



Technische Universität München
Ingenieur fakultät Bau Geo Umwelt
Lehrstuhl für Computergestützte Modellierung und Simulation
Prof. Dr.-Ing. André Borrmann

Code Compliance Checking

**Entwicklung einer Methode zur automatisierten Konformitätsüberprüfung
auf Basis einer graphischen Sprache und Building Information Modeling**

Cornelius Preidel B. Sc. (TUM)

Masterthesis
zur Erlangung des akademischen Grades
eines Master of Science (M. Sc.)

Autor: Cornelius Preidel B. Sc. (TUM)
Matrikelnummer: XXXXXXXXXX
Betreuer: Prof. Dr.-Ing. André Borrmann
Themenausgabe: 01. März 2014
Eingereicht am: 20. August 2014

I Abstract

With the development of new digital methods, such as the *Building Information Modeling (BIM)*, the construction industry has experienced a fundamental transformation. Due to the rapid technological progress of the last decades, the construction industry has gained the necessary resources to optimize the steps of a construction process in terms of effort, time and cost. These working-processes include the review of a planned project in terms of its compliance with the applicable standards of the construction industry, which has been carried out so far manually to a large extent. The so-called *Code Compliance Checking* is of particular importance, since this complex process must be carried out by the building authorities as well as the construction company. It requires a high level of expertise, experience and care of the reviser.

In the present master thesis previous developments concerning the *Code Compliance Checking* show the actual state of the art and subsequently its pros and cons are analyzed. On this basis requirements can be defined for the overall process, which guarantee a successful implementation.

Using these standards, a new method is introduced, which allows the translation of the contents of a set of rules into a machine-readable language with the help of a *Visual Language*, which improves the human-machine communication for the overall process.

Finally, the viability of the proposed approach is demonstrated by a practical implementation based on the *BIM* platform *bim+*.

II Kurzfassung

Durch die voranschreitende Entwicklung von neuen digitalen Möglichkeiten wie dem *Building Information Modeling (BIM)* erlebt die Baubranche gegenwärtig einen grundlegenden Wandel. Durch den rasanten technischen Fortschritt der vergangenen Jahrzehnte stehen dem Bauwesen die notwendigen Instrumente zur Verfügung, um die Arbeitsschritte bei der Durchführung eines Bauprozesses hinsichtlich Aufwand, Zeit und Kosten zu optimieren. Zu diesen Arbeitsprozessen gehört unter anderem auch die Überprüfung einer Gestaltungsplanung im Hinblick auf ihre Konformität mit den geltenden Normen des Bauwesens, welche bisher zu großen Teilen manuell durchgeführt wird. Dem sogenannte *Code Compliance Checking* kommt eine besondere Bedeutung zu, da dieser aufwendige Prozess sowohl von Seiten der Baubehörden als auch dem Bauunternehmen durchgeführt werden muss und ein hohes Maß an Fachwissen, Erfahrung und Sorgfalt des Bearbeiters bedarf.

In der vorliegenden Arbeit wird auf der Basis bisheriger Entwicklungen im Bereich des *Code Compliance Checking* der Stand der Wissenschaft vorgestellt und dieser hinsichtlich seiner Vor- und Nachteile analysiert. In der Folge werden Anforderungen an den Gesamtprozess formuliert, welche eine erfolgreiche Durchführung garantieren.

Mittels dieser Anforderungen wird nachfolgend eine neue Methode eingeführt, welche die Übersetzung der Inhalte eines Regelwerks in eine maschineninterpretierbare Sprache durch eine *visuelle Sprache* ermöglicht und somit die Mensch-Maschine-Kommunikation für den Gesamtprozess verbessert.

Abschließend wird die Tragfähigkeit des vorgestellten Ansatzes durch eine praktische Umsetzung auf Basis der *BIM*-Plattform *bim+* nachgewiesen.

III Inhaltsverzeichnis

I	Abstract	2
II	Kurzfassung	3
III	Inhaltsverzeichnis	4
1	Vorwort	6
2	Einleitung	7
2.1	Motivation	7
2.2	Ausgangspunkt	7
2.3	Ziel Arbeit	8
2.4	Aufbau der Arbeit.....	9
3	Stand der Wissenschaft.....	10
3.1	Bisherige Entwicklung des Code Compliance Checking.....	10
3.1.1	Logikbasierte Ansätze	10
3.1.2	Sprachbasierte Ansätze	18
3.1.3	Ontologische Ansätze.....	30
3.1.4	Code Compliance Checking auf BIM-Plattformen.....	39
3.2	Struktur des Code Compliance Checking.....	56
3.3	Anforderungen an den Überprüfungsprozess.....	57
3.3.1	Übersetzung und Auslegung der Regelwerke	57
3.3.2	Vor- und Aufbereitung des Gebäudemodells.....	59
3.3.3	Ausführung des Überprüfungsprozesses	60
3.3.4	Aufbereitung der Ergebnisse des Prozesse.....	61
4	Code Compliance Checking auf Basis einer visuellen Sprache	62
4.1	Methodik.....	62
4.1.1	Historische Entwicklung des Visual Programming	62
4.1.2	Grundlagen einer visuellen Sprache.....	66
4.2	Visual Code Checking Language	70
4.2.1	Elemente der VCCL	70
4.2.2	Grundsätze und Eigenschaften der VCCL	82
4.2.3	VCCL und das Gebäudemodell.....	84
4.3	Beispielhafte Abbildung einer Norm.....	86
5	Nachweis der Tragfähigkeit.....	92
5.1	bim+	92

5.2	Code Builder.....	100
5.2.1	Aufbau und Funktionsweise.....	100
5.2.2	Abbildung einer Norm mit Hilfe des Code Builder	106
6	Zusammenfassung und Ausblick.....	114
A	Abkürzungsverzeichnis	116
B	Anhang	117
C	Literaturverzeichnis.....	118
D	Abbildungsverzeichnis	124
E	Codeverzeichnis.....	128
F	Syntaxdiagrammverzeichnis.....	129

1 Vorwort

Ich möchte mich im Rahmen dieses Vorwortes sehr herzlich bei allen bedanken, die mich bei der Erstellung der vorliegenden Masterarbeit unterstützt haben.

Im Besonderen gilt mein Dank meinem ersten Betreuer, Herrn Prof. Dr.-Ing. André Borrmann, für die Anregung zu dieser Arbeit, sein Interesse daran und seine stets sehr gute Betreuung.

Des Weiteren möchte ich meinem Betreuer auf Seiten der *bim+ GmbH*, Herrn Dipl.-Ing. Carl-Heinz Oberender, dafür danken, dass er den engen Kontakt und die fruchtbare Zusammenarbeit mit der Praxis für mich ermöglicht hat.

Darüber hinaus möchte ich meinen Kollegen bei der *bim+ GmbH* für die stetige Unterstützung und die gute Atmosphäre danken, in welcher ich diese Arbeit anfertigen durfte.

2 Einleitung

2.1 Motivation

Durch die voranschreitende Entwicklung von neuen Technologien erlebt die Baubranche gegenwärtig einen grundlegenden Wandel, der insbesondere durch die digitale Methode *Building Information Modeling (BIM)* ins Leben gerufen wurde. Durch diesen Fortschritt und eine schnelle Vernetzung immer leistungsfähigerer Computer ergeben sich für die Baupraxis ständig neue Anwendungs- und Forschungsbereiche.

Dieser Wandel spiegelt sich nicht nur in den Strukturen und Strategien vieler Unternehmen in der Baubranche wider, die sich vermehrt für diese Methoden öffnen und Ressourcen für die Entwicklung und Anpassung an diese Technologien in ihren eigenen Reihen aufwenden, sondern gewinnt auch zunehmend an politischem und medialem Interesse (McGraw Hill Construction 2009). So haben insbesondere die in den letzten Jahren entstandenen erheblichen Kostensteigerungen und Verzögerungen bei der Durchführung von Großprojekten in Deutschland dazu beigetragen, dass ein Einsatz von digitalen Planungswerkzeugen zur Optimierung von Bauprozessen vermehrt in den Fokus der Öffentlichkeit rückt (Kammholz 14.05.2014).

Mit der Größe eines Bauprojektes steigt auch die Komplexität des gesamten Prozesses rapide an, da dieser durch die hohe Anzahl von Prozessen und Projektbeteiligten mit unterschiedlichen Interessen charakterisiert ist. Die Vielzahl von Schnittstellen bringt hohe Transaktionskosten und Fehleranfälligkeit für das Bauprojekt mit sich. Laut einer Kostenanalyse von Gallaher (2004) im Auftrag für die amerikanische Regierungsbehörde *National Institute of Standards and Technology (NIST)* belaufen sich die zusätzlichen Kosten in Folge des fehlerhaften oder unvollständigen Daten- und Informationsaustausch im Bauwesen alleine in den USA auf ca. 15,8 Milliarden US-\$ pro Jahr.

Trotz des technischen Fortschritts werden viele Prozesse innerhalb des Bauwesens noch immer manuell gesteuert und mittels zwei-dimensionaler Planung auf dem Papier bearbeitet. Zwar haben sich *CAD*-Systeme in der Baupraxis weitestgehend etabliert, doch werden diese innerhalb der einzelnen Bauprozesse zumeist nicht effizient angewendet. Somit werden die vorhandenen Potentiale nicht vollständig ausgeschöpft.

Um an dieser Stelle Optimierungen zu erzielen, werden zur Zeit Möglichkeiten und Grenzen neuere Instrumente und Funktionalitäten rund um die Anwendungsbereiche Architektur und Ingenieurwesen, die sogenannten *AEC*, ausgelotet (Garrett, Palmer und Salih 2014).

2.2 Ausgangspunkt

Im Bauwesen dienen Normen der Vereinheitlichung von Anforderungen und sichern auf diese Weise Technikstandards, um beispielsweise die Statik, Betriebssicherheit, Materialqualität und nicht zuletzt die Sicherheit des Nutzers zu garantieren. Da sich diese Regelwerke auf den gesamten Lebenszyklus eines Gebäudes und somit auch die zugehörigen Fachdisziplinen erstrecken, gibt es eine große Anzahl von Normen, welche in der Gestaltungsplanung eines

Bauwerkes berücksichtigt und erfüllt werden müssen. Die geltenden Vorschriften sind jedoch nicht nur von dem fachlichen, sondern insbesondere auch von dem nationalen bzw. regionalen Geltungsbereich abhängig, in welchem das Bauwerk errichtet werden soll. Trotz der voranschreitenden Vereinheitlichung von Richtlinien im Bauwesen, wie z.B. durch die Harmonisierung der geltenden Regelwerke in den EU-Staaten durch die Einführung der *Europäischen Normen (EN)*, ergibt sich eine enorme Anzahl von Vorschriften, welche während der Gestaltungsplanung berücksichtigt werden müssen (DIN 2014).

Bislang handelt es sich bei der Überprüfung einer Gebäudeplanung hinsichtlich ihrer Konformität mit den Regelwerken zumeist um einen immer wiederkehrenden, manuellen Kontrollprozess in der Planungsphase eines Bauwerks, welcher sich durch hohe Fehleranfälligkeit, Arbeitsaufwand und Kosten auszeichnet. Bei diesem umständlichen Prozess werden zweidimensionale Planungsunterlagen an Hand von schriftlich niedergelegten Normen und Regelungen auf ihre Eignung hin untersucht. Der manuelle Überprüfungsprozess erfordert von seinem Bearbeiter auf der einen Seite ein hohes Maß an Spezialwissen in dem jeweiligen Fachbereich und den zugehörigen Normen, gleichzeitig aber auch eine große Sorgfalt und Umsicht hinsichtlich des Gestaltungsplanungsprozesses mit seinen immer wiederkehrenden Planungsänderungen. Darüber hinaus muss dieser Prozess nicht nur von Seiten des Gestaltungsplaners sondern auch von Seiten der Behörden durchgeführt werden.

Mit der Einführung und Entwicklung neuer digitaler Methoden, wie dem *BIM*, und einheitlicher Datenstandards für Gebäudemodelle stehen dem Bauwesen Technologien zur Verfügung, die einer Optimierung dieses Prozesses dienen können. Während des *BIM*-Prozesses entsteht im Laufe der Planungsphase eines Bauwerks ein digitales Gebäudedatenmodell, welches sämtliche aktuellen Informationen für alle Projektbeteiligten und über den gesamten Lebenszyklus des Gebäudes zur Verfügung stellt. Es bietet sich an, diese bereits gebündelten Daten für eine solche Überprüfung eines Modells auf Einhaltung von Normen und Richtlinien zu verwenden und so den Prozess effizient zu gestalten.

2.3 Ziel Arbeit

Ziel dieser Masterarbeit ist die Entwicklung einer Methode, welche eine teil- bzw. vollautomatisierte Konformitätsüberprüfung von Gebäudemodellen hinsichtlich ausgewählter Normen ermöglicht.

Bei der Konformitätsüberprüfung von Gebäudemodell und Regelwerken handelt es sich um einen nur bedingt standardisierbaren Prozess, welcher in Abhängigkeit des jeweiligen Bauwerks individuell angepasst und von einer fortlaufenden Plausibilitätsüberprüfung der erhaltenen Ergebnisse begleitet werden muss. Dieses erfordert ein hohes Maß an Kompetenz und Erfahrung des Bearbeiters im jeweiligen Fachbereich der Überprüfung.

Bisherige Ansätze für eine Automatisierung dieses Prozesses sind insofern unzureichend, da sie entweder den Informationsgehalt eines Regelwerks zu komplex oder nur ungenügend abbilden oder aber die erhaltenen Ergebnisse aufgrund mangelnder Plausibilitätsüberprüfung unbrauchbar machen.

Die Idee, die in der vorliegenden Arbeit zur Verbesserung der bestehenden Ansätze entwickelt wird, basiert auf einer graphischen Sprache, welche Struktur und Inhalt eines Überprüfungsprozesses innerhalb eines Verarbeitungssystems abbildet. Graphisch basierte Informationssysteme, welche auch als visuelle Programmiersprachen bezeichnet werden, haben sich in den letzten Jahren in vielen Bereichen der Computerwissenschaften etabliert, da sie einen anschaulichen, übersichtlichen und flexiblen Umgang mit komplexen Systemen bieten und gleichzeitig keine hohen Anforderungen an die Programmierkenntnisse des Anwenders stellen. Mit Hilfe dieser graphischen Sprache ist der Nutzer in der Lage, die Struktur eines Prozesses abzubilden, diesen bei Bedarf anzupassen und jeden einzelnen Schritt nachzuvollziehen. Es bietet sich daher an, die Konformitätsüberprüfung mit Hilfe einer solchen graphischen Sprache zu automatisieren und zu optimieren.

2.4 Aufbau der Arbeit

Zunächst soll ein Überblick über die vorhandenen Methoden und Ansätze zum automatisierten Überprüfungsprozess von Regelwerken und Gebäudedatenmodell gegeben werden. Auf Grundlage einer Analyse der Vor- und Nachteile dieser Ansätze kann anschließend eine neue Methode entwickelt werden.

Hierzu soll zunächst der gesamte Überprüfungsprozess in seine minimal notwendigen Einzelschritte untergliedert und einzeln vorgestellt werden. Auf dieser Basis können Anforderungen an die graphische Sprache hinsichtlich des Überprüfungsprozesses formuliert werden, welche wiederum den grundlegenden Rahmen für den neuen Ansatz vorgeben.

Mittels dieser Anforderungen sollen im Anschluss die einzelnen Elemente und somit die Struktur der graphischen Sprache definiert werden. Zur Identifikation der benötigten Elemente wird beispielhaft ein gebräuchliches Regelwerk aus der Baupraxis vorgestellt und hinsichtlich der Eignung zur Übersetzung in die graphische Sprache analysiert. Die erhaltenen Elemente der graphischen Sprache sollen nun im Einzelnen hinsichtlich ihrer Funktionsweise erläutert und schließlich zu einem beispielhaften Überprüfungsprozess zusammengefügt werden.

Abschließend soll das innerhalb der vorliegenden Masterarbeit entwickelte Konzept auf seine Tragfähigkeit hin untersucht werden. Hierzu wird in Kooperation mit dem Softwarehersteller *Nemetschek* und auf Basis der Online-Plattform *bim+* eine Applikation entwickelt, welche die grundlegende Arbeitsweise mit der entwickelten graphischen Sprache und den finalen Überprüfungsprozess darstellt.

in der Informatik bildet. Sie beruht auf den Überlegungen des englischen Mathematikers George Boole (1815–1864), dass jedes System auf einfachste Verknüpfungen zwischen atomaren Elementen zurückzuführen ist. Ergebnis seiner Forschungen sind die sogenannten *Booleschen Ausdrücke* (siehe Abbildung 2), mit deren Hilfe auch heute noch in vielen wissenschaftlichen Bereichen Systeme in ihre Einzelteile zerlegt werden (Schenke 2013).

\wedge	Konjunktion	für „und“
\vee	Disjunktion	für „oder“
\neg	Negation	für „nicht“
\rightarrow	Implikation	für „wenn ... dann“
\leftrightarrow	Äquivalenz	für „genau dann ... wenn“

Abbildung 2: Formale Symbole der Aussagenlogik (Schenke 2013)

Die Aussagenlogik kann durch die Verwendung dieser logischen Operatoren sowohl Aussagen zur Mengenlehre, als auch zur logischen Falsifizierbarkeit einer Aussage selbst treffen. Hierfür stehen neben den Operatoren zusätzlich die beiden Konstanten *true* und *false* zur Verfügung, die jeweils identifizieren, ob eine Aussage vollständig zutrifft oder nicht. Mit diesen drei Elementen der Syntax - Mengenlehre, logischen Operatoren und Falsifizierbarkeit – ist eine Grundlage geschaffen, um Inhalte einer Vorschrift zu formalisieren (Schenke 2013).

Eine weitverbreitete Weiterentwicklung dieser Logik und Syntax ist die Prädikatenlogik, auch *First Order Logic* genannt, die ebenfalls über die drei grundlegenden Komponenten der Aussagenlogik verfügt, jedoch darüber hinaus eine Quantifizierung von Aussagen vornehmen kann. Die sogenannten *Quantoren* - definiert als logische Symbole \forall *Allquantor* und \exists *Existenzquantor* - sind vergleichbar mit den Sprachfragmenten „für alle“ oder „es gibt“ und helfen bei der Formulierung, ob eine Aussage auf eine Gesamt- oder Teilmenge zutrifft. Dadurch lässt sich die logische Falsifizierung viel individueller gestalten, da diese nicht nur Aussagen über wahr (*true*) oder falsch (*false*) trifft, sondern diese gleichzeitig auf eine definierte Menge begrenzt. Ein Beispiel für eine in der Prädikatenlogik formulierte mathematische Aussage wäre (Schenke 2013):

$$\forall X \forall Y (X < Y) \rightarrow (X + 1 < Y + 1)$$

„Für alle *X* und für alle *Y* gilt $X < Y$. Daraus folgt, dass $X+1 < Y+1$.“

Darüber hinaus können mit Hilfe der *First Order Logic* über sogenannte nichtlogische Symbole auch außermathematische Aussagen formuliert werden. Ein Beispiel hierzu (Schenke 2013):

```
gibt(haensel, gretel, ein_brot)
```

„Hänsel gibt Gretel ein Brot.“

Die Bestandteile dieser Aussage, *haensel*, *gretel*, *ein_brot* und *gibt()* werden als nichtlogische Bestandteile bezeichnet, da sie sich außerhalb einer mathematischen Formulierung bewegen. Das Gebilde *gibt()* wird als Prädikatssymbol P bezeichnet und fungiert als Funktion, die für eine bestimmte Anzahl von Objekten eine Aussage formuliert.

Das Beispiel stellt lediglich eine Instanziierung der allgemeinen Formel $\text{gibt}(X, Y, Z)$ mit den Objekten `haensel`, `gretel` und `ein_brot`, welche als Funktionssymbole p (oder q) bezeichnet werden, dar. Eine Aussage mit Prädikats- und Funktionssymbol gemäß der Prädikatenlogik kann also allgemein als

$$P(p, q, [\dots])$$

definiert werden, wobei $p, q, [\dots]$ eine beliebige Menge an Funktionssymbolen darstellt. Mit dieser Definition können sowohl neue Aussagen formuliert, als auch bestehende in ihre Bestandteile zerlegt werden (Schenke 2013).

Die Aussagen- und die Prädikatenlogik stellen die beiden wichtigsten logischen Formalismen der Informatik und weiterer Wissenschaften dar, mit welcher Aussagen sowohl für Mensch als auch Maschine les- und interpretierbar formuliert werden können. Ein Beispiel für eine etablierte Implementierung der Prädikatenlogik in den Computerwissenschaften ist die logische Programmiersprache *PROLOG* (Eastman, Lee, et al. 2009b, Schenke 2013).

Neben der *First Order Logic* gibt es noch weitere logische Familien, die auf der Aussagenlogik aufbauen und weitere logische Elemente in die Syntax einführen, um Aussagen auf andere Art und Weise präzise zu formulieren. In den nachfolgenden Unterkapiteln werden unterschiedliche Ansätze vorgestellt, die mittels solcher logischer Systeme Inhalte von Regelwerken formalisieren und so den Grundstein für das *Code Compliance Checking* legen.

3.1.1.1 Entscheidungstabellen

Einen ersten Einsatz eines logischen Systems für die automatisierte Konformitätsüberprüfung stellen die Bestrebungen in den USA beginnend im Jahre 1969 dar, die Regularien des *American Institute of Steel Construction* (2014) im Bereich des Stahlbaus in eine von Maschinen interpretierbare Sprache zu übersetzen. Die Basis dieser Methode sind die sogenannten *decision tables* (*Entscheidungstabellen*), welche einzelne Regeln der Gestaltungsplanung als logische Entscheidungsfälle abbilden. Der Ansatz erfüllt durch die zwingend beinhaltete logische Falsifizierbarkeit der abgebildeten Entscheidungsfälle, also die eindeutige Zuordnung, ob die jeweilige Aussage wahr oder falsch ist, nicht nur die Lesbarkeit für Mensch und Maschine, sondern gibt auch für komplexe Entscheidungsfälle durch das Zusammenwirken vieler einzelner Regeln innerhalb einer Entscheidungstabelle einen eindeutigen Rückgabewert und bildet den Inhalt eines Regelwerks somit präzise ab (Nawari 2012). Ein Beispiel für eine solche Entscheidungstabelle ist in Abbildung 3 dargestellt.

