Universität Kaiserslautern
Fachbereich Informatik
AG Heterogene Informationssysteme
Prof. Dr.-Ing. Stefan Deßloch

Daten- und Informationsverwaltung für virtuelle Labore im Bereich Verfahrenstechnik Anforderungen und Realisierungsmöglichkeiten

Diplomarbeit

von

Jürgen Göres

Betreuer:

Dipl. -Inform. Michael P. Haustein
Prof. Dr.-Ing. Stefan Deßloch

Juli 2003

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt habe.

Kaiserslautern, den 2. Juli 2003

Jürgen Göres

Inhaltsverzeichnis

Kapitel 1 Einleitung

Die Bedeutung von Simulationen oder virtuellen Experimenten für Forschung und Entwicklung hat mit stetig steigender Leistungsfähigkeit von Rechnersystemen in den letzten Jahren stark zugenommen. Sie ermöglichen nicht nur eine Reduzierung der Kosten, indem in Teilen auf den Aufbau aufwendiger Versuchsanlagen verzichtet werden kann. Vielmehr beschleunigen sie auch die wissenschaftliche Arbeit und erlauben so einen veränderten Umgang mit dem Werkzeug „Experiment“. Während herkömmliche Experimente wegen des Zeit- und Kostenaufwands nur begrenzt und mit Bedacht durchgeführt werden können, erlauben virtuelle Experimente ein schnelleres und direktes Verfolgen von Ideen und Konzepten. Auch potentiell gefährliche Experimente können durchgeführt werden, ohne ein Risiko für Mensch und Natur darzustellen.

1.1 Sonderforschungsbereich „virtuelle Laboratorien“

Bei den bisher verfügbaren Simulationslösungen handelt es sich im Allgemeinen um spezialisierte Software, die für einen relativ engen Anwendungsbereich konzipiert ist. Trotz der oft grundlegenden Unterschiede von Simulationen aus verschiedenen Domänen bestehen doch häufig Gemeinsamkeiten, beispielsweise in den Anforderungen an die Infrastruktur für eine Simulationsumgebung. So stellt jedes virtuelle Experiment im Wesentlichen einen komplexen Prozess dar, dessen Beschreibung interpretiert und ausgeführt werden muss. Als Ergebnis dieser Ausführung entstehen Daten, die in geeigneter Weise für Auswertung und Archivierung gespeichert sowie visualisiert werden müssen.

Bisher kam es daher oft zu Mehrfachentwicklungen solch grundlegender Module. Würde den Entwicklern von virtuellen Laboratorien diese Infrastruktur in Form von generischen Bausteinen zur Verfügung stehen, die an das jeweilige Aufgabengebiet angepasst werden können, ließen sich die Entwicklungszeit und damit auch die Kosten für die Implementierung von virtuellen Labors erheblich reduzieren. Der bei der Deutschen Forschungsgesellschaft beantragte disziplinenübergreifende Sonderforschungsbereich *Virtuelle Laboratorien* hat nun zum Ziel, eine Referenzarchitektur für virtuelle Laboratorien zu entwickeln.

Als Grundlage für die Ermittlung gemeinsamer Anforderungen an ein Laborframework werden in einem ersten Schritt konventionelle virtuelle Labore für drei Anwendungsgebiete entwickelt, die sich durch stark divergierende Eigenschaften auszeichnen. Ziel ist der Ge-

winn von Erfahrungen bezüglich des Einsatzes und der Realisierung von virtuellen Experimenten in der jeweiligen Domäne.

Der Projektbereich A *Siedlungsentwicklung* entsteht in Zusammenarbeit mit dem Fachbereich Raum- und Umweltplanung der Universität Kaiserslautern. In diesem Labor werden virtuelle Experimente zur Siedlungsentwicklung mit Schwerpunkten auf dem Gebiet der Siedlungswasserwirtschaft durchgeführt, für die es häufig keine Entsprechungen in realen Experimenten gibt.

Projektbereich B *Mehrphasensysteme (Verfahrenstechnik)* beschäftigt sich mit den Vorgängen in Flüssig-Flüssig- und Flüssig-Fest-Systemen. Hier besteht die Problematik der Skalierung von Erkenntnissen aus realen Experimenten im Labormaßstab auf den Maßstab von Großanlagen, wie sie beispielsweise in der chemischen Industrie anzutreffen sind.

Projektbereich C Wirkstoffe (Zellbiologie) erforscht die Effektivität von Wirkstoffen. Virtuelle Experimente bieten hier eine Ergänzung von realen Experimenten an lebenden Zellkulturen, mit der die Suche nach geeigneten Wirkstoffen erheblich beschleunigt werden kann.

In Projektbereich D sind die verschiedenen Disziplinen der Informatik und Mathematik zusammengefasst, deren Aufgabe die Entwicklung der generischen Laborplattform ist.

Im Zuge der Einrichtung dieser konventionellen Labore sollen zunächst die Anforderungen der jeweiligen Domänen an ein Labor-Framework ermittelt werden, um Gemeinsamkeiten zu identifizieren und diese als Grundlage für die Entwicklung generischer Bausteine des Labor-Frameworks heranzuziehen.

Als besondere Herausforderung dieser frühen Phase gilt die Überwindung von fachlichen und kommunikativen Barrieren zwischen den beteiligten Fachbereichen. Unterschiedliche Begriffswelten und Vorgehensweisen erfordern die enge Zusammenarbeit von Mitarbeitern des jeweiligen Anwendungsgebiets sowie der Mathematik und Informatik.

1.2 Die Domäne Verfahrenstechnik

Die Ingenieurswissenschaft der Verfahrenstechnik befasst sich nach [VDI] mit der „Erforschung, Entwicklung und technischen Durchführung von Prozessen, in denen Stoffe nach Art, Eigenschaft und Zusammensetzung verändert werden. Ihre Aufgabe ist es, die Umsetzungs- und Veränderungsprozesse zu erforschen, mathematisch zu erfassen oder zumindest zu beschreiben.“

Der Lehrstuhl für Thermische Verfahrenstechnik der Universität Kaiserslautern beschäftigt sich mit Grundlagenforschung zur Entwicklung und Verfeinerung technischer Verfahren, die unter anderem in der Großchemie eingesetzt werden. Ein zentrales Arbeitsgebiet ist die Forschung auf dem Gebiet der Mehrphasensysteme, die zum Ziel hat, die Vorgänge in Flüssig-Flüssig- beziehungsweise Flüssig-Fest-Systemen zu beschreiben.

Eine wichtige Anwendung ist die Flüssig-Flüssig-Extraktion. Dieses Verfahren erlaubt die Entfernung von Wertstoffen aus einer Träger-Wertstoff-Gemisch (Feed) mit Hilfe eines Lösungsmittels (Solvent). Um diesen Vorgang zu optimieren, muss eine möglichst große Oberfläche zwischen Solvent und Feed erreicht werden. Dazu wird das Lösungsmittel durch verschiedene Einbauten und mechanische Verfahren (Rührwerke) in möglichst kleine Tropfen aufgespalten. Ziel der Forschung ist es nun, die Konstruktion und die Betriebsparameter der Apparaturen diesbezüglich zu optimieren und zugleich die mathematischen Modelle, welche Tropfenzerfall, Koaleszenz und Stoffaustausch beschreiben, zu entwickeln und zu verbessern. Dabei besteht eine große Herausforderung darin, die in Versuchen im Labormaßstab gewonnenen Erkenntnisse auf jene Größenordnungen zu übertragen, wie sie im industriellen Einsatz benötigt werden.

1.3 Gliederung

Ziel der vorliegenden Arbeit ist die Erarbeitung der Anforderungen an ein virtuelles Labor für Mehrphasensysteme (Projektbereich B) in enger Kooperation mit Wissenschaftlern aus diesem Bereich. Der Schwerpunkt liegt dabei auf den Anforderungen an die Datenhaltungskomponente (Bereich D4). Die Ergebnisse dieser ersten Phase sind in Kapitel 2 dargestellt. Anschließend sollen in Kapitel 3 die Gemeinsamkeiten dieser Anforderungen mit denen des Labors „Siedlungsentwicklung/Siedlungswasserwirtschaft“ identifiziert werden. Diese werden von Alexander Hilliger von Thile in einer parallel laufenden Diplomarbeit ermittelt. Darauf aufbauend werden die Anforderungen an eine generische Datenhaltungskomponente für virtuelle Labore erarbeitet. Schließlich werden in Kapitel 4 verschiedene Ansätze und Realisierungsmöglichkeiten zur Erfüllung der gemeinsamen und domänenspezifischen Anforderungen untersucht. Dabei sollen insbesondere die verschiedenen Mechanismen, die moderne objektrelationale Datenbanksysteme anbieten, auf ihre Eignung zur Unterstützung dieser Ansätze bewertet werden. Erfolgsversprechende Lösungsansätze werden anschließend in Teilen prototypisch implementiert, die Ergebnisse der Implementierung sind in Kapitel 5 aufgeführt. Abschließend erfolgt in Kapitel 6 eine Zusammenfassung der Ergebnisse sowie ein Ausblick.

Kapitel 2 Anforderungen

Im folgenden Kapitel sollen die Anforderungen an ein virtuelles Labor im Bereich Verfahrenstechnik festgehalten werden. Als erstes wichtiges Produkt des Softwareengineering-Prozesses entsteht die Problembeschreibung in Zusammenarbeit mit den Auftraggebern, hier also dem Lehrstuhl für Thermische Verfahrenstechnik der Universität Kaiserslautern. Sie schildert alle an das zu entwickelnde System gestellten Anforderungen in Form eines Fließtextes und beschreibt so die *erwartete Systemfunktionalität*. Da diese Arbeit in einer Domäne stattfindet, die nur wenigen Lesern aus dem Fachbereich Informatik vertraut ist, wird vor die eigentliche Problembeschreibung eine Beschreibung des Arbeitsgebiets Verfahrenstechnik gestellt. Anschließend erfolgt eine textuelle Beschreibung der Anforderungen an das Labor. Um in späteren Kapiteln besser auf einzelne Bestandteile dieser Anforderungen Bezug nehmen zu können, werden diese zusätzlich in tabellarischer Form aufbereitet und in Anhang A aufgeführt.

2.1 Aktuelle Situation im Bereich Verfahrenstechnik

Die wissenschaftliche Arbeit zur Untersuchung von Mehrphasensystemen hat die Untersuchung des Verhaltens von Anlagen (*Kolonnen*), Prozessen und Stoffen zum Ziel, mit denen Wertstoffe aus einem Wertstoff-Träger-Gemisch mit Hilfe eines Lösungsmittels herausgelöst werden. Der Schwerpunkt liegt dabei auf der Gewinnung von Messwerten, die als Grundlage für die Entwicklung und Verbesserung von mathematischen Modellen zur Beschreibung der in diesen Anlagen stattfindenden Vorgänge dienen. Diese Modelle liegen in Form von Korrelationsgleichungen vor, die mit physikalischem Hintergrund jedoch oft auch empirisch ermittelt wurden. Durch kontinuierliche Untersuchungen sollen Parameter dieser Gleichungen so verbessert werden, dass sie die in Experimenten gemachten Beobachtungen möglichst exakt nachbilden. Dazu gilt es, die Abweichung von realem Messwert und dem vom Modell vorhergesagten Wert zu minimieren. Dazu wird im Allgemeinen auf das Verfahren der Parameterregression zurückgegriffen.

Ziel der Bemühungen ist es, die Entwicklung von Großanlagen im Industriemaßstab zu optimieren, indem Schlüsselparameter dieser Anlagen mit Hilfe der Modelle ermittelt werden. Dabei erweist sich jedoch der sogenannte *Scale-Up*, also der Schluss von den in Experimenten in kleinem Maßstab gewonnen Modellen auf Großanlagen als schwierig.

Im aktuellen Forschungsbetrieb werden zwei separate Verfahren eingesetzt. Zum einen werden reale Experimente an kleinen Laboranlagen und größeren *Technikumsanlagen* durchgeführt. Parallel dazu beschäftigen sich andere Mitarbeiter mit Verfahren und Modellen zur Simulation derartiger Versuche.

2.1.1 Reale Experimente

Um dem Verfasser als Fachfremdem ein besseres Verständnis der Domäne Verfahrenstechnik zu erlauben, wurde ihm von Mitarbeitern des Lehrstuhls Bart Einblick in die Durchführung von Experimenten ermöglicht. Wegen der Vielzahl an Experimentarten sei hier exemplarisch ein Experiment zur Untersuchung des Tropfenzerfalls herangezogen. Dazu wurde in der Werkshalle des Lehrstuhls eine insgesamt über vier Meter hohe Technikumsanlage aufgebaut. Zentraler Bestandteil bildet der *Mixer*, ein langer aufrecht stehender Glaszylinder, in dem sich Einbauten wie Rührelemente und Trennwände befinden. In diesen Glaszylinder kann von oben das sogenannte *Feed* eingebracht werden, eine Mischung des zu extrahierenden Wertstoffs mit einem Trägermittel. Vom unteren Ende wird das Lösungsmittel (*Solvent*) durch eine oder mehrere Düsen in die Apparatur eingebracht. Man spricht daher auch von *Gegenstromextraktionskolonnen*. An mehreren Stellen innerhalb der Anlage befinden sich Öffnungen, in die Messsonden eingebracht werden können. Diese messen bei der Experimentdurchführung Werte wie Tropfenanzahl oder -durchmesser und liefern diese auf elektronischem Wege an einen Messrechner. Dabei handelt es sich im Allgemeinen um einen Standard-PC, der mit speziell an die Messsonde angepassten Schnittstellen und Softwarelösungen ausgestattet ist. Diese Rechner sind oft seit langer Zeit im Einsatz. Dadurch liegen die anfallenden Daten in heterogenen und häufig proprietären Formaten vor. Neben diesen kontinuierlich arbeitenden Messverfahren gibt es auch solche, die einen Eingriff in das laufende Experiment zum Beispiel durch Entnahme einer Stoffprobe erfordern. Diese Messungen sind destruktiv, das bedeutet sie können nur einmal pro laufendem Experiment durchgeführt werden, das anschließend neu aufgesetzt werden muss.

Die Dimensionen dieser Anlage veranschaulichen bereits den erforderlichen materiellen und personellen Aufwand, der für die Durchführung eines solchen Experiments zu betreiben ist. Neben einer großen Anzahl allgemein erhältlicher Standardbauteile sind für den Aufbau häufig auch kostspielige Spezialanfertigungen erforderlich. Deren Herstellung erfordert oft einige Wochen und führt zu enormen zeitlichen Verzögerungen.

Ist der Aufbau einer Apparatur abgeschlossen, kann mit der Durchführung von Experimenten begonnen werden. Dazu müssen die Vorratsbehälter der Anlage mit den zu untersuchenden Substanzen befüllt werden. Weiterhin können bestimmte Betriebsparameter variiert werden. Neben der Drehzahl der Rührwerke lassen sich beispielsweise die Durchflussgeschwindigkeit oder die Eingangstropfengröße des Lösungsmittels anpassen oder das Mischungsverhältnis der beteiligten Stoffe ändern.

Ist das Experiment vorbereitet, wird die Anlage eingeschaltet. Um verlässliche Ergebnisse zu erhalten, lässt man vor den eigentlichen Messungen eine gewisse Zeit zur Stabilisierung verstreichen. Anschließend kann mit der Ermittlung der Messwerte begonnen werden. Die Dauer der Erfassung liegt im Bereich von wenigen Minuten bis mehreren Stunden.

Neben der hohen Kosten für den Aufbau werden aus dieser Beschreibung weitere Nachteile des realen Experiments deutlich. So erfordert die Durchführung eines neuen Experiments häufig Umbauarbeiten an der Anlage. Ein Austausch des Rührertyps lässt sich nur durch Zerlegung von großen Teilen des Aufbaus durchführen. Ein Wechsel der zu untersuchenden Stoffe macht eine aufwendige Reinigung des gesamten Systems erforderlich. Zudem ist der Verbrauch an den nicht immer ungefährlichen Substanzen hoch. Diese sind nicht nur für die Mitarbeiter eine Gefahr, sondern auch problematisch und daher teuer in der Entsorgung.

Trotz des hohen Aufwandes sind der Größe von Versuchsaufbauten enge Grenzen gesetzt. Während sich Laboranlagen im Bereich unter einem Meter bewegen, erreichen Technikumsanlagen bis zu fünf Meter Gesamthöhe. Im Industriemaßstab sind jedoch Anlagen im Bereich von 20 Metern Höhe und mehr üblich. Auch die Verwendung von existierenden Industrieanlagen zur Gewinnung von Messwerten für die Forschung ist schwierig, da diese als Produktivsysteme häufig nicht zur Verfügung stehen und zudem nicht für die Erfassung der erforderlichen Messwerte ausgestattet sind.

Die in den Experimenten gewonnen Ergebnisse werden archiviert und ausgewertet. Um die Ergebnisse untereinander vergleichbar und für spätere Analysen auffindbar zu machen, werden sie zusammen mit einer Beschreibung des durchgeführten Versuchs gespeichert.

2.1.2 Datenhaltung

In Abhängigkeit vom jeweiligen Experimenttyp werden die wichtigsten Daten über Apparatur und Experiment in semistrukturierter Form festgehalten. In existierender Software zur Verwaltung der Daten sind dazu für die verschiedenen Experimenttypen entsprechende Felder vorgesehen, die mit den im Experiment verwendeten Werten zu versehen sind. Dadurch ist innerhalb eines Experimenttyps in Grenzen eine Suche auf Basis der hier eingetragenen Werte möglich. Hier erweist sich der häufig informelle Charakter der Beschreibungen als Hindernis. So herrschen textuelle Beschreibungen vor, die für die Dokumentation von zahlreichen Aspekten eines Experiments und der verwendeten Apparatur wenig geeignet sind. Auch erschwert die Verwendung von Synonymen eine effektive Suche.

Die zur Beschreibung des Experiments aus obigem Beispiel zu speichernden Daten umfassen zahlreiche Bereiche, die im Folgenden kurz erläutert werden.

Während die wichtigsten geometrischen Daten über die Abmessungen zentraler Elemente der Apparatur wie Länge und Durchmesser des Mixers oder Typ und Abmessungen des Rührwerks noch strukturiert in den entsprechenden Feldern eingetragen werden können, erfolgt beispielsweise die Angabe der Art und Position von Messsonden in informeller Weise. Eine Dokumentation des Aufbaus mit Zeichnungen oder Fotos ist nicht üblich, zumal hier kein einheitliches Format für den Austausch dieser Daten zwischen verschiedenen Forschungsgruppen existiert.

Nach der Beschreibung der Apparatur werden die Parameter des Experiments erfasst. Diese beinhalten im Wesentlichen die verwendeten Substanzen und die Versuchsbedingungen wie Drücke, Umgebungstemperatur, Drehzahl des Rührwerks und den Stoffdurchsatz. Besonders bei der Beschreibung der verwendeten Substanzen ergibt sich die Schwierigkeit, dass es kei-

ne domänenweit einheitliche Katalogisierung gibt. So kommt auch hier eine rein textuelle Beschreibung zur Anwendung, mit den bekannten Problemen bei der späteren Suche in den Experimentdaten.

Die aus dem Versuch gewonnen Messdaten werden anschließend in tabellarischer Form gespeichert. Dabei handelt es sich in erster Linie um eindimensionale, diskrete Messwerte, die in regelmäßigen Zeitabständen ermittelt wurden. Die Speicherung erfolgt in Form von Textdateien oder als Dateien gängiger Tabellenkalkulationssoftware. Die Verwendung von proprietären Formaten zur Speicherung der Daten erschwert auch hier die Archivierung und verhindert experimentübergreifende Suchanfragen. Die Übernahme der Messwerte von den Datenquellen (Messrechner beziehungsweise Ausgabe des Simulationstools) erfolgt im Allgemeinen manuell, ebenso die Auswertung durch die Wissenschaftler. Diese erfolgt ohne weitergehende Rechnerunterstützung dateibasiert in Form von Spreadsheets. Diese Form der Auswertung ist langwierig und erfordert trotz eher repetitiver Arbeitsabläufe ständige Konzentration. In der Analysephase wird die große Zahl von Messwerten zu aussagekräftigeren Werten verdichtet. Im eingangs geschilderten Beispiel könnte es sich dabei um Messwerte der Tropfenanzahl und -größe in den einzelnen Stufen handeln. Daraus werden dann mittels mathematischer Verfahren unter anderem der Volumenanteil des Lösungsmittels, die mittlere Tröpfchengröße oder Werte für die Zerfallsrate der Tröpfchen und den Stoffaustausch an der Grenzfläche zwischen Feed und Solvent ermittelt. Diese Ergebnisse werden zur Verfeinerung der mathematischen Modelle genutzt, indem aus den Messwerten Parameter für die Modellgleichungen ermittelt werden.

2.1.3 Simulationen

Neben der Durchführung von realen Experimenten stellen Simulationen das andere Standbein der Forschungsaktivitäten dar. Stellvertretend für die Vielzahl von Simulationswerkzeugen, die in der Verfahrenstechnik eingesetzt werden, sollen hier zwei Lösungen kurz vorgestellt werden.

Fluent

Fluent ist eine Software, welche die Visualisierung des Strömungsverhaltens in Einphasensystemen ermöglicht. Ein Fluent vorgeschaltetes einfaches CAD-Tool (*Gambit*) erlaubt die Erstellung der Geometrie von Apparaturen oder den Import von Geometriebeschreibungen aus anderen CAD-Formaten. Bezug nehmend auf diese Geometrie werden dann Eigenschaften und Randbedingungen der Apparatur beschrieben. Diese Informationen dienen als Eingabe für das eigentliche Simulationswerkzeug Fluent, den sogenannte *Solver*, das die Berechnung des Strömungsverhaltens durchführt und verschiedene Verfahren zur Visualisierung anbietet. Trotz seiner mächtigen Funktionen weist Fluent einige Beschränkungen auf. So war bis zur aktuellen Version nur die Simulation von Einphasensystemen möglich. Zudem ist die Erstellung der CAD-Modelle aufwendig und kann nicht immer von den Experten aus der Anwendungsdomäne durchgeführt werden.

PopMod

Das Simulationstool PopMod ist eine Umsetzung des gleichnamigen Tropfenpopulationsbilanzmodells, das am Lehrstuhl für thermische Verfahrenstechnik entstanden ist. Das in Fortran entwickelte Programm erlaubt die Durchführung von Simulationen über den Tropfenzerfall in Extraktionskolonnen. In seiner jetzigen Form handelt es sich um ein nicht-graphisches Tool. Experimente werden durch Setzen von Parametern in Konfigurationsdateien beschrieben. Dazu gehören Informationen über die Kolonne, wie ihre Gesamthöhe, die Höhe der Einlässe und Art und Anzahl der Einbauten. Weiterhin werden die Eigenschaften der verwendeten Stoffe, beispielsweise Dichte und Viskosität, sowie Eigenschaften definiert, die für die Stoffpaarung von Bedeutung sind, zum Beispiel die Grenzflächenspannung. Zusätzlich ist es möglich, das für die Simulation zu verwendende mathematische Modell aus einer vorgegebenen Menge auszuwählen und einige Betriebsparameter zu setzen.

Beim Start der Simulation werden die Konfigurationsdateien eingelesen und als Eingaben für die Berechnung verwendet. Die Ergebnisse, im Wesentlichen handelt es sich dabei um Informationen über die Verteilung der Tröpfchengröße in verschiedenen Höhen der Kolonne, werden als Tabellen in Textdateien ausgegeben. Diese werden anschließend manuell in ein Tabellenkalkulationsprogramm importiert und dort weiter verarbeitet. Dazu werden Durchschnittswerte für die Tröpfchengröße in verschiedenen Höhen gebildet und der Stoffaustausch zwischen den Phasen berechnet.

2.1.4 Zusammenfassung

Prägend für die momentane Arbeitsweise im Bereich Verfahrenstechnik ist die stark getrennte Verwendung der Werkzeuge „Experiment“ und „Simulation“. So ist häufig auch eine Trennung der Arbeitsgebiete der Mitarbeiter zu beobachten. Während sich ein Teil hauptsächlich mit der Verfeinerung von Simulationstools beschäftigt, führt ein anderer Teil der Mitarbeiter reale Experimente durch und wertet sie aus. Hier könnte ein virtuelles Labor helfen, diese Kluft zu überwinden, indem es den bisher am realen Experiment arbeitenden Mitarbeitern einen intuitiveren Zugang zu den Möglichkeiten bietet, die durch moderne Computerhard- und -software zur Verfügung stehen. Insbesondere in Anbetracht des enormen finanziellen und zeitlichen Aufwands, den reale Experimente mit sich bringen, könnten sie durch den Einsatz virtueller Experimente ergänzt werden. So ließe sich bei gleichbleibender finanzieller Ausstattung ein größere Menge an Ergebnissen erzielen.

Existierende Simulationstools sind häufig auf ein sehr enges Anwendungsgebiet eingeschränkt, wenig benutzerfreundlich und als Insellösungen nur sehr schwer in vorhandene Systeme zu integrieren. Auch die Auswertung der Ergebnisse, die momentan häufig mit Spreadsheets durchgeführt wird, könnte durch eine geeignete Unterstützung im virtuellen Labor effizienter gestaltet werden. So könnten Ergebnisse von realen und virtuellen Experimenten in das Repository des virtuellen Labors eingebracht werden. Durch eine einheitliche Datenhaltung können den Benutzern mächtigere Anfragemöglichkeiten zur Verfügung gestellt werden. Die Automatisierung von Arbeitsabläufen bei der Auswertung bietet zudem ein erhebliches Potential für Zeitersparnisse.

2.2 Erwartete Systemfunktionalität

Im folgenden Kapitel werden die Anforderungen der beteiligten Fachbereiche an ein virtuelles Labor in Form eines natürlichsprachlichen Textes zusammengefasst. Er enthält sowohl die Anforderungen der Anwender der Verfahrenstechnik als auch die der Mathematik. Sie wurden hauptsächlich in Gesprächen mit Mitarbeitern aus den entsprechenden Fachbereichen entwickelt. Dabei wurden gegenseitig Vorschläge unterbreitet und erläutert, die teilweise in dieses Dokument übernommen wurden.

Die Anforderungen der Verfahrenstechnik konzentrieren sich auf die vom System zu unterstützenden Arbeitsabläufe, die gewünschte Funktionalität bei der Definition und der Durchführung von Experimenten sowie die Art der auftretenden virtuellen Messergebnisse und deren weitere Verwendung.

Die Anforderungen der Mathematik ergeben sich aus den für die spätere Durchführung einer Simulation benötigten Informationen über den Experimentaufbau wie Geometrieinformationen und Randbedingungen.

Es wurde noch keine Reduktion der Anforderungen auf die Aspekte der Datenhaltungskomponente durchgeführt, vielmehr wurden alle Aspekte des Gesamtsystems aufgenommen.

Im folgenden Text wurden soweit möglich die Begriffe aus dem Konzeptpapier zum geplanten Sonderforschungsbereich [VL2002] verwendet. Diese und neudefinierte Begriffe werden bei ihrer ersten Verwendung durch kursive Schreibweise kenntlich gemacht. Die Gliederung des Textes orientiert sich am Arbeitsablauf bei der Durchführung eines virtuellen Experiments.

2.2.1 Allgemeine Konzepte

Um den Benutzer von den Details der inneren Funktionsweise des virtuellen Labors zu entlasten, ist eine grundsätzliche Trennung der Sichten des Experimentators und der internen Darstellung des Systems einzuhalten. Die Experimentatorsicht, im Folgenden kurz *Nutzersicht* genannt, beschränkt sich in ihrer Darstellung auf die für den Benutzer wesentlichen Aspekte des virtuellen Labors und seiner Komponenten. Die Sicht, welche die Grundlage für die Ausführung des virtuellen Experiments durch das System bildet (*Systemsicht, interne Sicht*), bleibt im Normalbetrieb vor dem Benutzer verborgen.

Das virtuelle Labor soll in der Lage sein, Produkte und Ergebnisse (Objekte) den jeweiligen Experimentatoren zuzuordnen. Dazu ist eine Benutzerverwaltung mit Authentifizierungsmechanismen vorzusehen. Der Benutzer soll in der Lage sein, seine Experimente und Ergebnisse anderen Benutzern zugänglich zu machen und zugleich auf die Ergebnisse von Kollegen zurückgreifen zu können. Durch geeignete Freigabemechanismen soll dies möglichst feingranular auf der Ebene einzelner Benutzer, Gruppen sowie aller Benutzer des Labors möglich sein. Dabei sollen die Objekte wahlweise nur für lesenden Zugriff oder auch für Änderungen freigegeben werden. Als Zwischenstufe zwischen nur lesendem und schreibendem Zugriff besteht die Möglichkeit, das Ableiten von neuen Versionen zu erlauben.

Um den Anwendern aus dem Bereich der Verfahrenstechnik einen intuitiven Zugang zum virtuellen Labor zu ermöglichen, soll es die Arbeitsabläufe und Vorgehensweisen realer Experimente möglichst genau nachbilden.

Die Durchführung realer Experimente lässt sich in drei Phasen gliedern, die *Versuchsplanung*, die *Versuchsdurchführung* und die *Versuchsanalyse*. Diese drei Phasen sollen sich auch im virtuellen Labor widerspiegeln und werden in den folgenden Abschnitten 2.2.2, 2.2.3 und 2.2.4 näher erläutert.

2.2.2 Versuchsplanung

In dieser ersten Phase steht dem Experimentator eine Bibliothek von virtuellen Versuchsbausteinen oder *Bauteilen* zur Verfügung, die auf einer virtuellen Arbeitsfläche (*Workbench*) zu einer *virtuellen Apparatur* zusammengesetzt werden. Dies geschieht analog zur Vorgehensweise beim realen Experiment, bei dem reale Bausteine zu Apparaturen kombiniert werden.

Virtuelle Bausteine können sowohl einzelne (allgemein erhältliche) reale Bausteine repräsentieren als auch Familien von ähnlichen realen Bausteinen, die sich lediglich in einigen Parametern unterscheiden. Diese Typen von Bausteinen sollen als *echte Bausteine* bezeichnet werden. Neben diesen Bausteinen mit einer realen Entsprechung soll zudem die Möglichkeit bestehen, Bausteine auszuwählen, für die es kein reales Gegenstück gibt (*fiktive Bausteine*). Dadurch können beispielsweise geplante Spezialanfertigungen im System repräsentiert werden.

Die Trennung in echte und fiktive Bausteine soll zwar ersichtlich sein, aber auf die spätere Verwendung der Bausteine keinen Einfluss haben.

Um im späteren Versuch Messergebnisse ermitteln zu können, soll die Möglichkeit bestehen, als speziellen Bausteintyp auch *Messsonden* einzusetzen. Diese liefern zusätzlich zu ihren Eigenschaften als Baustein Messwerte bei der Durchführung des virtuellen Experiments analog zu den *Messzonen*, die in Abschnitt 2.2.2.8 beschrieben werden.

2.2.2.1 Beschreibung der Bausteine

Entsprechendem dem Grundkonzept des virtuellen Labors werden Bausteine in zwei Sichten beschrieben: einer Nutzersicht und einer Systemsicht.

In der Nutzersicht sind alle für den Experimentator relevanten Daten oder *Eigenschaften* (*Properties*) eines Bausteins zugänglich. Dazu gehören zum einen allgemeine Beschreibungen des Bausteins, wie eine eindeutige Bezeichnung, der Hersteller und Bestellinformationen, falls es sich um einen echten Baustein handelt, kurze beschreibende Texte und andere ergänzende Informationen. Zum anderen sind wichtige geometrische Größen wie Länge, Breite, Höhe oder Durchmesser sowie Angaben zum Material erforderlich, aus dem der Baustein beziehungsweise einzelne Elemente des Bausteins bestehen. Weiterhin sind Lage und Art von Verbindungselementen (mechanische Anschlüsse) des Bausteins verfügbar zu machen.

Eigenschaften haben einen innerhalb des Bausteins eindeutigen Namen und einen Wert, der von unterschiedlichem Typ sein kann. Neben *skalaren* Typen (Anzahl, Temperatur, Druck, Abmessungen etc.), die durch ganze oder rationale Zahlen dargestellt werden können, sollen auch *vektorielle* Typen (Strömungsrichtung, Geschwindigkeit etc.) spezifiziert werden können. Für Typen, die physikalische Größen repräsentieren, sind verschiedene Einheiten zu unterstützen, zwischen denen eine automatische Konvertierung erfolgen kann. Neben diesen numerischen Typen ist auch der Typ „Text“ möglich.

Andere Eigenschaften haben komplexe Typen, die selbst aus einfacheren Attributen zusammengesetzt sind. So verfügt beispielsweise ein Typ „Substanz“ über textuelle Beschreibungen wie Name und chemische Formel und über numerische Attribute wie Dichte und Viskosität. Komplexe Typen erlauben im Allgemeinen nur die Wahl zwischen vordefinierten Wertemengen (Auflistungen, *Enumeration*), zum Beispiel die Auswahl von chemischen Substanzen. Diese Werte können auch hierarchisch organisiert sein (Beispiel: Flüssigkeit → Alkohol → einwertiger Alkohol → Methanol).

Eigenschaften lassen sich in *verpflichtende Eigenschaften* und *optionale Eigenschaften* unterteilen. Verpflichtenden Eigenschaften wie der Länge eines Rohrs muss ein Wert zugewiesen sein, während dies bei optionalen Eigenschaften wie einer Beschreibung unterbleiben kann.

Parameter

Eigenschaften können variabel gehalten sein und werden dann als *Parameter* bezeichnet. Es können zwei Klassen von Parametern unterschieden werden:

Bausteinparameter beschreiben im Allgemeinen Eigenschaften eines Bausteins, die einmal vor seiner Verwendung in einer virtuellen Apparatur festgelegt und anschließend üblicherweise nicht mehr geändert werden. Ein Beispiel für Bausteinparameter sind Daten über die Geometrie des Bausteins. Eine Änderung dieser Parameter ist zwar auch nach dem Einbau möglich, aber wie beim realen Experiment mit erhöhtem Aufwand verbunden, da sie sich auch auf andere Bausteine innerhalb der virtuellen Apparatur auswirken können.

Im Gegensatz dazu beschreiben *Betriebsparameter* Eigenschaften des Bausteins, die von Versuch zu Versuch geändert werden können, ohne Änderungen in anderen Bausteinen zu erfordern. Beispiele hierfür sind die Position von beweglichen Elementen, Drehzahl, Temperatur, Durchfluss etc.

Bausteinvarianten

Bausteine, die sich nur in den Werten ihrer Parameter unterscheiden, werden als *Bausteinvarianten* bezeichnet. Zu jedem Baustein existiert mindestens eine Variante mit den Standardwerten für alle verpflichtenden Bausteinparameter (*Standardvariante*). Für einen Baustein können eine beliebige Zahl von weiteren Varianten in der Bausteinbibliothek vorliegen. Diese werden durch eindeutige Bezeichnungen voneinander unterschieden. Bausteine können entweder direkt in virtuellen Apparaturen genutzt werden, wobei zur korrekten Benutzung alle verpflichtenden Parameter vom Benutzer mit Werten zu versehen sind. Alternativ kön-

nen direkt Bausteinvarianten verwendet werden, das manuelle Setzen der Werte für verpflichtende Parameter ist so nicht erforderlich, da diese von der Variante vorgegeben werden. Eine Änderung der durch die Variante vorgegebenen Parameter ist jederzeit möglich, ohne dadurch die als Ausgangspunkt dienende Variante zu ändern.

Um gegenüber einer realen Experimentierumgebung weitere Flexibilität zu gewinnen, soll dem Benutzer die Möglichkeit gegeben werden, aus den Standardvarianten der Bausteine in der Bibliothek eigene Varianten abzuleiten, indem er selbst die Parameter anpasst. Dadurch können beliebige, in der Bausteinbibliothek nicht vorgesehene Bausteinvarianten erzeugt und im virtuellen Experiment eingesetzt werden. So lässt sich beispielsweise kostengünstig die Sinnhaftigkeit einer Spezialanfertigung prüfen. Benutzerdefinierte Bausteinvarianten sollen zusammen mit den vordefinierten Bausteinvarianten in die Bausteinbibliothek aufgenommen werden können, wobei eine Bezeichnung für die Variante zu vergeben ist.

Konsistenzbedingungen

Bei der Änderung von Parametern wie sie bei der Ableitung von neuen Bausteinvarianten erfolgt, sollen diese auf Konsistenz überprüft werden können, um nicht realisierbare Bauteile auszuschließen. Dazu können im System beliebige *Konsistenzbedingungen* (*Constraints*) formuliert werden, die unter anderem die gültigen Wertebereiche für die Parameter des Bausteins vorgeben. Konsistenzbedingungen können sowohl auf Ebene einzelner Parameter vorgegeben werden (beispielsweise Länge > 0), und gelten dann automatisch für alle Bausteine, die diesen Parameter beinhalten. Ebenso können Konsistenzbedingungen auf Ebene des Bausteins definiert werden, hier kann bei der Formulierung auf mehrere Eigenschaften des Bausteins Bezug genommen werden, zum Beispiel zur Formulierung der Bedingung „Innendurchmesser < Außendurchmesser“. Constraints für Bausteine können auch die Position und erlaubte Lage des Bausteins einschränken oder Mindestabstände zu benachbarten Bausteinen fordern.

Konsistenzbedingungen können ebenfalls auf der Ebene von Bausteingruppen spezifiziert werden. Dadurch können Anforderungen formuliert werden, welche die korrekte Nutzung eines Bausteins zusammen mit anderen Bausteinen beschreiben (Baustein A muss innerhalb von Baustein B liegen).

Um zugleich eine große Flexibilität bei der Formulierung von Konsistenzbedingungen zu erlauben und dem Benutzer einen intuitiven Zugang zu ermöglichen, sind die Konsistenzbedingungen durch eine Beschreibungssprache zu realisieren. Sie soll neben einfachen algebraischen Ausdrücken (A < B + C) auch komplexere Operatoren wie „Baustein A innerhalb Baustein B“ bieten.

2.2.2.2 Darstellung der Bausteine

Um eine Visualisierung der Bausteine auf der Workbench zu ermöglichen, sollte neben der textuellen Beschreibung der Bausteineigenschaften auch eine graphische Darstellung der Bausteine verfügbar sein. Diese Darstellung kann vom tatsächlichen Aussehen des Bausteins

zu Gunsten einer einfacheren und übersichtlicheren Ansicht abstrahieren. Sie sollte Änderungen von wichtigen Parametern des Bausteins widerspiegeln.

Neben einer rein symbolischen Darstellung auf der Workbench unter Verzicht auf eine exakte Darstellung von Proportionen sollte optional ein exakte Darstellung möglich sein. Dadurch kann der Benutzer beim Aufbau von virtuellen Experimenten auf Wunsch weiter von der Geometrie abstrahieren und so die virtuelle Apparatur übersichtlicher gestalten.

2.2.2.3 Interne Bausteinrepräsentation

Die Systemsicht auf die Bausteine umfasst zusätzlich zu den vereinfachenden Daten der Nutzersicht auch Daten über die exakte Geometrie von Bauteilen sowie weitere Randbedingungen, die für die Durchführung des virtuellen Experiments durch das System erforderlich sind.

Die Geometrie von Bausteinen wird in Form von geschlossenen geometrischen Körpern definiert. Komplexere Bausteine lassen sich bei der Erstellung durch Kombination von dreidimensionalen geometrischen Grundformen (*Solids, Primitives*) wie Quadern, Zylindern, Kugeln, Prismen oder Pyramiden beschreiben, die frei positioniert und skaliert werden können und durch Operationen wie Schnitt, Vereinigung und Differenz die Definition komplexerer Geometrien erlauben. Um auch aufwendiger geformte Objekte spezifizieren zu können, soll die Definition von Oberflächen später auch auf Basis mathematischer Beschreibungen wie Splines ermöglicht werden. Die Beschreibung der Geometrie soll exakt erfolgen, also ohne Vereinfachungen wie eine Approximation von gekrümmten Flächen durch Polyeder.

Einzelnen Oberflächen dieser Teile sollen weitere Randbedingungen zugeordnet werden können, beispielsweise Druck und Temperatur, Definition als Ein- oder Auslassöffnung etc.

Besondere Bedeutung für das spätere Zusammenfügen von Bausteinen zu einer virtuellen Apparatur haben die Verbindungselemente der Bausteine. Für sie muss eine Beschreibung existieren, die es erlaubt, die Kompatibilität von verschiedenen Bausteinen zu überprüfen. Dies kann durch direkte Beschreibungen der jeweils zu einem Verbindungselement kompatiblen anderen Verbindungselemente geschehen (Verbindungselement A passt auf Verbindungselemente B und C) oder durch Prüfung der Eigenschaften der Verbindungselemente auf (mechanische) Kompatibilität.

Da Änderungen der Baustein- und Betriebsparameter in der Nutzersicht Auswirkungen auf die Systemsicht haben können, muss für jeden Parameter der Nutzersicht beschrieben sein, welche Teile der exakten Darstellung der Systemsicht (Geometrie, Oberflächeneigenschaften) in welcher Weise beeinflusst werden. So bewirkt eine Änderung des Längenparameters eines Rohrs eine entsprechende Skalierung der Solids, welche die Rohrwände darstellen.

2.2.2.4 Zusammengesetzte Bausteine

Neben der grundlegenden Beschreibung von Bausteinen aus geometrischen Formen sollen Bausteine auch aus anderen existierenden Bausteinen zusammengesetzt werden können.

Diese werden dann als *komplexe Bausteine* im Gegensatz zu den *einfachen Bausteinen* bezeichnet.

Zusätzlich zu den Eigenschaften der einfachen Bausteine muss für komplexe Bausteine definiert werden, wie sich Änderungen der Parameter des komplexen Bausteins auf Parameter, Position und Orientierung seiner Komponenten auswirken.

2.2.2.5 Erstellung von Bausteinen

Für erfahrene Benutzer soll auf Wunsch die Systemsicht der Bausteine zugänglich gemacht werden. Hier soll die Möglichkeit bestehen, eigene Bausteine durch Erstellung aller Informationen für die System- und Nutzersicht zu definieren und diese in die Bausteinbibliothek aufzunehmen.

Bei der Definition eigener Bausteine sollen neben den Grundformen (Solids) auch eine beliebige Zahl existierende Bausteine als Ausgangsbasis genutzt werden können. Der Benutzer muss anschließend die Beziehungen zwischen Parametern der Nutzersicht und den Objekten der Systemsicht definieren. Um existierende Geometriedaten in das virtuelle Labor direkt übernehmen zu können, sind auch Importfilter für die gängigen CAD-Formate vorzusehen.

Der Ersteller eines Bausteins ist zugleich dessen Besitzer und kann ihn anderen Nutzern über die eingangs beschriebenen Freigabemechanismen zugänglich machen. Wird ein derartig freigegebener Baustein irgendwo im System genutzt, kann er auch vom Besitzer nicht mehr geändert werden. Die Ableitung einer neuen Version ist dagegen weiterhin möglich.

2.2.2.6 Bausteinbibliothek

Um ein einfaches Auffinden von Bausteinen zu ermöglichen, ist eine effiziente Suche innerhalb der Bausteinbibliothek zu ermöglichen. Der primäre Zugang zu den Bausteinen wird über eine Einordnung der Bausteine in hierarchisch organisierte Kategorien und Unterkategorien erfolgen. Der Benutzer navigiert durch diesen Kategoriebaum und trifft seine Auswahl anhand der Kategoriebezeichnungen. Kategorien können selbst über Eigenschaften verfügen, die wie die Bausteineigenschaften Name und Typ jedoch keinen Wert haben. Unterkategorien besitzen dabei grundsätzlich mindestens alle Eigenschaften ihrer übergeordneten Kategorie. Bausteine innerhalb einer Kategorie zeichnen sich durch ähnliche Einsatzmöglichkeiten aus und müssen über mindestens alle Eigenschaften ihrer Kategorie verfügen. So haben Rohre jeder Art eine Länge, einen Innen- und Außendurchmesser. Eine Unterkategorie von allgemeinen Rohren wären beispielsweise gekrümmte Rohre, bei denen zusätzlich die Position der Krümmung und der Krümmungsradius angegeben werden muss.

Durch die Eigenschaft der Kategorienhierarchie, dass alle Kategorien und Bausteine die Eigenschaften ihrer übergeordneten Kategorien erben, kann zur Ergänzung der manuellen Navigation durch die Kategorien eine Suche auf den Eigenschaften einer Kategorie stattfinden. Dazu können Suchbedingungen auf diesen Eigenschaften formuliert werden, auf die dann alle direkt oder indirekt innerhalb der gewählten Kategorie liegenden Bausteinvarianten überprüft werden.

Kategorien und ihre Eigenschaften können dabei ähnlich wie Bausteine über Konsistenzbedingungen verfügen, die für alle Objekte innerhalb dieser Kategorie erfüllt sein müssen.

Zusätzlich zur manuellen Suche in der Kategorienhierarchie soll bei der Suche auf Kompatibilität zu anderen Bausteinen Bezug genommen werden können („Finde alle Bausteine vom Typ t, die auf Anschluß 1 von Bauteil X passen“).

2.2.2.7 Kombination von Bausteinen zu einer virtuellen Apparatur

Die Bausteinvarianten werden vom Experimentator zu einer virtuellen Apparatur zusammengesetzt. Dazu können sie auf der Workbench platziert werden, indem Position und Orientierung des Bausteins festgelegt werden. Von der exakten maßstabsgetreuen Anordnung kann zugunsten einer übersichtlicheren Darstellung der virtuellen Apparatur verzichtet werden. Um die virtueller Apparatur übersichtlicher zu gestalten, kann auf eine Simulation der Schwerkraft beim Experimentaufbau verzichtet werden. Dadurch können Stützelemente ohne weitergehende Funktion für die Versuchsdurchführung entfallen.

Die Betriebs- und Bausteinparameter sowie sonstige Informationen über die Bausteine sollen dabei jederzeit zugänglich und änderbar sein. Sollten Änderungen von Parametern zu Inkonsistenzen innerhalb der Apparatur führen, ist der Nutzer darauf hinzuweisen. Inkonsistente Zustände sind jedoch als normales Ergebnis eines Konstruktionsprozesses zuzulassen.

Die Anschlüsse der Bausteine können verbunden werden, dabei soll eine Überprüfung erfolgen, ob die zu verbindenden Anschlüsse der Bauteile zueinander passen.

2.2.2.8 Messzonen

Um gegenüber realen Experimenten weitergehende Erkenntnisse zu ermöglichen, sollen zusätzlich zur Nutzung von virtuellen Messsonden als aktivem Baustein der Apparatur auch sogenannte *Messzonen* definiert werden können. Diese sind wie Bausteine Teil der virtuellen Apparatur und ermitteln Messwerte, ohne jedoch wie Messsonden das Experiment zu beeinflussen. Man bezeichnet sie daher auch als passive Bausteine, im Gegensatz zu den aktiven Bausteinen, welche Auswirkungen auf das Ergebnis eines Experiments haben. Über Messzonen lassen sich Messwerte gewinnen, die im realen Experiment nicht zu ermitteln wären. Sie stellen damit eine essentielle Funktionalität bereit, da sie es ermöglichen, in das Experiment „hinein zu sehen“.

Messzonen lassen sich analog zu Bausteinen auswählen, und können frei an beliebige Stellen in der Apparatur eingebracht werden. Alternativ können Teile einer Apparatur (zum Beispiel Gefäßwände) zu Messzonen erklärt werden.

Messzonen können punktförmig sein, zum Beispiel zur Messung von Temperatur oder Druck, aber auch durch Kurven, Flächen (Ermittlung der Strömungsgeschwindigkeit) oder Volumen beschrieben werden. Zusätzlich zu dem von einer Messzone abgedeckten Gebiet muss neben der Art der gewonnen Messwerte auch die Häufigkeit der Messungen (Messzeitpunkte) sowie die Dichte der Messpunkte innerhalb höherdimensionaler Messzonen be-

schrieben werden. Die Messpunkte innerhalb einer höherdimensionalen Messzone sind dabei je nach mathematischem Modell der Simulation entweder diskret oder sie beziehen sich auf einen endlichen, fest begrenzten Ausschnitt der Messzone zum Beispiel in Form eines Rechtecks oder Quaders.

Schließlich muss eine Bezeichnung für die Messzone angegeben werden, unter der die von ihr gelieferten Werte später aufzufinden sind.

2.2.2.9 Speicherung von virtuellen Apparaturen und Bausteinen

Zu jedem Zeitpunkt soll der aktuelle Zustand der Apparatur unabhängig von ihrer Konsistenz in eine Bibliothek von virtuellen Apparaturen (*Apparaturenbibliothek*) unter einer vom Benutzer wählbaren Bezeichnung aufgenommen werden können (*Check-In*). Bestehende virtuelle Apparaturen können aus dieser Bibliothek entnommen und modifiziert werden (*Check-Out*). Wurde eine bestehende virtuelle Apparatur aus der Bibliothek entnommen und verändert, kann sie als eigenständige Apparatur unter einer neuen, vom Benutzer zu vergebenden Bezeichnung erneut in die Bibliothek aufgenommen werden (*explizite Versionierung*). Die ursprüngliche Version bleibt dabei unverändert. Virtuelle Apparaturen, die von einem Benutzer bearbeitet werden, können nicht zeitgleich von einem anderen Benutzer bearbeitet werden. Die Ableitung einer neuen Version aus der letzten gespeicherten Version einer in Bearbeitung befindlichen Apparatur ist jedoch zulässig, wenn das entsprechende Recht zur Ableitung gegeben wurde.

Alternativ zur Speicherung einer Apparatur in der zentralen Bibliothek soll auch ein Export in eine Datei möglich sein.

2.2.3 Versuchsdurchführung

Nachdem eine Apparatur vollständig aufgebaut wurde, kann mit ihr ein Experiment durchgeführt werden. Zuvor soll eine abschließende Konsistenzprüfung der Apparatur erfolgen.

2.2.3.1 Parametrisierung der Apparatur und des Experiments

Vor der Durchführung des Experiments hat der Experimentator die Möglichkeit, die Betriebsparameter der Bausteine seiner Apparatur anzupassen, falls beim Aufbau bereits Parameter vorgegeben wurden. Dadurch besteht in Analogie zur Vorgehensweise bei der Durchführung realer Experimente die Möglichkeit, eine Apparatur zur Durchführung zahlreicher Experimente unter unterschiedlichen Bedingungen und mit verschiedenen Stoffen zu nutzen. Wurden einzelne Parameter nicht schon beim Aufbau spezifiziert, so müssen diese vor der Durchführung des virtuellen Experiments festgelegt werden. Der Benutzer kann ähnlich zu den Eingriffsmöglichkeiten in ein reales Experiment die Betriebsparameter auch während der Experimentdurchführung variieren. Dies kann manuell geschehen. Um eine bessere Reproduzierbarkeit derartiger Eingriffe zu erreichen und für langlaufende Experimente eine Automatisierung dieser Eingriffe zu gestatten, können Änderungen von Betriebsparametern auch vor der Experimentdurchführung für bestimmte Zeitpunkte festgelegt werden.

Neben den Parametern für die virtuelle Apparatur sind ferner globale Parameter für das Experiment anzugeben, beispielsweise die Umgebungstemperatur oder die Dauer des Simulationslaufs. Umfang und Art dieser globalen Parameter sind vom für die Durchführung des virtuellen Experiments erforderlichen mathematischen Modell vorgegeben.

Ein solcher Parametersatz soll in Verbindung mit der zugehörigen Apparatur als virtuelles Experiment gesichert werden können. Für jede Apparatur können beliebig viele Parametersätze und damit virtuelle Experimente existieren.

Messwerte

Die von Messsonden und Messzonen während des Experiments ermittelten Messwerte müssen in geeigneter Form für die Analysephase gesichert werden. Dazu können alle Messwerte (um-)benannt sowie die Messhäufigkeit bei Bedarf modifiziert werden. Nach der Durchführung liegen sie in tabellarischer Form vor, die als Grundlage für die Analysephase dient.

Aufgrund der Verschiedenartigkeit der Messsonden und -zonen können die Messergebnisse in zahlreichen Formate und Einheiten vorliegen. Um die unterschiedlichen Einheitensysteme, wie sie oft in verschiedensprachiger Fachliteratur vorkommen, zu berücksichtigen, sollen die Einheiten für Messergebnisse wählbar sein. Dies soll sowohl vor der Messung ermöglicht werden als auch schon bei vorliegenden Werten. Im letzteren Fall soll dann eine automatische Umrechnung der Einheiten auf die jeweils gewählte Einheit erfolgen.

Die einfachste Form von Messwerten stellen skalare Größen wie Temperatur (Kelvin, Grad Celsius, Grad Fahrenheit) oder Druck (kg/m², lbs./ft² etc.) dar.

Vektorielle Größen haben zusätzlich eine Richtungskomponente wie die Strömungsgeschwindigkeit (m/s, km/h etc.).

Zusätzlich können Messsonden und -zonen wie in Abschnitt 2.2.2.8 beschrieben ihre Messwerte nicht nur in einem Punkt, sondern auch auf einer Fläche oder in einem Raum bestimmen. Hier fallen pro Messzeitpunkt nicht ein einzelner, sondern eine Vielzahl von Messwerten für jeden Messpunkt innerhalb der Zone an, die in Form einer zwei- oder dreidimensionalen Matrix zu speichern sind.

Wahl des mathematischen Modells

Insbesondere in der Anfangsphase des virtuellen Labors werden verschiedene mathematische Modelle als Grundlage für die rechnerische Durchführung der virtuellen Experimente erprobt. Hierzu soll dem Experimentator die Möglichkeit gegeben werden, dieses Modell aus einer Liste der verfügbaren Modelle auszuwählen. Modelle können in Analogie zu globalen Parametern selbst parametrisiert werden, um sie beispielsweise an die verwendeten Stoffe oder die Größenordnung des Experimentaufbaus anzupassen.

2.2.3.2 Durchführung des virtuellen Experiments

Sind alle Parameter des virtuellen Experiments festgelegt, kann die Versuchsdurchführung gestartet werden. Auf Grundlage der Systemsicht der Bausteine wird zunächst eine Darstellung der Apparatur generiert, wie sie vom Modell benötigt wird, das der Simulation zugrunde liegt. Dieser Prozess erfolgt automatisch und ohne Nutzerinteraktion.

Bei der anschließenden Durchführung wird aus dem Experiment eine *Experimentinstanz* erzeugt. Sie enthält alle aus dem Experiment gewonnenen Messwerte und bildet die Grundlage für die Analysephase.

Um der möglicherweise langen Durchführungszeiten von virtuellen Experimenten Rechnung zu tragen, können alle Messwerte direkt zum Zeitpunkt ihrer Ermittlung dem Benutzer zugänglich gemacht werden. Dadurch ist bereits während des laufenden Experiments eine Kontrolle möglich. Sollten die Messwerte nicht den Erwartungen des Benutzers entsprechen, kann das Experiment vorzeitig abgebrochen werden. Durch Verknüpfung der anfallenden Daten mit Visualisierungswerkzeugen kann somit schon während der Durchführung eine „Live“-Beobachtung des Experiments erfolgen.

2.2.4 Versuchsanalyse

Bei der Analyse des virtuellen Experiments soll dem Experimentator Zugriff auf alle während der Durchführung angefallenen Messwerte angeboten werden. Neben der tabellarischen Darstellung der Messwerte sollen zusätzlich verschiedene Visualisierungsverfahren für die verschiedenen Klassen von Messdaten zur Verfügung stehen. Zudem muss eine Weiterverarbeitung von Messergebnissen möglich sein. Dazu gehören einfache arithmetische Operationen wie die Summation oder die Durchschnittsbildung über einer Reihe von Messwerten. Auch weitergehende Statistikfunktionen wie die Ermittlung von Erwartungswert und Varianz sind zur Aufbereitung der Messdaten erforderlich.

Visualisierung

Für die Darstellung eindimensionaler Größen sollen verschiedene Diagrammtypen zur Auswahl stehen, die den Verlauf des Messwertes über der Zeit darstellen. Mehrere Messwerte sollen in einem Diagramm zusammengefasst und verglichen werden können.

Zweidimensionale Messwerte, wie sie beispielsweise in einer Messzone für die Strömungsgeschwindigkeit ermittelt werden, können durch Diagramme dargestellt werden, in denen verschiedene Farbabstufungen die Messwerte symbolisieren. Innerhalb des Diagramms können einzelne Punkte ausgewählt werden, um so den exakten Messwert anzuzeigen.

Neben den hier geschilderten Visualisierungsverfahren sollen weitere hinzugefügt werden können.

2.2.4.2 Weiterverarbeitung

Die in virtuellen Experimenten ermittelten Messwerte dienen häufig zur Bestimmung von abgeleiteten Werten wie Stoffkonstanten. Um diese Bestimmung durchzuführen, sind komplexe Verarbeitungsschritte erforderlich, die in einer ersten Version noch außerhalb des virtuellen Labors stattfinden können. Daher sind geeignete Schnittstellen für den Export der Messdaten in andere Anwendungen vorzusehen.

2.2.4.3 Automatisierung

Um sich wiederholende Arbeitsschritte bei der Experimentauswertung nicht manuell durchführen zu müssen, sind sämtliche Schritte mit einem geeigneten Verfahren automatisierbar zu gestalten.

Kapitel 3 Domänenunabhängige Anforderungen

Nachdem in der ersten Phase der Diplomarbeit die Anforderungen an ein virtuelles Labor im Bereich Maschinenbau/Verfahrenstechnik ermittelt wurden, sollen im Folgenden die Gemeinsamkeiten dieser Anforderungen mit denen eines virtuellen Labors für Raum- und Umweltplanung/Siedlungswasserwirtschaft ermittelt werden. Diese wurden parallel zur vorliegenden Arbeit des Verfassers von Alexander Hilliger von Thile in seiner Diplomarbeit entwickelt.

In diesem Kapitel werden die Ergebnisse vorgestellt, die Alexander Hilliger von Thile und der Verfasser gemeinsam aus den ihnen vorliegenden domänenspezifischen Anforderungen erarbeitet haben. Zunächst wird eine kurze Beschreibung des virtuellen Labors Siedlungswasserwirtschaft gegeben. Diese dient lediglich zum groben Verständnis dieser Domäne, für weitere Informationen und eine detaillierte Beschreibung sei auf die Arbeit von Alexander Hilliger von Thile verwiesen [HvT2003]. Anschließend folgt eine Beschreibung aller Anforderungen, die beiden als geeignet für die Aufnahme in ein gemeinsames Informationsmodell erscheinen. Das gemeinsame Informationsmodell dient als Kern für die spätere Entwicklung domänenspezifischer Informationsmodelle.

Basierend auf den domänenunabhängigen Anforderungen wird anschließend ein Kernmodell entwickelt. Dazu haben der Verfasser und der Autor der parallel laufenden Diplomarbeit das für die Beschreibung von Informationsmodellen bewährte erweiterte Entity-Relationship-Modell gewählt. Auch wenn sie häufig als Grundlage für den Entwurf eines relationalen Datenbankschemas dienen, soll durch diese Wahl keine Präferenz für das relationale Datenmodell impliziert sein. Die Kardinalitäten der Relationships sind so zu lesen, dass ein Entity so oft an einer Beziehung teilnimmt, wie es die Kardinalität angibt, die an der Kante zwischen Relationship und dieser Entity-Menge angegeben ist. Damit scheint die Darstellung invertiert zur häufig genutzten Schreibweise, wie sie auch in UML eingesetzt wird. Sie erlaubt jedoch die korrekte und eindeutige Beschreibung der Kardinalitäten von Relationships mit einem Grad größer zwei.

Da das ERM auf Aussagen über Entity- und Relationship-Mengen beschränkt ist, lassen sich Entwurfsentscheidungen nicht immer vollständig aufgrund der Diagramme begründen. Wo es erforderlich ist, werden daher in den das entstehende Schema beschreibenden Texten Be-

züge zu den Instanzen dieser Mengen hergestellt, um die getroffenen Entscheidungen zu untermauern.

3.1 Kurzbeschreibung Siedlungswasserwirtschaft

Die Siedlungswasserwirtschaft ist eine Unterdisziplin des Bauingenieurwesens und beschäftigt sich mit den Aspekten der Wasserversorgung, der Siedlungsentwässerung und der Abwasserbehandlung. Zusätzlich zu den Tätigkeiten eines Bauingenieurs sind dazu auch zunehmend Anforderungen des Umweltschutzes und soziale Randbedingungen zu berücksichtigen.

3.1.1 Reale Experimente

Reale Experimente sind im Bereich Siedlungswasserwirtschaft nur stark eingeschränkt möglich, was sich einerseits aus den hohen Kosten, die beim Aufbau einer Siedlung anfallen, anderseits aus den langen Zeiträumen von der Experimentplanung bis zur Ergebnisgewinnung ergibt. Bei der Planung von virtuellen Experimenten kann daher nicht auf Erfahrungen aus realen Experimenten zurückgegriffen werden.

3.1.2 Anforderungen für virtuelle Experimente

Für den Bereich Siedlungswasserwirtschaft soll ein Softwaresystem entwickelt werden, das mit Hilfe von virtuellen Experimenten die Simulation des urbanen Wasserkreislaufs in einer existierenden Siedlung ermöglicht sowie einen optimalen Bebauungsvorschlag einer Siedlung im Sinne des urbanen Wasserkreislaufes für eine gegebene Bebauungsfläche unter bestimmten Voraussetzungen erstellt.

Virtuelle Experimente werden in drei Phasen durchgeführt: Versuchsaufbau, -durchführung und -analyse.

3.1.3 Versuchsaufbau

Als Vorgabe für ein Experiment dient eine zu bebauende Fläche. Die topographischen Daten dieser Fläche sollen entweder manuell eingegeben oder aus existierenden geographischen Informationssystemen (GIS) importiert werden können. Die Fläche soll mittels eines festzulegenden Rasters in Zellen eingeteilt werden, für die Bodenbeschaffenheitsparameter (beispielsweise Versickerungs- und Verdunstungsrate) erfasst werden. Die topographischen Daten dienen als Ausgangspunkt für alle weiteren Bearbeitungsschritte, sie beschreiben die Siedlung ohne Bebauung. Alle siedlungstechnischen Bebauungen (Straßen, Abwasserkanäle, Liegenschaftsflächen) sollen separat selbst erstellt oder importiert und bearbeitet werden können. Abschließend werden Daten erfasst, die sich auf das gesamte Gebiet beziehen, zum Beispiel Wetterdaten.

3.1.4 Versuchsdurchführung

Für die modellierte Siedlung soll der urbane Wasserkreislauf simuliert werden. Hierzu wird die Siedlung für einen festzulegenden Simulationszeitraum einem Wetterverlauf ausgesetzt.

3.1.5 Versuchsanalyse

Die Ergebnisse der Simulation des Wasserkreislaufs sollen in Relation zum Wetterverlauf dargestellt werden. Hierbei sind Bereiche der Siedlung zu markieren, in denen festzulegende Ereignisse wie die Überflutung einer Fläche oder die Unterdimensionierung eines Abwasserkanals aufgetreten sind. Für jeden Punkt innerhalb der Siedlung sollen zudem die Parameter Verdunstung, Versickerung und Abfluss über den Simulationszeitraum isoliert oder gemeinsam darstellbar sein.

3.2 Anforderungen an ein gemeinsames Informationsmodell

Trotz der augenscheinlich recht gegensätzlichen Anforderungen der beiden virtuellen Labore lassen sich gemeinsame Bestandteile in den Anforderungen an die Datenhaltungskomponente identifizieren. Diese umfassen mehrere Bereiche. Im Folgenden werden diese gemeinsamen Anforderungen zunächst erläutert und einheitliche Bezeichnungen eingeführt. Auf diese kann dann bei der späteren Entwicklung eines gemeinsamen Informationsmodells Bezug genommen werden.

3.2.1 Nutzerverwaltung

Beide virtuellen Labore benötigen eine Verwaltung ihrer Benutzer. Dabei sind nicht nur Authentifizierungs- und möglicherweise Konfigurationsdaten nutzerspezifisch zu verwalten. Vielmehr ist für einzelne Objekte des virtuellen Labors eine Zuordnung zu einem Besitzer erforderlich. Auch eine detaillierte Vergabe von Zugriffsrechten ist zu unterstützen. So können die Besitzer von Objekten anderen Benutzern des Labors diese Objekte wahlweise für einen rein lesenden Zugriff (read only) zugänglich machen oder ihnen auch Schreibrechte ermöglichen (read/write). Als Zwischenstufe kann das Ableiten von eigenen Versionen der Laborobjekte ermöglicht werden (derive). Als Objekte der Zugriffsverwaltung sind Bausteintypen und Experimentaufbauten von Bedeutung. Um die Effizienz bei der Speicherung von Zugriffsrechten zu verbessern und grobgranularere Freigaben zu erlauben, ist eine Rechtevergabe nicht nur für einzelne Nutzer, sondern auch für Nutzergruppen zu unterstützen.

3.2.2 Bausteine

Grundidee des schon in der Beschreibung des Sonderforschungsbereichs skizzierten virtuellen Labors sind die Experimentbausteine, die als Grundlage für den Aufbau eines Experimentes dienen.

3.2.2.1 Bausteintypen und - instanzen

Elementar ist zunächst die Trennung von Bausteintyp und Bausteininstanz. Ein Bausteintyp beschreibt alle wesentlichen Eigenschaften eines Bausteines, die für den Benutzer und für die Simulation relevant sind (analog zum Verständnis des Klassenbegriffs der objektorientierten Programmierung). Beim Aufbau eines Experiments werden Bausteintypen instanziert und damit verwendbar gemacht. Dazu werden ihren variablen Eigenschaften Werte zugewiesen und innerhalb des Experimentaufbaus platziert. Eigenschaften und Platzierung entsprechen gewissermaßen den Instanzvariablen oder Feldern von OO-Klassen.

Neben dieser grundsätzlichen Gemeinsamkeit fallen beim Vergleich der Anforderungen der beiden virtuellen Labore zunächst starke Diskrepanzen bezüglich der Anforderungen an ein Bausteinmodell auf. So kommen im Labor Siedlungswasserwirtschaft eine relativ begrenzte Zahl von Bausteintypen zum Einsatz, deren Eigenschaften und Parameter weitgehend konstant sind und die als Grundlage für die Simulation dienen. Diese Typen entsprechen grundlegenden Siedlungselementen wie Straßen, Grünflächen und Bebauung, die durch Änderung weniger Parameter in weitem Umfang angepasst werden können.

Im Labor Verfahrenstechnik ist dagegen mit einer großen Menge verschiedener Bausteintypen zu rechnen, die zudem von den Benutzern erweitert und ergänzt werden kann. Die Bausteine dieser Domäne sind im Allgemeinen weniger flexibel einsetzbar, zudem stellen die Eigenschaften der Bausteine nur eine abstrahierende Beschreibung der kompletten Bausteintypinformationen dar, die zusätzlich vollständig zu speichern ist.

Bei der Betrachtung der Instanzebene sind die Größenordnungsverhältnisse invertiert. In einem Experimentaufbau für Verfahrenstechnik kommt von den zahlreichen möglichen Bausteintypen im Allgemeinen nur eine begrenzte Zahl zum Einsatz, die Gesamtzahl von genutzten Bausteinen liegt im Bereich von wenigen hundert Instanzen. Dagegen werden zur vollständigen Beschreibung einer Siedlung meist alle vorhandenen Bausteintypen benötigt. Diese werden jedoch häufig in großer Zahl mit identischen Parametern instanziert. Insgesamt ist bereits für Siedlungen mittlerer Größe mit mehreren hunderttausend Bausteininstanzen zu rechnen.

Diese Diskrepanzen in den Größenordnungen erfordern entsprechende Vorkehrungen, um durch die Definition eines gemeinsamen Kerndatenmodells keine allzu einschneidenden Kompromisse eingehen zu müssen, die das Leistungsverhalten für die verschiedenen Domänen verschlechtern.

3.2.2.2 Baustein(typ)eigenschaften

Der Begriff der Bausteineigenschaft umfasst einen weiten Bereich von Informationen über Experimentbausteine. Sie können von verschiedenstem Typ sein. So sind neben numerischen Eigenschaften auch Texte oder größere unstrukturierte Informationen wie Graphiken denkbar. Auch komplexe Eigenschaften, deren Werte selbst weiter strukturiert sind, sind zu unterstützen. Eigenschaften können relevant für die Funktionsweise des Bausteins im späteren Experiment sein (*modellrelevant*) oder lediglich Informationen für den Benutzer bereitstellen (*nutzerrelevant*). Zugleich stellen Eigenschaften den wichtigsten Zugang des Benutzers zu

den Bausteinen dar, da sie eine abstrahierte Sicht auf die Experimentbausteine bieten und zudem die Grundlage für die Suche nach geeigneten Bausteintypen innerhalb der Menge aller verfügbaren Typen sind.

Durch diese große Vielfalt ist ein entsprechend flexibles Modell für die Darstellung von Bausteineigenschaften erforderlich, das einerseits generisch genug ist, um alle nötigen Informationen darstellen zu können, andererseits auch ein akzeptables Leistungsverhalten bietet.

3.2.2.3 Bausteingruppen

Für viele Anwendungen ist es sinnvoll, Bausteininstanzen in Gruppen zusammenzufassen. Dies vereinfacht die Handhabung von ähnlichen Objekten und erlaubt ihre gemeinsame Manipulation. Gruppen sollen dabei ähnlich wie Verzeichnisse eines Dateisystems baumartig ineinander verschachtelt werden können. Die maximale Höhe des Baumes soll nicht durch Begrenzungen des Schemas eingeschränkt werden.

3.2.3 Experimente

Experimente sind die zentralen Objekte eines virtuellen Labors. Sie umfassen einerseits den Experimentaufbau, also die Siedlung oder die Apparatur, als auch die Ergebnisse von Experimentdurchführungen. Zudem enthalten sie Informationen über den Ersteller, den Zeitpunkt der letzten Änderungen, einen Namen sowie eine Version.

3.2.4 Verarbeitungsmodell

Die Arbeitsweise mit den Daten eines virtuellen Labors unterscheidet sich stark von konventionellen datenbankgestützten Anwendungen, bei denen die von den Benutzern durchgeführten Änderungen sofort in eine gemeinsame Datenbasis eingebracht werden und ein hoher Grad an Parallelität bei der Benutzung einzelner Datensätze herrscht.

3.2.4.1 Lange Transaktionen

Die Bearbeitung eines Experimentaufbaus und die anschließende Durchführung des Experiments sind im Allgemeinen lang andauernde Bearbeitungsschritte. Zudem sind eine große Menge von Objekten involviert. Dies hat direkte Folgen für den Grad der Bindung der Benutzerschnittstelle des Systems an die Datenhaltungsschicht. Eine direkte Nachführung von Änderungen innerhalb der Datenhaltungsschicht wäre wegen der großen Objektmengen äußerst ressourcenaufwendig, würde eine permanent hohe Last auf dem verwendeten Datenbanksystem bedeuten und zugleich die Reaktionszeiten des Systems verschlechtern. Zudem wären der Skalierbarkeit des Systems enge Grenzen gesetzt. Zusätzlich erfordert die lange Bearbeitungsdauer mit dem „human in the loop“ ein große Anzahl von langen Sperren innerhalb der Datenbank, sodass die nebenläufige Nutzung stark eingeschränkt wäre. Daher ist statt enger DB-Kopplung und der Verwendung von DB-Transaktionen ein anderer Ansatz erforderlich, um ein ressourcenschonendes und performantes System zu entwickeln.

