strategie erzielt. Hingegen können zusätzlich eingeführte Interaktionsadapter zur Vermittlung zwischen unterschiedlichen Teilnehmern einer Interaktion in der Regel nicht automatisch erstellt werden. Dennoch lohnt sich der benötigte manuelle Aufwand, da somit eine deutlich höhere Anzahl unabhängig entwickelter Komponenten zur Konstruktion herangezogen werden kann.

Die Übersetzung des fertiggestellten Konstruktionsplans einer Prototyp-

Variante in ausführbaren Quellcode der verwendeten technischen Plattform erfordert im wesentlichen eine vollständige Umsetzung der jeweils ausgewählten Interaktionen hinsichtlich ihrer Belegung für spezifizierte Rollen und Parameter. Diese Zuordnung kann durch unterschiedliche Strategien bzw. deren Implementierung erfolgen, die als eigener Bestandteil des Frameworks definiert und eingesetzt werden. Somit lassen sich auch neuartige Strategien, wie adaptive oder interaktive Verfahren, in den übergeordneten Ansatz integrieren. Eine ergänzende Aufgabe der Varianten-Generierung ist die Erstellung einer geeigneten grafischen Oberfläche zur Interaktion mit dem Benutzer und Anzeige visueller Komponenten. Dies wird durch existierende Werkzeuge zur Gestaltung von GUI-Prototypen sowie eine deklarative Beschreibung der spezifizierten Oberfläche vereinfacht.

Der Vielzahl an unterschiedlich zusammengesetzten Prototyp-Varianten wird durch eine geeignete Heuristik zu ihrer Optimierung begegnet. Der hierfür in dieser Arbeit vorgeschlagene Genetische Algorithmus kann an Hand zukünftiger praktischer Erfahrungen deutlich verbessert werden. Beispielsweise erlauben Modifikationen der Genetischen Operatoren die gezielte Einführung von Redundanz oder eine gemeinsame Behandlung logisch zusammengehöriger Bereiche einer Variante im Verlauf der Optimierung. Zudem kann die verwendete Heuristik auch vollständig durch ein anderes Verfahren, wie etwa Simulated Annealing, ersetzt werden, weil die grundlegende Problemstellung und erforderlichen Informationen im Rahmen des Frameworks eindeutig definiert sind. In jedem Fall ermöglicht eine weitergehende Automatisierung der anwendungsbezogenen Bewertung einer Variante erhebliche Verbesserungen der insgesamt erzielten Effizienz. Aus diesem Grund wird die Einführung von Bewertungskomponenten vorgeschlagen, die Testfälle für ausgewählte Anwendungsfälle implementieren. Der Vergleich zwischen erwarteten und erhaltenen Ergebnissen nach Ausführung des getesteten Anwendungsfalls liefert die Grundlage zu dessen Bewertung. Somit kann in diesen Fällen auf eine vom Benutzer vorgenommene Beurteilung verzichtet werden.

Neben den lokalen Erweiterungen einzelner Bereiche werden auch übergreifende Weiterentwicklungen vorgeschlagen, die v.a. auf technischer Ebene des Frameworks zu langfristig bedeutsamen Veränderungen führen. So erlaubt die Einbeziehung zusätzlich verfügbarer Informationen über Software-Architektur eine erheblich vereinfachte Zuordnung von Teilnehmern einer Interaktion und mithin eine verbesserte Qualität der generierten Prototypen. Hierbei werden eine gemeinsame Struktur des Sytems und übergreifend bekannte Interaktionen vorausgesetzt, auf die sich verfügbare Komponenten

beziehen. Diese können somit ohne weitergehende technische Schwierigkeiten verknüpft werden.

Die konsequente Verfolgung dieses Ansatzes führt zum Begriff der abstrakten Interaktion, die ausschließlich mit Hilfe von Konzepten, Manipulationen und geeigneten operativen Elementen beschrieben wird. Vergleichbar mit Strategien zur Adapter-Generierung erfolgt eine Abbildung auf angebotene Funktionalität verfügbarer Komponenten erst später, bei Kenntnis der tatsächlich vorliegenden Verhältnisse. Auf diese Weise lassen sich anwendungsbezogene Interaktionen spezifizieren, ohne technischen Details der Implementierung berücksichtigen zu müssen. Dies erlaubt eine höhere Flexibilität und weitreichende Wiederverwendung bei Umsetzung einer solchen abstrakten Interaktion.

Zuletzt wird eine alternative praktische Umsetzung des vorgestellten Frameworks diskutiert. Sie verschränkt Anwendung und zeitliche Reihenfolge der eingesetzten Verfahren, um einen Übergang von statischer Generierung zur dynamischen Simulation funktionaler Prototypen zu erreichen. Hierbei können durch eine geeignete Umgebung auch partiell fertiggestellte Prototypen ausgeführt, bewertet und durch den Benutzer verändert werden. Dies erlaubt einen weitgehend interaktiven und inkrementellen Konstruktionsprozeß, welcher dem besonderen Charakter des explorativen Prototyping besser gerecht wird.

Die vorgeschlagenen Erweiterungen ergänzen die in Kapitel 3 beschriebenen Ergebnisse und verdeutlichen so die Flexibilität und Anpassungsfähigkeit des vorgestellten Frameworks für komponentenbasiertes Rapid Prototyping. Aufgrund der vorgegebenen Strukturierung mit klar definierten Aufgabenbereichen, Zwischenergebnissen sowie deren Abhängigkeiten kann ein Großteil der aufgeführten Vorschläge verhältnismäßig einfach und mit vertretbarem Aufwand in die Praxis umgesetzt werden. Diese Tatsache unterstreicht das Potential der erarbeiteten Lösung und wird daher in die folgende Diskussion des erreichten Erfolgs einbezogen.

6. DISKUSSION

Nach ausführlicher Erläuterung des konzeptuellen Frameworks für komponentenbasiertes Rapid Prototyping als zentrales Ergebnis der vorliegenden Arbeit, wird in diesem Kapitel der erreichte Erfolg sowie Bedeutung und Umfang des eigenständigen Beitrags diskutiert. Hierfür werden zunächst die in Abschnitt 3.1 aufgeführten Anforderungen an einen derartigen Ansatz herangezogen. Sie repräsentieren wesentliche Aspekte der in Kapitel 1 beschriebenen, übergeordneten Aufgabenstellung, die unter Einbeziehung erster praktischer Erfahrungen mit der erstellten Referenz-Implementierung genauer untersucht werden. Für eine umfassende Diskussion sind weiterhin die in Kapitel 5 vorgeschlagenen Erweiterungen des Frameworks angemessen zu berücksichtigen, weil diese auf besondere Weise die Flexibilität und das zukünftige Potential der vorgestellten Lösung verdeutlichen.

Der anschließende Vergleich mit bestehenden Ansätzen erlaubt eine Beurteilung der mit dieser Arbeit verbundenen Innovation und Verbesserung des aktuellen Stands der Technik. Darüber hinaus wird die Bezug und Einordnung bekannter Forschungsergebnisse in ausgewählten Bereichen des Frameworks erläutert, um bedeutsame Unterschiede und gemeinsame Merkmale herauszustellen. Dies erleichtert eine Abschätzung von Plausibilität und Qualität der erarbeiteten Modelle und Verfahren. Zuletzt werden die wesentlichen Erkenntnisse der Diskussion am Ende des Kapitels zusammengefaßt und der erarbeitete Ansatz abschließend bewertet.

6.1 Erfüllung der gestellten Anforderungen

Die in Abschnitt 3.1 beschriebenen Anforderungen an einen praxisgerechten Ansatz für komponentenbasiertes Rapid Prototyping repräsentieren eine Strukturierung und Konkretisierung der übergeordneten Problemstellung. Die Gesamtheit aller betrachteten Anforderungen führt zur Konzeption eines Frameworks, dessen Bestandteile bei Bedarf geeignet zu verändern oder zu erweitern sind. Aus diesem Grund werden im folgenden neben den zentralen Ergebnissen aus Kapitel 3 auch die in Kapitel 5 vorgeschlagenen Erweiterungen zur Diskussion herangezogen.

6.1.1 *Expressivität*

Die Ausdruckskraft der angebotenen Mittel zur Beschreibung von erwünschter bzw. angebotener Funktionalität bestimmt maßgeblich über eine zielführende Verknüpfung funktionaler Softwarekomponenten zu übergeordneten Systemen (vgl. Abschnitt 3.1.1). Hierfür entwickelt der vorliegende Ansatz das zentrale Modell eines komponentenbezogenen Anwendungsfalls als zusammengehörige Kombination anwendungsbezogener und technischer Elemente, wie in Abschnitt 3.4 ausführlich erläutert wird.

Auf anwendungsbezogener Ebene wird Funktionalität als Manipulation von Konzepten einer Ontologie aufgefaßt. Diese Kombination aus aktiven und passiven Elementen ähnelt einem einfachen Hauptsatz der natürlichen Sprache, wobei zusätzliche Bestandteile über festgelegte Präpositionen und qualifizierende Bezeichner eingeführt werden können (siehe Abschnitt 5.2.1). Somit läßt sich prinzipiell jede elementare Funktionalität angemessen beschreiben, die auch als entsprechend zusammengesetzter Hauptsatz in Prosa formuliert werden kann.

Demgegenüber ist eine Beschreibung zusammengesetzter Funktionalität im Rahmen der Funktionalen Spezifikation auf einfache Konstrukte, wie sequentielle Komposition, Iteration oder Fallunterscheidung, beschränkt (siehe Abschnitt 3.5 und 5.2.1). Hierbei wird die Verknüpfung unterschiedlicher Anwendungsfälle im wesentlichen über entsprechend gekennzeichnete Zwischenergebnisse vorgenommen. Allenfalls die zeitliche Abhängigkeit einzelner Schritte läßt sich durch zukünftige Integration einer Workflow-basierten Modellierung genauer und flexibler ausdrücken.

Auf diese Weise wird die Expressivität natürlicher Sprache offensichtlich nicht erreicht, da zahlreiche verbindende Informationen über kausale Abhängigkeiten, vorausgesetzte Annahmen sowie implizite oder explizite Konsistenzbedingungen des jeweiligen Anwendungsbereichs unberücksichtigt bleiben. Andererseits führt die möglichst genaue Abbildung derartig komplexer Verhältnisse zu entsprechend komplizierten und umfangreichen Modellen, die sich nur mit unverhältnismäßig großem Aufwand erstellen, pflegen und verwerten lassen.

Weiterhin ist der tatsächlichen Unbestimmtheit funktionaler Anforderungen an das zu entwickelnde System Rechnung zu tragen. Die erstellten Prototypen dienen ja gerade der Ermittlung und Konkretisierung erwünschter Funktionalität zu Beginn des Entwicklungsprozesses, wie in Abschnitt 2.1 ausführlich erläutert wird. Daher erscheint eine vereinfachte Modellierung zusammengesetzter Funktionalität ausreichend und angemessen, sofern die Beschreibung elementarer Funktionalität zumindest grundlegende Verhältnisse im betrachteten Anwendungsbereich hinreichend genau wiedergibt.

Auf technischer Ebene eines komponentenbezogenen Anwendungsfalls wird Funktionalität als Interaktion zwischen Softwarekomponenten aufgefaßt. Hierbei beschränken sich die gegenwärtig angebotenen Mittel zur Spezifikation einer derartigen Interaktion auf einfache Folgen von Operationsaufrufen der beteiligten Schnittstellen. Diese Modellierung erlaubt zwar bereits die Beschreibung zahlreicher nicht-trivialer Interaktionen, jedoch erfordert die Abbildung komplexer dynamischer Abläufe entsprechend ausdrucksvolle Modelle. Der vorgestellte Ansatz unterstützt solche Erweiterungen des mit einer Interaktion verbundenen Protokolls, sofern deren effektive und effiziente Umsetzung im Verlauf der Prototyp-Generierung gewährleistet ist (siehe Abschnitt 5.2.2).

Unter den oben genannten Gesichtspunkten werden die in Abschnitt 3.1.1 gestellten Anforderungen an Expressivität durch den vorliegenden Ansatz im wesentlichen gut erfüllt. Die in diesem Zusammenhang maßgebliche Auffassung von anwendungsbezogener Funktionalität als Manipulation von Konzepten entspricht einer durchaus typischen Benennung herkömmlicher Anwendungsfälle in der Anforderungsanalyse (vgl. Abschnitt 3.4). Diese Übereinstimmung ist ein deutlicher Hinweis auf die Eignung des erarbeiteten Modells zur abstrakten Beschreibung grundlegender Funktionalität. Darüber hinaus erlaubt die definierte Semantik des in Abschnitt 3.3 eingeführten Metamodells einer Ontologie die Ableitung weiterführender Aussagen über verwandte Konzepte. Auf diese Weise kann auch unterschiedliche, aber doch ähnliche Funktionalität angemessen ausgedrückt und ermittelt werden.

Hingegen ist die Expressivität des vorgeschlagenen Modells zur Beschreibung von Interaktion auf technischer Ebene des Frameworks nur bei überwiegend einfachen Verhältnissen ausreichend. Die entsprechenden Erweiterungen in diesem Bereich sind jedoch mit vertretbarem Aufwand zu integrieren, insbesondere falls hierfür wohlbekannte, unmittelbar zur Generierung geeignete Modelle von Verhalten herangezogen werden. Zudem sind die anwendungsbezogenen Teile des Frameworks von derartigen Veränderungen nicht betroffen. Somit kann der Abgleich zwischen erwünschter und angebotener Funktionalität weiterhin mit Hilfe des in Abschnitt 3.6.2 erläuterten Verfahrens erfolgen.

6.1.2 Komplexität

Die Komplexität des eingesetzten Modells zur Beschreibung von Funktionalität bestimmt ebenfalls entscheidend über den praktischen Erfolg einer vorgeschlagenen Lösung für komponentenbasiertes Rapid Prototyping. Ausgeprägt formale Modelle mit eindeutig definierter Syntax und Semantik erlauben die Angabe und Ableitung exakter Informationen über wichtige Merk-

male und Eigenschaften des betrachteten Systems oder dessen Bestandteile. Hingegen sind überwiegend informelle Ansätze in der Regel einfacher zu handhaben, besser verständlich und somit leichter in die praktische Systementwicklung zu übertragen.

Die vorliegende Arbeit repräsentiert in dieser Hinsicht einen pragmatischen Kompromiß zwischen beiden Extremen. Auf logischer Ebene des Frameworks wird Funktionalität als Manipulation wohlbekannter Konzepte des Anwendungsbereichs aufgefaßt. Diese vereinfachte, weitgehend deklarative Modellierung ist sowohl für den Entwickler als auch für den Anwender intuitiv verständlich, da sie den Möglichkeiten der natürlichen Sprache nachempfunden ist. Hierbei ergibt sich eine gewisse Toleranz bei Formulierung von erwünschter bzw. angebotener Funktionalität, die einerseits zur flexiblen Auswahl potentiell geeigneter Komponenten ausgenutzt wird, andererseits aber auch zu tatsächlich ungeeigneten oder sogar fehlerhaften Prototypen führen kann. Diese Unbestimmtheit unterstreicht somit den grundsätzlich experimentellen Charakter der vorgestellten Lösung als Folge der besonderen Aufgabenstellung (vgl. Abschnitt 1.2).

Ein bedeutender Vorteil des Ansatzes liegt in der expliziten Modellierung einer Ontologie zur Beschreibung des Anwendungsbereichs (siehe Abschnitt 3.3 und 5.1). Sie erlaubt eine eindeutige Definition des zur Verfügung stehenden Vokabulars, die Ableitung weiterführender Zusammenhänge sowie eine klare Trennung zwischen anwendungsbezogenen und technischen Aspekten bei Beschreibung von Funktionalität. Das erarbeitete, objekt-orientierte Metamodell ist aus gängigen Ansätzen der Systementwicklung vertraut und kann daher mit geringem Einarbeitungsaufwand eingesetzt werden. Darüber hinaus wird Erstellung und Pflege der Ontologie durch bekannte grafische Beschreibungstechniken und entsprechende Werkzeuge umfassend unterstützt, wie in Abschnitt 5.1.3 erläutert wird.

Die zunächst vorgesehene technische Realisierung eines so beschriebenen komponentenbezogenen Anwendungsfalls als Sequenz von Operationsaufrufen ist für durchschnittlich ausgebildete Entwickler leicht beherrschbar und ohne besondere Mühe in die zugehörige Spezifikation einer bereitgestellten Komponente zu integrieren. Darüber hinaus können für diese Aufgabe wiederum geeignete grafische Beschreibungstechniken und unterstützende Werkzeuge eingesetzt werden (siehe Abschnitt 5.2.3). Die eigentliche *Bedeutung* einer so angegebenen Funktionalität ergibt sich jedoch erst zur Laufzeit des Prototypen durch sein beobachtbares Verhalten.

Deshalb ist es grundsätzlich wünschenswert, alternative Modelle zur Beschreibung von Funktionalität zumindest als Ergänzung der vorgeschlagenen Lösung zu berücksichtigen. Auf diese Weise läßt sich beispielsweise eine formale Spezifikation geeigneter Software-Architekturen zukünftig zur ver-

besserten Konstruktion funktionaler Prototypen nutzen (vgl. Abschnitt 5.4). Wie bereits in Abschnitt 6.1.1 diskutiert, können entsprechende Erweiterungen mit vertretbarem Aufwand in das grundlegende Framework integriert werden.

Somit ist die Komplexität des vorgestellten Ansatzes zusammenfassend als durchaus angemessen einzuschätzen. Die anwendungsbezogene Beschreibung von Funktionalität ist intuitiv verständlich, basiert aber dennoch auf einem eindeutig definierten Modell des Anwendungsbereichs. Das zugrundeliegende Metamodell der Ontologie ist übersichtlich gestaltet und bei entsprechender Werkzeugunterstützung einfach einzusetzen. Die angebotenen Mittel zur Beschreibung von Verhalten auf technischer Ebene sind zwar ebenfalls grundsätzlich leicht zu handhaben, sollten aber im Hinblick auf zukünftige Anforderungen geeignet erweitert werden. Hierbei entspricht der insgesamt gewählte Grad der Formalisierung im wesentlichen dem Charakter der betrachteten Problemstellung.

6.1.3 Komponentenorientierung

Die im Rahmen des Frameworks vorgenommene Trennung zwischen logischer und technischer Ebene des Ansatzes erleichtert den Einsatz von Komponenten unabhängiger Hersteller erheblich. Somit kann die Auswahl potentiell geeigneter Komponenten zur Erfüllung geforderter Funktionalität zunächst ausschließlich nach anwendungsbezogenen Merkmalen erfolgen. Hierfür werden die jeweils beigefügten CUC-Beschreibungen ausgewertet und mit den Anwendungsfällen der Funktionalen Spezifikation an Hand der sich ergebenden logischen Kompatibilität abgeglichen (siehe Abschnitt 3.6.2).

Die später durchgeführte Verknüpfung der so ausgewählten Komponenten wird von den technischen Anteilen einer CUC-Beschreibung bestimmt. Das vorgeschlagene Modell einer Interaktion zwischen Komponenten erlaubt hierbei eine dynamisch ermittelte Zuordnung von Instanzen der beteiligten Komponenten zu Rollen oder Parametern der angegebenen Beschreibung (vgl. Abschnitt 3.6.4). Falls die zugehörigen Instanzen bzw. deren Schnittstellen nicht dem vorausgesetzten Typ entsprechen, so werden in einem zweiten Schritt entsprechende Adapter für Repräsentationen von Konzepten oder Teilnehmer von Interaktionen eingebunden (siehe Abschnitt 3.6.3 und 5.3.2).

Dieses Vorgehen ermöglicht eine durchgängige Black-Box Wiederverwendung der verfügbaren Komponenten, ohne die Details ihrer Implementierung zu kennen oder sogar zu verändern. Mit Ausnahme der Ontologie als zentrales Modell des Anwendungsbereichs werden hierbei zunächst keine gemeinsamen Annahmen über Struktur und Verhalten des Systems vorausgesetzt. Dies führt einerseits zu einer durchaus erwünschten, losen Kopplung unabhängig

entwickelter Komponenten, erfordert andererseits aber voraussichtlich eine relativ hohe Anzahl unterschiedlicher Adaptern.