Conditions					Actions			
Location	Type	Wall position with respect to ground	Primary heating source	Thermal resistance value: R-value (RSI)	Compliance checking result	Comments	Reference index	
Toronto (Region A, ON)	House	Above	Electricity	$R\text{-value} \geq 4.4$	Pass	Good	MNECH 3.3.1.1 (Toronto) Web Link 1	
			Propane, oil, heat pump	$R\text{-value} \geq 3.0$	Pass	Good		
			Natural gas	$R\text{-value} \geq 2.9$	Pass	Good		
			Electricity	$R\text{-value} < 4.4$	Fail	Increase insulation layers so that $R > = 4.4$ RSI		
			Propane, oil, heat pump	$R\text{-value} < 3.0$	Fail	Increase insulation layers so that $R > = 3.0$ RSI		
			Natural gas	$R\text{-value} < 2.9$	Fail	Increase insulation layers so that $R > = 2.9$ RSI		
			Below	Electricity	$R\text{-value} \geq 3.1$	Pass	Good	MNECH 3.3.2.1 (Toronto) Web link 1
				Propane, oil, heat pump	$R\text{-value} \geq 3.1$	Pass	Good	
		Natural gas		$R\text{-value} \geq 2.1$	Pass	Good		
		Electricity		$R\text{-value} < 3.1$	Fail	Increase insulation layers so that $R > = 3.1$ RSI		
		Propane		$R\text{-value} < 3.1$	Fail	Increase insulation layers so that $R > = 3.1$ RSI		
		Natural gas		$R\text{-value} < 2.1$	Fail	Increase insulation layers so that $R > = 2.1$ RSI		
		Building	Any	Any	Any	Exceptional	N.A.	MNECB Web link 2

Note: N.A. = not applicable.

Abbildung 3: Entscheidungstabelle zur Überprüfung des Wärmedurchgangswiderstandes einer Außenwand in Toronto (Tang, Hammad und Fazio 2010)

Eine erste umfassende Sammlung von solchen Entscheidungstabellen wurde im Jahre 1984 vom *US National Bureau of Standards* (heute *NIST*) mit dem sogenannten *SASE*-Datenmodell geschaffen. Mit Hilfe des *SASE* können Entscheidungstabellen aus den verschiedenen Fachdisziplinen des Bauwesens gesammelt, gespeichert und verwaltet werden. Somit stellt dieses Datenmodell den ersten Schritt in Richtung einer automatisierten Konformitätsüberprüfung dar (Dimyadi und Amor 2013).

Auf Grundlage der Entwicklung der Entscheidungstabellen und des *SASE* wurden mehrere Applikationen entwickelt, welche das Prinzip der Entscheidungstabellen in die Praxis überführen.

So wurde im Jahre 1984 von der *Carnegie Mellon University* die Applikation *STEEL-3D* vorgestellt, welche die Regelwerke des *AISC* im Bereich der Gestaltungsplanung von Stahlrahmen mit Hilfe der Entscheidungstabellen in ein *CAD*-System integrierte und auf diese Weise eine erste Konformitätsüberprüfung direkt während der Gestaltungsphase des Bauwerks ermöglichte (Dimyadi und Amor 2013).

Eine Integration des *SASE*-Datenmodells in eine *CAD*-Umgebung wurde von Lopez (1985) durch das *Standards Interface for Computer Aided Design (SICAD)* geschaffen. *SICAD* unterstützt den Planer nicht nur durch die Bereitstellung der Regelwerke während der Gestaltungsplanung, sondern fördert vielmehr eine Interaktion zwischen Nutzer und System, indem es bei fehlenden oder widersprüchlichen Informationen Nutzereingaben einfordert.

Eine weitere Implementierung des *SASE*-Datenmodells wurde im Jahre 1986 mit dem *Standards Processing Expert (SPEX)* entwickelt, welche sich insbesondere auf Konformitätsüberprüfungen im Bereich der Gestaltungsplanung von Material und Geometrie konzentriert (Dimyadi und Amor 2013).

3.1.1.2 Fire-Code Analyzer

Nach einem ähnlichen Prinzip wie dem der Entscheidungstabellen, verfolgen E. A. Delis und A. Delis (1995) mit ihren Entwicklungen zu dem Überprüfungswerkzeug *Fire-Code Analyzer (FCA)*, einen regelbasierten Ansatz für den Überprüfungsprozess. Mit dem *FCA* können einzelne konditionale Entscheidungsfälle mit Hilfe einer Implikation, also in der „Wenn-Dann“-Form, abgebildet und durch eine eigene Syntax deutlich individueller und detaillierter ausgestaltet werden. Durch die vorgegebene Syntax und Vorgehensweise ist die somit formulierte Vorschrift ebenfalls logisch falsifizierbar und kann auf ihre Gültigkeit hin überprüft werden.

Für die digitale Abbildung von Regelwerken ist das sogenannte Regel-System zuständig, welches aus einer Wissensdatenbank und einem Interpreter besteht. Die Datenbank des *FCA* ist auf die Paragraphen des *Life Safety Code* (National Fire Protection Association 2005), einem weltweit anerkannten Sicherheitsstandard im Bereich des Feuerschutz, ausgerichtet und kann eine Vielzahl von einzelnen, unabhängigen Regelwerken speichern und verwalten.

Jede der im *FCA* formulierten Regeln besteht aus einem Bedingungs- und Konsequenz-Objekt, welche zueinander in Abhängigkeit stehen. Nur wenn die voranstehende Bedingung, die auch aus mehreren Komponenten bestehen kann, erfüllt ist, kann auch die Konsequenz erfüllt sein. Das Konsequenz-Objekt bildet in der Regel ein Ergebnis oder eine Aktion, welche zu dem Resultat der Überprüfung führt (Delis und Delis 1995). Als Beispiel für eine Übersetzung eines solchen Regelwerks, soll an dieser Stelle der Paragraph 12.3.1 der *National Fire Protection* (2005) dienen:

“The fire resistance rating of enclosures in health care occupancies protected throughout by an approved automatic sprinkler system may be reduced to 1 hour in buildings up to an including, three stories in height.”

Eine Übersetzung dieses Paragraphen in die Syntax des *FCA* ist in Code 1 dargestellt.

Code 1: Abgebildete Norm NFPA 12.3.1 in der FCA-Syntax (Delis und Delis 1995)

```

1      (IF      (AND      (THE SPRINKLER-PRESENCE OF A BUILDING IS YES)
2                        (?SPACE IS IN CLASS VERTICAL-OPENINGS)
3                        (THE SUPERSPACE OF ?SPACE IS ?ZONE)
4                        (?ZONE IS IN CLASS FIRE-ZONES)
5                        (LISP « (THE STORIES-ABOVE-GRADE OF BUILDING) 4)))
6      (THEN   (A REQUIRED-ENCLOSURE-RATING OF ?SPACE IS 1))

```

In Zeile 1 bis 5 sind durch das Signalwort **IF** alle Bedingungen definiert, welche erfüllt sein müssen, damit die Konsequenz – markiert durch das Signalwort **THEN** – in Zeile 6 zum Tragen kommt. Dieses bedeutet, dass der Regel-Interpreter zunächst alle Anforderungen der Bedingungen überprüft und damit beginnt, beispielsweise die Anweisung aus Zeile 1, also die Existenz einer Sprinkleranlage, zu überprüfen. Sind nun alle Bedingungen erfüllt, kann die Konsequenz in Zeile 6 in Kraft treten und der Feuerwiderstand schließlich um eine Stunde gemindert werden.

Um die in der Regel enthaltenen Informationsabfragen durchführen zu können, muss auch der entsprechende Informationsgehalt von einem Datenmodell zur Verfügung gestellt werden.

Hierfür ist in dem *FCA* das sogenannte *frame system* verantwortlich, welches die Informationen eines Gebäudemodells als einzelne Objekte mit zugehörigen Parametern zur Verfügung stellt. Da die im *FCA* behandelten Regelungen stark geometrisch orientiert sind, gibt es überdies einen Satz von geometrischen Algorithmen, mit deren Hilfe zusätzliche Informationen erfasst werden können. Jedes einzelne Objekt des Datenmodells besitzt neben seinen festgelegten Parametern auch eine Liste von anwendbaren geometrischen Operationen, die weitere Informationen über einen festgelegten Algorithmus aufbereiten (Delis und Delis 1995). Eine solche Liste ist beispielhaft für das Datenobjekt Tür in Abbildung 4 dargestellt.

Frame : DOOR1			
Slot	ValueClass	Source	Comment
CONNECTED_SPACE	NIL	CALC	ROOMS and CORRIDORS connected by this DOOR
NET_BREADTH	NUMBER	INP	Feet
RATING	NUMBER	INP	Hours
SWINGS_INTO	NIL	INP	ROOM or CORRIDOR into which this DOOR swings
TRAVEL_DIS_TO_EXIT	NUMBER	CALC	Travel Distance in feet
BREADTH	NUMBER	CALC	Gross Breadth in feet
FCA_PROBLEM	STRING	CALC	Description of Problem
COORDS	NIL	INP	(X,Y) or (X,Y,Z)
INROOM_TRAVEL_DIS	NUMBER	CALC	Travel Distance in feet
AREA	NUMBER	CALC	Square feet
WALL	STRING	CALC	Name of WALL in which opening occurs
HEIGHT	NIL	INP	Measured in feet
NUM_LEAVES	NUMBER	INP	Number of leaves
DOOR_TYPE	NIL	INP	Special features of door (e.g., smoke barrier)
EXIT_PATH	NIL	CALC	List of Nodes
VISION_PANEL	NIL	CALC	Name of DOOR's vision panel, if any

Abbildung 4: Datenobjekt Tür mit den zugehörigen Parametern (Source = „INP“) und geometrischen Algorithmen (Source = „CALC“) im FCA (Delis und Delis 1995)

3.1.1.3 Deontische Logik

Wie bereits in Abschnitt 3.1.1 beschrieben gibt es neben der weitverbreiteten Prädikatenlogik auch noch weitere Formen logischer Familien, die auf der Aussagenlogik aufbauen und sich für die Formulierung von Aussagen eignen.

Der Begriff Deontologie stammt von den beiden altgriechischen Wörtern $\delta\epsilon\omicron\nu$ „Aufgabe/Pflicht“ und $\lambda\omicron\gamma\omicron\varsigma$ „Lehre“. Die Deontologie, die auch Deontische Ethik oder Pflichtethik genannt wird, umfasst sämtliche Theorien, die sich mit Pflichten und Rechten von Handlungen beschäftigen (Schenke 2013). Wie bereits bei der Prädikatenlogik (siehe Abschnitt 3.1.1) beschrieben lassen sich aus den Fragestellungen dieser philosophischen Theorie Elemente für eine logische Familie - der deontischen Logik - ableiten und als Elemente zu der Syntax der Aussagenlogik hinzufügen. Dabei ist zu beachten, dass sich die Elemente von Prädikaten- und deontischer Logik nicht widersprechen, sondern beide auf ihre Weise lediglich die Syntax erweitern. Daher ist auch eine Kombination verschiedener logischer Familien auf Basis der Aussagenlogik möglich.

In der deontischen Logik können die logischen Elemente mit den Sprachfragmenten „etwas ist obligatorisch“, „etwas ist erlaubt“ oder „etwas ist verboten“ erfasst werden. In der deontischen Logik werden diese als Symbole \circ („es ist obligatorisch, dass“), F („es ist verboten, dass“) und P („es ist erlaubt, dass“) zu der Formulierungssyntax hinzugefügt (Schenke 2013, Salama und El-Gohary 2013). An dieser Stelle soll ein Beispiel das Prinzip der deontischen Logik aufzeigen, wobei G ein Element der klassischen Aussagenlogik darstellt (Schenke 2013):

$$OG \leftrightarrow \neg P \neg G$$

„Wenn Aussage G obligatorisch ist, folgt daraus, dass es nicht erlaubt ist, dass G nicht zutrifft.“

$$FG \leftrightarrow \neg PG$$

„Wenn Aussage G verboten ist, folgt daraus, dass es nicht erlaubt ist, dass G zutrifft.“

Bereits in dem allgemeinen Beispiel ist zu erkennen, dass eine sehr ähnliche Art der Formulierung auch in Regelwerken und Vorschriften verwendet wird. Normen im Bauwesen stellen in der Regel zunächst einen Sachverhalt dar und fordern anschließend die Einhaltung von Randbedingungen und Grenzwerten ein. Diese Forderung kann mit Hilfe der deontischen Symbole sehr direkt und präzise formuliert werden. Daher eignet sich die deontische Logik prinzipiell sehr gut, um den Informationsgehalt von Vorschriften abzubilden.

Grundlagenforschung hierzu betreiben Salama und El-Gohary (2013) mit ihrem Ansatz, den Informationsgehalt von Normen mit Hilfe der deontischen Logik zu formulieren und so einen Soll-Ist-Vergleich für ein Gebäudemodell zu ermöglichen.

Wie in Abbildung 5 dargestellt ist, wird zunächst der Inhalt der einzelnen Regeln direkt von dem Fließtext mittels der Syntax in eine deontische Vorschrift übersetzt.

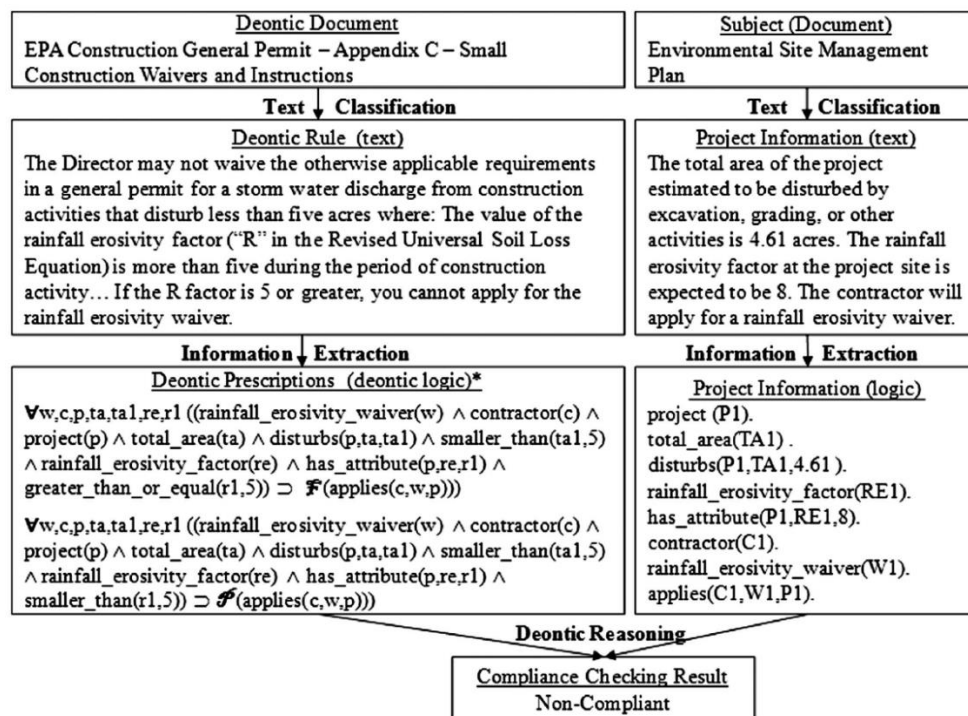


Abbildung 5: Beispiel für eine Übersetzung einer Vorschrift mit der Syntax der deontischen Logik (Salama und El-Gohary 2013)

Zwar kann so die Aussage einer Vorschrift sehr präzise formuliert werden, jedoch ergibt sich bei dem Zusammenspiel mehrerer Aussagen innerhalb eines Regelwerkes das Problem, dass sich diese ohne Klassifizierung gegenseitig widersprechen und entkräften können. Hierzu geben Salama und El-Gohary (2013) für jedes Teilelemente ihres Überprüfungsprozesses eine

Hierarchie (siehe Beispiel in Abbildung 6) vor, die den Unterelementen eine Priorität zuordnet und somit eine Rangfolge definiert.

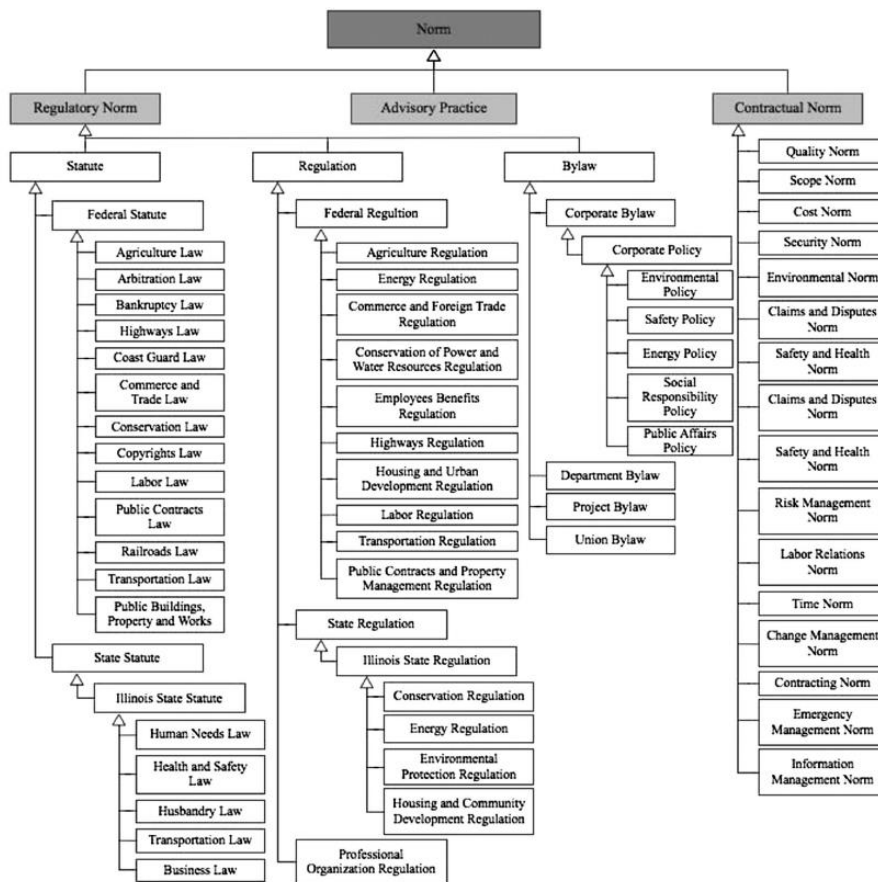


Abbildung 6: Hierarchisches Modell für das Element „Norm“ (Salama und El-Gohary 2013)

Wie die einzelnen Elemente des Prozesses nun miteinander ins Verhältnis gesetzt werden können, wird in dem sogenannten „deontischen Modell der höchsten Stufe“ (siehe Abbildung 7) definiert. In diesem Relationen-Modell ist aufgeführt, wie die einzelnen Elemente des Überprüfungsprozesses jeweils gegenseitig voneinander abhängig sind.

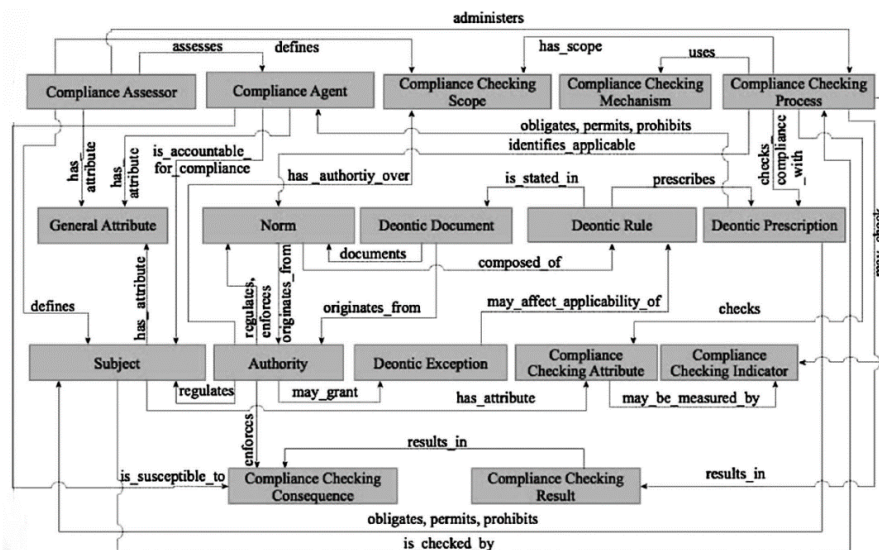


Abbildung 7: Deontisches Modell des Überprüfungsprozesses (Salama und El-Gohary 2013)

Ein vollständiger Prozessverlauf einer automatisierten Konformitätsüberprüfung ist schließlich in Abbildung 8 dargestellt.

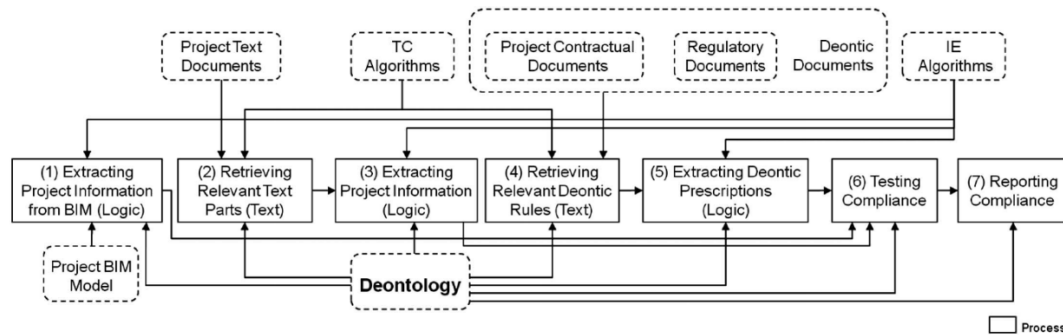


Abbildung 8: Ablauf einer (Salama und El-Gohary 2013)

Der Ansatz von Salama und El-Gohary (2013) ist der erste Versuch, eine automatisierte Konformitätsüberprüfung für das Bauwesen mit Hilfe der deontischen Logik durchzuführen. Dabei zeigen sie auf, dass die deontischen Logik ein großes Potential besitzt, Vorschriften präziser abzubilden als es mit anderen logischen Systemen möglich ist. Die Entwicklungen rund um diesen Ansatz stehen jedoch erst bei ihren Anfängen und es gibt bisher keine praxisrelevanten Tragfähigkeitsstudien (Salama und El-Gohary 2013).