Man verzichtet auf eine permanente Verbindung zur Datenbank und hält stattdessen alle für die Arbeit erforderlichen Objekte innerhalb eines Objektpuffers nahe beim Benutzer. Dieser Objektpuffer wird zu Beginn der Arbeit mit allen benötigten Objekten gefüllt. Dazu gehört das Experiment selbst und alle daran beteiligten Bausteintypen. Werden anschließend Verarbeitungsschritte durchgeführt, die weitere Objekte erfordern, können diese bei Bedarf in den Objektpuffer nachgeladen werden. Das initiale Füllen des Objektpuffers wird als Check-Out bezeichnet. Neben dem Laden aller Objekte muss dieser Schritt sicherstellen, dass keine Konflikte durch die zeitgleiche Bearbeitung von Objekten durch mehrere Benutzer entstehen. Die in Bearbeitung befindlichen Objekte können nicht oder nur eingeschränkt parallel von anderen Nutzern bearbeitet werden und sind dementsprechend als „benutzt“ zu markieren. Um Konflikte um einzelne Objekte beim Check-Out zu vermeiden, ist diese Operation transaktionsgeschützt auszuführen: Sind zum Zeitpunkt des Check-Out alle benötigten Objekte zugänglich, so wird er durchgeführt. Ist jedoch eine Teilmenge nicht zugänglich, wird der Check-Out als Ganzes verhindert. Dabei soll der Benutzer auf die nicht zugänglichen Objekte hingewiesen werden.

Nach dem Ende der Arbeit kann die bisher lediglich im Objektpuffer modifizierte Objektmenge wieder in die Datenbank eingebracht werden. Dieser Schritt wird als Check-In bezeichnet und muss atomar unter Transaktionsschutz erfolgen. Dabei kann eine Integritätsprüfung der Objekte erfolgen, wie sie im Kapitel 3.2.5 näher erläutert wird. Entsprechen die Änderungen nicht den Erwartungen des Benutzers, können sie alternativ auch verworfen werden.

Da eine Beeinflussung DB-interner Sperrmechanismen im Allgemeinen nicht möglich ist, sind hier anwendungsspezifische Sperrmechanismen zur Realisierung des Check-In/-Out Ansatzes vorzusehen, deren erforderliche Semantik im Anschluss näher erläutert wird. Je nach Objekttyp sind verschiedene Maßnahmen erforderlich, die den Grad der parallelen Arbeit an Objekten mehr oder weniger stark einschränken.

3.2.4.2 Verhinderung paralleler Änderungen an Experimenten

Ein von einem Benutzer mittels Check-Out geladenes Experiment befindet sich normalerweise bei der Bearbeitung in einem nicht-konsistenten Zustand. In der Datenbank befindet es sich dagegen bis zum Check-In im ursprünglichen, unveränderten Zustand. Um Konflikte beim Check-In zu vermeiden, ist grundsätzlich kein mehrfacher Check-Out eines Experiments durch verschiedene Benutzer zuzulassen. Ein mehrfacher Check-Out durch den selben Benutzer ist zuzulassen, um bei Fehlern im Client auf ein als „checked-out“ markiertes Experiment weiterhin zugreifen zu können.

Wie bei der Diskussion der Rechte in Kapitel 3.2.1 bereits erläutert, kann jedoch das Ableiten einer neuen Version eines momentan in Bearbeitung befindlichen Experiments gestattet sein. Ein solches abgeleitetes Experiment erhält eine neue Bezeichnung und/oder eine neue Versionsnummer und kann daher konfliktfrei per Check-In in die Datenbank eingebracht werden.

3.2.4.3 Verhinderung von Änderungen genutzter Bausteintypen

Da Experimente aus Experimentbausteininstanzen zusammengefügt werden, deren Typ unter Umständen mehreren Benutzern zugänglich ist, muss sichergestellt werden, dass ein in einem beliebigen Experiment in Form einer Instanz genutzter Bausteintyp nicht modifiziert werden kann, da dies die Integrität der betroffenen Experimente gefährden könnte. Auch hier kann jedoch die Ableitung einer neuen Version eines Bausteins ermöglicht werden.

3.2.5 Constraints

Wesentlich für jegliche Domäne eines virtuellen Labors sind Möglichkeiten zur Beschreibung der Integrität und Korrektheit von Objektgruppen und einzelnen Objekten. Derartige Korrektheitsdefinitionen oder Randbedingungen werden im Folgenden allgemein als *Constraints* bezeichnet. Aufgrund der großen Unterschiede im Bezug auf die mögliche Komplexität und der erforderlichen Ausdrucksmächtigkeit von Constraints für verschiedene Domänen ist die Entwicklung eines allgemeinen Informationsmodells für Constraints ein in dieser frühen Phase der virtuellen Labore äußerst ambitioniertes Unterfangen. Daher wurde die Beschreibung von Constraints nicht in das Kerninformationsmodell aufgenommen. Jedoch ließen sich Gemeinsamkeiten bezüglich des Wirkungsbereichs von Constraints erkennen, die daher in verschiedene Klassen eingeteilt wurden. So können sie für einzelne Bausteintypen gültig sein oder sich nur auf einzelne Eigenschaften eines Bausteintyps beziehen. Komplexere Constraints können sich auf Gruppen von Bausteininstanzen oder auf einen kompletten Experimentaufbau erstrecken. Andere Constraints haben systemweit Gültigkeit. Um hier eine Vorauswahl von zu überprüfenden Constraints treffen zu können, sollten sie daher aufgrund ihres Wirkungsbereichs vorselektiert werden können. Dadurch lässt sich die Menge der zu überprüfenden Constraints wirkungsvoll reduzieren.

3.2.5.1 Constraints auf Ebene einzelner Eigenschaften

Constraints beschreiben häufig den gültigen Wertebereich einer Bausteineigenschaft unabhängig von den anderen Eigenschaften eines Bausteins. So werden Längenmaße sinnvollerweise nicht negativ sein oder Ober- und Untergrenzen für numerische Werte existieren. Alternativ zur Spezifikation von gültigen Wertebereichen können auch explizit die erlaubten beziehungsweise nicht erlaubten Werte für eine Eigenschaft angegeben werden. Um sie spezifizieren zu können, muss der betroffene Bausteintyp und die betroffene Eigenschaft angegeben werden.

3.2.5.2 Constraints auf Bausteintypebene

Constraints auf einer Bausteintypeigenschaft lassen sich nicht immer losgelöst von den anderen Eigenschaften des Bausteins beschreiben. Sie sind daher auf Ebene des Bausteintyps angesiedelt. Beispielsweise kann der erlaubte Innendurchmesser eines Rohres in Abhängigkeit des Außendurchmessers und der für das verwendete Material erforderlichen Mindestwandstärke beschrieben werden. Für diese Klasse von Constraints sind sowohl der Bausteintyp als auch die betroffenen Eigenschaften zu spezifizieren. Auch die Positionierung eines Baustein-

typs innerhalb eines Experiments kann an Bedingungen geknüpft werden. So dürfen bestimmte Bausteine nur in einem begrenzten Lagebereich eingesetzt werden.

3.2.5.3 Constraints auf Ebene von Bausteingruppen

Um die erlaubten Nutzungsmöglichkeiten von Bausteinen zu definieren, muss die Beschreibung von Constraints auch im Kontext von Bausteingruppen möglich sein. So lassen sich erlaubte Kombinationen von Bausteinen oder Restriktionen für die Eigenschaften eines Bausteintyps in Abhängigkeit von anderen zeitgleich eingesetzten Typen beschreiben. Zur vollständigen Beschreibung sind sowohl alle von einem solchen Constraint betroffenen Bausteine als auch deren Eigenschaften zu referenzieren.

3.2.5.4 Systemweit gültige Constraints

Durch technische oder physikalische Restriktionen sind alle an einem Experiment beteiligten Objekte gewissen Gesetzen unterworfen, die zu jeder Zeit unabhängig vom jeweiligen Nutzungskontext erfüllt sein müssen. Ein einfaches Beispiel für ein derartiges Constraint ist die Unmöglichkeit der räumlichen Überlappung von zwei Objekten.

Die Auseinandersetzung mit dieser Klasse von Constraints wirft die Frage auf, ob alle Constraints in der Datenhaltungsschicht repräsentiert sein müssen, oder ob sie nicht hartkodiert durch die Anwendung zu erzwingen sind. Dadurch kommt man zu einer weiteren möglichen Einteilung von Constraints nach dem Ort ihrer Definition und nach dem Ort der Prüfung.

3.2.5.5 Ort der Definition

Scheint die Erzwingung von systemweit gültigen Constraints auf Ebene der Anwendung bei monolithischen Systemen als natürliche Realisierungsweise, so hat dieser Ansatz jedoch bei der Verwendung eines aus getrennten Komponenten zusammengesetzten Systems erhebliche Nachteile. Sind beispielsweise die Benutzeroberfläche für den Experimentaufbau und die Simulationskomponente getrennt ausgeführt, besteht die Gefahr, dass nicht explizit innerhalb der Datenbank repräsentierte Constraints uneinheitlich durchgesetzt werden. So könnte eine Teilkomponente aus Sicht einer anderen Komponente inkonsistente Daten produzieren. Durch explizite Beschreibung aller Constraints in der Datenhaltungskomponente und genaue Spezifikation der Interpretation dieser Constraints lässt sich dieses Problem vermeiden.

Auf weitere Aspekte bei der Darstellung von Constraints wird bei der Vorstellung des Bausteinmodells in Kapitel 4.3 eingegangen.

3.2.5.6 Ort der Prüfung

Die Prüfung und Durchsetzung der Constraints kann innerhalb der Datenhaltungskomponente stattfinden oder durch die Anwendung realisiert sein. Die Prüfung in der Anwendung hat bei dem vom virtuellen Labor verwendeten Check-In/-Out-Ansatz zur Realisierung lang an-

dauernder Transaktionen den Vorteil, dass der Benutzer unmittelbar auf bestehende Verletzungen von Constraints hingewiesen werden kann. Eine Prüfung nur in der Datenbank würde erst bei einem Check-In erfolgen. Durch die hohe Anzahl von möglichen Fehlern, die im Laufe einer Arbeitssitzung entstehen können, kommt es so beim Check-In zu einer großen Anzahl von Fehlermeldungen, die erst abgearbeitet werden müssen, bevor die Arbeit in das Repository eingebracht werden kann. Dies würde die Akzeptanz des virtuellen Labors bei den Benutzern deutlich verschlechtern. Daher ist eine anwendungsnahe Prüfung sinnvoll, die zusätzlich durch eine Prüfung in der Datenhaltungskomponente abgesichert werden kann, um die Integrität des Repository bei einem nicht-monolithischen System zu garantieren.

3.2.5.7 Allow- und Prohibit-Ansatz

Constraints können durch explizite Beschreibung der erlaubten Zustände definiert werden (Allow) oder durch Angabe der ungültigen Zustände (Prohibit). Je nach Art eines Constraints und dem Grad der von ihm beschriebenen Einschränkungen kann die eine oder die andere Form geeigneter sein. Daher sind beide Möglichkeiten vorzusehen.

3.2.6 Experimentdaten

Bei der Analyse der Anforderungen an ein Informationsmodell für die Speicherung der Ergebnisse von virtuellen Experimenten wurden erhebliche Diskrepanzen zwischen den beiden Domänen entdeckt: Im Bereich Siedlungswasserwirtschaft hat eine Experimentdurchführung einen transformierenden Charakter, der Experimentaufbau (die Siedlung) wird von einem Zustand in einen anderen überführt. Im Bereich Verfahrenstechnik ist nicht der veränderte Zustand der Versuchsapparatur von Interesse, sondern die Durchführung von Messreihen, welche die Zwischenzustände des Versuchs aufzeichnen.

Wegen dieser fundamentalen Unterschiede wurde auf die Entwicklung eines gemeinsamen Informationsmodells für die Experimentdaten verzichtet.

3.3 Domänenunabhängiges Informationsmodell

Nach der Schilderung der gemeinsamen Anforderungen wird im Folgenden der Kern eines Informationsmodells entwickelt, das diese in einer möglichst domänenneutralen Form erfüllt. Die erforderliche Flexibilität bedingt gelegentliche Kompromisse an die Eleganz und Präzision des Entwurfs, erlaubt dadurch jedoch eine effiziente Abbildung aller Aspekte. Um einen sprachlichen Bruch zwischen Entwurf und späterer Implementierung zu vermeiden, werden die Elemente des Informationsmodells mit englischen Bezeichnungen versehen. Bei der Entwicklung des Schemas werden zunächst Teilschemata für Einzelaspekte vorgestellt, die abschließend zu einem Gesamtschema zusammengefügt werden, das in Anhang B zu finden ist. Die Attribute von Entities und Relationships werden nur bei ihrem ersten Vorkommen aufgeführt, oder wenn sie für die betrachteten Aspekte von Bedeutung sind, um die folgenden Diagramme übersichtlicher zu gestalten.

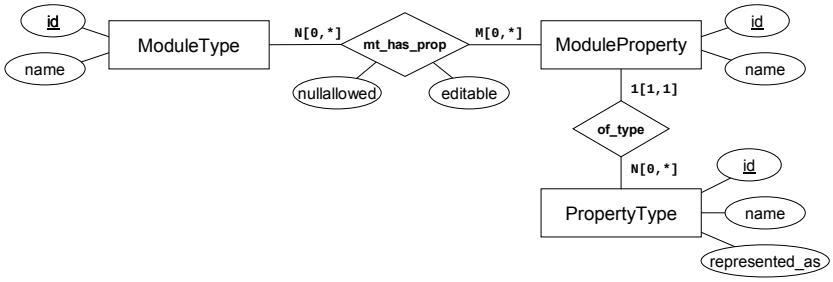
3.3.1 Bausteintypen und ihre Eigenschaften

Der Bausteintyp (*ModuleType*) stellt eines der zentralen Entities des gemeinsamen Entwurfs dar. Neben einfachen beschreibenden Attributen wie einem Namen wurde er zur systemweit eindeutigen Identifizierung mit einer Identifiktationsnummer versehen. Diese kann beispielsweise als Primärschlüssel eines relationalen Schemas verwendet werden. Abbildung 1 zeigt das zugehörige Teilschema.

Bausteineigenschaften (*ModuleProperty*) verfügen ebenfalls über einen Namen und einen systemweit eindeutigen Bezeichner. Zudem haben sie einen Typ (*PropertyType*). Sie wurden bewusst nicht als schwaches Entity von Bausteinen realisiert. Dadurch können verschiedene Bausteintypen über identische Eigenschaften verfügen, wodurch wie gefordert eine Suche innerhalb der Bausteinmenge auf Basis von Eigenschaften und deren Werten möglich ist. Bei einer Verwendung von als schwacher Entitymenge realisierten Eigenschaften müssten derartige Suchoperationen aufgrund von Namens- und Typgleichheit der Eigenschaften durchgeführt werden. Vor allem aber wird so eine redundante Darstellung gleicher Eigenschaften vermieden und ihre unabhängig Verwendung bei späteren Erweiterungen ermöglicht. Die Relationship „ModuleType has ModuleProperty“ (*mt_has_prop*) zwischen ModuleProperty und ModuleType ist folglich vom Typ n:m. Als Beziehungsattribute sind Flags vorgesehen, die angeben, ob die Angabe eines Wertes für die ModuleProperty für den jeweiligen ModuleType optional ist (*nullallowed*) beziehungsweise ob ein Ändern des Wertes der Eigenschaft durch den Benutzer erlaubt ist (*editable*).

Typen von Bausteineigenschaften (*PropertyType*) wurden zur Verbesserung der Erweiterbarkeit um domänenspezifische Eigenschaften wie Einheiten als eigenständiges Entity realisiert. Sie verfügen neben Name und Id über das Feld *represented_as*, das beschreibt, wie Werte des Typs repräsentiert sind, zum Beispiel als ganze Zahl, Gleitkommazahl, Zeichenkette oder als komplexer Typ, dessen Werte eigene Entities sind.

Abbildung 1 Bausteintypen und Eigenschaften



3.3.2 Bausteininstanzen

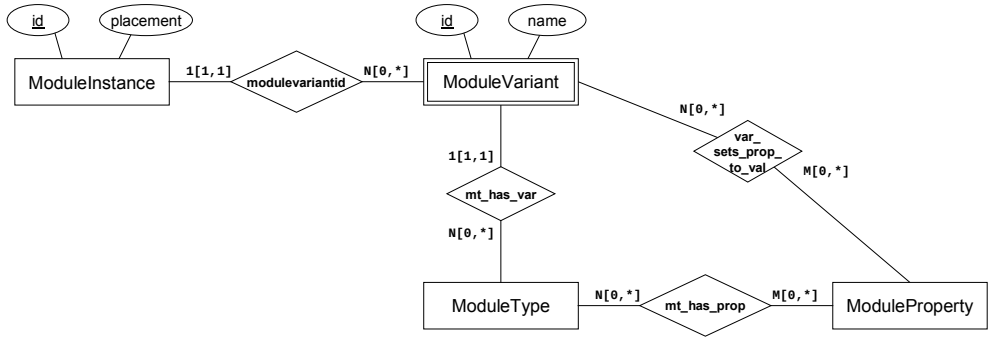
Jeder in einem Experiment verwendete Baustein ist eine Instanz eines Bausteintyps. Um den Typ nutzen zu können, sind zumindest die Eigenschaften, die keinen Null-Wert annehmen dürfen (beschrieben in der Relationship *mt_has_mp*) mit einem Wert zu versehen. Zusätzlich enthalten Bausteine Informationen über ihre Platzierung (*placement*). Die genaue Ausgestal-

tung dieses Attributs ist von der jeweiligen Domäne abhängig und umfasst zum Beispiel Position und Orientierung.

3.3.2.1 Bausteinvarianten

Da es in einzelnen Domänen wie im Labor Siedlungswasserwirtschaft erforderlich sein kann, eine sehr große Zahl von Experimentbausteininstanzen in einem einzigen Experimentaufbau zu unterstützen, die alle gleiche Werte für jede ihrer Eigenschaften aufweisen, wäre es ineffizient, diese identischen Werte für jede Instanz dediziert zu speichern. Um hier eine Platzersparnis zu ermöglichen, werden Bausteininstanzen nicht direkt mit Werten versehen. Stattdessen werden Bausteinvarianten (*ModuleVariant*) eingeführt, denen dann Werte zugewiesen werden. Bausteininstanzen basieren auf einer Variante, was durch die Relationshipmenge *mi_of_var* dargestellt wird. So können für den skizzierten Anwendungsfall identische Instanzen auf die gleiche Variante verweisen. Für den Fall individueller Varianten für jede Instanz, wie dies für das Labor Verfahrenstechnik typisch ist, bedeutet die zusätzliche Indirektion nur einen geringen Effizienzverlust. Zudem lassen sich mit diesem Mechanismus sehr einfach vordefinierte Varianten darstellen, die Bestandteil der Bausteinbibliothek sind. Für diese Form von Varianten ist ein Attribut *Name* vorgesehen. Abbildung 2 zeigt das zugehörige Schema: Jede Bausteininstanz ist über eine Relationship mi_of_var vom Typ 1:n mit ihrer zugehörigen Bausteinvariante verbunden. Bausteinvarianten sind existenzabhängig von ihrem Bausteintyp und daher ebenfalls über eine 1:n-Beziehung mit diesem verbunden. Jede Bausteinvariante weist den Eigenschaften ihres Bausteins Werte zu, was durch die n:m-Beziehung „Variant sets property to value“ (*var_sets_prop_to_val*) dargestellt wird.

Abbildung 2 Bausteininstanzen und -varianten



3.3.2.2 Typvielfalt

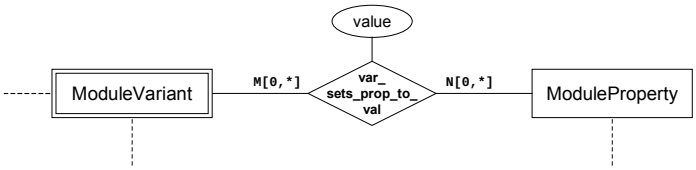
Werte von Bausteineigenschaften können von verschiedenem Typ sein. So gibt es einfache Typen wie numerische Werte, die ganzzahlig oder rational sein können oder Eigenschaften mit textuellen Werten. Auch komplexe Werte, die selbst ihrerseits aus einfachen Typen zusammengesetzt sein können, müssen effizient unterstützt werden. Da die Anforderungen an komplexe Typen in den Domänen stark unterschiedlich sein können, werden sie daher im folgenden nicht näher betrachtet.

Um die Speicherung verschiedener Typen zu ermöglichen, sind verschiedene Schemata denkbar. Einige mögliche Lösungsansätze sollen im Folgenden kurz vorgestellt werden. Eine Bewertung nur auf Grund der Schemabeschreibung als ERM ist nicht vollständig zu leisten, weshalb die Lösungsmöglichkeiten bezüglich ihrer Vor- und Nachteile bei Verwendung einer rein relationalen Darstellung bewertet werden.

Abbildung auf einen Typ

Um die Speicherung von verschiedenen Typen zu ermöglichen, könnte eine Abbildung auf einen einzigen Typ vorgenommen werden. Hierzu würde sich die Verwendung einer Repräsentation durch eine Zeichenkette (String) anbieten. So gäbe es im relationalen Schema nur eine einzige Tabelle für Werte, sodass sowohl das Laden von Property-Werten effizient möglich ist als auch referentielle Integrität genutzt werden kann. Nachteilig ist zum einen die erforderliche Konvertierung von und zur String-Darstellung, die zur Verbesserung von komplexen Suchanfragen durch geeignete Konvertierungsfunktionen in Form von *User Defined Functions* (UDFs) in der Datenbank zu unterstützen ist. Zudem ist das Verfahren speicherineffizient und problematisch, wenn auch die Speicherung großer Datentypen ermöglicht werden soll. Hierzu müssten Large Objects (zum Beispiel CLOBs) eingesetzt werden, die jedoch in vielen Datenbanksystemen nicht von Mechanismen zur Leistungsverbesserung erfasst werden. Dadurch käme es zu Leistungseinbußen auch für kleine Datentypen, was eine sinnvolle Nutzung dieses Lösungsansatzes ausschließt. Abbildung 3 zeigt das Teilschema für diese Art der Darstellung von Eigenschaftswerten. Die Relationship var_sets_prop_to_val enthält ein einzelnes Attribut value, das von einem Typ zur Speicherung von Zeichenketten ist.

Abbildung 3 Abbildung aller Werte auf einen Typ



Hierarchie von Werteentities

Eine alternative Darstellungsform nutzt Vererbung zur Speicherung verschiedener Typen. Ein Entity *Value* repräsentiert die Basisklasse, von der alle Werte-Entities für die verschiedenen zu unterstützenden Typen erben. Jedes Entity hat zusätzlich zu dem von Value geerbten Id-Attribut eine value-Attribut, das von einem jeweils geeigneten Typ ist. Abbildung 4 zeigt die entsprechende Typhierarchie. Auf diese Weise vermeidet man die ineffiziente Konversion und den erhöhten Platzverbrauch bei der Speicherung.

Erscheint diese Lösungsmöglichkeit wegen ihrer augenscheinlichen Eleganz als sehr vorteilhaft, so erfordert ihre Abbildung auf das Relationenmodell erhebliche Zugeständnisse. Nutzt man zur Abbildung der Vererbung das Hausklassenmodell, bei dem jedes Entity nur in ihrer Haus-Relation und nicht in den über ihr in der Hierarchie liegenden Relationen repräsentiert ist, so ist die abgebildete ternäre Beziehung var_sets_prop_to_val nicht durch ein referentiel-

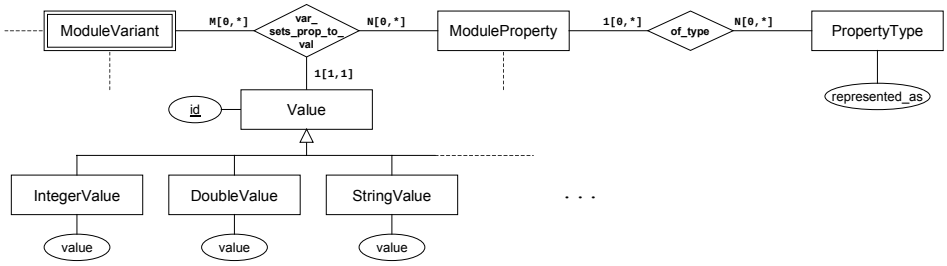
les Constraint zur Relation Value im DB-Schema abzubilden. Bei einer Anfrage müssten alle Relationen der Hierarchie vereinigt werden, um das Ergebnis aufzufinden. Da jedoch die value-Attribute der verschiedenen Relationen nicht vereinigungsverträglich sind, müsste entweder für jede Value-Relation eine separate Anfrage formuliert werden, was mit wachsender Anzahl von Typen einen stark zunehmenden Aufwand bedeuten würde, oder die Superrelation könnte zusätzlich ein Typ-Attribut erhalten, sodass eine Vereinigung aller Relationen der Hierarchie über die Attribute Id und Typ möglich wäre. Mit den so gefundenen Typinformationen könnte dann auf die Value-Tabelle für diesen Typ zugegriffen werden. Als naheliegende Verbesserung können an anderer Stelle im Schema Informationen über die für eine Property zu nutzenden Werte-Tabelle abgelegt werden. Hier bietet sich wie in Abbildung 4 gezeigt das Attribut represented_as der Entity PropertyType an. Mag diese optimierte Lösung für einfache Experimentaufbauten akzeptabel erscheinen, hat sie jedoch gravierende Nachteile bei größeren Mengen von Instanzen. So ist auf jeden Fall für jede Property, deren Wert geladen werden soll, eine zweite Anfrage erforderlich: erst nach dem Ermitteln des Typs, was noch für alle zu ladenden Properties in einer einzigen Anfrage geschehen kann, muss mit der zweiten Anfrage auf die jeweilige Wertetabelle der Wert geladen werden. Fällt dies bei wenigen zu ladenden Werten nicht ins Gewicht, kann der hohe Kommunikationsaufwand mit der Datenhaltungsschicht bei großen Experimentaufbauten jedoch Leistungsfähigkeit und Skalierbarkeit des Systems beeinträchtigen.

Auch andere Abbildungsformen auf das Relationenmodell lösen dieses Problem nicht: Bei Verwendung der „vollen Redundanz“, die ein Entity in jeder Relation auf dem Pfad bis zur Wurzel speichert, kann zwar die referentielle Integrität vom Datenbanksystem sichergestellt werden. Es bleibt jedoch das Problem der fehlenden Vereinigungsverträglichkeit und somit die Notwendigkeit für eine zweite Anfrage pro Wert. „Vertikale Partitionierung“ führt wegen der Einfachheit der Entites der Hierarchie zum gleichen Relationenmodell wie volle Redundanz.

Eine Möglichkeit, bei dieser Modellierung mit einer einzigen Query für alle Properties und deren Werte auszukommen, ist die Verwendung von *outer joins* mit allen Werterelationen. Die so entstehenden Anfragen würden jedoch mit jeder weiteren Werteentity komplexer und die erzielbare Leistung ist in diesem Fall stark von den Optimierungsfähigkeiten des jeweiligen Datenbanksystems abhängig.

Bei der Nutzung von objekt-relationalen Erweiterungen zur Abbildung der Vererbungshierarchie ist in Abhängigkeit von den OR-Fähigkeiten des Datenbanksystems eine effizientere Verarbeitung möglich. Eine allgemeingültige Beurteilung einer solchen Lösung ist wegen der erheblichen Unterschiede zwischen diesen Systemen, auf die in Kapitel 4.2 näher eingegangen wird, hier nicht zu leisten.

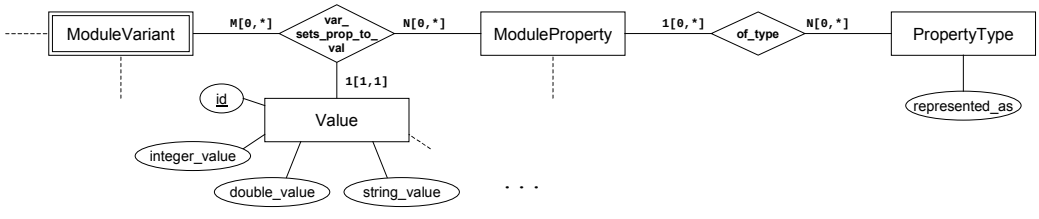
Abbildung 4 Werte in dedizierten Tabellen



Mehrere value-Attribute

Statt eigener Entities für die verschiedenen Typen kann ein Entity oder eine Relationship mit eigenen Attributen für jeden Typ verwendet werden. Nachteilig an diesem Ansatz ist die schlechte Speichereffizienz aufgrund der großen Menge von Nullwerten bei einer solchen Darstellung. Jedes Tupel in der relationalen Abbildung dieses Schemas würde von allen Wert-Attributen jeweils nur eines nutzen. Durch geeignete Wahl der Datentypen lässt sich der damit verbundene Overhead jedoch reduzieren. Während Standarddatentypen oft auch dann den vollen Platz belegen, wenn sie nullwertig sind, werden größere Datentypen wie Large Objects aber auch Long Varchars vom Datenbanksystem intern lediglich referenziert gespeichert. Sie belegen als Nullwerte also keinen Platz innerhalb des Tupels über die Referenz hinaus. Werte vom Typ Varchar belegen lediglich so viel Platz, wie für den jeweiligen Inhalt erforderlich ist, sind jedoch als Standarddatentyp für textuelle Eigenschaften wegen der meist zu geringen Maximallänge nicht geeignet. Kann die Anzahl der nötigen Typen gering gehalten werden, lässt sich durch Ausnutzung dieses Wissens eine inakzeptable Verschwendung von Speicherplatz vermeiden. Gleichzeitig gewinnt man den Vorteil echter referentieller Integrität und kann sämtliche Werte von Properties mit einer einzigen Anfrage auswerten. Die Identifikation des zu verwendenden Werteattributs kann bei der Auswertung der Ergebnismenge geschehen. Abbildung 5 zeigt das zugehörige Schema für diese Lösungsvariante, die Entity-Menge *Value* verfügt über ein dediziertes Attribut für jeden benötigten Typ.

Abbildung 5 Mehrere value-Attribute



Wahl des Schemas zur Wertedarstellung

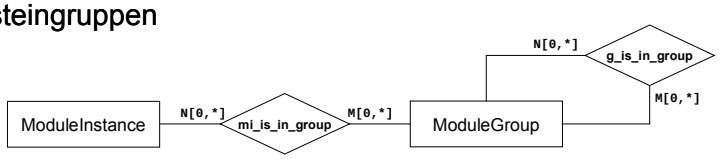
Wegen der beschriebenen Nachteile der ersten beiden Lösungsvarianten insbesondere bei einer großen Anzahl von Bausteinen (und damit auch Eigenschaftswerten) haben sich Alexander Hilliger von Thile und der Verfasser für die letzte Lösungsvariante entschieden. Sie er-

laubt eine gute Leistung, bietet Raum für die Erweiterung mit domänenspezifischen Typen und ihre Nachteile bezüglich der Speicherökonomie lassen sich durch die beschriebene Ausnutzung von Informationen über die Speicherung von großen Datentypen reduzieren. Die in der zweiten Variante mögliche Optimierung durch outer joins oder die Nutzung von OR-Erweiterungen sollte jedoch für das jeweils gewählte Datenbanksystem geprüft werden. Eine Umstellung ist zwischen diesen beiden Lösungen problemlos möglich, solange keine bestehenden großen Datenmengen zu migrieren sind. Durch geeignete Formulierung der Anfragen beschränken sich die Änderungen auf die Anfragen und das Schema, sonstiger Anwendungscode muss nicht modifiziert werden.

3.3.2.3 Bausteingruppen

Um die in den gemeinsamen Anforderungen geschilderte Bildung von Gruppen mit beliebiger baumartiger Schachtelungstiefe zu erlauben, werden Gruppen von Bausteinen durch die Entity *ModuleGroup* repräsentiert. Die Gruppenzugehörigkeit einer Bausteininstanz wird über die Relationship „ModuleInstance is in ModuleGroup“ (*mi_is_in_group*) ausgedrückt. Sie ist trotz der im Allgemeinen baumartigen Struktur der Gruppenhierarchie vom Typ n:m, um die Mehrgruppenmitgliedschaft von Bausteininstanzen zu ermöglichen. Auch die Schachtelung der Gruppen ist durch die Relationship „Group is in group“ (*g_is_in_group*) vom Typ n:m nicht auf Baumstrukturen eingeschränkt. Dadurch lässt sich dieser in Abbildung 6 gezeigte Teil des Informationsmodells auch zur Abbildung von Graphen für andere Domänen nutzen.

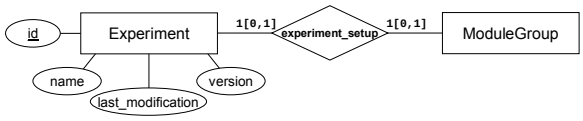
Abbildung 6 Bausteingruppen



3.3.3 Experimente

Ein Experiment ist durch das eigenständige Entity *Experiment* im System repräsentiert. Es enthält Informationen wie Name, Versionsnummer und Zeitpunkt der letzten Änderung. Ein Experiment hat einen Experimentaufbau, bei dem es sich um eine ausgezeichnete Bausteingruppe handelt. Diese bildet die Wurzel der Gruppenhierarchie aller Bausteininstanzen eines Experiments. Die Relationship *experiment_setup* verbindet Experiment und Experimentaufbau. Sie ist vom Typ 1:1, da jedes Experiment nur über einen Experimentaufbau verfügt, und keine Gruppe in mehreren Experimenten als Experimentaufbau eingesetzt werden kann. Abbildung 7 zeigt das entsprechende Teilschema.

Abbildung 7 Experimente



3.3.4 Nutzerverwaltung

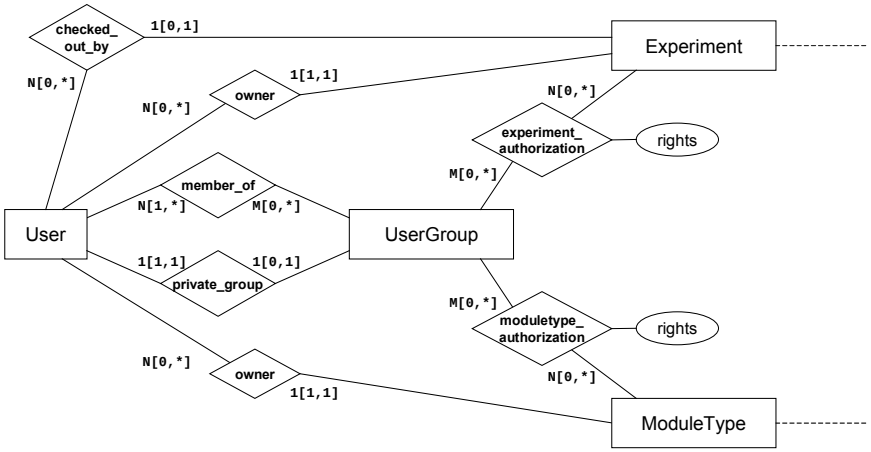
Um die Verwaltung von Nutzerinformationen zu ermöglichen, ist zunächst ein *User*-Entity vorzusehen, das neben den Login-Daten (Login-Name und Passwort) auch weitere möglicherweise domänenspezifische Erweiterungen mit nutzerbezogenen Informationen enthalten kann. Wie in der Beschreibung der Anforderungen an die Nutzerverwaltung gefordert, sollen Freigaben nicht nur auf Ebene einzelner Nutzer, sondern auch auf Ebene von Nutzergruppen stattfinden. Diese sind als eigenständiges Entity *UserGroup* mit den Attributen *Id* und *Name* repräsentiert. Ein Nutzer kann dabei Mitglied in einer beliebigen Menge von Gruppen sein, die Relationship *member_of* ist daher vom Typ n:m. Die Entities User und UserGroup können Rechte an Objekten erhalten und werden daher auch als Objekte der Rechtezuteilung bezeichnet. Ihnen gegenüber stehen die Objekte der Rechteverwaltung, also solche Objekte, für die Rechte vergeben werden können. In den gemeinsamen Anforderungen von Siedlungswasserwirtschaft und Verfahrenstechnik sind dies Experimente und Bausteintypen. Rechte werden durch die Relationship *authorization* ausgedrückt. Sie verfügt über das Beziehungsattribut *rights*, das die Art der Rechte, hier also lesen (read only), lesen und ableiten (derive) und lesen, ableiten und schreiben (read/write) beschreibt. Um die Rechte auf Basis von Nutzern und Nutzergruppen beschreiben zu können, sind in einem geradlinigen Ansatz dedizierte authorization-Relationships zwischen jedem der Entitypaare User - ModuleType, User - Experiment, UserGroup - ModuleType und UserGroup - Experiment erforderlich. Sie sind vom Typ n:m.

Bei dieser Modellierung ergibt sich die Zahl der notwendigen Relationships aus dem Produkt der Kardinalitäten der Mengen der Objekte der Rechtezuteilung und der Rechteverwaltung. Werden diese Mengen erweitert, nimmt dementsprechend die Anzahl notwendiger Relationships zu. Da es sich um n:m-Beziehungen handelt, bedeutet jede neue Relationship eine weitere Relation. Um diese starke Zunahme zu vermeiden und das Schema übersichtlicher zu gestalten, wird als Hilfsmittel die Relationship *private_group* zwischen User und UserGroup eingeführt. Sie beschreibt das Vorhandensein einer ausgezeichneten UserGroup für jeden User, in der nur er Mitglied ist, private_group ist daher vom Typ 1:1. Soll nun einem Nutzer ein Recht an einem Objekt der Rechteverwaltung zugeordnet werden, kann dazu nun die Relationship-Menge zwischen dem Objekt und UserGroup verwendet werden.

In unserem Schema reduziert sich die Menge der n:m-Relationships auf zwei, zwischen UserGroup und ModuleType sowie zwischen UserGroup und Experiment. Der Preis für diese Ersparnis ist eine weitere 1:1-Beziehung zwischen User und UserGroup, die jedoch bei Abbildung auf das Relationenmodell durch Attribute der beiden beteiligten Relationen dargestellt wird.

Neben der Beschreibung der Rechte an Objekten der Rechteverwaltung sind diese auch einem Benutzer als Eigentümer zugeordnet. Dies erfolgt durch 1:n-Beziehungen *owner* zwischen den Entities User und Experiment beziehungsweise ModuleType. Alternativ kann auch hier eine Relationship zu UserGroup genutzt werden. Dann muss jedoch durch ein geeignetes Constraint sichergestellt werden, dass diese Beziehungen nur zu solchen Elementen der Entitymenge UserGroup besteht, die eine private_group eines Benutzers sind.

Abbildung 8 Nutzer- und Rechteverwaltung



3.3.5 Check-In/-Out

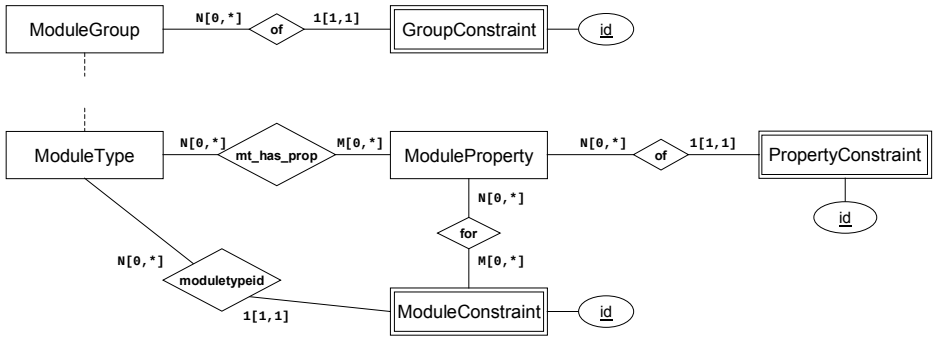
Um die nach einem Check-Out in Bearbeitung befindlichen Experimente zu kennzeichnen, genügt zunächst ein einfaches Flag-Attribut im Experiment-Datensatz. Dadurch kann ein unerwünschter weiterer Check-Out erkannt und verhindert werden. Kommt es jedoch zu einem Crash eines Clients, bevor das Experiment wieder per Check-In in die Datenbank eingebracht wurde, bliebe das Flag gesetzt und das Experiment wäre nicht mehr zugänglich, ohne einen Eingriff durch einen Administrator zu erfordern. Daher wird nicht nur die Information gespeichert, dass ein Experiment per Check-Out bearbeitet wird, sondern auch von wem es ausgecheckt wurde. Dieser Benutzer kann dann jederzeit das Experiment erneut per Check-Out laden, wird aber darauf hingewiesen, dass es zu Problemen kommen kann, wenn mehrere Clients das Experiment in ihrem Speicher halten. Der Zustand *checked-out* wird als weitere Relationship-Menge zwischen Experiment und User realisiert.

Um die Modifikation von in Benutzung befindlichen Bausteintypen zu verhindern, kann auf bereits bestehende Relationship-Mengen zurückgegriffen werden. Existiert eine Instanz der Relationship-Menge mi_of_var (siehe Abschnitt 3.3.2.1) für eine Variante des Bausteintyps, wird dieser Bausteintyp genutzt und darf daher nicht modifziert werden. Die Ableitung einer neuen Version basierend auf einem solchen Bausteintyp ist bei Besitz entsprechender Rechte jedoch zulässig.

3.3.6 Constraints

Wie bereits in Abschnitt 3.2.5 begründet lässt sich in dieser frühen Phase des virtuellen Labors kein allgemeines Informationsmodell für Constraints definieren. Allerdings lassen sich den einzelnen Objekten des virtuellen Labors, für die durch die Laborbenutzer Constraints definiert werden können, entsprechende Entities zuordnen. Die genaue Ausgestaltung der Constraint-Entities bleibt jedoch der jeweiligen Domäne überlassen.

Abbildung 9 Constraints



Für die drei in den Anforderungen identifizierten Bezugsobjekte von Constraints, einzelne Eigenschaften, Bausteintypen oder Gruppen von Bausteinen, wird jeweils eine existenzabhängige Constraint-Entity eingeführt, die mit einer n :1-Beziehung mit dem Bezugsobjekt verbunden ist. Zusätzlich kann für Constraints auf Bausteintyp-Ebene noch angegeben werden, auf welche der Eigenschaften sie sich bezieht, dies wird durch eine n:m-Beziehung zwischen ModuleConstraint und ModuleProperty ausgedrückt. Abbildung 9 zeigt dieses Grundgerüst eines Informationsmodells für Constraints.

3.3.7 Fazit - Informationsmodell

Das vorgestellte Informationsmodell deckt einen Kern von gemeinsamen Anforderungen an die Datenhaltungsschicht eines virtuellen Labors ab. Die Kompromisse, die für die Domänenunabhängigkeit einzugehen waren, schränken weder die Leistungsfähigkeit noch die Ausdrucksmächtigkeit für eine der Domänen zu sehr ein. Für Anforderungsbereiche, über die aufgrund des frühen Stadiums noch keine detaillierte Informationsmodellierung möglich war, wurde wie bei den Constraints eine Grundstruktur vorgegeben.

In einem nächsten Schritt sind neben der Ergänzung domänenspezifischer Erweiterungen verschiedene Abbildungsmöglichkeiten des Informationsmodells auf unterschiedliche Datenbanksysteme zu prüfen.

Kapitel 4 Realisierungs-möglichkeiten

Die in Kapitel 2 vorgestellten Anforderungen an ein virtuelles Labor eröffnen dem Entwickler eines solchen Systems trotz teilweise recht restriktiver Vorgaben einen großen Raum möglicher Lösungsansätze.

Im Folgenden sollen wesentliche Anforderungen an die Datenhaltungskomponente herausgegriffen und anschließend einige Realisierungsmöglichkeiten für jedes dieser Teilprobleme vorgestellt und bezüglich ihrer Eignung bewertet werden. Die Anforderungen an die Datenhaltungskomponente sind vielschichtig:

Es ist ein Informationsmodell zu entwickeln, das die Speicherung der in den verschiedenen Phasen des virtuellen Labors auftretenden Daten erlaubt. Für die Repräsentation und Speicherung des Experimentaufbaus sind hierzu insbesondere die exakte Beschreibung der Experimentbausteintypen und die Möglichkeiten ihrer Verwendung innerhalb eines Experiments von Bedeutung. Dabei werden die in Kapitel 3 ermittelten gemeinsamen Anforderungen und der dort entwickelte gemeinsame Kern eines Informationsmodells berücksichtigt und wo erforderlich erweitert. Ein zweiter wichtiger Bereich ist die Speicherung der während der Durchführung eines virtuellen Experiments anfallenden Experimentdaten, die wegen der großen Vielfalt bezüglich Dimension, Typen und Einheiten besondere Vorkehrungen erfordern.

Vor die Entwicklung dieses Informationsmodells sind Überlegungen hinsichtlich der Architektur des Systems und die Wahl eines geeigneten Datenhaltungssystems zu stellen, die beide einen entscheidenden Einfluss auf das Informationsmodell ausüben können.

Wegen des begrenzten Umfangs dieser Arbeit und des frühen Stadiums des Projekts erfolgt eine Konzentration auf die Anforderungen an die Experimentvorbereitung, deren Ergebnisse (Experimentaufbauten oder „virtuellen Apparaturen“) als Eingabe für die Experimentdurchführung dienen.

4.1 Architekturvorschlag

Neben der Entwicklung eines Informationsmodells, das alle Anforderungen des virtuellen Labors erfüllt, ist auch die Softwarearchitektur des gesamten Systems ein entscheidender Faktor für seine Leistungsfähigkeit und Flexibilität. Der Begriff der Architektur wird in der Informatik an vielen Stellen verwendet, eine einheitliche Definition existiert jedoch nicht. [CMU1] nennt allein über 100 Definitionen des Begriffs.

Im Rahmen dieser Arbeit soll unter Softwarearchitektur ein Modell eines Softwaresystems verstanden werden, das seine Komponenten und deren Interaktion in einem bestimmten Abstraktionsgrad beschreibt.

Softwarearchitekturen beruhen fast immer auf Grundmustern, die sich für den jeweiligen Typ von Anwendung bewährt haben. Transformierende Anwendungen, die große Datenmengen nach vorgegebenen Regeln ohne Nutzerinteraktion verarbeiten, verfolgen einen filterbasierten Ansatz, bei dem die Daten verschiedene Stufen oder Verarbeitungsschritte in einer meist weitgehend festen Reihenfolge durchlaufen. Für interaktive Anwendungen mit Anforderungen wie die Unterstützung von kooperativem Arbeiten durch mehrere Benutzer, einer großen gemeinsamen Datenbasis und der Forderung nach hoher Performanz haben sich dagegen schichtenbasierte Architekturen als geeignet erwiesen. Hier unterscheidet man zumindest die drei „klassischen“ Schichten Präsentation, Anwendungslogik und Datenhaltung.

4.1.1 Unterscheidung Layer - Tier

Wie eingangs erwähnt folgen interaktive Anwendungen, zu denen das in den Anforderungen beschrieben virtuelle Labor zu zählen ist, im Allgemeinen einem schichtenbasierten Ansatz.

Der Begriff der Schicht wird dabei in der englischsprachigen Fachliteratur weiter unterschieden in *Layer* und *Tier*. Auch diese Begriffe sind nicht einheitlich belegt oder werden gelegentlich synonym verwendet. Es hat sich jedoch eine allgemein akzeptierte Bedeutung etabliert:

Während man unter Layern Schichten im Bezug auf Software versteht, also durch wohldefinierte Schnittstellen gekapselte Module, die von übergeordneten Schichten im Sinne einer virtuellen Maschine ohne Wissen über die Implementierungsdetails genutzt werden können, bezeichnet der Begriff Tier Hardware-Schichten, also Schichten, die durch Systemgrenzen abgeteilt werden. Eine allgemeinere Definition versteht unter Tiers Schichten, die durch Prozessgrenzen getrennt sind. Gemäß dieses Verständnisses des Begriffs Tier würde beispielsweise ein einfaches Web-Informationssystem, bei dem Datenbank und Webserver auf einem System, aber in verschiedenen Betriebssystemprozessen angesiedelt sind, als 2-Tier-Architektur bezeichnet.

So kann ein Softwaresystem aus einer großen Zahl von Layern bestehen, während es zugleich in einem einzelnen Tier angesiedelt ist. Ein Beispiel hierfür sind klassische workstationbasierte Ingenieuranwendungen wie CAD-Software. Umgekehrt muss ein

Softwaresystem über mindestens so viele Layer wie Tiers verfügen. Der Begriff des Layers ist insbesondere im Bezug auf sein Granulat nicht genau definiert. Je nach Abstraktionsgrad wird man einzelne Komponenten in Layer trennen oder auch zusammengefasst als einen Layer betrachten.

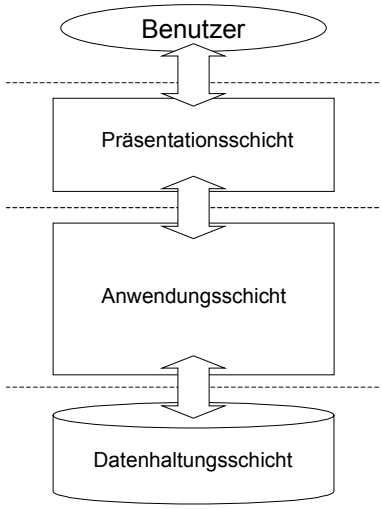
Die folgende Entwicklung eines geeigneten Schichtenmodells für das virtuelle Labor soll sich auf die Grobarchitektur beschränken. Dabei werden noch keine Aussagen über die Zuordnung von Layern zu Systemen beziehungsweise Prozessen gemacht, sodass im Folgenden der Begriff Schicht synonym mit „Layer“ verwendet wird. Bei der Implementierung des Prototypen wird eine feinere Unterteilung der Layer vorgenommen und zudem eine Zuordnung der Layer zu Tiers erfolgen.

4.1.2 Entwicklung der Systemarchitektur

Bei der Aufteilung der Funktionalität des virtuellen Labors in Layer sind neben den allgemeinen Qualitätskriterien bei der Entwicklung von Software wie Wart-, Skalier- und Erweiterbarkeit die besonderen Anforderungen des virtuellen Labors zu beachten.

Statt der üblichen Vorstellung verschiedener Architekturalternativen und der Diskussion ihrer Vor- und Nachteile verfolgt der Verfasser einen Ansatz, der orientiert an der klassischen Dreischichtenarchitektur die Abtrennung einzelner Komponenten durch Bezug auf die Systemanforderungen begründet.

Abbildung 10 Ausgangspunkt: das klassische Dreischichtenmodell



4.1.2.1 Kooperatives Arbeiten

Essentielles Element des virtuellen Labors ist das kooperative Bearbeiten von Aufgaben auf gemeinsamen Daten durch die Benutzer des Systems. Während ein direktes Zusammenarbeiten am Rechner über möglicherweise große Entfernungen für spätere Entwicklungsstufen des virtuellen Labors sinnvoll sein kann, soll zunächst der einzelne Benutzer auf die Experimente, Experimentbausteine und Ergebnisse seiner Kollegen Zugriff erhalten und sie für sei-

ne eigene Arbeit nutzen können. Die Kooperation kann also über den Zugriff auf eine gemeinsame Datenbasis erfolgen. Diese wird daher als unterer Layer des Systems realisiert, auf den verschiedenen Instanzen des virtuellen Labors zugreifen können, um so Kooperation zu ermöglichen. Die Wahl der Ausprägung der Datenhaltungsschicht bestimmt dabei erheblich die Gestaltung der oberhalb liegenden Schichten. Aufgrund dieser Bedeutung beschäftigt sich Kapitel 4.2 im Anschluss näher mit möglichen Realisierungen der Datenhaltungsschicht.

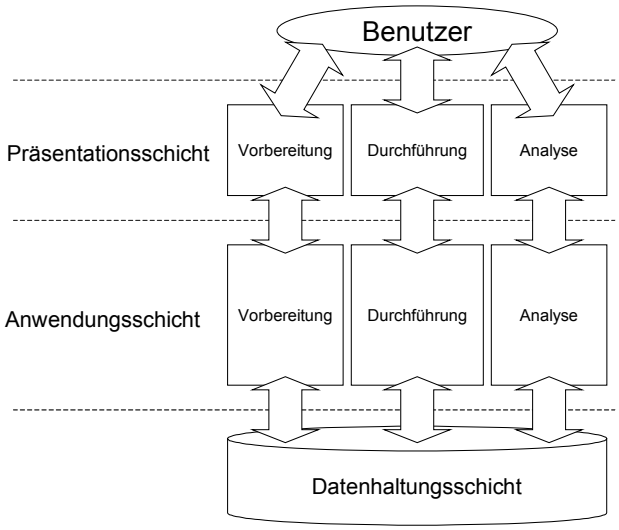
4.1.2.2 Phasen der Experimentdurchführung

Das Konzept, die reale Durchführung eines Experiments im virtuellen Labor nachzubilden, ist ein wesentlicher Bestandteil, um die Akzeptanz des Systems durch die Benutzer unter Beibehaltung ihrer bisherigen Arbeitsweise zu fördern. So sind die drei Hauptphasen des realen Experiments, die Experimentvorbereitung, die Durchführung und die anschließende Auswertung, vom System geeignet zu unterstützen.

Die Anforderungen an Präsentation, Nutzerschnittstelle und Applikationslogik unterscheiden sich zwischen diesen drei Phasen erheblich. Die Experimentvorbereitung ähnelt konzeptionell einer Ingenieursanwendung, die Durchführung ist dagegen durch begrenzte Interaktions- aber weitreichende Visualisierungsmöglichkeiten des laufenden Experiments geprägt. In der Analysephase kommt zu den Aspekten der Visualisierung der Umgang mit und die Weiterverarbeitung von großen Mengen von Experimentdaten hinzu.

Um diesen Unterschieden gerecht zu werden, erscheint eine horizontale Unterteilung der Präsentations- und Anwendungsschichten entsprechend der drei Phasen wie in Abbildung 11 gezeigt sinnvoll. Neben den konzeptionellen Unterschieden setzen insbesondere die Präsentationskomponenten der drei Phasen unterschiedliche technologische Schwerpunkte, sodass eine Realisierung durch Arbeitsgruppen verschiedener Fachgebiete erforderlich ist. Durch eine klare Trennung kann der Kommunikationsaufwand zwischen diesen Teams minimiert werden.

Abbildung 11 Horizontale Trennung von Präsentations- und Anwendungsschicht



4.1.2.3 Datenaustausch zwischen den Phasen

Jede Phase basiert auf den Daten, die in der zuvor durchlaufenen Phase entstanden sind: Die Experimentdurchführung benötigt die Beschreibung des Experimentaufbaus und die Auswertung erfordert die Ergebnisse der Durchführung. Die Kopplung der Phasen kann wie die Kooperation über die Datenhaltungsschicht erfolgen. Hierbei wäre ein direkter Zugriff der Komponenten der oberen Schicht auf die Datenhaltungsschicht mit deren jeweiligen Mechanismen denkbar. Auf ein (objekt-)relationales Datenbanksystem könnte mittels SQL-Schnittstelle zugegriffen werden, eine XML-Datenbank würde möglicherweise mit XQuery angesprochen. Erscheint diese Vorgehensweise insbesondere im Bezug auf die Leistungsfähigkeit vorteilhaft, man denke dazu insbesondere an die in Kapitel 3 beschriebenen Anforderungen bezüglich der zu verarbeitenden Datenmengen, so weist dieser Ansatz jedoch Probleme auf:

So ist zu berücksichtigen, dass je nach Realisierung der Datenhaltungsschicht eine große Diskrepanz zwischen dem dort verwendeten Datenmodell und dem der Anwendung vorliegt. Daher ist eine mehr oder weniger aufwendige Abbildung der Strukturen der Datenhaltung auf die Strukturen innerhalb der Anwendung erforderlich. Wird diese von jeder der Komponenten für die drei Phasen eigenständig realisiert, vergrößert dies nicht nur den Aufwand für deren Entwicklung. Darüber hinaus besteht die Gefahr, dass einzelne Aspekte der Daten von den verschiedenen Komponenten uneinheitlich interpretiert werden, und es so zu Inkonsistenzen bei der Kopplung kommen kann. Zwar können Datenhaltungssysteme, insbesondere die relationalen Datenbanken, selbst die Integrität der von ihnen verwalteten Daten garantieren. Derartige Mechanismen arbeiten dabei jedoch auf den Rohdaten, also mit den im Zuge der zuvor erwähnten Abbildung unter Umständen mehr oder weniger stark zerlegten Anwendungsobjekten. Je nach Grad der Dekomposition und Mächtigkeit der Mechanismen zur Definition von Integritätsbedingungen ist deren Formulierung schwierig, ineffizient und bisweilen schlicht nicht realisierbar.

4.1.2.4 Einheitliches Objektmodell

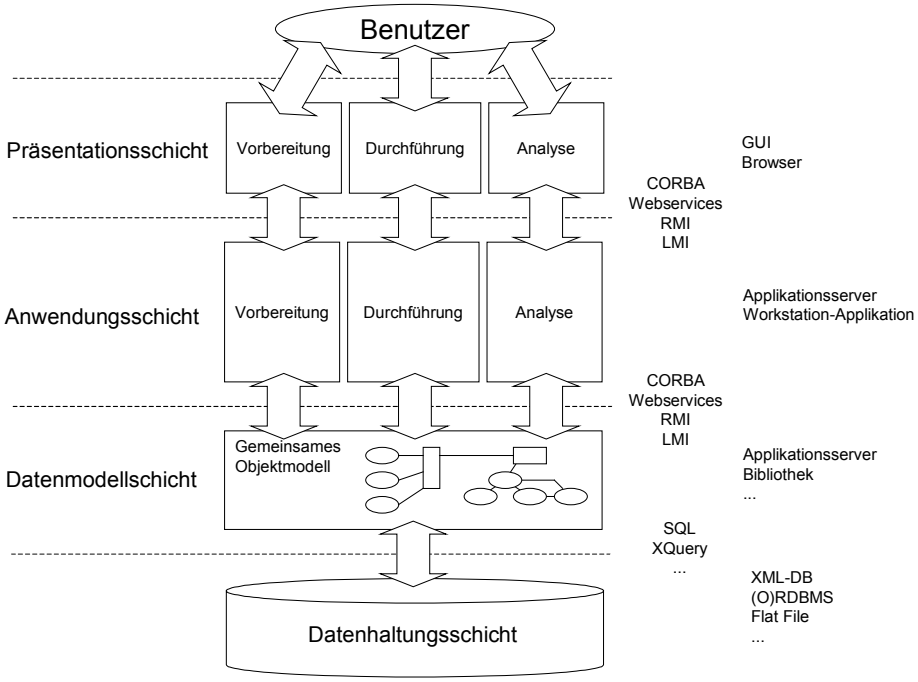
Als Alternative bietet sich die Einführung einer weiteren Zwischenschicht (Datenmodellschicht, *data model layer*) an, welche die Abbildung zwischen den Strukturen der Datenhaltung und denen der Anwendung vornimmt, und gleichzeitig die Integrität der Daten garantiert. Dazu ist ein einheitliches Objektmodell zu entwerfen, das von den Komponenten der oberen Schicht genutzt werden kann.

Durch das Vorliegen der Anwendungsdaten in ihrer natürlichen, nicht dekomponierten Form kann die Datenmodellschicht viele Funktionen zur Sicherung der Konsistenz der Daten effizient und einheitlich für das gesamte virtuelle Labor sicherstellen. Abbildung 12 zeigt den Architekturvorschlag für das virtuelle Labor.

Wird die Datenmodellschicht als eigener Tier ausgeführt, der von allen Clients genutzt wird, kann an dieser Stelle ein sehr effizientes Caching der Daten stattfinden, da keine Konsistenzprobleme durch verteilte Caches entstehen. So kann nicht nur die Rekonstruktion aus der Datenhaltungsschicht bei mehrfachen Zugriffen auf die gleichen Daten eingespart werden, vielmehr kann auch Wissen über die Nutzung der Anwendungsobjekte effizienzsteigernd verwendet werden.

Zudem kann die Datenmodellschicht eine direktere Kopplung der drei Phasen ermöglichen: Der Informationsaustausch muss nicht über die Datenhaltungsschicht in deren Informationsmodell erfolgen, sondern kann effizienter durch direkten Austausch der in der vorhergehenden Phase erstellten oder veränderten Objektstrukturen des gemeinsamen Objektmodells stattfinden.

Abbildung 12 Architektur des virtuellen Labors



4.1.2.5 Realisierungsmöglichkeiten einer Datenmodellschicht

Bei der Realisierung dieses Objektmodells müssen zwei mögliche Szenarien berücksichtigt werden:

Sind die Komponenten der Präsentations- und Anwendungslogik homogen, insbesondere mit Rücksicht auf die verwendeten Plattformen und Programmiersprachen, kann das Objektmodell in Form einer Bibliothek realisiert werden, die von den höheren Schichten zum Zugriff auf die Datenhaltung verwendet wird und die Daten in Form von Objektstrukturen des einheitlichen Objektmodells zurückliefert.

Da die Komponenten der oberen Schicht wegen der divergierenden Anforderungen jedoch nicht notwendigerweise homogen sind, also beispielsweise in verschiedenen Programmiersprachen entwickelt werden, sollte der Aspekt der Sprachunabhängigkeit bei der Realisierung des Objektmodells und der Kommunikation zwischen den Schichten berücksichtigt werden. Es bieten sich hierzu verschiedene Middleware-Technologien an, die eine sprachneutrale Kommunikation erlauben. Diese sollen im Folgenden kurz vorgestellt werden, wobei eine Beschränkung auf jene Aspekte erfolgen soll, die als Grundlage für die Realisierung des Objektmodells des virtuellen Labors erforderlich sind.

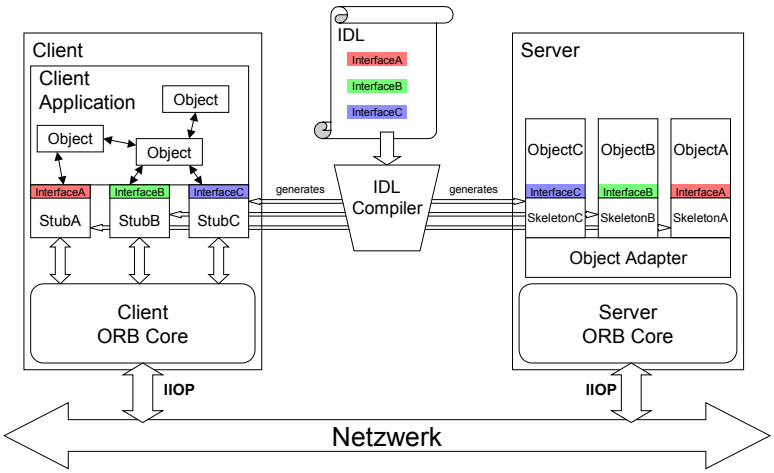
Common Object Request Broker Architecture (CORBA)

CORBA wurde von der Object Management Group als offene, herstellerunabhängige, objektorientierte Middleware-Plattform für die Entwicklung verteilter Anwendungen spezifiziert. Die aktuelle Version der CORBA Spezifikation [CORBA] mit über 1000 Seiten eignet sich als Nachschlagewerk, eine gute Übersicht zum Einstieg für Java-Entwickler gibt [CSC].

CORBA ermöglicht die transparente Kommunikation von Anwendungen in verschiedenen Programmiersprachen und auf verschiedenen Zielsystemen durch eine Zwischeninstanz, den *Object Request Broker* (ORB). Ein solcher ist auf jedem System installiert, das entfernte Objekte nutzen möchte oder selbst Dienste in Form von Serverobjekten anbietet. Die ORBs kommunizieren über das *Internet Inter Orb Protocol* (IIOP). Als Kommunikationsmechanismus dient eine erweiterte Form von *Remote Procedure Calls* (RPC). Ein Client richtet dazu einen aus seiner Sicht gewöhnlichen, lediglich um CORBA-spezifische Ausnahmebehandlungen erweiterten Methodenaufruf an ein lokal vorliegendes Stellvertreterobjekt (*Stub*). Dieses konvertiert die Beschreibung des Methodenaufrufs, bestehend aus Methodenname und den Parametern in ein sprachunabhängiges, binäres Austauschformat (*marshalling*). Mit Hilfe des lokalen ORB wird der Aufruf dann an das Zielobjekt weitergeleitet, das sich auf einem beliebigen System befinden, und in einer anderen Sprache implementiert sein kann. Serverobjekte werden über eine *Interoperable Object Reference* (IOR) identifiziert, die alle Information zum Senden einer Anforderung enthalten. Dazu gehören mindestens der Aufenthaltsort des Serverobjekts in Form einer IP-Adresse und die Identifikation des Prozesses auf dem Zielsystem. Eine IOR kann wahlweise fest in einer Anwendung kodiert sein oder vorzugsweise mit Hilfe eines Naming Service erfragt werden. Dieser löst logische Objektnamen in IORs auf. Die Anfrage wird vom ORB im Zielsystem entgegengenommen und an das serverseitige Gegenstück des Stubs, das sogenannte *Skeleton* weitergeleitet, das die Parameter extrahiert und in das Format der jeweiligen Plattform konvertiert (*unmarshalling*)

und anschließend mit diesen den Methodenaufruf auf dem eigentlichen Zielobjekt durchführt. Rückgabewerte des Aufrufs werden in analoger Weise in das Austauschformat konvertiert, und ebenfalls über den ORB an den Stub zurückgeleitet. Dieser setzt wiederum den Rückgabewert in das Format von Plattform und Sprache des Aufrufers um und liefert ihn an diesen zurück. Ein Grundprinzip von CORBA ist die strikte Trennung von Objekt- oder Komponentenschnittstelle und der Implementierung. Schnittstellen werden in der *Interface Definition Language* (IDL) beschrieben. Language Mappings spezifizieren eine Abbildung der IDL-Konstrukte auf die der unterstützten Zielsprachen. Mit Hilfe eines IDL-Compilers kann Code für Skeleton und Stub in der gewünschten Zielsprache generiert werden. Zusätzlich erzeugt dieser Vorgang auch Hilfsklassen für die Serialisierung von Methodenaufrufen und ihren Parametern. So kann ein Serverobjekt, das inC implementiert wurde und auf einem Unix-System residiert, von Clients genutzt werden, die selbst in C/C++, Java oder einer beliebigen anderen Sprache geschrieben sind und auf einer anderen Plattform laufen. Abbildung 13 zeigt den Ablauf der CORBA-Kommunikation und die Generierung der Stubs und Skeletons aus der IDL für die jeweilige Zielsprache.

Abbildung 13 CORBA Architektur



Das im Bild gezeigte Szenario zeigt einen CORBA-Client und einen CORBA-Server. Diese Trennung ist jedoch nicht im klassischen Sinne in Form einer festen Unterscheidung zwischen Client- und Server-Maschine zu verstehen. Vielmehr sind Client und Server Rollen. Jedes beteiligte System kann CORBA-Objekte für andere Rechner bereitstellen und als Server fungieren oder selbst die Dienste solcher Objekte in Anspruch nehmen, die über eine beliebige Menge von Rechnern verteilt sind. IORs und der Naming Service ermöglichen eine Abstraktion von den Details dieser Verteilung.

Obwohl CORBA als Grundlage für die Datenmodellschicht und die sprachunabhängige Kommunikation zwischen den Komponenten des virtuellen Labors geeignet erscheint, bestehen zahlreiche Defizite. So war die erste CORBA-Spezifikation nicht präzise genug und unvollständig, sodass viele CORBA-Produkte entstanden, die trotz Einhaltung dieser Spezifikation zueinander inkompatibel waren. So existierte anfangs kein einheitliches Kommunikationsformat, was in Form von IIOP erst verspätet nachgereicht wurde. Dennoch bleibt auch heute die Interoperabilität von CORBA-Produkten verschiedener Hersteller problematisch, insbesondere wenn man sich nicht auf die Grundfunktionalität beschränkt. So waren

die Nutzer von CORBA quasi gezwungen, sich für den ORB eines Herstellers zu entscheiden, der sein Produkt für alle benötigten Plattformen anbietet. Damit begibt man sich jedoch wieder in die Abhängigkeit von diesem Hersteller und seiner Produktpolitik. Angesichts dieser Schwierigkeiten konnte sich auch nicht der erhoffte Markt für fertige CORBA-Komponenten entwickeln. Zudem bestehen weitere technische Probleme. So ist selbst mehr als zehn Jahre nach der Entwicklung der ersten ORBs ihre Stabilität und Skalierbarkeit für kritische Anwendungen nicht ausreichend. Weiterhin erfordert IIOP einen großen Bereich freier Ports zur Kommunikation und stellt damit eine große Lücke im auf Firewalls basierenden Sicherheitskonzept von Unternehmen dar.

Dies hat dazu geführt, dass CORBA inzwischen als aussterbende Technologie betrachtet werden muss. Von den zahlreichen Produkten der Anfangszeit sind nur noch wenig am Markt erhältlich. Trotz des Komforts der transparenten plattformübergreifenden Kommunikation kommt CORBA damit als Basis einer neu zu entwickelnden Plattform nicht in Frage.

XML

XML (*eXtensible Markup Language*) [W3CXML] hat sich in den letzten Jahren als Metasprache für den Austausch strukturierter und semistrukturierter Daten insbesondere in heterogenen Rechnerumgebungen etabliert. Es ist textbasiert und damit prinzipiell menschenlesbar und strukturiert die Informationen mit Hilfe von geschachtelten Elementen (*Tags*), die zusätzlich über Attribute verfügen können. Ein Element kann Subelemente, nicht weiter durch Tags strukturierten Inhalt (content) oder beides (mixed content) enthalten. Die Dokumentstruktur ist grundsätzlich hierarchisch, Querverweise innerhalb und zwischen Dokumenten sind jedoch mit verschiedenen Verfahren möglich.