Während Adapter für verschiedene Repräsentationen des gleichen Konzepts in bestimmten Fällen automatisch generiert werden können, so müssen zumindest Interaktionsadapter in der Regel doch manuell durch den Benutzer erstellt werden. Der hierfür insgesamt erforderliche Aufwand ist nicht zu unterschätzen, auch wenn einmal bereitgestellte Adapter für jede weitere gleichartige Vermittlung zur Verfügung stehen. Daher erscheint eine in Abschnitt 5.4 erläuterte Erweiterung des Frameworks um gemeinsame, als bekannt vorausgesetzte Strukturen des Systems und zugehörige Interaktionen seiner Komponenten besonders vielversprechend. Sie erleichtert die Verknüpfung beteiligter Komponenten wesentlich, da somit ein insgesamt höherer Grad an Kompatibilität auf technischer Ebene angenommen werden kann. Allerdings ist für eine erfolgreiche Umsetzung dieses Ansatzes eine fortschreitende Standardisierung der Software-Architektur in dem jeweils betrachteten Anwendungsbereich zwingend erforderlich.

Die erarbeitete Lösung ermöglicht in diesem Zusammenhang eine evolutionäre Weiterentwicklung, bei der neben einer bisher eingeführten, lokal gültigen Beschreibung auch zusätzlich übergreifend spezifizierte Interaktionen zur Realisierung eines Anwendungsfalls angegeben werden. Diese können alternativ zur Auswahl und Verknüpfung weiterer kompatibler Komponenten herangezogen werden. Auf diese Weise läßt sich die Qualität der generierten Prototypen erheblich verbessern, falls weiterführende Informationen über geeignete Software-Architekturen sowie entsprechend angepaßte Komponenten verfügbar sind.

Die hiermit verbundene Flexibilität verdeutlicht die Leistungsfähigkeit der gewählten Konzeption. Für weitgehend unabhängig entwickelte Softwarekomponenten können unterschiedliche, vorwiegend einfach gehaltene Interaktionen genutzt werden, während sich eng zusammengehörige Komponenten mittels geeigneter Beschreibungen auch zu komplexen Strukturen verknüpfen lassen. In beiden Fällen bestimmt der gemeinsame Bezug zum betrachteten Anwendungsbereich maßgeblich über die Auswahl geeigneter Kandidaten. Somit werden die in Abschnitt 3.1.3 gestellten Anforderungen an einen komponentenorientierten Ansatz nahezu vollständig erfüllt.

6.1.4 *Effektivität*

Aufgrund der Unbestimmtheit initialer funktionaler Anforderungen an das zu entwickelnde System sollte ein effektiver Ansatz für exploratives Rapid Prototyping eine möglichst weitgehend automatisierte Erstellung unterschiedlicher Prototyp-Varianten unterstützen, wie in Abschnitt 3.1.4 angeführt wird. Die

vorgestellte Lösung erreicht die gewünschte Variantenbildung durch einen toleranten Abgleich zwischen entsprechend modellierten, komponentenbezogenen Anwendungsfällen (vgl. Abschnitt 3.6.2). Hierfür werden ausschließlich anwendungsbezogene Merkmale entsprechend den Zusammenhängen der Ontologie herangezogen. Im Verlauf der späteren Varianten-Generierung werden die jeweils ausgewählten Komponenten auf technischer Ebene verknüpft, wobei tatsächlich ungeeignete oder fehlerhafte Kombinationen durch eine flexibel anpaßbare Heuristik schrittweise eliminiert werden, wie in Abschnitt 3.6.4 und 3.6.5 ausführlich erläutert wird.

Dieses so zusammengefaßte Vorgehen erlaubt eine durchgehende Automatisierung des gesamten Verfahrens, sofern sich die erwünschte Funktionalität des Systems auf elementare, komponentenbezogene Anwendungsfälle abbilden läßt, diese über möglichst einfach gestaltete Interaktionen realisiert sind, sowie die jeweils eingesetzten Komponenten weitgehend kompatible Schnittstellen aufweisen. Während die zuerst genannte Annahme eine unabdingbare Voraussetzung der vorgestellten Lösung darstellt (vgl. Abschnitt 6.1.7), kann komplizierten Verhältnissen auf technischer Ebene durch unterschiedliche Strategien begegnet werden.

So bietet sich zunächst eine stärkere Einbeziehung des Benutzers an, um auftretende Schwierigkeiten bei Zuordnung benötigter Parametern und Rollen einer komplexen Interaktion zu lösen. Die hierfür erforderliche Erweiterung läßt sich auf unmittelbare Weise in das vorliegende Framework integrieren, wie in Abschnitt 5.3.3 erläutert wird. Darüber hinaus können weiterführende Informationen über geeignete Software-Architekturen die automatische Verknüpfung unterschiedlicher Komponenten wesentlich erleichtern. Schließlich erscheint langfristig eine enge Verzahnung von interaktiver und automatisierter Konstruktion funktionaler Prototypen besonders vielversprechend (siehe Abschnitt 5.4).

Die grundsätzlich weiterhin erforderliche Vermittlung zwischen technisch inkompatiblen Schnittstellen eigentlich logisch zusammengehöriger Komponenten erfolgt im Rahmen der vorliegenden Arbeit über entsprechend ausgezeichnete Adapter. Sie beinhalten die gesamte hierfür benötigte Funktionalität und können somit als eigenständiger Bestandteil eines erstellten Prototypen auch für weitere Abläufe des Verfahrens wiederverwendet werden. Hierbei ermöglicht der gemeinsame Bezug auf den jeweiligen Anwendungsbereich eine teilweise automatisierte Generierung von Adaptern für unterschiedliche Repräsentationen des gleichen Konzepts der Ontologie. Während die gegenwärtige Implementierung des Frameworks auf eine verhältnismäßig einfache Struktur der beteiligten Repräsentationen beschränkt ist, läßt sich zukünftig durch abstrakte Algorithmen auch eine vergleichbare Behandlung komplexer Zusammenhänge erreichen (siehe Abschnitt 3.6.3 und 5.3.2).

Somit werden insgesamt die in Abschnitt 6.1.4 zusammengefaßten Anforderungen an die Effektivität des Ansatzes überwiegend gut erfüllt. Allerdings kann das in dieser Arbeit beschriebene Verfahren eine vollständig automatisierte, fehlerfreie und tatsächlich zielführende Konstruktion funktionaler Prototypen im allgemeinen Fall nicht gewährleisten. Diese Tatsache ergibt sich letztlich aus der übergeordneten Zielsetzung, unabhängig entwickelte Komponenten weitgehend frei und ohne aufwendige formale Modelle ihres Verhaltens zu kombinieren (vgl. Abschnitt 3.1.2 und 3.1.3).

6.1.5 Technische Umsetzung

Die in Abschnitt 3.1.5 geforderte, möglichst weitgehende Übertragbarkeit der erarbeiteten Lösung auf verschiedene technische Plattformen wird durch die gewählte Konzeption wesentlich erleichtert. Die vorgegebene, klar definierte Unterscheidung zwischen logischer und technischer Ebene des Frameworks ermöglicht zunächst die Beibehaltung der aufgeführten Modelle und Verfahren für Spezifikation und Abgleich anwendungsbezogener Funktionalität. Lediglich die Umsetzung dieser Funktionalität im Rahmen angegebener Interaktionen zwischen beteiligten Komponenten ist an die jeweils unterschiedlichen Vorgaben und Verhältnisse anzupassen. Hierbei unterstützt die in dieser Arbeit vorausgesetzte, einfach gehaltene Auffassung eines komponentenbasierten Systems eine unmittelbare Abbildung auf gängige technische Infrastrukturen.

So beinhalten sowohl COM, Java als auch CORBA das explizite Konzept einer Schnittstelle, über deren Operationen die angebotene Funktionalität einer Softwarekomponente in Anspruch genommen wird (siehe Abschnitt 2.2). Das Konzept einer Operation wiederum schließt in allen Fällen die Übergabe von Parametern sowie den Erhalt eines definierten Ergebnisses nach deren Ausführung ein. Im übrigen läßt sich bei objekt-orientierten Sprachen ohne zwingend erforderliche Angabe einer solchen Schnittstelle, wie etwa Java oder C++ [Str91], auch die Gesamtheit aller öffentlich deklarierten Methoden einer Klasse als deren eigentliche Schnittstelle verstehen. Daher können alle Elemente des in Abbildung 3.8 und 3.9 dargestellten Modells eines komponentenbezogenen Anwendungsfalls auf technischer Ebene unmittelbar auf entsprechende Strukturen der genannten Plattformen abgebildet werden.

Obwohl die vorgestellte Referenz-Implementierung auf Basis der Java-Plattform realisiert wurde, setzen die in Abschnitt 3.6 zusammengefaßten Verfahren keine weitergehenden, spezifischen Annahmen über die eingesetzte Infrastruktur voraus. Lediglich die Ermittlung ggf. erforderlicher Adapter wird durch zur Laufzeit verfügbare Informationen über Kompatibilität der beteiligten Typen wesentlich vereinfacht. Neben Java steht allerdings auch

im Rahmen anderer technischer Plattformen ein vergleichbar leistungsfähiges Typsystem oder ähnliche Mechanismen zur Verfügung. Beispielsweise erweitert C# [Mic01b] die objekt-orientierte Programmiersprache C++ um entsprechende Typinformationen zur Laufzeit, während gängige CORBA-Implementierungen derartige Informationen über ein besonderes *Interface Repository* bereitstellen [Bor01b, ION01, Bro01].

Somit kann die vorgestellte Lösung ohne grundlegende Schwierigkeiten für unterschiedliche Infrastrukturen umgesetzt werden, sofern eine Implementierung der Varianten- und Adapter-Generierung die jeweils spezifischen technischen Eigenheiten, wie Ausnahmebehandlung, Repräsentation von Mengen oder Erzeugung von Instanzen für Komponenten, geeignet berücksichtigt. Weiterhin ist offensichtlich die Ausführung und interaktive Beurteilung generierter Prototyp-Varianten ebenfalls an die vorliegenden Gegebenheiten anzupassen. Hierbei erlaubt die technisch neutrale Beschreibung eines CUC prinzipiell auch den gemischten Einsatz von Komponenten verschiedener Plattformen, falls Initialisierung und Kommunikation über entsprechende Teile der Infrastruktur vermittelt wird (siehe Abschnitt 5.3.3). Eine solche Integration ist zumindest mittelfristig von strategischer Bedeutung für den Erfolg komponentenbasierter Softwareentwicklung und wird daher auch von den Herstellern vorangetrieben. Beispielsweise beinhaltet die als .NET bezeichnete, gegenwärtig aktuelle Weiterentwicklung von COM eine gleichartige Behandlung technisch unterschiedlich implementierter Komponenten [Mic01e].

Im Gegensatz zur weithin übertragbaren, synchronen Kommunikation über Operationsaufrufe repräsentiert die im Rahmen des Frameworks definierte, asynchrone Kommunikation über Ereignisse eine sehr spezifische, überwiegend technisch motivierte Lösung. Zwar läßt sich das in Abbildung 3.11 dargestellte Modell verhältnismäßig einfach auf alle bedeutsamen Plattformen abbilden, jedoch bleibt fraglich, ob diese besondere Form der Kommunikation zwischen Komponenten grundsätzlich dem angegebenen Muster folgt. So wird etwa gerade bei verteilten Systemen in der Praxis häufig eine vergleichsweise lose Kopplung über besondere Mechanismen oder Bestandteile der Infrastruktur realisiert [OMG01a, Mic01c]. Hierbei unterscheidet sich das jeweils vorausgesetzte Protokoll für Ausgabe, Entgegennahme und Verarbeitung von Ereignissen deutlich von den Vorgaben dieser Arbeit. Daher ist zukünftig ein erweiterter, verallgemeinerter Ansatz zur Modellierung und Umsetzung asynchroner Kommunikation wünschenswert. Diese Weiterentwicklung läßt sich überaus vorteilhaft mit dem in Abschnitt 5.4 vorgeschlagenen Übergang zur Simulation funktionaler Prototypen verbinden.

Ungeachtet dieser Kritik erfüllt der vorliegende Ansatz die in Abschnitt 6.1.5 gestellten Anforderungen an eine praxisgerechte technische

Umsetzung überwiegend gut. Das auf technischer Ebene gewählte Modell entspricht den wesentlichen Gemeinsamkeiten aller gegenwärtig bedeutsamen komponentenbasierten Infrastrukturen. Die entsprechenden Anteile einer CUC-Beschreibung können im Rahmen jeder gängigen technischen Plattform unmittelbar umgesetzt werden. Zudem werden die erstellten Beschreibungen dem ausführbaren Format einer Komponente lediglich beigefügt, so daß keine aufwendige Änderung ihrer Implementierung erforderlich ist. Dieser Umstand erleichtert eine erfolgreiche praktische Anwendung der erarbeiteten Ergebnisse erheblich.

6.1.6 Skalierbarkeit

Wie in Abschnitt 3.1.6 angeführt, repräsentiert die Skalierbarkeit hinsichtlich realistischer Problemgrößen eine maßgebliche Anforderung an einen tragfähigen Ansatz für komponentenbasiertes Rapid Prototyping. Die Effizienz des in dieser Arbeit vorgestellten, übergeordneten Verfahrens wird im wesentlichen durch Auswahl und Verknüpfung potentiell geeigneter Komponenten sowie Bewertung der so erhaltenen Prototyp-Varianten bestimmt. Hierbei wird die zuerst genannte Aufgabe auch durch Umfang und Komplexität der zugrundeliegenden Ontologie beeinflußt, während die beiden letzten Schritte ausschließlich von der Anzahl jeweils ausgewählter Komponenten bzw. ihrer möglichen Interaktionen abhängig sind. Daher ist die Auswirkung dieser Verhältnisse auf Laufzeit und Speicherplatzbedarf der eingesetzten Algorithmen besonders kritisch zu untersuchen. Zuletzt ist bei einer umfassenden Beurteilung des erreichten Erfolgs auch der insgesamt erforderliche manuelle Aufwand zu berücksichtigen.

Der in Abschnitt 3.6.2 erläuterte Abgleich logisch kompatibler Anwendungsfälle zwischen gegebenen CUC-Beschreibungen und Funktionaler Spezifikation kann sehr effizient durchgeführt werden. Die hierfür maßgebliche Bestimmung der semantischen Kompatibilität manipulierter Konzepte beruht auf festgelegten Äquivalenz-, Generalisierungs- und Interpretationsbeziehungen der Ontologie (siehe Abschnitt 3.3). Diese Zusammenhänge sind in nahezu konstanter Zeit zu ermitteln, insbesondere falls eine im wesentlichen „flache“ Hierarchie der über Generalisierung in Beziehung gesetzten Konzepte angenommen wird. Zudem können nicht unmittelbar ersichtliche Verhältnisse, etwa über transitive Relationen verbundene Konzepte, als vorberechnetes Ergebnis bereitgestellt werden. Somit ist letztlich in einer entsprechenden Analyse nur die Anzahl an insgesamt verfügbaren Komponenten bzw. der von ihnen angebotenen, komponentenbezogenen Anwendungsfälle zu betrachten.

Obwohl bei naiver Umsetzung jeder Anwendungsfall der Spezifikation mit

den CUC-Beschreibungen aller Komponenten verglichen werden muß, sind in der Praxis doch einige Optimierungen des eingesetzten Verfahrens möglich. So bietet es sich zunächst an, die jeweils berechnete semantische Kompatibilität zwischen gegebenen Konzepten der Ontologie im Rahmen einer tabellarisch organisierten Datenstruktur zu speichern. Nunmehr kann vorab für jeden CUC die jeweils gültige logische Kompatibilität leicht ermittelt und in einem entsprechenden Index abgelegt werden. Dieser wird zuletzt nach dem Grad der logischen Kompatibilität sortiert. Mit Hilfe der so erzeugten Indizes ist im weiteren Verlauf eine Auswahl geeigneter Komponenten für einen einzelnen Anwendungsfall der Spezifikation auch bei verändertem Schwellwert ϵ_k in nahezu konstanter Zeit möglich (vgl. Definition 3.6). Demgegenüber ist der erhöhte Bedarf an Speicherplatz zu vernachlässigen, auch wenn die vorberechneten Werte und Indizes offensichtlich bei Änderungen der Ontologie oder der Menge an verfügbaren Komponenten aktualisiert werden müssen.

Die eigentlich grundlegende Problematik liegt also in der Verknüpfung und Bewertung ausgewählter Komponenten als Bestandteil generierter Prototyp-Varianten. Schließlich steigt die Zahl unterschiedlich zusammengesetzter Varianten exponentiell an, wie aus Gleichung 3.5 ersichtlich ist. Aus diesem Grund führt der vorliegende Ansatz eine geeignete Heuristik ein, mit deren Hilfe nur ein jeweils beherrschbarer Ausschnitt aller möglichen Varianten betrachtet werden muß (siehe Abschnitt 3.6.5). Die für eine gegebene Variante erforderlichen Schritte des vorgeschlagenen Genetischen Algorithmus können entweder in konstanter Zeit (*Mutation*) oder in Abhängigkeit der überschaubaren Anzahl an Anwendungsfällen der Funktionalen Spezifikation (*Reproduktion*, *Rekombination*) durchgeführt werden. Somit wird der insgesamt erforderliche Aufwand zur Verknüpfung und Bewertung lediglich durch die festgelegte Anzahl an unterschiedlichen Prototyp-Varianten bestimmt. Diese entscheidende Größe kann jedoch weitgehend flexibel an die verfügbaren Ressourcen sowie die Ergebnisse der vorangegangenen Komponenten-Auswahl angepaßt werden, wie in Abschnitt 3.6.5 ausführlich erläutert wird. Darüber hinaus ist die vorgestellte Heuristik verhältnismäßig einfach zu parallelisieren, so daß grundsätzlich auch besonders umfangreiche Probleme angemessen behandelt werden können.

Dennoch beinhaltet das übergeordnete Vorgehen einige manuelle Schritte, etwa bei Erstellung von Adaptern oder anwendungsbezogener Bewertung generierter Ergebnisse. Diese interaktiven Abläufe beschränken die Anzahl betrachteter Prototyp-Varianten deutlich. Daher sieht die erarbeitete Lösung einen durchaus variablen Grad an Interaktion mit dem Benutzer vor, wobei manuell erstellte Bestandteile soweit wie möglich wiederverwendet werden. Zudem besteht zukünftig die Möglichkeit, über entsprechende Erweiterungen die automatische Generierung von Adaptern, Verknüpfung von Komponenten

und anwendungsbezogene Bewertung der erhaltenen Ergebnisse entscheidend zu verbessern (siehe Abschnitt 5.3.2, 5.3.4 und 5.4).

Somit kann die Skalierbarkeit des vorliegenden Ansatzes zusammenfassend als ausgesprochen hoch eingeschätzt werden. Der erforderliche Aufwand zur Anwendung des vorgeschlagenen Verfahrens wächst im wesentlichen linear mit der Anzahl an vorhandenen Komponenten bzw. tatsächlich betrachteten Prototyp-Varianten. Deshalb können auch realistische Problemgrößen ohne besondere Maßnahmen oder grundsätzliche Einschränkungen behandelt werden. Hierbei läßt sich die eingeführte Heuristik sehr flexibel an die jeweiligen Gegebenheiten und verfügbaren Ressourcen anpassen.

6.1.7 Anwendbarkeit

Die in Abschnitt 3.1.7 geforderte, möglichst weitreichende Übertragbarkeit der erzielten Ergebnisse auf unterschiedliche Anwendungsbereiche repräsentiert sicherlich ein besonders bedeutsames Kriterium zur Beurteilung des erreichten Erfolgs. Aufgrund der gewählten Modelle und Verfahren ist eine umfassende technische Umsetzung des Frameworks gewährleistet, wie zuvor in Abschnitt 6.1.5 ausführlich erläutert wird. Jedoch sind die jeweils vorausgesetzten Annahmen und Merkmale auf anwendungsbezogener Ebene differenziert zu betrachten. Hierfür liefert insbesondere der in dieser Arbeit exemplarisch untersuchte Anwendungsbereich wertvolle Erkenntnisse und Hinweise. Somit lassen sich drei wesentliche Voraussetzungen zur Anwendung des vorgeschlagenen Ansatzes identifizieren, die im folgenden gesondert diskutiert werden.