3.1.2 Sprachbasierte Ansätze

Die Formulierung der Inhalte von Normen auf Basis einer logischen Familie, wie sie mit einigen Beispielen in Abschnitt 3.1.1 vorgestellt wurde, garantiert eine eindeutige Falsifizierbarkeit für einzelne Entscheidungsfälle. Jedoch können diese Formulierungen mit zunehmender Komplexität der abgebildeten Vorschrift deutlich an Übersichtlichkeit verlieren, so dass eine Lesbarkeit von Seiten des Menschen nur noch schwer möglich ist und der Aufwand für die Übersetzung stark zunimmt. Eine Lösung dieses Problems stellen die sprachbasierten Ansätze zum *Code Compliance Checking* dar, die einen größeren Freiraum für die Formulierungen von Aussagen einräumen.

Die Idee hinter diesem Ansatz ist, die Formulierungssyntax um weitere domänenbasierte Elemente zu erweitern und somit auf das Niveau einer Sprache zu heben. Auf dieser Stufe kann die Interaktion zwischen Mensch und Maschine bei dem Übersetzungsprozess deutlich besser und präziser gestaltet werden während die logische Falsifizierbarkeit gewahrt bleibt.

Die Verwendung einer solchen Sprache ermöglicht die Abbildung deutlich komplexerer Sachverhalte auf einfachere Art und Weise, jedoch nimmt die Gefahr von Inkonsistenzen, wie z.B. Fehlinterpretationen, durch den weiter gesteckten Rahmen der Syntax deutlich zu. Damit die Konsistenz dennoch gewahrt bleibt, muss der Syntax der Sprache ein fester Rahmen gegeben werden, an welchen sich der Anwender bei seinen Übersetzungen halten muss.

Dieser Rahmen ist maßgeblich von der jeweiligen Domäne abhängig, in welcher sich die Sprache bewegt. Innerhalb der vorliegenden Arbeit beziehen sich alle Ansätze insbesondere auf den Bereich des Bauwesens oder aber dessen Teilbereiche, z.B. Fachdisziplinen. Daher ist es sinnvoll, der Syntax ausschließlich für diesen Bereich feste Definitionen vorzugeben und so den Handlungsrahmen des Anwenders zu begrenzen (Eastman, Lee, et al. 2009b).

Die folgenden Unterkapitel führen unterschiedliche Formen von sprachbasierten Ansätzen im Bereich des *Code Compliance Checking* auf.

3.1.2.1 Auszeichnungssprachen

Bereits seit langer Zeit dienen in den Computerwissenschaften Auszeichnungssprachen als weit verbreitetes Austauschformat von Informationen zwischen verschiedenen Computersystemen. Ziel einer sogenannten *Markup Language* (WC3 2014) ist es, den Inhalt eines Dokuments gemäß einer definierten Syntax so zu kennzeichnen und zu strukturieren, dass dieser für Maschinen les- und interpretierbar wird. Im Gegensatz zu Programmiersprachen stehen bei den Auszeichnungssprachen keine Befehle oder Deklarationen im Vordergrund, sondern lediglich die Übermittlung von Informationen (Liu und Özsu 2009).

Einer der bekanntesten Vertreter der Auszeichnungssprachen ist die *Extensible Markup Language* (*.xml), die plattform- und implementationsunabhängig zwischen Computersystemen eingesetzt werden kann und als offener Standard im Internet weit verbreitet ist (Liu und Özsu 2009). Zwar kann die XML im Prinzip vollkommen offen verwendet werden, jedoch gibt es Spezifikationen, wie die des *World Wide Web Consortium* (WC3 2014), die grundlegende Regeln im Umgang mit dieser Sprache vorgeben.

Code 2: Beispiel für eine Übersetzung einer Regel mit LegalRuleML (Palmirani, et al. 2011)

“A customer is “Premium” if their spending has been min 5000 dollars in the previous year.”

```

1   <Assert mapClosure="universal">
2       <Implies timesBlock="#t2" ruleType="defeasible" id="rule1">
3           <then timesBlock="#t1">
4               <Atom id="atm1">
5                   <Rel>premium</Rel>
6                   <Var>customer</Var>
7               </Atom>
8           </then>
9           <if timesBlock="#t1">
10              <Atom id="atm2" timesBlock="#t3">
11                  <Rel>previous year spending</Rel>
12                  <Var>customer</Var>
13                  <Var>x</Var>
14                  <Data>= 5000$ </Data>
15              </Atom>
16          </if>
17      </Implies>
18  </Assert>

```

Neben diesem domänenunabhängigen, offenen Standards gibt es eine Vielzahl von Abwandlungen der Auszeichnungssprache, die sich durch ihre Syntax speziell für bestimmte Wissensbereiche eignen. So gibt es beispielsweise den *RuleML*-Standard, welcher sich ausschließlich mit der Formulierung von Regelungen und Vorschriften beschäftigt (RuleML 2014, Palmirani, et al. 2011). *LegalRuleML* (OASIS 2014) ist eine Weiterentwicklung dieses Standards, welcher sich mit der Formulierung von rechtswissenschaftlichen Inhalten, also z.B. Gesetzen oder Verordnungen, beschäftigt. Ein Beispiel einer solchen *LegalRuleML*-Formulierung ist in Code 2 dargestellt.

Solche Spezifizierungen und Abwandlungen der Auszeichnungssprache sind auch für Datenmodelle des Bauwesens entwickelt worden. Adachi (2002) stellt die sogenannte *Partial Model Query Language (PMQL)* vor, die über die *Markup Language* nicht nur Informationen, sondern auch Zugriffs- und Suchanfragen für Gebäudemodelle formulieren kann.

Eine *Query Language* wird im Allgemeinen als eine spezifische Programmiersprache definiert, die sich für den Zugriff von Inhalten einer Datenbank eignet (Liu und Özsu 2009). In der Regel haben alle diese Sprachen die drei grundlegenden Zugriffsoperatoren `select` („Auswählen“), `update` („Aktualisieren/Anpassen“) und `delete` („Löschen“) gemeinsam. In seiner *PMQL*-Syntax verwendet Adachi (2002) diese Operatoren, stellt dem Anwender aber auch gleichzeitig weitere domänenspezifische Operatoren zur Verfügung, die sich insbesondere an der Struktur des offenen Gebäudemodellformats *IFC* orientieren. Auf diese Weise ist es dem Anwender möglich, auch komplexere Suchanfragen oder aber Vorschriften für das Modell zu formulieren.

Code 3: Abfrage einer Entität mit der PMQL (Adachi 2002)

```

1 <pmql>
2   <select type="entity" match="IfcSpace" action="get">
3     <cascades>
4       <select type="attribute" match="LocalPlacement" action="get"/>
5       <select type="attribute" match="BoundedBy" action="get"/>
6     </cascades>
7   </select>
8 </pmql>

```

In Code 3 ist eine *PMQL*-Abfrage dargestellt, welche über die `select`-Operation auf alle Entitäten des Typus `IfcSpace` (siehe Zeile 2) des Gebäudemodells zugreift, um anschließend deren Attribute `LocalPlacement` und `BoundedBy` (siehe Zeile 4 & 5) abzufragen. Das Ergebnis der Suchanfrage ist eine Auflistung dieser beiden Attribute sämtlicher Instanzen vom Typus `IfcSpace`, welche in dem Gebäudemodell enthalten sind.

Auf ähnliche Art und Weise lassen sich mit der *PMQL* auch Vorschriften oder Regelungen formulieren. In Code 4 wird hierzu der `where`-Operator verwendet, um die Suchanfrage nach der Entität `IfcWall` (siehe Zeile 2) zu präzisieren. *PMQL* erlaubt es, innerhalb dieses Operators Ausdrücke mit Hilfe der *Structured Query Language (SQL)*, eine der bekanntesten Vertreter der *Query Languages* (Liu und Özsu 2009), zu formulieren. Die beiden Ausdrücke in Zeile 4 und 5 sind Beispiele für solche *SQL*-Ausdrücke, mit denen die Suchanfrage präzisiert wird.

Code 4: Präzisierung der Abfrage mit SQL-Elementen (Adachi 2002)

```

1 <pmql>
2   <select type="entity" match="IfcWall" action="get">
3     <where>
4       <expr value="Label LIKE 'Wall#%'" />
5       <expr value="calcWallVolume < 3.0" />
6     </where>
7   </select>
8 </pmql>

```

Schließlich können mit Hilfe dieser und noch weiterer Elemente der *PMQL* auch komplexere Suchanfragen formuliert werden. Ein Beispiel einer solchen umfassenden Suchanfrage ist in Code 5 dargestellt.

Code 5: Umfassende PMQL-Abfrage (Adachi 2002)

```

1 <pmql>
2 <select type="entity" match="IfcWall" action="get">
3   <where>
4     <expr value="Label LIKE 'Wall#%'" />
5     <expr value="calcWallVolume > 7.0" />
6   </where>
7   <cascades>
8     <select type="attribute" match="LocalPlacement" action="get">
9       <cascades>
10        <select type="attribute" match="PlacementRelTo" action="get" />
11        <select type="attribute" match="RelativePlacement" action="get">
12          <cascades>
13            <select type="attribute" match="Location" action="get" />
14            <select type="attribute" match="Axis" action="get" />
15            <select type="attribute" match="RefDirection" action="get" />
16          </cascades>
17        </select>
18      </cascades>
19    </select>
20  </cascades>
21 </select>
22 </pmql>

```

Der zeitliche Ablauf einer *PMQL*-Anfragen kann auch als Graph dargestellt werden. In Abbildung 9 sind die einzelnen Schritte und die Ergebnisse des Beispiels aus Code 5 als Graph aufgeführt.

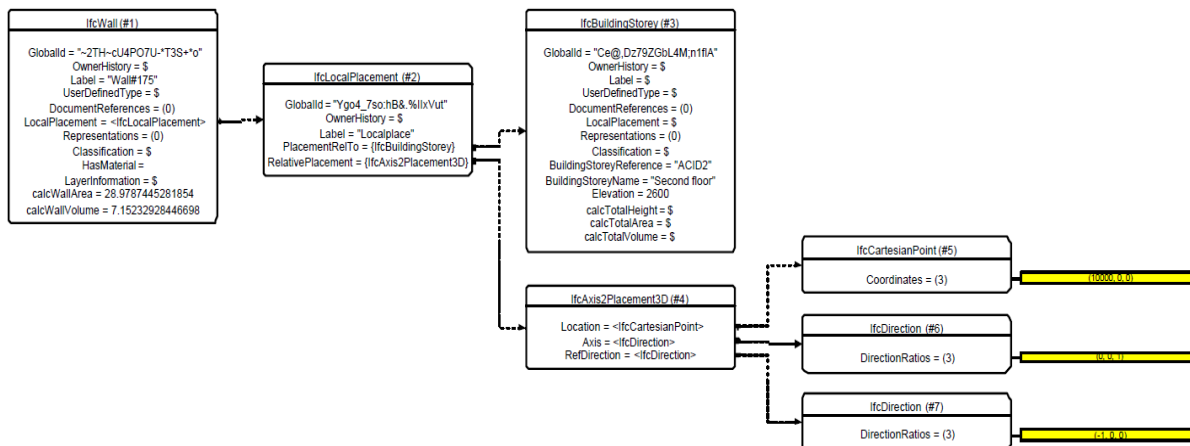


Abbildung 9: Schematische Darstellung der Ergebnisse der Suchanfrage mit PMQL (Adachi 2002)

Eine weitere Verwendung finden die Auszeichnungssprachen in der Entwicklung des Tools *ifcModelCheck* von Ebertshäuser und von Both (2013). Für das *Bundesamt für Bauwesen und Raumordnung (BBR)* entwickelten sie eine Methode, die Planungen aller Projektbeteiligten eines Pilotprojektes gemäß ihrer Konformität hinsichtlich der *Dokumentationsrichtlinien für den deutschen Gebäudebestand* automatisiert zu überprüfen.

Grundidee dieses Ansatzes ist die Identifikation der Elemente *Selection*, *Property*, *Criterion* und *Target Value* innerhalb der Vorschriften, um so den Informationsgehalt einer Anforderung

zu formalisieren. Zur Erleichterung der Formulierung der Vorschriften stehen dem Anwender die Objekt- und Datentypen des *IFC*-Datenmodell sowie die Elemente der *Object Constraints Language (OCL)*, einer Sprache zur Beschreibung von Randbedingungen bei der Modellierung von Informationssystemen, zur Verfügung. Die somit formulierten Vorschriften können anschließend in das *XML*-Format übertragen und gespeichert werden. Der gesamte Übersetzungsprozess ist schematisch in Abbildung 10 dargestellt.

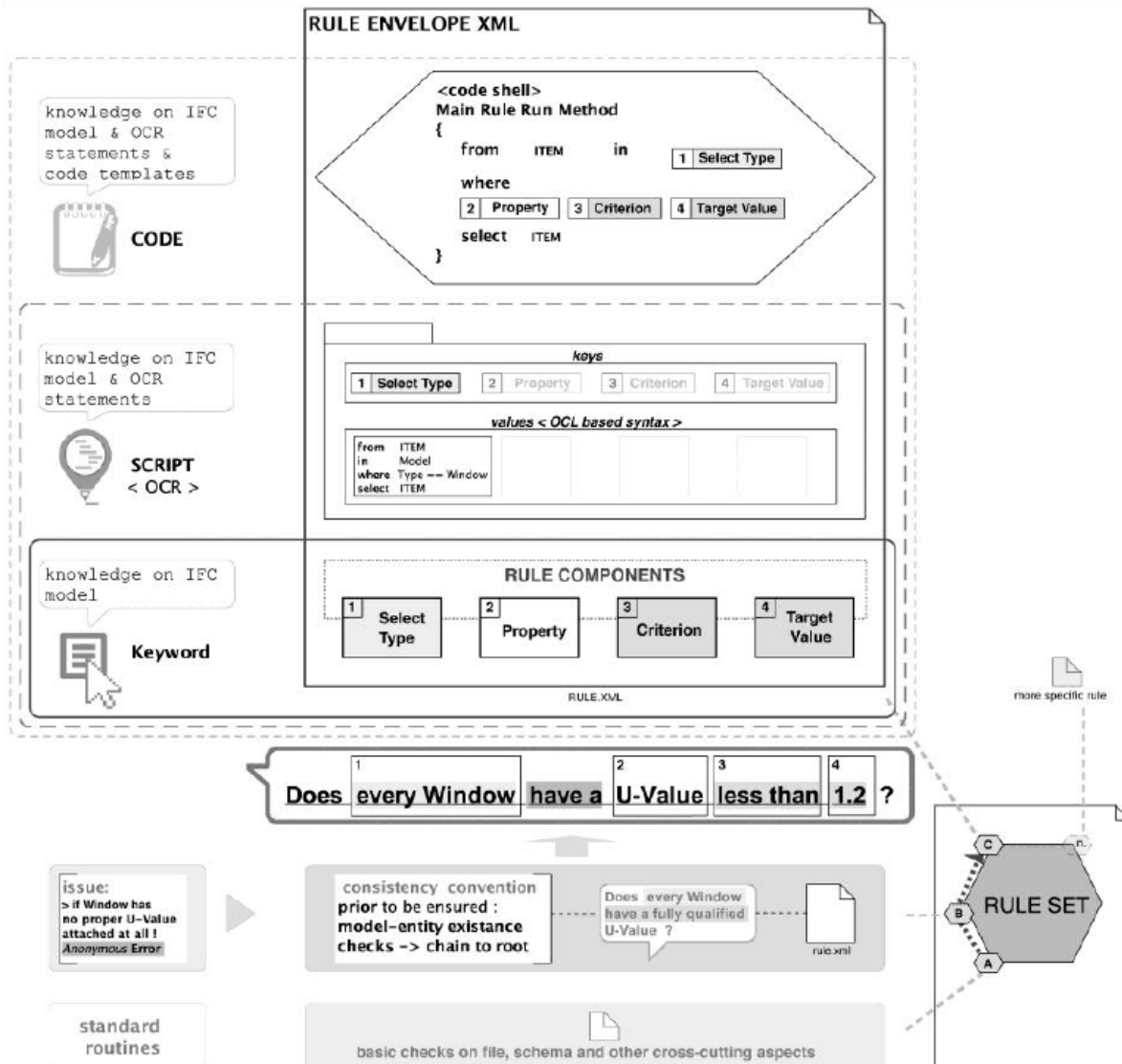


Abbildung 10: Softwarekonzept für die Abbildung der Inhalte eines Regelwerkes mit Hilfe von XML (Ebertshäuser und von Both 2013)

3.1.2.2 SMARTCodes & AEC3 RASE tools

Unabhängig von der jeweiligen Fachdisziplin wiederholen sich in Normen immer wieder einzelne Elemente, wie beispielsweise ingenieurwissenschaftliche Parameter oder deren Berechnungsmethoden. Dieser Umstand bringt ein hohes Optimierungspotential bei der Übersetzung der schriftlich niedergelegten Regelwerke in eine maschineninterpretierbare Sprache mit sich und legt nahe, die wiederkehrenden Objekte zu vereinheitlichen. Mit Hilfe einer Bibliothek solcher Elemente, die einen wiederholten Einsatz erlaubt, kann der Übersetzungsprozess deutlich vereinfacht und beschleunigt werden.

Mit der Entwicklung der *SMARTCodes* im Jahre 2006 beabsichtigte die *International Code Council* (2014) eine solche Vereinheitlichung mittels eines festgelegten Datenaustausch-Protokolls zu ermöglichen und gleichzeitig eine engere Verknüpfung zwischen schriftlichem Regelwerk und übersetztem Objekt herzustellen. Zwar wurde die Entwicklung von Seiten der *ICC* bereits im Jahre 2010 aufgrund der fehlenden Folgefinanzierung wieder eingestellt, jedoch wurde das Projekt von den Firmen *AEC3* (2014) und *DigitalAlchemy* (2014) übernommen und weiterentwickelt (Dimyadi und Amor 2013).

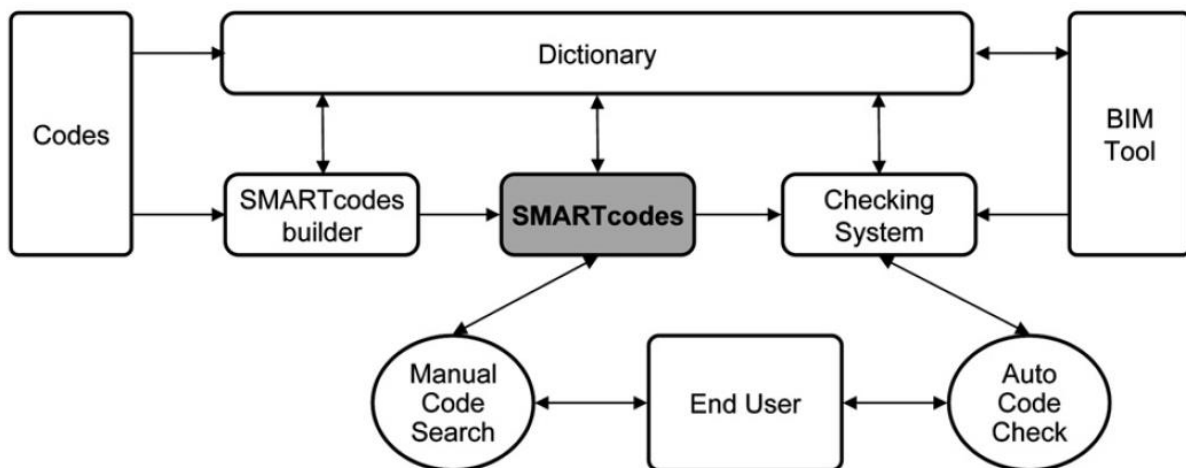


Abbildung 11: Struktur eines Überprüfungsprozesses im SMARTCodes-System (Eastman, Lee, et al. 2009b)

Wie in Abbildung 11 dargestellt stellt den Kern des Systems das Datenaustauschformat *SMARTCodes* selber dar. In diesem Kern werden die Inhalte von Normen, also die einzelnen Vorschriften und Regelungen in einer Form der *Markup Language* (siehe Abschnitt 3.1.2.1) gespeichert.

Die aktuelle Version dieser Sprache, das sogenannte *RASE*, wurde von *AEC3* entwickelt. Das grundlegende Prinzip der *RASE* ist die Strukturierung und Aufteilung der Inhalte einer einzelnen Vorschrift mittels der vier *RASE*-Objekten *Requirement* (Anforderungen), *Applicability* (Geltungsbereich), *Select* (Auswahl) und *Exceptions* (Einschränkungen). Ziel dieser Syntax ist es, in dem Fließtext einer Vorschrift diese *RASE*-Objekte zu identifizieren, diese als solche zu markieren und so die Struktur der Vorschrift herauszuarbeiten. Auf diese Weise sollen auch komplexe Inhalte einer Norm in ihre einzelnen Bestandteile zerlegt, definiert und somit für Maschinen interpretierbar gemacht werden können (Hjelseth und Nisbet 2011).

Als Beispiel für eine solche Formalisierung gemäß der *RASE*-Syntax ist die Struktur der Norwegischen Norm *NS 11001-1:2009*, welche die Barrierefreiheit von Gebäuden regelt, in Code 6 dargestellt.

Code 6: Formalisierung der Norwegischen Norm NS 11001-1:2009 gemäß der RASE-Syntax
(Hjelseth und Nisbet 2011)

<R>Standard NS 11001-1, Clause: 5.2 Dimensioning an <a>access route to a building

<R>The <a>access route for <s>pedestrians</s> <s>wheelchair users</s> shall
<r>not be steeper than 1:20 </r>. <E>For <a>distances of less than 3 metres, it may be
steeper, but <r>not more than 1:12</r>. </E> </R>

<R>The <a>access route shall have <r>clear width of a minimum of 1,8 m</r> and
<r>obstacles shall be placed so that they do not reduce that width </r>. <r>Maximum cross
fall shall be 2 %.</r> </R>

<R>The <a>access route shall have <r>a horizontal landing at the start and end of the
in-cline</r>, plus <r>a horizontal landing for every 0,6 m of incline</r>. <r>The landing
shall be a minimum of 1,6 m deep.</r> </R>

<R><r>Minimum clear height shall be 2,25 m</r>for the full width of the defined walking
zone of the entire <a>access route including crossing points. </R> </R>

Mittels dieser identifizierten Struktur kann die Vorschrift nun in ein maschinenlesbares Dokument gespeichert, zwischen Computersystemen ausgetauscht und somit als Grundlage für ein *Automated Code Compliance Checking* dienen. Damit die Informationen dieses Dokumentes jedoch von einer Maschine weiterverarbeitet werden können, müssen diese von einer Instanz interpretiert und ausgelegt werden. Daher sind die nach dem RASE-System strukturierten Informationen nur eine Vorstufe für eine Automatisierung.