XML-Grammatiken

Als Metaformat bringt XML keine vordefinierte Tagmenge mit, wie es beispielsweise bei HTML der Fall ist, sondern bietet dem Entwickler die Möglichkeit, selbst die Grammatik für seine Dokumente durch Angabe von Bezeichnungen und Struktur von Elementen und ihren Attributen zu definieren. Dazu stehen zwei Möglichkeiten zur Auswahl. Eine Document Type Declaration (DTD) ist der ursprüngliche Ansatz zur Beschreibung von XML-Grammatiken. Sie beschreibt den Namen einzelner Tags oder Elemente und Art und Anzahl der in einem Element enthaltenen Subelemente sowie seine Attribute. Beispiel 1 zeigt eine DTD für ein einfaches Dokument mit Kontaktinformationen

Beispiel 1 Document Type Declaration

```
<!ELEMENT contactbook (contact+)>
<!ELEMENT contact (name, address*, communication)>
<!ATTLIST contact id CDATA #required>
<!ELEMENT name (first, middle*,last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT address (street,number,city,zip)>
<!ELEMENT street (#PCDATA)>
```

```
<!ELEMENT number (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT communication (phone+,fax*,email*)>
<!ELEMENT phone (#PCDATA)>
<!ATTLIST phone type (home|office|mobile) "office">
<!ELEMENT fax (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

Beim Parsen eines XML-Dokuments kann das Dokument auf seine Konformität mit einer so definierten Grammatik überprüft werden. Entspricht es der Grammatik, spricht man von einem gültigen (*valid*) Dokument, während ein Dokument ohne Grammatikbeschreibung, aber ansonsten korrektem Aufbau bezüglich der Schachtelung von Tags als wohlgeformt (*well-formed*) bezeichnet wird. Die Konformität eines Dokuments zu einer DTD erlaubt somit der verarbeitenden Anwendungen gewisse vereinfachende Annahmen zu machen. Da eine DTD jedoch keinerlei Aussagen über die Bedeutung von Tags und Attributen macht, bleibt die Interpretation der Informationen der Anwendung überlassen. Die Verwendung von XML-basierten Datenformaten ist somit alleine kein Garant für Anwendungsunabhängigkeit.

Ein großes Defizit von DTDs ist die mangelnde Unterstützung von Typen. Alle Attribute und der Inhalt von Tags sind grundsätzlich uninterpretierter Text. Dies bedeutet, dass eine Überprüfung durch den Parser nur die Struktur berücksichtigen kann. Weitere Prüfungen, zum Beispiel, ob ein Attribut wie erwartet eine Zahl ist, müssen in der Anwendung selbst erfolgen.

Eine alternative Beschreibung von XML-Grammatiken ermöglicht XML Schema. Es bietet gegenüber DTDs eine erweiterte Ausdrucksfähigkeit. So kann zu jeder DTD ein äquivalentes Schema angegeben werden, umgekehrt ist dies nicht immer möglich. Im Gegensatz zu DTDs ist ein XML Schema selbst ein wohlgeformtes und gültiges XML-Dokument. Zusätzlich zur vollen Ausdrucksmächtigkeit von DTDs bietet es unter anderem ein Typsystem mit Vererbung. Dadurch ist eine Typprüfung schon durch den Parser möglich, was eine Vereinfachung der Verarbeitung in der Anwendung bedeutet. Beispiel 2 zeigt ein XML Schema, das Dokumente beschreibt, deren Aufbau dem der von der zuvor gezeigten DTD definierten Dokumente entspricht. Jedoch wurde die Beschreibung durch Spezifikation von Typen präzisiert und so die Anzahl der gültigen Dokumente eingeschränkt: Der Wert für Hausnummern (number) wurde in diesem Beispiel auf positive ganze Zahlen beschränkt, Postleitzahlen müssen aus fünf Ziffern bestehen, Email-Adressen mit einem Buchstaben beginnen und ein „@“ enthalten.

Beispiel 2 XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="contactbook">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="contact" minOccurs="0" maxOccurs="unbounded">
```

```
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="name">
       <xsd:complexType>
        <xsd:element name="first" type="xsd:string"/>
        <xsd:element name="middle" type="xsd:string" minOccurs="0"
         maxOccurs="unbounded"/>
        <xsd:element name="last" type="xsd:string"/>
       </xsd:complexType>
      </xsd:element>
      <xsd:element name="address" type="addresstype" minOccurs="0"
       maxOccurs="unbounded"/>
      <xsd:element name="communication">
       <xsd:complexType>
        <xsd:sequence>
         <xsd:element name="phone" type="phonetype" minOccurs="1"
          maxOccurs="unbounded"/>
         <xsd:element name="fax" type="phonetype" minOccurs="0"
          maxOccurs="unbounded"/>
         <xsd:element name="email" type="emailtype" minOccurs="0"
          maxOccurs="unbounded"/>
        </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

<xsd:complexType name="addresstype">
 <xsd:sequence>
  <xsd:element name="street" type="xsd:string"/>
  <xsd:element name="number" type="xsd:positiveInteger"/>
  <xsd:element name="city" type="xsd:string"/>
  <xsd:element name="zip">
   <xsd:simpleType>
    <xsd:restriction base="xsd:string">
     <xsd:pattern value="[0-9]{5}"/>
    </xsd:restriction>
   </xsd:simpleType>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>
```

```
<xsd:simpleType name="phonetype">
 <xsd:restriction base="xsd:string">
  <xsd:pattern value="[0-9]{7,15}"/>
 </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="emailtype">
 <xsd:restriction base="xsd:string">
  <xsd:pattern value="\D.*@\D.*"/>
 </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Nutzung von XML am Beispiel Java

Um auf XML-Dokumente zugreifen zu können, hat der Anwendungsentwickler mehrere Möglichkeiten. Die Eigenentwicklung eines XML-Parsers ist eine eher hypothetische Möglichkeit, da zahlreiche leistungsfähige APIs für den Umgang mit XML zur Verfügung stehen, die das Parsen und Validieren eines Dokumentes gegen eine XML-Grammatik übernehmen und einen Zugriff auf die Elemente mit mehr oder weniger stark abstrahierenden Verfahren erlauben. Besonders für Java als Sprache der Wahl für viele Anwendungen im professionellen Umfeld stehen inzwischen eine Vielzahl verschiedener APIs zur Wahl, von denen zwei kurz vorgestellt und bewertet werden sollen.

Java API for XML Processing (JAXP)

Die erste XML-Standardschnittstelle für Java war das *Java API for XML Processing* [JAXP]. Es beschreibt die Schnittstellen für Parser basierend auf dem *Document Object Model* (DOM) oder dem *Simple API for XML* (SAX), sowie für die Transformation von XML-Dokumenten mit *Extensible Stylesheet Language Transformations* (XSLT). Die konkrete Implementierung der Schnittstellen kann aus verschiedenen, häufig freien Produkten gewählt werden. Die Unterstützung einzelner XML-Features wie von XML Schema zur Beschreibung von Grammatiken und zur Validierung von Dokumenten oder von XML Namespaces variiert je nach Implementierung. Eine weitverbreitete Implementierung ist der Xerces Parser [XERCES] des *Apache XML Projects*, der unter Open Source Lizenz frei verfügbar ist. Er ersetzt oft den in seiner Funktionalität eingeschränkten Crimson Parser, der in neueren Versionen des Java Development Kit mitgeliefert wird.

Die beiden JAXP-Parser unterscheiden sich deutlich in ihrem Verarbeitungsmodell. Ein DOM-Parser generiert einen Baum von Objekten, der das gesamte XML-Dokument repräsentiert. Die Anwendung kann innerhalb dieses Baums navigieren, und Elemente und Attribute lesen und modifizieren. Ein DOM-Baum kann wieder in ein XML-Dokument umgewandelt und daher auch zur Generierung beziehungsweise zur Änderung von Dokumenten genutzt werden. Vorteil von DOM ist vor allem die Möglichkeit, jederzeit bei der Auswertung eines Dokuments auf jedes seiner Elemente zurückgreifen zu können. Dies ist jedoch auch zugleich der entscheidende Nachteil: Da der gesamte Dokumentbaum im Speicher gehalten wird, eignet sich DOM nur für kleinere Dokumente. Aus einem Dokument mit eini-

gen hundert Kilobyte werden im Speicher schnell einige Megabyte. Zudem wird das sprachunabhängig konzipierte DOM-API oft als zu wenig objektorientiert und daher mit Java schlecht handhabbar kritisiert. Projekte wie JDOM [JDOM] und DOM4j [DOM4J] haben zum Ziel, ein auf Java optimiertes und somit natürlicher einsetzbares XML-API zu entwickeln.

SAX-Parser arbeiten ereignisbasiert. Das Dokument wird sequentiell gelesen und für jedes auftretende Tag ein Ereignis erzeugt, das über Callback-Methoden an eine vom Entwickler der Anwendung erstellte Handler-Klasse übergeben wird und neben dem voll qualifizierten Namen des Elements auch seine Attribute und deren Werte enthält. Weitere Callback-Methoden signalisieren Dokumentanfang und -ende oder das Auftreten von Inhalt (content). Die Handler-Klasse verarbeitet diese Daten in einer anwendungsspezifischen Weise, SAX macht hierzu keine weiteren Vorgaben. Ein gängiges Verfahren ist ein Stack-basierter Ansatz. Für jedes Start-Tag wird ein dieses Tag, seinen Inhalt und seine Attribute symbolisierendes Objekt auf dem Stack abgelegt, der so immer die hierarchische Schachtelung der Tags an der aktuellen Position des Parsers im Dokument widerspiegelt. Wird ein Element geschlossen, wird der Stack wieder bis zu dem Objekt abgebaut, welches das Startobjekt symbolisiert. Vorteil dieser Lösung und prinzipiell des SAX-Konzepts ist der deutlich geringer Speicherbedarf, da der Entwickler selbst kontrolliert, welche Elemente er wie speichert. Wegen der linearen Verarbeitung ist die Entwicklung eines SAX-Parsers aufgrund der nötigen Verwaltung schon geparster Daten häufig komplexer als die eines funktional vergleichbaren DOM-Parsers. Bei vielen Implementierungen von JAXP nutzt der DOM-Parser selbst den SAX-Parser zum Aufbau des Dokumentenmodells.

Sowohl die Verwendung von DOM als auch von SAX bedeuten einen deutlichen Bruch zwischen XML-Dokument und der Repräsentation der Daten in der Anwendung. Der Entwickler selbst ist für die Implementierung der Abbildung zuständig, was sehr zeitaufwendig und fehlerträchtig ist.

Java Architecture for XML Binding (JAXB)

Die *Java* Architecture *for XML Binding* [JAXB] verfolgt einen direkteren Ansatz für die Abbildung zwischen XML und Java-Objekten. Ausgehend von einem XML Schema generiert ein Binding Compiler (xjc) Java-Klassen für jedes der Elemente des Schemas, die sogenannten *schema derived classes*. Zum Zugriff auf JAXB in der Applikation dient die Klasse `JAXBContext`, die das Erzeugen von `Marshaller`- und `Unmarshaller`-Objekten erlaubt. Ein Unmarshaller nimmt ein XML-Dokument aus einer Menge von möglichen Quellen entgegen, unter anderem Dateien, Stringbuffer, Inputstreams und DOM-Knoten. Aus dem Dokument wird dann die seinem Inhalt entsprechende Objekt-Struktur aus den zuvor generierten Klassen erzeugt. Auf Attribute eines Elements kann mit getter-Methoden direkt zugegriffen werden, Subelemente sind in Form eines `List`-Objekts zugänglich. Durch Setzen der Attribute mit setter-Methoden und Hinzufügen, Löschen oder Ändern von Elementen der Subelement-Listen kann die Objektstruktur manipuliert werden. Ein Marshaller-Objekt kann dann aus der Objektstruktur ein neues XML-Dokument generieren.

Die Abbildung von XML Schema-Elementen auf Java-Klassen erfolgt nach Standardregeln (*default binding rules*). Sollte dieses Regelwerk nicht zur Zufriedenheit des Entwicklers ar-

beiten, kann die Abbildung mit maßgeschneiderten Regeln, den *custom binding rules* angepasst werden. Dies kann insbesondere bei Namenskonflikten erforderlich werden, aber auch zur Änderung der automatisch aus dem Namespace des Schemas generierten Package-Namen. Die Abbildung von XML Schema-Typen auf Java-Typen kann mit Wissen über die Daten angepasst werden. Ebenso lässt sich die für die Darstellung von Listen von Elementen verwendete Klasse auswählen, die lediglich das Interface `java.util.List` implementieren muss. Die Angabe der Regeln kann auf zwei Arten erfolgen: „Inline“ im Schema in Form von annotation-Elementen des XML Schema-Namespace, der Wirkungsbereich ergibt sich aus der Position der Regeldefinition im Schema. Alternativ kann man custom binding rules über eine separate Binding-Datei spezifizieren, der Wirkungsbereich wird dazu mit einem XPath-Ausdruck angegeben. Die Binding-Datei selbst wird dem Binding-Compiler als zusätzlicher Parameter übergeben.

Bewertung

XML eignet sich als Metaformat, um komplexe Objekte, wie sie im virtuellen Labor vorkommen, plattform- und sprachunabhängig zu beschreiben. Als Nachteil gegenüber einem binären, möglicherweise sprachspezifischen Datenformat darf jedoch der Overhead der Textdarstellung und der Aufwand für das Parsen nicht vernachlässigt werden. Die Anwendung muss XML-Dokumente selbst in ihr internes Format konvertieren, um damit arbeiten zu können, sodass das eigentliche Ziel, mit der Datenmodellschicht direkt nutzbare Objektstrukturen zu erzeugen, so nicht erreicht werden kann. Binding-APIs wie JAXB lösen dieses Problem nicht vollständig. Zwar erzeugen sie im Gegensatz zu den generischen Objekten eines DOM-Baumes eine schemaspezifische Objektstruktur, die daher platzsparender ist und effizienter verarbeitet werden kann. Die generierten Klassen eignen sich jedoch nur bedingt zur direkten Weiterverwendung in der Anwendung. Zwar sind sie im Quellcode zugänglich, ein Einfügen von anwendungsspezifischen Methoden zur Manipulation, die über rudimentäre Setter-Methoden hinausgeht, ist jedoch aufwendig und wird nicht empfohlen.

Als Mittel für Export und Langzeitarchivierung von Experimentdaten ist XML Mittel der Wahl. Die Verwendung von XML in der Datenhaltungsschicht wird in Kapitel 4.2 diskutiert.

Web Services

Nahezu alle existierenden Komponententechnologien weisen Defizite auf, die für ihre universelle Anwendbarkeit und somit für eine weitere Verbreitung hinderlich waren. Unvollständige Spezifikationen provozieren Kompatibilitätsprobleme, die Beschränkung auf einzelne Rechner- und Betriebssystemplattformen oder Programmiersprachen schränkt die möglichen Anwendungsgebiete ein. Das Web Service-Konzept ist nun angetreten, die Vorteile seiner Vorgänger zu kombinieren und zugleich ihre Fehler zu vermeiden. Von zahlreichen Firmen und Institutionen als das Allheilmittel angepriesen, sollen so alle Probleme bisheriger Komponententechnologien zur Entwicklung verteilter Anwendungen gelöst werden.

[CSWSP] beschreibt Web Services als „modulare, selbstbeschreibende, abgeschlossene Applikationen, auf die über das Internet zugegriffen werden kann”. Gleichsam hat fast jedes Unternehmen, das sich Web Services auf die Fahnen schreibt, eine eigene Definition parat.

Aufgrund dieser Unklarheiten soll hier die Definition des World Wide Web Consortium (W3C) verwendet werden, das als herstellerunabhängiges Gremium die Standardisierung von Web Services und anderen Internet-Technologien vorantreibt:

„A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by internet protocols." [W3CWSA]

Die Anleihen an bestehenden Technologien sind unverkennbar. So lassen sich leicht Parallelen zu CORBA entdecken: die Serverobjekte von CORBA bieten Dienste in Form von Methoden an, werden über IORs weltweit eindeutig identifiziert und ihre Schnittstellen sind in der IDL beschrieben. Die Kommunikation erfolgt auch hier über das Internet mit IIOP. Webservices nutzen lediglich andere Technologien und Formate für diese Konzepte: URIs dienen der Identifikation, die Beschreibung der Schnittstellen erfolgt mit der *Web Service Description Language* (WSDL), die Kommunikation erfolgt (meistens) mit dem XML-basierten Simple Object Access Protocol (SOAP) über HTTP. Die Rolle des CORBA-Nameservice übernimmt eine UDDI-Registry.

Die wichtigsten Technologien im Zusammenhang mit Webservices sollen im Folgenden kurz erläutert werden.

WSDL

Die Web Service Description Language ermöglicht als Metasprache die Beschreibung alle nötigen Informationen für die Kommunikation zwischen dem Anbieter eines Web Service (Provider) und dessen Nutzer (Requester), also seiner Schnittstelle.

Jeder Web Service bietet eine Menge von Operationen an, die als *port type* bezeichnet wird. Eine Operation wird durch ihre Ein- und Ausgabenachrichten beschrieben. Diese enthalten den Namen der Operation, und sogenannte *parts*, welche die Rolle von Parametern und Rückgabewerten spielen.

Ein *Binding* definiert ein konkretes Protokoll und Datenformate für einen port type. Bisher stehen hier SOAP 1.1, HTTP get/post und MIME zur Auswahl. Zusätzlich wird entweder ein RPC-artiger oder ein dokumentenbasierter Nachrichtenstil (*Style*) definiert. Ein *Port* dient schließlich als Kommunikationsendpunkt (Endpoint) für eine Adresse, der einem Binding zugeordnet ist. Mehrere Ports bilden einen Web Service.

Die Bekanntgabe dieser Informationen durch den Anbieter eines Web Service (Provider) ermöglicht nun jedem potentiellen Interessenten (Requester) die Nutzung des Dienstes. WSDL-Dateien sind sehr komplex und werden im Allgemeinen aus anderen Formen von Schnittstellenbeschreibungen generiert.

SOAP

SOAP (Simple Object Access Protocol) ist ein XML-basiertes Protokoll, das einen Mechanismus zur Übergabe von Methodeaufrufen oder Befehlen und ihren Parametern definiert.

Während der Erstellung dieser Arbeit stand SOAP Version 1.2 kurz vor der Verabschiedung durch das W3C [W3CSoap]. Wie das gesamte Web Service-Konzept ist auch SOAP unabhängig von der verwendeten Plattform, dem Objektmodell und der Programmiersprache. Die wesentlichen Vorteile von SOAP im Vergleich zum Java Remote Method Protocol (JRMP) der Remote Method Invocation (RMI) oder CORBAs IIOP beruhen auf der Verwendung von XML: Als textbasiertes Format ist es leichter zu debuggen und in Verbindung mit HTTP unproblematisch im Umgang mit Firewalls. Neben der konventionellen synchronen Kommunikation in Form eines RPC ermöglicht SOAP auch dokumentenorientierte Nachrichten sowie die asynchrone Nutzung von Diensten.

UDDI

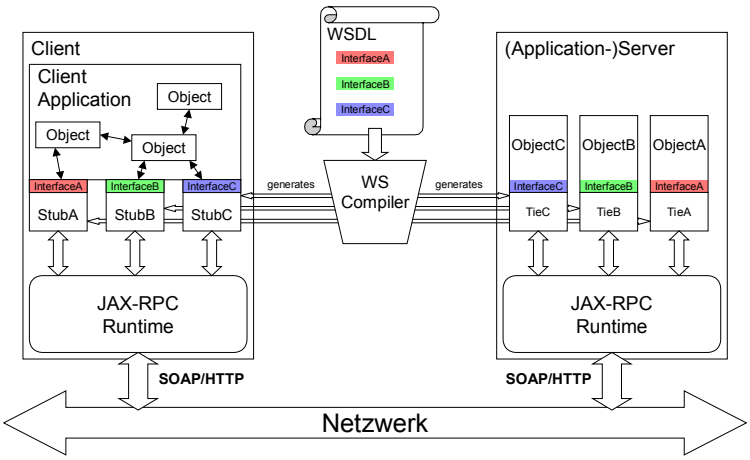
UDDI steht für *Universal Description Discovery and Integration of Web Services*. Eine UDDI-Registry speichert Informationen über Unternehmen und die von ihnen angebotenen Dienste in einem einheitlichen XML-Format. Die Registry spielt neben Service Requester und Service Provider die dritte wichtige Rolle im Web Service-Konzept.

Die Interaktion mit dem oft als Meta-Web Service bezeichneten UDDI erfolgt ebenfalls über SOAP. In Analogie zu Telefonbüchern trennt man eine UDDI-Registry in drei Bereiche: Die *white pages* enthalten Kontaktinformationen der Unternehmen wie ein konventionelles Telefonbuch, in den *yellow pages* sind die Unternehmen hierarchisch nach Sparten gegliedert, die *green pages* bieten technische Informationen zu den angebotenen Diensten.

Nutzung von Web Services am Beispiel Java

Der Zugriff auf Web Services variiert je nach gewählter Sprache erheblich. Als gängige Basistechnologie zur Implementierung eines Web Service werden J2EE-Applikationsserver eingesetzt, welche die erforderliche Infrastruktur mitbringen: Der Webcontainer ermöglicht HTTP-Zugriff, Datasources und JDBC erlauben den Zugriff auf Datenbanken, das Java Transaction API (JTA) ermöglicht die programmatische Durchführung von Transaktionen. Inzwischen werden von quasi jedem Hersteller von Applikationsservern Tools bereitgestellt, welche die Realisierung von Web Services vereinfachen sollen. IBMs *Websphere Application Developer* (WSAD) und das Sun ONE Studio sind nur zwei Beispiele. Sind diese Produkte nicht bereits vorhanden, bietet Sun dem an Web Services interessierten Entwickler das *Java Web Service Developer Pack* (JWSDP), zu finden bei [JWS], als kostenlosen Einstieg an. Es bringt neben den aktuellen Versionen aller Java XML-APIs einfache (Kommandozeilen-) Tools zur Generierung von Web Services und als Web Container *Tomcat* [AJPT] mit. Wegen der begrenzten Leistungsfähigkeit der Tools ist bei der Entwicklung einige Handarbeit erforderlich, für den produktiven Einsatz erlauben Produkte mit weiterreichender Unterstützung eine deutlich einfachere und schnellere Entwicklung.

Abbildung 14 Web Services mit dem JAX-RPC



Zentrale Element für die Nutzung von Web Services mit Java ist das *Java API for XML-based RPC* (JAX-RPC) [JAXRPC]. Abbildung 14 zeigt die an der Kommunikation beteiligten Komponenten. Um die konzeptionelle Ähnlichkeit mit CORBA zu betonen, wurde die Graphik analog zu Abbildung 13 angelegt. Mit Hilfe der in Form einer WSDL-Datei vorliegenden Beschreibung von Schnittstellen und Formaten des Web Service generiert ein Web Service Compiler die server- und clientseitigen Klassen, die auch als *Artefakte* (*artifacts*) bezeichnet werden. Wie bei CORBA heißen die Stellvertreterobjekte im Client *Stub*, während das Äquivalent zum CORBA-Skeleton hier als *Tie* bezeichnet wird. Auch hier werden für beide Seiten Hilfsklassen zur Serialisierung generiert. Um die WSDL-Dateien nicht selbst erzeugen zu müssen, können sie aus einem Java-Interface erstellt werden.

Die generierten Klassen werden nun in den Client integriert, dem zusätzlich die Klassen der JAX-RPC-Laufzeitumgebung in Form einiger Packages verfügbar gemacht werden müssen. Realisiert man den Web Service wie im Bild gezeigt auch mit Java, werden die serverseitigen Klassen zu einer Web-Applikation zusammengestellt und in einem Web-Container beziehungsweise in einem vollständigen J2EE-Applikationsserver zum Einsatz gebracht, ein Vorgang, der als *Deployment* bezeichnet wird. Die vom Client an die Stubs gerichteten Anfragen werden mit den Hilfsklassen in eine SOAP-Nachricht verpackt und per HTTP an den Server, also den Web Service Endpoint, weitergeleitet. Dieser entnimmt den Methodennamen und deserialisiert die Parameter, führt damit den Methodenaufruf durch, serialisiert den Rückgabewert und leitet ihn ebenfalls als SOAP-Nachricht zurück. Das Laufzeitsystem auf Clientseite extrahiert den Rückgabewert und leitet ihn über den Stub an den Aufrufer der Methode zurück. Dieser letzte Teil kann bei einem asynchronen RPC entfallen.

Die Verwendung von generierten Stubs und Ties hat im Wesentlichen den Zweck, die Durchführung eines RPC wie einen lokalen Methodenaufruf aussehen zu lassen und so die Nutzung zu vereinfachen. Neben dieser statischen Realisierung von Web Services, bei der schon zur Compilezeit die Kommunikationspartner und die verwendeten Schnittstellen feststehen, können mit JAX-RPC auch dynamische Aufrufe zur Laufzeit generiert werden. Dazu gibt es zwei verschiedene Möglichkeiten, die sich im Grad der Flexibilität unterscheiden. Ein sogenannter dynamischer Proxy kann mit Informationen über den zu nutzenden Web Service (Endpoint und Port) generiert werden. Dabei wird zur Laufzeit eine Implementierung des

Service basierend auf seiner WSDL-Beschreibung generiert, auf der die Methoden aufgerufen werden können. Die Methodenaufrufe erfolgen nach wie vor hartkodiert, lediglich das Objekt, auf dem sie aufgerufen werden, wird dynamisch erzeugt. Man spart so nur die Erzeugung der Artefakte bei der Entwicklung, gewinnt aber keine zusätzliche Flexibilität und verschlechtert möglicherweise das Verhalten zur Laufzeit. Zudem macht nun die Verwendung von anderen als den Standarddatentypen als Parameter Probleme, da keine Vorab-Generierung von Serialisierungsklassen möglich ist. Um dennoch spezielle Anwendungsobjekte transportieren zu können, kann alternativ ein selbst erstellter (De-)Serializer registriert werden. Dadurch wird jedoch der letzte Vorteil der Verwendung des dynamischen Proxy ins Gegenteil verkehrt.

Die zweite Möglichkeit für dynamische Aufrufe bietet im Vergleich zum dynamischen Proxy größere Flexibilität: Das *Dynamic Invocation Interface* (DII) ermöglicht Aufrufe von Methoden, deren Signatur erst zur Laufzeit bekannt wird, zum Beispiel durch Zugriff auf eine UDDI-Registry über das *Java API for XML Registries* [JAXR]. Es erfordert keine Generierung von Klassen, weder bei der Entwicklung noch zur Laufzeit. Nachteil ist die höhere Komplexität des Codes für den Methodenaufruf. Auch hier müssen zudem für komplexe Typen eigene Serialisierer und Deserialisierer erstellt werden. Dies schränkt den möglichen Grad an Dynamik ein.

Die Semantik eines Web Service-Aufrufs ist zudem grundsätzlich call-by-value, die als Rückgabewerte erhaltenen Objekte können zwar über Methoden zur Manipulation verfügen, diese Änderungen wirken sich jedoch ausschließlich lokal aus, das Objekt fungiert also nicht als Stub.

Bewertung

Sind Web Services wirklich mehr als ein „RPC mit XML“, wie sie von Kritikern gerne bezeichnet werden? Die Vision der unkomplizierten und formlosen Kooperation zwischen Unternehmen, die dazu vorher keine geschäftlichen Beziehungen unterhalten müssen, mit UDDI als Mittler und Marktplattform, hat sich bisher nicht verwirklicht. So bleiben die technischen Vorteile, die Web Services dem Entwickler einer verteilten Anwendung bieten:

Neben dem Vorteil der Sprach- und Plattformunabhängigkeit, welche die Verwendung von XML als Grundlage für alle Formate und Beschreibungen mit sich bringt, erlaubt insbesondere die Verwendung des HTT-Protokolls zur Kommunikation eine große Vereinfachung bei der Anwendung in komplexen Netzwerk-Infrastrukuren, da im Vergleich zu CORBA ein einzelner Port genügt, der nur selten durch Sicherheitsmaßnahmen wie Firewalls blockiert wird. Als Textformat sind SOAP-Nachrichten und WSDL-Dateien menschenlesbar, was das Debugging von komplexen Anwendungen erleichtert. Allerdings bringt XML einen erheblichen Overhead durch die im Vergleich zu einem effizienten Binärformat leicht um eine Größenordnung gewachsenen Nachrichten mit sich. Entscheidend ist letztlich der Vorteil, mit Web Services eine Technologie zur Verfügung zu haben, die im Gegensatz zum aufgegebenen CORBA von vielen Unternehmen mit großem Aufwand weiterentwickelt wird und zum Zentrum der eigenen Produktphilosophie erklärt wurde. Dadurch steht eine große Auswahl von Produkten zur Verfügung, die zudem aufrgund der Arbeit der Standardisierungsorganisationen schon jetzt deutlich besser zusammenarbeiten, als das bei CORBA je der Fall war.

In der Praxis bleiben noch einige Probleme, wie sie bereits bei der Schilderung der Nutzung von Web Services mit Java beschrieben wurden. So gibt es erhebliche Einschränkungen bei der Verwendung von anderen als den Standarddatentypen als Parameter, was insbesondere beim Einsatz der beiden dynamischen Verfahren Probleme bereitet. Die in [ThGö2002] vorgestellte Lösung zur transparenten Serialisierung beliebiger Java-Objekte ohne Erzeugung spezieller Serialisierer leidet bisher unter Leistungsproblemen, löst aber diesen entscheidenden Kritikpunkt und bietet zudem optional call-by-reference-Semantik.

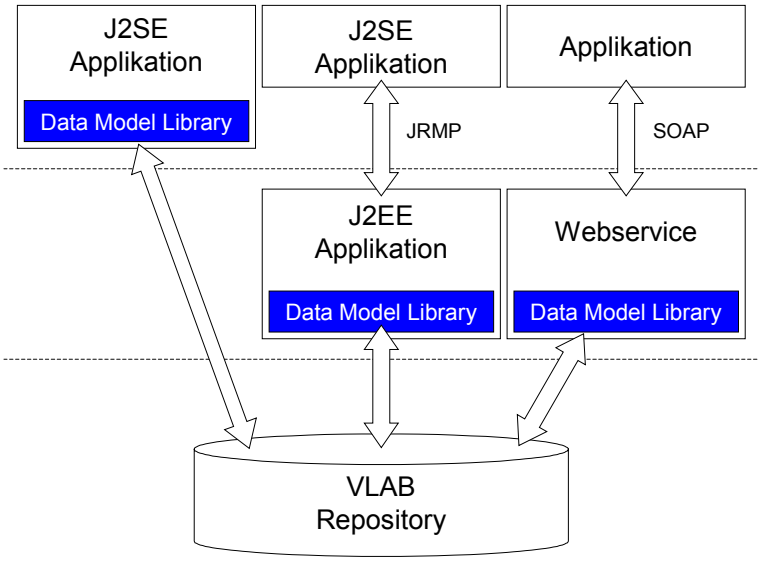
Vorteil einer einheitlichen Implementierungssprache

Die zuvor beschriebenen Technologien eignen sich prinzipiell alle für die Nutzung zur Definition des gemeinsamen Objektmodells. Der Preis für die Sprachunabhängigkeit ist jedoch ein nicht unerheblicher Effizienzverlust durch die immer erfolgende Umwandlung der Daten in ein Transportformat und den generell höheren Aufwand für die Kommunikation über Prozess- und Systemgrenzen hinweg. Bei interaktiven Anwendungen, die ständig auf die Datenhaltungsschicht zugreifen, könnte dies die Reaktionszeiten für die einzelnen Nutzer sowie den Gesamtdurchsatz des Systems verschlechtern. Allerdings erfolgt die Kommunikation zwischen der bezüglich der Reaktionszeit besonders kritischen Komponente für die Experimentvorbereitung und der Datenhaltungsschicht wegen des in Kapitel 3 beschrieben Check-In/Check-Out- Bearbeitungsparadigmas im Allgemeinen in Form von einzelnen Anfragen bei Beginn und Abschluss der Bearbeitung, bei der einmalig größere Datenmengen bewegt werden. Bei der in den Anforderungen D4 beschriebenen direkten Beobachtung der Experimentdurchführung mit Hilfe der Visualisierungs- und Auswertungskomponente kann je nach anfallender Datenmenge und Häufigkeit der Änderungen eine erheblicher Kommunikationsaufwand entstehen. Man sollte daher den Vorteil einer homogenen Realisierung des virtuellen Labors durch Nutzung einer einheitlichen Sprache für die Entwicklung nicht vernachlässigen. So könnte die für den systemübergreifenden Zugriff auf die Datenhaltung erforderliche Kommunikation effizienter mit einem sprachabhängigen Verfahren wie der Java Remote Method Invocation (RMI) erfolgen. Eine einheitliche Implementierungssprache kann jedoch nie für alle Komponenten des virtuellen Labors die Optimallösung darstellen. Hinzu kommt der unterschiedliche Hintergrund der Entwicklungsteams für die einzelnen Komponenten, die jeweils in einer ihnen vertrauten Sprache arbeiten wollen.

Realisierungsvorschlag

Als Ausweg bietet sich die Implementierung der Datenmodellschicht in Form einer universellen Bibliothek an, die dann sowohl direkt mit der gewählten Implementierungssprache als auch für die Realisierung einer sprachunabhängigen Zugriffsmöglichkeit zum Beispiel über Web Services verwendet werden kann. Als Sprache der Wahl bietet sich hier Java an. Bei geeigneter Auslegung der Bibliothek kann sie mit geringen Anpassungen sowohl direkt in einer Applikation genutzt werden, als auch als Grundlage für die Realisierung eines Web Service in einem J2EE-Applikationsserver oder Web Container zum Einsatz kommen.

Abbildung 15 Flexible Einsatzmöglichkeiten bei Realisierung der Datenmodellschicht als Java Bibliothek



4.2 Persistente Datenhaltung

Im vorigen Kapitel wurde eine zentrale Datenhaltungskomponente (Repository) als Mittel für kooperatives Arbeiten und den Datenaustausch zwischen Einzelkomponenten vorgestellt.

Ihre Eigenschaften bestimmen in entscheidendem Maße sowohl die spätere Leistungsfähigkeit des virtuellen Labors als auch die Entwicklung der Komponenten der höheren Schichten. Wegen der großen konzeptionellen Unterschiede zwischen den verschiedenen Technologien und Systemen, die zur Auswahl stehen, ist ein späterer Wechsel äußerst aufwendig. Dementsprechend sollte die Entscheidungen über die letztlich einzusetzende Technologie zur Realisierung des Repository nach sorgfältiger Abwägung erfolgen.

4.2.1 Anforderungen an die Datenhaltungsschicht

Als einer der zentralen Komponenten des virtuellen Labors, muss das Repository hohe Anforderungen an die Leistungsfähigkeit, Skalierbarkeit, Robustheit und Flexibilität erfüllen. Diese Anforderungen sollen zunächst kurz vorgestellt und bei der anschließenden Beschreibung der möglichen Alternativen berücksichtigt werden.

4.2.1.1 Leistungsfähigkeit

Bei der Durchführung von virtuellen Experimenten werden große Datenmengen in kurzer Zeit benötigt. Ein Experimentaufbau kann aus Hunderten von Experimentbausteinen bestehen, die über umfangreiche, sich gegenseitig referenzierende Datensätze beschrieben sind. Damit ein Check-In- oder Check-Out-Vorgang in akzeptabler Zeit stattfinden kann, muss die

Datenhaltungskomponente den effizienten Umgang mit diesen großen Datenmengen beherrschen.

Bei der Durchführung der Experimente entstehen je nach Komplexität der Simulationsmodelle und der Leistungsfähigkeit der Simulationsrechner sehr große Datenströme. Die Datenhaltungskomponente darf hierbei nicht zum limitierender Faktor werden, sondern muss diese Daten mit der Geschwindigkeit ihres Entstehens persistent speichern. Hier erweist sich besonders die stetig größer werdende Lücke zwischen der Leistungsfähigkeit der Rechner selbst und der üblicherweise als Sekundärspeicher eingesetzten Festplatten als problematisch. Im Gegensatz zu den meisten Nutzungsszenarien von Sekundärspeichern lässt sich in diesem Fall eine zu geringe Transferrate der Medien nicht durch Caching-Mechanismen verstecken.

Erschwerend kommt hinzu, dass bei der Verarbeitung der Daten in der Analysephase auf die ursprünglich als kontinuierlicher Strom angefallenen Daten nun wahlfrei zugegriffen werden muss. Es genügt also nicht, nur die Daten mit möglichst hoher Geschwindigkeit zu sichern, sie müssen auch in einer Form persistent gespeichert werden, die später einen schnellen direkten Zugriff auf einzelne Datensätze erlaubt.

4.2.1.2 Skalierbarkeit

Wegen ihrer zentralen Rolle für kooperatives Arbeiten und die Kopplung der Experimentphasen wird die Datenhaltungskomponente von allen Nutzern eines virtuellen Labors gemeinsam genutzt. Während Präsentations- und Anwendungslogik möglicherweise benutzernah auf dem Clientrechner angesiedelt sind und somit über private Ressourcen für den jeweiligen Benutzer verfügen, müssen die Ressourcen der Datenhaltungsschicht von allen Benutzern geteilt werden. Durch eine ausreichend dimensionierte Hardware kann garantiert werden, dass diese Komponente für die gegebene Anzahl an Nutzern ausreichende Leistungsreserven bietet. Werden jedoch neue Benutzer hinzugefügt, ist ein Ausbau dieser Reserven wesentlich schwieriger, als dies auf Clientseite durch Ergänzung neuer Workstations möglich ist. Die Datenhaltungsschicht sollte daher gut an steigende Leistungsanforderungen anpassbar sein. Dies kann durch Einsatz von leistungsfähigerer Hardware geschehen. Idealerweise ermöglicht die Datenhaltungsschicht eine Verteilung auf mehrere Rechner, wodurch eine Skalierung durch einfache Hinzunahme weiterer Systeme möglicherweise sogar im laufenden Betrieb erfolgen kann.

4.2.1.3 Ausfallssicherheit

Eine zentrale Datenhaltungskomponente erfordert nicht nur leistungsfähige und skalierbare Lösungen, sie bildet auch einen „Single Point of Failure“. Steht die Datenhaltungskomponenten zeitweilig nicht zur Verfügung, bedeutet dies einen fast vollständigen Ausfall des virtuellen Labors. Lediglich mit den lokal auf der Workstation vorhandenen Daten kann unter Umständen weitergearbeitet werden.

4.2.1.4 Robustheit

Der zuverlässigen und dauerhaften Speicherung aller Daten des virtuellen Labors kommt eine entscheidende Bedeutung zu. Neben Aspekten der Langzeitarchivierung von Experimenten und Experimentdaten für spätere Wiederverwendung und Analysen sind dabei insbesondere die Anforderungen, die durch die nebenläufige Nutzung durch zahlreiche Benutzer entstehen zu beachten. So muss der gleichzeitige Zugriff auf Daten durch mehrere Benutzer berücksichtigt und die so potentiell auftretenden Inkonsistenzen vermieden werden. Wurden große Änderungen an Objekten in einer Workstation durchgeführt, so müssen diese beim Check-In-Vorgang vollständig eingebracht werden. Eine nur teilweise Aktualisierung zum Beispiel durch einen zwischenzeitlich erfolgenden Systemfehler würde ein inkonsistentes und damit möglicherweise unbrauchbares Objekt hinterlassen.

4.2.1.5 Flexibilität und Änderbarkeit

Da ein virtuelles Labor an neue und wachsende Anforderungen seiner Benutzer angepasst werden muss, ist bei seiner Konzeptionierung auf leichte Änderbarkeit aller Komponenten zu achten. Für die Datenhaltungsschicht bedeutet dies, dass Erweiterungen des Datenmodells mit möglichst minimalen Änderungen an bereits bestehenden Elementen möglich sein sollen.

4.2.1.6 Abbildungsmächtigkeit

Während sich die bisher beschriebenen Anforderungen an die Datenhaltungskomponente objektiv beurteilen und häufig sogar quantifizieren lassen, ist eine objektive Bewertung der Abbildungsmächtigkeit vergleichsweise schwierig.

Unter dem Begriff der Abbildungsmächtigkeit soll die allgemeine Eignung der jeweiligen Technologie verstanden werden, mit ihrem zugrundeliegenden Meta-Informationsmodell die für die Anwendung benötigten Daten effizient zu speichern. Je besser eine Technologie hier abschneidet, umso geringer ist der Aufwand für Konversion und Transformation der Daten bei Speicherung und Retrieval. Ist eine Technologie zu unflexibel, können die Kosten für diese Abbildung einen Grad erreichen, der mögliche andere Vorteile im Bezug auf die sonstigen Bewertungskriterien überwiegt. Ein Metainformationsmodell kann unter Umständen auch völlig ungeeignet sein, wenn sich Teile der Daten nicht abbilden lassen.

4.2.2 Mögliche Datenhaltungssysteme

Zur Realisierung der Datenhaltungsschicht stehen verschiedene Technologien, Produkte und Konzepte zur Wahl: Neben dateibasierten Ansätzen mit oder ohne Versionsunterstützung bieten sich relationale Datenbanksysteme als robuste Grundlage für die Datenhaltung an. Durch objektrelationale Erweiterungen ergeben sich neue Möglichkeiten zur effizienten Unterstützung der Speicherung von komplexeren Objekten. Mit der zunehmenden Verbreitung von XML als universellem Metaformat für den Datenaustausch sind auch auf dieser Technologie basierende Ansätze denkbar. Die wichtigsten zur Wahl stehenden Systeme sollen im

Folgenden kurz vorgestellt werden und die jeweiligen Vor- und Nachteile für ihren Einsatz als Repository des virtuellen Labors diskutiert werden.

4.2.3 Dateibasierte Datenhaltung

Die Speicherung der Experimentdaten als sogenannte „Flat Files“, also Dateien des Betriebssystems, ist die einfachste Variante zur Realisierung der Datenhaltungskomponente. Die meisten Anwendungsprogramme nutzen Dateien zur Speicherung ihrer Daten. Genauso könnten für das virtuelle Labor eigene Dateiformate für Experimentaufbau und Experimentdaten entwickelt oder auf existierende Formate zurückgegriffen werden.

4.2.3.1 Allgemeine Eigenschaften dateibasierter Ansätze

Durch den Verzicht auf aufwendige Zwischenschichten erlauben dateibasierte Ansätze eine fast ungeminderte Nutzung der Leistungsfähigkeit des Datenträgers, wenn es gilt, große Datenmengen zusammenhängend zu speichern, wie es für die Experimentdaten erforderlich ist. Da jedoch keine Hilfsstrukturen existieren, erfolgt der wahlfreie Zugriff auf einzelne Datensätze dieses Datenstroms üblicherweise durch langwieriges sequentielles Durchsuchen der Datei. Dadurch wird das Leistungsverhalten beim Retrieval sehr schnell inakzeptabel, was sich besonders in der Analyse-, aber auch in der Vorbereitungsphase bemerkbar macht.

Auch die Skalierbarkeit ist problematisch. Zwar können bei Platz- oder Performancemangel weitere Datenträger hinzugefügt werden, doch dies erfolgt im Allgemeinen nicht transparent für den Benutzer. Ansätze wie RAID-Verfahren (Redundant Array of Inexpensive Discs) ermöglichen durch die Verbindung mehrerer Datenträger zu einem logischen Laufwerk eine Medientransparenz, verbessern die Leistungsfähigkeit gegenüber einzelnen Platten und bieten zugleich einen gewissen Schutz vor Mediendefekten. Die spätere Erweiterung eines solchen Arrays ist jedoch schwierig und oft nur durch einen kompletten Austausch möglich, der ein Herunterfahren des Systems erfordert.

Auch RAIDs schützen die Daten zudem nicht vor Schäden durch die Anwendung selbst. Es gibt keine Vorkehrungen zum Schutz der Dauerhaftigkeit und Integrität der Daten bei Änderungen. Kommt es beim Überschreiben einer Datei mit einer geänderten Version eines Entwurfsobjekts zu einem Fehler, ist die so teilweise neu geschriebene Datei mit großer Wahrscheinlichkeit anschließend unbrauchbar. Eine Möglichkeit zur Vermeidung dieses Problems wäre die Verwendung eines Systems zur Verwaltung von Versionen von Entwurfsobjekten in Dateiform, wie sie beispielsweise das Concurrent Versions System (CVS) bietet. Die sonstigen prinzipiellen Nachteile dateibasierter Verfahren lösen jedoch auch diese Systeme nicht.

So ist auch die Unterstützung für den nebenläufigen Zugriff auf Daten minimal, der Zugriffschutz erfolgt grundsätzlich durch Sperren auf Ebene einer kompletten Datei, also im Falle des virtuellen Labors zum Beispiel auf einem kompletten Entwurf, wodurch kooperatives Arbeiten erheblich erschwert wird. Teilt man große Objekte deshalb auf mehrere Dateien auf, sinkt dafür die Leistungsfähigkeit und die Effizienz der Speichernutzung. Einige dateibasierte Anwendungen verfügen über Mechanismen die Kooperation durch Kommunikation

oberhalb der Datenhaltungsschicht ermöglichen. Dies funktioniert jedoch nur in homogenen Umgebungen und erhöht die Komplexität der Anwendung.

Größter Vorteil dateibasierter Ansätze ist ihre Abbildungsmächtigkeit, die nur vom Format der Dateien beschränkt wird. Dies kann frei gewählt und so vollständig an die jeweilige Anwendung angepasst werden. Ist jedoch Interoperabilität mit existierenden Anwendungen gefordert, so muss möglicherweise ein vorhandene Dateiformat verwendet werden, was die Flexibilität stark einschränkt. Das Ändern von Dateiformaten zur Unterstützung neuer Funktionen ist nicht unproblematisch. Insbesondere bei Verwendung existierender Formate kommen zu Kompatibilitätsproblemen auch wirtschaftliche Interessen hinzu, was im Folgenden an einigen Beispielen verdeutlicht wird.

4.2.3.2 Dateiformate

Viele existierende Anwendungen aus dem CAx-Bereich nutzen Dateien zur Speicherung der Entwurfsobjekte. Durch die große Vielfalt an Anwendungen, die alle ihre hauseigenen Formate mitbringen, ist eine große Menge von konkurrierenden, zueinander inkompatiblen Dateiformaten entstanden. Jeder Hersteller versucht naturgemäß, die Verbreitung seines Formates zu fördern, um so die Käuferbasis für das eigene Produkt zu vergrößern. Aufgrund der thematischen Ähnlichkeit, welche die Experimentplanungskomponente des virtuelles Labors im Bereich Mehrphasensysteme mit dem CAD-Bereich aufweist, sollen die Probleme, die diese Menge an inkompatiblen Dateiformaten mit sich bringt, kurz am Beispiel des großen Marktes der CAD-Anwendungen verdeutlicht werden. In dieser Domäne hat sich das AutoCAD-Format als De-facto-Standard etabliert, was an der großen Zahl von Entwürfen liegt, die in diesem Format vorliegen. Die meisten konkurrierenden Produkte können dieses Format zumindest lesen. Es existiert in zwei Ausführungen, die an der Dateierweiterung unterschieden werden können. Während das neuere DWF-Format lediglich ein Betrachten des Entwurfs mit Hilfe eines Viewers erlaubt, lässt DWG als natives Format von AutoCAD auch eine Weiterverarbeitung zu. Da es sich um ein proprietäres Format handelt, unterliegt es vollständig der Kontrolle des AutoCAD-Entwicklers Autodesk. Mit jeder neuen Version des Produkts wird auch DWG erweitert, sodass neue Inkompatibilitäten entstehen und die Mitbewerber gezwungen sind, die Neuerungen zu integrieren. Das explizit zum Datenaustausch mit anderen Systemen vorgesehene Data Exchange Format DXF stammt ebenfalls von Autodesk, und hat ähnliche Defizite [ODDXF]. Die Spezifikationen werden häufig mit deutlicher Verzögerung veröffentlicht, sind unvollständig oder unpräzise. Zudem ist auch dieses Format auf die Interna und das Konzept des Produkts AutoCAD optimiert.

Um diese unbefriedigende Situation zu ändern, gibt es mehrere Ansätze, freie Standardformate zu etablieren. Ein Beispiel hierfür ist das IGES-Format (Initial Graphics Exchange Specification) [IGES], dessen Entwicklung bis auf das Jahr 1979 zurückgeht. Aktuell liegt es in Version 5.3 vor. IGES ist ein ANSI-Standard und gilt als eines der am weitesten verbreiteten herstellerneutralen Dateiformate. Die Spezifikation ist gegen Gebühr erhältlich. Es erlaubt die Speicherung zwei- und dreidimensionaler Entwurfsobjekte als Liste von Entities. Man unterscheidet geometrische Entities, die Punkte, Kurven, Oberflächen und Körper beschreiben, sowie nicht-geometrische Entities, mit denen Gruppen von Entities mit Anmerkungen oder Charakteristiken versehen werden können. Ein CAD-System mit IGES-

Unterstützung bildet seine interne Darstellung auf diese Entities ab. IGES wird fast ausschließlich in einer ASCII-Darstellung genutzt, die zu großen Dateien und erheblichen Leistungsproblemen führt. Auch umfasst es nicht alle Aspekte einer CAD-Produktbeschreibung und unterstützt nicht alle möglichen Fähigkeiten von CAD-Systemen. Da nicht jedes System mit IGES-Unterstützung alle Entities versteht, besteht zudem die Gefahr, dass Informationen beim Transport verloren gehen oder verfälscht werden. Es gibt einige Ansätze, die eine Beschränkung auf eine Untermenge der IGES-Spezifikation vorsehen, der bekannteste ist VDA IGES, der von der deutschen Automobilindustrie entwickelt wurde (auch VDA-FS, Verband der Automobilindustrie – Flächenschnittstelle).

Als Nachfolger von IGES wurde das STEP-Format (STandard for the Exchange of Product data) entwickelt. Die wichtigsten Ziele wurden im Hinblick auf die Erfahrungen mit IGES definiert: STEP sollte vollständig sein, also eine komplette Produktbeschreibung umfassen, und es muss Raum für domänenspezifische Erweiterungen bieten. Wichtiges Ziel war auch eine Verbesserung der Speicherplatz- und Verarbeitungseffizienz, den Rückgriff auf bestehende Standards soweit möglich und eine Reduktion der Redundanz in der Art, dass es nur noch eine mögliche Beschreibung für ein gegebenes Objekt gibt. Die Erkenntnis, dass ein komplexer Standard kaum vollständig umgesetzt werden würde, war eine der wichtigsten Lehren aus der Praxis mit IGES. Deshalb wurden für STEP von Anfang an Untermengen, so genannte Anwendungsprofile definiert. STEP eignet sich im Vergleich zu IGES nicht nur für Drahtgitter und Flächenmodelle, sondern auch für Volumendarstellungen. Die anfänglichen Probleme mit STEP durch mangelnder Reife von Produkten und Importfiltern werden zunehmend behoben, bei einigen Systemen fehlt die Unterstützung von STEP jedoch noch völlig. Eine Bewertung und Empfehlungen zur Verwendung neutraler CAD-Formate findet sich in [Bryan2002] und [ProSTEP].

Anhand der hier geschilderten Problematik wird deutlich, dass bei der Entscheidung für ein existierendes Dateiformat zahlreiche Kompromisse eingegangen werden müssen. So muss das anwendungsinterne Datenmodell aufwendig an das anderer Anwendungen angepasst werden, wenn deren Formate genutzt werden sollen. Setzt man eines der anwendungsunabhängigen Formate ein, erweist sich eine vollständige Implementierung wegen des Umfangs der Spezifikationen als äußerst aufwendig. Häufig fehlen in existierenden Formaten für die eigene Anwendung erforderliche Features. Fügt man seine eigenen Erweiterungen zu einem Format hinzu, bleibt der einzige Vorteil, die Kompatibilität zu bestehenden Systemen, auf der Strecke. Selbst wenn ein existierendes Format alle anfangs benötigten Eigenschaften aufweist, kann früher oder später eine Änderung oder Erweiterung der Anwendung gewünscht sein, die das gewählte Format nicht erfüllen kann. Die Folge ist auch hier entweder eine proprietäre Erweiterung oder die aufwendige Umstellung auf ein alternatives Format. Die Verwendung eines der angesprochenen Dateiformate zur Speicherung von virtuellen Experimenten oder Experimentbausteinen steht vor dem Hintergrund der angesprochenen Probleme außer Frage. Da jedoch sehr viele Geometriedaten in Dateiform vorliegen, ist für das virtuelle Labor ein Import einiger dieser Dateiformate in die interne Darstellung vorzusehen.

Die Entwicklung eines eigenen proprietären Dateiformats scheint eine gangbare Alternative zur Verwendung eines existierenden Formats. Eine aufwendige Adaption entfällt, da es ganz auf die Ansprüche der eigenen Anwendung zugeschnitten werden kann. Insbesondere die

Erweiterung ist so problemloser möglich. Da Dateien jedoch zumeist als ganzes eingelesen werden, bedeutet eine Veränderung des Dateiformats, dass auch eigentlich nicht direkt von den Änderungen betroffene Anwendungsteile unter Umständen angepasst werden müssen.

Aufgrund dieser und der eingangs erwähnten Nachteile sind dateibasierte Ansätze zur Realisierung des Repositories für das virtuelle Labor ungeeignet.

4.2.4 Datenhaltung in einem relationalen Datenbanksystem

Relationale Datenbanken (RDBMS) bilden die Grundlage fast jedes modernen betrieblichen Informationssystems. Aufgrund ihrer langen Verfügbarkeit handelt es sich bei heutigen RDBMS um robuste und zugleich hochoptimierte Systeme mit einem hervorragenden Leistungsverhalten. Indexstrukturen erlauben den effizienten wahlfreien Zugriff auf einzelne Datensätze, erfordern jedoch erhöhten Aufwand beim Speichern der Daten, sodass die Leistung beim kontinuierlichen Schreiben großer Datenmengen bei gleicher Hardware geringer ausfällt als die dateibasierter Ansätze. Dieser Leistungsverlust wird jedoch durch den Gewinn bei wahlfreiem Retrieval mehr als aufgewogen. Zudem können Datenbanksysteme durch Abstraktion von der Medienstruktur sehr viel leichter Vorteile aus der parallelen Nutzung mehrerer Datenträger ziehen. Viele Produkte bieten zudem Möglichkeiten für den Aufbau verteilter Datenbanken, sodass eine gute Skalierbarkeit gegeben ist.

Das Transaktionsparadigma garantiert die Konsistenz der Daten auch in Mehrbenutzerumgebungen und bei System- oder Anwendungsfehlern. Mit SQL existiert eine mächtige deskriptive Anfragesprache, die weitgehend standardisiert ist. Dadurch ist ein Wechsel des Datenbankanbieters zumindest theoretisch problemlos möglich. In der Praxis sind jedoch viele verschiedene SQL-Dialekte entstanden, die sich insbesondere bei der Verwendung mächtigerer Funktionen stark voneinander unterscheiden können. Bei Verwendung dieser Funktionen wird daher die Portabilität deutlich erschwert. SQL92 als letzter rein relationaler Standard bildet jedoch einen „kleinsten gemeinsamen Nenner“, der von allen wichtigen Produkten größtenteils unterstützt wird.

Das relationale Datenmodell erlaubt eine sehr natürliche Abbildung von geschäftlichen Daten, erfordert jedoch größeren Aufwand beim Einsatz als Repository in Entwurfsanwendungen. So können komplexe Objekte nicht als Ganzes gespeichert werden, sondern müssen in ihre Bestandteile zerlegt und auf mehrere Relationen verteilt werden. Dies bedingt eine aufwendige Rekonstruktion der komplexen Objekte und die Rückabbildung auf das Format der Anwendung. Weiterhin bilden relationale Systeme in ihrer „Miniwelt“ immer den aktuellen Zustand der Daten ab, sie bieten keine natürliche Versionsunterstützung oder Verfahren zur Speicherung der Entwurfshistorie. Diese müssen daher durch eigene Mechanismen nachgerüstet werden. Trotz dieser Nachteile für den Einsatz in Entwurfsanwendungen wie dem virtuellen Labor erfüllen relationale Datenbanksysteme die eingangs formulierten Anforderungen an eine Datenhaltungskomponente sehr gut.

4.2.5 Objektorientierte Datenbanksysteme

Mit der zunehmenden Verbreitung des objektorientierten Programmierparadigmas entstanden Bemühungen, objektorientierte Konzepte auch auf Datenbanksysteme zu übertragen. Erklärtes Ziel war es, Speicherung und Retrieval von komplexen Objekten ohne deren Zerlegung zu ermöglichen, das Hauptproblem beim Einsatz relationaler Datenbanken. Die Anfang der 90er Jahre verfügbaren objektorientierten Systeme konnten sich jedoch nicht gegen die etablierten relationalen Datenbanken durchsetzen. So sind viele Systeme eng an eine Programmiersprache gebunden. Defizite im Bereich der Mächtigkeit von Anfragen und ihrer Optimierung, der Transaktionsverarbeitung und Probleme mit der Skalierbarkeit verhinderten einen Markterfolg. Lediglich in Nischenmärkten konnten sie eine nennenswerte Verbreitung erzielen. Objektorientierte Datenbanksysteme werden daher als mögliche Grundlage für ein Repository des virtuellen Labors nicht weiter berücksichtigt.

4.2.6 Objekt-relationale Datenbanksysteme