Gemeinsames Verständnis der anwendungsbezogenen Zusammenhänge

Die explizite Modellierung des Anwendungsbereichs als Ontologie erfordert ein gemeinsames Verständnis der grundlegenden Konzepte und ihrer Beziehungen. Sie bilden den maßgeblichen Bezugspunkt für Beschreibung oder Abgleich von erwünschter und angebotener Funktionalität, gerade falls unabhängig entwickelte Komponenten unterschiedlicher Hersteller eingesetzt werden. Hierbei ist neben einer unmittelbaren Abbildung realer Verhältnisse auch eine gleichbedeutende Konzeptualisierung abgeleiteter Informationen zu berücksichtigen. So ist einerseits die beabsichtigte Bedeutung der Begriffe *Protein* oder *DNA* im Bereich der Biochemie eindeutig festgelegt, andererseits besteht auch in der Verwendung und Definition des Begriffs *Alignment* im Kontext der Sequenzanalyse weitgehend Einigkeit.

Demgegenüber können hoch dynamische Anwendungsbereiche mit fortwährenden, sprunghaften Innovationen, divergent aufgefaßten Konzepten

und willkürlich festgelegten Zusammenhängen nicht durch den vorgestellten Ansatz unterstützt werden. Dies betrifft beispielsweise den Bereich der Online-Zahlungssysteme für den elektronischen Handel, in dem eine Vielzahl konkurrierender Ansätze mit verschiedenen, weitgehend inkompatiblen Verfahren und Geräten den Markt bestimmt [Hen01]. Allerdings ist in derartig heterogenen Anwendungsbereichen ohnehin nur eine verhältnismäßig geringe Anzahl qualitativ hochwertiger und übergreifend wiederverwendbarer Softwarekomponenten zu erwarten.

Abbildung von Funktionalität auf Manipulation von Konzepten

Die Möglichkeit zur Beschreibung grundlegender Funktionalität als Manipulation von Konzepten des Anwendungsbereichs repräsentiert eine zweite bedeutende Voraussetzung des vorgestellten Ansatzes. Hierbei werden ggf. zusätzliche Konzepte als Parameter herangezogen und in vielen Fällen ein beobachtbares Ergebnis wiederum als klar definiertes Konzept zurückgeliefert. Eine solche Auffassung von Funktionalität ist sicherlich auf viele Anwendungsbereiche übertragbar, wie Definition und typische Benennung eines herkömmlichen Anwendungsfalls als Mittel der Anforderungsanalyse belegen (vgl. Abschnitt 3.4). Darüber hinaus kann die Expressivität des vorgeschlagenen Modells durch weitergehende Anlehnung an die natürliche Sprache verbessert werden, wie in Abschnitt 5.2.1 erläutert wird.

Dennoch sind Anwendungsbereiche zu berücksichtigen, in denen Funktionalität zweckmäßig über charakteristische, meist zusammengesetzte oder mit zusätzlichen Attributen versehene Substantive beschrieben wird, beispielsweise durch Begriffe wie *nicht-linearer Videoschnitt* oder *Personal Information Management*. Hierbei ist zu untersuchen, inwieweit sich die so festgelegten Verhältnisse gleichbedeutend auch durch eine Kombination aus aktiven und passiven Elementen ausdrücken lassen. In vielen Fällen dienen die jeweils verwendeten Begriffe ohnehin lediglich der prägnanten Bezeichnung zusammengefaßter Funktionalität. Diese läßt sich jedoch im Rahmen des vorgeschlagenen Ansatzes wesentlich genauer über hierarchisch strukturierte Anwendungsfälle beschreiben (vgl. Abbildung 3.7).

Lokale und einfache Interaktionen zur Realisierung von Funktionalität

Eine letztes wesentliches Merkmal der vorgeschlagenen Lösung betrifft die Realisierung von Anwendungsfällen über möglichst einfach gestaltete Interaktionen zwischen den beteiligten Komponenten. Es wird vorausgesetzt, daß anwendungsbezogene Funktionalität weitgehend lokal einzelnen Komponenten zugeordnet und nach Ausführung der zugehörigen Interaktion als abge-

schlossen betrachtet werden kann. Diese grundlegenden Annahmen entsprechen der Auffassung einer Softwarekomponente als Einheit der Funktion in der Systementwicklung (vgl. Abschnitt 2.2) sowie dem Wunsch nach einer durchgehenden Automatisierung des gesamten Verfahrens. Dennoch ist in der Praxis zu erwarten, daß bestimmte Funktionalität nur über komplexe Interaktionen mit zahlreichen gleichberechtigten Komponenten und unterscheidbaren Zwischenzuständen realisiert werden kann. So erfordert etwa die Bestellung eines Flugtickets in der Regel ein durchaus langwieriges Zusammenspiel der verschiedenen Komponenten eines Reisebuchungssystems, in dessen Verlauf die vom Kunden genutzte Funktionalität schrittweise erfüllt wird.

Das erarbeitete Framework kann derartig komplizierte Verhältnisse nicht angemessen unterstützen, insbesondere weil der Ansatz unterscheidbare Zustände für Konzepte des Anwendungsbereichs nicht berücksichtigt. Immerhin läßt sich zukünftig die Umsetzung von Interaktionen auf technischer Ebene durch eingeführte Informationen über Software-Architektur sowie zusätzliche interaktive Elemente erheblich verbessern (siehe Abschnitt 5.2.2 und 5.4). Zudem erlaubt eine Weiterentwicklung der Funktionalen Spezifikation die Angabe von abstrakten Interaktionen auf logischer Ebene der Aufgabenstellung, wie in Abschnitt 5.4 erläutert wird. Dieses Modell erscheint langfristig besonders geeignet, um anwendungsbezogene Abläufe oder Lösungsstrategien für komplexe, zusammengesetzte Aufgaben weitgehend unabhängig von technischen Details zu beschreiben.

Somit kann die Anwendbarkeit des vorgeschlagenen Ansatzes abschließend als befriedigend eingeschätzt werden. Die hierfür wesentlichen Voraussetzungen, also gemeinsames Verständnis grundlegender Konzepte und ihrer Zusammenhänge, Beschreibung von Funktionalität als Manipulation von Konzepten sowie weitgehend lokale und einfach gehaltene Realisierung dieser Funktionalität, werden von einer Vielzahl an unterschiedlichen Anwendungsbereichen zumindest teilweise erfüllt. So lassen sich wissenschaftliche Analyse und Bearbeitung umfangreicher Rohdaten besonders weitgehend unterstützen, wie der Vergleich mit dem exemplarisch untersuchten Anwendungsbereich nahelegt. Dies betrifft beispielsweise Informationssysteme im Bereich der Medizin, Geographie oder Physik. Dennoch ist es das Ziel, Übertragbarkeit und Qualität der erhaltenen Ergebnisse auf Basis zukünftiger praktischer Erfahrungen durch evolutionäre Weiterentwicklung des vorgestellten Frameworks zu verbessern.

6.2 Vergleich mit bestehenden Ansätzen

Wie bereits erwähnt, erlaubt der Vergleich mit bestehenden Ansätzen und Konzepten eine genauere Einordnung der vorgeschlagenen Lösung sowie eine weiterführende Beurteilung des erreichten Erfolgs und eigenständigen Beitrags dieser Arbeit. Zunächst werden umfassende Ansätze vorgestellt, welche die übergeordnete Problemstellung, also vereinfachte Konstruktion funktionaler Systeme aus vorhandenen Komponenten, in ihrer Gesamtheit berücksichtigen. Hierbei werden aufgrund der Vielzahl ähnlicher Lösungen nur jeweils besonders prägnante Vertreter unterschiedlicher Vorgehensweisen exemplarisch betrachtet. Anschließend wird der Bezug einzelner Bestandteile des Frameworks zu bekannten Modellen und Verfahren des Software Engineering aufgezeigt. Dies verdeutlicht nicht zuletzt die praktische Relevanz sowie das zukünftige Potential der erzielten Ergebnisse, wie abschließend in Abschnitt 6.3 angeführt wird.

6.2.1 Übergreifend vergleichbare Ansätze

Die im folgenden zum Vergleich herangezogenen Ansätze repräsentieren alternative Lösungen der in Abschnitt 1.2 zusammengefaßten Aufgabenstellung. Sie ermöglichen ebenfalls die Erstellung funktionaler Systeme aus vorhandenen Bestandteilen und werden daher unmittelbar den Ergebnissen dieser Arbeit gegenübergestellt. Hierbei unterscheiden sich die aufgeführten Ansätze in ihrer Konzeption, Verwertung anwendungsbezogener Informationen sowie Grad der erreichten Automatisierung durchaus erheblich. Daher erlaubt ihre Betrachtung eine Charakterisierung des vorgestellten Frameworks und zeigt interessante Möglichkeiten für dessen langfristige Weiterentwicklung auf.

Skriptsprachen und Rapid Application Development

Die flexible, einfache und rasche Verknüpfung existierender Komponenten zu übergeordneten Systemen wird gegenwärtig in vielen Fällen über ein geeignetes *Skript* durchgeführt. Ein solches, meist einfach gehaltenes Programm wird in einer spezifischen *Skriptsprache*, wie etwa JavaScript [WF97], Visual Basic [Mic98] oder Perl [WOC00], erstellt und üblicherweise durch eine entsprechende Laufzeit-Umgebung interpretiert bzw. ausgeführt. Die jeweils verwendete Skriptsprache ist hinsichtlich ihrer Expressivität in der Regel einer herkömmlichen Programmiersprache vergleichbar und erlaubt somit beispielsweise die Erstellung von Instanzen einer gegebenen Komponente, Aufruf von Operationen ihrer Schnittstelle oder Zuweisung von Zwischenergebnissen zu deklarierten Variablen.

Darüber hinaus ist für ausgewählte Skriptsprachen eine leistungsfähige Entwicklungsumgebung verfügbar, die eine weitgehend intuitive, visuelle und interaktive Komposition funktionaler Komponenten unterstützt. Eine solche Kombination aus interpretierter Programmiersprache, Laufzeit-Umgebung und visuellem Entwicklungswerkzeug wird häufig unter dem Begriff *Rapid Application Development* (RAD) zusammengefaßt. Hierbei eignet sich RAD besonders gut für eine komponentenbasierte Softwareentwicklung, da üblicherweise ein wesentlicher Anteil der Gesamtfunktionalität eines Systems auf vorhandene, leicht zu integrierende Komponenten abgestützt wird. Zudem wird im allgemeinen eine gemeinsam genutzte Infrastruktur bereitgestellt, die Verknüpfung und Wiederverwendung unterschiedlicher Komponenten erleichtert. Umgekehrt führt die zunehmende Popularität von RAD in bestimmten Fällen zu einer steigenden Anzahl an verfügbaren, teilweise hochwertigen Komponenten für zahlreiche unterschiedliche Anwendungsbereiche, wie etwa am Beispiel von Visual Basic mit COM als zugrundeliegender technischer Plattform deutlich wird [Szy98]. Daher wurde dieser erfolgreiche Ansatz auch auf andere Sprachen und Plattformen übertragen [Bor01a, IBM01b].

Andererseits bietet RAD keine Unterstützung bei anwendungsbezogener Auswahl und Verknüpfung funktionaler Komponenten. Der Benutzer muß möglicherweise geeignete Komponenten selbst auswählen und deren Funktionalität über entsprechende Mittel des eingesetzten Werkzeugs oder eigens erstellte Skripten bzw. Teile des entwickelten Programms in Anspruch nehmen. Hierfür ist in der Regel ein ausführliches Studium der jeweils einer Komponente beigefügten Dokumentation erforderlich, so daß sich gerade bei Betrachtung unterschiedlich zusammengesetzter Varianten ein erheblicher manueller und zeitlicher Aufwand einstellt. Zudem unterliegt die geeignete Vermittlung zwischen technisch inkompatiblen Komponenten unabhängiger Hersteller grundsätzlich der alleinigen Verantwortung des Benutzers, wobei keine Vorgaben hinsichtlich Struktur und Lokalisierung der hierfür benötigten Funktionalität getroffen werden.

Demgegenüber erlaubt der vorliegende Ansatz eine weitgehend automatisierte Konstruktion zahlreicher Prototyp-Varianten an Hand vorgegebener funktionaler Anforderungen sowie einem eindeutig definierten Modell des Anwendungsbereichs. Darüber hinaus können ggf. benötigte Adapter zumindest teilweise ebenfalls automatisch generiert und eingebunden werden. Hierbei wird die grundlegende Flexibilität und Zuverlässigkeit eines vollständig manuellen Verfahrens zugunsten eines insgesamt deutlich verringerten Aufwands eingeschränkt. Immerhin läßt sich die Qualität der erhaltenen Ergebnisse durch gezielte Interaktion mit dem Benutzer weiter verbessern, wie in Abschnitt 5.3.3 erläutert wird. Daher erscheint es vielversprechend, das vorgestellte Framework mit Ansätzen des RAD zu verbinden, um die spezi-

fischen Vorteile beider Ansätze zu kombinieren. Insbesondere wird hierdurch bei konsequenter Umsetzung der in Abschnitt 5.4 beschriebene Übergang zur interaktiven und inkrementellen Konstruktion funktionaler Prototypen erreicht.

Generative Softwareentwicklung

Die teilweise automatisierte Transformation abstrakter Spezifikationen zu funktionalen Systemen wird in zahlreichen generativen Ansätzen der Softwareentwicklung behandelt [BG91, BO92, Gom94, Nin94]. Hierfür werden im Gegensatz zu skriptbasierten Ansätzen und RAD üblicherweise auch weitergehende Informationen über Zusammenhänge des jeweils betrachteten Anwendungsbereichs herangezogen. Diese Modelle sowie das zu entwickelnde System können mittels einer spezifischen Sprache (engl. *domain-specific language*, DSL) [Kie96, DK97] oder Software-Architektur (engl. *domain-specific software architecture*, DSSA) [SEI90] beschrieben werden. Anschließend generiert ein besonderes Werkzeug aus den angegebenen Spezifikationen ausführbaren Code der eingesetzten technischen Plattform, wobei wiederverwendbare Bausteine zur Erfüllung erwünschter Funktionalität verknüpft werden.

Die übergeordnete Zielsetzung ist bei generativen Ansätzen in der Regel die Konstruktion des endgültigen Systems unter Berücksichtigung aller funktionaler aber auch nicht-funktionaler Anforderungen. So beschreibt beispielsweise [BCRW00] die weitgehend automatisierte Erstellung effizienter Datenstrukturen für Mengen durch Kombination elementarer Komponenten in einer spezifischen, hierarchisch organisierten Software-Architektur. Der hierfür grundlegende, als *GenVoca* bezeichnete Ansatz [BO92] erlaubt eine formale Spezifikation der verwendeten DSSA sowie gültiger Konsistenzbedingungen für eine korrekte Komposition der funktionalen Bestandteile [BG96]. Allerdings werden besondere, sowohl technisch als auch logisch kompatible Schnittstellen der beteiligten Komponenten vorausgesetzt.

Demgegenüber beschränkt sich der in dieser Arbeit vorgestellte Ansatz bewußt auf die Konstruktion funktionaler Prototypen mit deutlich geringeren Anforderungen an Zuverlässigkeit und Effizienz der generierten Ergebnisse. Deshalb kann zunächst auf die Annahme einer durchgängigen technischen Kompatibilität aller verwendeten Komponenten verzichtet werden. Somit lassen sich auch unabhängig entwickelte Komponenten verschiedener Hersteller auf verhältnismäßig einfache Weise nutzen (vgl. Abschnitt 3.1.3). Dies führt letztlich zu einer größeren Vielfalt an unterschiedlich zusammengesetzten Prototypen.

Weiterhin erleichtert ein angemessener Grad der Formalisierung die erfolgreiche praktische Umsetzung des vorgeschlagenen Frameworks. Die

gewählten Modelle und Beschreibungstechniken entsprechen in weiten Zügen bekannten Ansätzen der objekt-orientierten oder komponentenbasierten Softwareentwicklung, so daß sich diese ohne besondere Qualifikation des Benutzers einsetzen lassen. Hingegen erfordert die zuverlässige Generierung effizienter Systeme entsprechend anspruchsvolle Techniken und mithin einen erheblich höheren Einarbeitungsaufwand.

Zuletzt wird durch die vorgenommene Trennung zwischen anwendungsbezogener und technischer Ebene der vorgestellten Lösung eine überwiegend lose Kopplung der unterschiedlichen Aufgabenbereiche erreicht. Daher können die jeweils erarbeiteten Modelle und Verfahren weitgehend unabhängig voneinander angewendet oder weiterentwickelt werden, wie insbesondere durch die in Kapitel 5 aufgeführten Vorschläge verdeutlicht wird. Im Gegensatz hierzu sind beide grundlegenden Aspekte der Problemstellung bei existierenden generativen Ansätzen üblicherweise eng miteinander verknüpft. So beinhaltet eine gegebene DSSA genaue Vorgaben für Verhalten, Struktur und Interaktion des gesamten Systems sowie seiner funktionalen Bestandteile. Jede weitergehende Änderung der funktionalen Anforderungen führt somit zu potentiell aufwendigen Anpassungen der spezifizierten Architektur. Gerade derartige Änderungen sind jedoch im Verlauf eines explorativen Prototyping besonders häufig zu erwarten. Daher erscheint eine durch den vorliegenden Ansatz erreichte Flexibilität bei anwendungsbezogener Auswahl und technischer Verknüpfung funktionaler Komponenten zur Lösung des übergeordneten Problems besser geeignet.

Dennoch ist zumindest mittelfristig eine Verbesserung der erstellten Prototypen hinsichtlich ihrer Zuverlässigkeit und Performanz wünschenswert. Zu diesem Zweck können ausgewählte Teile der bestehenden, erfolgreich angewendeten generativen Ansätze in das Framework integriert werden. Beispielsweise läßt sich die Spezifikation geeigneter Software-Architekturen für eine erheblich vereinfachte Zuordnung und Verknüpfung funktionaler Komponenten nutzen, wie in Abschnitt 5.4 angeführt wird. In diesem Zusammenhang können DSL- oder DSSA-basierte Modelle und Verfahren vorteilhaft eingesetzt werden. Sie repräsentieren demnach eine vielversprechende Ergänzung des vorgestellten Ansatzes.

Software Agenten

Zunächst erscheint es überraschend, das vorliegende Framework mit bekannten Arbeiten über *Software Agenten* zu vergleichen. Schließlich werden derartige Ansätze gegenwärtig meist in hochgradig verteilten, dynamisch veränderlichen Systemen eingesetzt, um ausgewählte Aufgaben möglichst autonom, also ohne weitgehende Interaktion mit dem Benutzer, in dessen Auftrag zu

bearbeiten [Nwa96, WC01]. Im Zuge der Verbreitung des Internets wurden beispielsweise agentenbasierte Systeme entwickelt, die eine gezielte Suche nach vom Anwender gewünschten Informationen unterstützen [Lie95] oder einen virtuellen Handelsplatz für Waren aller Art einrichten [CM96]. Gerade durch vielfältige Interaktionen zwischen den beteiligten Agenten sowie ihrer Umgebung ergibt sich hierbei eine besondere Strategie zur Lösung komplexer Probleme, welche sich bei erster Betrachtung nur schwer mit dem geplanten, methodischen Vorgehen des traditionellen Software Engineering vereinbaren läßt.

Andererseits schlägt auch die vorliegende Arbeit einen unkonventionellen Ansatz vor, der mit Hilfe weitgehend deklarativer Angaben des Benutzers versucht, aus vorgegebenen Komponenten möglichst gut geeignete, funktionale Prototypen zu konstruieren. Hierbei ist weder das erhaltene Ergebnis noch der genaue Ablauf seiner Ermittlung durch das vorgestellte Verfahren eindeutig festgelegt. Vielmehr bestimmen jeweils verfügbare Komponenten, mögliche Strategien zu ihrer Verknüpfung sowie Verlauf der im wesentlichen zufallsbasierten Optimierung über den letztlich erzielten Erfolg. Dieser dynamische, unbestimmte Charakter erlaubt einen durchaus interessanten und aufschlußreichen Vergleich mit agentenbasierten Ansätzen.

Hierfür ist zunächst eine genaue Auffassung des Konzepts *Software Agent* erforderlich. Ungeachtet der Vielzahl an tatsächlich unterschiedlichen Modellen, läßt sich ein Agent grundsätzlich als ein System verstehen, das fortwährend Informationen aus seiner Umgebung über sog. *Sensoren* aufnimmt, diese verarbeitet und eigenständige Aktionen durchführt, um ein übergeordnetes Ziel zu erreichen [Woo99]. Weitere typische Eigenschaften eines Agenten sind Mobilität, kooperatives Verhalten und Lernfähigkeit, auch wenn diese nicht bei allen Ansätzen gleichbedeutend berücksichtigt sind [Nwa96, WC01]. Schließlich beinhaltet ein Agent häufig eine symbolische Repräsentation des Wissens über seine Umgebung und die von ihm zu erfüllende Aufgabe. Mit ihrer Hilfe kann eine Entscheidung über die jeweils nächsten, zielführenden Aktionen getroffen werden.