Nawari (Automating Codes Conformance 2012) verknüpft in seinem Forschungsansatz die *SMARTCodes* mit der *Language Integrate Query (LINQ)*, welche sich durch einen flexiblen Zugriff auf *Markup-Language*-basierte Daten auszeichnet und ein Teil des *Microsoft .NET Framework* ist.

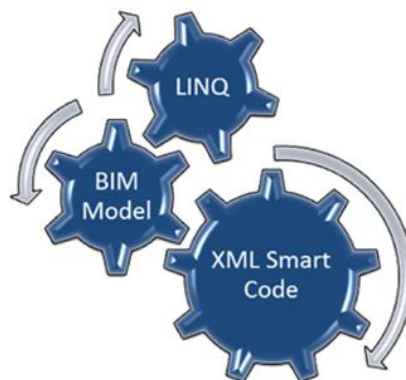


Abbildung 12: Verknüpfung zwischen LINQ, SMARTCodes und Gebäudemodell
(Nawari, Automating Codes Conformance 2012)

Da sich der Ansatz auf die Richtlinien des *National Green Building Code Standard (ICC 700-2008)* konzentriert, wird zur Speicherung des Gebäudemodells der *gbxml*-Standard (2014) verwendet. An dieser Stelle kann zwischen Gebäudemodell und Richtlinien, also *gbxml* und *SMARTCodes*, *LINQ* als Schnittstelle auf sämtliche Daten zugreifen und diese verarbeiten. Zu diesen Verarbeitungsprozessen gehören unter anderem auch die Überprüfung der Vorschriften der *SMARTCodes* und die entsprechenden Informationen des Gebäudemodells. Des Weiteren

kann der Anwender in *LINQ* sein Expertenwissen in den Überprüfungsprozess einbringen. Dieser Zusammenhang ist schematisch in Abbildung 12 dargestellt.

3.1.2.3 Building Environment Rule and Analysis Language

Um den hohen Anforderungen im Umgang mit Datenmodellen im Bauwesen gerecht zu werden, wurde von Lee (2010) die *Building Environment Rule and Analysis (BERA) Language* entwickelt. Wie in Abbildung 13 dargestellt, ist *BERA* in den Bereich der domänenspezifischen Programmiersprachen einzuordnen.

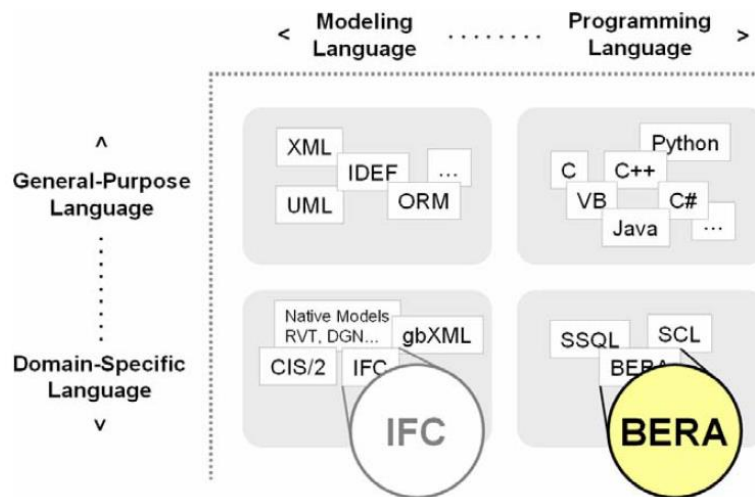


Abbildung 13: Klassifizierung von BERA als Programmiersprache (Lee 2010)

BERA baut auf dem offenen Datenstandard *IFC* auf und ist darauf ausgelegt, dem Nutzer nicht nur den Zugriff auf den Informationsgehalt eines Datenmodells, sondern auch die intuitive Formulierung von Regeln und somit auch Konformitätsüberprüfungen zu ermöglichen.

Lee (2010) nennt in seinen Ausführungen mehrere Ziele, die mit der Entwicklung verfolgt werden:

– *Ease to use*

Die Zielkunden von *BERA* sind insbesondere Anwender ohne fundierte Programmierkenntnisse. Die Sprache ist daher darauf ausgelegt, zum einen auf die Informationen eines *Building Information Model* so effizient und einfach wie möglich zuzugreifen und andererseits mittels einer leicht verständlichen Syntax auch die Verarbeitung dieser Daten zu erleichtern.

– *Portability*

BERA soll explizit als Sprachenstandard für den Umgang mit Gebäudemodellen verstanden werden. Zwar orientiert sich die Sprache an dem offenen *IFC*-Datenformat, jedoch kann hier die Struktur von Modell zu Modell unterschiedlich sein und bietet sich daher nicht als Basis für einen Standard an. Aus diesem Grund verwendet *BERA* eine vereinheitlichte Abstraktion des *IFC*-Modells, das sogenannte *BERA Object Model (BOM)*. Dieses garantiert, dass jeder Zugriff über *BERA* generisch gestaltet werden kann und für jedes Modell in gleicher Art und Weise

funktioniert. Auf diese Weise kann die Sprache als Standard für viele unterschiedliche Gebäudemodelle oder aber auf *BIM*-Plattformen (siehe Abschnitt 3.1.4) dienen.

– *Extensibility*

Als ein solcher Standard kann die Syntax von *BERA* ähnlich wie bei anderen Programmiersprachen sukzessive um neue Elemente, Objekte und Funktionen erweitert und ausgebaut werden. Dabei bleibt es dem Anwender überlassen, welche Elemente dieser in *BERA* integrieren möchte.

Durch diese Faktoren grenzt sich *BERA* deutlich von anderen existierenden Ansätzen ab (siehe Abbildung 14).

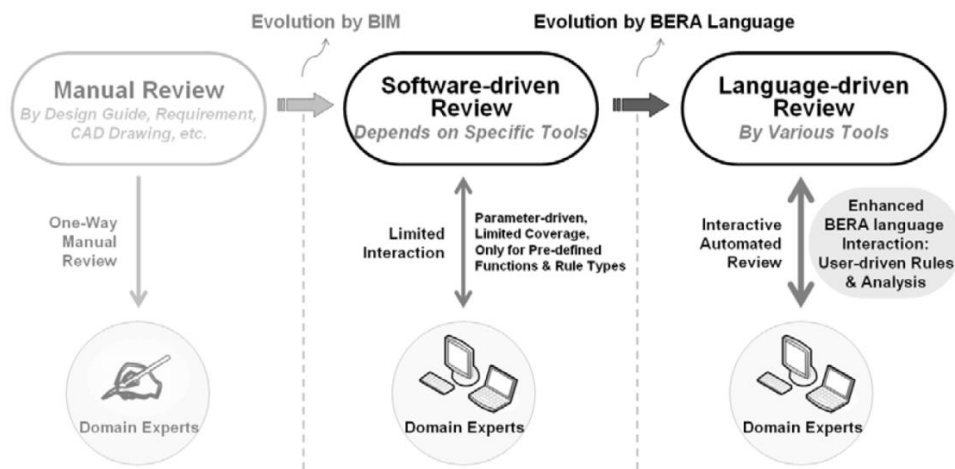


Abbildung 14: Einordnung von BERA in die unterschiedlichen Entwicklungsstufen der automatisierten Konformitätsüberprüfung (Lee 2010)

In dem Forschungsansatz von Lee (2010) konzentriert sich die Entwicklung von *BERA* zunächst ausschließlich auf räumliche und geometrische Informationen des Gebäudemodells. Wie bereits beschrieben müssen hierzu die Informationen des *IFC*-Datenmodells aufgrund der hohen Fehleranfälligkeit und der Komplexität der Datenstruktur in das sogenannte *BOM* abstrahiert werden, um so den Datenzugriff deutlich zu erleichtern. Eine beispielhafte Abstraktion solcher Informationen ist in Abbildung 15 dargestellt.

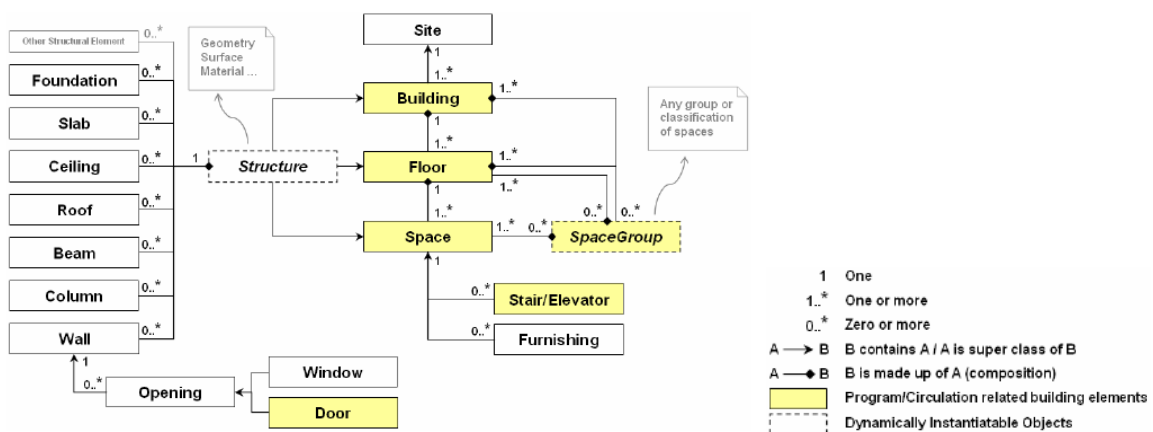


Abbildung 15: Abstraktion räumlicher Informationen des IFC-Datenmodells innerhalb des BOM (Lee 2010)

Welchen enormen Vorteil diese Abstraktion des Modells mit sich bringt, zeigt sich in dem Sprachdesign von *BERA*. Zur Verdeutlichung dieser Vereinfachung in der Syntax führt Lee (2010) ein Beispiel an, welches in Code 7 zu finden ist.

Code 7: Vergleich eines IFC- und BOM-orientierten Befehls in *BERA* (Lee 2010)

```

1 // Wanted: name of a space object called "department"
2 // IFC-centered statement
3 space.ifcRelAssociatesClassification.ifcClassificationReference.ifcLabel.
  getString();
4 // BOM-centered statement
5 space.department;
```

Grundlegend basiert das Sprachdesign von *BERA* auf vier Bestandteilen:

- *reference directive*

Wie in bekannten Programmiersprachen lassen sich auch in der *BERA*-Umgebung externe Datensätze oder Bibliotheken integrieren. Dieser Teil der Syntax ist vergleichbar mit den Direktiven `import` in *Java*, `using` in *C#*, oder aber `include` in *C/C++*.

- *Object Model definition and declaration*

Durch die Vereinheitlichung der Struktur des Gebäudemodells in dem *BOM* lassen sich Objekte und Parameter für das Sprachdesign präzise definieren. Lee (2010) implementiert die Deklarationen von Objektklassen mit den zugehörigen Parametern in der sogenannten *Object Model definition and declaration* und macht diese so in *BERA* für den Anwender als Klassen zugänglich. Auf die Informationen eines solchen Objektes kann in der *BERA*-Syntax mittels der sogenannten Punkt-Operation (*dot-notation*) intuitiv zugegriffen werden. So repräsentiert beispielsweise der Befehl

```
Space.Floor.name = "Level 1"
```

den Zugriff auf den Parameter `name` der Klasse `Floor`, welche wiederum eine Subklasse des Objektes `Space` ist, und belegt diesen über die Operation `=` mit der Zeichenkette `"Level 1"`. Innerhalb dieses Befehls ist also nicht nur eine Zugriffs-, sondern auch eine logische Operation enthalten. Die verfügbaren logischen Operatoren für unterschiedliche Datentypen in *BERA* sind nah verwandt mit denen von weitverbreiteten Programmiersprachen.

- *rule definition*

Speziell für die Definition von Vorschriften innerhalb des Gebäudemodells gibt es in der Syntax die sogenannte *rule definition*, welche eine objektorientierte Formulierung solcher Regelungen erlaubt. Ähnlich wie bereits bei den Klassen des *BOM*, werden auch die Regelungen als Objekt deklariert, wobei diesen zusätzlich zur Initialisierung weitere Objekte übergeben werden können. Wird ein Regel-Objekt in einem *BERA*-Befehl mit dem entsprechenden Parameter initialisiert, wird der Inhalt dieser Vorschrift direkt ausgeführt.

In dem Beispiel in Code 8 wird in Zeile 1 die Regel `myrule` definiert, welche bei der Initialisierung ein Objekt der Klasse `Space` übergeben bekommt. Bei Aufruf dieses Objektes in einem *BERA*-Befehl wird der Inhalt der Regel, also die Anforderungen in Zeile 2 bis 4 direkt an diesem Objekt überprüft.

Code 8: Definition einer Regel in BERA (Lee 2010)

```
1 Rule myrule(Space space) {
2     space2.area > 1000;
3     space2.Floor = "Level_1";
4     space2.security = "public";}
```

– *execution statement*

Der letzte Teil der *BERA*-Syntax beinhaltet die Deklaration von sogenannten *execution statements*, welche als freie objektorientierte Routinen in einem *BERA*-Befehl aufgerufen werden können. Ein Beispiel für eine solches *execution statement* ist die Anweisung `get(object)`, welches in Abhängigkeit des übergebenen `object` den Zugriff auf dieses Objekt durchführt.

Auf Grundlage dieses Sprachdesigns lassen sich in *BERA* sowohl Zugriffe, als auch Vorschriften für das Gebäudemodell formulieren. Für den Nachweis der Tragfähigkeit wurde für die *BERA*-Sprache ein Editor und Compiler in die Umgebung des *SMC* (siehe Abschnitt 3.1.4.3) integriert. Lee (2010) führt mittels dieses sogenannten *BERA Language Tool* (siehe Abbildung 16) einige Beispiele auf, um die Funktionsvielfalt von *BERA* nachzuweisen.

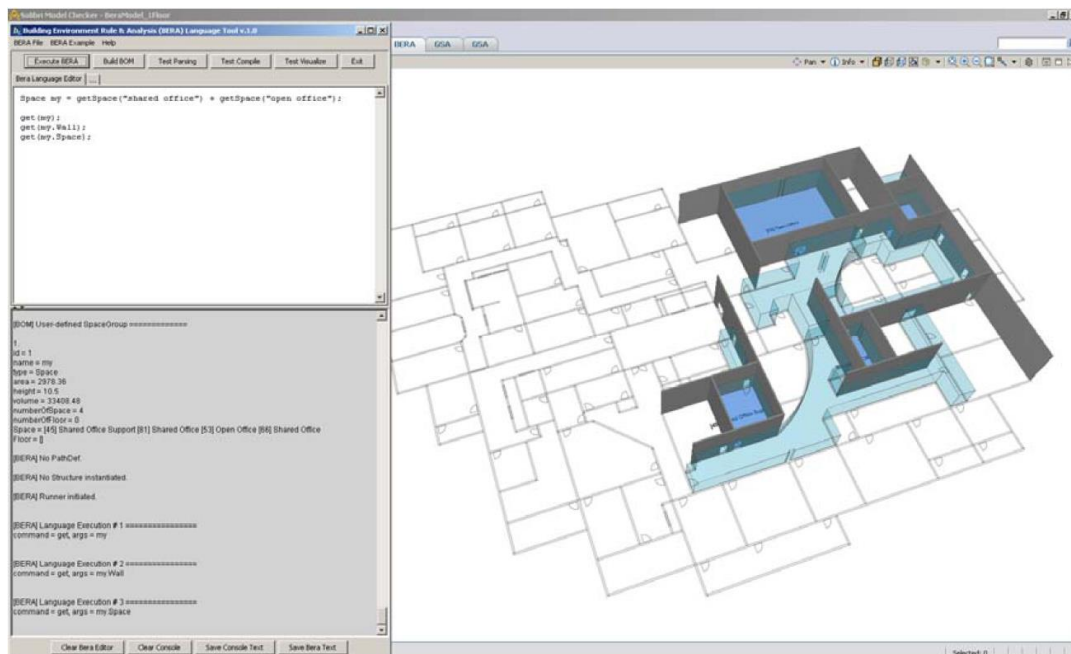


Abbildung 16: Integration des BERA Language Tool in der Umgebung des SMC (Lee 2010)

In Code 9 ist der *BERA*-Befehl für die Abfrage einer definierten Raumklasse aufgeführt. Hierzu wird in Zeile 1 zunächst das Objekt `midOffice` vom Typus `Space` (Raummodell) deklariert und in Zeile 2 bis 6 mit Werten belegt. Über den Befehl `get(midOffice)` in Zeile 8 werden nun in dem korrespondierenden *BOM* alle Objekte gesucht, die den definierten

Randbedingungen des Objektes `midOffice` entsprechen. Das Ergebnis dieser Abfrage kann abschließend graphisch in dem *SMC* dargestellt werden (siehe Abbildung 17).

Code 9: BERA-Befehl für die Abfrage einer definierten Raumklasse eines Gebäudemodells (Lee 2010)

```

1  Space midOffice {
2      Space.area > 600;
3      Space.area < 900;
4      Space.height > 9;
5      Space.name = "office";
6      Space.name != "shared";}
7  get(midOffice);

```



Abbildung 17: Ergebnis der Raumanalyse mit BERA (Lee 2010)

Wie bereits beschrieben lassen sich mit *BERA* auch Vorschriften und Richtlinien für ein Gebäudemodell formulieren. In Code 10 ist ein Beispiel für eine Laufweganalyse in der *BERA*-Syntax dargestellt. Ähnlich wie bereits bei der Raumanalyse zuvor werden in diesem Fall die zwei Objekte `start` und `end` der Klasse `Space` definiert. Die Klasse `Path` ist ein Beispiel für ein Objekt der *BERA*-Sprache, welches nicht direkt zu dem *BOM* gehört, aber zu der Weiterverbreitung von räumlichen Informationen eingeführt wurde. Innerhalb dieser Klasse sind Algorithmen zu Analyse von Raummodellen als Funktionen hinterlegt, welche mittels der Daten des *BOM* ausgeführt werden können. Das Ergebnis der Laufweganalyse kann abschließend ebenfalls in dem *SMC* graphisch dargestellt werden (siehe Abbildung 18).

Code 10: BERA-Befehl für Abfrage von Laufwegen innerhalb eines Gebäudemodells (Lee 2010)

```

1  Space start = getSpace("laboratory") + getSpace("lobby");
2  Space end {
3      Space.area > 600;
4      Space.Floor.number > 0;
5      Space.Floor.height > 10;
6      Space.name = "office";}
7  Path myPath = getPath(start, end);
8  get(myPath);

```



Abbildung 18: Ergebnis der Laufwegeanalyse mit BERA (Lee 2010)

Zwar konzentriert sich die Entwicklung von *BERA* sehr stark auf die Formulierung von räumlich bzw. geometrisch orientierten Richtlinien, jedoch werden innerhalb des vorgestellten Ansatzes viele Instrumente präsentiert, die den Grundstein für eine beachtliche Ausweitung dieser Sprache legen. Somit kann *BERA* als der fortschrittlichste Ansatz im Bereich der sprachbasierten Ansätze zum *Automated Code Compliance Checking* angesehen werden.

3.1.3 Ontologische Ansätze

Der Begriff Ontologie stammt aus dem Altgriechischen und setzt sich aus den beiden Wörtern ὄν (εἶναι) „seiend“ und λόγος „Lehre“ zusammen. In seiner ursprünglichen Bedeutung beschreibt der Begriff eine Disziplin der theoretischen Philosophie, die sich mit der Lehre alles Seienden beschäftigt. Im Vergleich zu der bereits vorgestellten Deontologie (siehe Abschnitt 3.1.1.3) behandelt die Ontologie Fragestellungen über die Existenz und Beschaffenheit von Objekten und Systemen. Zur Vollständigkeit ist in Abbildung 19 eine Auflistung verschiedener philosophischer Theorien und deren Zusammenhang mit logischen Familien sowie den Datenmodellen der Computerwissenschaften aufgeführt.

Comparative criterion	Ontology	Axiology	Deontology
Philosophical theory that the model originates from	Theory of existence	Theory of value	Theory of rights and obligations
Application area	Domain system representation	Value assessment	Normative reasoning
Types of concepts represented in the model	Concepts of the domain	Concepts of values and valuation	Concepts of normative reasoning
Type of logic used	First order logic	Axiological logic	Deontic logic

Abbildung 19: Auflistung der verschiedenen logischer Familien (Salama und El-Gohary 2013)

Der Begriff Ontologie beschreibt in den Computerwissenschaften eine Darstellungsweise eines spezifischen Wissensgebiets, einer sogenannten Domäne, welche die Semantik (Bedeutung) des Gebietes erfassen kann. Dieses bedeutet, dass die Ontologie in der Lage ist zu „verstehen“, welche Informationen in ihr selbst gespeichert sind und wie diese entsprechend verarbeitet werden müssen. Dieses Selbstverständnis der Ontologie kann mit Hilfe einer Taxonomie, einer Logik und Relationen, also der Darstellung von den internen Zusammenhängen, gebildet werden (Furrer 2014).

Gruber (1993) definiert die Ontologie als eine „formale explizite Beschreibung einer gemeinsam verwalteten Konzeption“. Eine Ontologie unterliegt einer formellen Struktur, beinhaltet somit eine Logik und ist dadurch für Mensch und Maschine interpretierbar.

Gleichzeitig beschreibt sie die Wissensgebiete explizit, d.h. Informationen werden auf Anfrage eindeutig zurückgegeben und es kommt zu keinen Inkonsistenzen im Datenmodell (Genesereth und Nillson 1987). Nicht zuletzt soll eine Ontologie die Informationen einer Domäne so beschreiben, dass sie in sämtliche Richtungen austauschbar sind und mit anderen ontologischen Wissensbereichen verknüpft werden können. Dieser Grundgedanke führte in der Neuzeit zu der Idee des sogenannten *Semantic Web*, einer einzigen vernetzten ontologischen Datenbasis, welche alle Domänen verwaltet und die entsprechenden Daten bei Bedarf abrufen kann. Daher lassen sich viele weitere Definition einer Ontologie finden, die den Fokus vermehrt von der Taxonomie, der Klassenhierarchie, auf die netzwerktechnische Verknüpfung der einzelnen Ontologien untereinander legen, wie beispielsweise bei Guarino und Giarette (1995).