Um die Defizite der relationalen Datenbanksysteme im Bezug auf die Speicherung komplexer Objekte zu beheben, aber zugleich ihre Vorteile zu erhalten, haben große Datenbankhersteller ihre Systeme um objekt-relationale Konzepte erweitert. Diese erlauben unter anderem die Definition von komplexen Datentypen, Typ- und Tabellenhierarchien und bieten neue Basisdatentypen wie Boolean, Character Large Objects (CLOB) und Binary Large Objects (BLOB) sowie Kollektionstypen. Mit SQL:1999 wurde ein erster Standard für objekt-relationale Datenbanksysteme geschaffen. Bisher wurde dieser Standard jedoch von keinem der etablierten Datenbankhersteller vollständig implementiert, stattdessen bringt jedes objekt-relationale DBMS einen proprietären SQL-Dialekt und eigene objekt-relationale Konzepte mit. Dadurch sind Anwendungen, welche die OR-Fähigkeiten der Produkte nutzen, oft nur mit hohem Aufwand auf ein anderes Produkt portierbar. Einen Überblick über die wesentlichen Unterschiede zwischen den Systemen von Oracle, IBM DB2 und Informix gibt [Tü2003].

4.2.6.1 Objekt-relationale DBMS am Beispiel IBM DB2

DB2 [DB2] ist ein Datenbankmanagementsystem der IBM Corporation. Es ist für zahlreiche Plattformen verfügbar, neben zahlreichen Unix-Derivaten (AIX, HP-UX, Solaris) auch für das freie Betriebssystem Linux und für die NT-basierten Betriebsysteme von Microsoft (Windows NT, Windows 2000, Windows 2003).

Ursprünglich ein rein relationales System wurde DB2 im Laufe der Jahre zunehmend um objekt-relationale Konzepte erweitert. Neben erweiterten Datentypen wie Large Objects (LOBs) werden benutzerdefinierten Typen und Vererbungsmechanismen unterstützt. Zur Verlagerung von Anwendungslogik in das Datenbanksystem können benutzerdefinierte Funktionen entwickelt werden. Als Implementierungssprache stehen neben IBM SQL inzwischen auch C/C++ und Java zur Verfügung. Aktive Datenbanken können mit Hilfe von Triggern realisiert werden.

Extender

Mit Hilfe von so genannten Extendern kann die Basisfunktionalität der DB2 erweitert werden. Der Net Search Extender rüstet Verfahren zur Volltextsuche auf großen Dokumenten nach, die sowohl innerhalb der Datenbank, in anderen Datenbanken oder extern als Dateien und in verschiedenen Formaten (zum Beispiel XML oder HTML) vorliegen können. Durch boolsche Operatoren können einzelne Worte oder Phrasen verknüpft oder ausgeschlossen werden. Der Net Search Extender unterstützt 37 Sprachen, ermöglicht die Angabe von Stop-Wörtern zur effizienteren Indexierung, Nachbarschaftssuche „Fuzzy-Search“ für ähnliche Wörter sowie die Unterstützung von Thesauri.

Der Spatial Extender erlaubt die Integration von räumliche Daten durch Ergänzung von neuen Datentypen und Funktionen. So sind Anfragen über geographische Nachbarschaft oder Distanzen möglich. [HvT2003] beschreibt die Nutzung des Spatial Extender für das virtuelle Labor Siedlungswasserwirtschaft. Wegen fehlender Unterstützung für dreidimensionale Informationen und der Fokusierung auf den Einsatz in geographischen Informationssystemen (*geographic information system*, GIS) konnte der Spatial-Extender nicht gewinnbringend für das virtuelle Labor Verfahrenstechnik eingesetzt werden. Zur Integration des zunehmend an Bedeutung gewinnenden Metadatenformats XML ergänzt der XML Extender Datentypen und Methoden zur Speicherung von XML in der Datenbank und zur Einbindung externer XML-Dateien. Er wird in Abschnitt 4.2.7.2 näher vorgestellt.

4.2.6.2 Eignung objekt-relationaler Datenbanken

Wegen der beschriebenen Heterogenität und der nicht immer ausgereiften Funktionen konnten sich die objekt-relationalen Konzepte trotz der großen Marktbedeutung der ORDBMS als Nachfolger von RDBMS-Produkten bisher nicht auf breiter Front durchsetzen. Oft nutzen Anwendungen weiterhin nur den relationalen Teil dieser Produkte, ergänzt um einzelne, für die jeweilige Domäne nützliche objekt-relationale Bestandteile. Die Unterstützung für komplexe Objekte bleibt begrenzt und wird durch einen zum Teil erheblich gestiegenen Aufwand bei Anfragen und Modifikation der Daten erkauft.

4.2.7 XML-Datenbanken

Mit der Entwicklung der Extensible Markup Language (XML) zum De-facto-Standard für den Datenaustausch in heterogenen Rechnerumgebungen entstand zunehmend der Bedarf, XML-Dokumente in effizienter Weise auch als Grundlage für die persistente Datenhaltung zu nutzen. So wurden zahlreiche Produkte entwickelt, die als XML-Datenbanken bezeichnet werden. Sie lassen sich unterteilen in native XML-Datenbanken und Datenbanken mit XML-Unterstützung (*XML-enabled*-Datenbanken). [Bo2003] gibt einen guten Überblick zum Thema XML-Datenbanken.

4.2.7.1 Native XML-Datenbanken

Auch wenn die Bezeichnung dies nahelegt, muss eine native XML-Datenbanken nicht notwendigerweise XML zur Speicherung der Daten nutzen. Viele existierende Systeme bauen beispielsweise auf relationalen Datenbanken auf. Gemäß der Definition nach [St2001] zeichnet sich eine native XML-Datenbank durch drei wesentliche Elemente aus:

- Sie definiert ein Modell für XML-Dokumente, das als Grundlage für Speicherung und Retrieval dient. Das Modell umfasst Elemente einschließlich ihrer Reihenfolge und ihrer Attribute sowie PCDATA.
- Als logische Speicherungseinheit dient ein XML-Dokument.
- Sie erfordert kein bestimmtes zugrundeliegendes physikalisches Speicherungsmodell, es kann beispielsweise auf einem relationalen, hierarchischen oder objektorientierten Datenbanksystem basieren oder ein proprietäres dateibasiertes Format verwenden.

Es gibt inzwischen zahlreiche kommerzielle und freie Produkte. Bekanntestes Beispiel ist der Tamino XML Server der Software AG.

4.2.7.2 Datenbanken mit XML-Erweiterungen

Viel Hersteller klassischer (objekt-)relationaler Datenbanken haben die zunehmende Bedeutung von XML erkannt und ihre Produkte um Mechanismen für die Unterstützung der Speicherung von XML erweitert. Sie werden im Allgemeinen als *XML-enabled* bezeichnet. Die Art der Unterstützung variiert stark. IBM stellt den XML Extender zur Verfügung, Oracle nennt seine Erweiterung XML DB. Beide erlauben ein dokumenten- oder datenzentriertes Speichern von XML in der Datenbank. IBM Informix erhält XML-Funktionalität über das Web- und das XSLT-Datablade.

Als Beispiel für Datenbanken mit XML-Erweiterungen soll kurz der IBM DB2 XML Extender [XMLExt] vorgestellt werden. Im Kern besteht er aus einer Menge von neuen Datentypen und Methoden zur Speicherung, Verarbeitung und Ausgabe von XML. Es werden zwei grundsätzliche Konzepte unterschieden:

XML Columns speichern komplette XML-Dokumente als Attribute in speziellen XML-Datentypen. XMLVARCHAR eignet sich für kleinere, XMLCLOB auch für sehr große Dokumente im Gigabytebereich. XMLFILE ermöglicht die Einbeziehung von nicht in der Datenbank gespeicherten XML-Dateien. Um eine effiziente Suche innerhalb der Dokumente zu ermöglichen, kann der Benutzer mit Hilfe von DAD (*Document Access Definition*)-Dateien angeben, auf welche Elemente und Attribute der Dokumente besonders häufig zugegriffen wird. Diese werden dann auf sogenannte s*ide tables* abgebildet, die bei Bedarf indexiert werden können. Dieses Verfahren eignet sich für Dokumente, die unverändert erhalten bleiben sollen und nur selten aktualisiert, sondern hauptsächlich gelesen werden.

Änderungsoperationen sind einerseits durch Austausch des kompletten Dokuments möglich, anderseits durch Änderung einzelner Elementinhalte und Attributwerte mit Hilfe einer Update-Funktion durch Angabe des zu ändernden Dokumentteils mittels XPath-Angabe und des neuen Wertes. In beiden Fällen wartet der XML-Extender die side tables automatisch. Ein wichtiger Vorteil der XML-Columns ist die round-trip-Fähigkeit, was bedeutet, dass ein Dokument unverändert entnommen werden kann, da insbesondere die Elementreihenfolge oder Kommentare erhalten bleiben. Das Retrieval einzelner Elemente oder Attribute erfolgt ebenfalls mit Funktionen, die neben der XML-Column einen XPath-Ausdruck entgegennehmen. Mit diesen Funktionen ist auch die kombinierte Ausgabe von relationalen Daten und Daten aus XML-Spalten mit SQL möglich.

XML Collections sind die zweite Variante, um XML-Dokumente in DB2 abzulegen. Die Dokumente werden dazu nicht als ganzes in Form getagter Daten gespeichert, sondern werden zerlegt und auf Relationen verteilt. Die Abbildungsvorschrift wird auch in diesem Fall mit Hilfe von DAD-Dateien angegeben. Das Zerlegen von XML-Dokumenten in Tupel und die Rekonstruktion von Dokumenten aus Tabellen erfolgt mit Stored Procedures.

4.2.7.3 XML-Datenbanken am Beispiel Tamino

Der Tamino XML-Server der Software AG war eine der ersten nativen XML-Datenbanken am Markt. Zum Zeitpunkt seiner Entwicklung waren die Standardisierungsbemühungen des W3C im Bezug auf XML-Anfrage- und Updatesprachen noch nicht weit fortgeschritten. In Folge dessen wurden für viele Probleme und Erfordernisse im Umgang mit XML eigene Lösungen entwickelt. So entstand unter anderem die proprietäre Anfragesprache X-Query (man beachte den Bindestrich). Sie basiert auf der XPath-Spezifikation, implementiert diese jedoch nicht vollständig, geht dafür an anderen Stellen über sie hinaus. Einen Überblick über die Erfahrungen mit Tamino als Grundlage für ein webbasiertes Informationsmanagementsystem gibt [SeKr2002]. Dort wurde vor allem die unvollständige Unterstützung wichtiger XML-Eigenschaften bemängelt. So fehlt eine interne Umsetzung von ID/IDREF Attributen, die Verknüpfungen von Inhalten müssen auf einen Tamino-spezifischen Linking-Mechanismus abgebildet werden. Schwerwiegendster Mangel war jedoch die fehlende Unterstützung von verschachtelten Anfragen, was fast dazu geführt hätte, das Projekt auf eine relationale Datenbank umzustellen. Das ebenfalls proprietäre Tamino Schema zur Informationsmodellierung unterstützt nicht die volle Ausdrucksmächtigkeit von DTDs, und eine Schemaevolution erfordert quasi zwingend ein Löschen aller Instanzen. In den neusten Versionen von Tamino wurden viele lange vermisste Standards wie XML Schema und XML Query nachgerüstet, sodass einige der aufgeführten Schwierigkeiten inzwischen entschärft wurden. Bisher existieren jedoch noch wenig Erfahrungen bezüglich Leistungsfähigkeit und Vollständigkeit der Unterstützung dieser neuen Technologien.

4.2.7.4 Eignung von XML-Datenbanken

XML eignet sich nicht nur für die Darstellung komplexer Objekte, es entwickelt seine Stärken besonders bei der Speicherung semistrukturierter Daten, wie sie zum Teil auch für das virtuelle Labor benötigt werden.

Als Haupthindernis beim Einsatz von XML-Datenbanken steht die Heterogenität der existierenden Produkte. So sind durch das Fehlen eines einheitlichen Standards für Speicherung und Retrieval von XML-Dokumenten verschiedene Anfragesprachen und APIs entstanden. Die meisten basieren auf XPath [W3CXP], eine Sprache, die jedoch ursprünglich nicht als Anfragesprache für eine Datenbank entwickelt wurde und daher einige Defizite aufweist. So gibt es keine Möglichkeit für dokumentübergreifende Anfragen, keine Unterstützung für Datentypen und Sortieroperationen. So entstanden wie im zuvor mit Taminos X-Query geschilderten Fall eine Vielzahl von proprietären XPath-Erweiterungen.

Mit XQuery befindet sich zur Zeit ein Kandidat für die Rolle als Standardanfragesprache in der Entwicklung. Die Spezifikation wurde mit kleinen Änderungen von Quilt übernommen, die ihrerseits XPath integriert. Die Spezifikation von XQuery hatte zum Zeitpunkt der Erstellung dieser Arbeit noch Draft-Status [W3CXQ].

Ein weiteres Problem von XML-Datenbanken, nämlich das Fehlen einer Sprache für die Durchführung von Updates, wird aber auch in Version 1.0 von XQuery nicht gelöst. Anfangs war dazu häufig das Retrieval eines Dokuments, die anschließende Änderung mit einem geeigneten API wie SAX oder DOM und das anschließende Wiedereinfügen in die Datenbank erforderlich. Viele Produkte bieten proprietäre APIs zur Manipulation von Dokumenten innerhalb des Servers an. Einige Open-Source-Produkte nutzen das XUpdate-API der XML:DB-Initiative [XMLDB].

Auch bezüglich des Leistungsverhaltens und der Skalierbarkeit von XML-Datenbanken existieren noch kaum Erfahrungen. Mit XMach-1 [XMach], [BöRa2001a] ist ein erster Benchmark-Ansatz für native und XML-enabled-Datenbanken entstanden, dem inzwischen weitere gefolgt sind [XBench]. Die bisherigen Erfahrungen sind geprägt von Problemen wegen der zuvor geschilderten mangelnden Standardisierung und dem unausgereiften Stand vieler Produkte [BöRa2001b], die ermittelten Leistungsdaten sind nicht ermutigend. Dadurch sind ausreichende Leistungsfähigkeit und Skalierbarkeit mit XML-Datenbanken nur schwer zu gewährleisten.

4.2.8 Wahl der Datenbanktechnologie für das virtuelle Labor

Nach der Vorstellung der möglichen Produkte oder Technologien, die als Grundlage für das Repository des virtuellen Labors in Frage kommen, und ihrer Bewertung bezüglich der zu Beginn formulierten Anforderungen an die Datenhaltungskomponente, gilt es ihre Vor- und Nachteile abzuwägen und eine Empfehlung auszusprechen.

Dateibasierte Ansätze scheiden wegen ihrer erheblichen Defizite im Bezug auf kooperatives Arbeiten sowie Datensicherheit und -konsistenz aus. Zudem wäre wegen der Defizite existierender Formate und dem Umfang des virtuellen Labors die Entwicklung eines eigenen Dateiformats unvermeidlich, sodass der einzige verbleibende Vorteil entfällt: der einfache Datenaustausch mit existierenden Anwendungen. Da das virtuelle Labor jedoch keine Insellösung sein darf, sollten für wesentliche Teilaspekte Im- und Export-Filter vorgesehen werden. Insbesondere können existierende Geometriedaten als Grundlage für den Aufbau einer Bau-

steinbibliothek genutzt oder Experimentdaten für die Weiterverarbeitung in andere Anwendungen übernommen werden.

XML-Datenbanken sind als relative neue Technologie noch zu wenig standardisiert, sodass man sich bei einer Entscheidung für die native Speicherung von XML fast zwangsläufig auf ein Produkt festlegen müsste. Zwar kann die Standardisierung von XQuery als weitgehend abgeschlossen angesehen werden, und man kann davon ausgehen, dass es sich am Markt etablieren wird. Es fehlt jedoch weiterhin eine Standard-Sprache für Änderungsoperationen. Die enormen Unterschiede zwischen den Produkten erschweren eine spätere Migration erheblich. Aufgrund der unsicheren Marktsituation kann auch die Wahl des technisch überzeugendsten Produkts letztlich eine Fehlentscheidung sein, wenn seine Weiterentwicklung überraschend eingestellt wird. Insbesondere bei einem langfristig angelegten Projekt wie dem virtuellen Labor erscheint dieses Risiko nicht tragbar. Wenn zu einem späteren Zeitpunkt der Standardisierungsprozess der Anfrage- und auch Update-Sprache abgeschlossen ist und entsprechende Produkte verfügbar werden, sollte die Verwendung von nativen XML-Datenbanken erneut geprüft werden.

So bleiben schließlich die relationalen Datenbanken, deren wichtigste kommerzielle Vertreter in den letzten Jahren auch als „objekt-relational“ beworben werden. Im Gegensatz zu den XML-Datenbanken gibt es hier bereits einen Standard mit SQL:1999, dessen Nachfolger SQL:2003 kurz vor der Verabschiedung steht. Alle großen Hersteller von klassischen relationalen Datenbanken bewerben die objekt-relationalen Eigenschaften ihrer Produkte, ohne jedoch SQL:1999 vollständig und standardkonform zu erfüllen. Wenn die leichte Portierbarkeit auf andere ORDBMS als wichtig erachtet wird, muss beim Einsatz ihrer nicht-standardkonformen Erweiterungen, der eine verstärkte Bindung an das jeweilige Produkt bedeutet, sorgfältig zwischen den so erzielbaren Vorteilen und dem möglichen Aufwand bei einer späteren Umstellung auf ein anderes Produkt abgewogen werden. Bedeutet die Änderung einzelner Anfragen wegen nicht-standardkonformer Schlüsselwörter nur minimalen Aufwand, kann beispielsweise der spätere Verzicht auf die Nutzung einer proprietären XML-Erweiterung zeitintensive Umstrukturierungen erforderlich machen.

Trotz der beschriebenen Defizite relationaler DBMS beim Umgang mit komplexen Objekten erscheinen sie zum jetzigen Zeitpunkt am Besten geeignet, um die Grundlage des Repositories des virtuellen Labors zu bilden. Da alle großen RDBMS objekt-relationale Funktionalität mitbringen, kann deren Nutzung unter Berücksichtigung der obigen Prämissen in Betracht gezogen werden. Durch die zunehmende Integration von XML-Techniken in OR-Datenbanken und die Etablierung von Standards für deren Nutzung besteht auch die Möglichkeit einer späteren sanften Migration zu XML.

4.3 Datenmodell für Experimentbausteine

Von zentraler Bedeutung für ein virtuelles Labors sind die Experimentbausteine (kurz: Bausteine), die als Grundelemente für den Aufbau eines Experiments dienen. So können sie vom Benutzer in Analogie zur Durchführung eines realen Experiments aus einer Bibliothek von

verfügbaren Bausteinen entnommen und zu einem Experimentaufbau oder einer „virtuellen Apparatur“ zusammengefügt werden.

Um in einem virtuellen Experiment einsetzbar zu sein, muss ein Datenmodell für Experimentbausteine entwickelt werden, mit dem alle wesentlichen Aspekte beschrieben werden können. Zugleich muss das Modell einen hohen Grad an Flexibilität aufweisen, um den sehr unterschiedlichen Bausteintypen Rechnung zu tragen.

Im folgenden Kapitel sollen zunächst die wesentlichen Aspekte eines solchen Bausteinmodells aus den in Kapitel 2.2 beschriebenen Anforderungen hergeleitet werden. Anschließend werden verschiedene Realisierungsmöglichkeiten zur Abbildung dieser Aspekte auf ein Datenmodell vorgestellt und ihre Vor- und Nachteile dargelegt. Bei dieser Abbildung wird wie in Kapitel 4.2 begründet das relationale Datenmodell zugrunde gelegt, ergänzt um objektrelationale Konzepte.

4.3.1 Wichtige Konzepte des Bausteinmodells

Bevor näher auf die Elemente des Bausteinmodells eingegangen wird, sollen zunächst einige seiner Grundprinzipien kurz erläutert werden.

4.3.1.1 Trennung Systemsicht - Benutzersicht

Ein wesentliches Konzept des virtuellen Labors ist die in Anforderung K1 (Trennung Nutzer-/Systemsicht) und detaillierter in Anforderung P2.2 (Bausteinbeschreibung) beschriebene Trennung in eine Nutzer- und eine Systemsicht auf die Elemente eines virtuellen Labors. Daher ist eine Berücksichtigung dieser verschiedenen Sichtweisen auch bei der Beschreibung der Bauteile erforderlich. Dabei ist neben einer explizit getrennten Beschreibung der beiden Sichten auch ein Ableiten der Nutzersicht oder einzelner ihrer Teile aus der Systemsicht denkbar. In jedem Fall stellt die Nutzersicht einen vereinfachten Ausschnitt der Systemsicht dar, sodass zunächst deren wesentliche Elemente spezifiziert werden, um schließlich bei der Beschreibung der Aspekte der Nutzersicht auf diese Bezug nehmen zu können.

4.3.1.2 Trennung Bausteintyp – Bausteininstanz

Die Unterscheidung von Bausteintyp und Bausteininstanz wurde schon bei der Entwicklung des gemeinsamen Kerndatenmodells in Kapitel 3 als Grundsatz aufgegriffen. Die Verwendung der Begriffe erfolgt analog zu den Begriffen Klasse und Instanz in der objektorientierten Programmierung. Ein Bausteintyp beschreibt die allgemeinen Eigenschaften eines Bausteins wie die Art und Anzahl seiner Eigenschaften, seinen internen Aufbau und die Auswirkungen der Änderung von Eigenschaften auf diesen internen Aufbau. Soll ein solcher Bausteintyp genutzt werden, muss er instanziert werden. Dazu ist es erforderlich, alle verpflichtenden Parameter gemäß Anforderung P2.2.2.2.4 mit Werten zu versehen, analog zur Instanzierung einer Klasse, bei der alle Instanzvariablen mit Werten versehen werden. So wie sich am grundlegenden Verhalten einer Objektinstanz durch diese Wertezuweisung nichts ändert,

bleiben auch die Eigenschaften eines Bausteintyps für jede seiner Instanzen im Rahmen der Parametrisierung erhalten.

4.3.1.3 Speicherung von Objekten

Beim Entwurf eines objekt-relationalen Datenbankschemas mit Verwendung komplexer Objekte, wie sie in Entwurfsanwendungen wie der Vorbereitungskomponente des virtuellen Labors vorkommen, hat der Entwickler die Entscheidung zu treffen, bis zu welchem Grad er seine Objekte zerlegen, also auf einzelne Elemente des Schemas abbilden und so der Datenbank zugänglich machen möchte, beziehungsweise in wie weit er diese Objekte monolithisch in einer nicht weiter strukturierten Form in der Datenbank ablegt. Die beiden Extreme dieses Freiheitsgrades, vollständige Dekomposition auf der einen, monolithische Speicherung als large object auf der anderen Seite, erscheinen gleichermaßen ungeeignet:

So mag eine vollständige Zerlegung des Objektes in seine „Atome“ zunächst wünschenswert erscheinen. Wird diese jedoch durchgeführt, ohne in der späteren Anwendung einen Nutzen durch Anwendung der mächtigen Anfrage- und Integritätssicherungsmechanismen des Datenbanksystems auf diese kleinsten Elementen zu ziehen, muss die Entscheidung für einen solchen Ansatz überprüft werden. Vor allem ist der enorme Aufwand für die vollständige Dekomposition und anschließende Rekonstruktion zu berücksichtigen. Dies äußert sich sowohl bei der Entwicklung des Systems durch eine größere Zahl von komplexeren Anfragen und einer aufwendigeren Datenbanklogik als auch im späteren Einsatz, wenn die hohe, aber nicht genutzte Komplexität des Schemas zu einem größeren Ressourcenverbrauch für die Anfrageverarbeitung und -auswertung führt.

Der vollständige Verzicht auf eine Zerlegung der Anwendungsobjekte in der Datenbank und ihre Speicherung als large object in einem anwendungsspezifischen Binär- oder Textformat stellt das andere mögliche Extrem dar. Zwar ist so der datenbankseitige Aufwand bei der Rekonstruktion minimal, da diese ohne DB-Beteiligung vollständig in die Anwendung verlagert wird. Zugleich verzichtet man so jedoch auf jegliche Möglichkeit zur leistungs- und effizienzsteigernden Nutzung der Fähigkeiten des Datenbanksystems. Insbesondere ist keine Vorauswahl der Objekte durch Angabe von Filtern auf ihren Eigenschaften möglich, sodass der Vorteil der einfacheren Rekonstruktion durch die Bewegung eigentlich nicht benötigter Daten ins Gegenteil verkehrt wird.

Statt der beiden Extreme sollte eine Mischform gewählt werden, bei der eine Zerlegung so weit erfolgt, dass sie sowohl eine Vorauswahl der Objekte mit DB-Mechanismen, die Unterstützung von Suchanfragen der Anwendung und möglicherweise auch zu einem gewissen Grad eine Konsistenzprüfung ermöglicht. Sobald ein Grad der Dekomposition erreicht ist, bei dem Teilobjekte nur noch „als Ganzes“ von der Anwendung angefordert und verarbeitet werden, kann auf eine weitere Zerlegung verzichtet werden.

4.3.2 Aspekte des Bausteinmodells

Experimentbausteine in einem virtuellen Labor für Verfahrenstechnik sind komplexe Objekte, die durch eine Vielzahl von Aspekten beschrieben werden. Diese sollen im Folgenden kurz vorgestellt werden.

4.3.2.1 Systemsicht

Die Systemsicht des Bausteinmodells umfasst alle für den Einsatz in einem virtuellen Experiment erforderlichen Informationen über einen Baustein.

Geometrisches Modell

Bei der Beschreibung vieler Anforderungen an die interne Bausteindarstellung im virtuellen Labor wird auf die Geometrie der Bausteine Bezug genommen. Abschnitt 2.2.2.3 (interne Bauteilrepräsentation) beschreibt die Definition der Bausteingeometrie durch Komposition aus geometrischen Primitiven. Das so beschriebene Verfahren ist als *Constructive Solid Geometry,* kurz *CSG* bekannt. Diese Anforderung legt jedoch nicht notwendigerweise die interne Speicherungsstruktur für Geometriedaten fest, sondern beschreibt lediglich die Vorgehensweise zu deren Erstellung und Bearbeitung aus der Sicht des Benutzers. Daher sind neben einer CSG-basierten Speicherung auch andere Datenmodelle zu überprüfen.

Eine wichtiger Anspruch an die Geometriedarstellung ist die in Anforderung P2.2.1.1.1 beschriebene exakte Darstellung der Oberflächen. Eine Approximation von Krümmungen durch Polyeder ist daher für die Speicherung der Geometriedaten nicht akzeptabel.

Da die Bausteingeometrie als Bezugspunkt für andere Aspekte des Bausteinmodells dient, muss das geometrische Modell auch berücksichtigen, auf welche kleinsten Elemente der Geometrie sich diese beziehen. Die Anforderungen nennen hier explizit die Zuordnung von Eigenschaften mindestens zu einzelnen Flächen der Bausteine. Daher muss eine Darstellung der Bausteine diese zumindest auf die Ebene einzelner Flächen zerlegen, um eine solche Zuordnung zu erlauben.

Kompositionsmodell

Anforderung P2.2.1.4 beschreibt die Definition von komplexen Bausteinen durch Zusammenfügen aus anderen Bausteinen. Das Bausteinmodell muss entsprechende Vorkehrungen für die Konstruktion dieser komplexen Bausteine und der Abhängigkeit ihrer Komponenten von ihren Eigenschaften bieten.

Funktionales Modell

Um über die reine Form der Bausteine hinausgehende Eigenschaften von Bausteinelementen zu beschreiben, weist das funktionale Modell einzelnen Elementen des geometrischen Modells weitere Charakteristiken zu, die auf verschiedene Weise das Ergebnis einer Experimentdurchführung beeinflussen.

Dies umfasst sowohl statische Merkmale wie das Material von Oberflächen, als auch dynamische Merkmale, die zum Beispiel bewegliche Elemente eines Bausteins oder Ein- und Auslassöffnungen festlegen.

Das funktionale Modell muss nun eine Zuordnung dieser Merkmale zu Elementen der Geometrie ermöglichen. Zugleich ist eine genaue Spezifikation der Bedeutung der Merkmale erforderlich, also eine Beschreibung ihrer Auswirkungen auf die spätere Experimentdurchführung.

Interaktionsmodell

Die Beschreibungen der Arbeitsweise mit dem virtuellen Labor in Kapitel 2.2.2 sieht vor, dass der Benutzer Experimentbausteine in Analogie zum realen Experiment zu einem Experimentaufbau verbindet. Dazu ist es erforderlich, die gültigen Kombinationsmöglichkeiten der verschiedenen Bausteine zu spezifizieren. Dadurch können fehlerhafte Experimentaufbauten erkannt und der Benutzer beim Aufbau unterstützt und auf Fehler aufmerksam gemacht werden. Zusätzlich ist eine Beschreibung der Art der Interaktion verbundener Bauteile für die spätere Experimentdurchführung erforderlich.

Konsistenzmodell - Systemsicht

Zur Vermeidung von nicht funktionsfähigen Bausteinen ist ein Konsistenzmodell zu entwickeln. Es stellt sicher, dass ein Bausteinentwickler bei der Definition der Systemsicht eines Bausteins keine inkonsistenten Elemente erzeugt.

4.3.2.2 Nutzersicht

Die Nutzersicht bildet einen von den für den Benutzer unwichtigen Details abstrahierenden Ausschnitt der Systemsicht.

Bausteineigenschaften

Wesentlicher Bestandteil der Nutzersicht auf Bausteine sind die in Anforderung P2.2.2.2 ff. beschriebenen Eigenschaften. Diese beinhalten zum einen unveränderliche Informationen über Bausteine, die für den Menschen, nicht jedoch für die Durchführung des virtuellen Experiments eine Bedeutung haben. Zugleich können Parameter jedoch auch experimentrelevante Eigenschaften der Bausteine in abstrahierter Form beschreiben. Diese lassen sich möglicherweise aus der Systemsicht extrahieren, können jedoch variabel gehalten sein. Ihre Manipulation durch den Benutzer hat dann wiederum Auswirkungen auf die Systemsicht.

Konsistenzmodell – Nutzersicht

Mit den veränderlichen Bausteineigenschaften („Parameter“) kann der Designer eines Bausteins dem späteren Benutzer ein mächtiges Werkzeug zur Beeinflussung des Bausteinverhaltens zur Verfügung stellen. Die so manipulierten Bausteine sind dann Ziel der Konsistenzprüfung der Systemsicht und des globalen Konsistenzmodells. Oft lassen sich jedoch Integritätsbedingungen einfacher direkt auf Ebene der Nutzersicht spezifizieren. Das Konsis-

tenzmodell für die Nutzerebene hat daher gemäß Anforderung P2.2.2.2.5 die Aufgabe, dem Designer eines Bausteins Spezifikationsmöglichkeiten für die Bausteinkonsistenz unter Bezugnahme auf die Elemente der Nutzersicht, im Wesentlichen also auf Bausteineigenschaften anzubieten. Dadurch können Fehlgebräuche der Bausteine verhindert werden.

Bausteinvisualisierung

Um dem Benutzer des virtuellen Labors einen intuitiven Umgang mit den Bausteinen zu erlauben, ist eine graphische Repräsentation erforderlich. Neben der naheliegenden Ableitung einer graphischen Darstellung aus dem geometrischen Modell der Systemsicht, möglicherweise ergänzt um Informationen des funktionalen Modells wie Farbe und Textur von Materialien, ist auch eine stark abstrahierende Darstellung durch Symbole denkbar. Diese Aspekte der Bausteinvisualisierung müssen geeignet spezifiziert werden.

4.3.2.3 Verbindung von System- und Nutzersicht

Wie in den vorherigen Abschnitten beschrieben stehen System- und Nutzersicht in enger Beziehung zueinander. Dabei ist nicht nur eine abstrahierende Abbildung von der Systemsicht auf die Sichtweise des Nutzers erforderlich, sondern wie bei der Beschreibung der Bauteileeigenschaften erläutert auch eine Rückabbildung der vom Nutzer veränderten Eigenschaften auf die Systemsicht. Hier gilt es ein Abstraktionsmodell zu entwerfen, das die Spezifikation der Auswirkungen dieser Änderungen erlaubt.

4.3.2.4 Globales Konsistenzmodell

Einige Konsistenzbedingungen sind nicht eindeutig der Systemsicht oder der Nutzersicht zuzuordnen. Sie gelten vielmehr systemweit in beiden Sichten und nicht nur für einen speziellen Baustein. Diese Bedingungen werden im globalen Konsistenzmodell beschrieben.

4.3.3 Geometrisches Modell

Aus der obigen Aufstellung der Anforderungen an ein Bausteinmodell geht hervor, dass der Darstellung der Geometrie von Bauteilen eine entscheidende Bedeutung zukommt, da sich alle weiteren Aspekte einer Bauteilebeschreibung auf Elemente dieser Geometrie beziehen. Die Möglichkeiten zur Beschreibung von dreidimensionalen Geometriedaten sind vielfältig. Im Folgenden sollen einige der bekannten Verfahren kurz vorgestellt und für die Eignung zur Darstellung der Geometrie von Experimentbausteinen bewertet werden. Dabei erfolgt eine Beschränkung auf jene Aspekte, die für die Speicherung der Geometrie in einem Repository relevant sind. Auf eine weitergehende Beschreibung der mathematischen Grundlagen wird verzichtet. Neben der Bewertung wird zudem für die wichtigsten Verfahren ein einfaches Informationsmodell vorgestellt. Zur Darstellung wird auch hier wie schon in Kapitel 3 ein erweitertes Entity-Relationship-Modell verwendet.

4.3.3.1 Beschreibung von Positionen und Rotationen

Für alle im Folgenden vorgestellten Verfahren zur Geometriebeschreibung ist die Spezifikation von Position und Orientierung der beschriebenen Objekte im Raum erforderlich.

Positionen

Die Angabe von Positionen im n-dimensionalem Raum R^n erfolgt mit Hilfe von Koordinatensystemen. Ein Koordinatensystem besteht aus einem Ursprung $O \in R^n$ und einer Menge von Basisvektoren $\{b_1, ... b_n\}$. Sind die Basisvektoren orthogonal, handelt es sich um ein sogenanntes *kartesisches Koordinatensystem*. Die Position eines Punktes P im Raum wird als Summe von Vielfachen der orthogonalen Basisvektoren angegeben:

$$P = x_1 \cdot b_1 + ... + x_n \cdot b_n.$$

$x_1, ... , x_n$ nennt man die *Koordinaten* des Punktes P. Neben den vertrauten kartesischen Koordinatensystemen gibt es noch weitere wie solche mit polaren Koordinaten. Hier werden Punkte im Raum durch Angabe von zwei Winkeln und der Entfernung zum Ursprung angegeben. Diese und weitere alternative Koordinatensysteme eignen sich für bestimmte Spezialanwendungen, für das virtuelle Labor sind jedoch dreidimensionale kartesische Koordinaten die naheliegende Wahl.

Jedes Objekt hat ein lokales Koordinatensystem, relativ zu dem seine geometrischen Eigenschaften angegeben werden, beispielsweise die Position von Eckpunkten. Die Positionierung des Objekts bedeutet nun ein Verschiebung des Ursprungs dieses lokalen Koordinatensystems innerhalb eines übergeordneten oder des globalen Koordinatensystems.

Auch wenn diese Form der Positionsbeschreibung unkritisch erscheint, müssen potentielle Probleme beachtet werden. So bereitet insbesondere die Genauigkeit der Zahlendarstellung im Rechner Probleme. Verwendet man Festkommazahlen, erhält man eine grundsätzlich exakte Darstellung der Koordinaten ohne Rundungsfehler. Allerdings besteht das Problem, dass dadurch ein virtuelles Raster im Raum spezifiziert wird. So können Positionen ausschließlich an den Eckpunkten dieses Rasters angegeben werden. Da ein Raster bei vielen Bearbeitungsschritten hilfreich ist, mag diese Einschränkung zunächst unerheblich erscheinen. Doch schon einfache Operationen wie das Ermitteln eines Schnittpunktes einer Gerade mit einer Ebene kann dazu führen, dass die Koordinaten eines Punktes anzugeben sind, der nicht auf dem Raster liegt.

Da die Dimensionen von Versuchsaufbauten je nach Domäne des virtuellen Labors variieren, aber innerhalb einer Domäne fest sind, kann durch die Definition einer für die jeweilige Anwendung geeigneten Basis-Längeneinheit (Meter, Millimeter, Nanometer) als Abstand zwischen zwei benachbarten Rasterpunkten eine ausreichende Genauigkeit erzielt und die nötige Flexibilität erhalten werden. Wird eine 32-Bit-Ganzzahl zur Darstellung der Koordinaten genutzt, kann man einen Bereich von ca. vier Metern mit sehr hoher Genauigkeit abdecken, wenn man als Einheit Nanometer wählt. Alternativ lässt sich ein Bereich von vier Millionen Kilometern mit der Genauigkeit von einem Meter abdecken. Wird die Basiseinheit zu Anfang jedoch ungünstig gewählt, genügt entweder die Genauigkeit nicht, oder aber der abgedeckte Bereich ist zu klein. Spätestens bei Anwendungen, die eine Kombination aus ho-

her Genauigkeit und großen Dimensionen erfordern, muss die Breite der Festkommazahl auf beispielsweise 64 Bit erhöht werden. Einige spezialisierte Anwendungen erlauben sogar die Verwendung von 256-Bit-Festkommazahlen mit den daraus resultierenden Leistungsverlusten. Damit ist subatomare Genauigkeit in galaktischem Größenmaßstab möglich.

Im Gegensatz zu Festkommazahlen können Gleitkommazahlen sowohl sehr große als auch sehr kleine Werte beschreiben. Allerdings besteht hier das Problem, dass nicht alle Zahlen exakt dargestellt werden können. Dies hat weitreichende Folgen für die Definition qualitativer Geometrieeigenschaften. So kann es passieren, dass zwei Objekte, die sich bei exakter Positionierung berühren, sich durch Rundungsfehler bei der Zahlendarstellung plötzlich überlappen oder einen gewissen Abstand voneinander haben. Ein winziger quantitativer Fehler kann hier also einen unter Umständen entscheidenden qualitativen Unterschied ausmachen.

Da die meisten Systeme zur 3D-Darstellung und auch moderne Grafikhardware intern mit Gleitkommazahlen arbeiten und die angesprochenen Probleme bezüglich der Genauigkeit durch Wahl von Zahlen mit doppelter Genauigkeit und Definition einer Epsilon-Umgebung entschärft werden kann, werden auch im virtuellen Labor Gleitkommazahlen verwendet.

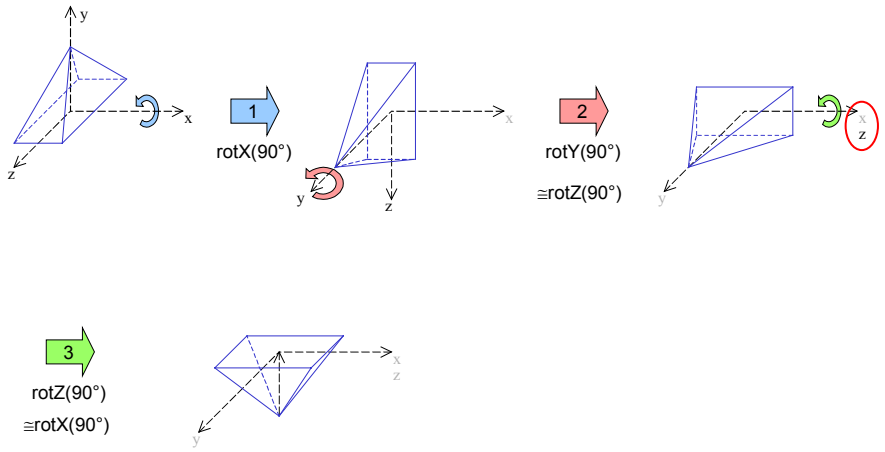
Rotationen

Während die Beschreibung der Position eines Objekts vergleichsweise einfach und naheliegend ist, stehen zur Angabe seiner Orientierung verschiedene Verfahren zur Auswahl, die anfangs alle geeignet erscheinen. Diese Verfahren sollen im Folgenden kurz vorgestellt werden. Für weitergehende Informationen wird auf entsprechende Quellen verwiesen.

Euler-Winkel

Eine Möglichkeit zur Angabe von Orientierungen besteht in der Spezifikation von Drehwinkeln um die drei Koordinatenachsen des kartesischen Koordinatensystems. Dieses Verfahren wird in der Literatur als Euler-Winkel (Euler-angle) bezeichnet. Der Name bezieht sich das eulersche Rotationstheorem, das beweist, dass jede Orientierung durch Angabe von drei Winkeln beschrieben werden kann. Während eine derartige Darstellung zunächst aufgrund ihrer leichten Verständlichkeit sehr vorteilhaft erscheint, hat sie jedoch den Nachteil, dass das Resultat abhängig von der Reihenfolge der Drehungen ist, da die Drehoperationen nicht kommutativ sind. Zwar kann die Darstellung durch Festlegung einer Reihenfolge eindeutig gemacht werden, ein weiteres Problem dieser Darstellungsform bleibt dabei jedoch bestehen. Als *gimbal lock* wird der Effekt bezeichnet, der entsteht wenn durch die aufeinanderfolgenden Drehungen zwei Achsen aufeinander zum Liegen kommen. Das Ergebnis ist eine unnatürliche Rotationsbewegung, die sich nicht verhält, wie vom Benutzer erwartet. Man spricht in diesem Zusammenhang auch von *Singularitäten*. Abbildung 16 verdeutlicht das Problem graphisch.

Abbildung 16 Gimbal Lock



Die erste Drehung um 90° um die X-Achse überführt die lokale Y-Achse auf die Z-Achse (1). Da durch die Verkettung der Operationen die Drehungen immer auf das lokale Koordinatensystem bezogen sind, wirkt sich folglich die 90°-Drehung um die Y-Achse (2) des Objekts wie eine Drehung um die ursprüngliche Z-Achse aus. Diese Drehung überführt nun die Z-Achse auf die ursprüngliche X-Achse, die schon ausgewertet wurde und sich daher nicht mehr mitdreht. Wird nun zum Schluss um die Z-Achse gedreht (3), wirkt sich dies wie ein Rotation um die ursprüngliche X-Achse aus, eine direkte Rotation um die Y-Achse ist daher so nicht möglich. Zwar hat im Endergebnis auch eine Rotation um die Y-Achse stattgefunden, diese wurde jedoch auf unnatürlichem Wege durch Kombination mehrerer Rotation erreicht.

Gimbal lock bedeutet also nicht, dass nicht jede mögliche Rotation durch Euler-Winkel dargestellt werden kann. Lediglich der Weg zur Erreichung der gewünschten Orientierung entspricht in vielen Fällen nicht dem vom Benutzer intuitiv erwarteten. Dadurch wird der prinzipielle Vorteil der einfachen Veranschaulichung der Euler-Winkel stark gemindert. Auch die Interpolation zwischen zwei Orientierungen lässt sich in Form von Euler-Winkeln nur sehr schwer beschreiben. Sie folgt nicht dem einem direkten Bogen zwischen den beiden Richtungen, sondern vollführt wilde Rotationen um die drei Achsen und ist daher für den Einsatz in vielen Anwendungsfällen die Animationen erfordern nicht geeignet.

Um die Probleme der Darstellung von Orientierungen durch Euler-Winkel zu vermeiden, bieten sich andere Mechanismen an, die eine beliebige Rotation in einem einzigen Schritt ausführen. Dadurch wird sowohl das Problem des gimbal lock als auch die Reihenfolgeabhängigkeit der Teilrotationen vermieden. Gängige Verfahren sind die Angabe von Achse und Winkel (Axis-angle), Quaternionen und Rotationsmatrizen.

Rotationsmatrizen

Eine Alternative zur Darstellung beliebiger Rotationen und sonstiger Transformationen (Skalierung, Scherung) stellen Matrizen dar. An dieser Stelle sollen nur einige wichtige Aspekte von Matrizen zur Beschreibung von Rotationen aufgeführt werden. Eine hervorragende Übersicht über die wesentlichen Aspekte von Matrizen zur Rotationsdarstellung bietet [MQFAQ].

Eine Transformationsmatrix ist eine 3x3-Matrix der Form

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

Zur Anwendung der in einer Matrix beschriebenen Transformation auf einen Punkt führt man eine Multiplikation der in Form eines Spaltenvektors beschriebenen Koordinaten des Punktes mit der Matrix durch:

$$P' = A \cdot P = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Eine Matrix, die eine Rotation um die Z-Achse des kartesischen Koordinatensystems um den Winkel ϕ bewirkt, hat die Form

$$rot_Z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Analog können Matrizen für die Rotation um X- oder Y-Achse definiert werden. Aber auch Rotationen um beliebige Achsen können durch Matrizen dargestellt werden.

Die drei Spalten einer 3x3-Matrix beschreiben die Basisvektoren des durch die Matrix rotierten Koordinatensystems. Handelt es sich um Einheitsvektoren (Länge = 1) und sind sie orthogonal zueinander, bezeichnet man die Matrix als *orthonormal*. Hat eine Matrix diese Eigenschaft, lassen sich vereinfachende Annahmen bei weiteren Operationen machen. Für die Verwendung als Rotationsmatrix wird daher im Allgemeinen die Eigenschaft der Orthonormalität vorausgesetzt.

Die Multiplikation von Matrizen bewirkt eine Verkettung der Rotationen. Da die Matrixmultiplikation nicht kommutativ ist, ist dabei die Reihenfolge zu beachten.

3x3-Rotationsmatrizen können durch Erweiterung auf 4x4-Matrizen auch zusätzlich zur Beschreibung von Translationen genutzt werden. So können sämtliche Transformationsoperationen durch lediglich eine mathematische Operation, die Matrixmultiplikation, dargestellt werden. Dadurch eignen sie sich hervorragend zur internen Darstellung beliebiger Transformationen und deren Verkettung. Sie werden daher häufig als Grundlage für Geometriepipelines in Hard- und Software für 3D-Graphiksystemen eingesetzt.

Eine allgemeine Transformationsmatrix ist eine 4x4 Matrix der Form

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Die obere linke 3x3-Matrix entspricht dabei der eingangs beschriebenen 3x3-Rotationsmatrix. Die Elemente a_{30}, a_{31} und a_{32} werden bei der Erweiterung auf 0 gesetzt, das Element a_{33} auf 1. Die Elemente a_{03}, a_{13} und a_{23} entsprechen den Elementen x_1, x_2 und x_3 eines Translationsvektors. Um Punkte im dreidimensionalen Raum mit einer 4x4-Matrix multiplizieren und so transformieren zu können, werden sie homogenisiert. Dabei erhalten sie eine vierte Koordinate w, der immer der Wert 1 zugewiesen wird.

Während sich Matrizen wegen ihrer Vereinheitlichung aller Arten von Transformationen sehr gut für die interne Nutzung auch im virtuellen Labor eignen, führen sie jedoch zur reinen Darstellung von Rotationen unnötige Informationen mit. Schwerer wiegt jedoch ihre wenig intuitive Darstellung von Orientierungen. Zudem leiden Matrizen unter dem bei der Beschreibung der Position erwähnten Problem der mangelnden Genauigkeit der Gleitkommazahldarstellung in Rechnern. Die Verkettung von zahlreichen Rotationen durch Multiplikation der Matrizen führt zur Kumulierung von Fehlern, sodass die Matrizen die erwünschte Orthonormalität verlieren. Zwar kann diese Eigenschaft wiederhergestellt werden, die so entstehende Matrix beschreibt jedoch die gewünschte Rotation nicht notwendigerweise am präzisesten. Für die Animation eigenen sich Matrizen nur bedingt, da zur Interpolation zwischen zwei Orientierungen eine Konversion in eine andere Darstellungsform erfolgen muss.

Achse und Winkel

Jede Rotation kann durch Angabe einer Drehachse im Raum und einen Rotationswinkel α dargestellt werden. Die Drehachse wird durch einen normierten Vektor n definierten. Achse und Winkel können daher als ein 4-Tupel der Form

$$\mathrm{rot}_{AW} = (n_x, n_y, n_z, \alpha)$$

geschrieben werden.

Das „Achse und Winkel“ -Verfahren behebt einige der Nachteile der Darstellung von Orientierungen mit Euler-Winkeln und ist überdies eine für den Menschen leicht nachvollziehbare Beschreibung einer Rotation.

Zugleich hat es jedoch einige Nachteile, die seine Verwendung als systeminternes Darstellungsformat für Rotationen einschränken. So gibt es keine einfache Methode, mehrere Rotationen in „Achse und Winkel“-Darstellung miteinander zu verknüpfen. Dazu ist zunächst eine Konversion zum Beispiel in eine Matrixdarstellung erforderlich. Auch die Interpolation zwischen zwei so beschriebenen Orientierungen, also die Ermittlung von Drehachse und Winkel, die eine Orientierung in die andere überführt, ist aufwendig.

Quaternionen

Konzeptionell ähnlich zur Darstellung einer Rotation durch Angabe von Achse und Winkel ist die Verwendung von Quaternionen. Dabei handelt es sich um eine Erweiterung von komplexen Zahlen, die statt über eine über drei komplexe Dimension verfügen. Statt nur einer Zahl i gibt es also drei Zahlen i, j und k, deren Quadrat jeweils –1 ergibt. Multipliziert man zwei dieser Zahlen miteinander, verhalten sie sich ähnlich wie das Kreuzprodukt der Einheitsvektoren eines euklidischen Koordinatensystems:

$$\mathbf{i} \cdot \mathbf{j} = -\mathbf{j} \cdot \mathbf{i} = \mathbf{k}$$

$$\mathbf{j} \cdot \mathbf{k} = -\mathbf{k} \cdot \mathbf{j} = \mathbf{i}$$

$$\mathbf{k} \cdot \mathbf{i} = -\mathbf{k} \cdot \mathbf{i} = \mathbf{j}$$

Ein Quaternion kann auf verschiedene Arten dargestellt werden.

- als Linearkombination von 1, i, j und k:

$$q = w \cdot 1 + x \cdot \mathbf{i} + y \cdot \mathbf{j} + z \cdot \mathbf{k}$$

- als Vektor der vier Koeffizienten dieser Linearkombination:

$$q = (w, x, y, z)$$

- als ein Skalar für den Koeffizienten von 1 und einen Vektor der Koeffizienten der imaginären Teilprodukte:

$$q = (s, \mathbf{v}) \text{ mit } s = w \text{ und } \mathbf{v} = (x, y, z)$$

Das Produkt zweier Quaternionen q_1 und q_2 ergibt sich aus

$$q_1 = (s_1, \mathbf{v}_1) = s_1 + x_1 \cdot \mathbf{i} + y_1 \cdot \mathbf{j} + z_1 \cdot \mathbf{k} \text{ und}$$

$$q_2 = (s_2, \mathbf{v}_2) = s_2 + x_2 \cdot \mathbf{i} + y_2 \cdot \mathbf{j} + z_2 \cdot \mathbf{k}$$

wie folgt:

$$q_1 \cdot q_2 = (s_1 + x_1 \cdot \mathbf{i} + y_1 \cdot \mathbf{j} + z_1 \cdot \mathbf{k}) \cdot (s_2 + x_2 \cdot \mathbf{i} + y_2 \cdot \mathbf{j} + z_2 \cdot \mathbf{k}) =$$

$$s_1 \cdot s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2 + s_1 \cdot \mathbf{v}_2 + s_2 \cdot \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2$$

Die Multiplikation von Quaternionen ist assoziativ, aber nicht kommutativ.

Die Definition des konjugierten Quaternions q' eines Quaternions q erfolgt analog zu der Definition einer konjugierten komplexen Zahl:

$$q' = w - x\mathbf{i} - y\mathbf{j} - z\mathbf{k}$$

Der Betrag eines Quaternions ist definiert als

$$|| q ||= \sqrt{q \cdot q'} = \sqrt{w^2 + x^2 + y^2 + z^2}$$

Ein Einheitsquaternion ist ein Quaternion mit ||q|| = 1.

Das Inverse q^{-1}eines Quaternions q ist definiert als

$$q^{-1} = \frac{q'}{q \cdot q'}$$

Ist q ein Einheitsquaternion, so ist sein Inverses gleich seinem konjugierten Quaterion:

$$q^{-1} = q'$$

Eine Erläuterung des vollständigen mathematischen Hintergrunds von Quaternionen geht über den Rahmen dieser Arbeit hinaus, daher sei für weitere Erläuterungen auf [MQFAQ] verwiesen. [Do2002] bietet eine umfassende Übersicht der für die Rotationsdarstellung relevanten Eigenschaften von Quaternionen und beschreibt auch die Konversion zwischen Quaternion und Rotationsmatrix.

Zur Repräsentation einer Drehung um einen Einheitsvektor **u** mit dem Winkel ϕ dient das Quaternion q = (s, **v**) mit:

$$s = \cos\frac{\phi}{2}$$

$$v = u \cdot \sin\frac{\phi}{2}$$

Die Darstellung ist also konzeptionell sehr ähnlich zum „Achse und Winkel“-Verfahren, hat jedoch andere mathematische Eigenschaften:

Um einen Punkt p zu rotieren, stellt man ihn durch das Quaternion P = (0, **p**) dar und berechnt den rotierten Punkt p' wie folgt:

$$p' = q \cdot P \cdot q^{-1}$$

Durch die mathematischen Eigenschaften von Quaternionen lässt sich die Verkettung von zwei durch die Einheitsquaternionen q_1 und q_2 dargestellten Rotationen sehr einfach wie folgt darstellen:

$$p' = q_2 \cdot q_1 \cdot P \cdot q_1^{-1} \cdot q_2^{-1}$$

Neben dem Vorteil der einfachen Verkettung von Rotationen mit Quaternionen sowie der Interpolation zwischen durch Quaternionen dargestellten Orientierungen sparen Quaternionen zusätzlich zahlreiche Operationen im Vergleich zur Matrixschreibweise ein. Erst wenn die Transformation eines Objekts berechnet werden soll, muss aus dem Quaternion eine Matrixdarstellung abgeleitet werden.

Zudem sind Quaternionen stabiler bezüglich Abweichungen durch die mangelnde Rechengenauigkeit von Gleitkommazahlen im Rechner. Eine Matrix, die durch die kumulierten Fehler von der Orthonormalität wegdriftet, kann zwar wieder orthonormal gemacht werden, stellt aber danach nicht mehr notwendigerweise die gewünschte Rotation dar. Zur Rotation verwendete Quaternionen verlieren dagegen zwar ihre Eigenschaft als Einheitsquaternion, diese kann jedoch durch Normieren wiederhergestellt werden, wobei das resultierende Einheitsquaternion automatisch das Optimale ist.

Fazit: Darstellung von Rotationen

Die vorgestellten Verfahren zur Beschreibung von Rotationen haben jeweils spezifische Vor- und Nachteile. Während Euler-Winkel sowie das „Achse- und Winkel"-Verfahren leicht verständliche Beschreibungen einer Rotation liefern, sind Matrizen und Quaternionen wenig intuitiv, jedoch für die interne Verarbeitung besser geeignet. So wird man zur Speicherung und Verarbeitung von reinen Rotationen Quaternionen verwenden, und für komplexere Transformationen, die auch Translation und Skalierung beinhalten, Matrizen einsetzen. Da im virtuellen Labor jedoch meist lediglich Starrkörperbewegungen erforderlich sind, bei denen keine Skalierung oder Scherung erfolgt, bietet sich als effizienteste Form die Beschreibung dieser Operation durch Angabe eines Translationsvektors und eines Quaternions für die Rotationskomponente an.

4.3.3.2 Constructive Solid Geometry

Die Grundidee des *Constructive Solid Geometry* (CSG)-Verfahrens basiert auf der Darstellung von komplexen Körpern (*Solids*) durch Komposition aus Grundkörpern, den sogenannten *Primitives*. Diese können als abgeschlossene Menge von Punkten im Raum aufgefasst werden. Jedes System stellt eine gewisse Menge dieser Primitives als Ausgangsbasis zur Verfügung. Üblicherweise sind dies Quader, Kugel, Zylinder, Prisma, Tetraeder, Pyramide und Kegel. Sie können bei Instanzierung skaliert werden, sodass beispielsweise mit einem Quader in Einheitsgröße beliebige quaderförmige Objekte dargestellt werden können. [Ho89] beschreibt die wesentlichen Aspekte von CSG-Verfahren.

Die Auswahl an zur Verfügung stehenden Primitives hat direkten Einfluss auf die mögliche Komplexität der Geometrien, die später beschrieben werden können. Viele CSG-basierte Ansätze beschränken sich auf Polyeder, also durch gerade Flächen darstellbare Körper. Kugeln, Zylinder und andere gekrümmte Flächen sind damit jedoch nicht exakt beschreibbar und können lediglich angenähert werden. Allerdings erreicht man durch diese Einschränkung eine erhebliche Reduktion der Komplexität der Darstellung. Wegen Anforderung P2.2.1.1.1 ist eine Approximation für das virtuelle Labor jedoch nicht ausreichend, sodass auch Primitive mit gekrümmten Flächen zu unterstützen sind.

Um komplexere Körper zu erzeugen, können die CSG-Solids mit Hilfe von geschlossenen binären Mengenoperationen wie Schnitt, Vereinigung und Differenz kombiniert werden, deren Ergebnis wieder Solids sind. Zudem ist als unäre Operation eine Verschiebung und Rotation (Starrkörperbewegung, *rigid motion*) der Solids möglich. Durch die wiederholte Anwendung von Starrkörperbewegungen und den Mengenoperationen können so komplexe Körper definiert werden. Die Operationen bilden dabei einen Baum, an dessen Wurzel der

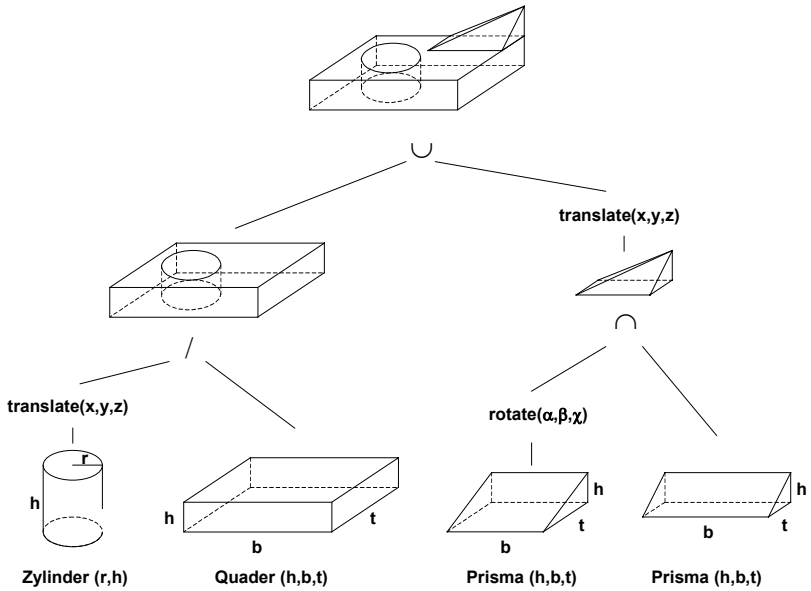
letztlich darzustellenden Körper steht. Die Primitives bilden die Blätter und die Operationen die inneren Knoten des Baumes. Dieser sogenannte CSG-Baum (CSG-tree) ist die natürliche Darstellungsart für CSG-Solids und beschreibt auf eindeutige Weise einen Solid. Umgekehrt gilt jedoch, dass ein Solid im Allgemeinen durch verschiedene CSG-Bäume dargestellt werden kann. Eine gängige formelle Definition für einen CSG-Baum nach [Rott2001] lautet:

Definition 1