In dieser Hinsicht kann auch das vorgestellte Framework bzw. dessen Implementierung als Software Agent aufgefaßt werden. Sie nimmt zu Beginn aus ihrer Umgebung Informationen über Zielsetzung und verfügbare Mittel als Funktionale Spezifikation und Menge vorgegebener CUC-Beschreibungen entgegen. Anschließend werden bestimmte Aktionen zur Verknüpfung von Komponenten durchgeführt, wobei die Ontologie als symbolische Repräsentation des Anwendungsbereich herangezogen wird. Zuletzt bestimmen weitere Informationen aus der Umgebung, also die Bewertung der generierten Prototyp-Varianten, über die nächsten Schritte eines fortwährenden, iterativ organisierten Verfahrens. Darüber hinaus kann durch entsprechende Er-

weiterungen um adaptive Anteile (siehe Kapitel 5) eine einfache Form von Lernfähigkeit erreicht werden.

Demgegenüber sind weiterführende Merkmale eines Agenten, wie Mobilität oder kooperatives Verhalten, im vorliegenden Ansatz nicht berücksichtigt. Ihre Integration eröffnet jedoch vielfältige Möglichkeiten für eine zukünftige, langfristig angelegte Weiterentwicklung des Frameworks. So lassen sich Auswahl und Verknüpfung geeigneter Komponenten durchaus auch auf verschiedene, autonom agierende Implementierungen verteilen, die ihre jeweils erzielten, partiellen Ergebnisse untereinander austauschen. Hierbei können etwa aufgrund lokaler, durch den jeweiligen Hersteller bereitgestellter Informationen bestimmte Kombinationen von Komponenten besonders zuverlässig verknüpft werden. Dieses Vorgehen führt zu einer höheren Qualität der erstellten Prototypen sowie einer insgesamt erheblich verbesserten Effizienz.

Umgekehrt können die in dieser Arbeit eingeführten Modelle und Verfahren zur Weiterentwicklung bestehender agentenbasierter Ansätze herangezogen werden. So erlaubt die in Abschnitt 3.3 vorgeschlagene Repräsentation des Anwendungsbereichs eine verhältnismäßig einfache und effizient durchzuführende Ableitung weiterführender Zusammenhänge. Diese besondere Eigenschaft ist auch bei Entwurf und Implementierung von Software Agenten überaus wünschenswert [Woo99]. Darüber hinaus läßt sich die vorgeschlagene Auffassung von grundlegender Funktionalität als Manipulation von Konzepten zur Beschreibung der angebotenen oder vorausgesetzten Fähigkeiten eines Agenten einsetzen. Somit ermöglichen abstrakte Interaktionen (vgl. Abschnitt 5.4) eine anwendungsbezogene Spezifikation zielführender Abläufe, die von Agenten im Zusammenspiel mit ihrer Umgebung oder anderen Agenten genutzt werden kann. Zuletzt lassen sich die in Abschnitt 3.6.3 und 5.3.2 aufgeführten Verfahren zur automatisierten Generierung von Adaptern zweckmäßig zur Vermittlung zwischen Agenten in heterogenen Umgebungen verwenden.

Die so zusammengefaßten Verbesserungen veranschaulichen das Potential einer weiterführenden Integration agentenbasierter Ansätze mit der vorgestellten Lösung. Jedoch ist der Charakter dieser Weiterentwicklung grundsätzlich verschieden von der in Abschnitt 6.2.1 diskutierten Berücksichtigung generativer Ansätze der Systementwicklung. Software Agenten führen zu einer ausgesprochen losen Kopplung mit dezentraler Kontrolle des übergeordneten Ablaufs und weitgehend unbestimmten Ergebnissen, während vorgegebene, spezifische Software-Architekturen eine unmittelbare Generierung eindeutig festgelegter Ergebnisse nahelegen. Dieser bemerkenswerte Gegensatz verdeutlicht nicht zuletzt die Flexibilität der erarbeiteten Lösung.

6.2.2 Partiell vergleichbare Ansätze

Nach den zuvor aufgeführten Ansätzen mit übergreifend vergleichbarer Zielsetzung, werden in diesem Abschnitt ausgewählte Bereiche des vorgestellten Frameworks isoliert betrachtet. Die hierbei auftretenden Gemeinsamkeiten mit existierenden, bewährten oder zumindest vielversprechenden Lösungen sind in der Regel keineswegs zufällig, sondern im Gegenteil bei Konzeption des Frameworks bewußt einbezogen, um dessen Plausibilität zu motivieren und eine erfolgreiche praktische Umsetzung zu gewährleisten. Darüber hinaus lassen sich somit zukünftige Erkenntnisse in den betreffenden Bereichen mit vertretbarem Aufwand in die vorliegende Arbeit integrieren. Dennoch erfordert der besondere Charakter der betrachteten Problemstellung eine geeignete Anpassung oder sogar Abgrenzung bestehender Konzepte und Lösungen, wie im folgenden erläutert wird.

Ontologie

Der Einsatz einer eindeutig definierten Ontologie zur erleichterten Verknüpfung unterschiedlicher Bestandteile eines übergeordneten Systems ist ein wohlbekanntes Mittel in wissensbasierten Ansätzen der Softwareentwicklung. So beschreiben beispielsweise [FW95] und [JWW+95] den Aufbau eines Systems zur Unterstützung von Raumfahrtmissionen durch Integration unabhängig entwickelter Teilsysteme über eine geeignete Ontologie des Anwendungsbereichs. Zahlreiche weitere Beispiele belegen die Notwendigkeit eines gemeinsamen Verständnisses grundlegender Konzepte und ihrer Beziehungen ohne Bezug zu technischen Details der Implementierung, auch wenn die jeweils gewählte Repräsentation sowie Expressivität der vorgeschlagenen Modelle mitunter erhebliche Unterschiede aufweist [GU96].

Diese Tatsache resultiert vielfach aus dem Wunsch, neben grundsätzlichen Definitionen auch anwendungsbezogenes „Wissen" in Form von Constraints, Prozessen oder Regeln im Rahmen der Ontologie zu repräsentieren. Dementsprechend setzen bekannte, übergreifende Ansätze, wie die *Conceptual Modelling Language* [SWA+94] oder das *Knowledge Interchange Format* [GF92], eine ausdrucksvolle Semantik auf Basis der Prädikatenlogik erster Stufe voraus. Sie ermöglicht die kompakte, weitgehend deklarative Beschreibung komplexer Zusammenhänge sowie eine teilweise automatisierte Ableitung weiterführender Informationen, etwa die Klassifikation von Konzepten an Hand vorgegebener Relationen [Bor95] oder die Planung zielführender Aktionen zur Erfüllung einer übergeordneten Aufgabe [Nwa96]. Allerdings führt die hohe Expressivität dieses sehr allgemeinen semantischen Modells zu typischen Problemen bei effizienter Umsetzung von Deduktion und Beweis abgeleiteter

Aussagen [Woo99]. Darüber hinaus stellt diese Form der Spezifikation hohe Ansprüche an Qualifikation und Erfahrung des Entwicklers, um tatsächlich konsistente, aussagekräftige und letztlich zielführende Ergebnisse zu erhalten.

Aus diesen Gründen beschränkt sich das vorgestellte Framework bewußt auf ein verhältnismäßig einfach gehaltenes, beherrschbares Metamodell einer Ontologie, das in weiten Zügen den bekannten Merkmalen objekt-orientierter Ansätze nachempfunden ist. So ergibt sich die Ableitung weiterführender Informationen ausschließlich über eine geringe Anzahl an Relationen mit vordefinierter Semantik, insbesondere der Generalisierungs- und Interpretationsbeziehung zwischen Konzepten. Dies ermöglicht zunächst eine weitgehend intuitive Definition grundlegender Konzepte und ihrer Zusammenhänge sowie später einen flexiblen und effizienten Abgleich zwischen erwünschter und angebotener Funktionalität. Die eigentliche Verknüpfung ausgewählter Komponenten erfolgt im weiteren Verlauf nach überwiegend technischen Anteilen der zugehörigen CUC-Beschreibungen, so daß hierfür kein entsprechend komplexes semantisches Modell im Rahmen der Ontologie vorauszusetzen ist. Diese Vereinfachung durch klare Trennung der grundlegenden Aufgaben unterscheidet die vorgeschlagene Lösung deutlich von herkömmlichen, logikbasierten Ansätzen.

Andererseits legen die so zusammengefaßten Eigenschaften des in Abschnitt 3.3 beschriebenen Metamodells den Vergleich mit objekt-orientierten Ansätzen der Systementwicklung nahe. Diese führen das explizite Konzept eines *Typs* für Objekte bzw. deren Klasse ein, um den gleichartigen und fehlerfreien Umgang mit zusammengehörigen Elementen hinsichtlich gemeinsamer Merkmale zu gewährleisten [CW85]. Das zugrundeliegende Typsystem einer entsprechenden Programmiersprache überprüft die Kompatibilität der an einer Interaktion beteiligten Objekte an Hand festgelegter Regeln und Beziehungen zwischen den zugehörigen Typen. Hierbei führt das Konstrukt der *Vererbung* zwischen Klassen zur Auffassung eines *Subtyps*, der an Stelle seines jeweils übergeordneten Typs im Rahmen von Ausdrücken und Anweisungen verwendet werden kann [Mey97]. Diese für objekt-orientierte Ansätze charakteristische Eigenschaft erlaubt eine angemessene Modellierung von Generalisierung und Klassifikation grundlegender Konzepte im Anwendungsbereich, die bereits bei frühen Sprachen wie Simula [DMN68] als besonderer Vorzug angesehen wird [MMMP90].

Allerdings ist das Konzept der Vererbung in gängigen objekt-orientierten Ansätzen so allgemein umgesetzt, daß auch andere Ziele der Systementwicklung mit seiner Hilfe verwirklicht werden. Insbesondere die vereinfachte Wiederverwendung bereits implementierter Funktionalität repräsentiert eine weitere bedeutsame Motivation für den Einsatz von Vererbung [Tai96]. In diesem Fall benutzt oder verändert die Implementierung einer Subklasse

die vorgegebenen Attribute und Methoden einer übergeordneten Klasse, um die jeweils spezifisch angebotene Funktionalität mit möglichst geringem Aufwand zu realisieren. Hierfür ist jedoch in der Regel eine detaillierte Kenntnis der Implementierung aller übergeordneten Klassen erforderlich. Darüber hinaus ist durch diese Form der Vererbung eine vollständige Kompatibilität mit dem Supertyp nicht zu gewährleisten, weil das Verhalten einer abgeleiteten Klasse beliebig weitreichend verändert werden kann. Aus diesen Gründen wird die Vermischung von konzeptueller Modellierung und Wiederverwendung bei objekt-orientierten Ansätzen in der Literatur durchaus kritisch betrachtet [Ame87, Sak89, Tai96].

Diese Problematik wird in der vorliegenden Arbeit vermieden, indem Modellierung des Anwendungsbereichs und Wiederverwendung existierender Komponenten durch eigene, klar getrennte Bereiche des vorgestellten Frameworks behandelt werden. Daher werden unterschiedliche, jeweils besonders geeignete Modelle und Verfahren bereitgestellt, welche den spezifischen Anforderungen auf logischer und technischer Ebene der Aufgabenstellung besser gerecht werden. So erlaubt beispielsweise die Interpretation als vordefinierte Relation der Ontologie eine genaue Beschreibung anwendungsbezogener Verhältnisse innerhalb unterschiedlicher Domänen, die sich mit bekannten objekt-orientierten Ansätzen nicht unmittelbar nachvollziehen läßt (vgl. Abbildung 3.6).

Darüber hinaus führt die festgelegte Bedeutung des eingeführten Metamodells zu einer weitergehenden, flexibel anwendbaren Auffassung der semantischen Kompatibilität zwischen Konzepten, die letztlich eine tolerante Auswahl geeigneter funktionaler Komponenten ermöglicht. Die hierbei verwendeten Definitionen und Regeln unterscheiden sich wesentlich von den strikten, stark eingeschränkten Vorgaben herkömmlicher objekt-orientierter Typsysteme [CW85], da zunächst keine umfassende Kompatibilität hinsichtlich Struktur und Verhalten der betrachteten Elemente vorausgesetzt wird. Dieser Aspekt wird erst später, auf technischer Ebene des Frameworks im Verlauf der Prototyp-Generierung berücksichtigt. Allerdings kann auf diese Weise keine fehlerfreie Interaktion der beteiligten Komponenten a priori gewährleistet werden, so daß ggf. entsprechende Adapter zur Vermittlung erforderlich sind. Der resultierende Mangel an Zuverlässigkeit im Vergleich zu den Ergebnissen einer durchgehend objekt-orientierten Systementwicklung erscheint gegenüber den erzielten Vorteilen hinsichtlich Flexibilität und Automatisierung bei Konstruktion funktionaler Prototypen vertretbar.

Eine solche Abwägung widersprüchlicher Zielsetzungen betrifft auch die Einschätzung gegenwärtiger Absätze zur komponentenbasierten Softwareentwicklung. Zwar verzichten diese weitgehend auf Vererbung als wesentliches Mittel der Wiederverwendung, jedoch setzen sie üblicherweise einen globalen

Namensraum für Schnittstellen voraus, deren festgelegte Beziehungen und beabsichtigte Bedeutung zur kompatiblen Verknüpfung vorhandener Komponenten herangezogen werden. Hierdurch ergibt sich zunächst eine stärkere Entkopplung der Bestandteile eines Systems sowie eine durchgängige Black-Box Wiederverwendung bereits implementierter Funktionalität [Szy98]. Allerdings ist die Annahme eines globalen Namensraums mit eindeutigen, durch ihren jeweiligen Typ spezifizierten Schnittstellen in der Praxis aufgrund unabhängiger Hersteller und konkurrierender Standards häufig nicht zutreffend, wie in Abschnitt 3.1.3 angeführt wird. Dies beschränkt die Möglichkeiten zugehöriger Typsysteme deutlich, auch wenn bereits entsprechend leistungsfähige Erweiterungen für hochgradig verteilte und dynamische Systeme vorgeschlagen werden [Nie93, BBSDS97, Pun97, Gie01].

Zudem ist die praktische Umsetzung und Anwendung eines derartigen Typsystems als verhältnismäßig aufwendig einzuschätzen, da es letztlich der endgültigen Konstruktion zuverlässiger und performanter Anwendungen aus vorhandenen Komponenten dient. Dies erfordert zumindest eine genaue, vollständige und formal fundierte Beschreibung des Verhaltens einer Komponente bzw. des Protokolls zur Benutzung ihrer Schnittstelle. Demgegenüber beschränkt sich der vorgestellte Ansatz auf die Erstellung vorläufiger, explorativer Prototypen, deren erwünschte Funktionalität zu Beginn des Verfahrens nicht eindeutig festgelegt ist. Deshalb kann auf ein entsprechend komplexes Typsystem und aufwendige Spezifikation von Verhalten zugunsten einer vereinfachten und pragmatischen Umsetzung verzichtet werden. Im Hinblick auf die betrachtete Problemstellung kann somit eine durch die Ontologie erreichte Entflechtung von logischer und technischer Kompatibilität als ein wesentlicher eigenständiger Beitrag der vorliegenden Arbeit aufgefaßt werden.

Component Use Cases

Das zentrale Konzept eines komponentenbezogenen Anwendungsfalls verbindet logische und technische Ebene des vorliegenden Ansatzes. Es erweitert die herkömmliche Auffassung eines Anwendungsfalls als wohlbekanntes Mittel der Anforderungsanalyse um technische Anteile zur Verknüpfung unterschiedlicher Komponenten. Hierbei wird die Modellierung eines CUC auf logischer Ebene bewußt auf einfache Kombinationen aus Elementen der Ontologie beschränkt, um einen toleranten Abgleich zwischen gewünschter und angebotener Funktionalität zu erreichen. Die vorgeschlagene Beschreibung von Funktionalität als Manipulation von Konzepten entspricht einer häufig gewählten Konvention zur Benennung von Anwendungsfällen, wie in Abschnitt 3.4 ausführlich erläutert wird. Auf technischer Ebene eines CUC wird

die so spezifizierte Funktionalität durch ein einfaches operatives Modell von Verhalten ergänzt, das eine effektive und effiziente Generierung funktionaler Prototypen ermöglicht.

Diese integrale Verbindung aus anwendungsbezogenen und technischen Elementen ist vergleichbar mit aktuellen Ansätzen, die sog. *Business Objects* als wesentliche Bestandteile der Systementwicklung propagieren [OMG01c, ES98]. Ein derartiges Business Object repräsentiert unmittelbar ein wohlbekanntes Element des jeweiligen Anwendungsbereichs, das als solches auf konzeptueller Ebene manipuliert, kommuniziert oder mit anderen Objekten kombiniert werden kann [Szy98]. Hierbei erlauben vorgegebene Standards und übergeordnete Geschäftsprozesse eine rasche und zuverlässige Konstruktion funktionaler Systeme aus vorhandenen Business Objects bzw. deren Implementierung. Allerdings wird die praktische Umsetzung dieses Ansatzes meist als umfangreiches objekt-orientiertes Framework realisiert, wie etwa das Projekt *SanFrancisco* [IBM01a] verdeutlicht. Dies führt nicht zuletzt durch den weitreichenden Einsatz von Vererbung häufig zu durchaus komplexen Abhängigkeiten zwischen den zahlreichen Bestandteilen eines solchen Frameworks und mithin zu einem verhältnismäßig hohen Aufwand für Einarbeitung und Anwendung. Zudem ist die Kombination unabhängig entwickelter Frameworks in der Regel äußerst schwierig, da letztlich unterschiedliche Software Architekturen zu integrieren sind [GAO95]. Schließlich fehlen gegenwärtig Ansätze, die Business Objects möglichst tolerant und weitgehend automatisiert zu übergeordneten Systemen verknüpfen.

Diese für exploratives Rapid Prototyping wesentliche Aufgabe wird durch den vorliegenden Ansatz weitaus besser erfüllt. Die hierbei prinzipiell ebenfalls zu erwartenden Schwierigkeiten bei Integration von Komponenten unterschiedlicher Hersteller werden durch eine Beschränkung auf möglichst einfache Interaktionen mit weitgehend lokalisierter Funktionalität vermieden oder zumindest deutlich verringert. Diese pragmatische Einschränkung erscheint hinsichtlich der übergeordneten Zielsetzung sowie den insgesamt erzielten Vorteilen vertretbar. Darüber hinaus können zukünftig mögliche Erweiterungen für komplexe Interaktionen auf Basis vorgegebener Software Architektur schrittweise in die vorgestellte Lösung integriert werden (siehe Abschnitt 5.4). Dies betrifft grundsätzlich auch Bemühungen um übergreifende Standards für Business Objects und deren Zusammenspiel in ausgewählten Anwendungsbereichen, wie sie gegenwärtig etwa die OMG in entsprechenden Arbeitsgruppen entwickelt [OMG01c].

Hierfür ist letztlich eine genaues Verständnis der Interaktion zwischen Komponenten des Systems erforderlich. Die in dieser Arbeit eingeführte Modellierung von Interaktionen als eigenständiger Bestandteil der Spezifikation einer Komponente entspricht in diesem Zusammenhang dem Kon-

zept des *Konnektors* in aktuellen Ansätzen zur Beschreibung von Software-Architektur [SG95, AG97]. Ein solcher Konnektor beinhaltet ein formal spezifiziertes Protokoll, das beteiligte Rollen und Ablauf einer Interaktion möglichst eindeutig charakterisiert. Hierbei erlaubt das zugrundeliegende semantische Modell die Überprüfung wichtiger Eigenschaften der Interaktion, beispielsweise eine korrekte Zuordnung zwischen Komponenten und Rollen hinsichtlich ihres Verhaltens [AG97].