Allen diesen Definitionen ist gemein, dass sie der Ontologie alle notwendigen Komponenten zuordnen, welche bereits in den logik- als auch sprachbasierten Ansätzen (siehe Abschnitt 3.1.1 und 3.1.2) zum *Code Compliance Checking* verwendet wurden. Die folgenden Unterkapitel beschreiben ausgewählte Ansätze, welche mit Hilfe einer Ontologie entweder eine theoretische Basis für das *Automated Code Compliance Checking* bilden oder darüber hinaus den Prozess der Konformitätsüberprüfung umsetzen.

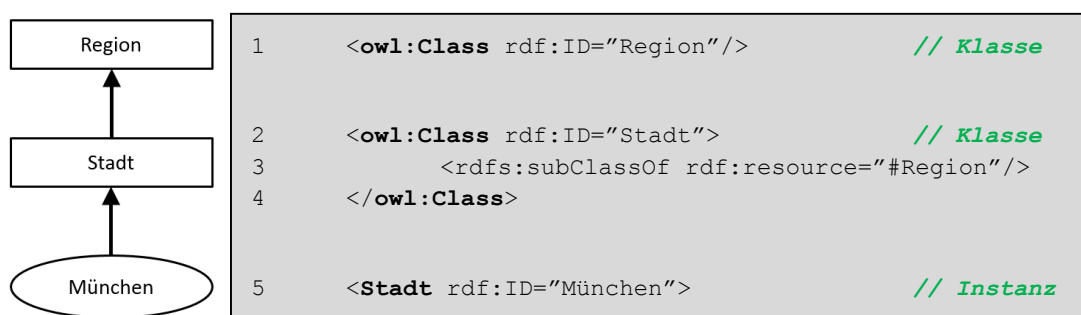
3.1.3.1 OWL

Zur Beschreibung einer Ontologie wurde vom *World Wide Web Consortium* (2014) die Auszeichnungssprache *Ontology Web Language (OWL)* entworfen. Diese Sprache beschreibt den Informationsgehalt einer Ontologie mit Hilfe der drei grundlegenden Sprachebenen *OWL Lite*, *OWL Description Logic* und *OWL Full* (Joo-Sung, et al. 2008).

Für die Erstellung einer Ontologie muss zunächst eine Basisstruktur für die Klassenhierarchie gebildet werden. Dieses kann mittels der ersten Sprachebene *OWL Lite* aufgestellt werden, indem die grundlegenden Klassen gemäß der Auszeichnungs-Syntax von *OWL* und mit Hilfe des graphisch-logischen Formulierungssystems *Resource Description Framework* (Pan 2009, Pauwels, et al. 2010) deklariert werden. In

Code 11 ist ein Beispiel für eine Deklaration von Klassen und Instanzen dargestellt.

Code 11: Übersetzung einer Klassenhierarchie in OWL (Kost 2013)



Auf Basis dieser Klassenhierarchie kann nun die logische Ebene der Ontologie hinzugefügt werden. Die *OWL Description Logic* beinhaltet eine definierte Menge an logischen Operatoren, die für einzelne, aber auch mehrere Klassen eingesetzt werden können. Eine Auflistung solcher Operatoren ist in Abbildung 20 aufgeführt.

Logical semantics	Description	Example statement
$\neg C$	General concept negation	"not a building component"
$\leq R.C, \geq R.C$	Qualified number restrictions	"bigger in size for a certain relationship of a class C"
$R = \langle y, x \rangle \mid \langle x, y \rangle \in R$	Inverse roles	"Contains is the inverse of IsInsideOf."
$(\langle x, y \rangle \in R) \wedge (\langle y, z \rangle \in R) \Rightarrow \langle x, z \rangle \in R$	Transitive Roles	"Product is the descendant of object and element is the descendant of product. So element is the descendant of object."

Abbildung 20: Auflistung logischer Operatoren der OWL Description Logic (Joo-Sung, et al. 2008)

Die logische Ebene wird mit Hilfe von relationalen Objekten gebildet, die die einzelnen ontologischen Klassen zueinander ins Verhältnis setzen. Ein Beispiel für eine solche Relation und die zugehörige Übersetzung innerhalb von *OWL* ist in Code 12 dargestellt.

Code 12: Beispiel für eine logische Relation im ontologischen Modell und die zugehörige OWL-Übersetzung (Joo-Sung, et al. 2008)

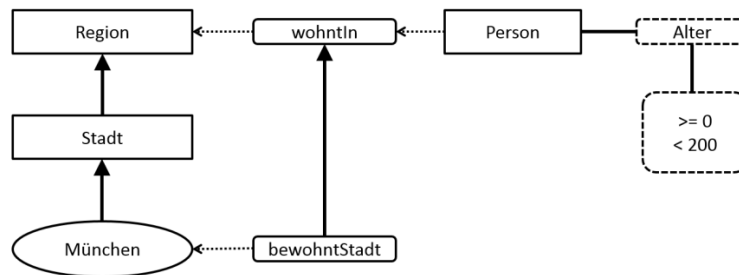


```

1  <owl:ObjectProperty rdf:ID="isLargerThan">                               // Relation
2    <rdfs:type rdf:resource="&Person ; TransitiveProperty"/>
3  </owl:ObjectProperty>
4
4  <owl:Class rdf:ID="Room1">
5    <isLargerThan ref: resource="#Room2"/>
6  </owl: Class >
7
7  <owl:Class rdf:ID="Room2">
8    <isLargerThan ref: resource="#Room3"/>
9  </owl: Class >
  
```

Diese beiden Sprachebenen setzen ein ontologisches System zusammen und beschreiben dieses umfassend. Ein einfaches Beispiel für ein solches Gesamtsystem ist in Code 13 dargestellt.

Code 13: Beispiel für ein vollständiges ontologisches Modell und die zugehörige OWL-Übersetzung (Kost 2013)



```

1  <owl:ObjectProperty rdf:ID="wohntIn">
2    <rdfs:domain rdf:resource="#Person"/>
3    <rdfs:range rdf:resource="#Gebiet"/>
4  </owl:ObjectProperty>
5
6  <owl:ObjectProperty rdf:ID="bewohntStadt">
7    <rdfs:subPropertyOf rdf:resource="#wohntIn"/>
8    <rdfs:range rdf:resource="#Stadt"/>
9  </owl:ObjectProperty>
10
11 <owl:DatatypeProperty rdf:ID="Alter">
12   <rdfs:domain rdf:resource="#Person" />
13   <rdfs:range rdf:resource="&dt;Alter"/>
14 </owl:DatatypeProperty>

```

Joo-Sung et. al. (2008) haben das Prinzip der Ontologie mit Hilfe von *OWL* und dem *IFC*-Datenmodell an dem Pilotprojekt „Gunpo Worker’s Welfare Centre“ auf das Bauwesen angewendet. Auf Grundlage einer Klassenhierarchie des Gebäudemodells (siehe Abbildung 21) lassen sich über Relationen und logische Operatoren Randbedingungen definieren.

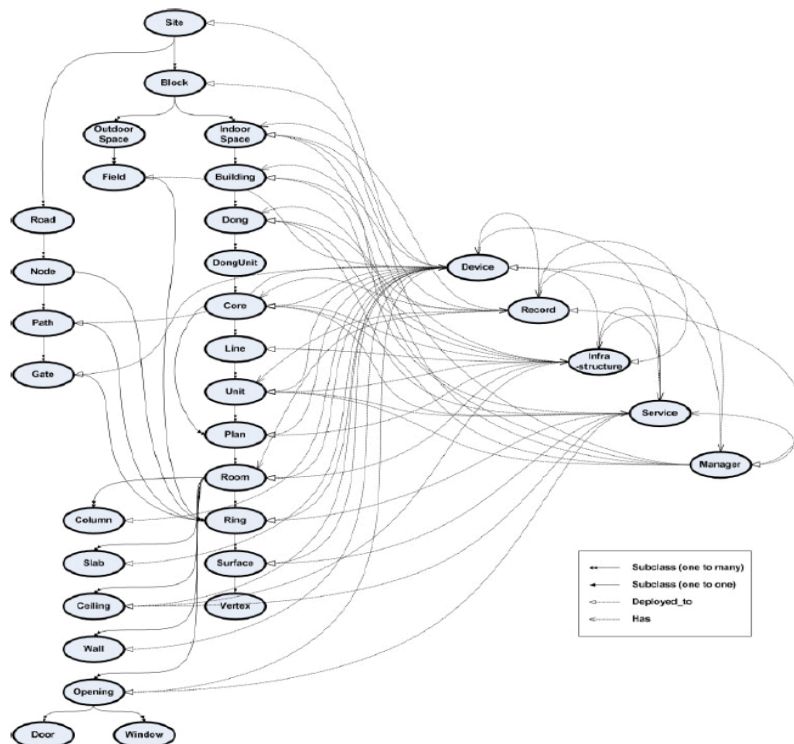


Abbildung 21: Vereinfachtes ontologisches Klassenmodell für ein Gebäudemodell im IFC-Format (Joo-Sung, et al. 2008)

Durch die Definition der relationalen Objekte innerhalb der Klassenhierarchie, wie diese in Code 12 vorgestellt wurden, konnte eine Reihe von Randbedingungen für die Gestaltung eines Grundrisses (siehe Abbildung 22) implementiert werden.

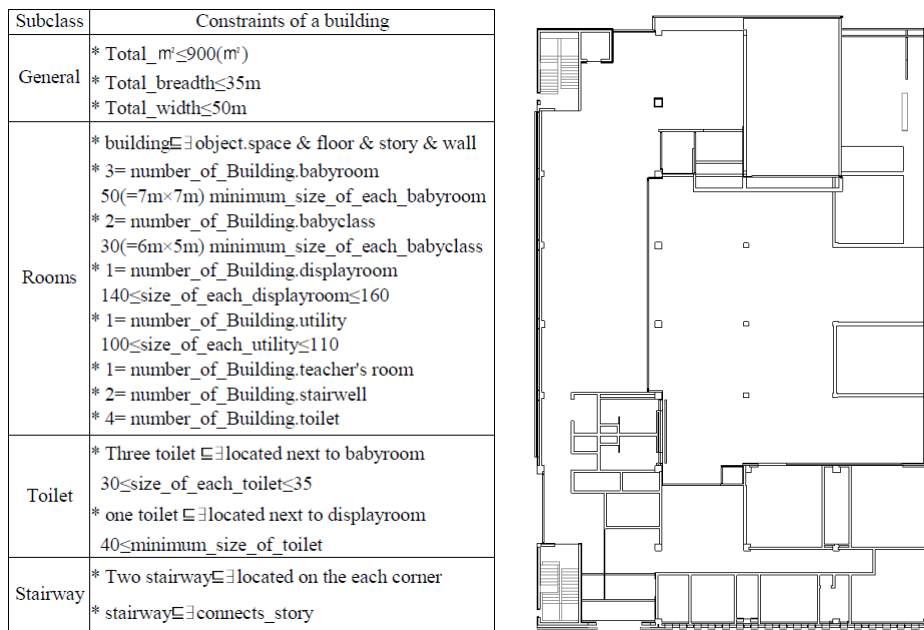


Abbildung 22: Randbedingungen für die Gestaltungsplanung eines Grundrisses (Joo-Sung, et al. 2008)

Da es sich hierbei jedoch lediglich um eine feste Implementierung in einem Beispielprojekt handelt, kann dieser Ansatz lediglich als Tragfähigkeitsnachweis angesehen werden.

Eine deutlich praxistauglichere Umsetzung wurde von Zhang und Raja (2010) entwickelt, die für das ontologische Datenmodell eines Gebäudes eine Softwarearchitektur für Datenzugriffe entworfen haben (siehe Abbildung 23). Diese Software ist in der Lage auf eine konkrete Fragestellung oder Anfrage über eine ontologische Datenbank einen konkreten Rückgabewert zu geben.

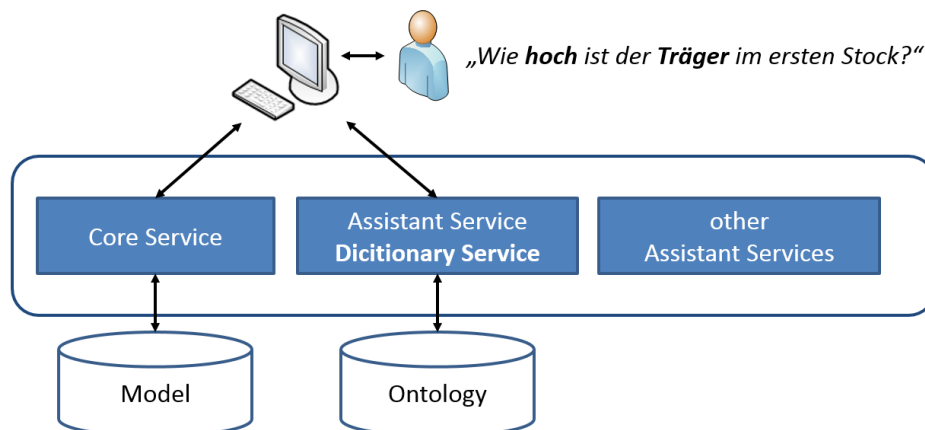


Abbildung 23: Softwarearchitektur für einen Datenzugriff auf das ontologische Gebäudemodell (Zhang und Raja 2010)

3.1.3.2 IfcOWL

Eine Ursache für die Entwicklung der Ontologie in den Computerwissenschaften ist auf die Interoperabilitäts-Probleme der existierenden Datenformate zurückzuführen. Viele der bereits vorgestellten Ansätze verwenden beispielsweise den offenen *IFC*-Standard, um in möglichst viele Richtungen kompatibel zu sein. Gleichzeitig bringt dieses Datenformat grundlegende Probleme mit sich, welche Beetz et al. (2009) auf drei Ursachen zurückführen:

- „*Lack of formal rigidity*“:

Die Beschreibung des Produktmodells mittels *IFC* basiert nicht auf einer strengen bzw. starren mathematischen Grundlage. Mittels Algorithmen, Axiomen und Theoremen könnte ein Modell intelligent gestaltet werden, so dass es auf äußere Veränderungen reagieren kann und dabei die innere Konsistenz bewahrt.

- „*Limited reuse and interoperability*“:

Das *STEP*-Format und die *EXPRESS*-Sprache, auf welchem der *IFC*-Standard beruht, sind nicht in allen Bereichen der Computerwissenschaften verbreitet und so gestaltet sich ein Austausch von Informationen mit anderen Wissensgebieten ab einem gewissen Punkt als sehr schwierig. Ziel sollte es sein, dass sämtliche ontologische Systeme, die eine Domäne beschreiben, auch miteinander verknüpft werden können.

- „*Lack of built-in distribution*“:

Nach der Vorstellung einer vollständig vernetzten, digitalen Welt müssen alle Elemente einer Einheit modular austausch- und verteilbar sein. Von der Organisation *buildingSMART* (2014), die grundlegend zu der Entwicklung des *IFC*-Standards beigetragen hat, sind zwar für das *IFC*-Datenformat grundlegende Regeln für den Umgang mit diesem Datenmodell definiert worden, jedoch lassen sich die Datenobjekte des *IFC* sehr frei verwenden. Die fehlende, aber zwingend erforderliche Struktur in der Syntax des *IFC*-Standards verhindert, dass Daten einheitlich gespeichert werden und daher bildet dieses Format keine passende Grundlage für einen modularen Austausch von Informationen.

Eine Lösung des Problems bieten Beetz et al. (2009) an dieser Stelle mit der *IfcOWL*, einer Kombination des *IFC*-Datenformates und der bereits vorgestellten *OWL*. Dabei handelt es sich nicht um eine Mischform, sondern vielmehr um eine Abbildung des *IFC*-Formates auf die *OWL*, um die Modelldaten auf ein ontologisches Niveau zu heben.

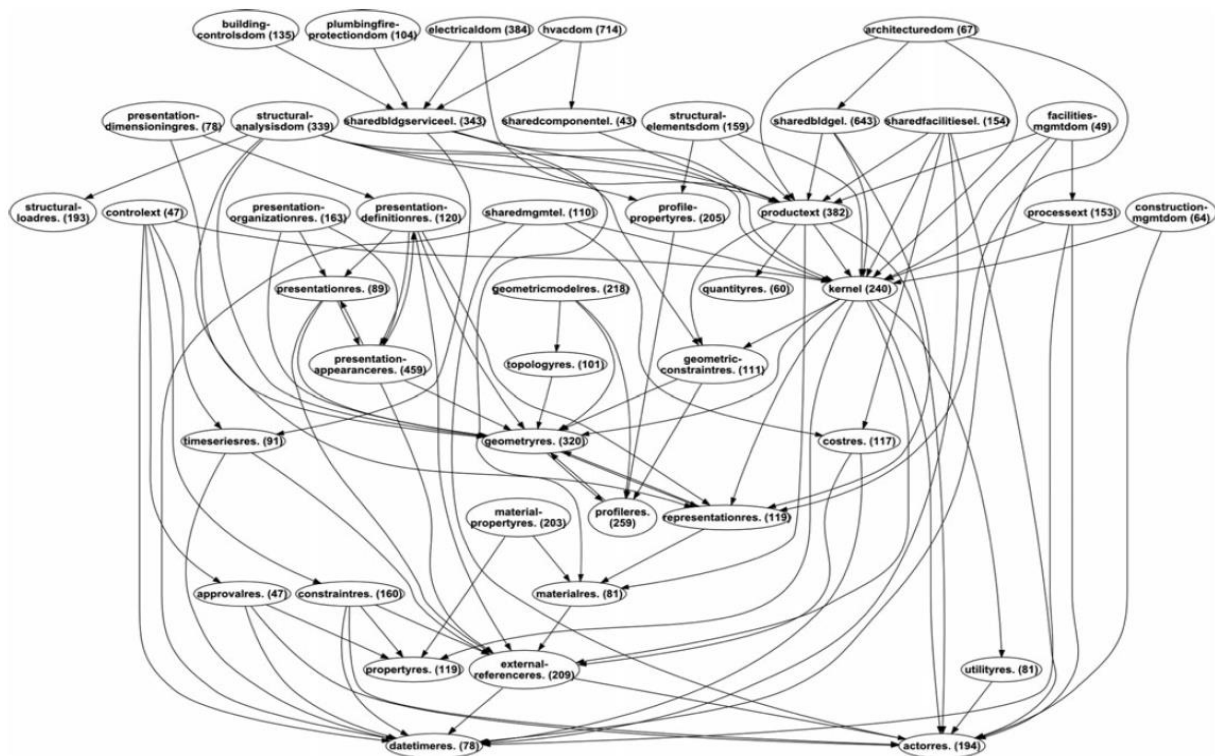


Abbildung 24: Klassenhierarchie des IFC-Datenmodells für IfcOWL (Beetz, van Leeuwen und de Vries 2009)

Wie bereits bei den vorigen Ansätzen, liegt auch hier eine Klassenhierarchie – in diesem Falle die des *IFC*-Datenmodells - zu Grunde (siehe Abbildung 24). Mittels dieser Taxonomie kann der Informationsgehalt eines *IFC*-Modells über fest implementierte Abbildungsroutinen auf ein ontologisches Schema abgebildet werden. Wie diese einzelnen Klassen ontologisch definiert und voneinander abhängig sind, ist in der sogenannten *Terminology Box* auf Basis des *RDF* hinterlegt und zusammengefasst. Eine beispielhafte Übersetzung solcher Klassen von der *EXPRESS*-Sprache in *IfcOWL* ist in Code 14 & Code 15 dargestellt.

Code 14: Vereinfachte Klassendeklaration in EXPRESS (Beetz, van Leeuwen und de Vries 2009)

```

1  ENTITY IfcElement
2      ABSTRACT SUPERTYPE OF (ONE OF (IfcBuildingElement,
3                                     IfcFurnishingElement, [...]))
4  END_ENTITY;

5  ENTITY IfcBuildingElement
6      ABSTRACT SUPERTYPE OF (ONE OF (IfcDoor, IfcWall, IfcSlab, [...]))
7      SUBTYPE OF (IfcElement)
8  END_ENTITY;

9  ENTITY IfcDoor
10     SUBTYPE OF (IfcBuildingElement)
11 END_ENTITY;

```

Code 15: Übersetzung der vereinfachten Klassendeklaration in IfcOWL (Beetz, van Leeuwen und de Vries 2009)

```

1   :IfcElement
2       a      owl:Class ;
3       rdfs:subClassOf owl:Thing

4   :IfcBuildingElement
5       a      owl:Class ;
6       rdfs:subClassOf      : IfcElement ;
7       owl:disjointWith   : IfcFurnishingElement

8   :IfcDoor
9       a      owl:Class ;
10      rdfs:subClassOf      : IfcBuildingElement
11      owl:disjointWith   : IfcWall, : IfcWindow

```

Nachdem nun das *IFC*-Klassenmodell auf dieses ontologische Modell abgebildet wurde, können auch die einzelnen Instanzen übertragen werden. Hierfür ist die sogenannte *Assertion Box* zuständig, welche die einzelnen Instanzen des Gebäudemodells den ontologischen Klassen der *Terminology Box* zuordnet. Die Struktur des gesamten Abbildungsprozesses ist in Abbildung 25 veranschaulicht.