```
<csg-tree> :=  <primitive> |
               <csg-tree> <binary set operation> <csg-tree> |
               <csg-tree> <rigid motion>
```

Dabei steht <primitive> für die Instanz einer der Grundformen, <binary set operation> steht für Vereinigung, Schnitt oder Differenz und <rigid motion> (Starrkörperbewegung) repräsentiert jede Form von Translation und Rotation des Körpers. Die Darstellung der Rotation kann dabei in einem der zuvor vorgestellten Verfahren erfolgen. Abbildung 17 zeigt beispielhaft die Konstruktion eines komplexen Solids im CSG-Verfahren.

Abbildung 17 Konstruktion eines Solids im CSG-Verfahren



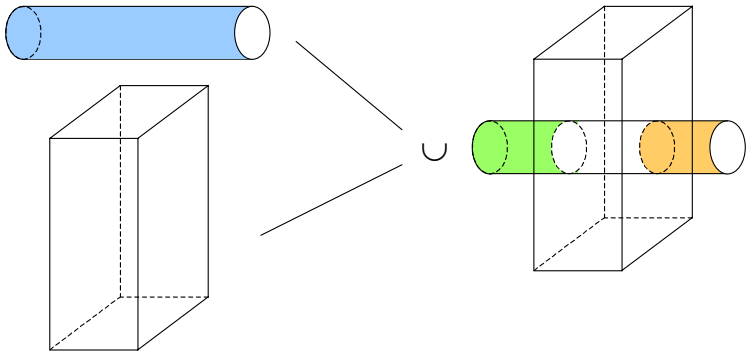
Die Darstellung von Geometrien im CSG-Verfahren eignet sich sehr gut für bestimmte Anforderungen. So ist zum Beispiel die Frage, ob ein beliebiger Punkt innerhalb, außerhalb oder auf dem Körper liegt relativ leicht zu beantworten, indem die Anfrage ausgehend von der Wurzel des CSG-Baums rekursiv an alle Knoten des Baumes weitergereicht wird. Die Ergebnisse der Auswertung auf Kindknoten werden je nach Operation im jeweiligen inneren Knoten aggregiert. Weiterhin sind alle CSG-Operationen geschlossen, was bedeutet, dass die Anwendung einer CSG-Operation auf einen oder zwei existierende CSG-Solids wieder einen gültigen (wenn auch möglicherweise leeren) CSG-Körper ergibt. Dies vereinfacht die Konsistenzprüfung enorm.

Problematisch ist CSG jedoch für die Visualisierung der Körper in einer Anwendung. Dazu ist im Allgemeinen eine Konversion in eine Grenzflächendarstellung erforderlich, ein Verfahren, das im Anschluss erläutert wird.

Neben diesen allgemeingültigen Vor- und Nachteilen des CSG-Verfahrens hat es auch spezifische Stärken und Schwächen für die Verwendung zur Darstellung der Bauteilegeometrie im virtuellen Labor.

Anforderung P2.2.1.1.2 beschreibt die Konstruktion der Bauteile aus Grundformen analog zur Vorgehensweise des CSG-Verfahrens. Dementsprechend könnte diese Anforderung mit einer CSG-Darstellung hervorragend abgedeckt werden. Schwierigkeiten bereitet jedoch Anforderung P2.2.1.2, die eine Zuweisung von Randbedingungen zu Elementen der Geometrie mindestens auf Ebene von einzelnen Flächen der Körper verlangt. Eine solche Zuordnung ist problematisch, da Flächen in CSG nur implizit beschrieben sind. Während die Vergabe einer eindeutigen Bezeichnung der Flächen von Primitives möglich und auf dieser Basis eine Zuordnung von Randbedingungen denkbar ist, versagt dieser Ansatz jedoch bei komplexeren Körpern. Abbildung 18 soll die Problematik verdeutlichen helfen.

Abbildung 18 Problematik: Adressierung von Flächen

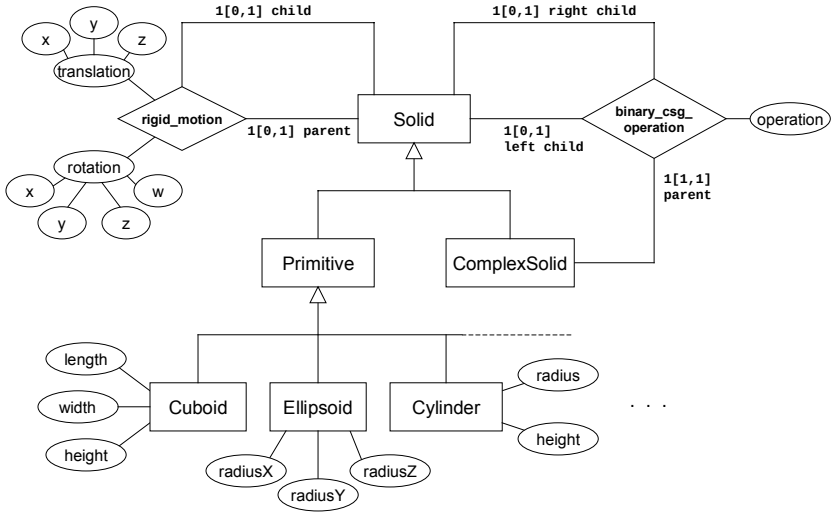


Wie leicht zu erkennen ist, entstehen aus einer einzigen Fläche eines Primitives (Seitenwand des Zylinders) im Zuge der Vereinigungsoperation zwei getrennte Flächen, denen unter Umständen verschiedene Randbedingungen zuzuordnen sind. Da für die so entstandenen Flächen keine direkte Repräsentation im CSG-Baum existiert und sie nicht mehr eindeutig der Fläche eines Primitives entsprechen, ist eine Zuordnung schwierig. Eine Möglichkeit wäre die Durchnummerierung aller Flächen des bei einer CSG-Operation entstehenden Körpers, auf die dann bei der Beschreibung von Randbedingungen Bezug genommen werden kann. Eine solch pragmatische Lösung bringt jedoch eine Reihe von Problemen mit sich. So kommt es zu einer Verlagerung von strukturellen Informationen aus den Daten in die Anwendung. Mag dies zunächst als rein akademisches Problem erscheinen, werden die Nachteile offensichtlich, wenn man von einer möglicherweise heterogenen Umgebung ausgeht, in der verschiedene Anwendungen die Daten nutzen. Hier müsste das Nummerierungsschema deterministisch und einheitlich in allen Anwendungen sein, die auf die Daten zugreifen. Auch die Eigenschaften einer solchen Darstellungsweise im Bezug auf Lösch- und Änderungsoperationen erweisen sich als ungünstig: Änderungen in einem CSG-Teilbaum wirken sich ebenfalls auf alle Nummerierungen oberhalb dieses Teilbaums aus. Die Nummerierung

auf einer Ebene könnte durch das Hinzukommen oder den Wegfall von Teilbäumen unterhalb dieser Ebene verändert werden, sodass hier ein erheblicher Wartungsaufwand entstünde. Zudem können durch eine derartige Verlagerung die Fähigkeiten der Datenhaltungsschicht für die Realisierung von Suchfunktionen nicht genutzt werden.

Trotz dieser Einschränkungen ist CSG ein Verfahren, das für die Modellierung der Bausteingeometrie in Frage kommt. Ein einfaches relationales Schema zur Darstellung von CSG-Bäumen beschreibt [HHM88]. Eine für die Anforderungen des virtuellen Labors erweiterte Darstellung zeigt die Abbildung 19.

Abbildung 19 Erweitertes ERM für CSG-Bäume



Komplexe Solids und Primitive sind Subklassen von Solid. Als Operanden für die binären CSG-Operationen können sowohl Primitive als auch komplexe Solids dienen, das Ergebnis einer Operation ist ein komplexer Solid. Die Klasse Primitive hat eine Menge von Subklassen, die je nach den Erfordernissen erweitert werden kann. Jede dieser Subklassen hat eine Menge von Attributen, die bei der Instanzierung gesetzt werden müssen.

Die unäre Operation „Starrkörperbewegung“ (rigid motion) kann auf alle Solids angewendet werden, und führt im Ergebnis wiederum zu einem Solid. Sie ist durch die Translation auf allen drei Koordinatenachsen und durch die Rotation hier in Quaterion-Darstellung beschrieben.

Nachteil dieses Datenmodells ist sein rekursiver Aufbau, was bei der Verwendung von Datenmodellen mit fehlender Unterstützung von Rekursion zu Problemen führen kann. So war Rekursion in früheren SQL-Dialekten nicht vorgesehen. Daher ist zum Einlesen eines Körpers in CSG-Darstellung eine vorher nicht bestimmbare Anzahl von Anfragen durchzuführen, um den Baum in allen seinen Stufen einzulesen. SQL:1999 unterstützt zwar Rekursion, sie wird jedoch nicht in allen DBMS implementiert und unterscheidet sich von System zu System in ihrem Funktionsumfang [Tü2003].

4.3.3.3 Boundary Representation

Eine weitere gängige Klasse von Verfahren zur Darstellung von Solids wird als *Boundary Representation* (kurz: *BRep*) bezeichnet. Sie beschreiben die Körper durch Angabe seiner Grenzflächen (*boundary*). Bei der Speicherung unterscheidet man geometrische und topologische Informationen. Die Topologie beschreibt den Aufbau eines Körpers aus einfacheren Elementen durch Beschreibung der Struktur zwischen diesen Elementen:

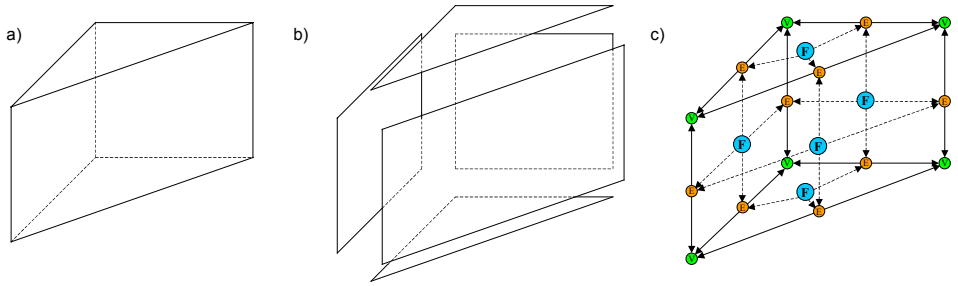
Je nach Komplexität der darzustellende Körper haben sie eine oder mehrere Hüllen (*shell*), die jeweils wiederum aus Flächen (*face*) bestehen. Um eine Unterscheidung zwischen dem Inneren und der Umgebung des Körpers zu ermöglichen, haben Flächen eine Oberflächennormale, die gemäß Konvention nach außen zeigt.

Flächen sind durch gerichtete Kanten (*edge*) begrenzt. Diese bilden mindestens einen Zyklus (loop), der dabei die äußere Grenze (*outer loop*) der Fläche bildet. Weitere Zyklen umschließen Löcher innerhalb der Fläche und werden daher *inner loop* genannt.

Um das Innere einer Fläche eindeutig von der Umgebung zu unterscheiden, sind die Kanten des outer loop von außerhalb des Körpers betrachtet im Uhrzeigersinn geordnet. Kanten von inner loops sind entgegen dem Uhrzeigersinn geordnet. Fehlt diese Ordnung, ist eine Ermittlung des Flächeninnern nur auf Grund der geometrischen Informationen und Annahmen bezüglich der Form der Fläche möglich. (Vergleiche hierzu auch Abbildung 24).

Jede Kante hat genau zwei benachbarte Flächen. Kanten werden von Endpunkten (*vertex*) begrenzt, in jedem Endpunkt schneiden sich alle benachbarten Kanten. Diese Topologie lässt sich in einem Graphen darstellen, bei dem Eckpunkte, Kanten und Flächen durch Knoten repräsentiert sind, die entsprechend ihrer Nachbarschaftsbeziehung verbunden sind. Abbildung 20 zeigt die Topologie eines Prismas unter Verzicht auf die Darstellung von Loops. Den Knoten der Topologie kann dann die geometrische Information zugeordnet werden. Im allgemeinen sind das die Koordinaten der Eckpunkte und je nach erforderlicher Darstellungsgenauigkeit auch die Ebenengleichungen für alle Flächen. Diese ist dabei so zu schreiben, dass die Normale der Fläche nach außen zeigt. Beschränkt man sich auf ebene Flächen, genügt die Angabe des Normalenvektors. Optional können auch die Kurvengleichungen für die Kanten angegeben werden.

Abbildung 20 Topologie eines Prismas in BRep-Darstellung



a) Körper b) Grenzflächen des Körpers c) Topologie des Körpers

Topologische Gültigkeit

Die bisher beschriebenen Einschränkungen bezüglich der Topologie erzwingen nicht notwendigerweise einen gültigen Körper. Durch Beschränkung auf sogenannte geschlossene, orientierbare 2-Manifold-Körper kann die topologische Gültigkeit erzwungen werden. Zugleich sind die mathematischen Eigenschaften dieser Körper gut erforscht.

Geschlossen bedeutet, dass der Körper durch seine Oberfläche vollständig umhüllt ist. Eine Oberfläche gilt als orientierbar, wenn man zwei verschiedene Seiten unterscheiden kann. Man wählt einen beliebigen Punkt P auf der Oberfläche und definiert willkürliche eine Rotation um diesen Punkt als „im Uhrzeigersinn erfolgend“. Unter Beibehaltung dieser Rotationsrichtung bewegt man sich nun auf beliebigen Pfaden auf der Oberfläche. Existiert ein Pfad, sodass man an den Punkt P zurückkehren kann und sich die Rotationsrichtung umgekehrt hat, ist die Oberfläche nicht orientierbar. Das bekannteste Beispiel für eine nichtorientierbare Oberfläche ist der Möbius-Streifen [MW1], wie er beispielsweise von M.C. Escher [MCE] dargestellt wurde, oder die Kleinsche Flasche [MW2].

Eine Oberfläche ist ein Manifold, wenn gilt, dass Kanten von genau zwei Eckpunkten begrenzt werden, jede Kante genau zwei benachbarte Flächen hat, deren Zyklen entgegengesetzt orientiert sind, und an jedem Eckpunkt genau ein „Kreis“ von Kanten und Flächen existiert. Eine andere Definition fordert, dass für jeden Punkt P auf der Oberfläche eines 2-Manifolds eine Umgebung mit finitesimal kleinem Radius exisitiert, die homeomorphisch zu einer Ebene ist, das heißt, sie kann durch Deformation in eine Ebene überführt werden, ohne dass sie dabei „zerrissen“ wird oder einer oder mehr Punkte der Fläche aufeinander zu liegen kommen. Dadurch werden alle sich selbst schneidenden oder berührenden Oberflächen ausgeschlossen. Weitere Informationen zu Darstellungen in Boundary Representation finden sich in [Ho89].

Die topologische Gültigkeit eines geschlossenen, orientierbaren 2-Manifold-Körpers lässt sich mit Hilfe der Euler- beziehungsweise Euler-Poincaré-Gleichung überprüfen.

So gilt für einen Körper ohne Löcher und Hohlräume im Innern, mit V Eckpunkten, E Kanten und F Flächen:

$$V - E + F - 2 = 0$$

Definiert man den Genus G eines Körpers als die Anzahl seiner „Löcher“, erlaubt jedoch nicht, dass einzelne Flächen Löcher haben dürfen, so gilt:

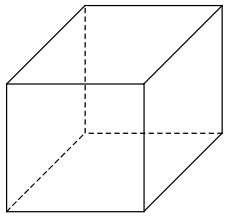
$$V - E + F - 2 (1 - G) = 0$$

Erlaubt man zusätzlich noch Hohlräume im Innern des Körpers und gestattet Löcher in Oberflächen, also Flächen, die von mehr als einem Kantenzyklus begrenzt sind, so wird der Körper von mehr als einer geschlossenen, orientierten 2-Manifold-Oberfläche oder „Hülle“ begrenzt. Sei S die Anzahl der Hüllen, L die Gesamtzahl aller Kanten-Zyklen, so gilt:

$$V - E + F - (L - F) - 2 (S - G) = 0$$

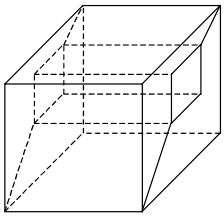
Abbildung 21 zeigt anhand einiger geschlossener, orientierbarer 2-Manifold-Körper Beispiele für die Euler- und Euler-Poincaré-Gleichungen.

Abbildung 21 Geschlossene, orientierte 2-Manifolds, Euler- und Euler-Poincaré-Gleichungen



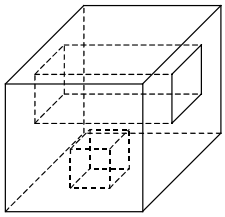
a) Quader
8 Eckpunkte
12 Kanten
6 Flächen

8 - 12 + 6 -2 = 0 ✓



b) Quader mit Loch
16 Eckpunkte
28 Kanten
12 Flächen
1 „Loch“
keine ringförmigen Flächen

16 - 28 + 12 - 2(1 - 1) = 0 ✓



c) Quader mit Loch und Hohlraum
24 Eckpunkte
36 Kanten
16 Flächen
18 Zyklen
1 „Loch“
2 Hüllen
ringförmige Flächen erlaubt

24 - 36 + 16 - (18 - 16) - 2(2 - 1) = 0 ✓

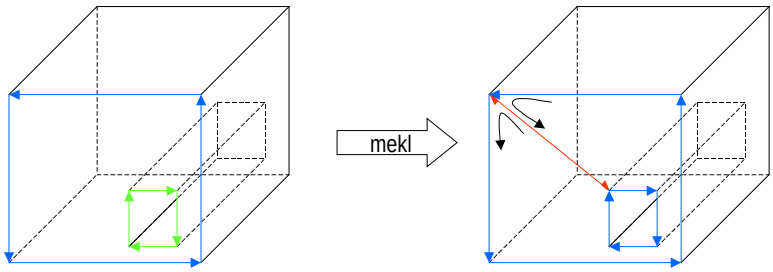
Euler-Operatoren

Operationen auf in BRep dargestellten Körpern sind je nach Komplexität der Darstellung sehr aufwendig, aber ausführlich erforscht. Einen Überblick über Algorithmen zur Implementierung der Operation liefert [Ho89].

Die sogenannten Euler-Operatoren verändern die Topologie eines Manifold-Solids, indem sie Oberflächen, Kanten, Eckpunkte, Zyklen und Löcher hinzufügen oder löschen. Es kann gezeigt werden, dass die Anwendung von Euler-Operatoren auf einen topologisch gültigen Manifold-Solid wieder einen topologisch gültigen Manifold-Solid ergibt. Allerdings kann die geometrische Integrität nicht garantiert werden, durch die Zuweisung von ungültigen geometrischen Informationen zu topologisch gültigen Solids können ungültige Körper entstehen.

Euler-Operatoren werden häufig in der Form mXkY geschrieben. ‚m‘ steht dabei für make, ‚k‘ für kill, X und Y stehen für eine beliebige Kombination von edge, vertex, loop oder face. Je nach Topologie kann zum Beispiel die Erzeugung einer Kante (make edge) zwischen existierenden Eckpunkten eine neue Fläche (make face) erzeugen (Operation: *mef*), oder eine bisher durch zwei Zyklen begrenzte Fläche in eine durch einen Zyklus begrenzte Fläche umwandeln (kill loop, Operation: *mekl*). Ein Beispiel für diese Euler-Operation zeigt Abbildung 22. Das Hinzufügen einer Kante bewirkt eine Durchtrennung der ringförmigen Fläche, die nun durch nur noch einen Loop begrenzt wird.

Abbildung 22 Euler-Operation „make edge, kill loop“ (mekl)



Darstellung gekrümmter Oberflächen

Je nach den Erfordernissen an die Art und Genauigkeit der Geometriedarstellung können neben geraden Kanten und ebenen Flächen auch gekrümmte Kanten und Flächen als Begrenzungen zugelassen werden. Diese kurze Betrachtung beschränkt sich auf die algebraischen Kurven und Flächen, welche die Beschreibung aller wesentlichen Formen für die Modellierung von Solids abdecken.

Algebraische Beschreibungen von Flächen werden im Allgemeinen in einer von zwei Formen dargestellt. Beide Formen haben spezifische Vor- und Nachteile.

Die implizite Darstellung einer Fläche besteht aus einer Gleichung, die für alle Punkte, die Teil der jeweiligen Fläche sind, erfüllt ist:

$$f(x,y,z) = 0$$

Die implizite Darstellung erlaubt einen sehr einfachen Test auf das Enthaltensein eines Punktes in der Fläche, macht jedoch das Finden von Punkten der Oberfläche schwierig.

Für einige, aber nicht für alle algebraischen Flächen existiert auch eine parametrische Darstellung. Sie besteht aus drei Gleichungen der Form:

$$x = h_1(s,t)$$

$$y = h_2(s,t)$$

$$z = h_3(s,t)$$

Für jedes Wertepaar s und t beschreiben diese Funktionen die Koordinate eines Punktes der Oberfläche im dreidimensionalen Raum.

Analog zu der Darstellung von Flächen existieren auch für Kurven zwei Darstellungsformen. Die implizite Darstellung beruht darauf, dass algebraische Kurven durch den Schnitt von zwei oder mehr Flächen definiert sind. Zur Modellierung von Festkörpern genügt jedoch im Allgemeinen die Berücksichtigung von Kurven, die durch den Schnitt zweier Flächen entstanden sind. Eine parametrische Darstellung existiert nicht für jede algebraische Kurve.

Gibt es eine solche Darstellung, wird sie ähnlich der parametrischen Darstellung von Flächen durch drei Funktionen angegeben:

$$x = s_1(t)$$

$$y = s_2(t)$$

$$z = s_3(t)$$

So erhält man für jeden Wert von t einen Punkt, der auf der durch die Funktionen s_i definierten Kurve liegen.

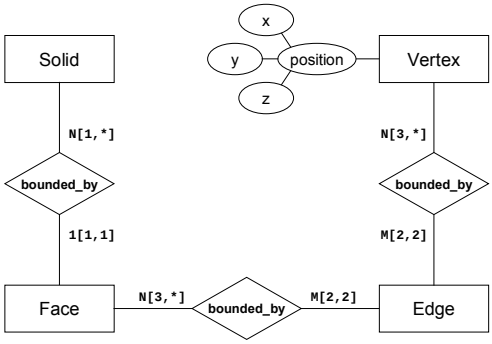
Interne Darstellung

Die systeminterne Darstellung von BRep-Körpern kann Eckpunkt-, Kanten- oder Flächenzentriert erfolgen. Im Folgenden sollen zwei mögliche Repräsentationsformen dargestellt werden. Sie unterscheiden sich in der Speicher-Effizienz und der Unterstützung verschiedener Anfragen.

Einfacher Ansatz

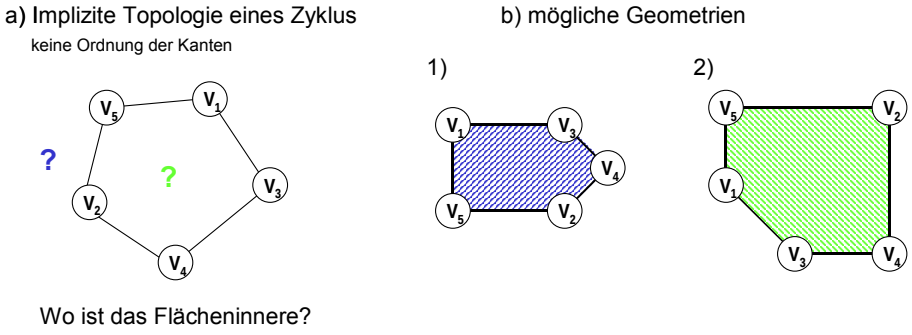
In der Literatur, die sich mit dem Thema der Speicherung von Geometriedaten in relationalen Datenbanken befasst, wird häufig ein Datenmodell vorgestellt, bei dem Flächen in einer n:m-Beziehung zu Kanten und diese wiederum in einer n:m-Beziehung zu Eckpunkten stehen. Für die Eckpunkte werden dann Koordinaten angegeben (Abbildung 23).

Abbildung 23 ERM für den einfachen Ansatz



Auch wenn damit eine Beschreibung möglich ist, fehlt es diesem Ansatz an Vollständigkeit und Eindeutigkeit. So ist die topologische Struktur eines in dieser Form beschriebenen Körpers nur unvollständig und implizit enthalten. Benachbarte Kanten lassen sich auffinden, indem man über ihre gemeinsamen Eckpunkte sucht. Dadurch ist es auch möglich, einen Zyklus von Kanten zu ermitteln, wenn ein solcher existiert. Da jedoch eine Orientierung dieses Zyklus fehlt, geht aus dieser Darstellung nicht explizit hervor, wo das Innere und das Äußere der Fläche liegen. Dies ergibt sich nur aus Anordnung der Kanten im Raum, unter Berücksichtigung der Eckpunktkoordinaten. Topologie und Geometrie sind also nicht vollständig und nicht unabhängig voneinander beschrieben. Abbildung 24 zeigt ein solches Problem für den einfachen Fall, dass nur Polyeder beschrieben werden.

Abbildung 24 Ermittlung der Topologie aus der Geometrie



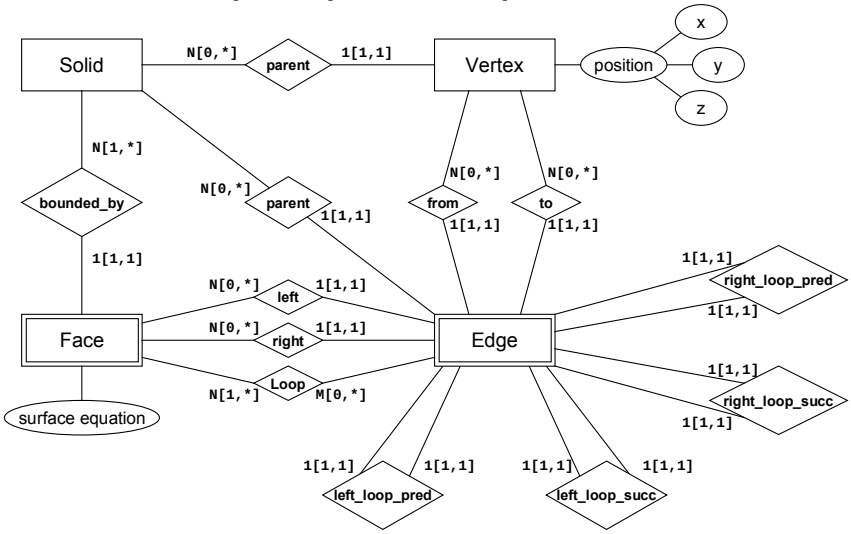
Hier ist die Frage nach der korrekten Darstellung noch eindeutig dadurch zu beantworten, dass man die Anordnung der Punkte in der Ebene untersucht, und so auf Flächeninneres und -äußeres schließen kann. Betrachtet man jedoch auch Körper mit gekrümmten Oberflächen und Kanten, entstehen leicht Fälle, in denen man keine eindeutige Darstellung aus den gegebenen Informationen zu ermitteln vermag.

Winged-Edge-Darstellung

Eine der ältesten Speicherungsformen für Polyeder in Boundary Representation ist die sogenannte Winged-Edge-Datenstruktur. Sie kann leicht auf Körper mit gekrümmten Oberflächen erweitert werden. Alle Informationen sind dabei in drei Listen mit den Eckpunkten, Kanten und Flächen des Körpers gespeichert, wobei die Kanten als zentrale Komponenten die wesentlichen Informationen tragen. Jede Kante hat eine Referenz auf ihre beiden Eckpunkte (Start und Ende) sowie auf die beiden Flächen, die sich in dieser Kante berühren. Diese werden als linke und rechte Fläche bezeichnet, definiert durch Betrachtung von außerhalb des Körpers unter Berücksichtigung der Richtung der Kante. Da die äußeren eine Fläche begrenzenden Zyklen wie eingangs erläutert immer im Uhrzeigersinn (von außerhalb des Körpers betrachtet) verlaufen, wird die Kante einmal vom Zyklus der linken und einmal vom Zyklus der rechten Kante traversiert. Eine der Traversierungen erfolgt dabei in Richtung der Kante, die andere entgegengesetzt dieser Richtung. Für beide Zyklen wird jeweils die vorherige und die nächste Kante der aktuellen Kante gespeichert. Als weitere Information existiert für jeden Eckpunkt ein Eintrag in der Vertex-Tabelle. Dort sind im Allgemeinen als geometrische Informationen die Koordinaten des Eckpunktes gespeichert, so wie eine Referenz auf eine beliebige der Kanten, die von diesem Eckpunkt begrenzt wird. Für jede Fläche existiert ein Eintrag in der Face-Tabelle. Hier ist neben einer eventuell vorhandenen algebraischen Darstellung der Oberfläche eine Referenz auf eine repräsentative Kante jedes diese Oberfläche begrenzenden Zyklus eingetragen.

Die Winged-Edge-Darstellung erlaubt eine sehr einfache Beantwortung von allen Nachbarschafts-Anfragen zwischen Eckpunkten, Kanten und Flächen. Die zu einer Kante benachbarten Eckpunkte und Flächen lassen sich direkt aus der Kanten-Tabelle entnehmen, ebenso die Kanten, die mit ihr einen Zyklus bilden. Andere Anfragen, wie die eine Fläche begrenzenden Kanten erfordern dagegen ein Navigieren durch die Datenstruktur. Eine Informationsmodell für die Winged-Edge-Datenstruktur beschreibt Abbildung 25:

Abbildung 25 ERM für die Winged-Edge-Darstellung



In dieser Darstellung entfällt die explizite Modellierung der „Referenzen“ von Eckpunkten und Flächen zu jeweils einer repräsentativen Kante, da Relationships im ERM symmetrisch sind. Dadurch können beispielsweise alle Kanten, die zu einem Eckpunkt in Kontakt stehen, leicht gefunden werden. Die Relationshipmenge Loop enthält für jeden Zyklus einer Fläche ein Element zwischen der Fläche und einer repräsentativen Kante des Zyklus.

4.3.3.4 Weitere Geometriedarstellungsmöglichkeiten

Neben den beiden zuvor vorgestellten populären Verfahren zur Geometriedarstellung existieren noch weitere Ansätze.

Räumliche Dekomposition

Eine Möglichkeit zur Beschreibung der Form eines Körpers ist die Unterteilung des Raums in ein gleichmäßigen Gitter von sogenannten Einheitswürfeln. In diesem Gitter markiert man alle Würfel, die das Innere des darzustellenden Körpers schneiden. Die Darstellung hat einige offensichtliche Nachteile. Zum einen ist die so erzielte Darstellung nur angenähert, je höher die geforderte Präzision ist, desto kleiner müssen die Einheitswürfel gewählt werden. Mit steigender Präzision nimmt zum anderen auch der Speicherplatzbedarf dieser Lösung zu, er wächst mit $O(n^3)$ wobei n die räumliche Auflösung in Richtung der Koordinatenachsen darstellt.

Eine Optimierung des Verfahrens stellen Octree-basierte Verfahren dar. Sie unterteilen den Raum in mehrere Gitter mit im Vergleich zum Vorgänger jeweils halbierter Gitterweite. Jeweils acht Einheitswürfel bilden einen Einheitswürfel im nächstgrößeren Gitter. Sind alle acht Würfel Teil des Körpers, werden sie zu einem nächst größeren Würfel aggregiert. Die Speicherung erfolgt in Form eines Octree, also eines Baumes, bei dem jeder Knoten maximal acht Nachfolger hat. Jede Ebene des Baumes repräsentiert ein Gitter, die Wurzel entspricht dem gröbsten, die unterste Ebene dem feinsten Gitter. Blätter beschreiben Würfel in der jeweiligen Größe ihrer Ebene, die nicht weiter aufgelöst werden müssen, weil sie entweder der

feinsten Auflösungsstufe entsprechen oder entweder alle oder keine ihrer Subwürfel Teil des Körpers sind. Innere Knoten repräsentieren dagegen Würfel, deren Subwürfel zum Teil innerhalb und zum Teil außerhalb des Körpers liegen. Dadurch können größere einheitliche Volumina effizienter beschrieben werden. Die prinzipiellen Nachteile der Dekompositionsverfahren bleiben jedoch erhalten. So lassen sich Operationen wie die Rotation nur mit gewaltigem Aufwand durchführen, und der unweigerliche Präzisionsverlust durch die Diskretisierung macht sie insbesondere für Strömungssimulationen – und damit für den Einsatz in einem virtuellen Labor für Verfahrenstechnik – unbrauchbar.

Für andere Domänen wie die Visualisierung in medizinischen Anwendungen oder in auf finiten Elementen basierenden Simulationssystemen haben Dekompositionsverfahren jedoch ihre Berechtigung.

Primitive Instancing

Während CSG-Verfahren mit einer begrenzten Anzahl von geometrischen Grundbausteinen oder Primitives arbeiten, die durch Operationen zu komplexeren Körpern kombiniert werden, wird beim *primitve instancing*-Verfahren der Begriff der Primitive weiter gefasst. Sie bieten eine größere Vielfalt von Formen, die durch Wahl von Parametern angepasst werden können. So kann ein Primitive „Quader mit zentralem zylinderförmigem Loch“ zum Beispiel durch die Parameter der Kantenlänge des Quaders sowie den Radius des Lochs beschrieben werden. Eine weitere Variation, wie sie zum Beispiel die Mengenoperationen des CSG-Verfahrens bietet, ist dagegen nicht möglich. Dadurch sind primitive instancing-Verfahren in ihrer möglichen Ausdruckskraft noch wesentlich stärker von den zur Verfügung stehenden Grundbausteinen abhängig, als dies bei CSG der Fall ist. Je nach Domäne können leicht hunderte Grundbausteine erforderlich sein, um die wesentlichen Anwendungsfälle abzudecken. Die Anforderungen des virtuellen Labors an die freie Definition der Bausteinformen auch durch parametrische Beschreibungen der Oberflächen könnte durch primitive instancing nicht mit vertretbarem Aufwand in der erforderlichen Flexibilität erreicht werden, sodass auch dieses Verfahren für das virtuelle Labor ausscheidet.

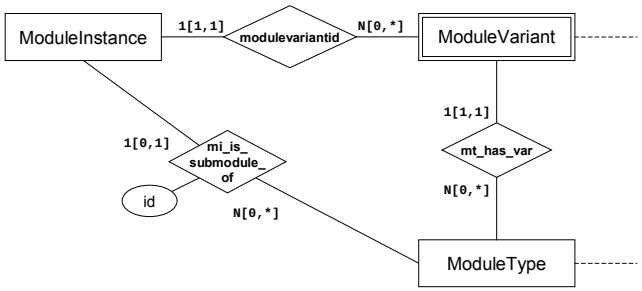
4.3.3.5 Wahl des Geometriedarstellungsverfahrens

Von den vorgestellten Geometriedarstellungsverfahren eignen sich prinzipiell sowohl CSG als auch Boundary Representation. Die in Anforderung P2.2.1.1.2 geschilderte CSG-artige Erstellung von Körpern drängt zwar eine dementsprechende Realisierung der Speicherung auf. Durch die explizite Repräsentation von Flächen im Schema und die für die graphische Darstellung erforderlichen Konversion von CSG- in BRep-Darstellung hat BRep jedoch entscheidende Vorteile für die interne Handhabung. So kann die Erstellung von Körpern im CSG-Verfahren, ihre Speicherung jedoch in BRep erfolgen. Alternativ kann ein hybrider Ansatz vorgesehen werden, bei dem die Entwurfshistorie eines Körpers aus dem CSG-Baum hervorgeht, die Zuordnung von Randbedingungen und die graphische Darstellung jedoch auf der BRep-Darstellung basiert.

4.3.4 Kompositionsmodell

Anforderung P2.2.1.4 beschreibt die Definition von komplexen Bausteintypen durch Kombination aus einfacheren Bausteintypen. Ein komplexer Bausteintyp verfügt wie ein einfacher Bausteintyp über Eigenschaften und möglicherweise eigene Geometrieelemente (Solids). Zusätzlich enthält seine Beschreibung Referenzen auf andere Bausteintypen die seine Subkomponenten bilden. Sie werden innerhalb des lokalen Koordinatensystems des komplexen Bausteins platziert und mit Werten für ihre Eigenschaften versehen. Die Komponenten weisen somit alle Eigenschaften einer Bausteininstanz auf. Folglich kann für ihre Modellierung auf die Bestandteile des Informationsmodells zur Bausteininstanzbeschreibung zurückgegriffen werden. Als Ergänzung ist lediglich eine weitere Relationshipmenge *mi_is_submodule_of* erforderlich, welche die Zugehörigkeit einer Instanz zu einem Bausteintyp als Subkomponente darstellt. Abbildung 26 zeigt die erforderliche Ergänzung.

Abbildung 26 Ergänzung des Informationsmodells zur Darstellung von komplexen Bausteinen



4.3.5 Funktionales Modell

Um ein Bauteil in einer Simulation einsetzen zu können, sind gemäß Anforderung P2.2.1.2 neben der bloßen Beschreibung seiner Form zusätzliche Merkmale erforderlich. Diese lassen sich unterteilen in statische und dynamische oder aktive Merkmale. Weiterhin können Bausteine als Quellen für Messergebnisse fungieren.

Statische Merkmale eines Experimentbausteins beeinflussen das Ergebnis einer Experimentdurchführung, bleiben aber während der gesamten Durchführung konstant und sind selbst nicht der Ursprung von Ereignissen oder Aktivitäten. Ein Experiment, dessen Bausteine nur statische Merkmale aufweisen, wird folglich seinen Zustand im Zeitverlauf nicht verändern. Beispiele für statische Merkmale sind insbesondere Material und Materialeigenschaften von Bausteinen sowie die Position von beim Aufbau beweglichen, bei der Durchführung aber fixierten Elementen des Bausteins. Die Geometrie des Bausteins kann also als ein spezielles statisches Bausteinmerkmal aufgefasst werden, dem lediglich wegen seiner zentralen Bedeutung als Bezugspunkt für alle anderen Merkmale eine besondere Stellung eingeräumt wurde.

Dynamische Merkmale sind Aktivitätsträger, die das vom Experimentaufbau beschriebene Gesamtsystem bei der späteren Durchführung anregen und so eine Veränderung seines Zustandes bewirken. Die dynamischen Merkmale werden im Folgenden als Aktoren bezeichnet. Aktoren haben Elemente der Geometrie zum Ziel, im Allgemeinen ganze Körper oder

einzelne Flächen. Die Liste der denkbaren dynamischen Merkmale in einem Experiment aus dem Bereich der Verfahrenstechnik umfasst unter anderem

- Ein- und Auslassöffnungen, die Substanzen in das System einbringen oder daraus entfernen.
- Rotierende Elemente wie Rührwerke oder Pumpen.
- Oszillierende Elemente wie Kolbenpumpen
- Heiz- und Kühlelemente, die eine andere als die Umgebungstemperatur haben.

Aktivitätsträger können wie Bausteintypen über variablen Eigenschaften verfügen, die ihr Verhalten beeinflussen. So kann für eine Einlassöffnung die Durchflussmenge und die einfließende Substanz definiert werden, rotierende Elemente erfordern eine Definition von Rotationsgeschwindigkeit und Drehachse etc.

Sollen diese Eigenschaften von Aktoren oder auch statische Merkmale des Bausteins vom Benutzer geändert werden können, müssen sie wie in Kapitel 4.3.10 bei der Vorstellung des Abstraktionsmodells geschildert in der Nutzersicht als Parameter zugänglich gemacht werden. So kann der Experimentator die Durchflussmenge oder die durch einen Einlass einfließende Substanz während des Experiments ändern und die Drehzahl, Oszillationsgeschwindigkeit oder Temperatur variieren. Da die Änderung der Parameter von Aktoren im Allgemeinen lediglich lokale Auswirkungen auf den Baustein hat, handelt es sich dabei um Betriebsparameter der Nutzersicht. Im Gegensatz dazu entsprechen statische Merkmale den Bausteinparametern, die nicht ohne weiteres im laufenden Betrieb einer realen wie virtuellen Apparatur geändert werden können.

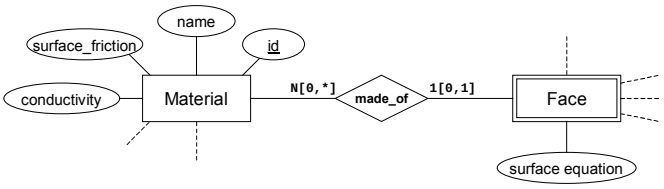
Im folgenden soll das in Kapitel 4.3.3 entwickelte Informationsmodell für die Geometriedarstellung zur Beschreibung statischer und dynamischer Merkmale erweitert werden.

4.3.5.1 Statische Merkmale (Characteristics)

Dem Material von Oberflächen kommt in der Simulation eine große Bedeutung zu. Seine Eigenschaften wie Reibungskoeffizient, Rauheit der Oberfläche oder die Wärmeleitfähigkeit können je nach dem verwendeten mathematischen Modell für die Durchführung des virtuellen Experiments entscheidenden Einfluss auf das Resultat haben. Zudem sind Farbe und Textur des Materials auch ein wichtiger Bestandteil bei der Visualisierung des Experimentaufbaus. Aufgrund dieser zentralen Rolle und der Tatsache, dass nahezu jeder Oberfläche ein Material zugeordnet wird, sollte ein eigener Entitytyp „Feststoff“ vorgesehen werden, der über die erforderlichen Attribute verfügt. Neben den zuvor genannten experimentrelevanten Attributen können hier auch eine Beschreibung, die chemische Zusammensetzung oder vergleichbare Informationen enthalten sein.

Die erforderlichen Informationen sind in Abbildung 27 als ERM dargestellt.

Abbildung 27 Informationsmodell für das statische Merkmal Material



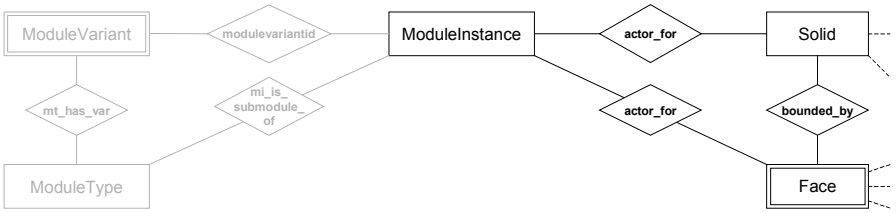
4.3.5.2 Dynamische Merkmale

Im Gegensatz zu den statischen Merkmalen lassen sich dynamische Merkmale oder Aktoren wegen ihrer Vielfalt und ihrer unterschiedlichen Parameter nicht ohne weiteres als einzelne Entities direkt im Informationsmodell repräsentieren.

Abbildung als Bausteintypen

Bei Betrachtung des Konzepts der Aktoren fällt auf, dass sie ähnlich wie Bausteintypen über eine Menge von Eigenschaften verschiedenen Typs verfügen. Daher ist es eine naheliegende Möglichkeit, Aktivitätsträger auf diese Strukturen des Informationsmodells abzubilden. Statt durch Angabe von Geometrie und Merkmalen frei definierbar zu sein wie Bausteintypen, werden die Aktoren direkt auf im System vordefinierte Objekte abgebildet. Zur Unterscheidung zwischen echten Bausteintypen und Aktoren kann ein Flag oder das Fehlen von Geometrieinformationen und Subkomponenten dienen. Um Aktoren den Geometriebestandteilen eines Bausteintypen zuzuweisen, sind zwei Relationshipmengen zwischen Solid und Bausteininstanz sowie zwischen Face und Bausteininstanz vorzusehen. Abbildung 28 zeigt das so entstehende Teilschema.

Abbildung 28 Aktoren als spezielle Bausteintypen



Nachteil einer derartigen Lösung ist die Vermischung von zwei Entitytypen, die zwar über eine Schnittmenge gemeinsamer Eigenschaften verfügen, jedoch völlig unterschiedliche Konzepte repräsentieren.

Aktortyp – Aktorparameter

Um diese Vermischung zu vermeiden, könnten jene Strukturen, die Aktoren und Bausteintypen gemeinsam haben, für Aktoren im Informationsmodell repliziert werden. In Analogie zu Bausteintypen und -eigenschaften gibt es Aktor(-typ) und Aktorparameter. Parameter sind wie Bausteineigenschaften getypt und ihnen wird ein Wert zugewiesen. Diese Form der Modellierung von Aktoren vermeidet die Vermischung von zwei Konzepten im Schema, spie-

gelt aber dem Experten-Benutzer, der Bausteintypen erstellt, eine mit Bausteintypen vergleichbare Flexibilität vor, die so nicht besteht. Aktoren werden von den Client-Anwendungen für den Experimentaufbau direkt auf festkodierte Anwendungslogik abgebildet, die über ein wohl definiertes und nur in Grenzen durch die festgelegten Parameter variierbares Verhalten verfügen. Zudem sind im Gegensatz zu den Bausteintypen keine komplexen Attribute erforderlich. Die mit dieser Modellierung mögliche Flexibilität ist somit nicht erforderlich und erhöht lediglich den Aufwand.

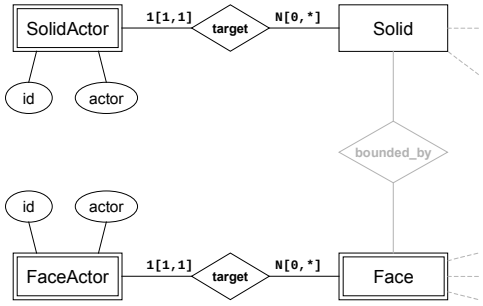
Benutzerdefinierter Typ Aktor

Bei Betrachtung der Merkmale von Aktoren, insbesondere der Existenz einer vom System vorgegebene Menge von Aktortypen mit jeweils einer fest definierten Menge von Parametern, bietet sich die Verwendung von benutzerdefinierten Datentypen in der Datenbank an. Als Basistyp dient ein nicht instanzierbarer Datentyp „Aktor“, der über die grundlegenden Attribute verfügt, die alle Aktoren gemeinsam haben, beispielsweise einen Identifikator, der den Aktortyp angibt. Von diesem Typ werden dann benutzerdefinierte Typen für die konkreten Aktoren abgeleitet. Die Attribute dieser Typen spiegeln dabei die Parameter des jeweiligen Aktortyps wider. Diese Lösung entspricht im Umfang ihrer Flexibilität genau den Anforderungen zur Darstellung von Aktoren. Es wird keine nicht genutzte Flexibilität im Schema eingeführt. Dennoch kann die Datenhaltungsschicht jederzeit leicht an neue Aktortypen angepasst werden, indem ein entsprechender neuer Subtyp des Datentyps Aktor erzeugt wird. Da es sich beim Ergänzen eines neuen Aktortyps in jedem Fall um einen administrativen Vorgang handelt, der von einem mit den Interna des Systems vertrauten Entwickler durchgeführt wird, stellt dies keine Erschwerung der Erweiterbarkeit dar.

Die Ergänzung des Schemas um Aktoren erfordert eine neue Entity-Menge mit einem Attribut vom Typ Aktor oder die Erzeugung einer Hierarchie von Entities entsprechend den verschiedenen Aktortypen. Elemente dieser Entity-Menge haben dann eine Beziehungen zu einer der Entities, die als Wirkungsziel eines Aktors fungieren können, hier also Körper und Flächen. Eine Erweiterung um Kanten- oder Punktaktoren ist problemlos möglich.

Alternativ kann auch für jedes Ziel eines Aktors eine eigene schwache Entity-Menge vorgesehen werden. Bei entsprechender Modellierung im relationalen Schema kann so sichergestellt werden, dass ein Aktor immer genau einem Ziel zugeordnet ist. Abbildung 29 zeigt das ERM für diese Realisierungsmöglichkeit.

Abbildung 29 Modellierung von Aktoren als schwache Entities



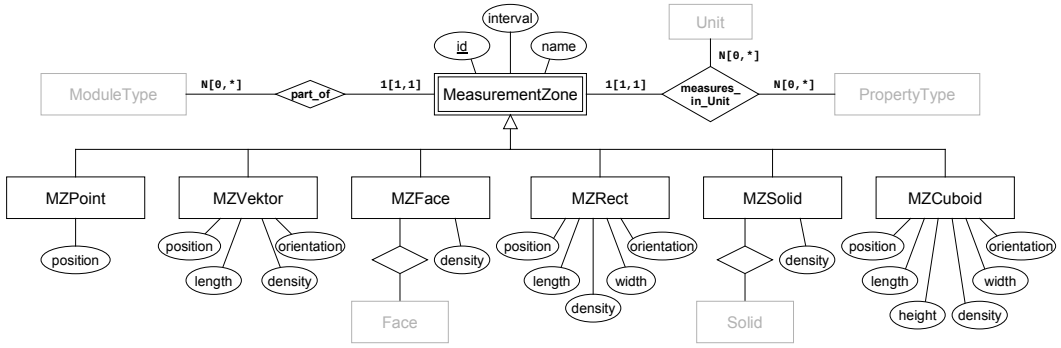
4.3.5.3 Messzonen

Neben statischen und dynamischen Merkmalen bilden Messzonen den dritten Bestandteil des funktionalen Modells. Sie beeinflussen die Experimentdurchführung nicht, sondern dienen der Datengewinnung. Gemäß Anforderung P2.5.1 können sie punkt-, kurven-, flächenförmig und dreidimensional sein. Messzonen können unabhängig von der bestehenden Geometrie des Bausteins sein, oder es kann einzelnen Flächen oder Körpern die Messzonen-Eigenschaft zugewiesen werden.

Messzonen verfügen ähnlich wie Aktoren über fest vorgegebene Parameter, die vom Bausteindesigner gewählt werden und wie Aktorparameter von Benutzer über Bausteinparameter modifiziert werden können. Daher bietet es sich an, eine gemeinsame Superklasse für alle Messzonen vorzusehen, die über entsprechende Attribute verfügt. Von dieser Klasse erben die speziellen Typen von Messzonen, deren Eigenschaften über zusätzliche Parameter beziehungsweise Relationships zu Geometrieelementen festgelegt werden. Abbildung 30 zeigt die Erweiterung des Informationsmodells. Eine Messzone (*MeasurementZone*) ermittelt Werte von einem bestimmten Typ in einer von dessen Einheiten. Hierzu wird das Einheits- und Typssystem für Bausteineigenschaften mitverwendet. Unter dem Namen einer Messzone ist später der Zugang zu den von ihr ermittelten Daten möglich, *interval* gibt die Messhäufigkeit an. Höherdimensionale Messzonen haben eine Attribut *density*, das die Dichte der Messpunkte innerhalb des von der Messzone abgedeckten Bereichs beschreibt. Zur Vereinfachung soll die Dichte der Messpunkte konstant und einheitlich in allen Dimension sein.

Für nicht an Geometrieelemente gebundene Messzonen sind zusätzlich Angaben zu Position, Orientierung und den Abmessungen zu machen.

Abbildung 30 ERM Messzonen



4.3.6 Interaktionsmodell

Um aus den Einzelbausteinen, die in einem Experimentaufbau verwendet werden, eine funktionsfähige Apparatur zu bilden, müssen die einzelnen Bausteininstanzen sinnvoll verbunden werden. Um den Benutzer bei dieser Tätigkeit unterstützen zu können, ist es erforderlich, die erlaubten Bauteilkombinationen und die Art der Interaktion zwischen den Bauteilen zu beschreiben. Die geschieht über Verbindungsobjekte, kurz Verbindung oder *Connection*. Zu einer Verbindung zwischen zwei Bausteintypen gehören immer (mindestens) zwei Bestand-

teile, die gewissermaßen die Gegenstücke bilden, jeder an der Verbindung beteiligte Bausteintyp verfügt über eines dieser im Folgenden als Verbindungselement oder Konnektor (*Connector*) bezeichneten Elemente.

4.3.6.1 Verbindungselemente

Um zwei Bauteile verbinden zu können, müssen sie über zueinander passende Verbindungselemente verfügen. Diese Verbindungselemente sind Teil der Geometrie eines Bauteils. Eine Verbindung kann symmetrisch sein, was bedeutet, dass beide an der Verbindung beteiligten Verbindungselemente identisch sind. Häufiger wird man jedoch unsymmetrische Verbindungselemente antreffen. Hier sind die beiden an einer Verbindung beteiligten Verbindungselemente nicht identisch wie Stecker und Steckdose, „Nut und Feder“ etc.

4.3.6.2 Beschreibung der Kompatibilität

Einige mögliche Verfahren zur Beschreibung dieser Informationen sollen im Folgenden vorgestellt werden.

Implizite Kompatibilität

Deckungsgleichheit von Flächen

Beschränkt man sich bei der Definition von Verbindungselementen auf einzelne Oberflächen der zu verbindenden Gegenstände, ließe sich die Kompatibilität auf Basis der Deckungsgleichheit dieser Flächen ausdrücken. Ein solcher Test ist bei ebenen Flächen mit vertretbarem Rechenaufwand durchzuführen. Häufig genügt eine gemeinsame Fläche jedoch nicht zur Definition eines Verbindungselementepaars.

Kompatibilität von Körpern

Sind einfache Flächen als Verbindungselemente nicht ausreichend, kann eine Prüfung erfolgen, ob ein Verbindungselement mechanisch auf das andere passt. Das bedeutet, dass die Form des einen Verbindungselements in einem gewissen Toleranzbereich das Negative der Form des anderen Verbindungselements ist, sie also Gegenstücke bilden. Ist eine solche Frage für einen Menschen meist nach kurzem Probieren recht schnell zu beantworten, so handelt es sich doch um ein komplexes algorithmisches Problem. So genügt es für die Kompatibilität, dass nur ein Teil der Formen zueinander passt. Es muss also auf Ebene der ganzen Form als auch auf Teilen nach Kompatibilität gesucht werden. Je weiter man diese Unterteilung bei der Suche fortführt, desto aufwendiger wird die Suche, und desto wahrscheinlicher wird das Auffinden von zueinander passenden Strukturen.

Ob die so aufgefundenen Kompatibilitäten ausreichend sind, um die Verbindungselemente als passend mit all den Implikationen dieses Begriffs bezüglich mechanischer Stabilität, Dichtheit oder Ähnlichem zu bezeichnen, lässt sich nur schwer automatisiert beurteilen. Ein weiteres Problem bereitet die Erfüllung der Anforderung P2.2.1.3.1: Sie beschreibt ein Suchverfahren, das die zu einem bekannten Verbindungsstück kompatiblen Elemente aufzeigt. Um diese Suche mit akzeptablem Laufzeitverhalten durchführen zu können, muss sie

vom Datenbanksystem unterstützt werden, da sonst jeweils alle Bausteintypen in die Anwendung geladen und dort untersucht werden müssen. Das DBMS müsste also eine entsprechende Prüfung selbst durchführen und dazu die Geometrie rekonstruieren. Eine solche komplexe Funktionalität mit DB-Operationen zu erreichen, kann als praktisch nicht realisierbar betrachtet werden.

Explizite Kompatibilität

Statt aufgrund von Eigenschaften der Verbindungselemente auf Kompatibilität zu schließen, kann sie auch durch Angabe einer Liste aller zu einem Verbindungselement kompatiblen Elemente explizit definiert werden. Während so die Probleme der automatischen Kompatibilitätsprüfung wie hoher Rechenaufwand und möglicherweise falsch positive Kompatibilitätsklassifizierung vermieden werden, bedeutet dieser Ansatz einen größeren Aufwand bei der Erstellung neuer Bauteile. So muss die Kompatibilität des neuen Bausteins zu allen relevanten bestehenden Bausteinen überprüft werden.

Eine Möglichkeit zur Vermeidung dieses Problems ist die Definition von Paaren oder Gruppen zueinander kompatibler Verbindungselemente. Auf diese kann dann beim Entwurf des Bauteils zurückgegriffen werden. Durch Integration des Verbindungselements in ein neues Bauteil werden automatisch alle Bausteine, die ein Gegenstück aus der gleichen Gruppe einsetzen, zu diesem kompatibel.

Da Verbindungselemente selbst parametrisiert werden können, zum Beispiel durch Veränderung des Durchmessers eines Anschlusses, kann die Kompatibilität zwischen den Mitgliedern dieser Gruppe von auf ihren Parametern formulierten Bedingungen abhängig gemacht werden. So sind die Verbindungselemente „Schraube“ und „Mutter“ grundsätzlich kompatibel, die Kompatibilität von zwei speziellen Instanzen kann jedoch von Durchmesser und Gewindeform abhängig gemacht werden.

4.3.6.3 Art der Interaktion

Neben der rein boolschen Information über die Kompatibilität zwischen zwei Verbindungselementen in der Form „Verbindungselement X passt/passt nicht zu Verbindungselement Y“ ist auch eine Beschreibung der Art des Zusammenwirkens erforderlich. Dazu genügt bei rein mechanischem Zusammenfügen von Bauteilen und einer Abstraktion beispielsweise von elektrischen Verbindungen, wie sie im virtuellen Labor Verfahrenstechnik erfolgt, eine Angabe, wie die Lage der beiden Verbindungselemente zueinander bei hergestellter Verbindung ist. Wird die Kompatibilität von Bauteilen durch Prüfung der Form wie oben unter „implizite Kompatibilität“ beschrieben durchgeführt, ist diese Lageinformation Teil des Prüfungsprozesses. Bei der expliziten Kompatibilitätsbeschreibung müssen Position und Orientierung jedoch bei der Erstellung der Verbindungselemente spezifiziert werden. Dabei ist es zulässig, dass mechanische Verbindungselemente in mehr als einer Position oder Orientierung zueinander passen. So können runde Anschlüsse wie beispielsweise Rohre in beliebigem Drehwinkel verbunden werden, während ein Vierkantrohr mit quadratischer Querschnittsfläche nur vier Orientierungen zulässt.

Gemeinsames lokales Koordinatensystem

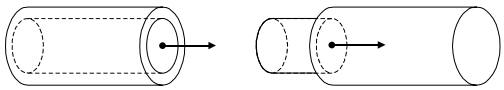
Eine einfache Art, die Lage der beiden an einer Verbindung beteiligten Verbindungselemente zu definieren, ist die Beschreibung der Form der Verbindungselemente in ihrer späteren verbundenen Lage in einem gemeinsamen Koordinatensystem. Bei der Herstellung einer Verbindung genügt es später, die Ursprünge der Koordinatensysteme in Deckung zu bringen, um die korrekte Lage der Elemente zueinander sicherzustellen. Dieser Ansatz beschreibt jedoch nur eine gültige Lage der Verbindungselemente.

Gemeinsamer Punkt mit Richtungsvektoren

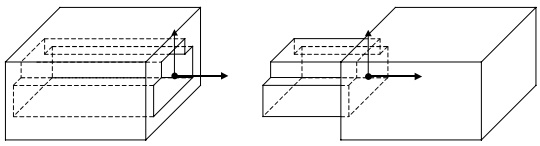
Eine Verallgemeinerung des Ansatzes des gemeinsamen Koordinatensystems beschreibt Punktepaare innerhalb der lokalen Koordinatensysteme der beteiligten Verbindungselemente, die bei erfolgter Verbindung zur Deckung gebracht werden, ergänzt um einen oder mehr Vektoren, welche die erlaubten Orientierungen beschreiben. Die gemeinsamen Punkte entsprechen dabei dem Ursprung des Koordinatensystems bei der zuvor vorgestellten einfachen Lösung, die Orientierungsvektoren den Koordinatenachsen.

Zur Definition einer möglichen Positionierung mit einer beliebigen Zahl von Freiheitsgraden genügt die Angabe des gemeinsamen Punktes. Um diesen können die beiden Elemente dann frei rotiert werden. Abbildung 31 zeigt Möglichkeiten zur Einschränkung der Bewegungsfreiheit: Die Beschränkung auf eine Rotationsachse erfolgt durch Angabe des Richtungsvektors dieser Rotationsachse (Achsenvektor). Nimmt man einen weiteren nicht zum Achsenvektor parallelen Richtungsvektor (Orientierungsvektor) hinzu, definiert man eine einzelne gültige Position. Um eine Zwischenstufe zwischen beliebiger freier Rotation und einer einzelnen Position beschreiben zu können, kann optional die Angabe mehrerer Orientierungsvektoren erlaubt werden.

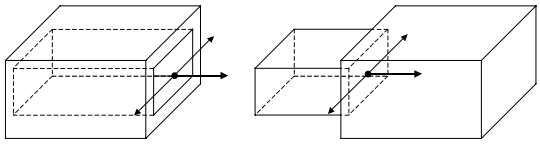
Abbildung 31 Positionierung von Konnektoren



a) Freie Drehbarkeit durch gemeinsamen Punkt und Achsenvektor



b) Eine zulässige Position durch Achsen- und Orientierungsvektor



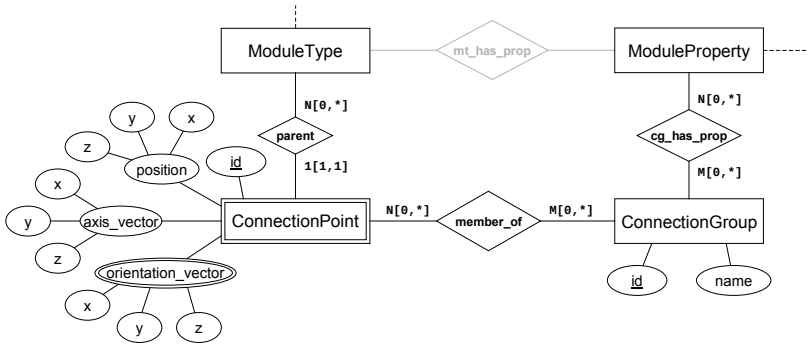
c) Mehrere zulässige Positionen durch Achsenvektor und Orientierungsvektoren

Um symmetrische Konnektoren effizient zu unterstützen, kann alternativ zu der abgebildeten Form von Achsen- und Orientierungsvektoren, die bei hergestellter Verbindung zur Deckung gebracht werden, auch festgelegt werden, dass jeweils der Gegenvektor dieser Vektoren zur Deckung gebracht wird.

4.3.6.4 Informationsmodell für explizite Kompatibilität

Konnektoren verfügen über Geometrie und Parameter. Sie können daher als spezielle Form von Bausteintypen betrachtet werden, die zusätzlich über Verbindungspunkte und deren optionale Achsen- und Orientierungsvektoren verfügen. Daher bietet sich eine entsprechende Erweiterung des domänenunabhängigen Baustein-Informationsmodells an. Abbildung 32 zeigt eine möglicher Umsetzung dieser Erweiterung. Ein ModuleType kann über eine beliebige Anzahl von Verbindungspunkten verfügen, die als schwaches Entity *ConnectionPoint* repräsentiert sind. Ein Verbindungspunkt verfügt über eine Position, einen optionalen Achsenvektor und über eine Menge von Orientierungsvektoren, die hier als mengenwertiges Attribut dargestellt werden. Ein Verbindungspunkt ist Mitglied in (Relationship *member_of*) einer oder mehrerer Verbindungsgruppen (Entity *Connection Group*). Alle Bausteintypen, die über Verbindungspunkte in einer Verbindungsgruppe verfügen, sind prinzipiell zueinander kompatibel. Um die Kompatibilität von Eigenschaftswerten einer konkreten Instanz eines Bausteintyps abhängig machen zu können, verfügt eine Verbindungsgruppe über eine Menge von Eigenschaften, die durch die Relationshipmenge *cg_has_prop* dargestellt werden. Alle Bausteine einer Verbindungsgruppe müssen über mindestens diese Eigenschaften verfügen, auf denen dann Integritätsbedingungen formuliert werden können, welche die Kompatibilität einschränken.

Abbildung 32 ERM für das Interaktionsmodell



Mit dem beschriebenen Informationsmodell lassen sich alle für die Domäne Verfahrenstechnik erforderlichen Informationen zur Verbindung von Bausteinen darstellen. Zusätzlich ermöglicht es die effiziente Suche nach kompatiblen Bausteintypen in der Datenbank durch Abbildung aller Aspekte im Schema.

4.3.7 Konsistenzmodell - Systemsicht

Bei der Entwicklung der domänenunabhängigen Anforderungen in Zusammenarbeit mit dem Bereich Siedlungswasserwirtschaft wurden in Kapitel 3.2.5 bereits Klassen von Integritäts-

bedingungen oder Constraints definiert, die in einem virtuellen Labor erfüllt sein müssen. Es wurden Constraints auf Ebene einzelner Eigenschaften beschrieben und solche, die für einzelne Bausteintypen gelten. Ebenso ist die Definition von Bedingungen an die Nutzung von Bausteintypen zu erlauben, die mit Bezug auf Gruppen von Bausteinen formuliert werden. Die Gruppe von Constraints mit dem größten Wirkungsbereich sind systemweit gültig und repräsentieren Invarianten des virtuellen Labors.

4.3.7.1 Zuordnung von Integritätsbedingungen zu Nutzer- oder Systemsicht

Wegen der konzeptionellen Trennung des Bausteinmodells in eine System- und eine im Anschluss beschriebene Nutzersicht soll eine Zuordnung der in den Anforderungen informell beschriebenen Integritätsbedingungen zu diesen beiden Sichten erfolgen.

Als Kriterium für die Zuordnung dient die Wirkung der Integritätsbedingungen auf die beiden grundlegenden Nutzerklassen: Wirken sich Constraints auf die Arbeit des einfachen Laborbenutzer aus, der vorhandene Experimentbausteine verwendet, werden sie dem Konsistenzmodell der Nutzersicht zugeordnet. Dies wird im Kapitel 4.3.9 beschrieben. Im Allgemeinen werden Constraints der Nutzersicht vom Entwickler eines Bausteintyps definiert.

Dienen Constraints dagegen dazu, die Entwicklung fehlerhafter Bausteintypen durch Expertennutzer zu verhindern, beziehen sie sich also auf die interne Darstellung des Bausteins, so sind sie Teil des Integritätsmodells der Systemsicht.

Neben Constraints, die eindeutig einer der Sichten zuzuordnen sind, sind andere für beide Sichten von Bedeutung. Dabei handelt es sich im Allgemeinen um die systemweit gültigen Invarianten. Sie werden im globalen Konsistenzmodell (Abschnitt 4.3.11) gesondert beschrieben.

4.3.7.2 Realisierungsmöglichkeiten für Constraints

Bevor näher auf einzelne Integritätsbedingungen der Systemsicht eingegangen wird, sollen die bereits in den Abschnitten 3.2.5.5 und 3.2.5.6 kurz vorgestellten Arten der Definition von Integritätsbedingungen und der Sicherstellung ihrer Einhaltung diskutiert werden. Die Überprüfung kann in jeder Schicht der in Kapitel 4.1 vorgestellte Systemarchitektur für das virtuelle Labor erfolgen.

Datenhaltungsschicht

Moderne objekt-relationale Datenbanksysteme, die nach der Vorstellung von möglichen Alternativen in Kapitel 4.2 als Vorschlag zur Realisierung der Datenhaltungskomponente ausgewählt wurden, bieten verschiedene Möglichkeiten zur Formulierung von Integritätsbedingungen. Der SQL:1999-Standard beschreibt dazu Assertions, Trigger und ECA-Regeln. Assertions erlauben die Formulierung von über die einfachen referentiellen Constraints des relationalen Datenmodells hinausgehenden Bedingungen auf den Daten. Diese dürfen sich über mehrere Tabellen erstrecken. Ihre Prüfung kann permanent oder erst am Ende einer Transaktion stattfinden. Trigger erlauben die Beschreibung von aktivem Datenbankverhal-

ten, indem bedingte Reaktionen auf Einfüge- Änderungs- oder Löschoperationen auf einzelnen Tabellen definiert werden können. ECA-Regeln sind ein Konzept, welches Assertions und Trigger vereint.

Diese Konzepte erlauben eine große Ausdrucksmächtigkeit, allerdings werden sie wie viele Elemente des SQL:1999-Standards von verfügbaren Produkten nur teilweise und unvollständig unterstützt. Allgemeine ECA-Regeln werden von keinem Datenbankprodukt angeboten, Assertions sind nur auf Ebene einer Relation möglich und oft fehlt die Möglichkeit, die Überprüfung bis zum Ende einer Transaktion zu verzögern. Wegen proprietärer Erweiterungen und nicht standardkonformer Syntax der Regeln ist deren Einsatz wegen der Verschlechterung der Portabilität nur bedingt sinnvoll.

Aufgrund der erforderlichen Dekomposition der Laborobjekte wie Bausteintypen und Experimente genügt die in der Praxis verfügbare Ausdrucksmächtigkeit nicht für die Formulierung komplexerer Integritätsbedingungen. Werden einzelne Bestandteile des Bausteinmodells in Form von aus Datenbanksicht unstrukturiertem Text wie einem XML-Dokument oder einem *binary large object* beschrieben, versagen diese Verfahren ebenfalls.

Oft ermöglicht die vorhandene Ausdrucksmächtigkeit jedoch die Realisierung einzelner Integritätsbedingungen. Hier muss jedoch neben dem möglicherweise erhöhten Aufwand bei der Formulierung auch der Aufwand zur Prüfung im Vergleich zu anderen Lösungen berücksichtigt werden. Da zudem für die Überprüfung durch die Datenbank das Einbringen der Daten mittels Check-In erforderlich ist, eignen sich Integritätsbedingungen in der Datenbank nicht für schon während der Bearbeitung zu prüfende Constraints. Sie können dennoch zum Schutz der Daten vor Fehlern in der Anwendung in die Datenbank integriert werden.

Datenmodellschicht

Einer der wichtigsten Gründe für die Einführung einer Zwischenschicht zwischen Datenhaltung und Anwendung war die Möglichkeit, an dieser Stelle eine Integritätsprüfung durchzuführen. Da die Datenmodellschicht bereits die endgültige Objektstruktur aus den Strukturen der Datenhaltungsschicht rekonstruiert beziehungsweise die Rückabbildung auf die Datenbank vornimmt, ist eine Formulierung der Integritätsbedingungen hier häufig deutlich einfacher und ihre Überprüfung weniger aufwendig. Die Constraints werden entweder als Teil der Daten in einer Beschreibungssprache in der Datenbank gespeichert, die dann von der Datenmodellschicht interpretiert und überprüft werden können, oder sie können hartkodiert realisiert werden. Für die stets anzuwendenden Constraints der Systemsicht wird man im Allgemeinen die zweite Variante wählen.

Wird ein Constraint wie zuvor beschrieben sowohl in der Datenmodellschicht als auch innerhalb der Datenbank geprüft, wäre es möglich, es dort zu extrahieren und auch in dieser Schicht zu nutzen. So würde eine doppelte und möglicherweise inkonsistente Formulierung der Constraints vermieden. Dagegen sprechen im Allgemeinen jedoch die großen Unterschiede in den Datenmodellen von Datenbank und Datenmodellschicht, welche die Vorteile der einfacheren Formulierung und Prüfung an dieser Stelle vergeben würden. Zudem würde die Abhängigkeit der Datenmodellschicht vom verwendeten DBMS vergrößert.

Anwendungsschicht

Die von der Datenmodellschicht rekonstruierten Objekte werden während der Bearbeitung durch den Benutzer von der Anwendungsschicht verwaltet und modifiziert. Um dem Benutzer unmittelbare Rückmeldung bei Fehlern geben zu können, muss eine Prüfung auch ohne den Umweg über die Datenhaltungsschicht möglich sein, deren Dienste normalerweise nur beim Lesen und Schreiben von Daten genutzt werden.

Indem die Datenmodellschicht die von ihr aus den Daten extrahierten Constraints den rekonstruierten Anwendungsobjekten mit auf den Weg gibt, können die dort entwickelten Mechanismen auch in der Anwendung genutzt werden.

Zusätzliche Integritätsbedingungen, die nicht für jede der drei Teilanwendungen erforderlich sind, können zudem hartkodiert werden.

Präsentationsschicht

Die Präsentationsschicht kann selbst einfache Aufgaben bei der Sicherstellung der Integrität übernehmen. Zwar hat sie im Wesentlichen die Aufgabe, den Benutzer auf Verletzungen der in den tieferen Schichten geprüften Constraints hinzuweisen, sie kann aber auch selbst einfache Fehler prüfen. Beispielsweise kann die korrekte Formatierung von Nutzereingaben sichergestellt werden.

4.3.7.3 Beispiele für Integritätsbedingungen der Systemsicht

Im Folgenden sollen einige wichtige Integritätsbedingungen der Systemsicht beschrieben und Vorschläge zur Wahl der jeweils geeigneten Realisierungsmöglichkeit gemacht werden.

Topologische Gültigkeit

In Kapitel 4.3.3.3 wurden die Euler- und Euler-Poincaré-Gleichungen als Möglichkeit vorgestellt, die topologische Konsistenz eines Solids in BRep-Darstellung zu überprüfen. Da für die Überprüfung dieser Gleichungen das Wissen über die Anzahl der topologischen Elemente des Solids ausreichend ist, kann eine derartige Prüfung bereits innerhalb des Datenbanksystems mit den dort vorhandenen Möglichkeiten erfolgen, wenn eine Boundary Representation zum Beispiel in Winged-Edge-Darstellung auf das Relationenmodell abgebildet wird.

Korrekte Nutzung von Aktoren

Bei der Definition des Verhaltens von Bausteinen greift der Entwickler auf die in Abschnitt 4.3.5.2 beschriebenen Aktoren zurück. Diese erfordern im Allgemeinen Parameter, die ihr Verhalten festlegen. Für diese Parameter muss sichergestellt werden, dass sie in einem zulässigen Bereich liegen.

Da sich Änderungen der Nutzersicht auf die Parametrisierung der Aktoren auswirken kann, genügt eine einmalige Prüfung dieser Parameter bei der Erstellung nicht, sie muss daher benutzernah zum Beispiel in der Anwendungsschicht erfolgen. Idealerweise verhindert jedoch das später in Kapitel 4.3.9 beschriebene Konsistenzmodell der Nutzersicht ungültige Werte

für Aktoren. Da es sich bei Aktoren um vom System vorgegebene Objekte handelt, die nicht ohne Spezialkenntnisse erstellt werden können, kann diese Prüfung hartkodiert durch den Aktor selbst erfolgen.

Verbindungselemente

Um wie in Kapitel 4.3.6 beschrieben die Kompatibilität von Bausteintypen mit gemeinsamen Verbindungspunkten durch Bezugnahme auf Eigenschaften der Bausteininstanzen einzuschränken, müssen alle Bausteintypen mit Verbindungspunkten in einer Verbindungsgruppe mindestens über alle Eigenschaften dieser Gruppe verfügen. Dies lässt sich einfach mit den zuvor beschriebenen Mitteln aktueller relationaler Datenbanksysteme sicherstellen.

4.3.7.4 Fazit - Konsistenzmodell Systemsicht

Wie die vorgestellten Beispiele zeigen, kann eine Implementierung von Konsistenzbedingungen der Systemsicht nicht allein auf einer der vorgestellten Realisierungsmöglichkeiten beruhen. Stattdessen verspricht ihre gemeinsame Nutzung eine Kombination ihrer Vorteile und eine Umgehung der Schwachstellen. So sollten Prüfungen innerhalb der Datenbank auf mit vertretbarem Aufwand realisierbare Constraints mit systemweiter Gültigkeit beschränkt werden, für die eine Prüfung beim Check-In ausreichend ist. Systemweit gültige Constraints, die auch während der Bearbeitung zu prüfen sind, sollten hartkodiert in der Anwendungsschicht geprüft werden. Um eine einheitliche Prüfung derartiger Constraints zu garantieren, können die Constraints in den von der Datenmodellschicht gelieferten Objekten implementiert sein, so dass eine Prüfung in der Anwendungsschicht beispielsweise durch Aufruf von Validierungsmethoden geschehen kann.

4.3.8 Bausteineigenschaften

Anforderung P2.2.2.2 beschreibt das Konzept der Bausteineigenschaften neben der graphischen Darstellung als zentralen Bestandteil der Abstraktion von der System- auf die Nutzersicht. Sie sind ein wichtiges Mittel, um dem Benutzer ein Verständnis vom Aussehen, der Funktionsweise und den Einsatzmöglichkeiten zu geben. Zugleich können sie durch Einbeziehung von für die Verwendung im virtuellen Labor zunächst unerheblichen, aber das reale Gegenstück eines Bausteins beschreibenden Informationen ein wichtiges Bindeglied zwischen realem und virtuellem Experiment darstellen. Zudem sind sie gemäß Anforderung P2.3.3.2 auch Grundlage für die Suche nach Bausteinen innerhalb der Bausteinbibliothek. Das Bausteinmodell muss bei der Realisierung der Bausteineigenschaften diese Anforderungen berücksichtigen.

Bei der Suche nach gemeinsamen Anforderungen der virtuellen Labore Verfahrenstechnik und Siedlungswasserwirtschaft wurde festgestellt, dass auch in anderen Domänen die Eigenschaften einen zentralen Bestandteil der Beschreibung der Bausteine bilden. Zwar unterscheiden sich diese Anforderungen erheblich bezüglich der Anzahl von Eigenschaften, den benötigten Typen und dem Bedarf nach freier Definierbarkeit von Eigenschaften für verschiedene Bausteine, wegen der bestehenden Gemeinsamkeiten wurden sie jedoch bei der Entwicklung eines domänenunabhängigen Informationsmodells in Kapitel 3.3.1 berücksich-

tigt. Das dort beschriebene Konzept erlaubt große Flexibilität bezüglich der möglichen Typen und Werte von Eigenschaften und lässt gleichzeitig die effiziente Speicherung einer Vielzahl von identischen Bausteinen zu.

Im Labor Siedlungswasserwirtschaft enthalten die Eigenschaften eines Bausteins zusammen mit seiner für den Benutzer sichtbaren Position und Form alle für die Durchführung eines virtuellen Experiments erforderlichen Informationen. Die Systemsicht ist also weitgehend deckungsgleich mit der Benutzersicht. Für das virtuelle Labor Verfahrenstechnik wird das gleiche Eigenschaftsmodell dagegen für die abstrahierenden Informationen der Benutzersicht verwendet, während für die interne Darstellung die in den vorherigen Kapiteln beschriebenen Konzepte die Systemsicht bilden.

Auf die erforderliche Abbildung zwischen der durch die Eigenschaften repräsentierten Nutzer- und der für die Simulation erforderlichen Systemsicht wird in Kapitel 4.3.10 näher eingegangen.

4.3.8.1 Einheiten

Anforderung P2.2.2.2.1 geht über die Möglichkeiten des domänenunabhängigen Informationsmodells hinaus: Zur an dieser Stelle beschriebenen Unterstützung von Einheiten für numerische Eigenschaftstypen und die zur automatischen Umrechung nötige Information ist eine Erweiterung nötig. Die wichtigsten Informationen über eine Einheit umfassen ihren Namen und ein Kürzel. Die Konversion benötigt prinzipiell Umrechnungsinformationen von jeder Einheit in jede andere Einheit eines Typs. Da so die Menge der Umrechnungsfaktoren mit der Anzahl der Einheiten quadratisch wächst und zudem der Aufwand für das Hinzufügen neuer Einheiten enorm ist, wird stattdessen eine Einheit des Typs als Standardeinheit definiert. Alle anderen Einheiten benötigen dann nur Konversionsinformationen von und zu der Standardeinheit. Als Konversionsinformationen für eine Einheit genügt ein Umrechnungsfaktor und ein Offset, wenn man sich auf Einheiten beschränkt, die linear ineinander umgerechnet werden können. Durch Multiplikation mit dem Konversionsfaktor und anschließende Addition des Offsets wird zur Standardeinheit umgerechnet, die Umkehrung dieser Operation konvertiert von der Standardeinheit in die jeweilige Nicht-Standardeinheit. Alternativ ist auch die inverse Definition von Faktor und Offset möglich.

Die Standardeinheit wird durch einen Offset von 0 und einen Konversionsfaktor von 1 ausgezeichnet. Die Einheit des Konversionsfaktors einer Nicht-Standard-Einheit ist immer implizit der Quotient aus der Standardeinheit durch die jeweilige Nicht-Standard-Einheit, die Einheit des Offset ist die Standardeinheit.

Beispiel 3 verdeutlich die Umrechnung:

Beispiel 3 Konversion von Temperaturen

Einheiten des Typs Temperatur:

Name	Konversionsfaktor	Konversionsoffset	Bemerkung
Kelvin [K]	1 [K/K = 1]	0 [K]	Std.-Einheit
Grad Celsius [°C]	1 [K/°C]	273,16[K]	
Grad Fahrenheit [°F]	$0,\overline{5}$ [K/°F]	$255,38\overline{2}$ [K]	

Konversion von 37°F nach °C:

1) Bekannte direkte Konversionsformel °F → °C:

$$\text{Wert in Celsius} = (\text{Wert in Fahrenheit} - 32°\text{F}) / 1{,}8\ \frac{°\text{C}}{°\text{F}}$$

$$(37°\text{F} - 32°\text{F}) / 1{,}8\ \frac{°\text{C}}{°\text{F}} = 2,\overline{7}\,°\text{C}$$

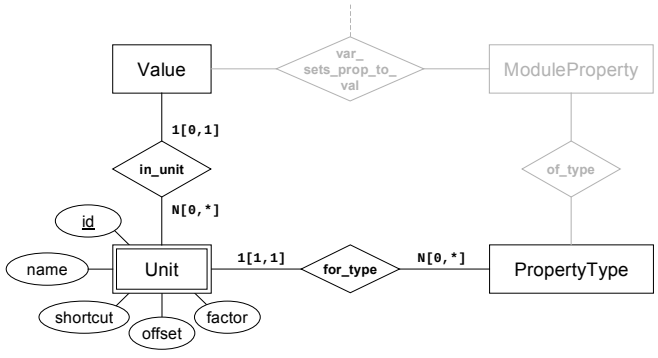
2) Indirekte Konversion über Standardeinheit:

$$37°\text{F} \cdot 0,\overline{5}\ \frac{\text{K}}{°\text{F}} + 255,38\overline{2} = 275,93\overline{7}\,\text{K}$$

$$(275,93\overline{7}\,\text{K} - 273,16\text{K}) / 1\ \frac{\text{K}}{°\text{C}} = 2,\overline{7}\,°\text{C}$$

Um die nötigen Informationen für die Unterstützung von Einheiten zu integrieren, ist als Ergänzung ein von PropertyType existenzabhängiges schwaches Entity *Unit* vorzusehen, die über die Attribute Name, Shortcut, (Konversions-)Faktor und Offset verfügt. Abbildung 33 zeigt diese Erweiterung. Werte können in einer bestimmten Einheit ihres Typs vorliegen, was durch die Relationship *in_unit* dargestellt wird. Ein Datenbankconstraint sollte sicherstellen, dass ein Wert nur in Einheiten seines Typs vorliegt.

Abbildung 33 Ergänzung des schwachen Entity Unit



4.3.8.2 Unterscheidung Baustein-/Betriebsparameter

Die in Anforderung P2.2.2.2.6 beschriebene Trennung von Baustein- und Betriebsparametern kann im Allgemeinen nicht automatisch erfolgen. Vielmehr muss der Bausteinentwickler beurteilen, ob die Veränderung eines Parameters ohne Nebenwirkungen auf andere Bausteine möglich ist oder nicht. Dazu wird im Kernschema das Flag *operational* zu Relationship *mt_has_prop* hinzugefügt. Für Parameter, bei denen dieses Flag gesetzt ist, kann dann entsprechend der Anforderung D2 dem Benutzer vor der Experimentdurchführung eine Modifikation ermöglicht werden.

4.3.9 Konsistenzmodell - Nutzersicht

Während das Konsistenzmodell der Systemsicht allgemeingültige Eigenschaften der Bausteine garantiert, hat das Konsistenzmodell der Nutzersicht die Aufgabe, die korrekte Verwendung des Bausteintyps sicherzustellen, wie sie von seinem Entwickler vorgesehen wurde.

Verwendung eines Bausteintyps bedeutet Instanzierung, also die Positionierung des Bausteins innerhalb eines Experiments (oder eines übergeordneten komplexen Bausteins) und die Wahl von Werten für alle verpflichtenden, variablen Eigenschaften. So können mögliche Fehlnutzungen durch ungeeignete Positionierung oder durch falsche Parameterwahl entstehen. Daher sind Mechanismen vorzusehen, die es dem Entwickler erlauben, Vorgaben für die zulässige oder unzulässige Benutzung seines Bausteintyps zu beschreiben.

Bei der Diskussion des Konsistenzmodells der Systemsicht in Abschnitt 4.3.7 wurden bereits verschiedene Möglichkeiten zur Realisierung von Constraints vorgestellt. Diese werden im Folgenden auf ihre Eignung zur Realisierung von Constraints der Nutzersicht geprüft.

4.3.9.1 Constraints von Eigenschaften

Der Definition von Integritätsbedingungen auf den Eigenschaften der Nutzersicht kommt eine enorme Bedeutung zu. Sie erlauben dem Entwickler eines Bausteintyps, dessen Verwendung auf die ihm zugedachten Einsatzmöglichkeiten zu beschränken. Da diese Integritätsbedingungen jeweils nur für einen einzelnen Bausteintyp Gültigkeit besitzen, können sie nicht fest in der Anwendungslogik kodiert werden. Auch eine Realisierung innerhalb der Datenbank erscheint nicht sinnvoll, da diese Constraints zum einen schon während der Erstellung von Experimentaufbauten und nicht erst beim Check-In geprüft werden müssen und zum anderen wegen ihrer großen Anzahl eine nicht unerhebliche Belastung des Datenbanksystems darstellen können. Daher sollten Integritätsbedingungen der Nutzersicht Teil der Bausteindaten sein.

Zulässigkeit von Nullwerten

Die für Parameter zu formulierenden Bedingungen sind so vielfältig wie die möglichen Parametertypen. Das Verbot von Nullwerten ist eine der wenigen Gemeinsamkeiten für alle Eigenschaften, es macht einen optionalen Parameter zu einem verpflichtenden Parameter (ver-

gleiche Anforderung P2.2.2.2.4). Wegen dieser Bedeutung ist die Zulässigkeit von Nullwerten bereits im gemeinsamen Informationsmodell in Kapitel 3.3.1 in Form des Flags *null allowed* berücksichtigt.

Bedingungen auf einzelnen Parametern

Für numerische oder komplexe Parameter können einzelne Werte als unzulässig erklärt oder umgekehrt die erlaubten Werte aufgezählt werden. Für numerische Parameter können auch gültige Wertebereiche definiert werden. Die Formulierung kann in diesem Fall durch einen algebraischen Ausdruck erfolgen, in dem in Form einer Variablen auf den beteiligten Parameter Bezug genommen wird. Die Datenmodellschicht kann die so formulierten Bedingungen beim Laden von Bausteinen interpretieren und als zur Laufzeit jederzeit auswertbares Objekt dem Baustein hinzufügen.

Bedingungen auf mehreren Parametern

Bedingungen können sich auch auf mehrere typkompatible Parameter (und auch nichtvariable Eigenschaften) beziehen. Die Vorgehensweise erfolgt analog zur Beschreibung von Bedingungen auf einem einzelnen Parameter durch Angabe eines algebraischen Ausdrucks, in dem nun mehrere Variablen stellvertretend für die beteiligten Parameter auftreten können.

4.3.9.2 Positionierung

Wie bei der Entwicklung von Bausteinen aus Solids ist auch bei der Bausteinverwendung eine Überschneidung der Solids verschiedener Bausteine zu vermeiden. Die Überprüfung kann durch den gleichen Mechanismus wie beim Konsistenzmodell der Systemsicht in Kapitel 4.3.7 beschrieben in der Anwendungsschicht erfolgen.

Neben dieser Invariante kann der Entwickler eines Bausteins laut Anforderung P2.2.5.1 weitere Bedingungen an dessen Positionierung formulieren. Dazu gehört das Einhalten von gewissen Mindestabständen zu anderen Bausteinen oder die Beschränkung auf einen gewissen Bereich von Orientierungen.

Mindestabstände

Bei der Definition von Mindestabständen muss ein Bereich um einen Baustein oder Teile eines Bausteins definiert werden, in dem keine Solids anderer Bausteine liegen dürfen. Statt einer komplizierten algebraischen Beschreibung können derartige „verbotene Zonen“ durch eine Ergänzung der Geometrie des Bausteins um spezielle Solids angegeben werden. Diese Solids werden zum Beispiel durch Verwendung eines speziellen Materials gekennzeichnet und wirken sich nicht auf die Experimentdurchführung aus. So kann die Einhaltung von Mindestabständen auf das Problem der Überschneidung von Solids zurückgeführt werden.

Orientierung

Ist ein Baustein nur in bestimmten Lagen oder Lagebereichen funktionsfähig, können diese durch Aufzählung der erlaubten Orientierungen in einer der in Kapitel 4.3.3.1 vorgestellten

Verfahren spezifiziert werden. Umgekehrt können einzelne verbotene Lagen angegeben werden. Für die Spezifikation von erlaubten oder nicht erlaubten Lagebereichen kann ein Bereich um eine Orientierung spezifiziert werden.

Mit diesen Angaben lassen sich bereits ein Großteil der erforderlichen Nutzungseinschränkungen für Bausteine beschreiben. Die Entwicklung einer mächtigeren Beschreibungssprache zur Formulierung von komplexen Beziehungen zwischen verschiedenen Bausteintypen würde den Rahmen dieser Arbeit überschreiten.

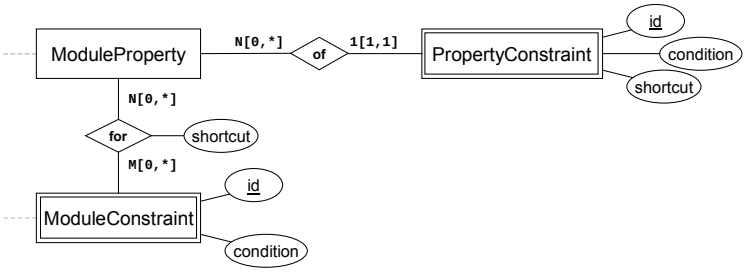
4.3.9.3 Verbindungen

Das in Kapitel 4.3.6 beschriebene Informationsmodell garantiert die Korrektheit von Bausteinverbindungen durch Beschreibung der Kompatibilität von Verbindungselementen. Zusätzlich sieht es eine weitere Beschränkung durch Constraints auf den gemeinsamen Eigenschaften einer Verbindungsgruppe vor. Eine einfache Möglichkeit besteht darin, implizit vorauszusetzen, dass die gemeinsamen Eigenschaften wertegleich sein müssen, um eine Verbindung zuzulassen. Dies erfordert keine weiteren Vorkehrungen im Informationsmodell. Sollen jedoch komplexere Anforderungen wie das Zulassen von gewissen Toleranzen beschrieben werden, muss auch hier ein Verfahren zur Definition dieser Bedingungen existieren. Die erforderliche Ausdrucksmächtigkeit entspricht der in Abschnitt 4.3.9.1 beschriebenen Formulierung von Bedingungen auf mehreren Parametern, nimmt hier jedoch Bezug auf die Eigenschaften der Verbindungsgruppe statt auf die des Bausteintyps.

4.3.9.4 Informationsmodell

Im in Kapitel 3.3.6 vorgestellten domänenunabhängigen Gerüst des Informationsmodells wurden lediglich Entities für Constraints auf Parameter- oder Bausteinebene (ModuleConstraint beziehungsweise PropertyConstraint) vorgesehen, die für die jeweilige Domäne erweitert werden können. Eine mögliche Erweiterung kann in Form eines Attributs *condition* erfolgen, das eine textuelle Beschreibung der Bedingungen mit Bezug auf die Parameter und Eigenschaften enthält. Die Gestaltung dieser Beschreibung hängt dabei vom Typ der Parameter ab. Um in der Beschreibung auf den beziehungsweise die Parameter Bezug nehmen zu können, kann für jede beteiligte Eigenschaft ein Variablenname vergeben werden. Bei PropertyConstraints kann hier ein Standardname eingeführt werden, oder man sieht ein weiteres Attribut hierfür vor. Für ModuleConstraints kann in der Relationship *for* ein solches Attribut vorgesehen werden. Abbildung 34 zeigt diese Erweiterungen.

Abbildung 34 Domänenspezifische Erweiterung für Constraints



Die Erweiterung des Interaktionsmodells von Bausteinen um Bedingungen auf Eigenschaften einer Verbindungsgruppe kann durch eine Entity *ConnectionGroupConstraint* erfolgen, ihr Aufbau gleicht dem von ModuleConstraint.

Über den genauen Aufbau der textuellen Beschreibung von Bedingungen kann je nach Typ der Eigenschaften und benötigter Ausdrucksmächtigkeit entschieden werden. Als Vorschlag für das Aussehen einer Beschreibungssprache für die textuelle Beschreibung von Bedingungen auf numerischen Parametern (Fest- oder Fließkomma) zeigt Abbildung 35 eine einfache Grammatik in erweiterter Backus-Naur-Form (EBNF).

Abbildung 35 EBNF einer einfachen Beschreibungssprache für Bedingungen