Allerdings erfordern derart weitreichende Ansätze zur Verknüpfung funktionaler Komponenten in aller Regel komplexe Spezifikationen für Komponenten und Konnektoren, die sich in ihrer Sprache typischerweise an dem vorausgesetzten semantischen Modell, etwa CSP [Hoa85], orientieren. Somit werden wiederum ausgesprochen hohe Anforderungen an Qualifikation und Erfahrung des Benutzers gestellt. Zudem ist die vollständig automatisierte Ableitung weiterführender Eigenschaften aufgrund der umfassenden Expressivität der verwendeten Modelle nur eingeschränkt möglich. Somit eignen sich diese Ansätze im Gegensatz zur vorliegenden Arbeit besser zur überwiegend manuellen Konstruktion endgültiger Systeme mit hohen Ansprüchen an Zuverlässigkeit und Performanz.

Zusammenfassend läßt sich feststellen, daß die in dieser Arbeit eingeführte Auffassung eines komponentenbezogenen Anwendungsfalls eine Reihe von bewährten, anerkannten oder zumindest vielversprechenden Ideen und Konzepten des Software Engineering aufgreift. Jedoch unterscheidet sich deren Ausprägung und gewählte praktische Umsetzung aufgrund der besonderen Zielsetzung deutlich von herkömmlichen Ansätzen. Diese Einschätzung kann als Beleg für Relevanz und Innovation der erarbeiteten Konzeption herangezogen werden.

Prototyp-Generierung

Wie in Abschnitt 3.6 erläutert wird, läßt sich die Erstellung funktionaler Prototypen in die grundlegenden Schritte Komponenten-Auswahl, Adapter-Integration sowie Varianten-Generierung und -Optimierung untergliedern (vgl. Abbildung 3.17). Während die tolerante Auswahl potentiell geeigneter Komponenten eine unmittelbare Anwendung der Regeln und Zusammenhänge der Ontologie darstellt, können die vorgeschlagenen Verfahren zur Konstruktion und Verbesserung der erhaltenen Prototyp-Varianten gesondert diskutiert und mit bestehenden Lösungen verglichen werden.

So folgt das in Abschnitt 3.6.3 eingeführte Modell eines Adapters grundsätzlich dem gleichnamigen Entwurfsmuster [GHJV94] zur Vermittlung zwischen technisch inkompatiblen Schnittstellen bei Verwendung der gleichen

Komponente. Allerdings orientiert sich die gewählte praktische Umsetzung dieses Modells an den besonderen Vorgaben der komponentenbasierten Softwareentwicklung, die in der Regel Delegation gegenüber Vererbung oder Modifikation der Implementierung zur Anpassung einer gegebenen Komponente bevorzugen [BRSV00a]. Auf diese Weise kann von technischen Details der Implementierung abstrahiert werden, da lediglich zwischen den beteiligten Schnittstellen vermittelt wird. Zudem ist die hierfür erforderliche Funktionalität im Rahmen des jeweiligen Adapters als Einheit zusammengefaßt, so daß deren zukünftige Änderung, Weiterentwicklung oder auch Wiederverwendung erheblich vereinfacht wird.

Obwohl ein derartiger Adapter im allgemeinen Fall manuell zu erstellen ist, läßt sich dieser unter bestimmten Voraussetzungen auch automatisch generieren. Diese wesentliche Verbesserung gegenüber bekannten Ansätzen betrifft insbesondere unterschiedliche Repräsentationen des gleichen Konzepts der Ontologie, deren jeweiliger Zustand durch einfache Manipulationen mit festgelegter Bedeutung wechselseitig abgeglichen werden kann. Hierbei ermöglicht der gemeinsame Bezug zur Ontologie sowie eine Modellierung der entsprechenden Manipulation als ausgezeichneter, komponentenbezogener Anwendungsfall eine weitgehend schematische Umsetzung des erforderlichen Abgleichs.

Allerdings kann die korrekte Funktion eines auf diese Weise generierten Adapters durch das vorgestellte Verfahren nicht umfassend gewährleistet werden. So werden gegenwärtig etwaige, lokal oder übergreifend gültige Konsistenzbedingungen für Konzepte und Relationen der Ontologie nicht berücksichtigt. Weiterhin sind Seiteneffekte der Implementierung einer Repräsentation nicht auszuschließen, welche den Zustand der in Beziehung gesetzten Konzepte auf unvorhergesehene Weise verändern. Zuletzt ist zu berücksichtigen, daß die vorausgesetzten Annahmen über Struktur und Zustand einer gegebenen Repräsentation bei unabhängig entwickelten Komponenten möglicherweise nicht übereinstimmen. Das vereinfachte Modell des Zustands einer Repräsentation als Zusammenfassung bestehender Beziehungen zu anderen Konzepten der Ontologie läßt sich bei solchen Verhältnissen offensichtlich nicht zur Generierung eines entsprechenden Adapters nutzen.

Immerhin wird durch das vorgeschlagene Verfahren zumindest ein teilweise funktionales Gerüst erstellt, das anschließend durch den Entwickler modifiziert und ergänzt werden kann. Hierbei liefert der Bezug zur Ontologie wertvolle Anhaltspunkte zur Ermittlung der tatsächlich benötigten Funktionalität. Dieses pragmatische Vorgehen unter Einbeziehung eines definierten Modells des Anwendungsbereichs unterscheidet das vorgeschlagene Verfahren von aufwendigen formalen Ansätzen, die besondere Protokolle als vollständige Spezifikation des beobachtbaren Verhaltens zur automatisierten

Adapter-Generierung nutzen [YS97]. Somit kann der vorliegende Ansatz hinsichtlich der untersuchten Aufgabenstellung als durchaus eigenständiger und innovativer Beitrag aufgefaßt werden. Zudem eröffnen die in Abschnitt 5.1.1 und 5.3.2 vorgestellten Erweiterungen um Constraints und abstrakte Strategien zur Konvertierung von Repräsentationen zahlreiche Möglichkeiten für zukünftige Verbesserungen.

Die in Abschnitt 3.6.5 erläuterte Optimierung der generierten Prototyp-Varianten durch einen Genetischen Algorithmus repräsentiert die Anwendung eines bekannten und vielfach bewährten Verfahrens zur Behandlung kombinatorischer Probleme [Gol89]. Die Betrachtung ausführbarer Programme als Gegenstand der Optimierung legt in diesem Zusammenhang einen Vergleich mit Ansätzen des *Genetic Programming* [Koz92] nahe. Ungeachtet der tatsächlichen Vielfalt an unterschiedlichen Arbeiten in diesem Bereich [LQ95], wird hierbei ebenfalls eine Population aus zahlreichen Programmen untersucht, die sich in einem evolutionären Prozeß durch wiederholte Anwendung verschiedener Genetischer Operatoren und Selektion nach überdurchschnittlicher Fitneß weiterentwickelt.

Ein solches Programm als Individuum der Population setzt sich aus *Terminalen* und *Funktionen* zusammen, die im Rahmen einer hierarchischen Struktur kombiniert werden. Beispielsweise läßt sich die Berechnung arithmetischer Ausdrücke durch einen Baum repräsentierten, dessen innere Knoten mit einem mathematischen Operator als Funktion assoziiert sind, während an den Blättern Konstanten oder Variablen als Terminal angegeben werden. In diesem Fall bestimmt die Annäherung an die Auswertung eines fest vorgegebenen Ausdrucks mit einer Reihe von Eingaben über die erzielte Fitneß eines so dargestellten Programms. Dementsprechend werden überdurchschnittlich genaue Programme über zufällige Veränderung ihrer Bestandteile oder Austausch von Teilbäumen zur Konstruktion neuer Individuen herangezogen.

Obwohl dieser grundlegende Lösungsansatz zur automatisierten Konstruktion funktionaler Systeme eine gewisse Bedeutung erreicht hat, etwa im Bereich der Signalverarbeitung oder Robotik, so ergeben sich doch häufig unbefriedigende Ergebnisse bei dessen praktischer Anwendung [LQ95]. Diese Tatsache resultiert einerseits aus der variablen Länge der gewählten Repräsentation für Individuen der Population, d.h. die hierarchische Struktur eines gegebenen Programms wird typischerweise im Verlauf des Verfahrens durch Rekombination fortlaufend erweitert, ohne die angebotene Funktionalität merklich zu verbessern [Lan00]. Andererseits besitzt Genetic Programming oftmals einen sehr generischen und zufälligen Charakter, der bei tatsächlich schwierigen Problemen wenig Aussicht auf Erfolg verspricht. Zu-

dem berücksichtigt der ursprüngliche Ansatz keine weiterführenden Informationen über den jeweiligen Anwendungsbereichs oder besondere technische Gegebenheiten. Beispielsweise wird häufig ein universell kompatibler Typ für Terminale vorausgesetzt, um syntaktische Schwierigkeiten bei Kombination unterschiedlicher Funktionen zu vermeiden. Jedoch bestimmt gerade ein leistungsfähiges und reichhaltiges Typsystem über die erfolgreiche Konstruktion übergeordneter Systeme aus vorhandenen Komponenten, wie in Abschnitt 6.2.2 diskutiert wird.

Demgegenüber beschränkt sich der vorliegende Ansatz bewußt auf eine evolutionäre Optimierung ausführbarer Programme, deren Struktur und Zusammensetzung durch manuell erstellte Spezifikationen sowie geeignete Modelle des Anwendungsbereichs bereits in groben Zügen festgelegt ist. Somit ist lediglich die genaue Ausprägung der jeweils erwünschten Funktionalität aus einer überschaubaren Menge an potentiell geeigneten Komponenten auszuwählen bzw. mit anderen Individuen der Population zu neuen Varianten zu kombinieren. Hierbei kann die eigentliche Verknüpfung einfach durchgeführt werden, weil die wesentliche Funktionalität des Systems durch vorhandene Komponenten über klar definierte Interaktionen erbracht wird. Daher besitzt die Repräsentation eines Programms, also der Konstruktionsplan einer Prototyp-Variante, eine fest vorgegebene Länge, welche durch die Anzahl an unterschiedlichen Anwendungsfällen der Funktionalen Spezifikation bestimmt wird. Dies erleichtert die praktische Umsetzung eines heuristischen Verfahrens mit Hilfe Genetischer Operatoren erheblich und ermöglicht eine unmittelbare Analogie zu den realen Verhältnissen in der Biologie (vgl. Abschnitt 5.3.4).

Insgesamt ergibt sich somit durch die gewählte Konzeption eine deutlich bessere Aussicht auf eine erfolgreiche Anwendung des evolutionären Ansatzes zur automatisierten Konstruktion funktionaler Systeme. Erste praktische Erfahrungen mit der in Kapitel 4 vorgestellten Referenz-Implementierung erscheinen in diesem Zusammenhang durchaus vielversprechend. Allerdings muß die grundlegende Vorgehensweise letztlich durch umfangreiche empirische Untersuchungen mit unterschiedlichen Anwendungsbereichen und realistischen Problemgrößen gestützt werden. Die so erhaltenen Erkenntnisse bestimmen maßgeblich über mögliche Verbesserungen, die zum großen Teil bereits in Abschnitt 5.3.4 angedeutet sind.

6.3 Zusammenfassung und Bewertung

Die Beurteilung des vorgeschlagenen Frameworks hinsichtlich der in Abschnitt 3.1 zusammengefaßten Anforderungen an einen Ansatz für komponen-

tenbasiertes Rapid Prototyping ermöglicht eine umfassende und weitgehend objektive Einschätzung des erreichten Erfolgs. So ist die Expressivität der angebotenen Mittel zur Beschreibung von Funktionalität der grundlegenden Aufgabenstellung angemessen. Die auf logischer Ebene gewählte Auffassung von Funktionalität als Manipulation von Konzepten entspricht einer intuitiven, an die natürliche Sprache angelehnten Formulierung, auch wenn weitergehende Zusammenhänge und Abhängigkeiten nicht berücksichtigt werden. Demgegenüber ist die Expressivität des Modells auf technischer Ebene zur Beschreibung von Interaktion zwischen Komponenten durch entsprechende Erweiterungen zu verbessern.

Ein besonderer Vorzug der erarbeiteten Lösung ist die verhältnismäßig geringe Komplexität der gewählten Modelle und Verfahren. Das Metamodell einer Ontologie zur Beschreibung des Anwendungsbereichs ähnelt bekannten objekt-orientierten Ansätzen, während die technischen Anteile eines komponentenbezogenen Anwendungsfalls durch einfache Konstrukte, wie Aufruf einer Operation oder Belegung eines Parameters, angegeben werden können. Dies erleichtert einerseits die praktische Umsetzung des vorgestellten Ansatzes, führt andererseits aber auch zu möglichen Abstrichen bei Zuverlässigkeit, Effektivität und Effizienz der generierten Prototypen.

Die klare Trennung zwischen logischer und technischer Ebene des Frameworks vereinfacht die Verwendung unabhängig entwickelter Komponenten erheblich. So erfolgt die Auswahl potentiell geeigneter Komponenten zunächst ausschließlich nach anwendungsbezogenen Kriterien, welche durch Zusammenhänge der Ontologie festgelegt sind. Anschließend wird in einem gesonderten Schritt des Verfahrens die technische Kompatibilität der beteiligten Komponenten bzw. ihrer Schnittstellen überprüft. Hierbei werden zur Verknüpfung ggf. entsprechende Adapter herangezogen, die zumindest in Teilen ebenfalls automatisch erstellt werden können. Somit wird der vorliegende Ansatz den Gegebenheiten der komponentenorientierten Softwareentwicklung weitgehend gerecht.

Das Zusammenspiel der erarbeiteten Modelle und Verfahren erlaubt eine weitgehend automatisierte Konstruktion funktionaler Prototypen aus vorhandenen Komponenten. Darüber hinaus wird durch den toleranten Abgleich zwischen erwünschter und angebotener Funktionalität eine Betrachtung unterschiedlicher Prototyp-Varianten unterstützt. Beide Merkmale repräsentieren wesentliche Anforderungen an einen effektiven Ansatz für komponentenbasiertes, exploratives Rapid Prototyping. Jedoch kann eine vollständig automatisierte, fehlerfreie und letztlich zielführende Erstellung ausführbarer Prototypen nicht gewährleistet werden. Daher ist die Qualität der erzielten Ergebnisse sowie der Grad der insgesamt erreichten Automatisierung durch entsprechende Erweiterungen zu verbessern.

Hingegen ist eine erfolgreiche Umsetzung der vorgestellten Lösung für alle gegenwärtig bedeutsamen technischen Plattformen sichergestellt, da keine spezifischen Annahmen vorausgesetzt werden. Aufgrund der Abbildung eines komponentenbezogenen Anwendungsfalls auf eine einfache Folge von Operationsaufrufen können sogar unterschiedliche Plattformen gemeinsam eingesetzt werden, sofern die gewählte Infrastruktur eine entsprechende Vermittlung der synchronen Kommunikation bereitstellt. Allerdings ist das zugrundeliegende Modell der asynchronen Kommunikation zu spezifisch und sollte daher für den allgemeinen Einsatz in heterogener Umgebung überarbeitet werden.

Die Skalierbarkeit hinsichtlich realistischer Problemgrößen repräsentiert eine bedeutende Anforderung an eine praxisgerechte Lösung. Während der erforderliche Aufwand zur toleranten Auswahl geeigneter Komponenten im wesentlichen linear mit ihrer Anzahl wächst, führt die Kombination verschiedener Komponenten zu einer exponentiell steigenden Anzahl unterschiedlich zusammengesetzter Prototyp-Varianten. Die vorliegende Arbeit begegnet dieser grundlegenden Problematik durch Anwendung einer bewährten Heuristik zur kombinatorischen Optimierung, welche den Umfang an tatsächlich betrachteten Varianten effektiv beschränkt. Jedoch bedingt die erforderliche, anwendungsbezogene Bewertung der generierten Varianten im Verlauf des Genetischen Algorithmus einen nicht unerheblichen manuellen Aufwand. Deshalb ist eine Anpassung oder Weiterentwicklung der gewählten Heuristik nach zukünftigen praktischen Erfahrungen anzustreben.

Eine Abschätzung der Anwendbarkeit des vorgestellten Ansatzes erfordert eine differenzierte Betrachtung der getroffenen Annahmen über den Anwendungsbereich sowie Umfang und Komplexität der betrachteten Funktionalität. So wird zunächst ein gemeinsames Verständnis der grundlegenden Konzepte und ihrer Beziehungen zwingend vorausgesetzt, das im Rahmen einer Ontologie angemessen modelliert werden kann. Weiterhin ist eine Auffassung von Funktionalität als Manipulation von Konzepten erforderlich, die vornehmlich bestimmten Komponenten zugeordnet werden kann. Zuletzt wird angenommen, daß die technische Realisierung eines komponentenbezogenen Anwendungsfalls durch einfach gehaltene Interaktionen erfolgt, deren Ausführung nur weitgehend lokale Auswirkungen auf den Zustand des Gesamtsystems aufweist. Die genannten Voraussetzungen erlauben somit keine allgemeine Übertragung der erzielten Ergebnisse auf beliebige Anwendungsbereiche mit komplex zusammengesetzter Funktionalität. Dennoch können eine Reihe von bedeutsamen Aufgaben wirkungsvoll unterstützt werden, etwa die Verarbeitung und Analyse von Rohdaten, wie der Vergleich mit dem exemplarisch untersuchten Anwendungsbereich belegt. Zudem repräsentiert bereits die Berücksichtigung eines Ausschnitts der insgesamt angebotenen

Funktionalität eine deutliche Verbesserung gegenüber dem aktuellen Stand der Technik.

Die oben zusammengefaßte Einschätzung wird auch durch den Vergleich mit bestehenden Ansätzen und Konzepten bestätigt. Er erlaubt darüber hinaus eine Einordnung der vorgeschlagenen Lösung sowie eine weiterführende Beurteilung des eigenständigen Beitrags. So können gegenüber vollständig manueller Entwicklung, etwa mit Hilfe von Skriptsprachen und deren integrierten Werkzeugen, funktionale Prototypen wesentlich schneller und mit geringerem Aufwand konstruiert werden. Allerdings ist hierbei eine verminderte Flexibilität und Zuverlässigkeit bei Verknüpfung bereitgestellter Komponenten zu erwarten. Diese Abwägung ist hinsichtlich der übergeordneten Aufgabenstellung vertretbar.

Im Vergleich mit bekannten generativen Ansätzen der Softwareentwicklung zeichnet sich das vorgestellte Framework durch eine besondere Berücksichtigung unabhängig entwickelter Komponenten aus. Zudem stellt die Komplexität der eingeführten Modelle und Verfahren keine besonderen Anforderungen an Qualifikation und Erfahrung des Benutzers. Demgegenüber erlauben generative Ansätze auch die Entwicklung endgültiger Systeme unter Einbeziehung bedeutender nicht-funktionaler Anforderungen. Dennoch erscheint der vorliegende Ansatz im Hinblick auf die untersuchte Problemstellung letztlich besser geeignet, da keine vollständige Spezifikation der verwendeten Software-Architektur zwingend vorausgesetzt und somit ein überwiegend experimentelles und exploratives Vorgehen unterstützt wird.

Aufgrund der gewählten Konzeption läßt sich das vorgestellte Framework auch mit Forschungsarbeiten über Software Agenten vergleichen. Hierbei repräsentiert die Erfüllung einer übergeordneten Aufgabe durch eine vom Benutzer nicht festgelegte Kombination elementarer Schritte das gemeinsame Merkmal beider Ansätze. Allerdings werden Software Agenten meist als weitgehend autonome Systeme in sehr dynamischen, verteilten und heterogenen Umgebungen eingesetzt. Dementsprechend werden in der Regel nur einfache und eindeutig festgelegte Aufgaben erfüllt, welche den Anforderungen des explorativen Rapid Prototyping nicht gerecht werden.

Neben den oben aufgeführten Ansätzen mit ähnlicher Zielsetzung beinhaltet das erarbeitete Framework auch Bezüge zu partiell vergleichbaren Lösungen. So sind bereits Sprachen bekannt, die eine Modellierung des Anwendungsbereichs als Ontologie ermöglichen. Diese weisen aufgrund ihrer zugrundeliegenden Semantik eine ausgesprochen hohe Expressivität auf. Diese Eigenschaft führt andererseits im Vergleich mit der vorliegenden Arbeit aber auch zu einer höheren Komplexität bei Erstellung, Pflege und Anwendung

der angebotenen Modelle. Zudem erscheint der objekt-orientierte Charakter der vorgestellten Lösung für die praktische Systementwicklung besser geeignet. Im Vergleich mit objekt-orientierten Typsystemen wiederum wird durch die Ontologie eine klare Trennung zwischen Modellierung des Anwendungsbereichs und Wiederverwendung vorhandener Funktionalität erreicht. Darüber hinaus erlaubt das eingeführte Metamodell eine genauere Abbildung des Anwendungsbereichs sowie eine flexible und tolerante Auswahl unabhängig entwickelter Komponenten.