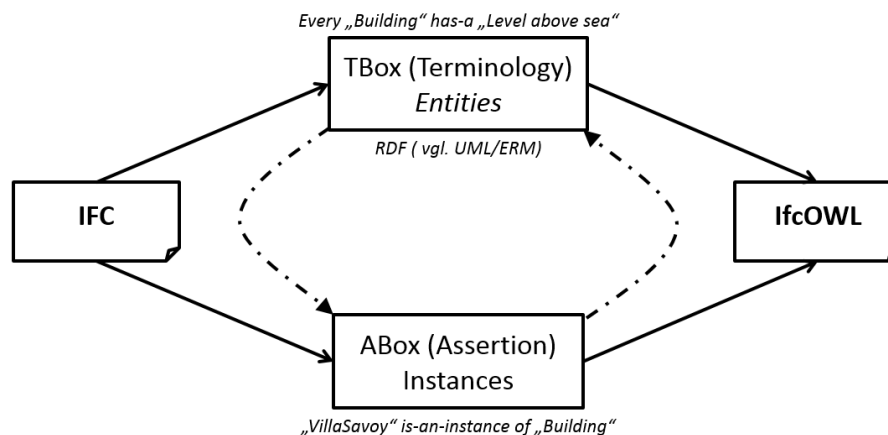


Abbildung 25: Schematische Abbildung des IFC auf IfcOWL (Beetz, van Leeuwen und de Vries 2009)

Beetz et al. (2009) stellen außerdem ein *Java*-basiertes Tool¹ als Prototyp für diese Übersetzung zur Verfügung und machen so einen ersten Schritt in Richtung der Automatisierung dieses Prozesses.

Neben der Entwicklung von *IfcOWL* existieren noch einige weitere Ansätze, die auf ähnliche Art und Weise versuchen, das Prinzip der Ontologie auf ein Gebäudemodell und somit auf das Bauwesen zu übertragen. Eine ausführliche Beschreibung und Analyse weiterer *OWL*-basierter Ansätze geben Pauwels et al. (2010).

¹ <https://github.com/mmlab/IFC-to-RDF-converter>

3.1.3.3 C3R

Yurchyshyna et al. (2008) konzentrieren sich bei ihren Forschungen nicht auf den eigentlichen Übersetzungsprozess des *Automated Code Compliance Checking*, sondern vielmehr auf den Aufbau einer ontologischen Wissensbasis, mit der schließlich eine Konformitätsüberprüfung möglich sein soll. Umgesetzt wurde dieser Ansatz von Yurchyshyna und Zarli (2009) innerhalb des Softwaretools *C3R (Conformance Checking in Construction with the help of Reasoning)*, welches einen halb-automatisierten Überprüfungsprozess erlauben soll.

Zur Erstellung dieser Wissensbasis müssen zunächst die gewünschten Vorschriften im *Knowledge Acquisition Module* von *C3R* mittels der *Protocol And RDF Query Language (SPARQL)* formuliert werden. Yurchyshyna et al. (2008) beschränken sich in ihrem Ansatz auf räumlich orientierte Richtlinien zur Barrierefreiheit, die der elektronischen Datenbank *CD REEF* entstammen. Die *SPARQL*-Übersetzung einer solchen Beispiel-Richtlinie ist in Code 16 aufgeführt.

Code 16: Formulierung einer Vorschrift in SPARQL
(Yurchyshyna, et al. 2008)

Constraint: "The minimum width of a door is 90 cm"

```
1  select ?door display xml
2  where
2  { ?door rdf:type ifc:IfcDoor
3  OPTIONAL { ?door ifc:overallWidth ?width
4  FILTER ( xsd:integer(?width) >= 90)}
5  FILTER (! bound( ?width) )}
```

Erst in einem zweiten Schritt des Konzeptes kommt die Theorie der Ontologie zum Tragen. Aus den in *SPARQL* verfassten Vorschriften, wird mit Hilfe der *OWL* (siehe Abschnitt 3.1.3.1 oben) ein ontologisches Schema formuliert. Parallel hierzu werden alle für dieses Schema relevanten Daten eines *IFC*-Gebäudemodells in dem *ifcXML*-Format (buildingSMART 2014), eine auf das *IFC*-Datenformat zu geschneiderte Abwandlung der *Markup Language*, aufbereitet und anschließend auf das ontologische Schema abgelegt. Bisher laufen alle Prozessschritte der Übersetzung nicht selbstständig, sondern müssen von einem Experten begleitet werden, der sein Fachwissen über diverse Hilfsmittel -insbesondere mit Graph-basierten Schemata in *RDF*- in den Überprüfungsprozess einbringt. Daher kann an dieser Stelle lediglich von einem halb-automatisierten Prozess gesprochen werden (Yurchyshyna, et al. 2008).

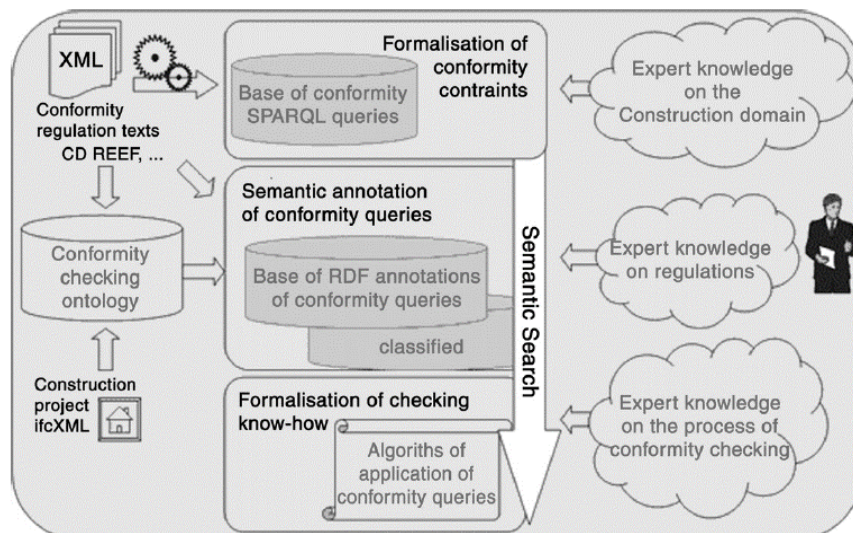


Abbildung 26: Schematischer Ablauf in C3R (Yurchyshyna und Zarli 2009)

Das mit den Gebäudedaten besetzte ontologische Schema bildet schließlich die Wissensbasis, auf welche die einzelnen oder kombinierten *SPARQL*-Anfragen angewendet werden können. Technisch gesehen basiert die Durchführung einer solchen Suchanfrage auf einer Graphen-Projektion, einem sogenannten *Homomorphismus* von zwei Graphen, welche die Validierung einer Wissensbasis erlaubt. Hierzu wird die Aussage der Suchanfrage, die prinzipiell auch den Inhalt einer Vorschrift enthalten kann, so in einen *RDF*-Graphen übersetzt, dass dieser das Gegenteil der Anfrage aussagt, also z.B. (vgl. Code 16)

„*The width of the door is less than 90cm*”

Anschließend kann der Graph hinsichtlich seines Wahrheitsgehalts an der ontologischen Wissensbasis überprüft werden. Ist das Ergebnis dieser Überprüfung eine Schnittmenge von Anfrage und Ontologie, so entspricht mindestens ein Element des Gebäudemodells der gegenteiligen Vorschrift und somit ist die Konformität von Regelwerk und Modell nicht gegeben (Yurchyshyna, et al. 2008). Ein schematischer Aufbau dieser einzelnen Module und der Suchanfrage selbst ist in Abbildung 26 dargestellt.

3.1.4 Code Compliance Checking auf BIM-Plattformen

Mit zunehmender Bekanntheit und Akzeptanz des *Building Information Modeling* in den vergangenen Jahrzehnten haben auch die Entwicklungen rund um diese digitale Methode weltweit zugenommen. Zu diesen zählen insbesondere die meist staatlichen Bestrebungen, zentrale Plattformen für digitale Gebäudemodelle zu entwickeln und mit deren Hilfe Bauprojekte und deren Prozesse zu optimieren. Diese zentralen Datensysteme stellen nicht nur grundlegende Funktionen im Umgang mit den Gebäudemodellen zur Verfügung, sondern dienen überdies als Ausgangspunkt für die Entwicklung von zusätzlichen Funktionalitäten rund um den Lebenszyklus eines Bauwerkes.

Die folgenden Unterkapitel stellen die Entstehung und Funktionsweise verschiedener *BIM*-Plattformen sowie zugehörige Entwicklungen im Bereich der automatisierten Konformitätsüberprüfungen vor.

3.1.4.1 CORENET & FORNAX

Im Jahr 1995 wurde von der Regierungsbehörde *BCA* in Singapur (2006) die Entwicklung der sogenannten *CORENET*-Plattform mit der Intention gestartet, sämtliche Informationen eines Bauprojektes zentral zu speichern und mit Hilfe von digitalen Werkzeugen die Bauprozesse zu optimieren. Eines dieser Instrumente ist die Applikation *CORENET BP-Expert*, welche zum Ziel hatte, eine erste Konformitätsüberprüfung von digitalen, zweidimensionalen Zeichnungen in den Bereichen Barrierefreiheit und Feuerschutz zu ermöglichen.

Im Jahr 1998 wurde das Datenmodell der *CORENET*-Plattform auf das *IFC*-Datenmodell (ISO 16739:2013) umgestellt und somit um eine dreidimensionale Konformitätsüberprüfung erweitert. Die aktuelle Version des Tools wurde unter dem Namen *CORENET e-Plan Check* im Jahre 2002 veröffentlicht und bietet eine Konformitätsüberprüfung von Gebäudemodellen und einem großen Teil der singapurischen Regelwerken der Bereiche Gebäudesteuerung, Barrierefreiheit, Brandschutz und Umweltgesundheit (Dimyadi und Amor 2013, buldingSMART 2014).

Die Überprüfung von Modell und Regelwerken innerhalb von *CORENET* basiert auf fest einprogrammierten Routinen, welche für den Nutzer nicht einsehbar sind. Es handelt sich dabei um eine sogenannte *Black-Box*-Methode (von Bertalanffy 1972), welche lediglich die Eingabe- und Ausgabeinformationen zur Verfügung stellt. Der eigentliche Verarbeitungsprozess der Informationen ist für den Nutzer nicht sichtbar und bietet daher nur geringen Spielraum für zusätzliche Prozesse, wie beispielsweise die Plausibilitätsprüfung der erhaltenen Ergebnisse.

Der Überprüfungsprozess von *CORENET* teilt sich grundlegend in drei Phasen und orientiert sich dabei an dem Informationsgehalt des zu überprüfenden Datenmodells. In einer ersten Phase wird überprüft, welche Daten direkt aus dem Informationsgehalt des Modells verwendet werden können und welche Datensätze auf Umwegen bezogen werden müssen. Anschließend wird auf erweiterte, untergeordnete Informationsebenen des Modells zugegriffen, um diese nicht direkt vorhandenen Informationen zu erhalten. Sollten auch auf dieser Ebene diese Informationen nicht verfügbar sein, wird schließlich in einem letzten Schritt die fehlende Information aus bereits vorhandenen Daten über Routinen abgeleitet (Eastman, Lee, et al. 2009b).

Um eine solche Aufbereitung des Datenmodells in den beiden letzteren Prozessschritten zu ermöglichen, wurde parallel zu *CORENET* mit *FORNAX* von der Firma *novaCITYNETS* (2014) eine objektorientierte *C++*-Bibliothek geschaffen, welche die Regelwerke als einzelne, semantische Objekte abbildet und diese in das Datenschema des *IFC*-Modells eingliedert.

Durch den Zugriff der *FORNAX*-Objekte auf die Informationen des Gebäudemodells ist es nicht nur möglich, die Routinen zur Datenbeschaffung, sondern auch Überprüfungsrountinen zu formulieren und diese direkt in dem Datenmodell zu speichern. Ein Beispiel für Funktionen eines solchen *FORNAX*-Objekts ist für ein Treppenhaus in Abbildung 27 dargestellt (Eastman, Lee, et al. 2009b).

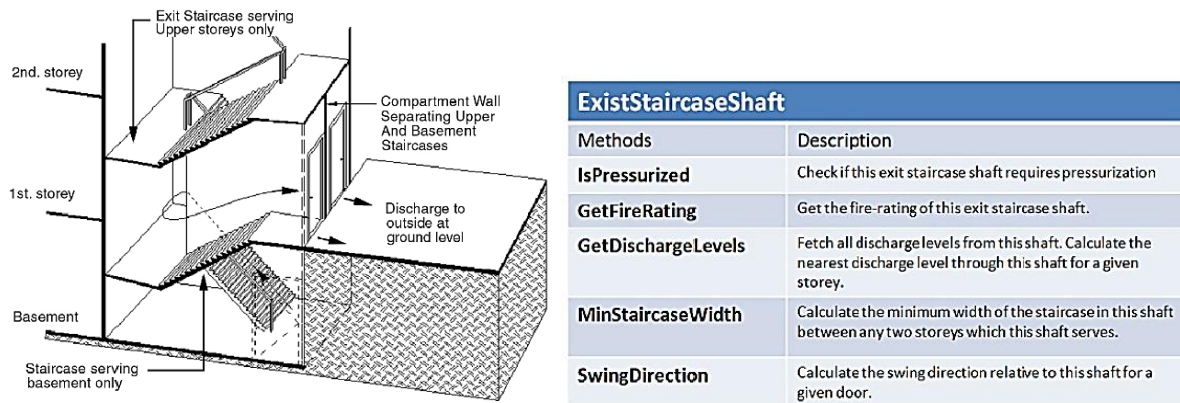


Abbildung 27: Beispiel für die Funktionsweise eines FORNAX-Objektes (Eastman, Lee, et al. 2009b)

Da viele Informationen im Bauwesen stark geometrisch orientiert sind, bieten die *FORNAX*-Objekte mit Hilfe der Entwickler-Umgebung *OpenCASCADE* und dem Geometrie-Kern *ACIS* umfassende Möglichkeiten, geometrische Operationen zu formulieren (Open CASCADE Technology 2014, Spatial Corp. 2014). So kann mit Hilfe der Objekte der Überprüfungsprozess innerhalb des *CORENET*-Systems vervollständigt werden, indem alle für den Prozess notwendigen Daten gesammelt oder aber über Routinen aufbereitet werden. In Abbildung 28 ist der strukturelle Aufbau dieses Überprüfungsprozesses abgebildet.

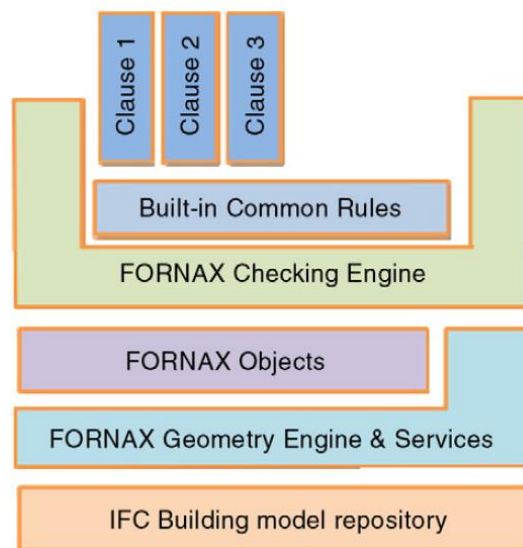


Abbildung 28: Struktur eines CORENET-Projektes (Eastman, Lee, et al. 2009b)

Die Entwicklung von *CORENET* stellt einen der frühesten, aber auch heute noch einen der fortschrittlichsten Ansätze zur automatisierten Konformitätsüberprüfung von Regelwerken und Gebäudemodell im Bauwesen dar. Im Jahr 2008 wurden die singapurischen Regelwerke *IBP* (*Building Plan*) zu 92 % und *IBS* (*Building Service*) zu 77 % von *CORENET* abgedeckt und von annähernd 2500 Unternehmen der *AEC*-Branche genutzt (Eastman, Lee, et al. 2009b). Parallel gibt es einige weitere Ansätze, die die Idee der *FORNAX*-Objekte aufgreifen und weitere Regelwerke erfassen, wie beispielsweise die Entwicklung von Xu, Solihin und Huang (2004).

3.1.4.2 EXPRESS Data Manager & DesignCheck

Parallel zu den Entwicklungen in Singapur wurde im Jahr 1998 von dem norwegischen Technologieunternehmen *Jotne EPM Technology* (2014) die Kollaborations-Plattform *Express Data Manager (EDM)* geschaffen. Diese basiert auf einer objektorientierten Datenbank, welche mit der ISO-zertifizierten *EXPRESS*-Sprache (ISO 10303-11) arbeitet und darauf ausgerichtet ist, Produktdatenmodelle mehrerer Ingenieurdisziplinen zu verwalten. In Abbildung 29 ist ein schematischer Aufbau des *EDM*-Systems dargestellt.

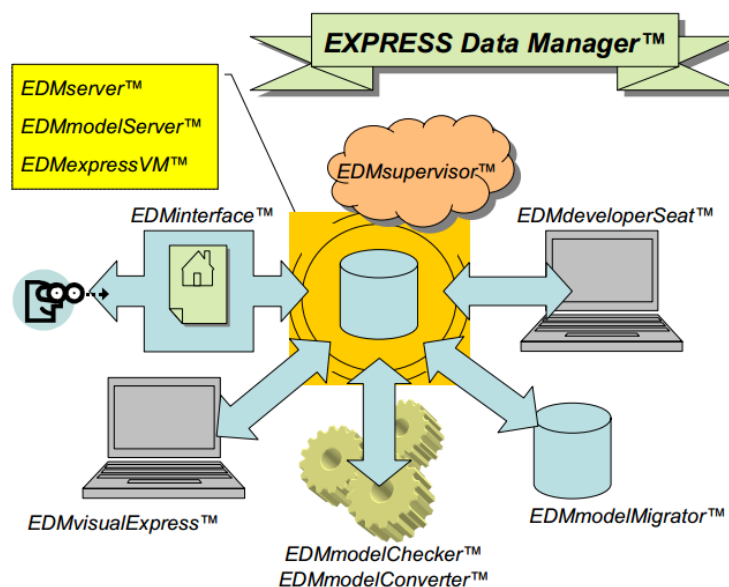


Abbildung 29: Aufbau des EDM (Jotne EPM Technology 2004)

Die Datenmodellierungs-Sprache *EXPRESS* dient innerhalb des *EDM* dazu, ein hohes Maß an Flexibilität im Umgang mit den komplexen Datenmodellen zu erreichen. *EXPRESS* ist ein Teil des *Standard for the exchange of product model data* (STEP, ISO 10303), einem weit verbreiteten Format für Produktdatenmodelle, und ermöglicht dem *EDM* daher die Kompatibilität mit einer großen Anzahl von verschiedenen Datenmodell-Formaten. Über ein integriertes Konvertierungsmodul, dem sogenannten *EDMmodelConverter*, können die gespeicherten Datenmodelle mit einer geringen Fehleranfälligkeit auf andere Formate abgebildet werden. Das *EDM*-Datenbanksystem ist somit insbesondere auch mit dem *IFC*-Datenformat kompatibel, welches ebenfalls auf der *EXPRESS*-Sprache basiert (L. Ding, et al. 2004).

Von *Jotne EPM Technology* selbst steht neben dem Datenkern der *EDM*-Plattform bereits eine Vielzahl weiterer Module zur Verarbeitung der Daten zur Verfügung. Im Bereich der Konformitätsüberprüfung ermöglicht das herstellereigene Modul *EDMmodelChecker*, Vorschriften mit Hilfe der *EXPRESS*-Sprache zu formulieren und anschließen auf die Informationen eines gespeicherten Gebäudemodells anzuwenden (Jotne EPM Technology 2004, L. Ding, et al. 2004).

Die Offenheit und Flexibilität der *EDM*-Plattform stellt ein wichtiges Entscheidungskriterium für die Bestrebungen externer Entwickler dar. Daher dient diese Plattform als Basis für eine Reihe von Ansätzen im Bereich der automatisierten Konformitätsüberprüfung.

Eine externe Erweiterung für die *EDM*-Plattform zum Thema Konformitätsüberprüfung wurde in Australien von dem *Cooperative Research Center for Construction Innovation* (2006) mit dem Softwaretool *DesignCheck* geschaffen. Zu Beginn der Entwicklungen wurden zwar zunächst mehrere Plattformen als Basis für die Entwicklungen in Betracht gezogen, doch aufgrund der offenen Struktur und der bereits integrierten Datenbankverwaltung, begrenzte man schließlich *DesignCheck* auf die *EDM*-Umgebung (Eastman, Lee, et al. 2009b).

Ähnlich wie bereits bei dem *EDMmodelChecker* können mit Hilfe des *DesignCheck*-Systems können einzelne Regeln oder Regelwerke mittels der *EXPRESS*-Sprache als Objekte direkt in das *EDM*-Datenmodell geschrieben werden. Für die Übersetzung eines solchen Regelwerks wird von den Entwicklern empfohlen, Inhalt und Struktur des gewünschten Objektes zunächst in Pseudocode zu definieren, um dieses erst anschließend per Hand oder automatisiert in ein Regelobjekt zu transformieren. Zur Veranschaulichung der Struktur eines solchen Objekts ist ein Pseudocode-Beispiel in Abbildung 30 für die Zugänglichkeit von Raumobjekten dargestellt.

```

CLAUSE 7: DOORWAYS, DOORS AND CIRCULATION SPACE AT DOORWAYS

Clause 7.1 Provision of Entrances
Description:
The requirements for entrances to buildings are as follows:
(a) Accessible entrances shall be incorporated in an accessible path of travel.
Performance Requirements:
There is an uninterrupted path of travel from an accessible entrance to an accessible space required.
Objects:
{Space, Door}
Object Properties:
{Door_external, Door_accessible, Door_type, Door_width, Space_accessible, Space_identification, Space_area}
Object Relationship:
{Contain (Space, Door)}; {Adjacent (Space, Space)}
Domain-specific knowledge for Interpretation:
(to be implemented with functions, procedures, etc.)

AssessibleExteriorDoor (Doors)
{IF Door_external and Door_accessible are found, THEN return AssessibleExteriorDoors}

AssessibleEntranceSpace (AssessibleExteriorDoors)
{IF AssessibleExteriorDoors are contained by Spaces, THEN return AssessibleEntranceSpaces}

AssessibleSpaceRequired (Spaces)
{IF Space_accessible is found, THEN return AssessibleSpacesRequired}

A_Path_from_AssessibleEntranceSpace_to_AssessibleSpaceRequired (Spaces, Doors)
{IF Spaces and Doors are located in the path from AssessibleEntranceSpace to AssessibleSpaceRequired, THEN return a set of the Spaces and a set of the Doors}

Criteria_for_anUninterruptedPath
{IF Spaces and Doors located in the path satisfy the requirement of Door_width, Door_type, Space_area, etc. THEN return TRUE}

```

Abbildung 30: Pseudocode-Darstellung einer Norm für *DesignCheck* (L. Ding, et al. 2006)

Um ein Regelwerk-Objekt in *DesignCheck* zu definieren, müssen zunächst die für die Überprüfung relevanten (Bau-)Objekte und deren zugehörige Parameter deklariert werden. Da diese Objekte sehr häufig untereinander in einer bestimmten, beispielsweise einer räumlichen, Beziehung stehen, muss auch dieses Verhältnis definiert werden. Abschließend können nun auf Basis dieser Deklarationen die eigentlichen Anweisungen übersetzt werden, welche sich auch

aus mehreren Teilen zusammensetzen können. Jede dieser einzelnen Teilfunktionen beschreibt mit Hilfe einer Implikation, den deklarierten Objekten und Parametern die Anweisungen, welche gemeinsam den Überprüfungsprozess zusammensetzen.