```
condition::=          simplecondition |
                      complexcondition |
                      'NOT'? '(' condition ')'

simplecondition::=    term comparator term

comparator::=         '=' | '!=' | '<' | '>' | '<=' | '>='

complexcondition::=   condition booleanoperator condition

booleanoperator::=    'AND' | 'OR' | 'XOR'

term::=               number |
                      variable |
                      term binaryoperator term |
                      unaryoperator '(' term ')'
                      '(' term ')'

number::=             digitlist |
                      digitlist '.' digitlist |
                      number unit

digitlist::=          digit |
                      digitlist | digit

digit::=              '0'-'9'

variable::=           letterlist |
                      letterlist digitlist variable?

letterlist::=         letter |
                      letterlist letter

letter::=             'a'-'z' | 'A'-'Z' | '$' | '&' | '§' | '_' | ...
```

```
binaryoperator::=    '+' | '-' | '*' | '/' | '%'

unaryoperator::=     'sin' | 'cos' | 'tan' | 'asin' | 'acos' |
                     'atan' | 'abs' | 'exp' | ...

unit::=              '[' variable ']'
```

4.3.10 Abstraktionsmodell

Nach der Beschreibung der wichtigsten Aspekte von System- und Nutzersicht des Bausteinmodells erfolgt nun die Beschreibung der Kopplung zwischen diesen beiden Sichten.

Dazu muss je nach Teilaspekt entweder eine Abbildung der Systemsicht auf die Nutzersicht beziehungsweise eine Beschreibung der Auswirkungen von Änderungen in der Nutzersicht auf die Systemsicht erfolgen.

4.3.10.1 Bausteindarstellung

Als Grundlage für die graphische Darstellung der Bausteine in der Experimentierumgebung kann die Bauteilegeometrie herangezogen werden. Ergänzt um Elemente des funktionalen Modells, insbesondere Farbe und Textur der verwendeten Materialien, kann so sehr einfach eine realitätsnahe dreidimensionale Darstellung erreicht werden. Eine optionale vereinfachende Darstellung durch Symbole für die einzelnen Bausteine erfordert deren explizite Speicherung und eine Beschreibung ihres Aussehens in Abhängigkeit der aktuellen Eigenschaften des Bausteins. Sie wurde im Rahmen dieser Arbeit nicht berücksichtigt.

4.3.10.2 Eigenschaften

Während bei der graphischen Darstellung die Abbildung von der System- auf die Nutzersicht erfolgt, sind bei den Bausteineigenschaften zunächst beide Abbildungsrichtungen zu beachten. So bewirkt einerseits die Änderung einer Eigenschaft eine Veränderung der Systemsicht, andererseits könnten sich Eigenschaften wie die Angabe der Bauteilabmessungen aus dessen Geometrie ergeben.

Würden beide Abbildungsrichtungen implementiert, könnte durch gegenseitige Abhängigkeiten bei der Abbildung zwischen den Sichten sehr leicht ein Zyklus von wechselseitigen Änderungen entstehen. Eine Berücksichtigung dieser Wechselwirkungen würde eine sichere Definition des Abstraktionsmodells erschweren, weshalb eine Vereinfachung auf nur eine Abbildungsrichtung erfolgt, von der Benutzer- auf die Systemsicht. Der Entwickler spezifiziert, wie sich Eigenschaften seines Bausteins auf die Elemente der Systemsicht auswirken. Bei der Instanzierung eines Bausteintyps und jeder Änderung der Eigenschaften der Instanz werden diese Spezifkationen für die jeweiligen Werte der Eigenschaften angewandt. So ist sichergestellt, dass die interne Darstellung der Sicht des Nutzers entspricht.

4.3.10.3 Spezifikation der Abbildung – Mutatorkonzept

Eine Änderung von Parametern kann nahezu jeden Aspekt der Systemsicht betreffen. Trotz der großen Unterschiede lässt sich ein einheitlicher Aufbau der Abbildungsspezifikation ausmachen:

Eine Abbildungsspezifikation bezieht sich auf einen einzelnen Bausteintyp M.

Eine Abbildungsspezifikation nutzt eine Menge von Eigenschaften P des Bausteins als Eingabe, die zur Beschreibung der Auswirkungen auf die Systemsicht genutzt werden. Die Abbildung wirkt auf die Attribute eines oder mehrere Elemente der Systemsicht, die als Menge der Ziele oder *Targets* T bezeichnet werden sollen.

Eine Abbildungsvorschrift ω ist eine Funktion der Form $\vec{p} \times \vec{t} \rightarrow \vec{t'}$, wobei $\vec{p}$ mit $p_i \in P$ die Menge der relevanten Eigenschaften, $\vec{t}$ mit $t_i \in T$ die Menge der Ziele und $\vec{t'}$ mit $t'_i \in T$ die Menge der nicht notwendigerweise veränderten Ziele nach der Anwendung der Abbildungsvorschrift ist.

Der beeinflusste Bausteintyp M, die Menge der Eigenschaften P, der Ziele T und die Abbildungsvorschrift ω bilden zusammen einen *Mutator*, der als 4-Tupel der Form

$$\{M, P, T, \omega\}$$

beschrieben werden kann.

Ein Mutator wird aktiviert, wenn sich eine oder mehrere der für ihn relevanten Eigenschaften ändern.

4.3.10.4 Bedingte Mutatoren

Während die meisten Mutatoren bei jeder Änderung der für sie relevanten Eigenschaften aktiviert werden, kann es für bestimmte Anwendungszwecke sinnvoll sein, die Ausführung der von einem Mutator beschriebenen Abbildung an Bedingungen auf seinen Eigenschaften zu koppeln. Die Formulierung der Bedingungen erfolgt analog zu der in Kapitel 4.3.9 geschilderten Definition von Integritätsbedingungen mit Hilfe von algebraischen Ausdrücken. Zur Formulierung dieser Bedingungen können weitere Eigenschaften in die Menge P der Mutatordefinition aufgenommen werden. Diese müssen nicht für die Beschreibung der Auswirkungen genutzt werden. Eine Bedingung χ ist eine Funktion der Form $\vec{p} \rightarrow \{true, false\}$ mit $p_i \in P$. Ein bedingter Mutator ist ein 5-Tupel der Form

$$\{M, P, T, \chi, \omega\}.$$

Um das Erstellen einer Abbildungsspezifikation und die Implementierung der Mutatoren zu vereinfachen, werden statt eines generischen Mutators verschiedene Klassen von Mutatoren unterschieden, die sich durch unterschiedliches Verhalten und unterschiedliche mögliche Ziele auszeichnen.

4.3.10.5 Klassen von Mutatoren

Eine Unterteilung der Mutatoren erfolgt auf Basis der zulässigen Klasse(n) von Zielobjekten der Systemsicht. Innerhalb einer Zielobjektklasse kann weiter nach den vom Mutator beeinflussten Eigenschaften des Zielobjekts und seiner Funktion unterschieden werden.

Zielobjekt: Geometrie

Die Änderung der Geometrie eines Bausteins wurde als einer der wichtigsten möglichen Auswirkung von Eigenschaftsänderungen mehrfach als Beispiel aufgeführt. Mutatoren, die derartige Änderungen beschreiben, haben also die Basiselemente der Geometrie, die Solids zum Ziel. Sie können je nach gewählter Geometriedarstellung (vergleiche Kapitel 4.3.3) verschiedene Eigenschaften der Solids modifizieren. Im Folgenden sollen am Beispiel einer Boundary Representation mögliche Solid-Mutatoren vorgestellt werden.

Position und Orientierung

Die Position eines Solids innerhalb des lokalen Koordinatensystems eines Bausteins wird wie in Kapitel 4.3.3.1 geschildert durch Angabe seiner Koordinaten relativ zum lokalen Koordinatensystem seines Bausteins beschrieben. Ein Mutator, der die Position eines Solids verändert, beschreibt den Wert einer oder mehrerer Koordinaten als Term, in dem numerische Eigenschaften als Variablen eingesetzt werden können.

Analog erfolgt die Änderung der Orientierung, indem beispielsweise die Werte der Darstellung als „Achse und Winkel“ beschrieben werden.

Topologie

Auch die Topologie eines Solids kann Ziel von Änderungen werden. Zur Beschreibung kann auf die in Kapitel 4.3.3.3 erläuterten Euler-Operatoren zurückgegriffen werden. Ein solcher Mutator erfordert die Angabe der zu erstellenden (*make*) und die der zu eliminierenden (*kill*) Elemente. Da die zu erstellenden Topologieelemente nicht immer eindeutig sind, sind weitere Informationen erforderlich. So muss beispielsweise für eine mekl-Operation (make edge, kill loop) angegeben werden, welche beiden Eckpunkte die neue Kante verbindet.

Form

Auch ohne Modifikation der Topologie kann sich die Form eines Bausteins durch Änderung von Position und Lage von Flächen und Kanten ändern. Diese Änderungen können direkt durch Angabe von Rotation und Translation angegeben werden, lassen sich jedoch alle auf die Änderung der Position von Eckpunkten abbilden. Dazu definieren Terme den Wert einer oder mehrerer Koordinaten der betroffenen Eckpunkte.

Aktivierung

Solids sind fester Bestandteil einer Bausteintypdefinition. Das Erscheinen eines Solids im Experimentaufbau und seine Wirkung auf das Ergebnis einer Versuchsdurchführung kann jedoch vom Erfülltsein von Bedingungen abhängig gemacht werden. Dies kann durch Spezifikation eines bedingten Mutators erfolgen. Sind die Bedingungen des Mutators erfüllt, wer-

den seine Ziel-Solids aktiviert, sie erscheinen in der graphischen Darstellung und wirken sich auf das Experiment aus. Ist die Bedingung nicht erfüllt, werden sie deaktiviert.

Zielobjekt: Funktionales Modell

Die Veränderung von Eigenschaften eines Bausteins kann auch statische und dynamische Merkmale des funktionalen Modells eines Bausteins sowie Messzonen beeinflussen.

Material

Neben den geometrischen Eigenschaften zeichnen sich Solids und einzelne Flächen durch die Materialbeschaffenheit aus. Das Material eines Solids oder einzelner seiner Flächen kann als Parameter in der Nutzersicht repräsentiert werden. Wird dieser Parameter geändert, muss sich auch das Material ändern. Ein Mutator muss dazu lediglich die betroffenen Flächen auflisten, deren Material-Eigenschaft dann gleich dem Parameter der Nutzersicht gesetzt wird.

Aktoren

Wie schon bei der Beschreibung der Aktoren in Kapitel 4.3.5.2 geschildert kann die Parametrisierung eines Aktors durch Änderungen in der Nutzersicht beeinflusst werden. Dazu werden die Werte einzelner oder aller seiner Parameter als Term definiert.

Messzonen

Die Parameter der in Kapitel 4.3.5.3 beschriebenen Messzonen sind schon während des Experimentaufbaus vom Benutzer beeinflussbar. Neben einem direkten Zugriff auf die Parameter der Messzonen eines Bausteins kann dieser auch indirekt über Bausteineigenschaften erfolgen.

Zielobjekt: Kompositionsmodell

Bei der Definition komplexer Bausteintypen aus einfacheren Komponenten sind diese nicht mehr direkt durch den Benutzer zu beeinflussen. Stattdessen können die Parameter der Komponenten in Abhängigkeit der Parameter des komplexen Bausteins definiert werden. So ändert sich beispielsweise die Länge einer Komponente in einem festen Zusammenhang mit der Länge des Gesamtbausteins.

Position und Orientierung

Position und Orientierung von Komponenten eines komplexen Bausteins können wie bei Solids basierend auf Eigenschaften variiert werden. Die Formulierung der Abbildungsvorschrift erfolgt wie dort geschildert.

Parameter

Da die Komponenten eines komplexen Bausteins für den Benutzer nicht mehr direkt zugänglich sind, müssen deren Parameter vom Ersteller des übergeordneten Bausteins mit Werten versehen werden. Diese Werte können fix sein, oder in Abhängigkeit von den Eigenschaften des komplexen Bausteins definiert werden. Für numerische Parameter erfolgt die Spezifikation durch Formulierung eines algebraischen Ausdrucks mit Bezug auf die Eigenschaften des

übergeordneten Bausteins. Für komplexe Parameter wird im Allgemeinen eine direkte Zuweisung des Parameterwerts des übergeordneten Bausteins zu dem der Komponente erfolgen.

Aktivierung

Wie Solids können auch Komponenten eines Bausteins aktiv oder inaktiv sein. Die Definition erfolgt in analoger Weise.

Zielobjekt: Interaktionsmodell

Wird das Interaktionsmodell durch Modellierung der Konnektoren als Bausteine realisiert, werden die meisten seiner Anforderungen an das Abstraktionsmodell mit den zuvor beschriebenen Mechanismen abgedeckt. Lediglich die Änderung der Position der Verbindungspunkte und die Orientierung ihrer Achsen- und Orientierungsvektoren muss ergänzend beschrieben sein. Auch hier kann auf die einzelnen Koordinaten der Punkte beziehungsweise die der Vektoren im Mutator Bezug genommen und ihr Wert als Term definiert werden.

4.3.10.6 Ausführungsreihenfolge von Mutatoren

Die Beschreibung der Abstraktion von Nutzer- auf Systemsicht durch eine Abbildung in nur einer Richtung vermeidet Probleme, die durch zyklische Änderungen entstehen können. Kritisch bleibt jedoch die Ausführungsreihenfolge der Mutatoren. Beziehen sich zwei oder mehr Mutatoren auf das gleiche Zielobjekt, kann je nach Reihenfolge ein unterschiedliches Ergebnis zustande kommen. Da Mutatoren nur innerhalb eines Bausteintyps wirken, könnte durch Angabe einer Reihenfolge eine bestimmte Ausführungsfolge und somit ein deterministisches Verhalten sichergestellt werden.

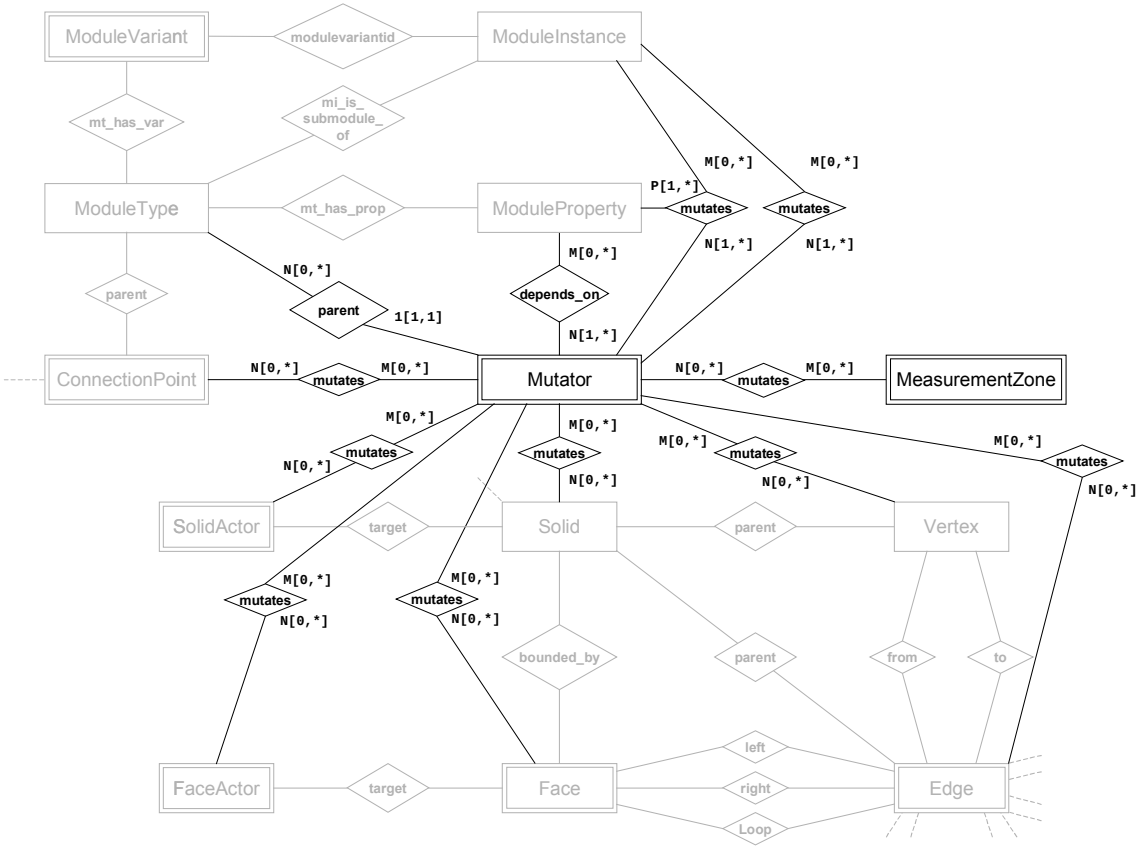
Das Problem kann jedoch auch hingenommen werden, da mehrere Mutatoren des gleichen Typs mit Wirkung auf das gleiche Zielobjekt durch einen einzelnen Mutator ersetzt werden können, der alle ihre Auswirkungen beinhaltet.

4.3.10.7 Informationsmodell für Mutatoren

Bei der Entwicklung einer Ergänzung des Informationsmodells um Mutatoren fällt insbesondere die enorme Komplexität des so entstehenden Teilschemas in Form von zahlreichen Relationships zwischen den beteiligten Entities auf. Wie Abbildung 36 zeigt, stehen Mutator-Entities nicht nur in Beziehung zu ihrem Bausteintyp und dessen Eigenschaften, sondern können auch Beziehungen zu nahezu jedem Element der Systemsicht unterhalten, das ein Zielobjekt eines Mutators sein kann. Da es sich hierbei um n:m-Beziehungen handelt, werden sehr viele Relationen benötigt. Alternativ ließen sich die verschiedenen Mutator-Typen als eigenständige Entities abbilden, die Subtypen einer abstrakten Mutator-Entity sind. In jeden Fall wird deutlich, dass eine relationale Abbildung des Abstraktionsmodells wenig geeignet wäre. Das Retrieval eines Mutators wäre sehr aufwendig, ohne dass die so detailliert modellierten Strukturen in irgendeiner Weise gewinnbringend für Suchanfragen genutzt werden. Daher bietet es sich an, Mutatoren in Form eines large objects und somit unstruktu-

riert abzulegen. Nachteil hierbei ist jedoch, dass die Integrität der Beziehungen zwischen Mutator und relational abgelegten Elementen des Bausteinmodells nicht mehr vom Datenbanksystem garantiert werden kann. Dies kann jedoch toleriert werden, wenn man davon ausgeht, dass Bausteintypen zwar von mehreren Applikationen oder Komponenten des virtuellen Labors gelesen werden, ihre Erstellung jedoch mit Hilfe eines einzigen Tools erfolgt, das dann die Integrität sicherstellt. Eine mögliche Alternative wird im Kapitel 5.1.1.3 bei der Vorstellung des Prototypen diskutiert.

Abbildung 36 Informationsmodell für Mutatoren



4.3.11 Globales Konsistenzmodell

Zusicherungen bezüglich der Gültigkeit eines Experimentaufbaus, die sowohl bei der Definition der Systemsicht als auch bei der Verwendung eines Bausteins Anwendung finden, sind unabhängig von einzelnen Bausteintypen und müssen global erfüllt sein. Eine Beschreibung dieser Konsistenzbedingungen in der Datenhaltungsschicht ist nicht erforderlich, wenn sichergestellt wird, dass die Überprüfung in einer der höheren Schichten des virtuellen Labors erfolgt.

Globale Konsistenzbedingungen können vom verwendeten Simulationsmodell für die Versuchsdurchführung abhängig sein. Ein Überprüfung erfolgt dann bei der in Anforderung D1 beschriebenen Validierung vor der Versuchsdurchführung durch die Simulationskomponente.

Die für global zuzusichernde Konsistenzbedingungen durchzuführenden Überlegungen sollen am Beispiel des Verbots von Überschneidungen zwischen Solids vorgestellt werden:

Die Solids innerhalb eines Bausteins beziehungsweise die Solids verschiedener Bausteine innerhalb eines komplexen Bausteins oder eines Experiments dürfen sich nicht räumlich überlappen. Eine detaillierte Vorstellung von Verfahren zur Schnittbestimmung zwischen Körpern an dieser Stelle würde den Rahmen dieser Arbeit verlassen. Die Kapitel 3 und 6 in [Ho89] geben einen Einblick in die Komplexität derartiger Algorithmen.

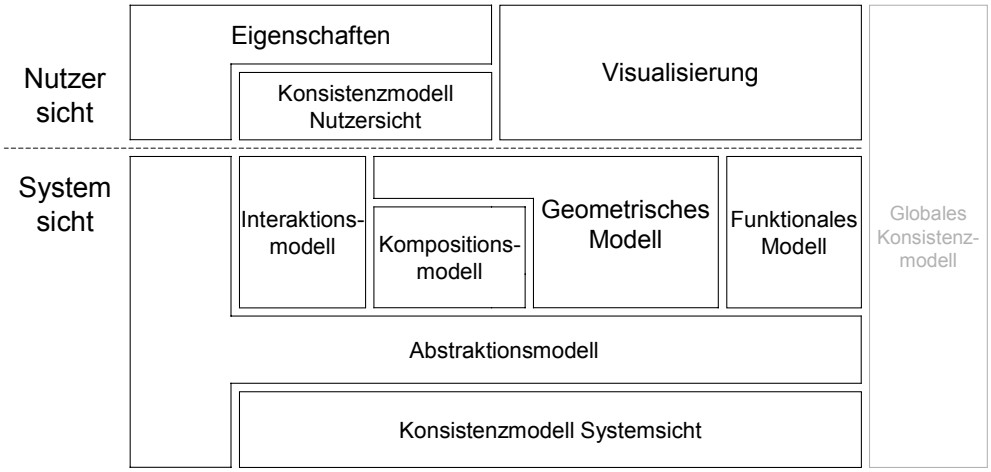
Eine Überprüfung dieser Integritätsbedingung innerhalb des Datenbanksystems würde die Auswertung komplexer Objektstrukturen erfordern. So müssen die Positionsinformationen der Solids zunächst auf das Koordinatensystem des Bausteins abgebildet werden. Anschließend muss der Solid aus der verwendeten Darstellung rekonstruiert und eine Schnittoperation durchgeführt werden. Ist die Position oder Größe eines Solids zudem über Mutatoren von Bausteineigenschaften abhängig, müssen auch diese ausgewertet werden. Derartige Operationen sind äußerst komplex und überschreiten die Mächtigkeit der in relationalen Datenbanksystemen verfügbaren Ausdrucksformen für Integritätsbedingungen. Zudem muss eine vergleichbare Überprüfung auf Überschneidungen auch während der Bearbeitung eines im Objektpuffer der Anwendung befindlichen Experiments erfolgen. Wegen des hohen Aufwands, der beschränkten Ausdrucksmöglichkeiten in Datenbanksystemen und der sowieso notwendigen Prüfung in der Anwendung erscheint eine Prüfung im Datenbanksystem daher nicht sinnvoll. Stattdessen bieten viele Bibliotheken zur Darstellung und Arbeit mit dreidimensionalen Objekten robuste Funktionen für die Prüfung auf Überschneidungen an, auf die hier zurückgegriffen werden kann.

4.3.12 Fazit - Datenmodell für Experimentbausteine

Aufgrund der vielfältigen Informationen, die das virtuelle Labor zu Beschreibung aller relevanten Aspekte von Experimentbausteinen verwalten muss, weist das vorgestellte Informationsmodell einen vergleichsweise hohen Komplexitätsgrad auf. Aus der großen Zahl von Entities und ihren Beziehungen lässt sich erkennen, dass eine rein relationale Abbildung dieses Modells schnell zu Problemen bei der Rekonstruktion der Informationen sowie der Leistung führen wird. Daher sollte bei der Abbildung für einzelne Komponenten des Modells geprüft werden, ob eine Speicherung in aus DBMS-Sicht unstrukturierter Weise die Verarbeitung erleichtern und zugleich die Leistungsfähigkeit des Systems verbessern kann. Bei den Elementen, bei denen eine solche Umsetzung für sinnvoll erachtet wurde, ist dies im Text erwähnt.

Anhand von Abbildung 37 soll ein abschließender Überblick über das Bausteinmodell und seine Komponenten gegeben werden:

Abbildung 37 Übersicht Bausteinmodell



Zentraler Bestandteil der Systemsicht bildet das geometrische Modell, zu dessen Elementen die Komponenten Funktionales Modell, Interaktions- und Kompositionsmodell in engem Bezug stehen. Die Visualisierung basiert auf geometrischem und funktionalem Modell. Die Werte von Eigenschaften werden durch das Konsistenzmodell der Nutzersicht kontrolliert und wirken sich über das Abstraktionsmodell auf alle Bestandteile der Systemsicht aus. Das Konsistenzmodell der Systemsicht stellt die Integrität von Bausteintypen sicher, das globale Konsistenzmodell sichert allgemeingültige Invarianten des Bausteinmodells.

4.4 Datenmodell für Experimentdaten

Nach der Speicherung von Experimentbausteinen und kompletten Experimenten hat die Datenhaltungsschicht des virtuellen Labors als zweite wichtige Aufgabe die Speicherung der bei der Experimentdurchführung angefallenen Daten zu leisten und gemäß Anforderung A1 dem Benutzer zur Verfügung zu stellen. Entscheidend ist dazu nicht nur der effiziente sequentielle Zugriff, wie er für Visualisierung und das Browsing erforderlich ist, sondern auch die Unterstützung von Suchanfragen auf den Daten. Neben den reinen Messwerten müssen zusätzlich die Durchführung beschreibende Metadaten gesichert werden. Dazu gehören das gewählte Simulationsmodell und dessen Parameter, die Einstellungen der Messzonen des Experiments und die Werte von Betriebsparametern der Bausteininstanzen.

4.4.1 Experimentmetadaten

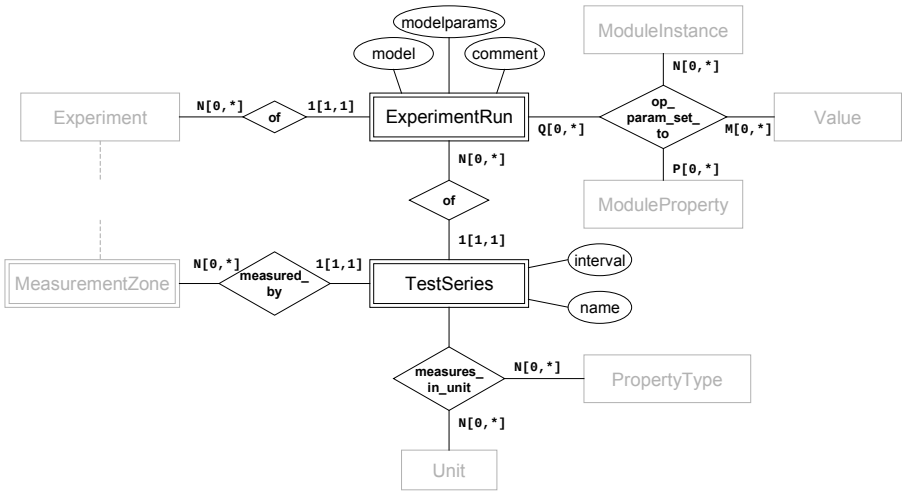
Abbildung 38 zeigt ein Informationsmodell für Experimentmetadaten. Das schwache Entity *ExperimentRun* repräsentiert globale Informationen über eine Experimentdurchführung. Dazu gehören Kommentare des Experimentators und das gewählten Simulationsmodell. Gemäß Anforderung D2.2.1 kann das Simulationsmodell eine Parametrisierung erfordern. Zahl und Typ dieser Parameter sind für jedes Modell fest. Eine flexible Modellierung wie für Bausteineigenschaften ist hier also nicht erforderlich. So kann ähnlich wie bei den Aktoren für jedes Modell ein eigenständiger benutzerdefinierter Typ mit allen Parametern des Modells

unter einem gemeinsamen Supertyp ModelParameters erzeugt werden. Im ERM ist ein Attribut *modelparams* von diesem Typ vorgesehen.

Parameter einer Messzone sind für alle von dieser in Form einer Messreihe ermittelten Messwerte konstant und werden als Attribute der jeder Messzone zugeordneten Entity *TestSeries* gesichert. Interval erlaubt die Modifikation der Messhäufigkeit, Name ein Umbenennen der Ergebnisse einer Messzone. Die Beziehung measures_in_unit erlaubt eine Veränderung der gemessenen physikalischen Größe beziehungsweise der Einheit, in der die Werte gesichert werden.

Die Betriebsparameter aller Bausteininstanzen des Experimentaufbaus, die mit Hilfe des operational-Flags von Bausteinparametern unterschieden werden, müssen ebenfalls als Teil der Metadaten festgehalten werden. Hierzu bietet sich eine Modellierung ähnlich der für die Speicherung von Eigenschaftswerten im Experimentaufbau vorgestellten an. Die Relationship *op_param_set_to* weist den gegenüber dem Experimentaufbau modifizierten Betriebsparametern (ModuleProperty) jeder Bausteininstanz (ModuleInstance) einen Wert (Value) zu, der den dort gesetzten Wert bei der Durchführung ersetzt, ohne dabei jedoch den Experimentaufbau selbst zu ändern. Für die Modellierung der Value-Entity kann auf eine der in Kapitel 3.3.2.2 vorgestellten Methoden zurückgegriffen werden.

Abbildung 38 Informationsmodell für Experimentmetadaten



4.4.2 Experimentdaten

Für jede Messzone des Experimentaufbaus werden während der Experimentdurchführung in kontinuierlichen Abständen Daten ermittelt und gespeichert, die Daten der Messzone bilden eine *Messreihe*. Im Informationsmodell für Experimentdaten (Abbildung 39) ist sie durch die schon für die Metadaten einer Messzone eingeführte Entity TestSeries repräsentiert.

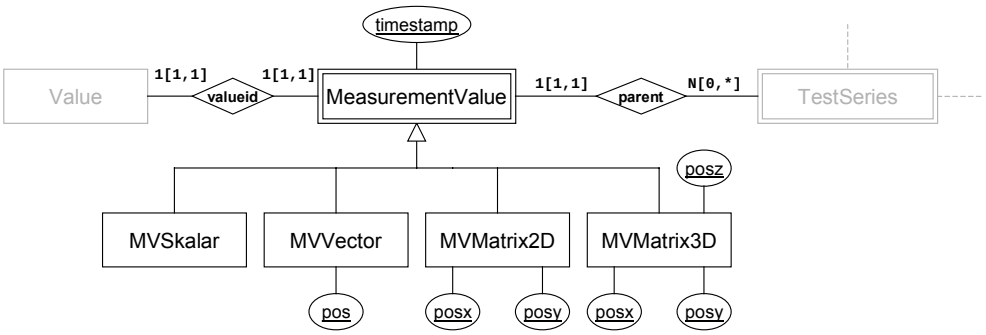
Zu jedem Messzeitpunkt fallen je nach Dimension der Erfassung ein einzelner Wert, ein Vektor oder eine zwei- oder dreidimensionale Matrix von Werten an. Da die Anzahl von Messwerten in jeder Dimension einer Messzone nur abhängig von ihren Abmessungen und

der Dichte der Messpunkte und daher nicht nach oben beschränkt ist, erfordert ein Modell für die Experimentdaten einen entsprechenden Mehraufwand: Für jeden Messpunkt in einer höherdimensionalen Messzone muss seine Position innerhalb des durch die Messpunktdichte definierten Rasters eigenständig verwaltet werden. Dies könnte durch eine implizite Durchnummerierung aller Messpunkte einer Messzone geschehen und wäre in vielen Fällen ausreichend. Durch diese künstliche Reduktion der Messwerte höherdimensionaler Messzonen auf nur eine Dimension werden jedoch Anfragen auf den Experimentdaten erschwert, die eine Analyse der räumlichen Veränderung der Werte zum Ziel haben. Daher sollten die Informationen über die Position eines Messwerts innerhalb der Messzone zusätzlich gespeichert werden.

Eine Superklasse für alle Messwerte, das schwache Entity *MeasurementValue*, speichert den Zeitpunkt der Ermittlung des Messwerts (Attribut *timestamp*). Die Subklassen fügen die für die Positionierung innerhalb der Messzone erforderlichen Attribute hinzu. So erhält MVMatrix3D Angaben zu den drei Koordinaten des Messpunktes. Beschränkt man sich wie in Kapitel 4.3.5.3 beschrieben auf eine gleichmäßige Dichte der Messpunkte in allen Dimensionen, genügen für die Positionsangabe ganze Zahlen.

Da die Werte selbst ähnlich wie die Eigenschaften von Experimentbausteinen von beliebigen, im Allgemeinen jedoch numerischen Typen sein können, würde eine direkte Speicherung des Messwerts in der MeasurementValue-Entity bedeuten, dass für jeden möglichen Messwerttyp vier eigene Entities und damit Relationen nötig wären. Deshalb werden auch hier die Werte in einer eigenen Value-Entity gespeichert und nur von MeasurementValue aus referenziert. Die Value-Entity selbst kann in einem der in Kapitel 3.3.2.2 beschriebenen Verfahren modelliert sein.

Abbildung 39 Informationsmodell für Experimentdaten



4.4.3 Abhängigkeit zwischen Experimentdurchführung und Experimentaufbau

Da sich Messwerte und Experimentmetadaten auf Elemente des Experimentaufbaus beziehen, könnte dessen nachträgliche Modifikation eine Falschinterpretation der Ergebnisse bewirken. Um dieses Problem zu vermeiden, darf ein Experimentaufbau nach Durchführung einer Messung nicht mehr modifiziert werden. Sind Änderungen erforderlich, muss eine neue Version aus dem Experimentaufbau abgeleitet werden. Änderungen von Parametern

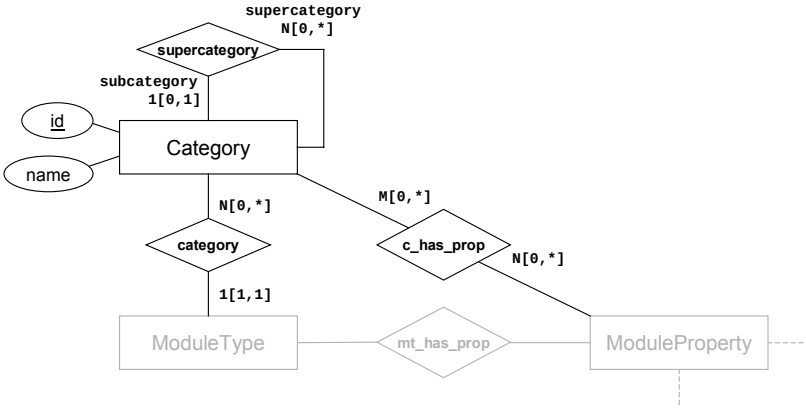
sind davon nicht betroffen, da sie als Experimentmetadaten für jede Durchführung getrennt vom Experimentaufbau erfasst werden.

4.5 Organisation der Bausteinbibliothek

Neben der Speicherung aller Komponenten des Bausteinmodells muss das Informationsmodell des virtuellen Labors im Bereich Verfahrenstechnik effiziente Suchfunktionen in der Menge von Bausteinen der Bausteinbibliothek ermöglichen. Anforderung P2.3 und Untergeordnete beschreiben eine Organisation der Bausteinbibliothek in eine Kategorienhierarchie. Jeder Baustein ist genau einer Kategorie zugeordnet. So wird dem Benutzer ein navigierender Zugriff auf die Bausteine ähnlich einem Dateisystem geboten. Kategorien erhalten zusätzlich wie Bausteintypen Eigenschaften. Wird sichergestellt, dass jede Kategorie mindestens alle Eigenschaften ihrer übergeordneten Kategorie hat und zusätzlich jeder Baustein mindestens alle Eigenschaften seiner Kategorie aufweist, lässt sich auf Grundlage dieser Eigenschaften eine im Vergleich zum einfachen Browsen der Hierarchie erweiterte Suchfunktion realisieren. Dazu kann eine Kategorie gewählt und auf ihren Eigenschaften ein Suchprädikat formuliert werden, das rekursiv auf alle Bausteine und Kategorien angewendet wird, die unterhalb der selektierten Kategorie liegen.

Das Informationsmodell zu Realisierung dieser hierarchischen Struktur zeigt Abbildung 40. Eine Kategorie verfügt über eine Bezeichnung und kann eine beliebige Anzahl von untergeordneten Kategorien haben, selbst jedoch in maximal einer Kategorie enthalten sein. Jeder Bausteintyp wird genau einer Kategorie zugeordnet. Eine Kategorie kann ähnlich wie ein Bausteintyp über eine beliebige Anzahl von Eigenschaften verfügen. Diesen wird jedoch kein Wert zugewiesen, sie beschreiben lediglich die gemeinsamen Eigenschaften aller dieser Kategorie untergeordneten Kategorien und Bausteintypen.

Abbildung 40 Organisation der Bibliothek in Kategorien



Das Informationsmodell erzwingt die in den Anforderung P2.3.2.1 und P2.3.2.2 geforderte Vererbung innerhalb der Kategorienhierarchie jedoch nicht. Diese Integritätsbedingung muss außerhalb des Schemas zum Beispiel mit Mitteln des Datenbanksystems geprüft werden. Bei Verwendung objektrelationaler Datenbanksysteme nach SQL:1999 bieten sich dazu ECA-Regeln oder Assertions an.

4.6 Schnittstellen der Datenmodellschicht

Alle Komponenten, welche die beiden oberen Schichten des virtuellen Labors bilden, müssen für den Zugriff auf das Repository die Datenmodellschicht nutzen. Unabhängig von der technischen Realisierung des Zugriffs als lokaler Methodenaufruf, Remote Method Invocation oder Web Service muss folglich eine klar definierte Schnittstelle für die Datenmodellschicht definiert werden. Die Funktionen der Schnittstelle lassen sich in drei Bereiche unterteilen: Die Experimentverwaltung bietet Funktionen zum Erzeugen und Löschen von Experimenten sowie für Check-In und Check-Out. Die Bibliotheksverwaltung erlaubt das Durchsuchen und in Grenzen ein Umgestalten der Bausteinbibliothek, die Experimentdatenverwaltung bietet Methoden zum Speichern der bei einer Experimentdurchführung anfallenden Daten und zum Lesen der Daten durch Analysekomponenten. Im Folgenden soll die Schnittstelle in sprachunabhängiger Weise durch Angabe der Signaturen der wichtigsten Methoden beschrieben werden. Zur Beschreibung der Signaturen wurde die allgemein bekannte UML-Schreibweise verwendet:

```
<methodName>(<param1>:<type1>, <param2>:<type2>, ...): <returntype>
```

Um vom verwendeten Programmiermodell unabhängig zu bleiben, sind die Methoden kontextfrei, das heißt ihr Verhalten ist nur von den übergebenen Parametern (und natürlich den Daten der Datenmodellschicht) abhängig und nicht direkt von vorherigen anderen Methodenaufrufen.

Dadurch wird die Schnittstelle naturgemäß etwas aufwendiger, da für einen Methodenaufruf mehr Informationen benötigt werden. Beispielsweise sind die Ergebnisse fast aller Methoden von den Rechten des aufrufenden Benutzers abhängig, sodass die Benutzerdaten zur Authentifizierung als User-Objekt mitübergeben werden müssen. Alternativ kann die Datenmodellschicht auch Sitzungen für die angemeldeten Benutzer verwalten. Genügen die Rechte des Benutzers nicht für die jeweilige Operation, wird dies durch eine Ausnahme signalisiert.

Die Parameter und Rückgabewerte der Methoden sind wegen der komplexen Natur der an der Schnittstelle bewegten Daten oft keine primitiven Typen, sondern Anwendungsobjekte, bei denen es sich um verkettete Objektstrukturen handeln kann. Über ihren genauen Aufbau soll an dieser Stelle keine Aussage gemacht werden. Vielmehr soll die Schnittstellenbeschreibung eine Übersicht über die von den Anwendungen benötigte Funktionalität geben. Kapitel 5 (Prototyp) beschreibt eine mögliche Implementierung von Teilen der Schnittstelle und zeigt beispielhaft den Aufbau der Anwendungsobjekte.

4.6.1 Experimentverwaltung

```
listExperiments (user : User) : ExperimentMetaData[]
```

Gibt eine Liste mit Informationen über alle für den als Parameter übergebenen Benutzer zugänglichen Experimente zurück.

```
checkOutExperiment (experiment : ExperimentMetaData, user: User)
                   : Experiment
```

Liefert das vom ExperimentMetaData-Objekt repräsentierte Experiment zurück und markiert es in der Datenbank als checked-out.

```
checkInExperiment (experiment : Experiment, user : User) : void
```

Bringt das Experiment in die Datenbank ein. Bei einem neuen Experiment wird ein neuer Eintrag erstellt, ein bereits in der Datenbank vorhandenes Experiment wird überschrieben. Das Experiment bleibt als checked-out markiert.

```
checkInNewVersion (experiment : Experiment, user : User) : void
```

Bringt ein bereits vorhandenes Experiment als neue Version in die Datenbank ein. Für ein neues Experiment verhält sich diese Funktion wie checkInExperiment.

```
releaseExperiment (experiment : ExperimentMetaData) : void
```

Gibt ein als checked-out markiertes Experiment wieder frei ohne zwischenzeitliche Änderungen in die Datenbank einzubringen.

4.6.2 Bibliotheksverwaltung

Die Schnittstelle der Bibliotheksverwaltung lässt sich in rein lesende Operationen und solche zum Ändern der Bausteinbibliothek unterteilen. Weiterhin gibt es Funktionen, die einen Bausteintyp oder eine Bausteinvariante instanzieren.

Browsing

```
getRootCategories (user : User) : Category[]
```

Liefert die Wurzelkategorien der Bausteinbibliothek zurück und berücksichtigt die Zugriffsrechte des Benutzers.

```
getSubCategories (superCategory : Category, user : User)
                 : Category[]
```

Gibt eine Liste aller Unterkategorien der übergebenen Kategorie zurück und berücksichtigt dabei die Zugriffsrechte des Benutzers.

```
getModuleTypes (category : Category, user : User)
               : ModuleTypeMetaData[]
```

Liefert eine Liste mit allen Bausteintypen in der übergebenen Kategorie zurück und berücksichtigt dabei die Zugriffsrechte des Benutzers.

```
getModuleVariants (moduleType : ModuleTypeMetaData, user : User)
                  : ModuleVariantMetaData[]
```

Liefert alle Varianten des als Parameter spezifizierten Bausteintyps zurück und berücksichtigt dabei die Zugriffsrechte des Benutzers.

Editieren

```
modifyCategory (category : Category, user : User) : Category
```

Bringt die modifizierte Kategorie in die Datenbank ein.

```
modifyModuleType (moduleType : ModuleTypeMetaData, user : User)
                 : ModuleTypeMetaData
```

Bringt den modifizierten Bausteintyp in die Datenbank ein. Mögliche Änderungen sind der Name und die Rechte des Bausteintyps.

```
modifyModuleVariant (moduleVariant : ModuleVariantMetaData,
                     user : User) : ModuleVariantMetaData
```

Bringt die modifizierte Bausteinvariante in die Datenbank ein. Mögliche Änderungen betreffen den Namen und die Werte von Eigenschaften.

```
moveCategory (category : Category, newSuperCategory : Category,
             user : User) : void
```

Verschiebt die angegebene Kategorie in die Kategorie newSuperCategory. Das Verschieben schlägt fehl, wenn die Kategorie nicht mindestens über die Eigenschaften von newSuperCategory verfügt.

```
moveModuleType (moduleType : ModuleTypeMetaData,
               newCategory : Category, user : User) : void
```

Verschiebt den angegeben Bausteintyp einschließlich aller Varianten in die Kategorie newCategory. Das Verschieben schlägt fehl, wenn der Bausteintyp nicht mindestens über die Eigenschaften von newCategory verfügt.

```
deleteCategory (category : Category, force : boolean, user : User)
               : void
```

Löscht die angegebene Kategorie. Enthält diese noch Unterelemente, also Bausteintypen oder Subkategorien, wird eine Ausnahme generiert. Durch Setzen des Parameters *force* auf true kann das Löschen erzwungen werden. Sind jedoch Bausteintypen innerhalb der Kategorie in Benutzung wird das Löschen auch dann verhindert.

```
deleteModuleType (moduleType : ModuleTypeMetaData, user : User)
                 : void
```

Löscht den angegebenen Bausteintyp. Wird dieser in einem Experiment oder in einem komplexen Baustein verwendet, wird ein Löschen verhindert.

```
deleteModuleVariant (moduleVariant : ModuleVariantMetaData,
                     user : User) : void
```

Löscht die angegebene Bausteinvariante.

Instanzieren

```
createModuleInstance (moduleType : ModuleTypeMetaData, user : User)
                     : ModuleInstance
```

Erzeugt eine Instanz des angegebenen Bausteintyps. Die Anwendung muss sicherstellen, dass alle verpflichtenden Parameter in der Instanz gesetzt werden.

```
createModuleInstance (moduleVariant : ModuleVariantMetaData,
                     user : User) : ModuleInstance
```

Erzeugt eine Instanz basierend auf der angegebenen Bausteinvariante.

4.6.3 Experimentdatenverwaltung

```
createNewExperimentRun(experimentId : int,
                       configuration : ExperimentRunConfiguration,
                       user : User) : int
```

Erzeugt eine neue Experimentdurchführung für das Experiment mit der angegebenen Id. Der Rückgabewert enthält die ID, welcher der Experimentdurchführung zugewiesen wurde.

```
storeMeasurementValue(experimentId : int, experimentRunId : int,
                      moduleInstanceId : int, zoneId: int,
                      timeCode : long, value : Value, user : User)
                      : void
```

Speichert den als Parameter *value* angegebene und zum Zeitpunkt *timeCode* ermittelten Messwert der Messzone mit *zoneId* des Bausteins mit *moduleInstanceId* für das Experiment mit *experimentId* und die Experimentdurchführung mit *experimentRunId*. Die Klasse Value steht dabei für eine der möglichen Arten von Werten, Skalar oder Vektor, jeweils als einzelner Wert, als Vektor, zwei- oder dreidimensionale Matrix.

```
storeMultipleValues(experimentId : int, experimentRunId : int,
                    moduleInstanceId : int, zoneId: int,
                    startTimeCode: long, interval: long,
                    values : Value[], user : User) : void
```

Um den Aufwand für die Speicherung einer großen Menge von Messwerten zu reduzieren, erlaubt die Methode *storeMultipleValues* das Speichern von zeitlich aufeinanderfolgenden Messwerten. *startTimeCode* gibt den Zeitpunkt der Erfassung des ersten Messwertes an, *interval* beschreibt den konstanten Abstand zwischen den Messwerten. Der Parameter values enthält die Folge von Messwerten, die übrigen Parameter entsprechen denen der storeMeasurementValue-Methode.

```
retrieveMeasurementValue(experimentId : int, experimentRunId : int,
                         moduleInstanceId : int, zoneId: int,
                        timeCode : long, user : User) : Value
```

Liest einen Messwert aus der Datenbank. Die Parametrisierung entspricht der von storeMeasurementValue.

```
retrieveMultipleValues (experimentId : int, experimentRunId : int,

                        moduleInstanceId : int, zoneId: int,
                        startTimeCode: long, endTimeCode: long,
                        user : User) : Value[]
```

Ebenfalls zur Reduzierung der Anzahl notwendiger Methodenaufrufe erlaubt *retrieveMultipleValues* das Auslesen aller von einer Messzone stammenden Werte im von *startTimeCode* und *endTimeCode* angegebenen Interval.

4.6.4 Allgemeine Funktionen

```
validateUser (userName : String, password : String) : boolean
```

Erlaubt die Überprüfung von Benutzerdaten.

4.6.5 Fazit - Schnittstelle

Die exemplarisch aufgeführten Schnittstellen erlauben den Zugriff auf und die Manipulation aller wichtigen Daten im Repository des virtuellen Labors. Sie berücksichtigen die erforderlichen Rechte des eingeloggten Benutzers. Die vorgestellten Methodensignaturen sind als Vorschlag zu verstehen und könnten leicht durch funktionell ähnliche ersetzt werden. So könnte retrieveMultipleValues statt Start- und Endzeitpunkt eines Intervalls auch mit dem Startzeitpunkt und der gewünschten Maximalzahl von Werten parametrisiert werden.

Kapitel 5 Prototyp

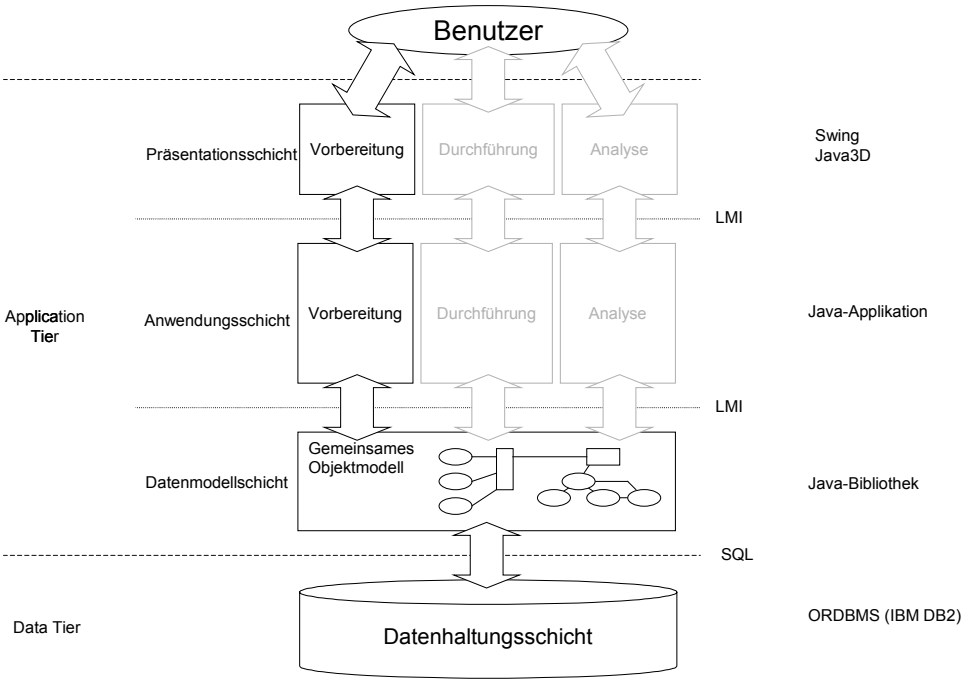
Nach der Ermittlung der Anforderungen an das Repository des virtuellen Labors wurden einige der in Kapitel 4 vorgestellten Technologien und Konzepte in Form einer prototypischen Implementierung realisiert, um nach ihrer theoretischen Bewertung die Leistungsfähigkeit und Tauglichkeit im praktischen Einsatz zu beweisen. Wegen des immensen Umfangs wird nur ein Ausschnitt der Konzepte umgesetzt.

In diesem Kapitel wird zunächst die realisierte Architektur des Prototypen auf den in Kapitel 4.1 entwickelten Architekturvorschlag abgebildet. Anschließend erfolgt eine kurze Vorstellung der in den einzelnen Schichten eingesetzten Technologien. Schließlich werden die wichtigsten und interessantesten Komponenten und Konzepte des Prototypen vorgestellt. Eine weitergehende Beschreibung des Prototypen würde den Rahmen dieser Arbeit überschreiten, der interessierte Leser sei dazu auf die Softwaredokumentation verwiesen.

5.1 Architektur und Technologien

Bei der Entwicklung des Prototyps wurde der Schwerpunkt auf eine Umsetzung des in Kapitel 4.3 entwickelten Informationsmodells für Experimentbausteine sowie die Erprobung einzelner vorgestellter Technologien gelegt. Der in Kapitel 4.1 beschriebene Architekturvorschlag wurde dazu zunächst in Form einer 2-Tier-Applikation umgesetzt. Abbildung 41 zeigt die Architektur des Prototypen. Die unterste Schicht (*Data Tier*) wird durch ein objektrelationales Datenbanksystem (IBM DB2 Universal Database) realisiert. Bei der Abbildung des Bausteininformationsmodells wurden verschiedene zum Teile proprietäre Features dieser Datenbank erprobt. Die Datenmodell-, Anwendungs- und Präsentationsschicht wurde im Application Tier zusammengefasst. Zum Einsatz kommt die Java 2 Standard Edition (Version 1.4.1), ergänzt um optionale Komponenten (*optional packages*). Die Datenmodellschicht wurde als eigenständige gekapselte Bibliothek realisiert, welche die Anwendungsobjekte des gemeinsamen Objektmodells in Form von JavaBeans zur Verfügung stellt. Bei Präsentations- und Anwendungsschicht wurde sich wie in den vorherigen Kapiteln auf die Phase der Experimentvorbereitung beschränkt. Beide Schichten sind in Form einer Java-Applikation realisiert. Die Benutzerschnittstelle wurde mit dem Java Swing API umgesetzt, ergänzt um Java 3D für die Darstellung des Experimentaufbaus. Die Anwendungsschicht setzt die Nutzeraktionen in entsprechende Änderungen des Objektmodells im Speicher der Anwendung um.

Abbildung 41 Architektur des Prototypen



Ausgehend von dieser Architektur wurden verschiedene Möglichkeiten untersucht, um die Anbindung von nicht in Java realisierten Komponenten an die Datenhaltungsschicht zu ermöglichen. So wurde ein Teil der Schnittstelle der Datenmodellschicht als Web Service verfügbar gemacht. Ein anderer Ansatz überprüft die Möglichkeiten des XML Extenders der DB2, um eine direkte Abbildung der relationalen Strukturen auf XML-Dokumente vorzunehmen. Auf eine Erprobung von CORBA wurde wegen der bereits in Kapitel 4.1 beschriebenen Nachteile verzichtet.

5.1.1 Datenhaltungsschicht

Als Datenhaltungsschicht wurde wie in Kapitel 4.2 vorgeschlagen ein objekt-relationales Datenbanksystem eingesetzt. Es standen sowohl IBM Informix Dynamic Server als auch die schon in Abschnitt 4.2.6.1 vorgestellte IBM DB2 Universal Database zur Verfügung. Trotz vorhandener positiver Erfahrungen mit Informix fiel die Wahl auf IBM DB2 Universal Database. Grund für diese Entscheidung war insbesondere das absehbare Ende der Weiterentwicklung von Informix. IBM plant nach der Übernahme der Datenbanksparte der Informix Corporation im Jahr 2001 mittelfristig, die beiden Datenbanksysteme zu einem Produkt zusammenzuführen.

5.1.2 Datenmodellschicht

Um maximale Flexibilität für die Erprobung verschiedener Technologien zur Verbindung von Anwendungs- und Datenhaltungsschicht zu ermöglichen, wurde die Datenmodellschicht in Form einer stark gekapselten Java-Bibliothek (Package) realisiert. Sie nutzt JDBC zum

Zugriff auf die Datenbank, rekonstruiert die dort zerlegt gespeicherten Anwendungsobjekte und macht sie den höheren Schichten mit Hilfe von Factory-Methoden zugänglich.

5.1.2.1 JDBC

Die Java Database Connectivity ist das Standard-API für den Zugriff auf objekt-relationale Datenbanksysteme. Es bietet ein standardisiertes Call-Level-Interface (CLI), das von den systemspezifischen Schnittstellen der verschiedenen Datenbanken abstrahiert. Um eine Datenbank mit JDBC nutzen zu können, muss ein Treiber für das Zieldatenbanksystem zur Verfügung stehen. Diese werden vom DBS-Hersteller oder von Drittanbietern bereitgestellt und lassen sich je nach Art der Verbindung zur Datenbank in vier Typen unterscheiden.

Die Vorgehensweise zur Nutzung von JDBC ist für alle Treiber identisch. Mit Hilfe der `DriverManager`-Klasse kann unter Angabe der in ihrem Aufbau herstellerspezifischen Datenbank-URL sowie Name und Passwort ein Verbindungsobjekt erzeugt werden. Mit diesem lassen sich `Statement`-Objekte erzeugen. Einfachen Statements wird die Anfrage oder Änderungsoperation in Form eines Strings übergeben, und anschließend wird das Statement ausgeführt. Handelt es sich um eine Anfrage, wird das Ergebnis in Form eines `ResultSet`-Objekts zugänglich gemacht, das ein Iterieren über die Ergebnismenge und den Zugriff auf einzelne Attribute der Tupel mit getter-Methoden ermöglicht. Scrollable- und Updatable-Resultsets erlauben bei einfachen Statements die freie Navigation in der Ergebnismenge und das Ändern von Tupeln mit Java-Methoden statt SQL. Um bei häufig durchgeführten Datenbankoperationen den Aufwand für das jedesmal nötige Übersetzen der Statements zu sparen, kann mit `PreparedStatement`-Objekten gearbeitet werden. Auch hier wird die Operation als String übergeben, es können jedoch Platzhalter vorgesehen werden. Das Statement kann dann vorübersetzt werden. Bei der späteren Nutzung müssen nur noch die Platzhalter durch konkrete Werte ersetzt werden. Mit einem `CallableStatement` ist die Nutzung von *Stored Procedures* und *User Defined Functions* möglich. Über diese grundlegenden Funktionen hinaus bieten JDCB-Schnittstellen Zugriff auf DB-Metadaten, Transaktionskontrolle und Steuerung der Isolationslevel.

Eine Erweiterung stellt das JDBC Optional Package dar. Ursprünglich als selbständige Erweiterung eingeführt, wurde es zuerst fester Bestandteil der Java 2 Enterprise Edition (J2EE) und ist inzwischen auch in die Standard Edition integriert. Es bietet durch JNDI-Unterstützung die Möglichkeit, Verbindungsobjekte datenbankunabhängig zu erzeugen, erlaubt Connection Pooling, also die Wiederverwendung von Verbindungsobjekten, und unterstützt verteilte Transaktionen. Zudem bringt es mit den `RowSet`-Objekten ein verbessertes ResultSet mit.

5.1.2.2 JavaBeans

JavaBeans sind definiert als abgeschlossene, wiederverwendbare Softwarekomponenten, die mit einem graphischen Entwurfstool modifiziert und kombiniert werden können. Beans verfügen über Properties, also Eigenschaften, auf die mit Methoden nach einem einfachen Benennungsschema zugegriffen werden kann (getter- und setter-Methoden). Weiterhin haben sie gewöhnliche Methoden, die sie nach außen sichtbar machen. Die Kommunikation zwi-

schen Beans erfolgt über ein Ereignismodell. Eine Komponente, die bestimmte Ereignisse verarbeiten kann, implementiert ein entsprechendes Listener-Interface (ein Subinterface von `EventListener`) und registriert sich bei Ereignisquellen (event source), die für jeden von ihnen unterstützten Listener-Typ add- und remove-Methoden anbieten. Wenn die Ereignisquelle ein Ereignis generiert, wird dies durch Aufruf einer Callback-Methode des jeweiligen Listener-Interfaces im Normalfall an alle registrierten Listener weitergeleitet (Multicast).

Eigenschaften, Methoden und Ereignisse können einfach durch Anwendung der Zugriffsmodifikatoren (*public*-Methoden) und durch Einhaltung der Namenskonventionen (*get*PropertyName und *set*PropertyName, *add*ListenerName, *remove*ListenerName) propagiert werden. Alternativ kann für ein JavaBean auch eine BeanInfo-Klasse erstellt werden, die Zugriff auf Properties, Methoden und Ereignisse erlaubt. Dadurch können auch ohne Einhaltung der Namenskonventionen die Bestandteile einer Bean deklariert werden. Verzichtet man auf eine eigene BeanInfo-Klasse, kann die Hilfsklasse Introspector (`java.beans.Introspector`) zur Laufzeit eine JavaBean analysieren und mit Kenntnis der Namenskonventionen ein BeanInfo-Objekt generieren.

JavaBeans werden intensiv in vielen Standard-APIs von Java genutzt. So sind beispielsweise alle Komponenten des Swing-Toolkits für die Entwicklung von GUIs als JavaBeans ausgeführt, das Ereignismodell wird dort für die Weitergabe von Nutzerinteraktionen an die Komponenten genutzt.

Bei der Entscheidung, die Komponenten der Datenmodellschicht als JavaBeans zur realisieren, war weniger die Nutzung von Entwurfstools zur Rekombination der Komponenten ausschlaggebend. Vielmehr können JavaBeans wesentlich flexibler in verschiedenen Umgebungen genutzt werden. Zudem gibt es viele Technologien, die entweder ausschließlich mit JavaBeans funktionieren oder so deutlich einfacher zu nutzen sind. Insbesondere verschiedener Ansätze zur Serialisierung wie der XMLEncoder arbeiten nur mit JavaBeans.

Weitere Techniken, die in der Datenmodellschicht zum Einsatz kommen, werden bei der Beschreibung der einzelnen Komponenten aufgeführt.

5.1.3 Anwendungsschicht

Da die Anwendungslogik bei der Experimentvorbereitung im Prototypen im Wesentlichen aus dem Erzeugen und Modifizieren der Anwendungsobjekte besteht, fällt die Anwendungsschicht vergleichsweise einfach aus. Sie hat lediglich die Aufgabe, die Nutzeraktionen auf entsprechende Operationen der Anwendungsobjekte umzusetzen und diese wiederum für die Anzeige an der Benutzeroberfläche aufzubereiten. Sie wird daher in den folgenden Kapiteln nicht weiter erläutert.

5.1.4 Präsentationsschicht

Da diese Arbeit in Bereich D4 des geplanten Sonderforschungsbereichs angesiedelt ist, war die Entwicklung einer Benutzeroberfläche für das virtuelle Labor im Bereich Verfahrenstechnik kein zentrales Thema dieser Arbeit. Bei der Kommunikation mit den Anwendern zur

Ermittlung der Anforderungen wurde jedoch festgestellt, dass einige Verständnisprobleme auf beiden Seiten besser beseitigt werden können, wenn die Diskussion anhand eines Beispiels geführt werden kann, das einzelne Ideen und Konzepte in einfacher Form demonstriert. Dies führte zur Entscheidung, eine rudimentäre graphische Benutzeroberfläche zu implementieren. Zugleich halfen die so gemachten Erfahrungen bei der Nutzung des Informationsmodells auch bei dessen Verfeinerung und Vervollständigung.

Neben den allgemeinen Anforderungen an eine graphische Benutzeroberfläche galt es, zusätzlich die Visualisierung von Experimentbausteinen und -aufbauten zu leisten. Da die Entwicklung eines Abstraktionsmodells für eine vereinfachende symbolische Darstellung des Experiments und seiner Komponenten im Rahmen dieser Arbeit nicht zu leisten war, wurde stattdessen einer direkten graphischen Darstellung der Bausteingeometrie der Vorzug gegeben. So entstand zusätzlich der Bedarf, dreidimensionale Objekte zu visualisieren.

5.1.4.1 Swing

Für die konventionellen Bedienelemente des Prototypen wie Elemente zum Durchsuchen der Bausteinbibliothek, zur Anzeige von Bausteindaten oder zur Manipulation, Speicherung und Verwaltung von Experimenten und Experimentbausteinen wurde die Java Bibliothek Swing eingesetzt (`javax.swing`). Sie bietet im Vergleich zur anderen für die GUI-Entwicklung geeigneten Java-Standard-Bibliothek, dem Abstract Window Toolkit (AWT), deutliche Vorteile:

Im Gegensatz zu AWT sind die Swing-Komponenten (*Window gadgets*, kurz „*Widgets*“) als leichtgewichtige (*lightweight*) Komponenten ausgeführt. Das bedeutet, dass sie vollständig in Java realisiert wurden und so ihr Aussehen und Verhalten nicht von der jeweiligen Systemplattform abhängig ist. Die sich daraus in der Anfangszeit von Swing ergebenden Geschwindigkeitsdefizite wurden in aktuellen Version der Java-Plattform weitgehend ausgeräumt. Swing bietet zudem eine größere Auswahl von Widgets als das AWT, sodass die Entwicklung leistungsfähiger GUIs deutlich vereinfacht wird. So existieren bereits Widgets zur Realisierung von Baumdarstellungen (JTree), wie sie zum Beispiel für die Bausteinbibliothek genutzt werden können, oder Tabellen zur Darstellung von strukturierten Informationen (JTable). Diese Komponenten sind als JavaBeans in weiten Bereichen an eigene Bedürfnisse anpassbar, viele Entwicklungsumgebungen bieten Tools zum komfortablen Generieren einer GUI aus Swing-Komponenten.

Während Swing-Komponenten analog zu ihren AWT-Pendants genutzt werden können, bieten sie zusätzlich Unterstützung für ein mächtiges Entwurfsmuster für graphische Benutzeroberflächen: Das Model-View-Controller Pattern (MVC) trennt die darzustellenden Daten (*model*) von der Logik zu ihrer Anzeige (*view*) und der für die Interaktion (*controller*). Bei Swing werden controller und view in einem sogenannten *delegate*-Objekt zusammengefasst.

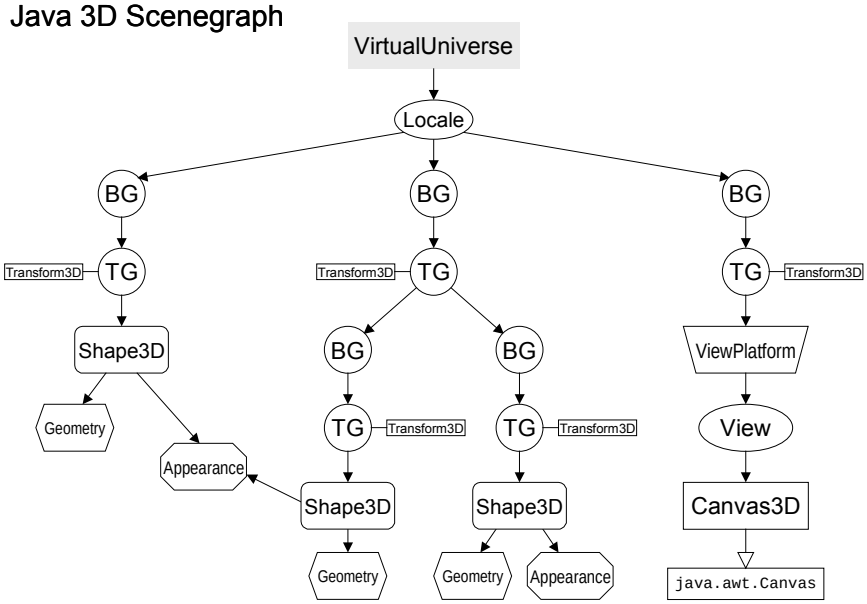
5.1.4.2 Java 3D

Das Java 3D API [J3DAPI] ist eine optionale Erweiterung (optional package) zur Java 2 Standard Edition und liegt zur Zeit in Version 1.3.1 vor. Ziel ist die Integration von dreidi-

mensionalen, statischen und bewegten Graphiken in beliebige Java-Anwendungen oder Applets. Als optional package ist die Menge der unterstützen Plattformen bisher im Vergleich zum Kern von J2SE kleiner: Von Sun direkt ist Java 3D zur Zeit lediglich für die Windows- und Solaris/SPARC-Plattformen erhältlich. Unter Windows nutzt Java 3D wahlweise die Direct3D- oder OpenGL-Schnittstelle und kann so auch von gängiger 3D-Beschleuniger-Hardware profitieren. Zusätzlich stellen IBM, HP und SGI für ihre Betriebssysteme, AIX, HP-UX beziehungsweise IRIX eigene Implementierungen der Plattform bereit. Blackdown entwickelt und pflegt eine Linux-Version. Für den Einstieg in die Entwicklung mit Java 3D eignet sich insbesondere [J3DTUT]. Bei [J3DORG] finden sich weitergehende Informationen und Unterstützung bei Problemen, auf die man bei der Realisierung fortgeschrittener Anwendungen stößt. Viele der in Kapitel 4.3.3 vorgestellten Konzepte zur Beschreibung von Positionen, Orientierungen und Geometrien finden sich in Java 3D in abgewandelter Form wieder.

Java 3D ist nicht zuletzt wegen seiner Abstraktion von einzelnen Hardware- und Betriebssystemplattformen ein High-Level-API, das sich in weiten Teilen an OpenGL orientiert. Kernkonzept ist der sogenannte *Scenegraph*, in dem alle Objekte einer 3D-Szene als Knoten (`Node`) enthalten sind und der so ihre gemeinsame Manipulation ermöglicht. Die Blätter des Scenegraph sind die sogenannten *Shapes*, also 3D-Objekte. Diese können in Gruppen zusammengefasst und transformiert werden. *Appearance*-Objekte beschreiben das Aussehen von einem oder mehreren Shapes. Dies umfasst Farbe, Textur, Beleuchtungseigenschaften und Transparenzeffekte. *Behaviour*-Objekte erlauben die Manipulation von Shapes, Appearances und anderen Teilen des Szenegraphs. So lassen sich Animationen und Benutzerinteraktion realisieren. Der Scenegraph enthält zusätzlich auch Objekte jener Klasse, welche die Rolle einer Kamera oder eines Fensters in die 3D-Szene übernehmen, die sogenannte *ViewPlatform*. Abbildung 42 illustriert anhand eines einfachen Scenegraph die wichtigsten Klassen des Java 3D API.

Abbildung 42 Java 3D Scenegraph



Ein `VirtualUniverse` enthält ein oder mehrere `Locale`-Objekte. Diese geben in Form von 256-Bit-Festkommazahlen die hochauflösenden Koordinaten (`HiResCoord`) des Ursprungs des lokalen Koordinatensystems des Scenegraph innerhalb des Universums an. Innerhalb eines Locale werden Koordinaten mit Gleitkommazahlen doppelter Genauigkeit (`Double`) angegeben. Ein Locale enthält eine Menge von Subgraphen, deren Wurzel ein spezieller Knoten, eine `BranchGroup` ist. Dies ist die einzige Klasse von Knoten, die aus einem momentan gerenderten Scenegraph, man spricht von einem „live scenegraph“, entfernt werden können. Eine BranchGroup ist eine Subklasse von `Group` und kann beliebig viele Kindknoten enthalten.

Eine `TransformGroup` nutzt ein `Transform3D`-Objekt, das intern eine 4x4-Matrix enthält, um eine Transformation (Translation, Rotation, Skalierung und Scherung) zu beschreiben. Diese Transformation wirkt sich auf alle Kindknoten der TransformGroup aus. Die Transformation kann durch ein Behaviour-Objekt abhängig vom Zeitverlauf oder von Benutzerinteraktionen geändert und so eine Animationen realisiert werden. Objekte der Klasse `Shape3D` repräsentieren schließlich die in der Szene sichtbaren Objekte. Sie nutzen ein `Geometry`-Objekt, das ihre Form beschreibt. Die Beschreibung kann beispielsweise in Form von zusammenhängenden Dreiecken (`TriangleArray`) oder einer Liste von Eckpunkten (`PointArray`) erfolgen. Zusätzlich enthält die Beschreibung Angaben über die Oberflächennormalen und die Farben an einem Punkt, die für das Rendering, insbesondere für die Beleuchtungsberechnung genutzt werden. Ein `Appearance`-Objekt erlaubt eine genauere Definition des Aussehens eines Shape3D. Auch hier kann die Farbe des Gesamtobjekts angegeben werden. Des weiteren erlaubt die Angabe von Material, Transparenz und Textur eine realistischere Darstellung. Während ein Shape3D-Objekt nur Kind eines einzigen Group-Knotens sein darf, können sich Shapes Geometry- und Appearance-Objekte teilen. Ein `ViewPlatform`-Objekt beschreibt die Position eines `View`, durch Änderung der Transformation einer oberhalb im Graphen gelegenen TransformGroup kann der View durch die Szene bewegt werden. Ein View ist schließlich mit einem `Canvas3D` verbunden, einer Subklasse von Canvas aus dem Abstract Window Toolkit AWT, auf dem die Ausgabe der Szene aus Sicht der ViewPlatform erfolgt. Da es sich bei Canvas3D wie bei allen AWT-Komponenten um eine „schwergewichtige“ Komponente handelt, müssen bei der Kombination mit den „leichtgewichtigen“ Swing-Widgets einige Besonderheiten beachtet werden. Insbesondere bei der Verdeckung eines Canvas3D durch eine Swing-Element, wie zum Beispiel ein `JMenu` oder ein `JInternalFrame`, überschreibt der Canvas3D die Swing-Komponente. Dies reduziert die Flexibilität beim GUI-Entwurf, so wurde zum Beispiel auf die Realisierung eines *Multiple Document Interface* (MDI) mit JInternalFrames verzichtet. Weiterer Bestandteil jedes Szenegraphen sind verschiedene Lichtquellen (Subklassen von Light). Mit ihnen lässt sich diffuses Umgebungslicht definieren, das von überall zu kommen scheint, aber auch Licht aus einer bestimmten Richtung. Ein parallel verlaufender Strahlengang erlaubt die Simulation weit entfernter Lichtquellen. Ihre Helligkeit ist in der gesamten Szene konstant. Beleuchtung, die ihren Ursprung in der Szene haben, können durch rundumstrahlende punktförmige Lichtquellen oder Spotlights realisiert werden. Ihre Helligkeit nimmt mit zunehmender Entfernung vom Ursprung des Lichts ab. Eine automatische Berechnung von Schatten ist mit den Bordmitteln von Java 3D nicht möglich, alle Shapes sind für Lichtquellen transparent.

Durch Verkettung mehrerer TransformGroups lassen sich aus globaler Sicht komplexe Bewegungen stark vereinfachen. So kann im mittleren Teilgraphen von Abbildung 42 die obere TransformGroup beispielsweise die Bewegung eines Roboterarms steuern. Die beiden Shapes könnten die Zangen eines Greifers repräsentieren. Bewegt sich der Arm durch Änderung der oberen TransformGroup, bewegen sich die Zangen mit. Diese können jedoch auch individuell und unabhängig von ihrem Vaterknoten lediglich unter Berücksichtigung ihres lokalen Koordinatensystems bewegt werden. Durch dieses Konzept wird nicht nur die Entwicklung erleichtert, sondern auch die Integration vorhandener komplexer 3D-Objekte in andere Objekte ermöglicht.

Zur Optimierung des Scenegraph kann dieser vor dem Rendering kompiliert werden. Dabei werden insbesondere Redundanzen durch Zusammenfassen von Knoten eliminiert. Zum Beispiel können zwei TransformGroups durch eine einzige ersetzt werden, deren Transformation das Produkt der beiden anderen ist. Eine derartige Optimierung kann jedoch Probleme bereiten, wenn die Objekte bewusst getrennt wurden, um sie unabhängig voneinander manipulieren zu können. Deshalb verfügt jeder Knoten über *Capabilities*, die dem Scenegraph-Compiler Hinweise geben, welche Bestandteile eines Graphen statisch sind und optimiert werden dürfen beziehungsweise wo Änderungen und auch lesende Zugriffe auf Knoten und ihre Attribute möglich sind. Standardmäßig sind alle Capabilities deaktiviert, was bedeutet, dass eine Änderung oder das Auslesen von Eigenschaften mit Beginn des Rendering nicht mehr möglich ist. Je nach Art des Knotens kann er über verschiedene Capabilities verfügen: ein Shape3D kann zum Beispiel das Lesen und Verändern seiner Geometrie oder seiner Appearance ermöglichen, Gruppen können das Lesen, Ändern oder Hinzufügen von Kindknoten gestatten. Eine TransformGroup kann eine Manipulation ihres Transformationsobjekts erlauben, eine BranchGroup als aus dem Graphen entfernbar markiert werden.

Um die für 3D-Darstellung notwendigen Berechnungen zu vereinfachen, enthält das Java 3D API neben den Klassen, die das virtuelle Universum und die Knoten des Szenegraphen darstellen, auch eine große Anzahl von Klassen für Vektor- und Matrizenrechnung (`javax.vecmath`). Zusätzlich stehen für die von Sun unterstützen Plattformen zahlreiche Hilfsklassen bereit. Diese vereinfachen zum Beispiel die Erzeugung von Geometrien oder bringen vordefinierte Behaviour-Objekte mit. Neben der Darstellung von bewegten und statischen 3D-Objekten erlaubt Java 3D auch die Positionierung von Schallquellen im Raum.

5.1.5 Konfiguration

Der Prototyp ist in weiten Teilen an die Bedürfnisse des Benutzers und die Gegebenheiten der Systemumgebung anpassbar. Das Package `vlab.config` beinhaltet alle Klassen, welche das Konfigurationsmanagement übernehmen. Die Klasse `Config` dient als Zugangspunkt für alle Klassen des Systems zu den Konfigurationseinstellungen in Form von statischen Variablen. Diese sind zunächst in einer sicheren Standardeinstellung vorkonfiguriert. Um dem Benutzer des virtuellen Labors die Möglichkeit zu geben, ohne Neukompilation dauerhafte Konfigurationsänderungen durchzuführen, sucht das System standardmäßig nach einer Konfigurationsdatei config.xml im Installationsverzeichnis. Alternativ kann mit dem Schalter `-conf` ein absoluter oder relativer Pfad zu einer Konfigurationsdatei angegeben werden. Wird keine Datei gefunden, werden die Standardeinstellungen beibehalten. Die

Konfigurationsdatei hat einen einfachen Aufbau: Geklammert zwischen den Wurzeltags (*config*) befindet sich eine Liste mit *variable*-Tags. Diese Verfügen über ein Attribut *name*, das den Namen der zu setzenden Konfigurationsvariablen angibt. Der Wert der Variablen wird als Inhalt des variable-Tags angegeben. Abbildung 43 zeigt diese einfache Grammatik als DTD.

Abbildung 43 DTD für Konfigurationsdateien