Das zentrale Konzept eines komponentenbezogenen Anwendungsfalls entspricht auf logischer Ebene des Frameworks einem bewährten Modell der Anforderungsanalyse. Seine integrale Verbindung mit technischer Umsetzung durch entsprechend spezifizierte Interaktionen legt den Vergleich mit Business Objects als Mittel der Systementwicklung nahe. Allerdings werden entsprechende Ansätze in der Regel als durchgängig objekt-orientiertes Framework realisiert. Außerdem wird hierbei eine pragmatische und weitgehend automatisierte Konstruktion funktionaler Prototypen nicht unterstützt. Diese Aussage trifft ebenfalls auf formale Ansätze zur Spezifikation von Software Architektur zu. Jedoch beinhalten aktuelle Arbeiten auf diesem Gebiet das Konzept einer Interaktion zwischen Komponenten der Architektur als eigenständiges Element der Modellierung. Sie stützen somit das erarbeitete Modell eines komponentenbezogenen Anwendungsfalls und verdeutlichen eine mögliche Richtung zukünftiger Weiterentwicklung.

In vergleichbarer Weise kann die Generierung und Integration von Adaptern zur Vermittlung zwischen unabhängig entwickelten Komponenten als Anwendung bekannter und bewährter Techniken aufgefaßt werden. Hierbei eröffnet der Bezug zur gemeinsamen Ontologie des Anwendungsbereichs weiterführende Möglichkeiten zur teilweise automatisierten Erstellung dieser wichtigen Bestandteile komponentenbasierter Prototypen.

Der vorgeschlagene Genetische Algorithmus zur Optimierung unterschiedlich zusammengesetzter Prototyp-Varianten entspricht einer oftmals erfolgreichen Heuristik zur Lösung kombinatorischer Probleme. Im Unterschied zu bekannten Ansätzen des Genetic Programming wird hierbei Struktur und Zusammenhang des gesuchten Systems bereits in groben Zügen vorgegeben. Diese Tatsache führt in Verbindung mit dem engen Bezug zum jeweiligen Anwendungsbereich zu einer verbesserten Qualität automatisch erstellter Programme.

Mit der getrennten Betrachtung maßgeblicher Anforderungen an einen tragfähigen Ansatz für komponentenbasiertes Rapid Prototyping sowie dem umfassenden Vergleich mit bekannten Ansätzen läßt sich abschließend eine

Charakterisierung und Bewertung der erzielten Ergebnisse dieser Arbeit angeben. So können die wesentlichen Vorzüge der vorgestellten Lösung wie folgt zusammengefaßt werden:

- Das Framework repräsentiert einen pragmatischen Ansatz zur weitgehend automatisierten Konstruktion funktionaler Prototypen aus vorhandenen Komponenten. Die eingeführten Modelle und Verfahren weisen einen angemessenen Grad der Formalisierung auf, der keine besonderen Anforderungen an Qualifikation und Erfahrung des Benutzers stellt.

- Die grundlegende Konzeption erlaubt eine überwiegend einfache praktische Umsetzung der erarbeiteten Ergebnisse für alle gegenwärtig bedeutsamen technischen Plattformen. Aufgrund der hohen Skalierbarkeit können auch umfangreiche Problemgrößen behandelt werden.

- Das gewählte Vorgehen beinhaltet eine klare Ausrichtung auf unabhängig entwickelte Komponenten sowie den grundsätzlich unbestimmten Charakter initialer funktionaler Anforderungen an das zu entwickelnde System.

- Die vorgeschlagenen Lösungen in den jeweiligen Bereichen des Frameworks berücksichtigen bekannte und bewährte Ansätze aus Forschung und Praxis der Softwareentwicklung. Diese werden in der vorliegenden Arbeit zweckmäßig erweitert, angepaßt und auf innovative Weise verknüpft.

- Die erreichte Trennung zwischen logischer und technischer Ebene der Aufgabenstellung führt zu einer insgesamt übersichtlichen Struktur der vorgestellten Lösung. Somit können unterschiedliche Bereiche überwiegend unabhängig voneinander weiterentwickelt und verbessert werden.

Darüber hinaus lassen sich einzelne Bestandteile der Lösung auch isoliert für andere Zwecke der Systementwicklung einsetzen. Beispielsweise erlaubt die weitgehend deklarative Beschreibung von Funktionalität als Manipulation von Konzepten eine prägnante, anwendungsbezogene Indizierung verfügbarer Komponenten in umfangreichen, weltweit zugänglichen Verzeichnissen. Diese können anschließend mit den beschriebenen Verfahren auf tolerante Weise nach potentiell geeigneten Bausteinen des späteren Systems durchsucht werden. Somit wird die bisher übliche Einteilung nach unterschiedlichen Kategorien wirkungsvoll ergänzt. Weiterhin läßt sich etwa die Ontologie mit ihren definierten Konzepten und Beziehungen zur Unterstützung von Anforderungsanalyse oder Systemspezifikation einsetzen. Ein derartiges Modell des

Anwendungsbereichs fördert das gemeinsame Verständnis zwischen Anwender und Entwickler, erleichtert die konsistente Bezeichnung in Spezifikation oder Dokumentation des Systems und kann zudem in weiteren Projekten wiederverwendet werden.

Den genannten positiven Merkmalen stehen jedoch folgende Nachteile und Einschränkungen gegenüber:

- Die vorgeschlagenen Modelle und Verfahren können fehlerfreie Konstruktion sowie korrekte Funktion der erstellten Prototypen nicht gewährleisten. Zudem ist in der Regel eine vollständig automatisierte Generierung nicht zu erreichen.

- Der vorgestellte Ansatz ist nicht universell anwendbar. Seine vorausgesetzten Annahmen schränken die möglichen Anwendungsbereiche und Komplexität der unterstützten Funktionalität merklich ein.

- Trotz einer prototypischen Referenz-Implementierung und exemplarisch untersuchtem Anwendungsbereich wurden die erarbeiteten Ergebnisse nicht umfassend empirisch überprüft. Somit kann deren Übertragbarkeit auf die praktische Systementwicklung nicht abschließend nachgewiesen werden.

Gerade die letztgenannte Einschränkung ist als nicht unerheblicher Mangel einer grundsätzlich anwendungsorientierten Lösung einzuschätzen. Dennoch repräsentiert die vorliegende Arbeit einen innovativen und durchaus vielversprechenden Ansatz für komponentenbasiertes Rapid Prototyping. Das gewählte Vorgehen wird als Kompromiß aus angemessener Formalisierung und pragmatischer Umsetzung den besonderen Anforderungen der untersuchten Aufgabenstellung weitgehend gerecht. Hierbei verdeutlichen die zahlreichen möglichen Erweiterungen mit ihrem durchaus unterschiedlichen Charakter auf anschauliche Weise das Potential der vorgestellten Konzeption. Die Bestimmung einer bestmöglichen Strategie nach umfangreichen praktischen Erfahrungen stellt somit eine überaus interessante Herausforderung für zukünftige Arbeiten dar.

7. ZUSAMMENFASSUNG

Die Entwicklung umfangreicher und komplexer Software-Systeme ist eine überaus anspruchsvolle Aufgabe, deren Durchführung mit einem hohen Aufwand an Zeit und Kosten verbunden ist. Daher ist es wünschenswert, die funktionalen Anforderungen an ein zukünftiges System möglichst genau und vollständig zu erfassen, um den Bedürfnissen des Anwenders tatsächlich gerecht zu werden und auftretende Mängel frühzeitig zu entdecken. Funktionale Prototypen des späteren Systems erleichtern diese Teilaufgabe der Systementwicklung erheblich, da somit eine tragfähige Grundlage für den Dialog zwischen Anwender und Entwickler geschaffen wird.

Software-Komponenten bieten sich in besonderer Weise zur Konstruktion funktionaler Prototypen an, weil diese bereits implementierte Funktionalität zusammenfassen und über ausgezeichnete Schnittstellen ihrer Umgebung zur Verfügung stellen. Durch Komposition entstehen so verhältnismäßig schnell übergeordnete Systeme mit umfangreicher Funktionalität. Dennoch verbleiben Suche, Auswahl und Verknüpfung geeigneter Komponenten als überwiegend manuelle Schritte der Konstruktion, die zudem durch ungenügende Beschreibung der angebotenen Funktionalität erheblich erschwert werden.

In dieser Arbeit wird daher ein fortgeschrittener Ansatz für komponentenbasiertes Rapid Prototyping entwickelt, der eine weitgehend automatisierte Erstellung funktionaler Prototypen an Hand vorgegebener Anforderungen und bereitgestellter Komponenten ermöglicht. Auf diese Weise können auch zahlreiche, unterschiedlich zusammengesetzte Prototyp-Varianten mit vertretbarem Aufwand betrachtet werden. Darüber hinaus lassen sich so potentielle Komponenten für Entwurf und Implementierung des späteren Systems frühzeitig hinsichtlich ihrer Eignung beurteilen. Die Effektivität und praktische Relevanz des vorgestellten Ansatzes wird durch eine Referenz-Implementierung sowie einen exemplarisch untersuchten Anwendungsbereich, die Biomolekulare Sequenzanalyse, sichergestellt.

Das zentrale Ergebnis der vorliegenden Arbeit ist ein *konzeptuelles Framework*, das eine klare Trennung zwischen anwendungsbezogener und technischer Ebene der Lösung vorgibt. Der betrachtete Anwendungsbereich wird auf logischer Ebene durch eine *Ontologie* aus verschiedenen Domänen mo-

delliert. Jede Domäne beinhaltet charakteristische Konzepte und deren Beziehungen, wobei ausgezeichnete Relationen mit vordefinierter Semantik, wie Generalisierung oder Interpretation, eine Ableitung weiterführender Zusammenhänge über *semantisch kompatible* Konzepte erlauben.

Mit Bezug auf eine derartige Ontologie kann Funktionalität als Manipulation von Konzepten des Anwendungsbereichs verstanden werden. Diese grundlegende Auffassung dient zur Beschreibung von erwünschter und angebotener Funktionalität im Rahmen einer *Funktionalen Spezifikation* bzw. eines komponentenbezogenen Anwendungsfalls (*Component Use Case*). Letzterer beinhaltet neben diesen anwendungsbezogenen Elementen auch operative, technische Anteile, welche typische Interaktionen bei Benutzung der zugehörigen Komponente angeben. Eine solche Interaktion läßt sich als Sequenz von Operationsaufrufen der beteiligten Schnittstellen modellieren, wobei Konzepte der Ontologie als Parameter oder beobachtbares Ergebnis der Interaktion auftreten. Diese umfassende Beschreibung der angebotenen Funktionalität kann dem ausführbaren Format bestehender Komponenten beigefügt werden, ohne deren Implementierung zu verändern.

Die Suche und Auswahl geeigneter Komponenten wird somit als Abgleich zwischen Funktionaler Spezifikation und komponentenbezogenen Anwendungsfällen verstanden. Hierbei führt der zuvor definierte Begriff der semantischen Kompatibilität unmittelbar zur Auffassung *logisch kompatibler* Anwendungsfälle. Dies ermöglicht eine grundlegende Flexibilität und Toleranz des eingesetzten Verfahrens. Die so getroffene Auswahl führt zu zahlreichen, unterschiedlich zusammengesetzten Prototyp-Varianten, deren Komponenten über ausgezeichnete Rollen der Interaktion sowie Konzepte als Parameter oder Ergebnisse von Operationsaufrufen verknüpft sind. Allerdings erfordert die Kombination von Komponenten unabhängiger Hersteller in der Regel eine Vermittlung zwischen verschiedenen Repräsentationen des gleichen Konzepts der Ontologie. Diese Aufgabe wird von *Adaptern* erfüllt, die entweder explizit durch den Entwickler oder – bei hinreichend einfachen Verhältnissen – auch selbsttätig durch das Framework bereitgestellt werden.

Für die zusammengestellten Prototyp-Varianten kann anschließend Quellcode der verwendeten technischen Plattform generiert, übersetzt und ausgeführt werden. Da nach diesem Schritt mit tatsächlich ungeeigneten oder sogar fehlerhaften Ergebnissen zu rechnen ist, wird in der Folge eine heuristische Optimierung durchgeführt. Der hierfür vorgeschlagene *Genetische Algorithmus* betrachtet Prototyp-Varianten als Individuen einer Population, die sich durch fortgesetzte Anwendung von Selektion, Mutation und Rekombination in einem evolutionären Prozeß weiterentwickelt. Als maßgeblicher Bestandteil einer solchen Heuristik wird eine geeignete Fitneß-Funktion definiert, die sich aus einer automatisch durchgeführten Beurteilung der tech-

nischen Merkmale sowie einer interaktiven Beurteilung der anwendungsbezogenen Merkmale einer Variante zusammensetzt. Dieses Vorgehen erlaubt eine flexible Ausrichtung an verfügbare Ressourcen und erwünschten Grad der Automatisierung. Darüber hinaus kann das Verfahren jederzeit unterbrochen und durch Änderung der Spezifikation ein erneuter Abgleich der Funktionalität herbeigeführt werden. Somit ergibt sich ein insgesamt iterativ organisiertes Verfahren zur Konstruktion funktionaler Prototypen aus vorhandenen Komponenten.

Erweiterbarkeit und Anpassungsfähigkeit repräsentieren wesentliche Merkmale des erarbeiteten Frameworks für komponentenbasiertes Rapid Prototyping. Daher werden zahlreiche Verbesserungen, Weiterentwicklungen und mögliche Alternativen diskutiert, die gegenwärtig bestehende Einschränkungen aufheben, eine höhere Qualität der erstellten Prototypen ermöglichen oder eine praxisgerechte Umsetzung der erzielten Ergebnisse erleichtern.

So können im Bereich der Ontologie *Constraints* als Prädikate über Konzepte und Relationen eingeführt werden, um komplexe Verhältnisse im Anwendungsbereich angemessen zu modellieren. Die Integration von Manipulationen in das zentrale Metamodell der Ontologie führt zu einer insgesamt durchgängigen und systematischen Behandlung aller wesentlichen Elemente zur Beschreibung von Funktionalität. Zuletzt lassen sich Bearbeitung und Pflege einer Ontologie durch geeignete Beschreibungstechniken und Werkzeuge vereinfachen. Aufgrund der gewählten Modellierung können hierfür entsprechende Diagramme und Werkzeuge der UML angepaßt werden. Demgegenüber wird Austausch und Integration unabhängig erstellter Ontologien durch eine textuelle, vorzugsweise XML-basierte Beschreibung vereinfacht. Somit können Teile des erstellten Modells wiederverwendet oder als lokal vorausgesetzte Annahmen über den Anwendungsbereich dem ausführbaren Format einer Komponente beigefügt werden.

Auf logischer Ebene des Frameworks läßt sich die Expressivität der Funktionalen Spezifikation und komponentenbezogener Anwendungsfälle durch Einführung zusätzlicher, über Präpositionen verknüpfte Konzepte der Ontologie weiter verbessern. Auf technischer Ebene können auch komplexe dynamische Abläufe durch neu eingeführte operative Elemente, wie Iteration, Fallunterscheidung oder bedingte Ausführung, angemessen beschrieben werden. Zudem erlaubt die Modellierung eines Component Use Case auch alternative Beschreibungen einer Interaktion, etwa die Angabe eines entsprechenden Zustandsautomaten. Wiederum vereinfacht die Verwendung bekannter grafischer Beschreibungstechniken und Werkzeuge eine Bearbeitung der erstellten Modelle. Langfristig erscheint eine weiterführende Auffassung von Funktionaler Spezifikation und komponentenbezogenen Anwendungsfällen als auto-

matisierbarer Teil eines übergeordneten *Workflows* vielversprechend.

Im Bereich der Prototyp-Generierung läßt sich zunächst der eingeführte Begriff der logischen Kompatibilität zwischen Anwendungsfällen auf Basis der Ontologie verallgemeinern, um eine gesteigerte Toleranz bei Auswahl und Verknüpfung potentieller Komponenten zu erreichen. Weiterhin kann der Grad der Automatisierung bei Generierung von Adaptern durch *Strategien* erhöht werden. Diese legen einen abstrakten Algorithmus zur Konvertierung von unterschiedlich repräsentierten Konzepten fest. Sie lassen sich zudem für eine verbesserte Zuordnung von Komponenten zu Rollen einer Interaktion einsetzen. Zuletzt kann die verwendete Heuristik zur Optimierung von Prototyp-Varianten durch alternative Verfahren, wie etwa *Simulated Annealing*, ersetzt oder ergänzt werden, wobei eingeführte Bewertungskomponenten mit vorgegebenen, anwendungsbezogenen Testfällen eine weitere Verringerung des erforderlichen manuellen Aufwands erlauben.

Neben lokalen Erweiterungen des Frameworks werden auch eine Reihe von übergreifenden, langfristig bedeutsamen Weiterentwicklungen diskutiert. So erlaubt die Einbeziehung von *Software-Architektur*, also zusätzlich bereitgestellte Informationen über geeignete Strukturen und Interaktionen des Systems, eine erheblich verbesserte Qualität der erzielten Ergebnisse. Die konsequente Verfolgung dieses Ansatzes führt zum Begriff der *abstrakten Interaktion*, die ausschließlich mit Hilfe von Konzepten, Manipulationen und operativen Elementen beschrieben wird. Auf diese Weise lassen sich anwendungsbezogene Abläufe spezifizieren, ohne die technischen Details der Implementierung berücksichtigen zu müssen. Abschließend wird eine alternative praktische Umsetzung des Frameworks vorgeschlagen, die Anwendung und zeitliche Abfolge der eingesetzten Verfahren verschränkt. Hierdurch wird ein Übergang von statischer Generierung zu dynamischer Simulation funktionaler Prototypen erreicht, welcher dem besonderen Charakter des explorativen Prototyping besser gerecht wird.

Die Beurteilung und Diskussion der vorgestellten Lösung erfolgt zunächst an Hand der zuvor aufgestellten, allgemeinen Anforderungen an einen tragfähigen Ansatz für komponentenbasiertes Rapid Prototyping. So ist die Expressivität der angebotenen Mittel zur Beschreibung von Funktionalität der betrachteten Aufgabenstellung angemessen. Ein besonderer Vorzug ist die verhältnismäßig geringe Komplexität der gewählten Modelle und Verfahren. Die erreichte Trennung zwischen logischer und technischer Ebene der Problemstellung vereinfacht die Verwendung unabhängig entwickelter Komponenten erheblich. Hierbei wird der kombinatorischen Vielfalt an unterschiedlich zusammengesetzten Prototyp-Varianten durch eine flexibel anpaßbare, evolutionäre Heuristik begegnet. Der vorgeschlagene Ansatz ermöglicht so-

mit eine effektive, skalierbare und weitgehend automatisierte Konstruktion funktionaler Prototypen, die sich auf alle gegenwärtig bedeutsamen technischen Plattformen übertragen läßt.

Eine Abschätzung der praktischen Anwendbarkeit erfordert eine differenzierte Betrachtung seiner wesentlichen Annahmen und Beschränkungen. So wird zunächst ein gemeinsames Verständnis des jeweiligen Anwendungsbereichs zwingend vorausgesetzt, das sich im Rahmen einer entsprechenden Ontologie abbilden läßt. Weiterhin ist eine grundlegende Auffassung von Funktionalität als Manipulation von Konzepten erforderlich, deren Implementierung vornehmlich eindeutig bestimmten Komponenten zugeordnet werden kann. Zuletzt wird angenommen, daß die technische Realisierung eines komponentenbezogenen Anwendungsfalls durch überwiegend einfach gehaltene Interaktionen erfolgt, deren Ausführung nur weitgehend lokale Auswirkungen auf das übergeordnete System aufweist. Die genannten Voraussetzungen erlauben somit keine allgemeine Übertragung der erzielten Ergebnisse auf beliebige Anwendungsbereiche mit komplex zusammengesetzter Funktionalität. Dennoch können eine Reihe von bedeutsamen Aufgaben wirkungsvoll unterstützt werden, wie durch den exemplarisch untersuchten Anwendungsbereich belegt wird.