In dem vorliegenden Beispiel aus Abbildung 30 werden zunächst die Datenobjekte *Raum* und *Tür* deklariert und deren zugehörige Parameter definiert. In den Relationen wird erklärt, dass die beiden Objekte in einer bestimmten Abhängigkeit stehen, da ein *Raum* eine *Tür* enthält und die *Räume* untereinander benachbart sind. Mit Hilfe dieser Angaben können anschließend mehrere Entscheidungsfälle formuliert werden, welche wiederum den gesamten Prozess der Konformitätsüberprüfung bilden.

Zwar ist die Beschreibung eines Informationssystems mittels dieser Syntax sehr umfassend, jedoch kann es Inhalte innerhalb einer Norm geben, welche sich nur unzureichend mittels des Pseudocodes beschreiben lassen. Für solche Fälle wurde eine Darstellung von Algorithmen mit Hilfe von Graphen entwickelt, welche im Gegensatz zu dem Pseudocode in der Lage ist, diese Inhalte abzubilden. In Abbildung 31 ist ein solcher Graph dargestellt, welcher beispielhaft den Algorithmus zur Ermittlung einer behindertengerechten Route durch ein Gebäude gemäß der australischen Norm *AS 1428.1* darstellt. Innerhalb des Graphen ist beschrieben, wie sich die behindertengerechte Zugänglichkeit eines Raumes durch seine Nachbarschaft zu einem anderen angrenzenden Raumes auswirkt.

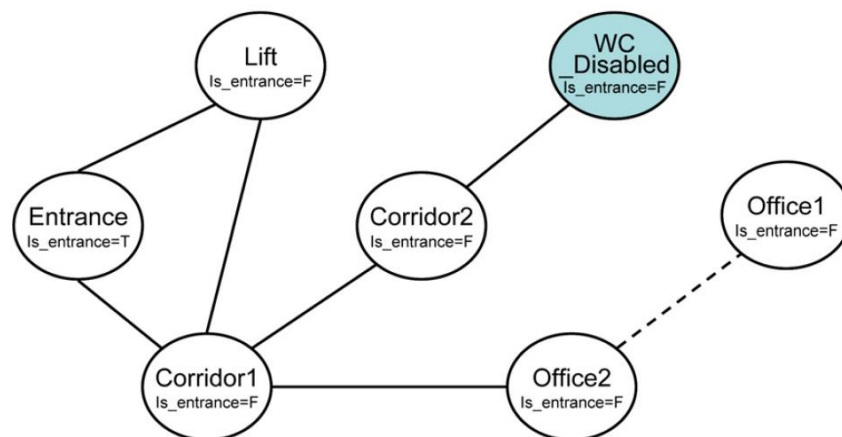


Abbildung 31: Graphische Unterstützung der Formulierung von Algorithmen in DesignCheck (Eastman, Lee, et al. 2009b)

Um eine Angliederung eines Regelwerk-Objektes an das *EDM*-Datenmodell möglichst einfach zu gestalten, wird dieses ebenfalls in die *EXPRESS*-Sprache übersetzt und kann somit sehr einfach in der Datenbank gespeichert werden. Der objektorientierte Ansatz bringt außerdem den Vorteil mit sich, dass unabhängig zu dem Datenmodell eine Bibliothek von einzelnen Prozessen bzw. Routinen entsteht, auf welche bei der Erstellung weiterer individueller Prozesse zurückgegriffen werden kann (Eastman, Lee, et al. 2009b).

Der schematische Ablauf eines vollständigen Überprüfungsprozesses mit *DesignCheck* in dem *EDM*-System ist in Abbildung 32 dargestellt. Aus externen CAD-Programmen eingelesene Informationen können zunächst über vom Anwender vorgegebene Schemata auf die Überprüfung hin vorbereitet werden. Diese vorbereiteten Datenteilmengen können

anschließend mit Hilfe der in der Datenbank gespeicherten Regelwerk-Objekten überprüft werden (L. Ding, et al. 2004).

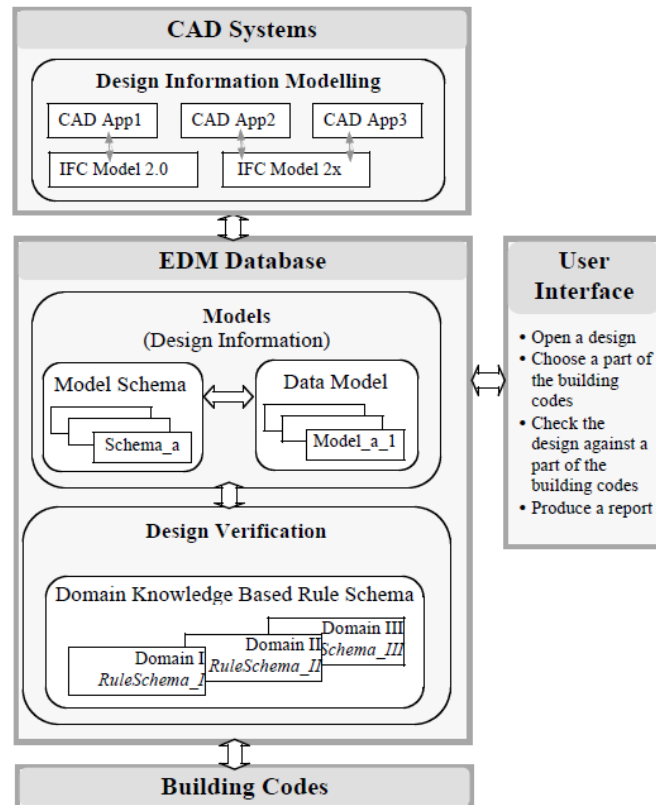


Abbildung 32: Aufbau von DesignCheck (L. Ding, et al. 2004)

Die Ergebnisse der Überprüfung werden dem Anwender in Form von *XML*- bzw. *HTML*-Dokumenten aufbereitet und beinhalten detaillierte Informationen zu dem Prozess selbst und den einzelnen untersuchten Objekten enthalten. Durch ein interaktives Userinterface (siehe Abbildung 33) soll abschließend insbesondere eine Plausibilitätsüberprüfung der Ergebnisse ermöglicht werden. Eine graphische Analyse des Prozesses ist derzeit noch nicht möglich, diese soll jedoch in einer künftigen Version des Systems ebenfalls verfügbar sein (L. Ding, et al. 2006).

The screenshot displays the **Design Check Report** interface. It includes a table of results and a detailed view of a specific check.

Results	Clause	Object Type	Object Name	Space Name	Detail
COMPLIANCE	7_1a_2	LIFT	LIFT_07	LIFT_07	There is a path of travel betw...
COMPLIANCE	7_1a_2	LIFT	LIFT_05	LIFT_05	There is a path of travel betw...
COMPLIANCE	7_1a_2	LIFT	LIFT_06	LIFT_06	There is a path of travel betw...
COMPLIANCE	7_1a_2	LIFT	LIFT_08	LIFT_08	There is a path of travel betw...
INFORMATION_NEEDED	7_1a_3	Any Space	All	CLASS_5_OFFICE_BUIL...	The space elevation is not de...
NON_COMPLIANCE	7_1a_3	CORRIDOR	CORRIDOR_002	CORRIDOR_002	Accessible entrance shall be...
NON_COMPLIANCE	7_1a_3	CORRIDOR	CORRIDOR_001	CORRIDOR_001	Accessible entrance shall be...
NON_COMPLIANCE	7_1a_3	COMMERCIAL_SPA...	COMMERCIAL_SPACE_01	COMMERCIAL_SPACE_01	Accessible entrance shall be...
COMPLIANCE	7_1a_3	COMMERCIAL_SPA...	COMMERCIAL_SPACE_04	COMMERCIAL_SPACE_04	There is a path of travel betw...
COMPLIANCE	7_1a_3	COMMERCIAL_SPA...	COMMERCIAL_SPACE_03	COMMERCIAL_SPACE_03	There is a path of travel betw...
COMPLIANCE	7_1a_3	COMMERCIAL_SPA...	COMMERCIAL_SPACE_02	COMMERCIAL_SPACE_02	There is a path of travel betw...
COMPLIANCE	7_1a_3	COMMERCIAL_SPA...	COMMERCIAL_SPACE_06	COMMERCIAL_SPACE_06	There is a path of travel betw...

User Input

Object Name: LIFT_07 Clause: 7_1a_2 Result: COMPLIANCE

Model Information Missing | Specification Needed | Checker Designer Comments

Required Information:
There is a path of travel between entrance and lift stair space

Information Inputs:

Check Results

Clause: 7_1c
Object Type: Revolving
Object Name: RevolvingDoor_01
Space Name:
Result: NON_COMPLIANCE
Details: Where revolving doors or turnstiles are installed, an alternative hinged or sliding door shall be provided
Checker Comment: no comment
Designer Comment: Non-compliance

Check Results

Clause: 7_1d
Object Type: Threshold Ramp
Object Name: All

Abbildung 33: Abschlussbericht des Überprüfungsprozesses in DesignCheck (L. Ding, et al. 2006)

Die Entwicklungen rund um das *DesignCheck*-Tool befinden sich momentan zwar noch in einem frühen Stadium, allerdings zeigen sie bereits auf, dass dieses Planungsinstrument mit Hilfe der *EDM*-Umgebung sehr flexibel und offen aufgestellt ist. Allerdings benötigt der Anwender trotz der definierten Syntax bei der Übersetzung der Regelwerk-Objekte ein hohes Maß an Fachkompetenz im Umgang mit der *EXRESS*-Sprache (Eastman, Lee, et al. 2009b).

3.1.4.3 Solibri Model Checker & GSA Design Guides

Der *Solibri Model Checker* (2014) ist eine Java-basierte Plattform der finnischen Technologiefirma *Solibri*, welche ursprünglich als Qualitäts- und Validierungswerkzeug für *IFC*-Gebäudemodelle im Jahre 2000 veröffentlicht wurde. Innerhalb des *SMC* werden die Daten eines *IFC*-Modells automatisch auf ein proprietäres, internes Datenmodell abgebildet und können anschließend weiterverarbeitet werden. Die Abbildung zwischen den beiden Datenmodellen basiert auf fest implementierten Routinen innerhalb des *SMC* und bringt den Vorteil mit sich, dass das System in nur geringem Maße fehleranfällig für Inkonsistenzen in dem Modell ist. Daher hat sich der *SMC* in den vergangenen Jahren vermehrt zu einem eigenständigen und verbreiteten Werkzeug für Konformitätsüberprüfungen entwickelt (Dimyadi und Amor 2013, Solibri 2014). In der aktuellen Version 9.1 bietet der *SMC* eine Vielzahl grundlegender Funktionalitäten an, wie beispielsweise automatisierte Überprüfungen zur Gestaltungsplanung (Solibri 2014).

In dem sogenannten Regelsätze-Manager, welcher in Abbildung 34 dargestellt ist, können die verschiedenen Möglichkeiten zur Konformitätsüberprüfung von Modell und Regelwerk verwaltet werden.

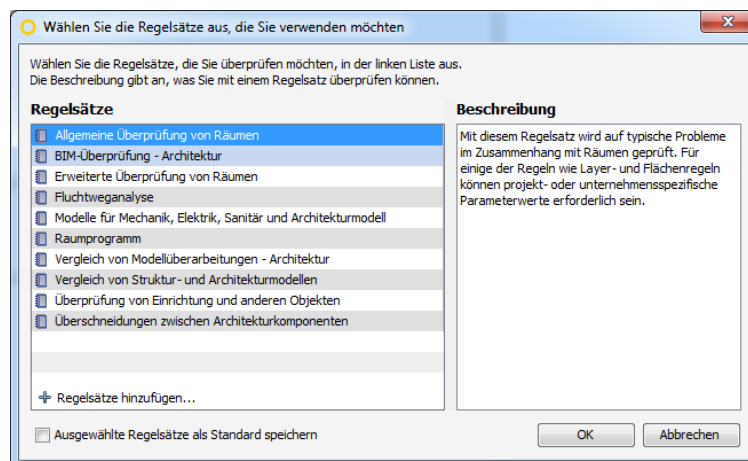


Abbildung 34: Regelsätze-Manager in SMC

Die Regelsätze innerhalb des *SMC* sind als feste Funktionen implementiert, welche über eine proprietäre Programmierschnittstelle auf die Informationen des Datenmodells zugreifen und diese weiterverarbeiten. Diese Schnittstelle steht jedoch nicht öffentlich zur Verfügung und macht den *SMC* somit zu einer *Black-Box*-Applikation, welche keine Informationen des verarbeitenden Prozesses sichtbar macht. Eine externe Entwicklung von neuen bzw. individuellen Regelsätzen ist lediglich in Zusammenarbeit mit der Firma *Solibri* möglich (Eastman, Lee, et al. 2009b).

Die Konformitätsüberprüfung einer Vorschrift im *SMC* besteht üblicherweise aus einer Abfolge mehrerer einzelner Kontrollprozesse, welche nacheinander auf das Gebäudemodell angewendet werden. In Abbildung 35 ist ein solcher Überprüfungsprozess beispielhaft für eine Fluchtwegeanalyse mit den einzelnen Prozessschritten und den graphischen Ergebnissen dargestellt.

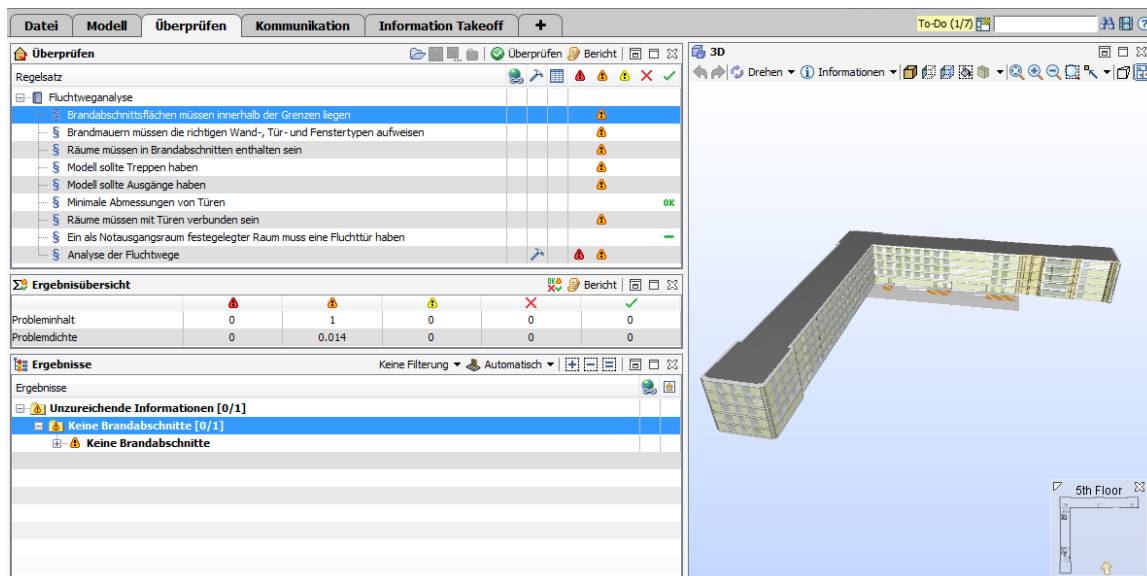


Abbildung 35: Fluchtwegeüberprüfung in SMC

Trotz der Geschlossenheit des *SMC*-Systems gibt es auch externe Entwicklungen, welche im Bereich des Code Compliance Checking in Zusammenarbeit mit der Firma *Solibri* betrieben worden sind.

So wurde in den USA im Jahre 2003 von der Regierungsbehörde *U. S. General Services Administration* (2014), die für das US-amerikanische Wirtschaftsministerium den größten Anteil der öffentlichen Gewerbeflächen verwaltet, im Rahmen eines staatlichen Programmes zur Förderung des *Building Information Modeling* unter anderem auch ein Tool zur Konformitätsüberprüfung der Gestaltungsplanung von öffentlichen Gebäuden entwickelt (Eastman, Lee, et al. 2009b). Ziel dieser Entwicklung, welche in enger Zusammenarbeit mit dem *Georgia Institute of Technology* betrieben worden sind, war es, bereits in sehr frühen Phasen der Gestaltungsplanung die architektonischen Entwürfe auf deren Eignung hinsichtlich der geltenden Gestaltungsrichtlinien automatisiert zu überprüfen und somit den Planer zu unterstützen. Diese Gestaltungsvorschriften, die sogenannten *U.S. Courts Design Guide* werden von dem *Administrative Office of the U.S.* herausgegeben und beinhalten eine Vielzahl von Raum-, Umwelt-, Sicherheit- und Gebäudetechnikanforderungen insbesondere für Justizgebäude, die aufgrund des US-amerikanischen Rechtssystems besonderen Auflagen unterliegen (Administrative Office of the U.S. Courts 2007). Überdies fordern die Richtlinien der *GSA* von den Planern die Vorlage von mindestens drei unterschiedlichen Raumkonzepten, unter denen die Behörde dann den Favoriten auswählen kann. Dieses führt dazu, dass sowohl auf Seiten der Planer, als auch auf der Seite der Behörde ein hoher Aufwand entsteht, die Planungen hinsichtlich ihrer Eignung zu den Vorschriften zu überprüfen.

Speziell für die Überprüfung dieser Richtlinien wurde das *Design Assessment Tool* als Plug-In für den *SMC* entwickelt, welches sich auf die Einhaltung der räumlichen Vorschriften konzentriert und gleichzeitig auch eine Energie- und Kostenanalyse erlaubt. Ein schematischer Ablauf einer Überprüfung mit Hilfe dieses Planungswerkzeuges ist in Abbildung 36 dargestellt.

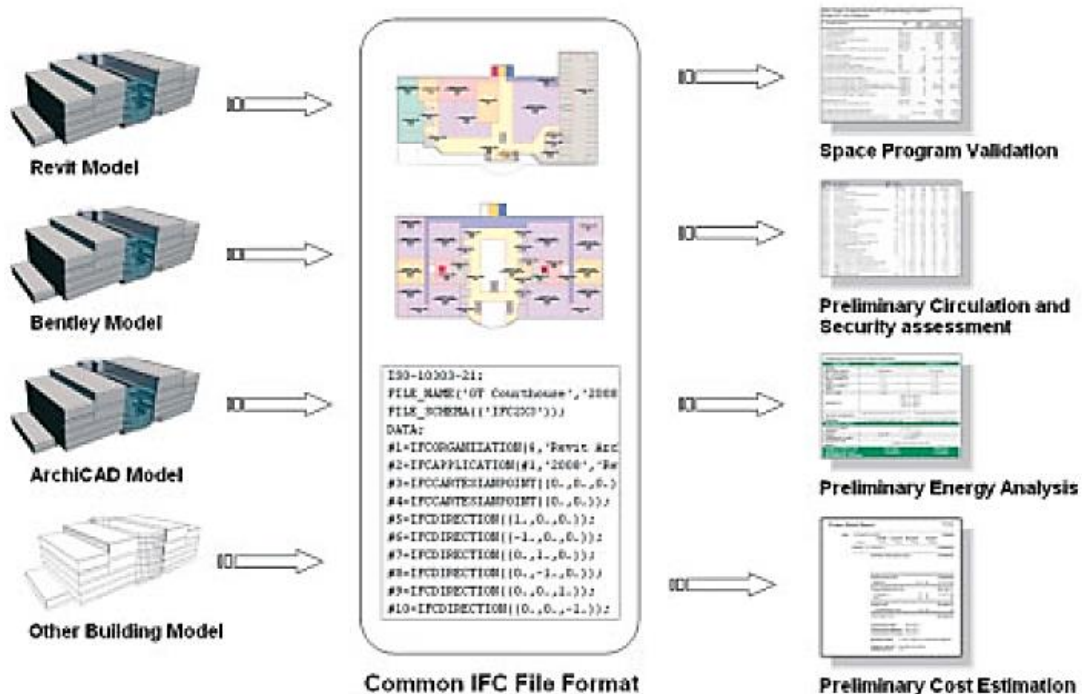


Abbildung 36: Schematischer Ablauf des Preliminary-Design-Tool der GSA (Eastman 2009)

Für die Durchführung des Konformitäts-Checks werden Minimalanforderungen an den Informationsgehalt des Datenmodells, welches aufgrund der Anlehnung an den *SMC* im *IFC*-Format gespeichert ist, gestellt. Eine entsprechende Auflistung dieser Anforderungen für die jeweiligen Überprüfungsarten wurde von der *GSA* und dem *Georgia Institute of Technology* in dem begleitenden Dokument *GSA Preliminary Concept Design BIM Guide* (*GIT College of Architecture 2008*) herausgegeben. Damit die Ergebnisse der Überprüfung nicht an Aussagekraft verlieren, wird in dem ersten Schritt die Einhaltung dieser Minimalanforderungen an das Datenmodell überprüft.

Da eine Vielzahl der Richtlinien des *GSA* raumbasiert formuliert ist, ist der Ausgangspunkt des *DAT* das Raummodell, welches über die Bezeichnungen der einzelnen Räume deren Funktion und Rolle im Gebäude zuordnet. Da diese Rolle jedoch je nach Überprüfungsart wechseln kann, muss das Raummodell jeweils auf die Anforderungen der Kontrolle zugeschnitten werden. Hierzu dient eine festgelegte Zuordnung, ein sogenanntes *Mapping*, welches die Räume je nach Anforderungen umsortiert. In Abbildung 37 ist diese Zuordnung schematisch dargestellt.