```
<!ELEMENT config (variable*)>
<!ELEMENT variable (#PCDATA)>
<!ATTLIST variable name CDATA #required>
```

Eine Beschreibung aller Konfigurationsvariablen soll an dieser Stelle nicht erfolgen. Statt dessen sei auf die Javadoc-Dokumentation der Klasse Config verwiesen, die eine umfassende Beschreibung aller Variablen und ihrer Auswirkungen enthält. Wichtige Konfigurationsvariablen werden zudem in den folgenden Kapiteln bei der Erläuterung der Komponenten vorgestellt, auf die sie sich auswirken.

5.2 Datenhaltungsschicht

Nach der Entscheidung für IBM DB2 als Basis für die Datenhaltungsschicht galt es, die von diesem Datenbanksystem zur Verfügung gestellten Konzepte zur Erfüllung der Anforderungen des Bausteinmodells aus Kapitel 4.3 zu nutzen.

5.2.1 Informationsmodell

Bei der Abbildung des in Form eines erweiterten ERM entwickelten Informationsmodells des virtuellen Labors auf die Strukturen von DB2 wurde nach dem Grundsatz verfahren, dass zumindest alle für Suchoperationen benötigten Informationen in strukturierter Weise in der Datenbank abgelegt werden. Dies erlaubt die volle Nutzung der Mächtigkeit von SQL beim Retrieval. Jene Teile des Informationsmodells, die in den vorgesehenen Nutzungsszenarien grundsätzlich als Ganzes geladen oder gespeichert werden, wurden dagegen exemplarisch in aus Datenbanksicht unstrukturierter Form abgelegt. Bei der Umsetzung des ERM auf die relationalen Strukturen wurden weitgehend die anerkannten Abbildungsregeln genutzt. Eine ausführliche Erläuterung dieser Abbildung ist daher nicht erforderlich. Lediglich die Abweichungen von diesen Regeln insbesondere die unstrukturiert gespeicherten Informationen sollen hier näher vorgestellt werden.

5.2.1.1 Mutatoren

Bei der Beschreibung des Informationsmodells für Mutatoren in Abschnitt 4.3.10.7 wurde bereits festgestellt, dass wegen der Vielzahl an möglichen Zielobjekten eines Mutators bei einer Abbildung auf das Relationenmodell eine Vielzahl von Tabellen für die n:m-Beziehungen erforderlich ist. Da Mutatoren grundsätzlich als Ganzes beim Laden oder Er-

zeugen einer Bausteininstanz aus der Datenbank entnommen werden, sind keine Suchanfragen auf Mutatoren zu erwarten. Lediglich eine Zuordnung eines Mutators zu seinem Bausteintyp ist erforderlich. Daher wurde lediglich die Relationship *parent*, welche die schwache Entity Mutator mit ihrer Bausteininstanz verbindet, umgesetzt. Alle anderen Bestandteile werden als large object gespeichert.

Als Format zur Beschreibung von Mutatoren sind prinzipiell Binär- und Textformate geeignet. Um in der Anfangsphase Mutatoren ohne Werkzeughilfe erstellen zu können, bietet ein menschenlesbares Textformat erhebliche Vorteile. Wegen der vielseitigen Unterstützung in gängigen Programmiersprachen war die Wahl eines XML-basierten Mutatorformats die naheliegende Entscheidung. Um eine Validierung von Mutator-Dokumenten zu erlauben und gleichzeitig dem Bausteindesigner eine Anleitung für die Entwicklung von Mutatoren zu geben, wurde die Grammatik zur Mutatordefinition als XML Schema entworfen. Die Beschreibung der Bedingungen und Operationen selbst ist wegen der besseren Lesbarkeit als nicht weiter strukturierter Text realisiert, dessen Grammatik der EBNF-Darstellung in Abschnitt 4.3.9.4 entspricht. Bedingungen haben einen Aufbau gemäß der condition-Produktion, für Operationen auf Zielen gilt die term-Produktion. Beispiel 4 zeigt einen einfachen Mutator. Er beschreibt die Position der Komponente 1 eines komplexen Bausteins (submodule) in Abhängigkeit von den Eigenschaften mit den IDs 4 (Breite, width) und 6 (Höhe, height) des komplexen Bausteins. Der Entwickler gibt als Inhalt der property-Tags eine Zeichenkette an, die der variable-Produktion der EBNF-Darstellung in Abschnitt 4.3.9.4 entsprechen muss und die in den folgenden Bedingungen oder Termen stellvertretend für den Wert der Eigenschaft genutzt werden kann. Es handelt sich um einen bedingten Mutator, der nur aktiv ist, wenn die Höhe kleiner 50cm bleibt. Die Position des Ursprungs des lokalen Koordinatensystems des Submoduls ergibt sich für x- und y-Koordinate in Abhängigkeit von Breite und Höhe, die z-Koordinate ist konstant. Die Rotation erfolgt um die y-Achse, der Rotationswinkel ist von der Breite abhängig.

Beispiel 4 Mutator in XML-Darstellung

```
<?xml version="1.0"?>
<mutator xmlns="http://www.informatik.uni-kl.de/vlab/mutator"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.informatik.uni-
         kl.de/vlab/mutator file:///C:/DA/schemas/mutator.xsd"
         type="MoveSolidMutator">
        <properties>
            <property id = "4">$width$</property>
            <property id = "6">$height$</property>
        </properties>
        <conditions>
               <condition>$width$&lt;50[cm]</condition>
        </conditions>
        <targets>
            <solid id = "1">
                <translation axis ="x">$width$/2</translation>
                <translation axis ="y">$height$/2</translation>
```

```
                <translation axis ="z">2</translation>
                <rotation axisx="0" axisy="1" axisz="0"
                 angle="PI/2*$width$"/>
            </solid>
    </targets>
</mutator>
```

Die Speicherung der XML-Dokumente kann als Attribut vom Typ VARCHAR oder für große Dokumente als CLOB erfolgen. Um eine spätere Nutzung des XML Extenders zu erlauben, wurden dessen aus diesen Basistypen abgeleiteten Datentypen XMLVARCHAR und XMLCLOB verwendet. Dabei stellte sich heraus, dass XMLVARCHAR eine Maximalgröße von lediglich 3000 Zeichen hat, was für umfangreichere Mutatoren nicht ausreichend ist. XMLCLOB ist dagegen mit einer maximalen Größe von zwei Gigabytes mehr als ausreichend dimensioniert. Da für CLOBs ab einer Größe von einem Gigabyte jedoch kein Logging mehr möglich ist, muss nun das Logging für die XML-Spalte deaktiviert werden. Zur Umgehung kann vor der Aktivierung einer Datenbank für den XML-Extender ein eigener distinct type XMLCLOB mit einer maximalen Größe von weniger als einem Gigabyte angelegt werden.

5.2.1.2 Constraints

Beim Vergleich von Constraints und Mutatoren fällt auf, dass Constraints als bedingte Mutatoren ohne Zielobjekte aufgefasst werden können. Dementsprechend bietet sich auch hier eine XML-Darstellung an. Bei geeigneter Auslegung des für die Auswertung der Mutatoren zuständigen Parsers lässt er sich auch zum Parsen der Constraints einsetzen.

5.2.1.3 Verknüpfung von relationalen und XML-Daten

Bei der bisher diskutierten Darstellung von Mutatoren und Constraints als XML-Dokumente bleibt das Problem, dass in den Dokumenten Bezug auf Elemente des relationalen Schemas durch Angabe ihrer Primärschlüssel beziehungsweise schwachen Schlüssel genommen wird. Die Gültigkeit dieser Beziehungen wird jedoch nicht vom Datenbanksystem sichergestellt, sondern muss vom Entwickler eines Bausteintyps sichergestellt werden. Geht man davon aus, dass die Bausteindefinition letztlich toolunterstützt stattfinden wird, kann man diesen Zustand im Vertrauen auf die Korrektheit des Entwurfswerkzeugs akzeptieren. Insbesondere beim Prototypen stand jedoch ein solches Tool nicht zur Verfügung. Daher wurde als alternative Möglichkeit die Eignung des XML-Extenders zur Integritätssicherung geprüft. Die bereits als XML-Extender-Datentypen definierten Spalten von Mutator und Constraint-Tabellen wurden als XML-Column aktiviert und durch Angabe einer DAD-Datei side tables für jene Attribute erstellt, die eine Beziehung zum relationalen Teil des Datenmodells herstellen. Die side tables enthalten dann für jedes Attribut oder Element einen Eintrag mit seinem Wert. Kann ein Attribut oder Element mehrfach in einem Dokument vorkommen, enthält die Tabelle zusätzlich noch eine Sequenznummer, welche die Position des jeweiligen Attributwertes im Dokument angibt. Dieses eigentlich zur Indexierung des XML-Dokuments vorgesehene Konzept lässt sich dann tatsächlich zur Integritätssicherung zwischen relationa-

ler und XML-Welt nutzen, indem man referentiellen Integritätsbedingungen zu den Spalten hinzufügt, welche die Attributwerte beinhalten. Versucht man nun ein XML-Dokument einzufügen, das auf eine Property oder ein Zielobjekt verweist, das es nicht gibt, wird man mit einer Fehlermeldung auf das verletzte Constraint hingewiesen. Dieses Verfahren funktioniert jedoch nur bei atomaren Fremdschlüsseln, bei zusammengesetzten Fremdschlüsseln, wie sie zum Beispiel bei schwachen Entities vorkommen, ist sein Einsatz nicht möglich. Beispiel 5 zeigt eine DAD-Datei, welche die id-Attribute der Property-Tags von Constraints auf eine side table *property_ids* abbildet. Dazu wird der Name der XML-Spalte, der SQL-Typ des Attributs, ein XPath-Ausdruck mit seiner Position und eine Angabe über ein- oder mehrfaches Auftreten im Dokument (Attribut *multi_occurence*) benötigt.

Beispiel 5 DAD-Datei für eine XML-Column

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "C:\Programme\IBM\dxx\dtd\dad.dtd">
<DAD>
        <dtdid>constraint.dtd</dtdid>
        <validation>YES</validation>
        <Xcolumn>
        <table name="property_ids">
        <column name ="propertyid"
                type="INTEGER"
                path="/constraint/properties/property/@id"
                multi_occurrence="YES"/>
        </table>
        </Xcolumn>
</DAD>
```

5.2.1.4 Validierung von XML-Dokumenten

Der XML-Extender bietet die Option, einzufügende XML-Dokumente auf Einhaltung einer Grammatik zu überprüfen. Bisher können diese Grammatiken lediglich als DTDs formuliert werden, sodass die weitergehenden Einschränkungen von Dokumentinstanzen eines XML Schemas nicht garantiert werden können. Um die Überprüfung zu aktivieren, wird die DTD in eine bei der Aktivierung des XML Extenders in der Datenbank automatisch angelegte Tabelle DTD_REF eingefügt. In der DAD-Datei, welche die XML-Column oder -Collection beschreibt, kann wie in Beispiel 5 zu sehen mit dem Tag *dtdid* eine Referenz auf die jeweilige DTD angegeben und die Validierung mit dem *validation*-Tag aktiviert werden.

5.2.2 Globale Integritätsbedingungen

Bei der Entwicklung des Informationsmodells für Experimentbausteine wurden zahlreiche Integritätsbedingungen wegen systemweiter Gültigkeit als Kandidaten für eine Prüfung in der Datenhaltungsschicht identifiziert. Dies ermöglicht ein Sicherstellen der Integrität der Daten unabhängig von der Anwendung. Es galt also, diese ausgewählten Constraints in

DB2-Mechanismen umzusetzen. Für einfache Bedingungen können referentielle Constraints genutzt werden, wie sie von jedem relationalen DBMS angeboten werden. Weitergehende Integritätsbedingungen auf den Attributen einer einzigen Relation können mit CHECK-Constraints umgesetzt werden. Für komplexere Integritätsbedingungen wie die in Abschnitt 4.3.7.3 geschilderten sind diese Mechanismen jedoch nicht ausreichend. Da die ECA-Regeln und Assertions des SQL:1999-Standards von DB2 nicht unterstützt werden, musste zur Formulierung anspruchsvollerer Regeln auf Trigger zurückgegriffen werden. Normalerweise dazu vorgesehen, bei Änderungsoperationen auf einzelnen Tabellen und optional dem Erfülltsein von Bedingungen weitere Aktionen innerhalb der Datenbank auszulösen, können Trigger auch zur Integritätssicherung eingesetzt werden. Dazu werden die zu realisierenden Constraints als Bedingungen für die Trigger formuliert, so dass der Trigger bei deren Nichterfüllung aktiv wird. Als getriggerte Aktion wird dann nur ein Funktionsaufruf verwendet, der die Signalisierung von beliebigen Fehlercodes an die Anwendung ermöglicht.

Beispiel 6 Die Euler-Formel als Trigger

```
CREATE TRIGGER euler_formula_edge
AFTER INSERT ON edge
REFERENCING NEW as newedge
FOR EACH ROW MODE DB2SQL
WHEN (NOT (
   (SELECT COUNT (*) FROM edge WHERE solidid = newedge.solidid AND
   moduletypeid = newedge.moduletypeid) +
   (SELECT COUNT (*) FROM vertex WHERE solidid = newedge.solidid AND
   moduletypeid = newedge.moduletypeid)+
   (SELECT COUNT (*) FROM face WHERE solidid = newedge.solidid AND
   moduletypeid = newedge.moduletypeid)  = 2))
BEGIN ATOMIC
   SIGNAL SQLSTATE '89765' SET MESSAGE_TEXT = 'Euler-Formula not
   fullfilled for edge';
END
```

5.2.2.1 Topologische Integrität

Die Euler-Formel erlaubt wie in Kapitel 4.3.3.3 beschrieben eine Überprüfung der topologischen Integrität durch Angabe einer Gleichung, die für die Zahl der Geometrie-Elemente erfüllt sein muss. Als DB2-Triggers lässt sie sich wie in Beispiel 6 gezeigt umsetzen.

Trigger mit identischem Aufbau sind für alle Einfüge-, Lösch- und Änderungsoperationen auf den Tabellen Edge, Vertex und Face zu formulieren.

5.2.2.2 Vererbung von Eigenschaften

Auch die Formulierung komplexerer Bedingungen als das einfache Zählen und Summieren von Tupeln ist möglich. Um die Suchfunktionen innerhalb der Kategorienhierarchie der

Bausteinbibliothek auf Grundlage von Eigenschaften der Kategorien durchzuführen, muss sichergestellt sein, dass alle Kategorien und Bausteintypen, die direkt oder indirekt in einer Kategorie sind, über mindestens alle Eigenschaften dieser Kategorie verfügen. Beispiel 7 zeigt, wie sich dies als Trigger formulieren lässt.

Beispiel 7 Trigger zur Erzwingung der Vererbung von Eigenschaften auf Bausteintypen

```
CREATE TRIGGER mt_prop_inh
AFTER INSERT ON moduletype
REFERENCING NEW as newtype
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT * FROM c_has_prop hp
       WHERE hp.categoryid = newtype.categoryid
       AND NOT EXISTS (SELECT * FROM mt_has_prop hp2
                       WHERE hp2.modulepropertyid
                       =hp.modulepropertyid
                       AND hp2.moduletypeid = newtype.id)))
BEGIN ATOMIC
  SIGNAL SQLSTATE  '89801' SET MESSAGE_TEXT = 'A moduletype lacks at
  least one property of its category';
END
```

In natürlicher Sprache formuliert: „Falls die Kategorie, in die der neue Bausteintyp einsortiert wird, über eine Eigenschaft verfügt, die der neue Bausteintyp nicht hat, signalisiere einen Fehler.“ Auch hier sind analoge Trigger für Änderungsoperationen auf der Tabelle ModuleType, auf Lösch- und Änderungsoperationen von mt_has_prop sowie auf Einfüge- und Änderungsoperationen von c_has_prop erforderlich. Kombiniert mit einem ähnlich aufgebauten Trigger, der die Vererbung zwischen Kategorien und ihren Unterkategorien erzwingt, lässt sich so die Integrität der gesamten Bausteinhierarchie sicherstellen.

5.2.2.3 Schwierigkeiten bei der praktischen Nutzung

Als problematisch bei der Umsetzung von globalen Constraints hat sich nicht wie erwartet eine mangelnde Ausdrucksmächtigkeit der Trigger erwiesen. Vielmehr besteht für DB2-Trigger in ihrer aktuellen Form die Limitierung, dass sie grundsätzlich sofort auslösen, wenn die von ihnen überwachte Änderungsoperation ausgeführt wird und die Bedingungen erfüllt sind. Dies ermöglicht eine Integration der Trigger-Operationen in den Ausführungsplan der Nutzeroperationen und damit eine kombinierte Optimierung aller Operationen. Jedoch ist auf diese Weise eine Verzögerung der Ausführung des Triggers bis an das Ende einer Transaktion nicht möglich. Dadurch konnten viele prinzipiell realisierbare Constraints letztlich nicht als Trigger in die Datenbank aufgenommen werden, da durch das so entstehende komplexe Regelwerk ein Einfügen von neuen Objekten nur unter großen Schwierigkeiten durch Einhalten bestimmter Einfügereihenfolgen oder nachträgliches Ändern von Tupeln oder auch überhaupt nicht möglich ist. Als Beispiel sei hier das Einfügen eines neuen Bausteintyps genannt, der in eine Kategorie einsortiert wird, die über Eigenschaften verfügt. Zunächst muss der Bausteintyp eingefügt werden, erst dann können ihm wegen der Prüfung referentieller In-

tegrität Eigenschaften hinzugefügt werden. Der zuvor gezeigte Trigger löst jedoch sofort aus. Zur Umgehung des Problems erzeugt man den Eintrag in der Tabelle ModuleInstance zunächst ohne einen Eintrag für categoryid, fügt die Eigenschaften zum Baustein hinzu und setzt dann die categoryid auf den gewünschten Wert. Es können aber auch leicht Fälle konstruiert werden, bei denen auch derartige Maßnahmen nicht funktionieren, da es keine Einfügereihenfolge gibt, die nicht in einem Zwischenzustand eine der Integritätsbedingungen verletzt und so einen der Trigger auslöst. Der Trigger zur Prüfung der Euler-Formel ist ein solches Beispiel. Zunächst erzeugt man einen neuen Solid durch Einfügen eines Solid-Tupels. Bei einer strikten Auslegung der Regel müsste schon jetzt der Euler-Trigger geprüft werden und würde folglich auslösen, da 0 Eckpunkte - 0 Kanten + 0 Flächen - 2 ≠ 0 ist. Verzichtet man für die Tabelle Solid noch auf die Überprüfung wird spätestens beim Einfügen des ersten Eckpunkt-, Kanten- oder Flächentupel eine Ausnahme ausgelöst. Da es sich um schwache Entities handelt, besteht hier auch nicht die Möglichkeit, die Fremdschlüssel auf Solid erst anschließend zu setzen, da sie Teil des Primärschlüssels sind.

Doch auch die einfache Verwendung referentieller Integrität kann zu solchen Problemen führen. Wie bei Triggern bietet DB2 keinen gangbaren Weg, die Prüfung dieser Constraints an das Ende einer Transaktion zu verschieben. Zwar lässt sich die Prüfung aller oder einzelner Constraints einer Tabelle mit dem Kommando

```
SET INTEGRITY FOR <tablename> <constraints> IMMEDIATE UNCHECKED
```

deaktivieren. Die Tabelle wird daraufhin jedoch in den *check-pending*-Status versetzt, in dem keine Einfüge- und Änderungsoperationen erlaubt sind, sondern lediglich das Laden der Tabelle möglich ist.

Spätestens bei direkten zyklischen Referenzen muss folglich auch hier mit Update-Operationen gearbeitet werden. Dürfen die Fremdschlüssel nicht auf NULL gesetzt werden, kann auch hier schnell eine Situation gefunden werden, bei der ein Einfügen nicht möglich ist. Gutes Beispiel ist auch hier die Edge-Relation, deren Vorgänger und Nachfolgerfremdschlüssel wieder auf andere Kanten verweisen und nicht nullwertig sein dürfen. Zur Umgehung dieses Problems muss mit einem `ALTER TABLE`-Statement das störende Constraint gelöscht und anschließend neu erzeugt oder auf `NOT ENFORCED` und am Ende wieder auf `ENFORCED` gesetzt werden. Beide Lösungen können je nach interner Umsetzung jedoch mit jedem Reaktivieren eine Überprüfung der kompletten Relation erfordern und sind wegen der so zu erwartenden Leistungseinbußen nicht akzeptabel.

5.2.3 Primärschlüssel

Um die Primärschlüssel für neu zu speichernde Anwendungsobjekte zu erzeugen, bietet DB2 wie viele Datenbanksysteme eine automatische Generierung der Schlüssel an. Dies kann beim Erzeugen einer Relation durch das Schlüsselwort `generated` geschehen:

```
CREATE TABLE some_table(
id INTEGER GENERATED ALWAYS AS IDENTITY
                    (START WITH 42 INCREMENT BY 64)
...)
```

Vieles spricht für die Nutzung dieser Möglichkeit. Im Gegensatz zu anderen Ansätzen, die den jeweils aktuellen Primärschlüssel für jede Tabelle in einer Primärschlüssel-Tabelle speichern, ihn für das Speichern eines neuen Tupels daraus entnehmen und dann inkrementieren, haben automatisch generierte Schlüssel den Vorteil der besseren Nebenläufigkeit, da die Primärschlüsseltabelle einen potentiellen Engpass darstellt. Dennoch wurde auf automatisch erzeugte Schlüssel verzichtet, da der von JDBC vorgesehene Mechanismus für das Erfragen des bei einem Insert erzeugten Primärschlüssels, die Methode `getAutoGeneratedKeys` des Interface `Statement` vom aktuellen JDBC-Treiber von DB2 nicht unterstützt wird. Stattdessen steht die benutzerdefinierte Funktion IDENTITY_VAL_LOCAL zur Verfügung. Das Anbieten dieses nicht-standardkonformen Mechanismus statt einer vollständigen Implementierung der JDBC-Spezifikation erschwert die Portabilität, daher wurde auf eine Primärschlüsseltabelle zurückgegriffen.

5.3 Datenmodellschicht

Die Datenmodellschicht wurde gemäß der Überlegungen in Abschnitt 4.1.2.5.5 als universell einsetzbare Java Bibliothek realisiert. Sie umfasst sowohl Klassen, welche die Objekte der Anwendung repräsentieren, als auch Komponenten für den Datenbankzugriff und Factories für komplexe Objekte.

5.3.1 Anwendungsobjekte

Zentraler Bestandteil der Datenmodellschicht sind die Anwendungsobjekte, die sie den übergeordneten Schichten des virtuellen Labors zur Verfügung stellt. Sie befinden sich im Package `vlab.model` und sind weiter in Unterpackages strukturiert. Ein großer Teil der Anwendungsobjekte repräsentiert einzelne Elemente des Bausteinmodells.

5.3.1.1 Eigenschaften

Das Package `vlab.model.property` enthält alle Bestandteile des Eigenschaftsmodells des virtuellen Labors.

Property und Subklassen

Die abstrakte Klasse `Property` dient als Superklasse für alle Eigenschaften und implementiert deren gemeinsame Schnittstelle in Form von (zum Teil abstrakten) Methoden. Dazu gehören zunächst *getter*- und *setter*-Methoden für Name, Typ und Id (Primärschlüssel) der Properties und Methoden für das Erfragen bestimmte Eigenschaften (Nullwerte erlaubt, Wert editierbar, Wert vorhanden). Weiterhin bietet sie eine Schnittstelle für das Registrieren von Constraint- und Mutatorobjekten, die dann bei Änderungen des Wertes der Eigenschaft über diese Änderung gemäß dem Observer-Pattern informiert werden, sowie eine abstrakte Methode, welche die möglichen Vergleichsoperatoren für die Eigenschaft liefert. Dies wird für Suchfunktionen benötigt.

Die abstrakte Klasse `SimpleProperty` erbt von Property und repräsentiert Eigenschaften basierend auf primitiven Datentypen. Sie ergänzt Methoden für den Zugriff auf den Typ einer Eigenschaft und für den Umgang mit Einheiten bei numerischen Typen.

Die nicht-abstrakten Klassen `IntegerProperty`, `DoubleProperty` und `StringProperty` erben von SimpleProperty, bieten ein Feld für den Wert dieser Eigenschaften und implementieren alle abstrakten Methoden von Property.

Die abstrakte Klasse `ComplexProperty` repräsentiert nicht atomare Eigenschaften, die selbst aus mehreren primitiven Datentypen zusammengesetzt sind. Als Beispiel für eine komplexe Eigenschaft wurde die Klasse `ComplexSolidMaterialProperty` implementiert. Sie verfügt über Name, Reibungskoeffizient und Dichte. Werte dieser Eigenschaft werden durch die Klasse `ComplexSolidMaterialPropertyValue` repräsentiert.

PropertyType und Subklassen

Die Klasse `PropertyType` dient als Wurzel der Vererbungshierarchie von Eigenschaftstypen, so wie die Klasse Property die Wurzel für Eigenschaften ist. Die Klasse bietet Methoden für das Erfragen von Name und Primärschlüssel des Eigenschaftstyps sowie für den zur Repräsentation von Werten des Eigenschaftstyps verwendeten Datentyp. Die Klasse `SimplePropertyType` repräsentiert alle Typen, die auf primitive Datentypen abgebildet, also durch eine `Integer-`, `Double-` oder `StringProperty` repräsentiert werden. Sie bietet mit verschiedenen Methoden Zugriff auf alle Einheiten eines Typs. So kann seine Standardeinheit erfragt werden, eine einfache Liste der Einheiten generiert oder eine Hashtable erzeugt werden, die einen Zugriff auf die Einheiten anhand ihres Kürzels erlaubt. Die abstrakte Klasse `ComplexPropertyType` repräsentiert komplexe Typen. Da diese wahlweise als *binaryvalue* gespeichert oder aber in einer eigenen Relation untergebracht werden, ermöglicht sie, die Art der Speicherung zu erfahren. Die beiden Subklassen `HierarchicalPropertyType` und `EnumerationPropertyType` stehen für hierarchisch organisierte beziehungsweise „flache“ komplexe Eigenschaften.

Unit

Numerische Typen können über eine beliebige Zahl von Einheiten verfügen, jeder Wert eines solchen Typs liegt in einer dieser Einheiten vor. Die Klasse `Unit` repräsentiert eine Einheit mit Name, Kürzel, Konversionsfaktor und -offset. Zusätzlich bietet sie statische Methoden zur Konversion von Integer- oder Double-Werten von einer Ausgangs- in eine Zieleinheit.

Interface PropertyContainer

Um für alle Anwendungsobjekte, die über Eigenschaften verfügen können, eine einheitliche Schnittstelle zu definieren, beschreibt das Interface `PropertyContainer` Methoden für das Setzen und Auslesen der Eigenschaften.

5.3.1.2 Bibliotheksobjekte

Das Package `vlab.model.library` enthält alle Objekte, die für die Darstellung der Bausteinbibliothek benötigt werden. Dazu gehören insbesondere Objekte, die das PropertyContainer-Interface implementieren, wie Kategorien, Bausteintypen und Varianten. Diese Anwendungsobjekte sind teilweise bereits im virtuellen Labor durch Klassen repräsentiert: So kann ein Objekt `ModuleInstance` auch zur Darstellung einer Bausteinvariante verwendet werden. Es wäre jedoch wenig zweckmäßig, alle Informationen zum Beispiel über die Geometrie schon beim Browsen in der Bausteinbibliothek mitzuführen. Vielmehr genügen hier die Eigenschaften der Variante und ihre Werte. Daher wurden für die Bibliotheksfunktionen eigenständige Klassen implementiert, die sich auf diese Bestandteile beschränken. Da die Darstellung der Bibliothek im GUI in Form eines Baumes erfolgt, erben die Klassen `Category`, `ModuleTypeMetaData` und `ModuleVariantMetaData` von einer gemeinsamen Superklasse `LibraryNode`, die direkt als Knoten eines JTree verwendet werden kann. Die Klasse `ExperimentMetaData` repräsentiert die Metadaten eines Experiments wie Name, Besitzer und Zeitpunkt der letzten Modifikation. Sie enthält keine Informationen über die Bausteine im Experiment und wird nur bei der Auflistung der Experimente genutzt.

5.3.1.3 Experimente und Bausteine

Im Package `vlab.model.experiment` sind die Klassen zusammengefasst, die Anwendungsobjekte zur direkten Nutzung in einem Experiment repräsentieren. Da die meisten dieser Objekte später vom GUI mit Java 3D dargestellt werden, wurden sie zunächst als Subklassen von Java 3D Klassen realisiert und konnten so direkt an den Renderer übergeben werden. Diese Modellierung ist effizient, schränkt aber die Verwendung der Objekte ein. Zudem ist dieser Ansatz bei der Verwendung von Middleware-Technologien problematisch, da die relativ komplexen Objekte nicht ohne weiteres serialisiert werden können. Daher wurde eine Trennung der Objekte in Datenmodell und Delegate vorgenommen, ähnlich wie dies im Swing-API für Widgets vorgesehen ist.

Vertex

Ein Objekt der Klasse `Vertex` repräsentiert einen Eckpunkt eines Solids. Es enthält in Instanzvariablen die Koordinaten des Punktes relativ zum Koordinatensystem des Solids. Zusätzlich verfügt ein Vertex über eine Liste mit allen Flächen (*Faces*), die ihn zum Eckpunkt haben. Diese Flächen können dann benachrichtigt werden, wenn Koordinaten des Eckpunkts zum Beispiel durch einen Mutator geändert werden.

Loop

Ein `Loop` repräsentiert einen Zyklus von Kanten. Die Kanten sind nicht direkt modelliert, sondern lediglich durch eine Abfolge von Eckpunkten als Vertex-Objekte. Diese sind in einem Array im Uhrzeigersinn geordnet, wenn es sich um einen outer loop handelt, beziehungsweise entgegen dem Uhrzeigersinn bei inner loops. Ein Feld erlaubt die Unterscheidung zwischen inner und outer loops.

Face

Objekte der Klasse `Face` beschreiben einzelne Flächen eines Solids. Im Prototypen wurde zur Reduktion der Komplexität eine Beschränkung auf ebene Flächen mit Strecken als Kanten vorgenommen. Flächen verfügen über eine Farbe und bestehen optional aus einem Material (ein Objekt der Klasse ComplexSolidMaterialPropertyValue). Eine Fläche wird wie in Kapitel 4.3.3 beschrieben durch genau einen äußeren Kantenzyklus (outer loop) begrenzt. Zusätzlich kann sie über eine beliebige Anzahl von inneren Kantenzyklen (inner loops) verfügen, die Löcher in der Fläche beschreiben. Die Klasse Face bietet Methoden, welche die Eckpunkte eines sie vollständig umschließenden, an den Koordinatenachsen ausgerichteten Quaders (bounding box) ermitteln.

Solid

Ein `Solid` besteht aus einer Menge zusammenhängender Flächen. Referenzen auf die Flächen und auf alle Eckpunkte sowie Informationen über Position und Orientierung werden in Instanzvariablen gespeichert.

ModuleInstance

Instanzen von Bausteintypen werden durch die Klasse `ModuleInstance` repräsentiert. Sie implementiert das PropertyContainer-Interface und hält Referenzen auf alle Bestandteile einer Bausteininstanz: Eigenschaften, Solids, Moduleinstanzen, die als Subkomponenten fungieren, Constraint- und Mutatorobjekte.

Interface Selectable

Der Benutzer des virtuellen Labors kann Bausteininstanzen, Subkomponenten, Solids und Faces auswählen. Das `Selectable`-Interface bietet Methoden zum Selektieren und Deselektieren dieser Objekte. Dies kann sich auf die Darstellung des Objekts auswirken. Zusätzlich bietet es eine Methode zum Erfragen des approximierten geometrischen Mittelpunkts des selektierten Objekts. Dieser kann dann als Zentrum für Rotationsbewegungen oder für die Ausrichtung der Sichten im GUI genutzt werden.

5.3.1.4 Operationen

Sowohl bei der Definition von Mutatoren als auch für Constraints kann der Entwickler eines Bausteins Bedingungen und Terme formulieren. Diese können wie in der in Abschnitt 4.3.9.4 vorgestellten Grammatik aus einfachen Operationen, Literalen und Variablen als Platzhalter für Eigenschaften bestehen. Um die Bedingungen zur Laufzeit schnell auswerten zu können, werden sie geparst und in eine Struktur von Objekten übersetzt.

SimpleOperationParser

Diese Klasse nimmt einen String entgegen, der die Bedingung beziehungsweise den Term (kurz: Operation) beschreibt, dazu ein Array der in dieser Operation als Variablen genutzten Eigenschaften (Property-Objekte) sowie der für diese Eigenschaften deklarierten Variablennamen. Um bei der Formulierung der Operationen Literale mit Einheiten versehen zu kön-

nen, kann der Klasse zusätzlich eine Menge von Hashtables übergeben werden, in der die Einheiten der verwendeten Eigenschaftstypen mit dem Kürzel der Einheit als Schlüssel enthalten sind. Mit diesen Informationen kann der Operationsstring geparst werden. Als Ergebnis liefert der SimpleOperationParser eine `Comparator`-Objekt für Bedingungen beziehungsweise ein `Term`-Objekt für Terme zurück. Dieses Objekt stellt dabei die Wurzel eines Baums aus Comparator- und Term-Objekten dar, der die Operation nachbildet.

Interface Term und Term-Klassen

Eine Klasse, die das Term-Interface implementiert, repräsentiert unäre oder binäre Operatoren, ein Literal oder eine Variable. Während die Operatoren die inneren Knoten des Operations-Baumes darstellen, sind Literale und Knoten seine Blätter. Ein binärer Operator verfügt über zwei Kindknoten, die selbst Terme sind, ein unärer Operator verfügt über ein Kind.

Das Interface Term definiert eine Methode evaluate, mit welcher der Term ausgewertet und das Ergebnis der Auswertung zurückgeliefert wird. Je nach Term-Typ erfolgt die Auswertung unterschiedlich. Ein Literal liefert seinen bei seiner Erzeugung gespeicherten Wert zurück, eine Variable den aktuellen Wert der Eigenschaft, die sie repräsentiert. Dazu verfügt sie über eine Referenz auf diese Eigenschaft. Ein Operator-Knoten muss dagegen zunächst sein Kind beziehungsweise seine Kinder auswerten und ruft dazu dessen beziehungsweise deren evaluate-Methode(n) auf. Die so generierten Ergebnisse werden entsprechend dem konkreten Typ des Operators verknüpft, zum Beispiel wird ein Multiply-Operator sie multiplizieren. Eine weitere evaluate-Methode nimmt eine Hashtable als Parameter entgegen. Diese enthält hypothetische Werte für einzelne der Eigenschaften, die im Term als Variable auftreten. Damit kann ein Term testweise ausgewertet werden, ohne die Eigenschaften zu verändern.

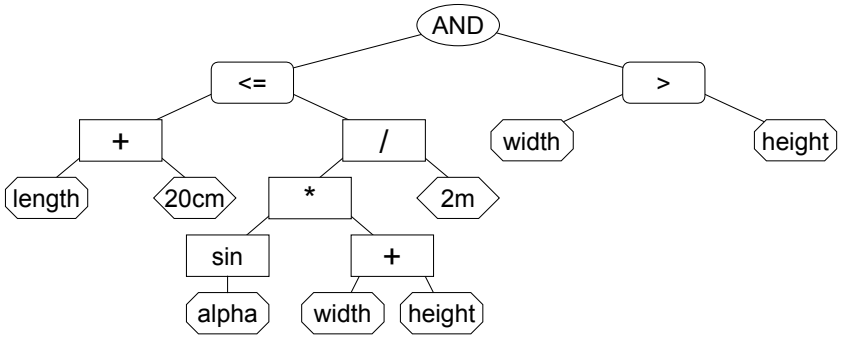
Interface Comparator und Comparator-Klassen

Während Terme genügen, um die Auswirkungen eines Mutators zum Beispiel auf die Koordinaten eines Solids zu beschreiben, sind für Bedingungen zusätzlich Vergleichsoperatoren wie „gleich“, „nicht gleich“ oder „größer als“ erforderlich. Diese Vergleichsoperatoren oder Comparatoren implementieren das `Comparator`-Interface. Sie funktionieren ähnlich wie binäre Operatoren und verfügen über die beiden zuvor beschriebenen evaluate-Methoden. Diese liefern jedoch einen boolschen Wert als Ergebnis zurück. Um mehrere Vergleichoperatoren verknüpfen zu können, gibt es zusätzlich die boolschen Operatoren AND, OR und XOR. Abbildung 44 zeigt ein Beispiel für eine Operation und den zugehörigen Operationsbaum.

Abbildung 44 Beispiel für einen Operationsbaum

Operation:

```
length + 20[cm] <= sin(alpha) * (width + height) / 2[m]
AND width > height
```



5.3.1.5 Constraints

Das Integritätsmodell der Nutzersicht wird im Package `vlab.model.constraints` und seinen Unterpackages implementiert. Die Klasse `Constraint` dient als abstrakte Superklasse für die beiden im Prototypen implementierten Constraint-Klassen, `PropertyConstraint` und `ModuleConstraint`. Während sich PropertyConstraints auf genau eine Bausteineigenschaft beziehen, erfassen ModuleConstraints mehrere Eigenschaften. Die Funktionsweise der Klassen ist konzeptionell gleich und wird hier am Beispiel von ModuleConstraint erläutert. Ein ModuleConstraint referenziert eine Menge von Eigenschaften eines Bausteins, also Property-Objekte, und enthält eine Menge von Bedingungen, die auf diesen Eigenschaften formuliert sind. Die Speicherung von Constraints in der Datenhaltungsschicht wurde in Kapitel 5.2.1.2 beschrieben. Die XML-Darstellung wird in einem ersten Schritt von einem SAX-Parser (Klasse `ConstraintParser`) geparst. Er arbeitet mit einem Stack, der für jedes Element zwischen Dokumentwurzel und aktueller Position im Dokument ein Objekt einer Subklasse von `ElementHandler` enthält. Der Parser extrahiert die vom Constraint referenzierten Properties und die Beschreibung der Bedingungen gemäß der in Kapitel 4.3.9.4 vorgestellten Grammatik. Diese Informationen dienen als Eingabe für die Klasse `ConstraintFactory`, die ein neues Constraint-Objekt erstellt. Diesem übergibt sie Referenzen auf die für das Constraint relevanten Property-Objekte und die Strings mit den Bedingungen. Das Constraint registriert sich als Listener bei den Properties und lässt vom SimpleOperationParser die Operationsbäume für die Bedingungen erstellen. Wann immer eine der Properties geändert wird, ruft sie die evaluate-Methode aller ihrer Constraints auf, die das Ergebnis der Auswertung zurückliefern. Durch die Übergabe einer Hashtable mit hypothetischen Werten für die Eigenschaften lassen sich Constraints prüfen, ohne die Eigenschaften zu verändern.

5.3.1.6 Mutatoren

Mutatoren weisen einige konzeptionelle Ähnlichkeiten mit Constraints auf. Auch sie sind von einer Menge von Eigenschaften eines Bausteins abhängig und verfügen über Bedingungen. Diese Gemeinsamkeiten spiegeln sich auch in der XML-Darstellung wider. Daher teilt sich der `MutatorParser` einige Komponenten mit dem ConstraintParser, insbesondere die ElementHandler für gemeinsame Tags. Zusätzlich verfügt er über ElementHandler für die möglichen Zielobjekte des Mutators. Die aus dem Dokument gewonnen Informationen werden an die Klasse `MutatorFactory` übergeben. Da die Mutatoren je nach Zielobjekt sehr unterschiedlich sind, gibt es verschiedene spezielle Mutator-Klassen, die alle von der abstrakten Superklasse Mutator erben. So hat ein `MoveVertexMutator` einen oder mehrere Eckpunkte eines Solids zum Ziel, ein `MoveSolidMutator` einen Solid, eine `FaceMaterialMutator` eine oder mehrere Flächen. Die MutatorFactory versucht nun, mit dem Java Reflection API die im type-Attribut spezifizierte Mutator-Klasse zu instanzieren und übergibt dieser die geparsten Information. Der Mutator nimmt diese entgegen und speichert zunächst Referenzen auf die für ihn relevanten Properties, bei denen er sich ähnlich wie ein Constraint als Listener registriert. Anschließend generiert er Bedingungsobjekte mit Hilfe des SimpleOperationParser. Diese Bedingungen werden immer zuerst geprüft, wenn der Mutator über eine Änderung einer seiner Properties mit der `propertyChanged`-Methode informiert wird. Nur wenn alle Bedingungen erfüllt sind, wird der Mutator aktiviert. Ein aktivierter Mutator ändert entsprechend seines Typs seine Zielobjekte. Diese werden ebenfalls beim Erzeugen des Mutator-Objekts als Referenzen gespeichert. Für Zielobjekte, deren Änderungen sich aus einem oder mehreren vom Bausteinentwickler spezifizierten Termen ergeben, wie die neuen Koordinaten bei einem MoveSolidMutator, werden diese Terme mit dem SimpleOperationParser in Operation-Objekte umgewandelt. Wird der Mutator aktiviert, wird die Operation ausgewertet, und die so ermittelten neuen Werte werden im Zielobjekt gesetzt.

5.3.1.7 Alternatives Parsing von Constraints und Mutatoren

Neben dem zuvor Beschriebenen SAX-Parser zur Auswertung von Constraint- und Mutator-Dokumenten wurde versuchsweise JAXB für das Parsing genutzt. Basierend auf dem XML Schema von Constraints und Mutatoren erzeugt das xjc-Tool Klassen für alle Elemente des XML Schemas. Veranlasst man das Parsing eines Dokuments, erhält man eine Struktur von Objektinstanzen dieser Klassen, die den Aufbau des Dokuments widerspiegeln. Die Navigation innerhalb dieser Objektstruktur erfolgt im Gegensatz zu DOM statt mit generischen mit spezialisierten Methoden, die auf Elemente und Attribute des Schemas zugeschnitten sind. Da die von JAXB erstellten Objekte nicht direkt in der Applikation verwendet werden können, müssen auch sie zunächst ausgewertet und in Mutator- und Constraint-Objekte übersetzt werden. Die Implementierung erwies sich als etwas einfacher als die eines entsprechenden SAX-Parsers, ein möglicherweise vorhandener Leistungsnachteil war nicht zu erkennen.

5.3.2 Datenbankzugriff

Der Zugriff auf das Repository des virtuellen Labors ist in einem eigenen Package `vlab.db` gekapselt. Es enthält kontextfreie Klassen mit statischen Methoden zum Laden der Anwendungsobjekte aus vlab.model. Die Klassen fassen die Methoden nach Thematik zusammen und werden nach einer Beschreibung allgemeiner Konzepte beim DB-Zugriff in Ausschnitten vorgestellt.

5.3.2.1 Allgemeine Konzepte

Transaktionen

Für zahlreiche der komplexeren Datenbankoperationen wie das Laden eines Experiments sind mehrere SQL-Anfragen durchzuführen. Viele dieser Anfragen werden von mehreren Operationen benötigt und wurden daher in eigenen Methoden gekapselt. Jede der Methoden fordert eine Datenbankverbindung an, führt damit die Anfragen beziehungsweise Änderungsoperationen innerhalb einer Transaktion durch und gibt die Verbindung anschließend wieder frei. Bei dieser Realisierung wird jedoch jeder Methodenaufruf in einer eigenen Transaktion durchgeführt, sodass ein Aufruf einer komplexen Methode, die selbst mehrere Hilfsmethoden nutzt, als Ganzes nicht mehr atomar wäre. Daher kann jede der Hilfsmethoden als optionalen Parameter zusätzlich eine Referenz auf ein Verbindungsobjekt entgegen nehmen. Erhält eine Methode so eine Verbindung, nutzt sie diese statt selbst eine anzufordern. Die Operation findet damit innerhalb des Transaktionskontextes des Aufrufers statt. Damit wird eine Semantik realisiert, die an die des Transaktionsattributs *required* bei *container-managed transactions* in einem J2EE-Applikationsserver angelehnt ist: Eine bestehende Transaktion wird genutzt, falls jedoch keine vorhanden ist, wird mit der neuen Verbindung eine neue Transaktion erzeugt.

Authentifizierung und Authorisierung

Das Ergebnis einzelner Methoden für den Datenbankzugriff, zum Beispiel das Auflisten der privaten Experimente, ist abhängig vom jeweiligen Benutzer. Weiterhin sind für viele Operationen Nutzerrechte erforderlich. Zwar wurde auf eine Implementierung der in Kapitel 3.3.4 vorgestellten Rechteverwaltung verzichtet, dennoch ist die Infrastruktur für die nachträgliche Ergänzung des Authorisierungsmechanismus vorgesehen. Da die Zugriffsmethoden der Datenmodellschicht mit dem Ziel entwickelt wurden, kontextfrei zu funktionieren, um später möglichst flexibel einsetzbar zu sein, muss bei allen Methoden, die eine Authorisierung erfordern, ein User-Objekt mit Login und Passwort übergeben werden.

5.3.2.2 Komponenten

Eine vollständige Übersicht über die Komponenten des db-Packages gibt die Javadoc-Dokumentation. An dieser Stelle sollen lediglich einige zentrale Klassen kurz mit einer Beschreibung der Kernfunktionalität vorgestellt werden.

DBConnection

Die Klasse `DBConnection` dient als Factory für Datenbankverbindungsobjekte. Die anderen Klassen können mit der `getConnection`-Methode ein Verbindungsobjekt anfordern. Um die Datenmodellschicht flexibel einsetzen zu können, kann DBConnection sowohl den direkten Verbindungsaufbau mit JDBC versuchen, als auch über JNDI eine Datenquelle (datasource) suchen (lookup) und diese zum Erzeugen eines Verbindungsobjekts nutzen. So können die Komponenten wahlweise in einer Standalone-Applikation ohne Namensdienst oder auch innerhalb einer Applikationsservers oder Webcontainers eingesetzt werden.

Um die Häufigkeit des kostspieligen Neuerstellens von Verbindungen zu minimieren, bietet sich die Verwendung von *Connection Pooling* an. Verwendet man Datasources für den Verbindungsaufbau, verwaltet entweder der JDBC-Treiber der jeweiligen Datenbank oder der Applikationsserver einen Pool von Verbindungsobjekten. Der Anwendungsentwickler muss nur sicherstellen, benutzte Verbindungen wieder mit der close-Methode freizugeben. Statt die Objekte der garbage collection zu überlassen, werden sie in den Pool freier Verbindungen zurückgegeben.

Um von den Fähigkeiten bezüglich Connection Pooling von Applikationsserver und Datenbanktreiber unabhängig zu sein, verwaltet DBConnection zusätzlich einen internen Pool von Verbindungen. Die Konfigurationsvariable `connection_pool_maximum_size` steuert die Größe dieses Verbindungspools. Durch Setzen auf 0 kann das Connection Pooling seitens der DBConnection deaktiviert werden. Bei Anforderung von Verbindungen werden diese dann direkt neu erzeugt beziehungsweise ein Applikationsserver- oder JDBC-Treiberspezifisches Pooling verwendet.

ExperimentLoader

Diese Klasse ermöglicht das Laden einer Liste aller Experimente eines Benutzers oder aller für einen Benutzer im Rahmen seiner Rechte zugänglichen Experimente. Zusätzlich bietet es eine Methode für den Check-Out eines Experiments. Diese Methode bedient sich dazu anderer Methoden zum Laden der Bausteininstanzen des Experiments und markiert ein Experiment als „checked-out“, wie es in Kapitel 3.3.5 beschrieben wurde.

ExperimentWriter

Um Experimente in die Datenbank einzubringen, bietet der `ExperimentWriter` zwei Methoden an: `checkInExperiment` speichert ein neues Experiment in der Datenbank oder überschreibt ein bestehendes. Ist ein Überschreiben nicht möglich oder gewünscht, kann mit `checkInNewVersion` eine neue Version erstellt werden. Um Änderungen an einem Experiment, das als „checked-out“ markiert ist, zu verwerfen und es wieder freizugeben, wird die Methode `releaseExperiment` benutzt.

ModuleInstanceLoader

Das Laden der Bausteininstanzen wurde an diese Klasse delegiert. Bausteininstanzen können in zwei Situationen vorkommen: als Bausteine eines Experimentaufbaus und als Komponenten eines komplexen Bausteins. Die beiden Methoden

`loadModuleInstancesForExperiment` und `loadSubModulesForModuleType` sind für das Retrieval von Variante, Typ, Position und Orientierung zuständig. Für das Laden der weiteren Bestandteile einer Bausteininstanz wie Properties, Solids, Mutatoren und Constraints werden auch hier Hilfsmethoden verwendet.

ModuleInstanceWriter

Während das Retrieval einer Bausteininstanz wegen ihrer zahlreichen Bestandteile vergleichsweise aufwendig ist, ist das Speichern wesentlich einfacher: Es genügt, die Variante und damit die Werte der Eigenschaften der Instanz sowie ihre Position und Orientierung zu speichern. Das Schreiben der geänderten Eigenschaftswerte wird an den PropertyWriter delegiert. Da sich alle anderen Elemente einer Bausteininstanz entweder direkt aus dem Bausteintyp oder über Mutatoren aus den Eigenschaften ergeben, müssen Solids, Subkomponenten und andere Bestandteile nicht geschrieben werden.

PropertyLoader

Die Eigenschaften werden als zentraler und zugleich sehr vielfältiger Bestandteil der Bausteine vom `PropertyLoader` geladen. Die Klasse bietet Methoden für das Laden der Eigenschaften für alle Klassen von Anwendungsobjekten an, die über Eigenschaften verfügen können und daher das PropertyContainer-Interface implementieren. Für Kategorien und Bausteintypen werden die Eigenschaften nur mit Name und Typ geladen, für Bausteinvarianten und Bausteininstanzen kommt der jeweilige Wert hinzu. Mit Standard-Eigenschaftstypen (Integer, Double, String) kann der PropertyLoader direkt umgehen. Für komplexe Eigenschaftstypen wird das Laden und Erzeugen der Property-Objekte an die Klasse *ComplexPropertyFactory* delegiert

PropertyWriter

Der `PropertyWriter` bietet zwei Methoden zum Schreiben von Eigenschaften an. `insertPropertyValue` erzeugt bei einer neuen Bausteininstanz die erforderlichen Werte-Einträge, `updatePropertyValue` aktualisiert sie bei bereits in der Datenbank vorhandenen Bausteininstanzen.

SolidLoader

Das Laden der Solids erfordert eine Rekonstruktion der in der Datenbank in Winged-Edge-Darstellung gespeicherten Topologie und Geometrie der Körper. Nach Laden von Position und Orientierung des Solids sowie aller Eckpunkte müssen die Kantenzyklen aller Flächen rekonstruiert werden. Dies könnte in der Anwendung geschehen, indem die Kanten-Tupel einzeln geladen und die Topologie, also die Verbindung der Kanten zu Zyklen, von der Datenmodellschicht wiederhergestellt wird. Diese Lösung ist aufwendig, mit den Mitteln von SQL92 jedoch die einzige Realisierungsmöglichkeit, da eine Verknüpfung der Kanten eines Zyklus einen Join mit einer vorher nicht bekannten Anzahl von Referenzen der Edge-Relation in der FROM-Klausel erfordern würde. Zwar könnte man durch zusätzliches Speichern der Kantenzahl pro Zyklus ein jeweils geeignetes Statement generieren, doch auch diese Lösung erscheint wenig elegant. Ein Mechanismus von SQL:1999 bietet jedoch eine Alternative: Durch Verwendung eines rekursiven SQL-Statements kann die Topologie bereits

von der Datenbank rekonstruiert werden. Dazu werden für jede Fläche eines Solids zunächst die Referenzkanten aller sie begrenzenden Kantenzyklen und die Art des Kantenzyklus (inner oder outer loop) ermittelt. Mit diesen Informationen kann dann der Zyklus mit einem rekursiven SQL-Statement rekonstruiert werden, das in Beispiel 8 gezeigt wird.

Beispiel 8 Rekursives SQL-Statement zur Rekonstruktion von Kantenzyklen

```
WITH rek (id, fromid, toid, left, right, leftLoopPred, leftLoopSucc,
rightLoopPred, rightLoopSucc, count) AS
(SELECT id, fromid, toid, left, right, leftLoopPred, leftLoopSucc,
   rightLoopPred, rightLoopSucc, 1
  FROM edge e
  WHERE ( left = :currentFaceId OR right = :currentFaceId)
  AND id = :currentLoopStartEdge
  AND moduletypeid = :moduleTypeId AND solidid = :solidId
 UNION ALL
  SELECT e.id, e.fromid, e.toid ,e.left, e.right, e.leftLoopPred,
   e.leftLoopSucc, e.rightLoopPred, e.rightLoopSucc, count+1
  FROM rek r, edge e
  WHERE ((r.leftloopsucc= e.id AND r.left = :currentFaceId)
  OR (r.rightloopsucc = e.id AND r.right = :currentFaceId))
  AND e.moduletypeid = :moduleTypeId AND e.solidid = :solidId
  AND NOT e.id = :currentLoopStartEdge AND count < :edgeCount
) SELECT id,  fromid, toid, left, right FROM rek
```

Die SELECT-Anweisung oberhalb des UNION ALL-Schlüsselworts dient zum Initialisieren der Rekursion. Sie nimmt die Referenz- oder Startkante des Zyklus in die temporäre Tabelle rek auf. Kanten sind existenzabhängig von ihrem Solid und dieser wiederum von seinem ModuleType. Daher besteht der Primärschlüssel einer Kante aus Kanten-ID, Solid-ID und ModuleType-ID. Die SELECT-Anweisung nach UNION ALL führt nun die eigentliche Rekursion durch, indem sie jeweils die Kanten hinzufügt, die Nachfolger der bereits in der Tabelle rek vorhandenen Kanten sind. UNION ALL stellt dabei sicher, dass jede Kante nur einmal in rek aufgenommen wird. Die Rekursion läuft so lange, bis bei einem Rekursionszyklus keine neue Kante mehr aufgenommen wird. Dies tritt dann ein, wenn die Abbruchbedingungen eintritt, also die Startkante des Zyklus wieder erreicht ist. Um sicherzustellen, dass es bei fehlerhaften Topologieinformationen nicht zu einer Endlos-Rekursion kommt, weil die Startkante nicht wieder erreicht wird, kann zusätzlich für jeden Zyklus die Anzahl seiner Kanten gespeichert werden. Zählt man dann bei der Rekursion wie im Beispiel gezeigt die Anzahl der bereits ermittelten Kanten mit, kann auf Basis der Kantenzahl eine zusätzliche Abbruchbedingung zur Sicherheit integriert werden. Dies ist besonders deshalb empfehlenswert, weil IBM DB2 eine Endlosrekursion nicht erkennt und sich diese auch nicht mehr kontrolliert abbrechen lässt.

Das obige Beispiel zeigt die Navigation entlang eines outer loops, bei dem immer der Referenz auf die Nachfolgerkante gefolgt wird (leftloopsucc beziehungsweise rightloopsucc). Für

inner-loops muss dagegen der Referenz auf die Vorgängerkante (leftlooppred beziehungsweise rightlooppred) gefolgt werden.

Da inzwischen alle großen objekt-relationalen Datenbanksysteme rekursive Statements unterstützen, schränkt ihre Verwendung die Portabilität kaum ein. Leichte Variationen der genauen Syntax machen lediglich eine Anpassung der Statements erforderlich. [Tü2003] gibt eine Erläuterung zur Verwendung von Rekursion mit SQL und beschreibt die Unterschiede bei der Nutzung mit Oracle, IBM DB2 und Informix.

LibraryAccess

Zugang zur Bauteilebibliothek ermöglich die Klasse `LibraryAccess`. Methoden für das Laden der Top-Level-Kategorien ermöglichen den Einstieg in die Bausteinhierarchie. Weitere Methoden ermöglichen das Laden aller Unterkategorien beziehungsweise Bausteintypen in einer Kategorie und das Laden von Bausteinvarianten. Alle Ergebnisse werden in Form von Bibliotheksobjekten zurückgeliefert. Zusätzlich wurden exemplarisch einige Methoden für die Manipulation der Bibliothek implementiert, zum Beispiel für das Umbenennen, Löschen und Erstellen von Kategorien.

Eine weitere Methode ermöglicht die Suche innerhalb der Kategorienhierarchie der Bausteinbibliothek durch Angabe von Suchbedingungen auf den Eigenschaften einer Kategorie. Die Methode `searchHierarchyForVariants` nimmt als Parameter die Kategorie, in der gesucht werden soll entgegen, sowie Arrays mit den Elementen der Suchbedingungen, also den Eigenschaften, den Vergleichsoperatoren, den Werten mit Einheiten sowie den boolschen Operatoren, welche die einzelnen Bedingungen verknüpfen. Aus diesen wird dann eine entsprechende Anfrage generiert. Beispiel 9 zeigt eine solche Anfrage für die Suche mit zwei Bedingungen. Zeichen, deren Wert von der jeweiligen Suchanfrage abhängt, sind hervorgehoben.

Im ersten Teil der Anfrage werden mit einem rekursiven Statement die Kategorie, von der die Suche ausgeht (im Beispiel ist dies die Kategorie mit dem Primärschlüssel 23) sowie alle ihre Unterkategorien in der temporären Tabelle subcats zusammengefasst. Mit dieser Tabelle wird dann ein Join mit ModuleType und ModuleVariant durchgeführt, sodass man alle Varianten von Bausteintypen erhält, die in einer der Kategorien liegen. Durch die Bedingung, dass der Name der Variante nicht NULL sein darf, werden die anonymen Varianten, die zur Darstellung von Bausteininstanzen genutzt werden, herausgefiltert. Auf den verbleibenden benannten und damit der Bausteinbibliothek zugehörigen Varianten werden nun die vom Benutzer formulierten Bedingungen geprüft. Die erste Subquery selektiert alle Varianten, deren Eigenschaft 4 einen Wert kleiner 42.5, ausgedrückt in der Einheit 5 des Typs dieser Eigenschaft hat. Die benutzerdefinierte Funktion `convertDoubleValue` wurde speziell zur Realisierung einheitenunabhängiger Suchanfragen implementiert und rechnet sowohl den Wert der Eigenschaft als auch den Suchwert 42.5 in die Standardeinheit (dritter Parameter = -1) um. Die zweite Bedingung ist durch ein „oder“ mit der ersten verknüpft und wählt jene Varianten, in deren (String-)Property mit der ID 8 der Teilstring „vierkant“ vorkommt. Die DB2-Standard-UDF `lcase` konvertiert die Strings in Kleinbuchstaben, sodass der Vergleich unabhängig von Groß- und Kleinschreibung wird.

Beispiel 9 Anfrage für die Suche in den Kategorien

```
WITH subcats (id, name, supercategoryid) AS (
       SELECT id, name, supercategoryid
       FROM category root
       WHERE root.id = 23
       UNION ALL
       SELECT child.id, child.name, child.supercategoryid
       FROM category child, subcats s
       WHERE child.supercategoryid = s.id
)
SELECT c.id, mt.id, mt.name, mv.id, mv.name
FROM subcats c, moduletype mt, modulevariant mv
WHERE c.id=mt.categoryid AND mt.id = mv.moduletypeid
AND NOT mv.name IS NULL AND (
       EXISTS (SELECT * FROM var_sets_prop_to_val v
                     WHERE v.moduletypeid = mt.id
                     AND v.modulevariantid= mv.id
                     AND v.modulepropertyid= 4
                     AND library.convertdoublevalue
                      (v.doublevalue,v.unitid,-1) <
                      library.convertdoublevalue(42.5, 5, -1)
       ) OR (
       EXISTS (SELECT * FROM var_sets_prop_to_val v
                     WHERE v.moduletypeid = mt.id
                     AND v.modulevariantid= mv.id
                     AND v.modulepropertyid= 8
                     AND sysfun.lcase(v.stringvalue) like
                     sysfun.lcase('%vierkant%')
)) ORDER BY mt.id
```

Für jede weitere Bedingungen wird ein zusätzlicher EXISTS-Ausdruck mit Subquery in die Anfrage integriert. Zuvor sollte eine Optimierung stattfinden, sodass Bedingungen auf der gleichen Eigenschaft in einer Subquery unter Beachtung der Operator-Prioritäten zusammengefasst werden.

Exceptions

Zur Behandlung und Weiterleitungen von Fehlern beim Datenbankzugriff steht eine Hierarchie von Ausnahmen zur Verfügung. `DBException` ist die gemeinsame Superklasse und dient zur Signalisierung allgemeiner Ausnahmen. Spezielle Fehlertypen werden durch Subklassen von DBException signalisiert, zum Beispiel die Meldung, dass ein Experiment bereits als „checked-out“ markiert ist oder ein zu löschendes Objekt anderweitig benutzt wird und daher nicht gelöscht werden kann.

5.3.3 Schnittstellen der Datenmodellschicht

Von den drei in Kapitel 4.6 vorgestellten Teilschnittstellen der Datenmodellschicht wurden zwei im Prototypen umgesetzt, die Experimentverwaltung und die Bibliotheksverwaltung. Sie sind in Form von zwei Klassen `ExperimentAccess` und `LibraryAccess` im Package `vlab.access` realisiert, die alle essentiellen Funktionen als statische Methoden anbieten. Als Parameter und Rückgabewerte fungieren die in Kapitel 5.3.1 beschriebenen Anwendungsobjekte.

5.3.4 Realisierung der Datenmodellschicht als Web Service

Die zuvor vorgestellte Realisierung der Datenmodellschicht in Form einer Java-Bibliothek bietet maximalen Komfort und Leistung bei der Realisierung der Client-Komponenten des virtuellen Labors mit Java. Sollen jedoch auch Clients zum Einsatz kommen, die in anderen Sprachen entwickelt werden, muss eine der in Kapitel 4.1 vorgestellten Alternativen für den Zugang zur Datenmodellschicht genutzt werden. Um im Zuge der prototypischen Realisierung des virtuellen Labors Verfahrenstechnik die Tauglichkeit von Web Services als Middlewaretechnologie zu untersuchen, wurde dazu exemplarisch eine Teil-Schnittstelle der Datenmodellschicht in Form eines Web Service realisiert.

Als Ziel der Umstellung wurden die Funktionen für den Zugriff auf die Bausteinbibliothek in der Klasse LibraryAccess ausgewählt. Um einen einfachen Wechsel zwischen Web Service-Nutzung und direktem DB-Zugriff zu erleichtern, erlaubt das Setzen des Konfigurationsparameter `library_use_web_service` die schnelle Umstellung. Wird er auf `false` gesetzt, führt LibraryAccess wie bisher einen direkten Zugriff auf die Methoden des db-Package durch. Wurde er auf `true` gesetzt, werden die Anfragen über mit dem wscompile-Tool des Java Web Service Developer Pack erstellte statische Stubs an eine konfigurierbare Endpunkt-URL weitergeleitet (Parameter: `library_web_service_endpoint_url`) .

Generierung und Installation des Web Service

Der Web Service ist in Form einer J2EE-Web-Applikation (ein WAR-Archiv mit speziellem Aufbau) realisiert. Hier soll kurz die erforderliche Vorgehensweise beschrieben werden.

Das Tool wscompile benötigt Informationen über den zu erstellenden Web Service in Form einer (XML-)Konfigurationsdatei. Diese kann auf eine WSDL-Datei verweisen, die dann als Basis für die Generierung der Stub-, Tie- und sonstiger Hilfsklassen dient und zum Beispiel aus einer UDDI-Registry oder direkt vom Anbieter des Web Service stammt. Im Fall des Web Service für den Bibliothekszugriff muss die WSDL-Datei jedoch selbst erstellt werden. Um diesen aufwendigen und fehlerträchtigen Vorgang nicht manuell vornehmen zu müssen, kann in der Konfigurationsdatei alternativ auch auf ein Java Interface verwiesen werden, das als Grundlage für die Erstellung der Artefakte und einer WSDL-Datei fungiert. Das Interface muss dazu von `java.rmi.Remote` erben, und alle Methoden müssen die `java.rmi.RemoteException` in ihrer `throws`-Klausel deklarieren. Zusätzlich muss man die Klasse angegeben, die den Service implementiert.

Zunächst wurde die Schnittstelle als Java-Interface `LibraryAccessIF` beschrieben. Anschließend wurde eine Klasse `LibraryAccessImpl` erstellt, die dieses Interface implementiert und die Methodenaufrufe ähnlich wie die Klasse LibraryAccess auf die der Datenmodellschicht abbildet.

Diese beiden Klassen wurden in die Konfigurationsdatei config.xml eingetragen, und damit wscompile mit der Option –gen:client ausgeführt. Die so erstellten clientseitigen Klassen wurden in die Package-Struktur des Prototypen kopiert und können so von LibraryAccess genutzt werden.

Zur Vorbereitung des Web Service wurden die für den Server benötigten Klassen (LibraryAccessIF und LibraryAccessImpl) gemäß der Konventionen für J2EE-Web-Anwendungen in das Verzeichnis WEB-INF/classes kopiert. Damit die Komponenten der Datenmodellschicht der Web-Anwendung ebenfalls zur Verfügung stehen, wurden sie als JAR-Archiv gepackt und zusammen mit dem DB2-JDBC-Treiber in das WEB-INF/lib-Verzeichnis des WAR-Archivs kopiert. Direkt im Verzeichnis WEB-INF findet sich der (weitgehend leere) Deployment-Deskriptor der Web-Applikation und die Datei jaxrpc-ri.xml, die Informationen über den Web Service enthält, zum Beispiel seinen Namen, seine URL relativ zum Stammverzeichnis, eine Beschreibung und die voll-qualifizierten Namen der Interface- und Implementierungsklassen. WEB-INF und alle seine Unterverzeichnisse können nun zu einer „transportablen“ (portable) WAR-Datei gepackt werden. Diese kann noch nicht direkt in einem Web-Container installiert werden. Dazu muss das wsdeploy-Tool gestartet werden. Es nimmt als Eingaben die portable WAR-Datei und die dort vorhandene jaxrpc-ri.xml. Daraus werden die serverseitigen Artefakte (Ties und Hilfsklassen) und weitere Dateien mit Informationen für den Web-Container generiert. Als Ergebnis entsteht eine *deployable* WAR-Datei, die in einem Web-Container installiert werden kann. Bei der Generierung trat ein Fehler von wsdeploy zutage: Die Dateinamen einzelner Classfiles wurden mit falscher Groß- und Kleinschreibung erzeugt, sodass der Classloader des Web-Containers sie nicht finden kann. Durch manuelles Umbenennen der Dateien lies sich das Problem jedoch beheben.

Als Web-Container wurde der beim JWSDP mitgelieferte Apache Tomcat genutzt. Das Deployment kann durch Kopieren des Web-Archivs in das Unterverzeichnis webapp des Tomcat-Installationsverzeichnisses und anschließendes Neustarten von Tomcat erfolgen. Alternativ kann die Manager-Applikation von Tomcat, eine vorinstallierte Web-Applikation, genutzt werden, die ein Entfernen und erneutes Installieren von Web-Applikationen im laufenden Betrieb ermöglicht.

Um alle beschriebenen Schritte nicht bei jeder Änderung der Applikation erneut manuell durchführen zu müssen, kann das Tool *Ant* benutzt werden. In seiner Funktionsweise vergleichbar mit *make*, können in einer XML-Datei sogenannte *targets* definiert werden, die verschiedene Operationen wie das Kompilieren eines Projekts, das Kopieren oder Packen von Dateien und auch das Aufrufen beliebiger Kommandos wie wscompile durchführen. Durch Aufruf des Tools mit dem Target-Namen als Parameter wird die Target-Operation ausgeführt. Durch Angabe von Abhängigkeiten zwischen den Targets können auch komplexe Sequenzen von Befehlen automatisiert werden. Tomcat bietet selbst Ant-Targets für das Verwalten von Web-Applikation über die Manager-Applikation an, sodass auch das Deployment mit Ant vereinfacht werden kann.

Erzeugung weiterer Serialisierungsklassen

wscompile erstellt für alle in den Signaturen der Methoden der Web Service-Schnittstelle als Parameter und Rückgabewerte vorkommenden Klassen Hilfsklassen zur Serialisierung. Dies kann jedoch zu Problemen bei der Verwendung von Superklassen oder Collections in der Schnittstelle führen. Für diese werden zwar Serialisierer erstellt, nicht jedoch für die auch an dieser Stelle einsetzbaren Subklassen beziehungsweise für die Elemente der Collections. Daher können in der Konfigurationsdatei für wscompile weitere Klassen angegeben werden, für die dann ebenfalls Serialisierer generiert werden.

5.3.5 Realisierung der Datenmodellschicht mit XML

Statt einer Realisierung mit Web Services kann auch der direkte Austausch von anwendungsspezifischen XML-Dokumenten eine mögliche Alternative darstellen. Die Entwicklung einer Adaptionsschicht, die von diesen Dokumenten auf die Methoden der Datenmodellschicht abbildet, ist leicht möglich. Theoretischen Vorteilen durch die Umgehung einzelner Beschränkungen von Web Services steht jedoch ein erhöhter Aufwand für die Entwicklung einer funktional letztlich gleichwertigen Schnittstelle entgegen. Dieser Ansatz wurde daher nicht weiter verfolgt.

Eine mögliche Alternative zur Implementierung einer Zwischenschicht ist die direkte Nutzung von XML-Dokumenten mit dem DB2 XML-Extender. Wie bereits in Abschnitt 4.2.7.2 beschrieben, bietet er neben der Speicherung von XML-Dokumenten als *XML Columns* auch die Erstellung von XML-Dokumenten aus relationalen Daten beziehungsweise die Zerlegung von XML-Dokumenten und die Speicherung ihres Inhalts in Relationen an, ein Verfahren, das als *XML Collections* bezeichnet wird.

Sowohl die Zerlegung als auch die Erzeugung von XML-Dokumenten erfolgt über Stored Procedures. Diese existieren in zwei Varianten: `dxxGenXML` und `dxxShredXML` arbeiten auch ohne ein zuvor aktivierte XML Collection und nehmen eine die Abbildung beschreibende DAD-Datei als Parameter entgegen. Sie eignen sich besonders für eher seltene Nutzung. Die von dxxGenXML erzeugten Dokumente werden in einer Tabelle abgelegt, auf welche die Anwendung dann zugreifen kann. Für häufige Nutzung sind `dxxRetrieveXML` und `dxxInsertXML` vorgesehen. Sie Arbeiten auf aktivierten XML Collections, die DAD-Datei ist in diesem Fall in einer Hilfstabelle des XML-Extenders gespeichert. Sie sind funktional gleichwertig zu den beiden anderen Prozeduren. Zur Vermeidung des Umwegs über die Ergebnistabellen existieren die Prozeduren `dxxGenXMLCLOB` und `dxxRetrieveXMLCLOB`. Sie liefern das erste Ergebnisdokument in Form eines CLOB-Output-Parameters zurück, der Zugriff auf weitere Dokumente ist damit jedoch nicht möglich.

Es stehen zwei prinzipielle Verfahren zur Definition der Abbildung zur Verfügung. *SQL Mapping* erfordert die Angabe eines einzelnen SQL-Statements, das eine beliebige Zahl von Tabellen mit Joins verknüpft. Neben der Schwierigkeit, alle Daten für das XML-Dokument mit einer Anfrage zu extrahieren, ergibt sich zusätzlich das Problem, dass die Reihenfolge der Joins exakt der hierarchischen Darstellung im XML-Dokument entsprechen muss. Zu-

dem kann SQL Mapping nur für die Komposition, also die Erstellung von XML-Dokumenten aus relational vorliegenden Daten genutzt werden.