Der Vergleich mit bekannten Ansätzen erlaubt eine weitergehende Einordnung der erarbeiteten Lösung sowie eine umfassende Beurteilung der erreichten Innovation. So können gegenüber vollständig manueller Konstruktion, etwa bei Verwendung von Skriptsprachen, funktionale Prototypen wesentlich schneller und mit geringerem Aufwand aus vorhandenen Komponenten zusammengestellt werden. Andere Ansätze der generativen Softwareentwicklung erlauben zwar ebenfalls eine weitgehend automatisierte Konstruktion ausführbarer Systeme, jedoch berücksichtigen sie unabhängig entwickelte Komponenten nur ungenügend und stellen aufgrund komplexer Modelle und Verfahren häufig hohe Ansprüche an Qualifikation und Erfahrung des Entwicklers. Ausgewählte Konzepte aus agentenbasierten Ansätzen können hingegen vorteilhaft mit dem vorgestellten Framework kombiniert werden.

Darüber hinaus weisen dessen konzeptuelle Bestandteile auch Bezüge zu partiell vergleichbaren Lösungen auf. So zeichnet sich das vorgeschlagene Metamodell einer Ontologie gegenüber bestehenden Modellen oder objektorientierten Typsystemen durch geringere Komplexität bzw. höhere Flexibilität aus. Das zentrale Konzept eines Component Use Case entspricht auf anwendungsbezogener Ebene einem bewährten Ansatz der Anforderungsanalyse. Seine integrale Verbindung mit technischer Realisierung durch geeignet spezifizierte Interaktionen legt den Vergleich mit *Business Objects* als Mittel der Systementwicklung nahe, obwohl derartige Ansätze eine weitgehend automatisierte Konstruktion funktionaler Prototypen nicht unterstützen.

Dies trifft ebenfalls auf formale Ansätze zur Spezifikation von Software-Architektur zu, jedoch beinhalten aktuelle Arbeiten das Konzept einer Interaktion als eigenständiges Element der Modellierung. Sie stützen somit das erarbeitete Modell eines komponentenbezogenen Anwendungsfalls und verdeutlichen eine mögliche Richtung zukünftiger Weiterentwicklung. Das evolutionäre Vorgehen zur Optimierung von Prototyp-Varianten repräsentiert eine oftmals erfolgreiche Heuristik zur Lösung kombinatorischer Probleme. Im Unterschied zu bekannten Ansätzen des *Genetic Programming* wird hierbei Struktur und Zusammenspiel des gesuchten Systems bereits in groben Zügen vorgegeben, so daß eine deutlich höhere Qualität der generierten Ergebnisse zu erwarten ist.

Mit Hilfe der oben getroffenen Feststellungen läßt sich abschließend eine Charakterisierung und Bewertung des vorliegenden Ansatzes angeben:

- Das Framework repräsentiert eine pragmatische Lösung der Problemstellung, deren Modelle und Verfahren keine besonderen Anforderungen an Qualifikation und Erfahrung des Benutzers voraussetzen.

- Die gewählte Konzeption erlaubt eine effektive, skalierbare und komponentenorientierte Umsetzung für alle gegenwärtig bedeutsamen technischen Plattformen.

- Die vorgeschlagenen Lösungen in den jeweiligen Bereichen des Frameworks berücksichtigen bekannte und bewährte Ansätze aus Forschung und Praxis der Softwareentwicklung.

- Die klare Strukturierung des Frameworks erleichtert dessen Weiterentwicklung und praktische Umsetzung sowie den Einsatz ausgewählter Bestandteile für andere Aufgaben.

Demgegenüber ergeben sich folgende Nachteile und Einschränkungen:

- Die vorgestellte Lösung kann eine vollständig automatisierte und fehlerfreie Konstruktion funktionaler Prototypen nicht gewährleisten.

- Die vorausgesetzten Annahmen des Ansatzes schränken dessen unterstützte Anwendungsbereiche merklich ein.

- Die eingesetzten Modelle und Verfahren wurden nicht umfassend empirisch überprüft.

Dennoch repräsentiert die vorliegende Arbeit einen innovativen und durchaus vielversprechenden Ansatz für komponentenbasiertes Rapid Prototyping, welcher den besonderen Anforderungen der untersuchten Aufgabenstellung weitgehend gerecht wird. Seine zahlreichen, vielfältigen Erweiterungen verdeutlichen zudem das zukünftige Potential der erarbeiteten Konzeption.

ANHANG

A. TECHNISCHE REALISIERUNG

In diesem Teil der Arbeit werden ausgewählte technische Details der prototypischen Referenz-Implementierung vorgestellt (siehe Kapitel 4). Diese sind für ein grundlegendes Verständnis der vorgeschlagenen Lösung nicht wesentlich, verdeutlichen aber die praktische Umsetzung und Anwendung der erarbeiteten Ergebnisse. Dies betrifft insbesondere die im folgenden aufgeführten Grammatiken zur Beschreibung von Ontologie, Funktionaler Spezifikation und komponentenbezogenen Anwendungsfällen. Schließlich sind die erstellten Modelle und Beschreibungen vom Benutzer des Frameworks als zentrale, für den weiteren Verlauf maßgebliche Informationen bereitzustellen (vgl. Kapitel 3). Demgegenüber sind die intern eingesetzten Algorithmen und Datenstrukturen von untergeordneter Bedeutung und werden daher an dieser Stelle nicht berücksichtigt. Weiterführende Informationen über diesen Bereich der Implementierung sowie deren vollständigen Java-Code und zahlreiche Anwendungsbeispiele sind unter [Vil01b] zugänglich.

A.1 Beschreibung der Ontologie

Ein konkretes Modell des jeweiligen Anwendungsbereichs gemäß der in Abschnitt 3.3 erläuterten Ontologie wird zweckmäßig in einer textuellen Notation angegeben. Die hierfür eingesetzte Syntax wird im folgenden durch eine entsprechende BNF-Grammatik beschrieben, aus der sich Teile der Implementierung, wie beispielsweise ein Parser, automatisch generieren lassen. Ihre Regeln entsprechen im wesentlichen einer unmittelbaren Umsetzung des in Abbildung 3.5 dargestellten Klassendiagramms. Aufgrund der Verwendung von Vorarbeiten des Tambis-Projektes [BBB+99] beinhaltet die Grammatik jedoch auch redundante oder für die Zwecke dieser Arbeit nicht benötigte Elemente.

Jede Domäne der Ontologie wird in einer eigenen Datei beschrieben, deren Name die betreffende Domäne eindeutig identifiziert. Somit können innerhalb einer Domäne (`Domain`) ihre beinhalteten Konzepte (`Concept`) und deren Beziehungen über entsprechende Aussagen (`ConceptStatement`) definiert werden. Zusätzlich besteht die Möglichkeit, weitere Domänen an Hand ihres Bezeichners (`DomainId`) zu importieren (`ImportStatement`). Sämtli-

che Aussagen (`ImportStatement`) können durch nachgestellte Kommentare (`Comment`) ergänzt werden[1].

```
<Domain> ::= <Comment> <Statement>+

<Statement> ::= (<ImportStatement> | <ConceptStatement>)
                [<Comment>]

<ImportStatement> ::= 'uses' <DomainId> '.'

<DomainId> ::= <Identifier>

<Comment> ::= '"' <Text> '"'
--------------------------------------------------------
Beispiel:
  "Domain SequenceAnalysis"
  uses Chemistry.
  uses Biochemistry.
  uses AbstractStructure. "sequence, tree, etc."
```

Eine Aussage über Konzepte (`Concept`) definiert diese an Hand ihres eindeutigen Bezeichners und setzt sie mit anderen Konzepten in Beziehung. Optional können Kardinalität (`Cardinality`) und Domäne eines Konzepts angegeben werden, falls dessen Namensgebung es erfordert.

```
<ConceptStatement> ::= <NewGeneralisation> '.' |
                       <AddGeneralisation> '.' |
                       <Aggregation> '.'       |
                       <Alternative> '.'       |
                       <Interpretation> '.'    |
                       <Relation> '.'          |
                       <NameDef> '.'           |
                       <Concept> '.'

<Concept> ::= [<Cardinality>] [<DomainId>'/']<Identifier>

<Cardinality> ::= <Number>|'0..1'|'0..*'|'1..*'
```

[1] Elementare Definitionen für Bezeichner, Zahlen oder Fließtext werden vorausgesetzt und nicht näher erläutert.

Die Generalisierung zwischen Konzepten kann neu eingeführt (`NewGeneralisation`) oder in Relation zu einem bereits definierten Konzept etabliert werden (`AddGeneralisation`). Hierbei läßt sich auch eine Liste spezieller Konzepte (`ConceptList`) angeben. Die Richtung der Generalisierung wird durch die Bezeichnung des verknüpfenden Bestandteils der Aussage (`NewGenKey, AddGenKey`) festgelegt.

```
<NewGeneralisation> ::=
  <Concept> <NewGenKey> <Concept>                |
  <Concept> <NewGenKey> <ConceptList>            |
  <Characterization> <NewGenKey> <Concept>       |
  <Characterization> <NewGenKey> <ConceptList>

<NewGenKey> ::=
  'newSub'             |
  'isGeneralisationOf' |
  'specializes'

<AddGeneralisation> ::=
  <Concept> <AddGenKey> <Concept>                |
  <Concept> <AddGenKey> <ConceptList>            |
  <Characterization> <AddGenKey> <Concept>       |
  <Characterization> <AddGenKey> <ConceptList>

<AddGenKey> ::=
  'addSub' |
  'isGeneralisationOf' |
  'specializes'

<ConceptList> ::= '[' (<Concept>)+ ']'
--------------------------------------------------------
Beispiel:
  Biopolymer newSub [Protein NucleicAcid].
  Biopolymer addSub Polysaccharide.
  NucleicAcid isGeneralisationOf DNA.
  RNA specializes NucleicAcid.
```

In vergleichbarer Weise wird die Aggregationsbeziehung (`Aggregation`) zwischen Konzepten angegeben. Wiederum bestimmt die Namensgebung des verknüpfenden Elements (`AggregationKey`) über die Richtung der Aggregation.

```
<Aggregation> ::=
  <Concept> <AggregationKey> <Concept>              |
  <Concept> <AggregationKey> <ConceptList>          |
  <Characterization> <AggregationKey> <Concept>     |
  <Characterization> <AggregationKey> <ConceptList>

<AggregationKey>  ::=
  'isComponentOf' |
  'hasComponent'
--------------------------------------------------------
Beispiel:
  NucleicAcid hasComponent 1..* Nucleotide
  Nucleotide hasComponent [1 Phosphate, 1 Sugar, 1 Base].
```

Auch die alternative Auswahl zwischen Konzepten (`Alternative`) wird in entsprechender Weise definiert.

```
<Alternative> ::=
  <Concept> <AlternativeKey> <ConceptList>          |
  <Characterization> <AlternativeKey> <ConceptList>

<AlternativeKey>  ::= 'isOneOf'
--------------------------------------------------------
Beispiel:
  ElectricalCharge isOneOf [NegativeCharge, PositiveCharge].
```

Bei Angabe einer Interpretation (`Interpretation`) können zugehörige Subinterpretationen in geschweiften Klammen spezifiziert werden.

```
<Interpretation> ::=
  <Concept> 'interpretedAs' <Concept>
  ['{' <Interpretation> (',' <Interpretation>)* '}']
--------------------------------------------------------
Beispiel:
  NucleicAcid interpretedAs AbstractStructure/Sequence
       {Nucleotide interpretedAs AbstractStructure/Element}.
```

Spezifische Beziehungen zwischen Konzepten (`Relation`) werden durch ihren Bezeichner identifiziert. Der optional eingeführte qualifizierende Bestandteil (`Qualifier`) erlaubt die Angabe von Einschränkungen, die jedoch gegenwärtig von der Referenz-Implementierung nicht berücksichtigt werden.

```
<Relation> ::=
  <Concept> [<Qualifier>] <Identifier> <Concept>                |
  <Concept> [<Qualifier>] <Identifier> <Characterization>       |
  <Concept> [<Qualifier>] <Identifier> <ConceptList>            |
  <ConceptList> [<Qualifier>] <Identifier> <Concept>            |
  <ConceptList> [<Qualifier>] <Identifier> <Characterization>   |
  <ConceptList> [<Qualifier>] <Identifier> <ConceptList>        |
  <Characterization> [<Qualifier>] <Identifier> <Concept>       |
  <Characterization> [<Qualifier>] <Identifier> <ConceptList>

<Qualifier> ::=
  'grammaticallyAndSensibly' |
  'grammatically'            |
  'sensibly'                 |
  'necessarily'
--------------------------------------------------------
Beispiel:
  Alignment hasLength Common/Length.
  Alignment necessarily hasName 1 Common/Name.
```

Die Charakterisierung von Konzepten an Hand ausgewählter Beziehungen (`Characterization`) erlaubt die Einführung ausgezeichneter Begriffe bzw. neuer Konzepte (`NameDef`). Auf diese Weise können charakteristische Zusammenhänge im jeweiligen Anwendungsbereich sehr kompakt beschrieben werden. Allerdings werden diese Informationen durch die gegenwärtige Implementierung nicht berücksichtigt.

```
<NameDef> ::= <Characterization> 'name' <Concept>

<Characterization> ::=
  '(' <Concept> 'which' <Identifier> <Concept> ')'            |
  '(' <Concept> 'which' <Identifier> <ConceptList> ')'        |
  '(' <Concept> 'which' <Identifier> <Characterization> ')'
--------------------------------------------------------
Beispiel:
  (Protein which catalyses Reaction) name Enzyme.
```

Insgesamt ergibt sich somit bei disziplinierter Anwendung der aufgeführten Grammatik eine übersichtliche und leicht verständliche Beschreibung der Ontologie. Allerdings ist für die Bearbeitung umfangreicher Modelle und Analyse abgeleiteter Verhältnisse ein entsprechend leistungsfähiges, visuelles Werkzeug überaus hilfreich, wie in Kapitel 4 erläutert wird.

A.2 Beschreibung der Funktionalen Spezifikation

Die Beschreibung der gewünschten Funktionalität im Rahmen einer Funktionalen Spezifikation wird ebenfalls zweckmäßig in einer einfachen textuellen Notation angegeben (siehe Abschnitt 3.5). Die hierfür eingesetzte Grammatik entspricht unmittelbar dem in Abbildung 3.13 dargestellten Klassenmodell. Somit wird eine intuitive Beschreibung der funktionalen Anforderungen ermöglicht, die zukünftig durch geeignete operative Konstrukte ergänzt werden kann (vgl. Abschnitt 5.2.2).

Eine gegebene Funktionale Spezifikation (`FunctionalSpecification`) beinhaltet den Namen des zu erstellenden Prototypen (`PrototypeDeclaration`) sowie eine nicht-leere Folge von Funktionalen Aussagen (`FunctionalStatement`).

```
<FunctionalSpecification> ::=
  <PrototypeDeclaration> (<FunctionalStatement>)+

<PrototypeDeclaration> ::= 'Prototype' <Identifier> ';'
---------------------------------------------------------
Beispiel:
  Prototype SequenceTool;
  ...
```

Jede Funktionale Aussage gruppiert eine Folge von Anwendungsfällen (`UseCase`), die über einen gemeinsamen Auslöser (`Trigger`) kontrolliert werden. Hierbei wird zwischen unmittelbaren und deklarativen Anwendungsfällen (`ImmediateUseCase` bzw. `DeclarativeUseCase`) unterschieden. Letztere deklarieren eine Referenz (`Reference`) auf Ergebnisse ihrer Ausführung, die von Ersteren üblicherweise als Parameter genutzt werden können.

```
<FunctionalStatement> ::=
  [<Trigger>] <UseCase> (',' <UseCase>)* ';'

<UseCase> ::= <ImmediateUseCase> | <DeclarativeUseCase>

<DeclarativeUseCase> ::= <Manipulation> <Concept> <Reference>

<ImmediateUseCase> ::= <Manipulation> <Reference>

<Reference> ::= '"' <Identifier> '"'
---------------------------------------------------------
```

```
Beispiel:
  Search 1..* DNA "result";
  Display "result";
```

Die Ausführung einer Funktionalen Aussage wird entweder explizit vom Benutzer angestoßen (`UserCommand`) oder durch ein eingetretenes Ereignis (`EventCondition`) ausgelöst. Ein solches Ereignis wird als eigener Anwendungsfall angegeben.

```
<Trigger> ::= (<EventCondition> | <UserCommand>) ':'

<EventCondition> ::= 'On' <UseCase>

<UserCommand> ::= 'On Command'
--------------------------------------------------------
Beispiel:
  On Select DNA "selected": ...
  On Display "phylogeny": ...
```

A.3 Beschreibung eines Component Use Case

Die Erstellung und Bearbeitung komponentenbezogener Anwendungsfälle erfolgt durch ein eigenes, durchaus komfortables Werkzeug der Referenz-Implementierung (vgl. Abbildung 4.2). Somit kann die erzeugte XML-basierte CUC-Beschreibung unmittelbar dem ausführbaren Format der betreffenden Komponente beigefügt werden. Dennoch wird im folgenden die Grammatik einer solchen Beschreibung in Form ihrer zugehörigen *Document Type Definition* (DTD) [Hol00] vorgestellt. Dies erlaubt eine Einbindung weiterer Werkzeuge, beispielsweise eines Editors oder Parsers, mit deren Hilfe weiterführende Funktionalität implementiert werden kann. Aufgrund einer nahezu schematischen Umsetzung der entsprechenden Klassendiagramme in Abbildung 3.7, 3.8 und 3.9 wird auf eine ausführliche Erläuterung der DTD verzichtet.

```
<?xml encoding="UTF-8"?>
<!-- Version: 1.3, File: component_description.dtd,
     Author: Alexander Vilbig -->

<!ELEMENT ComponentUseCaseDescription
          ((Type | Interface)*, Component)>
  <!ATTLIST ComponentUseCaseDescription
```

```
  author  CDATA #IMPLIED
  version CDATA #IMPLIED
  name    CDATA #IMPLIED>

<!ELEMENT Component
          (AbstractState+, Component*, Interface*, UseCase*)>
  <!ATTLIST Component
  id                 ID     #REQUIRED
  name               CDATA  #REQUIRED
  initialState       IDREF  #REQUIRED
  requiredInterfaces IDREFS #IMPLIED>

<!ELEMENT Type EMPTY>
  <!ATTLIST Type
  id   ID    #REQUIRED
  name CDATA #REQUIRED>

<!ELEMENT AbstractState EMPTY>
  <!ATTLIST AbstractState
  id   ID    #REQUIRED
  name CDATA #REQUIRED>

<!ELEMENT Interface (Operation*)>
  <!ATTLIST Interface
  id   ID    #REQUIRED
  name CDATA #REQUIRED>

<!ELEMENT Operation (Parameter*)>
  <!ATTLIST Operation
  id         ID    #REQUIRED
  name       CDATA #REQUIRED
  returnType IDREF #REQUIRED>

<!ELEMENT Parameter EMPTY>
  <!ATTLIST Parameter
  id   ID    #REQUIRED
  name CDATA #REQUIRED
  type IDREF #REQUIRED>

<!ELEMENT UseCase (UseCase*, Variable*, Role*, MMConcept*,
 Interaction*, Manipulation?, Hint*)>
```

```
 <!ATTLIST UseCase
  id   ID    #REQUIRED
  name CDATA #REQUIRED>

<!ELEMENT Hint EMPTY>
  <!ATTLIST Hint
  tag   CDATA #REQUIRED
  value CDATA #REQUIRED>

<!ELEMENT Manipulation EMPTY>
  <!ATTLIST Manipulation
  id                   ID    #REQUIRED
  name                 CDATA #REQUIRED
  cardinality          CDATA #IMPLIED
  manipulatedMMConcept IDREF #IMPLIED>

<!ELEMENT MMConcept EMPTY>
  <!ATTLIST MMConcept
  id   ID    #REQUIRED
  name CDATA #REQUIRED>

<!ELEMENT Interaction (Variable*, OperationSequence?,
 EventPattern?)>
  <!ATTLIST Interaction
  id            ID     #REQUIRED
  name          CDATA  #REQUIRED
  involvedRoles IDREFS #IMPLIED>

<!ELEMENT Role EMPTY>
  <!ATTLIST Role
  id                      ID    #REQUIRED
  name                    CDATA #REQUIRED
  correspondingInterface IDREF #IMPLIED>

<!ELEMENT OperationSequence (OperationCall)+>
  <!ATTLIST OperationSequence
  id             ID     #REQUIRED
  requiredStates IDREFS #IMPLIED
  resultingState IDREF  #REQUIRED>

<!ELEMENT OperationCall (Value?, ParameterSubstitution*)>
```

```
  <!ATTLIST OperationCall
  seqNumber       CDATA #REQUIRED
  caller          IDREF #REQUIRED
  callee          IDREF #REQUIRED
  calledOperation IDREF #REQUIRED>

<!ELEMENT ParameterSubstitution (Value)>
  <!ATTLIST ParameterSubstitution
  substitutedParameter IDREF #REQUIRED>

<!ELEMENT Value (Expression?)>
  <!ATTLIST Value
  id          ID    #REQUIRED
  cardinality CDATA #IMPLIED
  variable    IDREF #IMPLIED
  expression  IDREF #IMPLIED
  role        IDREF #IMPLIED
  MMConcept   IDREF #IMPLIED>

<!ELEMENT Variable EMPTY>
  <!ATTLIST Variable
  id   ID    #REQUIRED
  name CDATA #REQUIRED
  type IDREF #REQUIRED>

<!ELEMENT Expression EMPTY>
  <!ATTLIST Expression
  id    ID    #REQUIRED
  value CDATA #REQUIRED>

<!ELEMENT EventPattern (Event, OperationCall,
 OperationCall, OperationCall)>
  <!ATTLIST EventPattern
  id ID #REQUIRED>

<!ELEMENT Event EMPTY>
  <!ATTLIST Event
  name    CDATA #REQUIRED
  subject IDREF #REQUIRED
  cause   IDREF #REQUIRED>
```

LITERATURVERZEICHNIS

[AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connections. *ACM Transactions on Software Engineering and Methodology*, 1997.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language - Third Edition*. Addison-Wesley, Reading, MA, 2000.