Relational Database (RDB) Mapping ist ein flexiblerer Ansatz. Er ermöglicht neben der XML-Komposition auch die Zerlegung und anschließende Speicherung von XML-Dokumenten. Die DAD-Datei beschreibt hier die Position von Elementinhalten und Attributwerten im DB-Schema, durch Angabe von Filterbedingungen kann die Erstellung des Dokuments auf eine Untermenge der Tupel der beteiligten Relationen beschränkt werden.

Bei beiden Abbildungsverfahren können das SQL-Statement beziehungsweise die Filterbedingungen durch Nutzung eines Override-Mechanismus in den Generierungsfunktionen angepasst werden.

Bei der exemplarischen Implementierung einzelner Funktionen der Datenmodellschicht mit den XML Collections wurden die starken Einschränkungen dieses Verfahrens deutlich. Insbesondere können komplexere Anfragen wie die SQL-Rekursion zur Auflösung der Kantenzyklen nicht genutzt werden. Die Mächtigkeit dieses Mechanismus genügt daher häufig nicht für die Durchführung der Abbildung vom (objekt-)relationalen Datenmodell auf ein direkt in der Anwendung einsetzbares Datenmodell. Unabhängig von diesen Einschränkungen bleiben weitere Probleme: Der Zugriff aus der Anwendung erfolgt über eine sprachspezifische SQL-Schnittstelle der jeweiligen Programmiersprache. Die Prozeduren des XML-Extender müssen aufwendig parametrisiert und durch die Override-Funktion an die jeweilige Anfrage angepasst werden. Dazu ist sehr spezielles Wissen über den Aufbau des Datenbankschemas und die Funktionsweise des XML-Extenders erforderlich, wodurch das eigentliche Ziel, die gewünschte Abstraktion an der Schnittstelle von den Strukturen der Datenhaltungsschicht, nicht erreicht wird. Auch können so keine Zusicherungen an die Einhaltung der vorgesehenen Zugriffsschutzmechanismen gemacht werden. Es wäre also in jedem Fall eine Zwischenschicht vorzusehen, auf die dann wieder zum Beispiel über Web Services oder RMI zugegriffen würde.

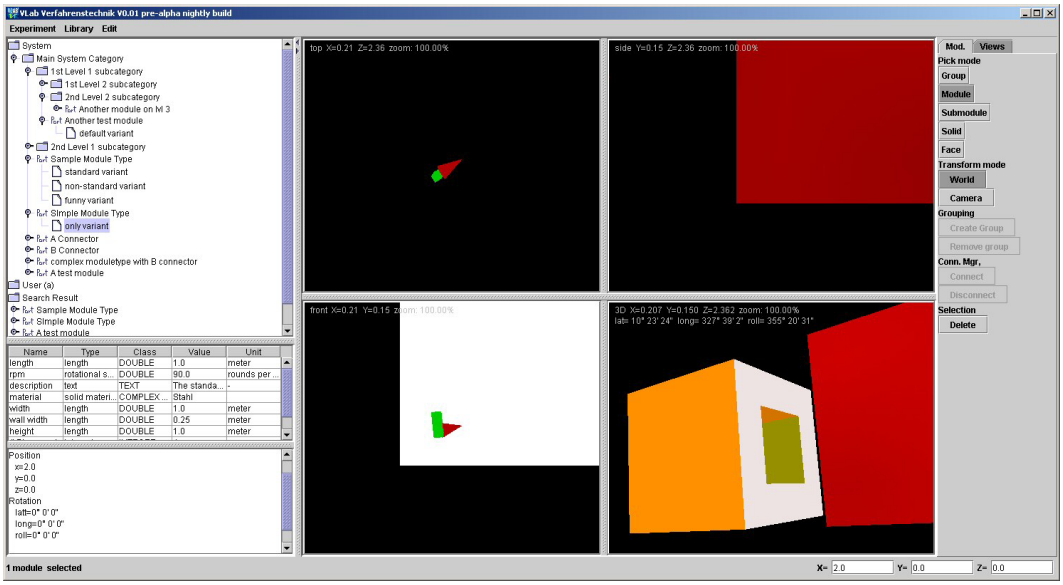
Wenn sich die XML-Collections in ihrer jetzigen Form auch nicht zur direkten Implementierung einer XML-Schnittstelle eignen, können sie doch als Grundlage für den Export von Daten aus dem Repository untersucht werden.

5.4 Benutzeroberfläche

5.4.1 Wichtige Komponenten

Um eine Vorstellung von der Funktionsweise der Benutzerschnittstelle des Prototyps zu geben, werden die wichtigsten Komponenten kurz vorgestellt und interessante Aspekte ihrer Realisierung beschrieben. Abbildung 45 zeigt das Hauptfenster des Prototypen, auf dem die wichtigsten Bedienelemente zu sehen sind.

Abbildung 45 Hauptfenster des Prototyps



5.4.1.1 Bedienelemente

LibraryTree

Der `LibraryTree` zeigt dem Laborbenutzer alle im Rahmen seiner Rechte zugänglichen Bausteintypen der Bausteinbibliothek. Wie die Bezeichnung andeutet, erbt LibraryTree von `JTree` und stellt die Kategorienhierarchie der Bausteine durch einen Baum dar. JTree wurde dazu in verschiedener Weise modifiziert: Neben domänenspezifischen Icons für die verschiedenen Knoten des Baums (Kategorie, Baustein, Variante) wurde ein Popup-Menü realisiert, das je nach selektiertem Knoten die passenden Operationen anbietet: Für Kategorien ist das Erzeugen einer neuen Subkategorie oder das Löschen der selektierten Kategorie möglich. Dabei wird der Benutzer gewarnt, wenn die zu löschende Kategorie noch Elemente wie Subkategorien oder Bausteine enthält. Für Bausteintypen kann eine neue Variante erstellt werden, oder sie können wie Kategorien mit vorheriger Warnung gelöscht werden. Varianten können in das aktuelle Experiment eingefügt oder gelöscht werden. Bestehen entsprechende Rechte, kann der Benutzer alle Knoten umbenennen. Als weitere Operation kommt später das Verschieben von Teilbäumen in Betracht. Hier muss eine Prüfung erfolgen, ob das verschobene Objekt an seiner neuen Position eingefügt werden kann.

Wie jedes Swing-Element folgt auch ein JTree und seine Subklasse LibraryTree dem Model-View-Controller-Pattern. Als Model dienen Klassen, die das Interface `TreeModel` implementieren. Das TreeModel erlaubt den Zugang zu den Knoten, Klassen des TreeNode-Interfaces, welche die eigentliche Baumstruktur repräsentieren. Für die Zwecke des Prototypen genügt hier das bei Swing mitgelieferte `DefaultTreeModel`, für die Knoten wurden jedoch angepasste Klassen implementiert: Die Objekte der Datenmodellschicht, `Category`, `LibraryModuleType` und `LibraryModuleVariant`, erben von der Klasse `DefaultMutableTreeNode`, der Standardimplementierung für TreeNodes, und können so direkt im LibraryTree genutzt werden. Ist ein derart direkter Zuschnitt der Datenmodell-

schicht für den Prototypen nicht erwünscht, können alternativ Wrapper-Klassen genutzt werden.

PropertyTable

Die `PropertyTable` erbt von der Swing-Klasse `JTable` und ermöglicht die Darstellung von Eigenschaften. Wegen der großen Bedeutung des Eigenschaftskonzepts als Mittel zur Erlangung von Informationen über Bausteine und zu ihrer Manipulation kommt der PropertyTable eine zentrale Rolle in der GUI zu.

Dazu wurde ein proprietäre Subklasse eines `TableModel` entwickelt, das `GenericPropertyTableModel`. Sie fungiert als Adapter, der jede Klasse, die das Interface PropertyContainer implementiert, entgegennehmen kann und ihre Methoden auf die des TableModel-Interfaces abbildet. So werden die einzelnen Eigenschaften auf die Tabellenzeilen und Felder der Eigenschaften auf Tabellenspalten umgesetzt. Die PropertyTable lässt sich sowohl dazu einsetzen, die Eigenschaften von Objekten der Bausteinbibliothek zu zeigen als auch die der aktuell im Experiment selektierten Bausteininstanz.

Je nach Art des PropertyContainer variiert das Aussehen der Tabelle. Bei einem PropertyContainer, dessen Eigenschaften über keine Werte verfügen, wie eine Kategorie oder ein Bausteintyp, werden nur Name und Typ der Eigenschaft angezeigt. Handelt es sich dagegen um eine Klasse, deren Eigenschaften über Werte verfügen, zum Beispiel eine Bausteinvariante oder eine Bausteininstanz, so werden diese einschließlich eventuell vorhandener Einheiten angezeigt und sind direkt in der Tabelle editierbar, wenn dies für die einzelne Eigenschaft zulässig ist. Je nach Typ des Bausteins erfolgt das Ändern des Eigenschaftswertes direkt innerhalb der Zelle der Tabelle oder bei komplexen Eigenschaften durch Auswahl eines Wertes in einem eigenen Fenster, dem ComplexValueWindow. Schließt der Benutzer die Eingabe oder Auswahl des neuen Wertes ab, wird dieser im Property-Objekt gesetzt. Dies veranlasst nun eine Prüfung aller Constraints, die für diese Eigenschaft global oder innerhalb dieses PropertyContainers gültig sind. Werden eines oder mehrere der Constraints verletzt, wird dem Benutzer ein Fenster mit einer detaillierten Fehlermeldung präsentiert. Der Benutzer kann das Editieren dann entweder abbrechen oder einen gültigen Eigenschaftswert eingeben. Erfüllt der neue Wert schließlich alle Bedingungen, wird er in der Property gesetzt. Handelt es sich beim aktuellen PropertyContainer um eine Bausteininstanz, werden nun alle von der geänderten Eigenschaft abhängigen Mutatoren ausgelöst, sofern ihre Bedingungen erfüllt sind. Die Änderungen am Baustein werden so sofort sichtbar. Bei numerischen Eigenschaften mit Einheiten kann mit einer JComboBox eine der Einheiten ausgewählt werden. Der Wert wird dann automatisch in die neue Einheit konvertiert.

MultiPurposePanel

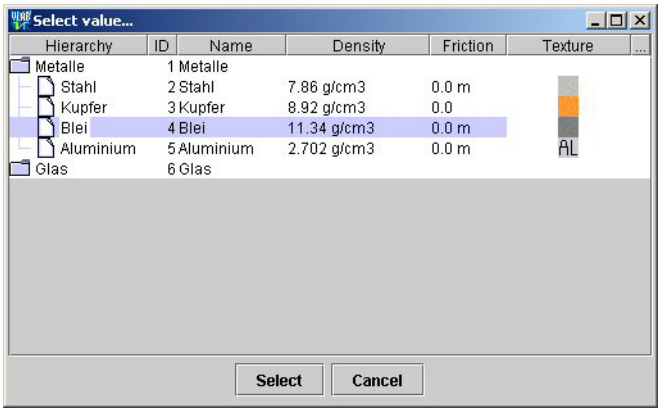
Das `MultiPurposePanel` dient vielen Komponenten der Bedienoberfläche zum Anzeigen von textuellen Informationen. So zeigt es Position und Orientierung eines ausgewählten Bausteins an oder die für eine selektierte Eigenschaft gültigen Constraints. Zudem kann das MultiPurposePanel auch als erweitertes Anzeigeelement und als Editor für Eigenschaftswerte fungieren. So ist der Platz für Werte innerhalb der PropertyTable auf eine Zeile mit geringer Breite begrenzt. Das Lesen und Editieren von längeren Werten, zum Beispiel ein be-

schreibender Text, wäre so sehr mühsam. Daher wird der selektierte Wert auch hier mit deutlich mehr Platz angezeigt und kann auch parallel zur Tabellenzelle editiert werden.

ComplexValueWindow

Für komplexe Eigenschaften, deren Werte nur aus einer vorgegebenen Menge ausgewählt werden können, ist ein Editieren direkt in der Tabelle zum Beispiel durch Auswahl in einer JComboBox wie bei den Einheiten oft nicht ausreichend. Insbesondere bei sehr vielen möglichen Werten oder zur Darstellung einer Wertehierarchie stößt dieser Ansatz schnell an seine Grenzen. Daher wird beim Editieren eines solchen komplexen Wertes ein neues Fenster geöffnet, das `ComplexValueWindow`. „Flache“ komplexe Typen werden hier mit ihren Attributen in einer Tabelle aufgelistet. Für hierarchisch organisierte komplexe Typen wurde ein neues hybrides Widget, die `TreeTable` implementiert. Sie setzt einen JTree in die erste Spalte einer JTable und erlaubt so zugleich das Aufklappen einzelner Knoten und die Anzeige der Attribute der Werte. Abbildung 46 zeigt das ComplexValueWindow am Beispiel des komplexen Eigenschaftstyps „SolidMaterial“, also Feststoffe, die zum Beispiel die Wände eines Bausteins bilden können.

Abbildung 46 ComplexValueWindow mit einer hierarchischen Eigenschaft

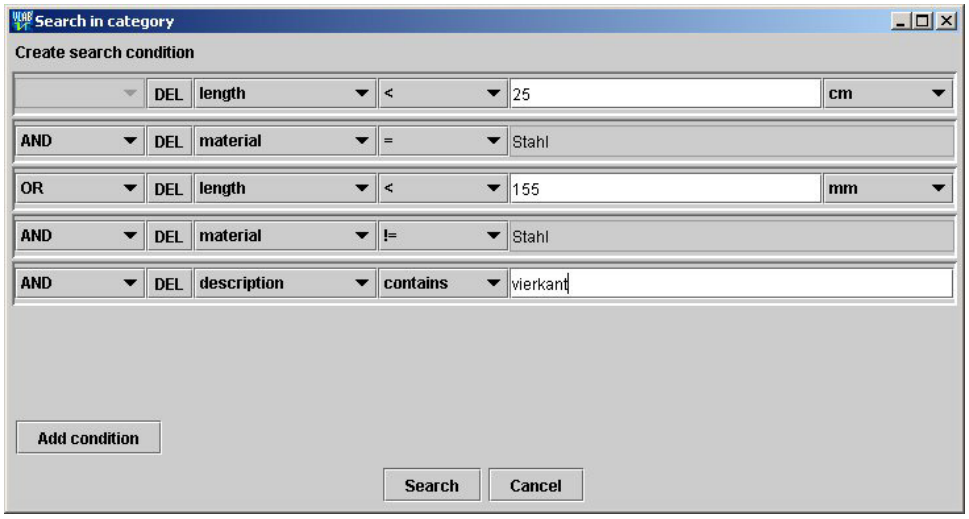


SearchInCategoryWindow

Neben der Auswahl von Bausteinen mittels Navigation durch die Kategorien der Bausteinbibliothek ist die Suche anhand von Bausteineigenschaften die wichtigste Zugangsmöglichkeit. Für jede Kategorie, die über Eigenschaften verfügt, kann eine Suche veranlasst werden. Das in Abbildung 47 gezeigte `SearchInCategoryWindow` wird geöffnet, und erlaubt das Zusammenstellen einer Suchbedingung aus einzelnen Bedingungen auf den Eigenschaften der Kategorie. Für jede Einzelbedingung kann man zunächst die Eigenschaft wählen, auf die sie sich bezieht, und dann je nach Typ dieser Eigenschaft den Vergleichsoperator selektieren und einen Wert eingeben beziehungsweise bei komplexen Eigenschaften einen Wert mit dem ComplexValueWindow auswählen. Für numerische Typen kann zusätzlich die Einheit ausgewählt werden, in welcher der Wert angegeben ist. Die Einzelbedingungen können mit AND beziehungsweise OR verknüpft werden. Die so zusammengestellte Bedingung wird dann an die in Abschnitt 5.3.2.2 vorgestellte Suchfunktion übergeben. Alle Bausteintypen, die über Varianten verfügen, welche die Bedingung erfüllen, werden anschließend als weite-

rer Top-Level-Knoten in den LibraryTree eingehängt und können dort normal genutzt werden.

Abbildung 47 Fenster für die Suche in der Kategorienhierarchie



Sidebar

In der Sidebar am rechten Rand des VLab-Fensters (Abbildung 45) sind grundlegende Funktionen zur Interaktion mit dem Experimentaufbau zusammengefasst. Eine JTabbedPane erlaubt das Umschalten zwischen mehreren Seiten über „Karteireiter“. Hier lassen sich zum Beispiel die einzelnen Sichtfenster auf ein Objekt zentrieren und der Auswahlmodus zwischen Fläche, Solid, Submodul, Modul und Gruppe umschalten. Weiterhin gibt es Funktionen zum Gruppieren von Bausteininstanzen sowie zum Verbinden von Instanzen, die über Verbindungselemente verfügen. Die Sidebar eignet sich für das Einbinden weiterer Interaktionsmöglichkeiten.

5.4.1.2 Darstellung des Experiments

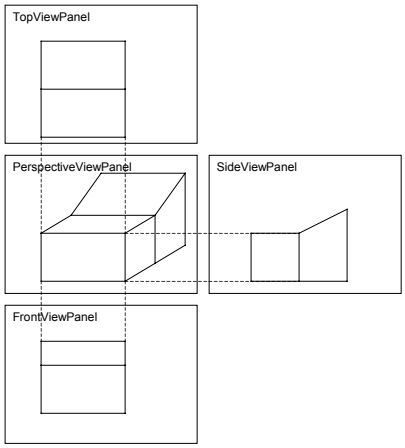
Die Realisierung der Visualisierung von Experimentaufbauten orientiert sich an der von gängigen CAD-Werkzeugen bekannten Oberfläche. Dominierendes Element der Benutzeroberfläche sind vier Fenster, die dem Benutzer graphisch den aktuell in Bearbeitung befindlichen Experimentaufbau zeigen. Sie sind in verschachtelte `JSplitPanes` eingebettet, sodass sich der Anteil der Fenster an der verfügbaren Fläche variieren lässt.

ParallelViewPanel, ParallelCamera

Drei der vier Fenster zeigen die Szene in Form einer Parallelprojektion. Oben links befindet sich das `TopViewPanel`, oben rechts das `SideViewPanel` und unten links das `FrontViewPanel`. Alle erben von `ParallelViewPanel` und letztlich von `JPanel`, einer Swing-Komponente zur Aufnahme anderer Komponenten. Jedes enthält ein Canvas3D-Objekt, entsprechend ihrem Namen zeigen sie eine Sicht auf das Experiment jeweils von oben (entgegen der Richtung der Y-Achse), von der Seite (entgegen der Richtung der X-

Achse) oder von vorne (entgegen der Richtung der Z-Achse) wie in Abbildung 48 dargestellt. Für jedes Panel existiert ein eigenes Objekt der Klasse `ParallelCamera`, die von der abstrakten `Camera`-Klasse erbt. Diese fasst zur einfacheren Handhabung die für die grundlegenden Kamerafunktionen benötigten Objekte zusammen: Wie bei der Vorstellung von Java 3D geschildert wird eine ViewPlatform als Blatt in den Szenegraphen eingefügt und repräsentiert dort die Position und Orientierung der Kamera. Ein View verbindet ViewPlatform und den zur Ausgabe genutzen Canvas3D. Die ViewPlatform selbst ist Kind einer TransformGroup, mit der die Orientierung und Position der Kamera gesteuert werden kann. Zusätzlich bietet Camera Methoden zur Kontrolle des Vergrößerungsfaktors (scale) der jeweiligen Sicht. Die konkrete Subklasse ParallelCamera aktiviert das Rendering mit Parallelprojektion und bietet Methoden zur Bewegung der Kamera, die entsprechend die Transformation der TransformGroup modifizieren. Je nach Perspektive kann die Kamera auf nur zwei zur Blickrichtung orthogonalen Achsen bewegt werden, die Kamera für das SideViewPanel (Blick parallel entlang der Y-Achse zum Ursprung) kann sich beispielsweise nur auf X- und Z-Achse bewegen. Die Steuerung der Kameraposition erfolgt mit in der Konfigurationsdatei frei wählbaren Tasten oder durch Gedrückthalten der rechten Maustaste und Bewegung der Maus im jeweiligen Fenster. Die Blickrichtung der Kamera ist fest.

Abbildung 48 Views auf den Experimentaufbau



PerspectiveViewPanel, PerspectiveCamera

Um dem Benutzer eine flexiblere Betrachtung des Experiments zur ermöglichen, zeigt das rechts unten angebrachte `PerspectiveViewPanel` das Bild einer frei in der Szene beweglichen Kamera (`PerspectiveCamera`), ebenfalls eine Subklasse von Camera. Die Methoden zur Bewegung der Kamera sind hier deutlich flexibler. Neben Translation auf allen drei Achsen beherrscht die Kamera auch Rotationsbewegungen: Rollen (*roll*, Rotation um die Z-Achse), Gieren (*yaw*, Rotation um die Y-Achse) sowie Neigen (*pitch*, Rotation um die X-Achse). Alle Bewegungen erfolgen dabei mit Bezug auf die Achsen des lokalen Koordinatensystems der Kamera. Die Kontrolle der Translation und Rotation erfolgt wie bei den anderen Views über konfigurierbare Tasten (Konfigurationsvariablen `camera_move_XYZ` und `camera_rotate_XYZ`), die Rotation selbst kann auch durch Ziehen der Maus bei gedrückter rechter Maustaste erfolgen. Horizontale Bewegungen der Maus drehen die Kamera um die Y-Achse, vertikale um die X-Achse. Durch Halten konfigurierbarer Tasten lassen sich

die Rotations- und Translationsbewegungen verlangsamen oder horizontale Mausbewegungen zum Rollen der Kamera nutzen. Über die Konfigurationsvariable `camera_rotation_direction_fps` kann bestimmt werden, ob sich die Kamera in Richtung der Mausbewegung dreht (`true`) beziehungsweise entgegen der Bewegungsrichtung (`false`). Durch Setzen der Konfigurationsvariable `camera_rotation_style_fps auf true` kann ein alternatives Bewegungsverhalten aktiviert werden, bei dem direkte und indirekte Rollbewegungen unterbunden werden. Die Bewegung ähnelt so der eines aufrecht durch das Experiment laufenden Benutzers, was die Orientierung erleichtert. Um dem Benutzer einen Anhaltspunkt über seine Position zu geben, kann die Kamera in den drei anderen Sichten als Pfeil sichtbar gemacht werden (Variable `camera_position_visible`). Mit einem Button in der Sidebar können die Parallelsichten zudem auf die Position der PerspectiveCamera zentriert werden.

Interaktion

Wenn auch der Großteil der Manipulationen von Bausteininstanzen im Experiment über andere GUI-Elemente wie die PropertyTable erfolgt, so muss der Nutzer jedoch mit den Views auswählen, welche Instanz modifiziert werden soll. Dazu kann er zunächst wie bei der Sidebar beschrieben den Selektionsmodus auswählen. Durch Klicken in einer der Sichten lässt sich dann das Objekt unter dem Mauszeiger selektieren. Dazu wird das Picking-System von Java 3D eingesetzt. Sollen mehrere Objekte selektiert werden, zum Beispiel um sie gemeinsam zu verschieben, kann dies durch Halten der Shift-Taste geschehen. Selektierte Objekte werden farblich hervorgehoben. Die Klasse `SelectionManager`, ein Singleton, verwaltet die selektierten Objekte. Das Positionieren einer selektierten Bausteininstanz kann wahlweise durch Direkteingabe der Koordinaten geschehen oder durch Verschieben und Drehen mit der Maus beziehungsweise mit konfigurierbaren Tasten (Konfigurationsvariablen `object_translate_XYZ`, `object_rotate_XYZ`). Hierbei lässt sich in der Sidebar wählen, ob die Bewegungen mit Bezug auf das Kamera- oder das Weltkoordinatensystem erfolgen sollen.

UniverseManager

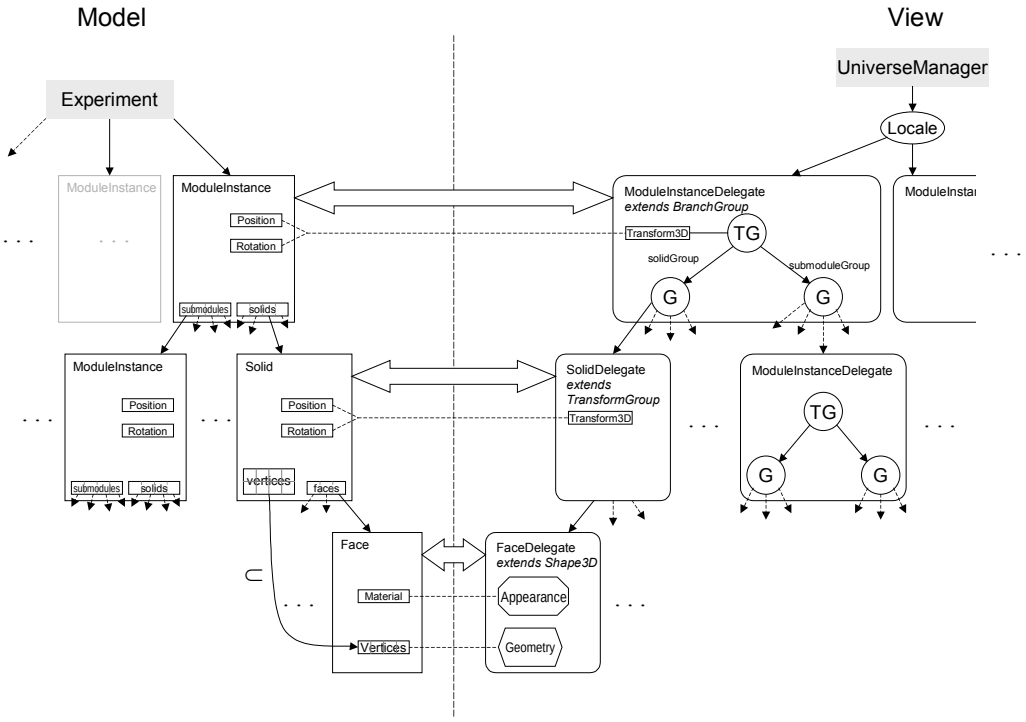
Ein VirtualUniverse-Objekt dient als Container für alle Szenegraphen. Um die nötige Anpassung an die Erfordernisse des virtuellen Labors zu ermöglichen, wurde der UniverseManager, eine Subklasse von VirtualUniverse implementiert. Zusätzlich zur Verwaltung des Szenegraphen mit den Delegates der Bausteininstanzen verwaltet er auch die Objekte des Experimentmodells. Er bietet unter anderem Methoden zum Einfügen neuer Experimentinstanzen in den Experimentaufbau, erlaubt dessen komplettes Löschen zum Beispiel beim Verwerfen eines Experiments oder vor einem Check-Out. Das Prüfen eines Experiments auf ungesicherte Änderungen wird unter anderem für Sicherheitsabfragen verwendet, um den Benutzer auf diesen Umstand hinzuweisen.

Delegate-Klassen für Bausteininstanzen

Wie in Kapitel 5.3.1.3 kurz beschrieben, werden die Objekte der Datenmodellschicht, die Bausteininstanzen und ihre sichtbaren Bestandteile repräsentieren, nicht in einer Form realisiert, die ein direktes Rendering durch Einfügen in den Scenegraph erlauben. Diese Aufgabe

übernehmen Delegate-Objekte. Abbildung 49 zeigt das Verhältnis von Modell-Objekt und Delegate und deren Nutzung im Java 3D Scenegraph. Jeder Delegate verfügt über eine Referenz auf sein Modell und registriert sich bei diesem als Listener, um Änderungen des Modells unmittelbar umzusetzen. Der Delegate einer Bausteininstanz (`ModuleInstanceDelegate`) erbt von BranchGroup und kann so beim Löschen der von ihm dargestellten Bausteininstanz aus dem Experiment jederzeit aus dem Scenegraph entfernt werden. Intern besitzt er eine TransformGroup deren Transformation der Position und Orientierung des Bausteins entspricht und die selbst über zwei Gruppen für die beiden Typen von Kindknoten verfügt: Die `submoduleGroup` nimmt weitere ModuleInstanceDelegates auf, welche die Bausteininstanzen für Komponenten im Scenegraph vertreten. Die `solidGroup` enthält `SolidDelegate-Objekte` für alle Solids der Bausteininstanz. Die beiden Gruppen entsprechen somit direkt der Liste von Referenzen der ModuleInstance auf Komponenten und Solids.

Abbildung 49 Delegate-Objekte im Scenegraph



SolidDelegates erben von der Klasse TransformGroup. Sie verfügen daher über ein Transformationsobjekt, welches der Position und Orientierung des Solids innerhalb des lokalen Koordinatensystems des Bausteintyps entspricht. Solids verfügen über Referenzen auf ihre Vertices und Faces, der SolidDelegate hält für jedes Face seines Solids eine Referenz auf einen Kindknoten vom Typ `FaceDelegate` und zusätzlich Referenzen auf alle Vertices.

Die Klasse FaceDelegates komplettiert die Adapterklassen und ist Subklasse von Shape3D. Sie kann daher gerendert und in der Visualisierung des Experimentaufbaus sichtbar werden. FaceDelegates halten Referenzen auf die Loops und deren Vertices, welche die Begrenzung des Face bilden. Aus diesen Informationen erzeugt der FaceDelegate ein Geometry-Objekt. Aus dem Material-Objekt von Face wird die Appearance für den FaceDelegate erzeugt.

5.5 Fazit - Prototyp

Wegen des großen Funktionsumfangs des virtuellen Labors kann der zuvor mit seinen wesentlichen Komponenten vorgestellte Prototyp nur einen kleinen Ausschnitt der vollen Funktionalität demonstrieren. Einige wichtige Konzepte, die im Laufe dieser Arbeit entstanden sind, konnten jedoch rudimentär implementiert werden. Dadurch kann der Prototyp mehr als nur dem Gewinn von Erfahrung im Umgang mit einzelnen Technologien und einer Einschätzung ihrer Eignung für die Nutzung im virtuellen Labor dienen. Vielmehr erhofft sich der Verfasser, dass er sich als Grundlage für die weitere Kommunikation und Diskussion mit den Anwendern des Fachbereichs Verfahrenstechnik eignet. Die Demonstration eines Konzepts erleichtert gegenüber einer reinen Schilderung gerade beim Überwinden der Verständigungsprobleme zwischen Personen mit unterschiedlichem wissenschaftlichen Hintergrund das Verständnis und die Akzeptanz und regt auch die Kreativität der Anwender an.

Kapitel 6 Zusammenfassung und Ausblick

Zum Abschluss der Arbeit soll eine kurze Zusammenfassung präsentiert und dann ein Ausblick auf interessante Arbeitsgebiete und offene Fragestellungen gegeben werden.

6.1 Zusammenfassung

Nach der Vorstellung des geplanten Sonderforschungsbereichs in Kapitel 1 und der Domäne Verfahrenstechnik in Kapitel 2.1 wurden in Kapitel 2.2 die in Zusammenarbeit mit den Anwendern aus diesem Fachbereich entwickelten Anforderungen an ein virtuelles Labor festgehalten. Als zentraler Bereich für weitere Untersuchungen hat sich dabei die Verwaltung der Daten von Experimentaufbauten bei der Experimentvorbereitung herausgestellt. Anschließend wurde in Kapitel 3 in Zusammenarbeit mit einer parallel entstehenden Diplomarbeit, welche virtuelle Labore im Bereich Siedlungswasserwirtschaft untersucht, eine Schnittmenge von gemeinsamen Anforderungen ermittelt. Als Ergebnis dieser Kooperation entstand der Kern eines domänenunabhängigen Informationsmodells, das anschließend für beide Arbeiten als Grundlage für die Entwicklung eines domänenspezifischen Informationsmodells diente. In Kapitel 4.1 wurde zunächst ein Vorschlag für die Architektur des virtuellen Labors aus den Anforderungen abgeleitet. Als Ergebnis entstand ein Vier-Schichten-Modell, welches die Funktionalität des virtuellen Labors in Datenhaltungs-, Datenmodell-, Anwendungs- und Präsentationsschicht trennt. Zudem wurden verschiedene Technologien vorgestellt, die mögliche Kandidaten für den Einsatz bei der Realisierung des Labors sind. In Kapitel 4.2 wurden anschließend mögliche Konzepte zur Umsetzung der Datenhaltungsschicht anhand von aus den Anforderungen abgeleiteten Bewertungskriterien untersucht. Der Vergleich von dateibasierte Datenhaltung, relationalen, objekt-relationalen und XML-Datenbanken fiel zugunsten der leistungsfähigen und zugleich bewährten (O)RDBMS aus. In Kapitel 4.3 bis 4.5 wurde das domänenunabhängige Informationsmodell aus Kapitel 3 konkretisiert und, wo erforderlich, für den Bereich Verfahrenstechnik ergänzt. Die Herausforderung lag hier in der vollständigen Speicherung aller Informationen, die für die Erzeugung und Nutzung eines virtuellen Experimentaufbaus erforderlich sind. So entstand ein Bausteinmodell, welches das Grundkonzept des virtuellen Labors, die Trennung in einer vereinfachte Benutzer- und eine vollständige Systemsicht sowie die Verbindung zwischen diesen

Sichten umsetzt. Kapitel 4.6 beschreibt auf abstrakten Niveau die Schnittstellen für den Zugriff auf das virtuelle Labor.

Um die zuvor vorgestellten Technologien und entworfenen Konzepte auch praktisch zu erproben, wurde ein Prototyp eines virtuellen Labors implementiert, der grundlegende Funktionen der Experimentvorbereitungsphase demonstriert. Neben einer Realisierung der Bausteinbibliothek mit Suchfunktionen kann der Benutzer Experimentbausteine auf einer virtuellen Arbeitsfläche platzieren, ihre Eigenschaften modifizieren und den Experimentaufbau in die Datenbank einbringen. Das Informationsmodell wurde in großen Teilen auf die Konzepte von IBM DB2 abgebildet. Dabei wurde der Portabilität zu Gunsten der Erprobung proprietärer Konzepte eine geringere Priorität eingeräumt. Die für die Datenbankanbindung und Übersetzung der relationalen Strukturen in Anwendungsobjekte zuständige Datenmodellschicht wurde mit dem Ziel realisiert, flexibel in verschiedenen Umgebungen einsetzbar zu sein. Neben des als 2-Tier-Applikation ausgeführten Prototypen wurden verschiedene Verfahren zur Anbindung heterogener Client-Anwendungen an die Datenhaltungs- beziehungsweise Datenmodellschicht erprobt. Exemplarisch für um XML-Funktionalität erweiterte objekt-relationale Datenbanken wurden dazu verschiedene Möglichkeiten des DB2 XML Extender untersucht. Schließlich wurde eine Teilschnittstelle der Datenmodellschicht als Web Service zugänglich gemacht. Neben den so gewonnen technischen Erfahrungen soll der Prototyp auch als Grundlage für die weitere Diskussion bei der Entwicklung des virtuellen Labors dienen.

6.2 Ausblick

Bei der Umsetzung des Informationsmodells auf ein objekt-relationales Datenbanksystem traten Schwächen zu Tage, die zumeist auf der mangelnden Konformität der Produkte mit dem SQL:1999-Standard beruhen. Insbesondere das Fehlen dedizierter Mechanismen zu Formulierung komplexer Integritätsbedingungen wie ECA-Regeln oder Assertions sowie eine verzögerte Überprüfung am Transaktionsende haben eine praktische Umsetzung deutlich erschwert. Ein SQL:1999-konformes Produkt hätte hier in vielen Fällen eine Vereinfachung und eine weitergehende Verlagerung der Integritätssicherung in die Datenhaltungsschicht ermöglicht. Trotz dieser Schwächen hat sich DB2 wie vermutet für die Implementierung des virtuellen Labors als geeignet erwiesen. Die Datenhaltungsschicht kann viele der Schwächen kompensieren und so eine Sicherung der Datenintegrität unabhängig von der Anwendung garantieren. Ob die Entscheidung für ein OR-Datenbanksystem langfristig die richtige ist, lässt sich zu diesem Zeitpunkt schwer abschätzen. Nativen XML-Datenbanken wird allgemein eine wachsende Bedeutung vorausgesagt. Nicht wenige Stimmen prophezeien den XML-Datenbanken jedoch ein ähnliches Schicksal, wie es die objektorientierten Datenbanken ereilt hat: Die objekt-relationalen Datenbanksysteme werden die besten Eigenschaften von XML-Datenbanken integrieren und diese aufgrund ihrer großen vorhandenen Kundenbasis vom Markt verdrängen. Selbst wenn eine um XML-Unterstützung erweiterte Datenbank nicht den vollen Leistungsumfang einer nativen XML-DB aufweist, werden die gebotenen Funktionen vielen Kunden bei weitem ausreichen. Kombiniert mit dem Versprechen, die Datenhaltung ihres Unternehmens schrittweise um XML zu erweitern, ohne eine harten Schnitt in Form einer kostspieligen Umstellung zu riskieren, kann dies letztlich ausschlaggebend für

viele Kunden sein, auf die Evolution der bewährten ORDBMS zu vertrauen und die XML-Datenbanken als mögliche Alternativen nicht zu berücksichtigen.

Die Verwendung von Web Services zur Anbindung der Datenmodellschicht an die Client-Applikationen hat sich unabhängig des Hypes, dessen sich diese Technologie erfreut, als Mittel zu transparenten Kommunikation über System- und vor allem Plattformgrenzen erwiesen. Einzelne Probleme bei der Umstellung lassen sich auf die Verwendung weniger leistungsfähiger Tools zurückführen und werden sich mit der zunehmenden Reife von Web Services lösen lassen.

Einzelne für die Entwicklung eines voll funktionsfähigen, domänenunabhängigen virtuellen Labors wichtige Aspekte konnten im Rahmen dieser Arbeit nicht bearbeitet werden. Vor allem die Entwicklung einer generischen Schnittstelle zwischen Datenmodellschicht und den Anwendungen für die drei Experimentphasen ist wegen der großen Unterschiede der Anwendungsobjekte der beiden untersuchten Domänen in Anzahl und Komplexität nur auf einem hohen Abstraktionsniveau zu leisten. Nur eine kleine Teilmenge wird tatsächlich in beiden Labors benötigt, insbesondere das Check-In/Check-Out-Verfahren und allgemeine Infrastruktur wie Benutzer- und Rechteverwaltung. Doch selbst hier sind die Unterschiede erheblich, da die hier ausgetauschten Daten nur wenige Gemeinsamkeiten aufweisen. Wo die Verfahrenstechnik eine überschaubare Anzahl in weiten Bereichen konfigurierbarer komplexer Bausteine benötigt, erfordert der Bereich Siedlungswasserwirtschaft eine große Menge deutlich einfacher aufgebauter Bausteine. Diese enormen Gegensätze der Kardinalitäten der Anwendungsobjekte werden ein Haupthindernis bei der Entwicklung einer einheitlichen Schnittstelle sein. Aber auch die Vorgehensweise beim Erstellen eines Experimentaufbaus divergiert. Kann im Bereich Siedlungswasserwirtschaft häufig auf bestehende Siedlungsdaten zurückgegriffen werden, die dann modifiziert werden, erfolgt zumindest in der Anfangszeit der Aufbau eines Experiments in der Verfahrenstechnik ohne einen vorhandenen Ausgangspunkt. Der Bausteinbibliothek kommt daher eine zentrale Bedeutung zu, da sie im Gegensatz zum Bereich Siedlungswasserwirtschaft mit der dort überschaubaren und weitgehend fixen Auswahl von Bausteintypen nicht einfach als Bestandteil der Applikationen hartkodiert werden kann, sondern umfangreiche Maßnahmen zur Definition eigener Bausteintypen und aufwendige Such-, Zugriffs- und Rechteverwaltungsmechanismen erfordert.

Auch innerhalb der Domäne Verfahrenstechnik bleiben bisher nicht weiter untersuchte Bereiche. Von großem Interesse wäre eine abstrahierende graphische Darstellung, welche die Erstellung von virtuellen Apparaturen vereinfacht, indem die genaue Anordnung der Bausteine im Normalfall der Anwendung überlassen werden kann. Für die Formulierung von komplexen Integritätsbedingungen auf Gruppen von Bausteinen, die nicht nur auf den leicht zu handhabenden Bausteineigenschaften beruhen, reichen die im Prototypen implementierten Constraints nicht aus. Hier muss eine leicht verständliche und zugleich mächtige Beschreibungssprache entwickelt werden.

Weiterhin müssen die Schnittstellen der Datenmodellschicht um Unterstützung für den Expertenmodus erweitert werden, der das Editieren und Neuerstellen von Experimentbausteintypen erlaubt.

Kapitel 7 Literaturverzeichnis

[AJPT] The Apache Jakarta Project
Jakarta Tomcat
Juni 2003
http://jakarta.apache.org/tomcat/index.html

[Bo2003] Ronald Bourett
XML and Databases
Januar 2003
http://www.rpbourret.com/xml/XMLAndDatabases.htm

[BöRa2001a] T. Böhme, E. Rahm
XMach-1: A Benchmark for XML Data Management
Dezember 2000
http://dol.uni-leipzig.de/pub/2001-1/en

[BöRa2001b] T. Böhme, E. Rahm
Benchmarking XML Database Systems – First Experiences
2001
http://www.research.microsoft.com/~jamesrh/hpts2001/submissions/ErhardRahm.pdf

[Bryan2002] Mike Bryan
Principles of product data exchange
November 2002
http://www.chyanbre.demon.co.uk/cookbook/cook_book.htm

[CORBA] Object Management Group, Inc.
The Common Object Request Broker: Architecture and Specification
Version 3.0
Juli 2002
http://doc.ece.uci.edu/CORBA/formal/02-06-33.pdf

[CMU1] Carnegie Mellon University
How Do You Define Software Architecture?
Juni 2003
http://www.sei.cmu.edu/architecture/definitions.html

[CRIMSON] The Apache XML Project
Crimson 1.1
Oktober 2001
http://xml.apache.org/crimson/index.html

[CSC] MageLang Institute
Introduction to CORBA Short Course
1999
http://developer.java.sun.com/developer/onlineTraining/corba/corba.html

[CSWSP] Cape Science
Web Services Primer
September 2002
http://www.capescience.com/education/primer/index.shtml

[DB2] International Business Machines Corporation
IBM DB2 Universal Database
http://www-3.ibm.com/software/data/db2/udb/

[Do2002] Laura Downs
Using Quaternions to represent rotations
Juni 2003
http://www.cs.berkeley.edu/~laura/cs184/quat/quaternion.html

[DOM4J] The dom4j Project
Juni 2003
http://www.dom4j.org/

[HMNR95] T. Härder, B. Mitschang, U.Nink, N. Ritter:
Workstation/Server-Architekturen für datenbankbasierte Ingenieurs-anwendungen
1995

[Ho89] Christoph M. Hoffman
Geometric and Solid Modelling
1992

[HvT2003] Alexander Hilliger von Thile
Daten- und Informationsverwaltung für virtuelle Labore im Bereich Siedlungswasserwirtschaft - Anforderungen und Realisierungsmöglichkeiten
Juli 2003

[IFMXXML] Susan L. Cline
International Business Machines Corporation
Using the IBM Informix XSLT and Web DataBlades
März 2003
http://www7b.software.ibm.com/dmdd/zones/informix/library/techarticle/0303cline/0303cline.html

[IGES] IGES Project
Juni 2003
http://www.nist.gov/iges/

[J3DAPI] Sun Microsystems Inc.
Java 3D API
Juni 2003
http://java.sun.com/products/java-media/3D/

[J3DORG] j3d.org – The Java 3D Community
http://www.j3d.org/

[J3DTUT] Sun Microsystems Inc.
Getting Started with the Java 3D API
A Tutorial for Beginners
http://developer.java.sun.com/developer/onlineTraining/java3d/

[JAXP] Sun Microsystems, Inc.
Java API for XML Processing (JAXP)
Juni 2003
http://java.sun.com/xml/jaxp/index.html

[JAXB] Sun Microsystems, Inc.
Java Architecture for XML Binding
Juni 2003
http://java.sun.com/xml/jaxb/index.html

[JAXR] Sun Microsystems, Inc.
Java API for XML Registries (JAXR)
Juni 2003
http://java.sun.com/xml/jaxr/index.html

[JAXRPC] Sun Microsystems, Inc.
Java API for XML-based RPC
Juni 2003
http://java.sun.com/xml/jaxrpc/index.html

[JBEANS] Sun Microsystems, Inc.
Java Beans Component Architecture
April 2003
http://java.sun.com/products/javabeans/

[JDBC] Sun Microsystems, Inc.
JDBC Data Access API
Juni 2003
http://java.sun.com/products/jdbc/

[JDOM] jdom.org
Juni 2003
http://www.jdom.org/

[JWS] Sun Microsystems, Inc.
Java Technology and Web Services
Juni 2003
http://java.sun.com/webservices

[MCE] M.C.Escher - The official Website
Moebius Strip II
Juni 2003
http://www.mcescher.com/Gallery/recogn-bmp/LW441.jpg

[MQFAQ] j3d.org
The Matrix and Quaternion FAQ Version 1.20
Januar 2003
http://www.j3d.org/matrix_faq/matrfaq_latest.html

[MW1] mathword.wolfram.com
Moebius Strip
Juni 2003
http://mathworld.wolfram.com/MoebiusStrip.html

[MW2] mathword.wolfram.com
Klein Bottle
Juni 2003
http://mathworld.wolfram.com/KleinBottle.html

[ODDXF] OpenDWG Alliance
Why isn't DXF good enough?
2002
http://www.opendwg.org/about/whtpaper/whynot.htm

[OXMLDB] Oracle Corporation
Oracle XML DB - An Introduction
2002
http://otn.oracle.com/tech/xml/xmldb/htdocs/xmldb_intro.html

[ProSTEP] ProSTEP iViP Association
Vergleich der neutralen Schnittstellen STEP, IGES und VDAFS
http://www.prostep.org/file/10700.GESSTEPVAF

[RaVo2003] Erhard Rahm, Gottfried Vossen
Web & Datenbanken - Konzepte, Architekturen, Anwendungen
dpunkt.verlag
September 2002

[Rott2001] DI. Franz Rottensteiner
Semi-automatic extraction of buildings based on hybrid adjustment using 3D surface models and management of building data in a TIS (PhD Thesis)
2001

[SeKr2002] Sebastian Kraus
Tamino Experience Report: Advantages and Pitfalls of an XML Database Management System
August 2002
http://www.pms.informatik.uni-muenchen.de/publikationen/ projektarbeiten/Sebastian.Kraus/tamino_exp_rep.pdf

[Sh2002] Dr. C.-K. Shene
Introduction to Computing with Geometry
Dezember 2002
http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/notes.html

[St2001] Kimbro Staken
Introduction to Native XML Databases
Oktober 2001
http://www.xml.com/pub/a/2001/10/31/nativexmldb.html

[Tü2003] Dr. Can Türker
SQL:1999 & SQL:2003
Objektrelationales SQL, SQLJ & SQL/XML
dpunkt.verlag
Februar 2003

[VDI] Verein Deutscher Ingenieure e.V.
Gesellschaft Verfahrenstechnik und Chemieingenieurwesen
VDI-GVC

[VL2002] Virtuelle Laboratorien
Konzept für einen DFG-Sonderforschungsbereich an der Universität Kaiserslautern
September 2002

[W3CSoap] W3C
SOAP Version 1.2 Part 0: Primer
Mai 2003
http://www.w3.org/TR/soap12-part0/

[W3CWSA] W3C
Web Services Architecture
Mai 2003
http://www.w3.org/TR/ws-arch/

[W3CXML] World Wide Web Consortium
Extensible Markup Language
Juni 2003
http://www.w3.org/XML/

[W3CXP] World Wide Web Consortium
XML Path Language (XPath)
November 1999
http://www.w3.org/TR/xpath

[W3CXQ] World Wide Web Consortium
XML Query
Juni 2003
http://www.w3.org/XML/Query

[XBench] Benjamin Bin Yao
XBench - A Family of Benchmarks for XML DBMSs
Dezember 2002
http://db.uwaterloo.ca/~ddbms/projects/xbench/Publications.html

[XERCEX] The Apache XML Project
Xerces 2 Java Parser
Juni 2003
http://xml.apache.org/xerces2-j/index.html

[XMach] XMach-1
Benchmarking XML Data Management Systems
Mai 2003
http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html

[XMLDB] XML:DB Initiative for XML Databases
XUpdate - XML Update Language
November 2000
http://www.xmldb.org/xupdate/

[XMLExt] International Business Machines Corporation
DB2 XML Extender Administration and Programming
Juni 2003
http://www-3.ibm.com/software/data/db2/extenders/xmlext/index.html

Anhang A Tabellarische Anforderungen

Um bei der Diskussion von Realisierungsmöglichkeiten eine möglichst genaue Bezugnahme auf einzelne Benutzeranforderungen an das virtuelle Labor zu ermöglichen, werden die Anforderungen aus der in Kapitel 2.2 beschriebenen erwarteten Systemfunktionalität an dieser Stelle verkürzt in tabellarische Form nochmals aufgeführt und mit einer Kurzbezeichnung versehen. Bei Verweisen auf Anforderungen werden diese Kurzbezeichnungen verwendet.

Die Anforderungen sind gemäß ihrer Thematik hierarchisch organisiert. Durch diese Einteilung soll keine Aussage über die Bedeutsamkeit einzelner Anforderungen getroffen werden.

Allgemeine Konzepte

K1 Trennung Nutzersicht-Systemsicht

Das virtuelle Labor unterscheidet zwei Sichten auf die Experimentierumgebung: Die Systemsicht umfasst alle verfügbaren Informationen über ein Experiment und seine Bestandteile, während die Benutzersicht von den Details abstrahiert und für den Benutzer so im normalen Umgang mit dem System leichter zu handhaben ist.

K2 Nutzerverwaltung

Das virtuelle Labor verfügt über eine Nutzerverwaltung mit Authentifizierung.

K2.1 Nutzergruppen

Benutzer können Mitglied in Benutzergruppen sein.

K2.1.1 Administratoren

Eine ausgezeichnete Benutzergruppe sind die Administratoren. Benutzer, die Mitglied dieser Gruppe sind, können andere Nutzer einer Gruppe zuordnen.

K2.2 Objektbesitz

Einzelne Objekte des virtuellen Labors wie Bausteintypen oder Experimente, können einem einzelnen Benutzer als Besitzer zugeordnet werden.

K2.3 Objektfreigabe

Der Besitzer eines Objekts kann dies anderen Benutzern oder Benutzergruppen durch Vergabe von Zugriffsrechten (siehe K2.4) zugänglich machen.

K2.4 Zugriffsrechte

Ein Benutzer kann an einem Objekt Leserechte (*read*), Schreibrechte (*write*) oder das Recht zur Ableitung einer neuen Version erhalten (*derive*). Das Recht *derive* beinhaltet dabei das Leserecht *read*, das Recht *write* sowohl das Leserecht *read* als auch das Ableitungsrecht *derive*.

K2.4.1 Keine Weitergabe von Zugriffsrechten

Ein Benutzer, der Zugriffsrechte an einem Objekt hält, das nicht sein eigenes ist, kann dieses Recht nicht an andere Benutzer weitergeben.

K3 Nachbildung der realen Arbeitsweise

Das virtuelle Labor bildet die Arbeitsweise eines Experimentators in einem realen Experiment nach. Reale Experimente lassen sich in die drei Phasen Versuchsplanung, Versuchsdurchführung und Versuchsanalyse gliedern. Anforderungen, die sich auf diese Phasen beziehen, werden im Anschluss gesondert aufgeführt.

Versuchsplanung

P1 Virtuelle Arbeitsfläche

Der Aufbau eines virtuellen Experiments erfolgt auf einer virtuellen Arbeitsfläche (Workbench).

P2 Bausteinbibliothek

Dem Benutzer steht eine Bibliothek von Experimentbausteinen zur Verfügung, die er auf der virtuellen Arbeitsfläche zu einem Experimentaufbau kombinieren kann.

P2.1 Echte und fiktive Bausteine

Die Bausteine der Versuchsbibliothek können eine reale Entsprechung haben (echte Bausteine) oder rein virtuell existieren (fiktive Bausteine). Diese Trennung hat keinen Einfluss auf ihre spätere Verwendung.

P2.2 Bausteinbeschreibung

Die Beschreibung der Experimentbausteine enthält alle relevanten Informationen. Entsprechend der Anforderung K1 werden diese Informationen in eine Benutzersicht und eine Systemsicht unterteilt

P2.2.1 Systemsicht

Die Systemsicht enthält alle für die Verwendung des Bausteins bei Versuchsaufbau und -durchführung relevanten Informationen.

P2.2.1.1 Geometrie

Die Systemsicht enthält eine Beschreibung der Geometrie des Bausteins, zusammengesetzt aus einzelnen Festkörpern.

P2.2.1.1.1 Exakte Geometriedarstellung

Die Darstellung soll ohne Approximation gekrümmter Flächen durch Polyeder auskommen.

P2.2.1.1.2 Erstellung aus Grundformen

Die Geometrie von Bausteinen soll durch Kombination von Grundformen wie Quadern, Kugeln, Zylindern, Prismen, Pyramiden oder Kegeln mittels der Aneinanderreihung von Verschiebe-, Dreh- und Mengenoperationen (Schnitt, Vereinigung, Differenz) erstellt werden können.

P2.2.1.1.3 Komplexe Formen

Um aus den Grundformen nur schwer herleitbare komplexere Formen der Körper zu unterstützen, ist eine mathematische Beschreibung dieser Formen zu ermöglichen.

P2.2.1.1.4 Import

Bei der Erstellung von Bausteinen sollen vorhandene Geometriedaten aus anderen Anwendungen importiert werden können.

P2.2.1.2 Randbedingungen

Einzelnen Flächen oder Körpern der Geometriedarstellung sollen Randbedingungen zugeordnet werden können. Dazu gehören Materialeigenschaften, Temperaturen, die Definition als Ein- und Auslassöffnungen oder als bewegliches Element.

P2.2.1.3 Verbindungselemente

Um Bausteine zu einem Experimentaufbau zusammenfügen zu können, müssen sie über Verbindungselemente verfügen. Die Beschreibung dieser Verbindungselemente soll sowohl die Art des Zusammenwirkens von so verbundenen Bausteinen beschreiben als auch eine Beschreibung der anderen zu diesem Element passenden Verbindungselemente enthalten.

P2.2.1.3.1 Suche nach kompatiblen Bausteinen

Der Benutzer soll für ein ausgewähltes Verbindungselement nach Bausteinen suchen können, die zu diesem Element kompatibel sind. Alle passenden Elemente sollen dem Benutzer als Suchergebnis zugänglich gemacht werden.

P2.2.1.4 Zusammengesetzte Bauteile

Bauteile können durch Zusammenfügen von anderen Bauteilen erstellt werden.

P2.2.1.5 Zugang zur Systemsicht

Dem Benutzer ist auf Wunsch die Systemsicht der Bausteine zugänglich zu machen.

P2.2.2 Nutzersicht

Die Nutzersicht eines Bausteins umfasst nur die Informationen über einen Baustein, die für den Benutzer im normalen Umgang mit dem Baustein von Bedeutung sind. Die Informationen der Nutzersicht können entweder aus denen der Systemsicht gewonnen oder explizit für die Nutzersicht formuliert werden. Die beiden Hauptkomponenten der Nutzersicht sind die graphische Darstellung des Bausteins und seine Eigenschaften.

P2.2.2.1 Graphische Darstellung

Zur Visualisierung des Experimentaufbaus ist dem Benutzer eine graphische Darstellung der Bausteine zugänglich zu machen.

P2.2.2.2 Eigenschaften

Alle nicht-graphischen Informationen über ein Bauteil werden als Eigenschaften (properties) bezeichnet. Eigenschaften haben einen Namen und einen Wert, der von einem bestimmten Typ ist.

P2.2.2.2.1 Typen von Eigenschaften

Eigenschaften können beschreibende Texte, numerisch oder komplex sein. Numerische Typen können auf ganzen oder rationalen Zahlen basieren. Numerische Typen können Skalare oder Vektoren sein. Numerische Typen können über eine Einheit verfügen. Eine Umrechnung zwischen verschiedenen Einheiten eines Typs ist möglich.

P2.2.2.2.2 Komplexe Eigenschaften

Komplexe Eigenschaften sind nicht rein numerisch beschreibbar. Sie sind selbst in weitere numerische oder textuelle Eigenschaften strukturiert. Der Benutzer kann bei komplexen Eigenschaften aus einer vordefinierten Menge von möglichen Werten wählen. Die Werte komplexer Eigenschaften können hierarchisch organisiert sein.

P2.2.2.2.3 Variable Eigenschaften (Parameter)

Eigenschaften eines Bausteins können als variabel deklariert werden. Sie können dann vom Benutzer verändert werden. Änderungen wirken sich unter Umständen auf die Systemsicht und damit das Verhalten des Bausteins im Experiment aus.

Parameter lassen sich in Bauteileparameter und Betriebsparameter unterteilen. Bauteileparameter legen grundsätzliche Eigenschaften eines Bausteins fest und können nach Integration in einen Experimentaufbau nicht ohne Folgen für andere (mit dem geänderten Baustein verbundene Bausteine) geändert werden.

Betriebsparameter hingegen sind ohne Auswirkungen für verbundene Bausteine änderbar.

P2.2.2.2.4 Verpflichtende und optionale Parameter

Nicht alle variablen Eigenschaften eines Bausteins müssen mit einem Wert versehen werden, sie werden als optionale Parameter bezeichnet. Im Gegensatz dazu müssen die verpflichtenden Parameter mit einem gültigen Wert versehen werden, um den Baustein in einem Experiment einsetzen zu können.

P2.2.2.2.5 Konsistenzbedingungen für Parameter

Der Ersteller eines Bausteins kann für dessen Parameter den gültigen Wertebereich spezifizieren. Diese Spezifikation kann auf Ebene einzelner Parameter erfolgen oder bausteinübergreifend auf mehreren Parametern. Die Typen der an diesen parameterübergreifenden Konsistenzbedingungen beteiligten Parameter müssen kompatibel sein.

P2.2.2.2.6 Trennung Baustein-/Betriebsparameter

Parameter eines Bausteins, die ohne Einfluss auf andere mit diesem Baustein verbundene Bausteine modifiziert werden können (Betriebsparameter), müssen von den nicht nebenwirkungsfrei veränderbaren Parametern (Bausteinparameter) unterschieden werden.

P2.2.3 Bauteilevarianten

Für Bausteine mit variablen Eigenschaften (Parametern) können in der Bausteinbibliothek Varianten spezifiziert werden, welche zumindest die verpflichtenden Parameter des Bausteins mit vordefinierten Werten versehen. Ein in Form einer Variante ins Experiment eingefügter Baustein ist so direkt benutzbar.

P2.2.3.1 Variantenbezeichnung

Um verschiedene Varianten eines Bausteins unterscheiden zu können, erhalten sie eine innerhalb des Bausteins eindeutige Bezeichnung.

P2.2.3.2 Unabhängigkeit von Varianten und darauf basierenden Bausteinen

Die Parameter eines in Form einer Variante ins Experiment eingefügten Bausteins können ohne Auswirkungen auf die Ursprungsvariante verändert werden.

P2.2.3.3 Erzeugen neuer Varianten

Der Benutzer kann auf Basis eines Bausteins, dessen Parameter modifiziert wurden, eine neue Variante erstellen. Diese neuen Varianten werden unter einer neu zu vergebenden Bezeichnung in die Bausteinbibliothek aufgenommen.

P2.2.4 Verbindung System- und Nutzersicht

Da die Nutzersicht einen Ausschnitt aus der Systemsicht darstellt, muss die Verbindung zwischen diesen Sichten definiert werden.

P2.2.4.1 Abbildung Nutzersicht auf Systemsicht

Damit die Änderung der Bausteinparameter die Funktionsweise des Bausteins modifizieren können, ist eine Beschreibung der Auswirkungen dieser Änderungen auf die Systemsicht erforderlich.

P2.2.4.2 Abbildung Systemsicht auf Nutzersicht

Für Teile der Nutzersicht, die direkt aus der Systemsicht abgeleitet werden, muss die Art dieser Ableitung definiert werden.

P2.2.5 Konsistenzbedingungen für die Verwendung von Bausteinen

Der Ersteller eines Bausteins kann Bedingungen formulieren, welche die möglichen Nutzungsformen seines Bausteins einschränken.

P2.2.5.1 Position von Bausteinen

Für Bausteine kann einen Mindestabstand von anderen Bausteinen oder eine Beschränkung der möglichen Orientierung definiert werden.

P2.2.5.2 Kombination mit anderen Bausteintypen

Der Entwickler eines Bausteins kann dessen Verwendung im Kontext mit anderen Bausteintypen einschränken.

P2.3 Aufbau der Bausteinbibliothek

Die Bausteine sind in der Bausteinbibliothek in hierarchisch organisierte Kategorien unterteilt. Die maximale Tiefe dieser Hierarchie ist nicht vorgegeben. Kategorien in jeder Ebene der Hierarchie können sowohl über Unterkategorien als auch über Bauteile verfügen.

P2.3.1 Kategoriezugehörigkeit von Bausteinen

Ein Bausteintyp muss genauer einer Kategorie zugeordnet werden.

P2.3.2 Eigenschaften von Kategorien

Kategorien können wie Bausteintypen über Eigenschaften verfügen. Diese Kategorieeigenschaften haben nur Name und Typ, jedoch keinen Wert.

P2.3.2.1 Vererbung von Eigenschaften zwischen Kategorien

Eine Kategorie muss mindestens alle Eigenschaften ihrer übergeordneten Kategorie aufweisen.

P2.3.2.2 Eigenschaften von Bausteinen innerhalb von Kategorien

Ein Baustein muss mindestens alle Eigenschaften einer Kategorie aufweisen, um in diese eingeordnet werden zu können.

P2.3.3 Suche in Kategorien

Die Kategorien der Bausteinbibliothek dienen der Suche nach Bausteinen.

P2.3.3.1 Manuelle Suche (Browsing)

Der Benutzer kann die Kategorien manuell nach Bausteinen durchsuchen, indem er durch die Baumstruktur navigiert.

P2.3.3.2 Suche auf Eigenschaften

Der Benutzer kann über den Eigenschaften einer Kategorie ein Suchprädikat formulieren. Dieses Prädikat wird für alle Bausteinvarianten geprüft, die zu Bausteinen gehören, die direkt oder indirekt innerhalb dieser Kategorie liegen. Alle Varianten, welche das Such-

prädikat erfüllen, werden als Ergebnis der Suchoperation dem Benutzer zugänglich gemacht.

P2.3.4 Benutzerabhängige Darstellung

Die Bausteinbibliothek zeigt immer nur die für den aktuellen Benutzer mindestens lesend zugänglichen Bausteine (siehe dazu K2.4).

P2.3.4.1 Änderungen der Kategorien

Benutzergruppen können das Recht erhalten, neue Kategorien zu erstellen oder existierende zu löschen oder umzubenennen.

P2.4 Aufbau einer virtuellen Apparatur

Der Benutzer kann Bausteintypen oder -varianten auswählen und auf der Arbeitsoberfläche zu einer virtuellen Apparatur zusammenfügen.

P2.4.1 Positionierung von Baustein

Bausteine können im Rahmen der für sie geltenden Konsistenzbedingungen frei innerhalb des virtuellen Versuchsaufbaus positioniert und orientiert werden.

Auf die Berücksichtigung von Stützelementen kann verzichtet werden, sodass Bausteine frei schwebend im Raum positioniert werden können.

P2.4.2 Änderbarkeit von Bausteinen

Die Parameter von ins Experiment integrierten Bausteinen sind jederzeit zugänglich und innerhalb der für den Bausteintyp geltenden Konsistenzbedingungen änderbar.

P2.4.3 Hinweis auf fehlerhafte Apparaturen

Entsteht durch Manipulationen an Bausteinen eine ungültige Apparatur, ist der Benutzer darauf hinzuweisen. Ungültige Apparaturen sind zuzulassen.

P2.5 Ermittlung von Messwerten

Die Ermittlung von Messwerten erfolgt mit Hilfe von Messzonen. Messzonen erhalten eine experimentweit eindeutige Bezeichnung, unter der die von ihnen ermittelten Werte zugänglich sind.

P2.5.1 Form von Messzonen

Messzonen können null- (punktförmig), ein- (kurvenförmig), zwei- (flächenförmig) oder dreidimensional (Körper) sein.

P2.5.2 Dichte der Messpunkte

Innerhalb einer höherdimensionalen Messzone kann der Benutzer die Dichte der diskreten Messpunkte festlegen.

P2.5.3 Art der ermittelten Messwerte

Messzonen können für die Ermittlung einer Vielzahl von skalaren und vektoriellen physikalischen Größen genutzt werden.

Die von einer bestimmten Messzone ermittelte Art von Messwert muss vor ihrer Verwendung ausgewählt werden.

P2.5.4 Häufigkeit der Messwertermittlung

Der Benutzer kann die Häufigkeit der Messwertermittlung festlegen.

P2.5.5 Wirkung von Messzonen auf das Experiment

Messzonen beeinflussen das Ergebnis eines Experiments nicht.

P2.5.6 Messsonden

Messzonen können in Bausteine integriert werden, die dann als Messsonden bezeichnet werden. Sie haben so alle Eigenschaften einer Messzone, beeinflussen jedoch zusätzlich durch ihre geometrischen und funktionalen Eigenschaften wie ein gewöhnlicher Baustein das Experiment.

P2.6 Speicherung von Experimentaufbauten

Ein Experimentaufbau kann zu jedem Zeitpunkt unabhängig von seiner aktuellen Konsistenz in die Datenbank eingebracht werden (Check-In). Ein so gesichertes Experiment wird bei einem späteren Laden (Check-Out) exakt wiederhergestellt.

P2.6.1 Bezeichnungen

Der Benutzer muss für seinen Experimentaufbau eine Bezeichnung vergeben. Diese Bezeichnung muss unter allen Experimenten eines Benutzers eindeutig sein.

P2.6.2 Exklusive Nutzung

Ein Experiment kann zu jedem Zeitpunkt von maximal einem Benutzer per Check-Out genutzt werden.

P2.6.2.1 Ableitung einer neuen Version

Wird ein Experiment von einem Benutzer bearbeitet, kann ein anderer Benutzer, der mindestens die Berechtigung *derive* für dieses Experiment hat, daraus eine neue Version ableiten. Das so abgeleitete Experiment geht in den Besitz des ableitenden Benutzers über, der eine neue eindeutige Bezeichnung vergeben kann.

P2.6.3 Überschreiben beim Check-In

Wird ein Experiment eingecheckt, wird der ursprüngliche Experimentzustand in der Datenbank vom neuen Zustand überschrieben.

P2.6.3.1 Explizite Versionierung

Ist ein Überschreiben des alten Zustandes nicht gewünscht, kann das Experiment unter einer neuen Bezeichnung oder durch Vergabe einer Versionsnummer in die Datenbank eingebracht werden. Der ursprüngliche Zustand des Experiments beim letzten Check-Out bleibt so in der Datenbank erhalten.

Versuchsdurchführung

D1 Validierung

Vor der Versuchsdurchführung erfolgt eine Validierung des Experimentaufbaus. Der Benutzer wird auf Fehler aufmerksam gemacht und erhält die Möglichkeit zur Korrektur.

D1.1 Nicht-kritische Fehler

Der Benutzer kann nicht-kritische Fehler ignorieren, die bei der Validierung entdeckt wurden und mit der Versuchsdurchführung fortfahren.

D2 Parametrisierung des Experiments

Vor der Experimentdurchführung ist eine Anpassung des Experiments ohne Änderung des Experimentaufbaus möglich.

D2.1 Anpassung der Apparatur

Der Benutzer kann die Betriebsparameter von Bausteinen innerhalb des Versuchsaufbaus vor der Versuchsdurchführung modifizieren.

D2.1.1 Parameteränderung während der Experimentdurchführung

Der Benutzer kann eine Änderung von Betriebsparametern während der Experimentdurchführung vornehmen

D2.1.2 Zeitgesteuerte Parameteränderung

Der Benutzer kann durch Angabe von Zeitpunkt und Wert Parameteränderungen während der Experimentdurchführung automatisiert auslösen.

D2.1.3 Modifikation der Messsonden und -zonen

Die im Experimentaufbau vorhandenen Messsonden und -zonen können durch Umbenennen und Änderung der Messhäufigkeit angepasst werden.

D2.2 Wahl des Simulationsmodells

Der Benutzer kann aus einer Menge von Simulationsmodellen das Modell auswählen, das für die Versuchsdurchführung eingesetzt werden soll.

D2.2.1 Parametrisierung des Modells

Das Simulationsmodell kann über global gültige Parameter verfügen, die vom Benutzer modifiziert werden können.

D2.3 Speicherung der Parametrisierung

Die vom Benutzer gewählten Parameter für Experimentaufbau und Simulationsmodell können in Verbindung mit dem Experimentaufbau gespeichert werden. Die Anzahl dieser Parametersätze ist nicht begrenzt.

D3 Versuchsdurchführung

Nach Beseitigung aller kritischen Fehler im Experimentaufbau und Setzen aller erforderlichen Parameter kann der Benutzer die Durchführung eines virtuellen Experiments starten.

D4 Messwerte

Die bei der Versuchsdurchführung entstehenden Messwerte werden dem Benutzer schon während der Experimentdurchführung zugänglich gemacht.

D4.1 Visualisierung

Die Messwerte können mit verschiedenen Verfahren wie in der Versuchsanalyse beschrieben visualisiert werden.

D5 Abbruch der Versuchsdurchführung

Der Benutzer kann ein laufendes virtuelles Experiment jederzeit abbrechen.

Analysephase

A1 Zugang zu Messwerten

Der Benutzer kann auf alle während der Experimentdurchführung angefallenen Messwerte zugreifen.

A1.1 Tabellarische Darstellung

Der Benutzer kann die Messwerte textuell dargestellt in Tabellenform einsehen.

A1.2 Visualisierung

Messwerte können in Form von Diagrammen dargestellt werden.

A1.2.1 Zeitverlauf

Der Benutzer kann Messwerte als Kurven über den Zeitverlauf anzeigen lassen. In ein Diagramm können verschiedene Messwerte integriert werden. Die Skalierung von Zeit- und Wertachse ist frei wählbar.

A1.2.2 Verteilung

Der Benutzer kann die Verteilung der Messwerte als Diagramm anzeigen lassen. Dazu kann der Benutzer Wertebereiche spezifizieren, die im Diagramm zusammengefasst werden.

A1.2.3 Hinzufügen von Visualisierungswerkzeugen

Die Menge der zur Auswahl stehenden Visualisierungswerkzeuge soll erweiterbar sein.

A2 Weiterverarbeitung

A2.1 Interne Weiterverarbeitung

Der Benutzer kann die Messwerte innerhalb der Analysekomponente weiterverarbeiten. Dazu sind mathematische Werkzeuge ähnlich denen einer Tabellenkalkulation zur Verfügung zu stellen.

A2.2 Externe Weiterverarbeitung

Zur Durchführung von komplexen Weiterverarbeitungsschritten kann der Benutzer die Messwerte in andere Anwendungen exportieren. Es sind geeignete Exportschnittstellen und Formate zur Verfügung zu stellen.

A2.3 Automatisierung

Der Benutzer kann sich wiederholende Weiterverarbeitungsschritte automatisieren.

Anhang B Gemeinsames Informationsmodell