[Ame87] P. America. Inheritance and Subtyping in a Parallel Object-oriented Language. In *ECOOP '87: European Conference on Object-oriented Programming*, Lecture Notes in Computer Science 276, pages 234–242. Springer Verlag, 1987.

[BBB+99] Patricia Baker, Sean Bechhofer, Andy Brass, Carol Goble, Norman Paton, and Robert Stevens. An Ontology For Bioinformatics Applications. *Bioinformatics*, 15(6):510–520, 1999.

[BBC00] Ray Brown, Wade Baron, and William D. Chadwick. *Designing Solutions with COM+ Technologies*. Microsoft Press, 2000.

[BBR+00] Klaus Bergner, Manfred Broy, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A Formal Model for Componentware. In Gary T. Leavens and Murali Sitamaran, editors, *Foundations of Component-Based Systems*, chapter 9, pages 189–210. Cambridge University Press, January 2000.

[BBSDS97] H. Bowman, C. Briscoe-Smith, J. Derrick, and B. Strulo. On Behavioural Subtyping in LOTOS. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS97)*, pages 21–36, Canterbury, UK, 1997. Chapman and Hall, London.

[BCRW00] Don Batory, Gang Chen, Eric Robertson, and Tao Wang. Design Wizards and Visual Programming Environments for GenVoca Generators. *IEEE Transactions on Software Engineering*, 26(5), Mai 2000.

[BDD+92] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The Design of Distributed Systems - An Introduction to FOCUS. Technical Report TUM-I9202, Technische Univerität München, 1992.

[BG91] L. Blaine and A. Goldberg. DTRE - A Semiautomatic Transformation System. In *Constructing Programs from Specifications*. Elsevier Science Publishers, 1991.

[BG96] Don Batory and Bart J. Geraci. Validating Component Compositions in Software System Generators. International Conference on Software Reuse, 1996.

[BJR98] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley Longman, Reading, MA, 1998.

[BJR+01] Klaus Bergner, Carsten Jacobi, Andreas Rausch, Marc Sihling, and Alexander Vilbig. Make-or-Buy von Softwarekomponenten. *OBJEKTspektrum*, 1, January 2001.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture*. John Wiley & Sons, 1996.

[BO92] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM TOSEM, October 1992.

[Boe88] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[Bor95] A. Borgida. Description Logics in Data Management. *IEEE Transaction on Knowledge and Data Engineering*, 7:671–682, 1995.

[Bor01a] Borland Corporation. JBuilder Homepage. `http://www.inprise.com/jbuilder/`, 2001.

[Bor01b] Borland Corporation. Visibroker Homepage. `http://www.inprise.com/visibroker/`, 2001.

[Bro01] Gerald Brose. JacORB Homepage. `http://jacorb.inf.fu-berlin.de/`, 2001.

[BRSV98a] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A Componentware Development Methodology based on Process Patterns. In *PLOP 98, Proceedings of the 5th Annual Conference on the Pattern Languages of Programs*, August 1998.

[BRSV98b] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. An Integrated View On Componentware - Concepts, Description Techniques, and Development Process. In Roger Lee, editor, *IASTED 98, Proceedings of IASTED Conference on Software Engineering*. ACTA Press, October 1998.

[BRSV99a] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. Componentware - Methodology and Process. In *CBSE 99, Proceedings of the International Workshop on Component-Based Software Engineering*, May 1999.

[BRSV99b] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. Structuring and Refinement of Class Diagrams. In *HICSS 32, Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, January 1999.

[BRSV00a] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. Adaptation Strategies in Componentware. In *ASWEC, Proceedings of the 2000 Australian Software Engineering Conference*, IEEE Computer Society, April 2000.

[BRSV00b] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. Putting the Parts Together - Concepts, Description Techniques, and Development Process for Componentware. In *HICSS 33, Proceedings of the Thirty-Third Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, January 2000.

[CHOT99] Siobhan Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code. In *Confernece on Object-Oriented Programming, Systems, Languages, and Applications, November 1-5, 1999, Denver, Colerado, USA*, pages 325–339, 1999.

[CM96] A. Chavez and P. Maes. Kasbah: An Agent Marketplace for Buying and Selling Goods. In *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 75–90, April 1996.

[Con01] Concurrent Versions System. CVS Homepage. `http://www.cvshome.org/`, 2001.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Deu89] P. Deutsch. Design Reuse and Frameworks in the Smalltalk-80 System. *Software Reusability*, 2, 1989.

[DK97] A. Van Duersen and P. Klint. Little Languages: Little Maintenance? First ACM SIGPLAN Workshop on Domain Specific Languages, 1997.

[DMN68] O-J. Dahl, B. Myhrhaug, and K. Nygaard. SIMULA 67 Common Base Language. Technical Report, Norwegian Computer Center, 1968.

[DW98] Desmond D'Souza and Alan C. Wills. *Objects, Components, and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, Reading, MA, 1998.

[DW99] Wolfgang Dröschel and Manuela Wiemers. *Das V-Modell 97*. Oldenbourg, München, 1999.

[ES98] Peter Eeles and Oliver Sims. *Building Business Objects*. John Wiley & Sons, 1998.

[FG98] M. S. Fox and M. Gruninger. Enterprise Modelling. *AI Magazine*, pages 109–121, Juni 1998.

[Fin79] N. V. Findler, editor. *Associative Networks: Representation and Use of Knowledge by Computer*. Academic Press, New York, 1979.

[Fuj01] Fujitsu Inc. Fujitsu i-Flow. `http://www.i-flow.com`, 2001.

[FW95] J. Fuchs and J. Wheadon. Prospective Applications of Ontologies for Future Space Missions. In *The Impact of Ontologies on Reuse, Interoperability, and Distributed Processing*, Unicom Seminars, pages 83–96, 1995.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch or Why it's hard to build systems out of existing parts. In *Proceedings of the Seventeenth International Conference on Software Engineering*, Seattle, WA, April 1995.

[GF92] M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.

[GG95] N. Guarino and P. Giaretta. Ontologies and Knowledge Bases - Towards a Terminological Clarification. In N. J. Mars, editor, *Towards very large Knowledge Bases - Knowledge Building and Knowledge Sharing 1995*, IOS Press, pages 25–32. IOS Press, 1995.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[Gie01] Holger Giese. *Object-Oriented Design and Architecture of Distributed Systems*. Dissertation, Westfälischen Wilhems-Universität, Münster, 2001.

[Gol89] David Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[Gom94] H. Gomaa. A Prototype Domain Modeling Environment for Reusable Software Architectures. ICSR, 1994.

[GP95] D. Garlan and D. Perry. Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21, April 1995.

[Gru93] Thomas Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Aquisition*, 5:199–220, 1993.

[GU96] Michael Gruninger and Mike Uschold. Ontologies: Principles, Methods and Applications. *Knowledge Engineering Review*, 11(2), Juni 1996.

[GV00] Holger Giese and Alexander Vilbig. Towards Aspect-oriented Design and Architecture. In *15th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications October 15-19, 2000, Minneapolis, Minnesota, USA. Workshop: Advanced Separation of Concerns, Monday, 16 October, 2000*, 2000.

[Haj88] B. Hajek. Cooling Schedules for Optimal Annealing. *Mathematics of Operations Research*, 13:311–329, 1988.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 3(8):231–274, 1987.

[Hen01] Joachim Henkel. Bezahlen auf Draht. *Magazin für Computer Technik*, 6:270281, 2001.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[Hol00] Steven Holzner. *Inside XML*. New Riders Publishing, 2000.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Series in Computer Science. Addison-Wesley, Reading, MA, 1979.

[IBM01a] International Business Machines Corporation. SanFrancisco Homepage. `http://www-4.ibm.com/software/ad/sanfrancisco/`, 2001.

[IBM01b] International Business Machines Corporation. VisualAge for Java Homepage. `http://www-4.ibm.com/software/ad/vajava/`, 2001.

[Int01] Intrinsyc Software. J-Integra Homepage. `http://www.linar.com/jintegra/doc/`, 2001.

[ION01] IONA Technologies Inc. Orbix Homepage. `http://www.iona.com/products/orbhome.htm`, 2001.

[ISO89] ISO. Lotos — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, International Organization for Standardization (ISO) — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

[Jab95] S. Jablonski. *Workflow Management Systeme: Modellierung und Architektur*. Thompson Publishing, Bonn, 1995.

[Jac92] Ivar Jacobson. *Object-Oriented Software Engineering: A Use-Case-Driven Approach*. Addison-Wesley, Reading, MA, 1992.

[JWW+95] M. Jones, J. Wheadon, D. Whitgift, M. Niezatte, R. Timmermans, I. Rodriguez, and R. Romero. An Agent-based Approach to Spacecraft Mission Operations. In N. J. Mars, editor, *Towards very large Knowledge Bases - Knowledge Building and Knowledge Sharing 1995*, IOS Press, pages 259–269. IOS Press, 1995.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[Kie96] R. Kieburtz. A Software Engineering Experiment in Software Component Generation. Internationel Conference on Software Engineering, 1996.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS. Springer Verlag, 1997.

[Kor01] Thierry Kormann. The Koala User Interface Language. `http://www-sop.inria.fr/koala/kuil/`, 2001.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.

[Kru98] Philippe Kruchten. *The Rational Unified Process - an Introduction.* Addison Wesley, 1998.

[Krü00] Ingolf Krüger. *Distributed System Design with Message Sequence Charts.* Dissertation, Technische Universität München, 2000.

[LAK+95] D. C. Luckham, L. M. Augustin, J. J. Kenny, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[Lan00] William B. Langdon. Quadratic Bloat in Genetic Programming. In *Proceedings of the 2000 Genetic and Evolutionary Computation Conference (GECCO-2000)*, July 2000.

[Lew90] Benjamin Lewin. *Genes IV.* Oxford University Press, New York, 1990.

[LG91] Wen-Hsiung Li and Dan Graur. *Fundamentals of Molecular Evolution.* Sinauer Associates, Sunderland, MS, 1991.

[Lie95] H. Lieberman. Letizia: An Agent that Assists Web Browsing. In *Proceedings of IJCAI*, AAAI Press, 1995.

[LQ95] William B. Langdon and Adil Qureshi. Genetic Programming – Computers using Natural Selectionto generate programs. Research Note RN/95/76, University College London, Gower Street, London WC1E 6BT, UK, October 1995.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. 2nd edition.

[Mic98] Microsoft Corporation. *Microsoft Visual Basic 6.0: Programmer's Guide*. Microsoft Press, 1998.

[Mic01a] Microsoft Corporation. BizTalk Homepage. `http://www.biztalk.org/home/default.asp`, 2001.

[Mic01b] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, 2001.

[Mic01c] Microsoft Corporation. Microsoft Message Queuing Homepage. `http://www.microsoft.com/msmq/`, 2001.

[Mic01d] Microsoft Corporation. Microsoft Office - Visio Homepage. `http://www.microsoft.com/office/visio/`, 2001.

[Mic01e] Microsoft Corporation. *The Microsoft .NET Framework*. Microsoft Press, 2001.

[MMMP90] O. L. Madsen, B. Magnusson, and B. Mller-Pedersen. Strong Typing of Object-oriented Programming Revisited. In *OOPSLA/ECOOP '90 Conference Proceedings*, ACM Sigplan Not. 25, pages 397–406, October 1990.

[MW91] Tim Maude and Graham Willis. *Rapid Prototyping - The Management of Software Risk*. Pitman Publishing, London, 1991.

[NAG01] The Numerical Algorithms Group Ltd. IRIS Explorer Homepage. `http://www.nag.com/Welcome_IEC.html`, 2001.

[Nie93] Oscar Nierstrasz. Regular Types for active Objects. In *Proceedings OOPSLA'93*, volume 28 of *ACM SIGPLAN Notices*, pages 1–15, October 1993.

[Nin94] J. Q. Ning. An Architecture-Driven, Business-Specific, and Component-Based Approach to Software Engineering. ICSR, 1994.

[Nwa96] Hyacinth S. Nwana. Software Agents: An Overview. Knowledge Engineering Review, 1996.

[OMG99] Object Management Group. OMG Unified Modeling Language Specification, Version 1.3, Chapter 7 'Object Constraint Language Specification', 1999.

[OMG00] Object Management Group. Meta Object Facility Specification. ftp://ftp.omg.org/pub/docs/formal/00-04-03.pdf, 2000.

[OMG01a] Object Management Group. Event Service Specification. ftp://ftp.omg.org/pub/docs/formal/01-03-01.pdf, 2001.

[OMG01b] Object Management Group. Object Management Group Homepage. http://www.omg.org, 2001.

[OMG01c] Object Management Group. OMG Business Object Domain Task Force RFI-1. ftp://ftp.omg.org/pub/docs/bom/97-06-02.pdf, 2001.

[OMG01d] Object Management Group. OMG XML Metadata Interchange (XMI) Specification. ftp://ftp.omg.org/pub/docs/formal/00-11-02.pdf, 2001.

[PD87] R. Prieto-Diaz. Classifying Software for Reusability. *IEEE Software*, 4:6–16, 1987.

[Pop98] Alan Pope. *The Corba Reference Guide : Understanding the Common Object Request Broker Architecture*. Addison-Wesley, Reading, MA, 1998.

[Pun97] Franz Puntigam. Coordination Requirements Expressed in Types for Active Objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97- Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland*, number 1241 in Lecture Notes in Computer Science, pages 367–388. Springer Verlag, June 1997.

[Rat01a] Rational Software Corporation. Rational ClearCase Homepage. http://www.rational.com/products/clearcase/index.jsp, 2001.

[Rat01b] Rational Software Corporation. Rational Rose Homepage. http://www.rational.com/products/rose/index.jsp, 2001.

[RBG+97] A.L. Rector, S. Bechhofer, C.A. Goble, I. Horrocks, W.A. Nowlan, and W.D. Solomon. The GALEN modelling language for medical terminology. *AI in Medicine*, 9:139171, 1997.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, 1998.

[Rog96] Dale Rogerson. *Inside COM.* Microsoft Press, 1996.

[Rog01] Dmitriy Rogatkin. xBox Homepage. `http://drogatkin.openestate.net/xbox.html`, 2001.

[Sak89] M. Sakkinen. Disciplined Inheritance. In *ECOOP '89: European Conference on Object-oriented Programming*, The British Computer Society Workshop Series, pages 39–56. Cambridge University Press, 1989.

[SEI90] Software Engineering Institute. Workshop on Domain-Specific Software Architectures, 1990.

[SG95] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. *Lecture Notes in Computer Science*, 1000, April 1995.

[SG96] Mary Shaw and Davis Garlan. *Software Architecture: Perspectives on an emerging Discipline.* Prentice Hall, 1996.

[Som87] Ian Sommerville. *Software-Engineering.* Addison-Wesley, Reading, MA, 1987.

[Sta01] Stanford University. Stanford KSL Network Services. `http://www-ksl-svc.stanford.edu:5915/`, 2001.

[Str88] Lubert Stryer. *Biochemistry.* W. H. Freeman And Company, New York, 1988.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1991.

[Sun98] Sun Microsystems. JavaBeans Bridge for ActiveX. `http://java.sun.com/products/javabeans/software/bridge/`, 1998.

[Sun00a] Sun Microsystems. JavaBeans Homepage. `http://java.sun.com/products/javabeans/docs/spec.html`, 2000.

[Sun00b] Sun Microsystems. Javadoc Homepage. `http://java.sun.com/j2se/javadoc/index.html`, 2000.

[SWA+94] Guus Schreiber, Bob Wielinga, Hans Akkermans, Walter van de Velde, and Anjo Anjewierden. The CommonKADS Conceptual Modelling Language. In *Proceedings of 8th European Knowledge Acquistion Workshop EKAW'94*, volume 867 of *LNCS*, Berlin/Heidelberg, September 1994. Springer Verlag.

[Swi01] SwingSoft Ltd. SwingBuilder Homepage. `http://www.swingsoft.com/builder.html`, 2001.

[Szy98] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison Wesley, 1998.

[Tai96] Antero Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3):438–479, 1996.

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996. Third edition.

[Tog01] TogetherSoft Corporation. Together Homepage. `http://www.togethersoft.com/together/matrix.html`, 2001.

[Ver01] Verve Inc. Verve Homepage. `http://www.verve.com`, 2001.

[Vil96] Alexander Vilbig. Berechnung der Gibb'schen Freien Energie Mittels Molekulardynamik-Simulation am Beispiel des Streptavidin-Biotin-Systems. Master's thesis, Technische Universität München, 1996.

[Vil01a] Alexander Vilbig. Project CASA Homepage. `wwwbroy.in.tum.de/~vilbig/casa/`, 2001.

[Vil01b] Alexander Vilbig. Project Prototyper Homepage. `wwwbroy.in.tum.de/~vilbig/prototyper/`, 2001.

[Wat95] Michael S. Waterman. *Introduction to Computational Biology : Maps, Sequences and Genomes*. Chapman & Hall, London, 1995.

[WC01] Michael Wooldridge and Paolo Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In Michael Wooldridge and Paolo Ciancarini, editors, *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in AI*. Springer Verlag, January 2001.

[Wen95] Oliver Wendt. *Naturanaloge Verfahren zur approximativen Lösung Kombinatorischer Optimierungsprobleme*. Dissertation, Johann Wolfgang Goethe-Universität, Frankfurt am Main, 1995.

[WF97] Janice Winsor and Brian Freeman. *Jumping JavaScript*. Prentice Hall, 1997.

[Wie99] Karl E. Wiegers. *Software Requirements*. Microsoft Press, 1999.

[Win89] Patrick H. Winston. *Artificial Intelligence*. Addison-Wesley, Reading, MA, 1989.

[WMC95] Workflow Management Coalition. The Workflow Reference Model. WfMC-TC-1003, 1995.

[WMC99] Workflow Management Coalition. Interface 1: Process Definition Interchange. WfMC-TC-1016-P, 1999.

[WMC01] Workflow Management Coalition. Workflow Management Coalition Homepage. `http://www.wfmc.org`, 2001.

[WOC00] Larry Wall, Jon Orwant, and Tom Christiansen. *Programming Perl, 3rd Edition*. O'Reilly & Associates, 2000.

[Woo75] W. A. Woods. What's in a Link? Foundations for Semantic Networks. In D. G. Bobrow and A. Collins, editors, *Representation and Understanding*. Academic Press, New York, 1975.

[Woo99] Michael Wooldridge. Intelligent Agents. In G. Weiss, editor, *Multiagent Systems*. MIT Press, April 1999.

[XML01] XML Consortium. XML Homepage. `http://www.xml.org/`, 2001.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, September 1997.