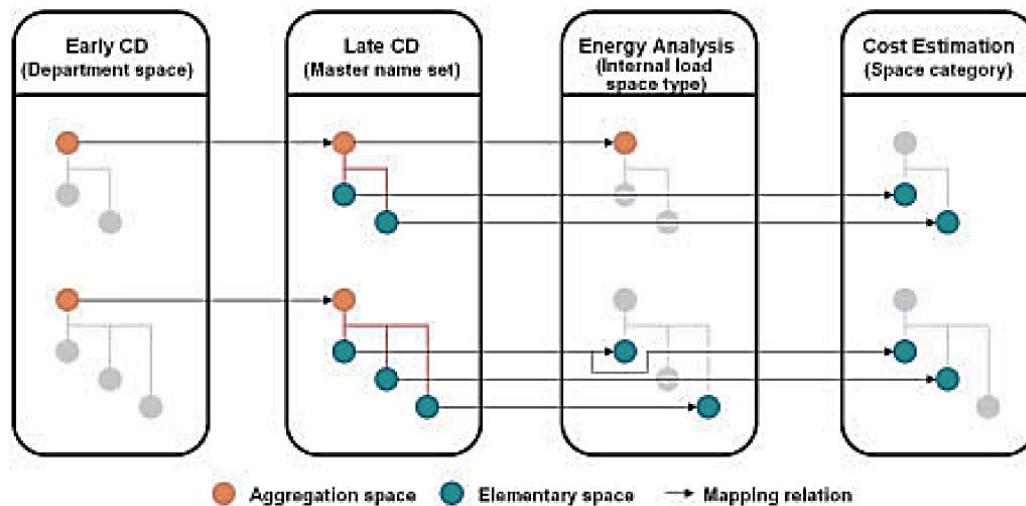


Abbildung 37: Zuordnung der Bezeichnungen von Räumen zur Vorbereitung der jeweiligen Überprüfung (Eastman 2009)

Mittels dieser Zuordnung der Räume kann nun das Gebäudemodell auf die einzelnen Richtlinien hin überprüft werden.

Eine der zentralen Regelungen innerhalb der *U.S. Courts Design Guide* stellt die Richtlinie für das Sicherheitskonzept im Raumprogramm der Justizgebäude dar. Daher soll sich innerhalb der vorliegenden Arbeit auf diese beschränkt werden.

Aufgrund der Struktur des US-amerikanischen Rechtssystems unterliegt das Gebäude in der Aufteilung und Zugänglichkeit der Räume besonderen Anforderungen, die in jedem einzelnen Gestaltungskonzept eingehalten werden muss. Ein Beispiel für eine solche Anforderung ist in Abbildung 38 dargestellt. Diese beinhaltet die Regelung, dass das Büro des Staatsanwalts mit dem Tagungsraum der Jury über eine Sicherheitszone zu erreichen sein muss.

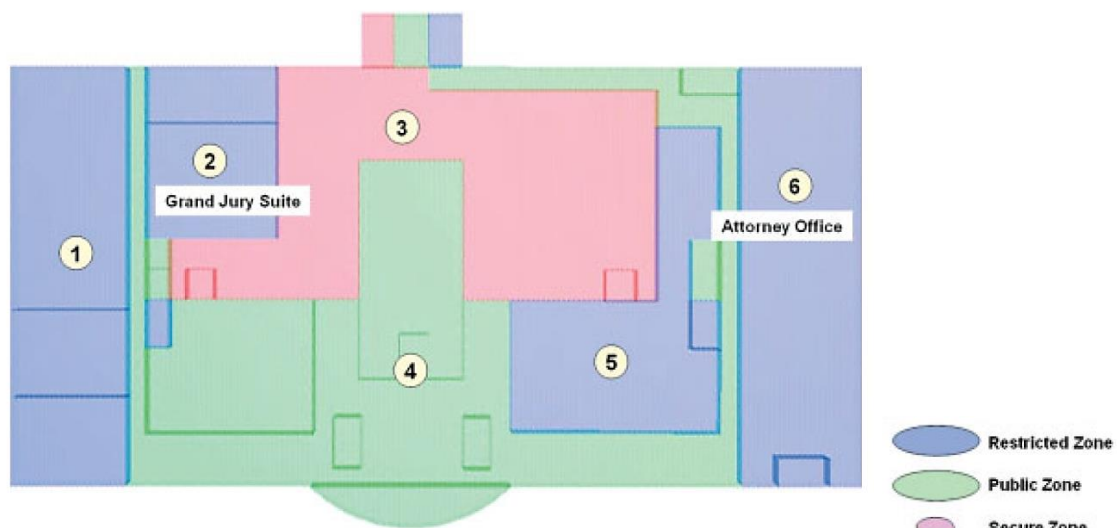


Abbildung 38: Beispiel für eine Richtlinie des Raumkonzeptes der GSA (Eastman 2009)

Bei der Entwicklung des *DAT* wurden von dem Entwicklungsteam 216 solcher Regelungen in der *U.S. Courts Design Guide* identifiziert und als Funktionen in das Plug-In fest implementiert. Da viele dieser Vorschriften die Zugänglich- und Erreichbarkeit von Räumen untereinander

regeln, wurde ein Relationen-Modell (siehe Abbildung 39) entwickelt, in dem die im Gebäudemodell gegebenen Verhältnisse innerhalb eines Graphen abgebildet werden. Mittels solcher Graphen können sowohl topologische als auch metrische Zusammenhänge im Raummodell dargestellt und somit nicht nur Zugänglichkeiten, sondern auch Distanzen berechnet werden (Eastman 2009).

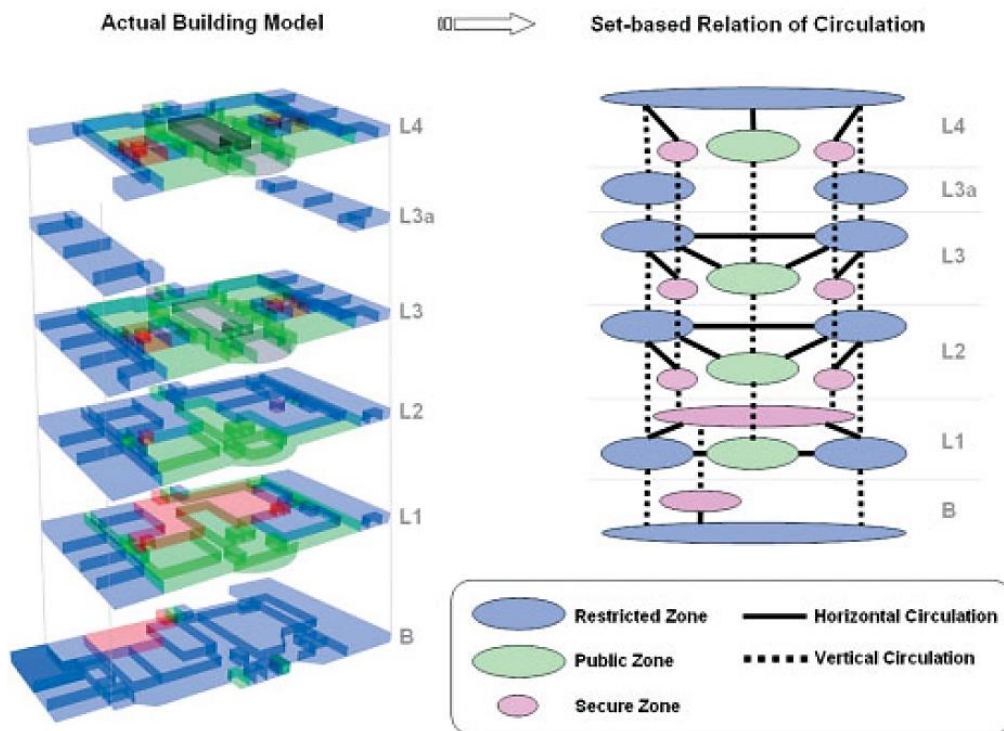


Abbildung 39: Relationen-Modell der Räume (Eastman 2009)

Das Ergebnis einer solchen Überprüfung wird im *SMC* dreidimensional (siehe Abbildung 40) dargestellt.

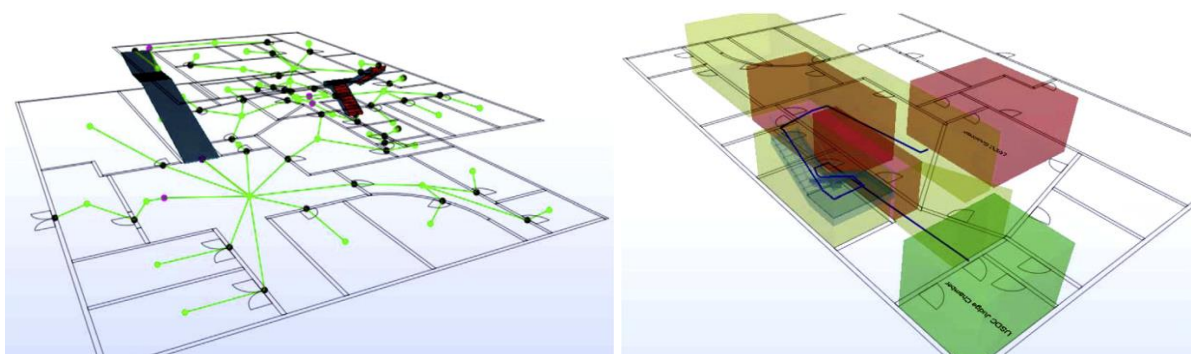


Abbildung 40: links – Visualisierung der Laufwege im Raummodell
rechts – Visualisierung einer fehlerhaften Laufroute
(Eastman, Lee, et al. 2009b)

Hinsichtlich der Anforderungen der *GSA* stellt das *DAT* eine gute Möglichkeit dar, den aufwendigen Überprüfungsprozess durch Automatisierung zu optimieren. Das Werkzeug wurde bis heute bei mehreren Bauprojekten von Justizgebäuden erfolgreich eingesetzt und automatisiert die Konformitätsüberprüfung zu ca. 90 %. Aber auch hier ergibt sich das Problem,

dass der eigentliche Prozess aufgrund der geschlossenen Struktur des *SMC* für den Anwender nur teilweise sichtbar ist (Eastman 2009).

3.1.4.4 HITOS

Im Jahr 2005 wurde von der norwegischen Behörde *Statsbygg* (2014), welche für die Immobilienverwaltung des norwegischen Staates zuständig und damit der größte Kunde im norwegischen Bauwesen ist, das Projekt *HITOS*² in Zusammenarbeit mit der Norwegischen Universität Tromsø ins Leben gerufen. Im Gegensatz zu den bereits vorgestellten Plattformen handelt es sich bei *HITOS* um ein F&E-Pilotprojekt, welches zum Ziel hat, die Forschung und Entwicklung der Plattform selbst, aber insbesondere auch moderner Planungsinstrumente und deren Interaktion untereinander voranzutreiben. Als Grundlage für das Projekt dient der Neubau der Gebäude der Fakultät für Ingenieur- und Erziehungswissenschaften der *Universität Tromsø* mit einer Investitionssumme von ca. 16 – 18 Millionen-€ und einer Nettogeschossfläche von ca. 5.150 m². Eine Planungsskizze des Bauprojektes ist in Abbildung 41 dargestellt (Mohus, Kvarsik und Lie 2006, Eberg, et al. 2006).

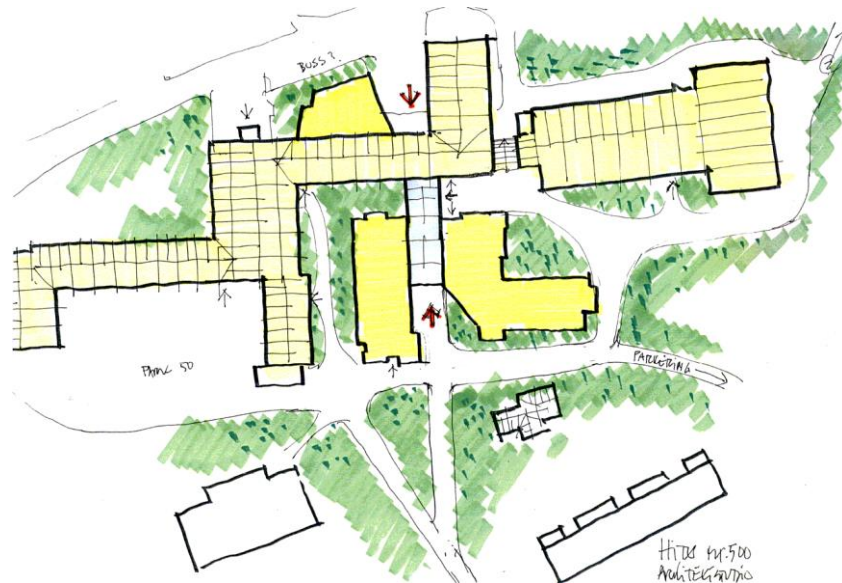


Abbildung 41: Skizze für den Neubau der Universität Tromsø (Mohus, Kvarsik und Lie 2006)

Auf Basis eines zentralen *IFC*-Modells sollen Informationen über sämtliche Lebenszyklen des Bauwerkes für alle Projektbeteiligten verfügbar sein und die Plattform so zu einem universellen Instrument für digitale Planung gemacht werden. Im Laufe des Projektes wurden unter anderem die bereits behandelten Softwarelösungen *CORENET e-PlanCheck*, *SMC* und der *EDM Model Server/Checker* eingesetzt, getestet und im Vergleich zu herkömmlichen Planungsmethoden abschließend bewertet.

Zu den Funktionalitäten der Plattformen zählen unter anderem auch *Code-Compliance-Checking*-Werkzeuge, welche sich insbesondere auf räumliche Analysen konzentrieren. Wie die Aufstellung und Struktur dieser Funktionen in Abbildung 42 zeigt, ist eine Kontrolle der

² Synonym für die Universität Tromsø

räumlichen Aufteilung des Gebäudemodells mit dem Tool *dRofus* (2014) sowie der Barrierefreiheit mit dem *SMC* umgesetzt worden (Eastman, Lee, et al. 2009b).

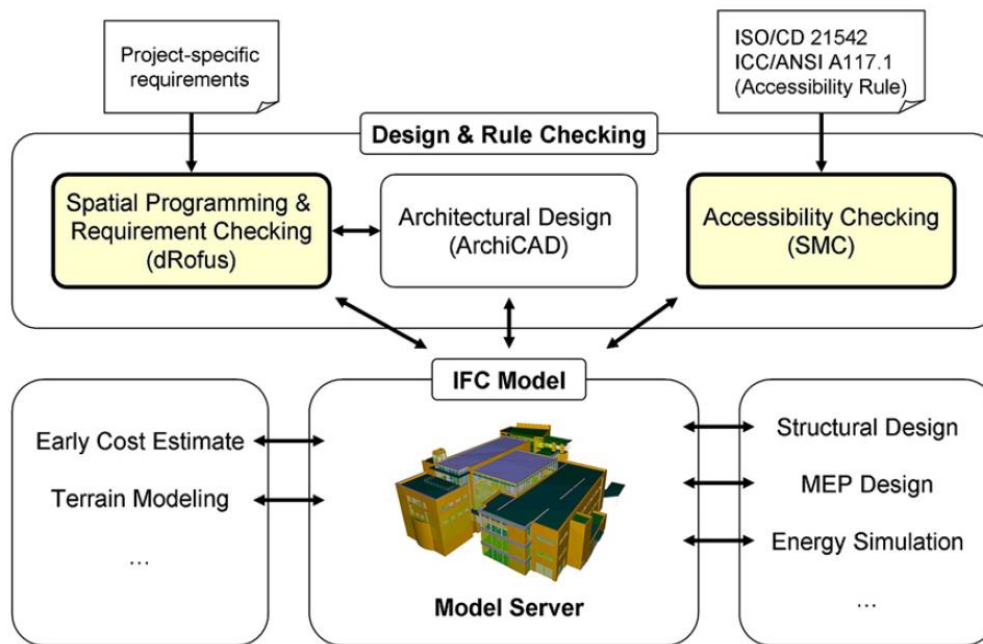


Abbildung 42: Code Compliance Checking im HITOS Projekt (Eastman, Lee, et al. 2009b)

Auf der *HITOS*-Plattform dient das Management- und Planungs-Tool *dRofus* dazu, Anforderungen an einzelne Räume hinsichtlich ihrer Lage und technischen Ausstattung für ein gesamtes Raummodell eines Bauwerkes zu formulieren und deren Einhaltung anschließend an Hand des zentralen *IFC*-Datenmodells zu kontrollieren. Die Anforderungen müssen in diesem Falle nicht an ein spezifisches Regelwerk gebunden sein, sondern können je nach individuellem Anspruch an das Bauwerk definiert werden. Hierfür werden in einem Editor Vorgaben für unterschiedliche Raumklassen deklariert und können den einzelnen Räumen eines Gebäudemodells zugewiesen werden (siehe Abbildung 43 rechts).

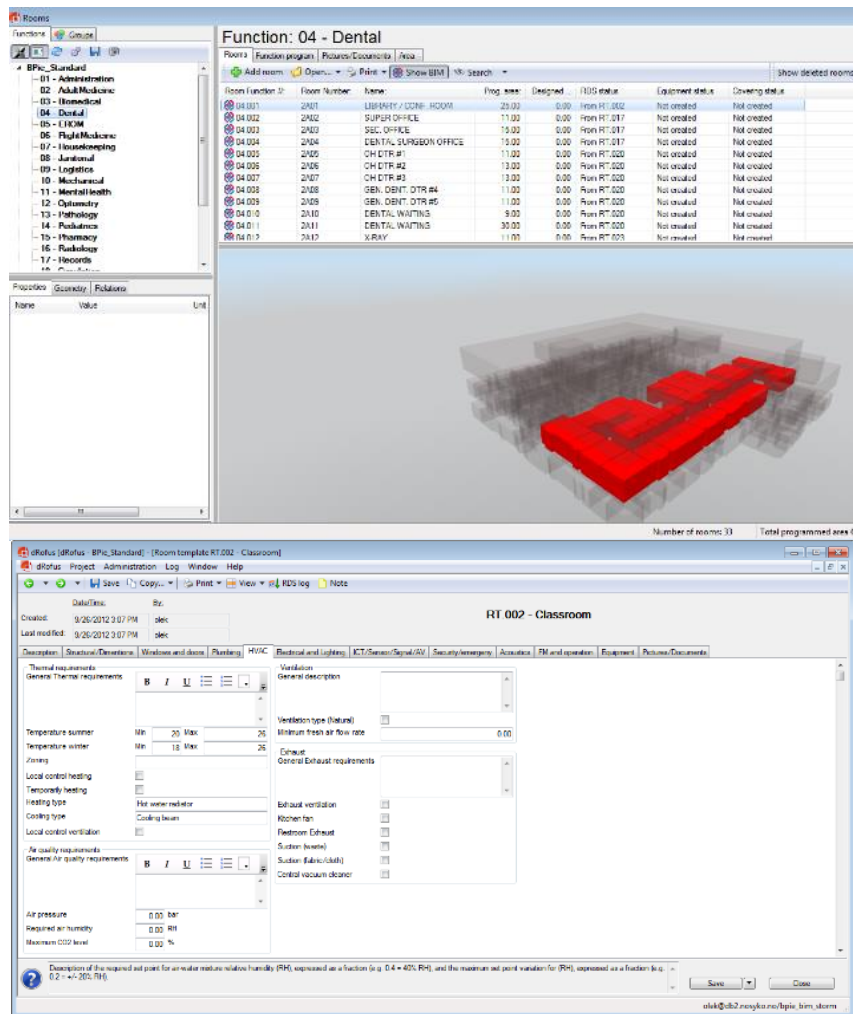


Abbildung 43:
oben - Userinterface von dRofus
unten – Definition von Anforderungen über Raum-Templates
(dRofus 2014)

Über ein webbasiertes Interface (siehe Abbildung 43 links) können parallel mehrerer Bauprojekte verwaltet werden. Daher eignet sich das Tool auch für den Einsatz bei mehrteiligen Bauprojekten, wie dem *HITOS*. *dRofus* greift lediglich auf eine fest definierte Teilmenge des jeweiligen zentralen Datenmodells zurück, welche die geplante Struktur des Raummodells und die Beschaffenheit einzelner Räume beschreibt und diese schließlich mit einer hierarchischen Datenstruktur, die sich aus den Anforderungen und den zugewiesenen Raumtemplates des Anwenders ergibt, vergleicht (Eastman, Lee, et al. 2009b).

Neben der Raummodell-Analyse bietet die *HITOS*-Plattform auch eine Überprüfung der Barrierefreiheit eines Gebäudemodells. Hierzu wurde eine interaktive Schnittstelle von den internationalen Normen *ISO 21542:2011* (ISO 2011) und *ICC/ANSI A117.1*. (ISO 2003) zu dem *SMC* geschaffen, welcher bereits in Abschnitt 3.1.4.3 vorgestellt wurde.

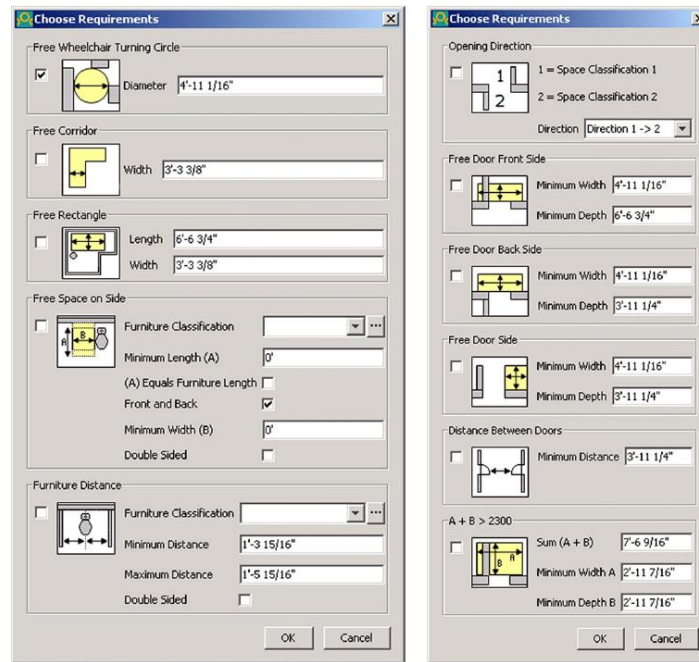


Abbildung 44: Eingabe der Parameter für die Überprüfung der Barrierefreiheit mit dem SMC auf der HITOS-Plattform (Eastman, Lee, et al. 2009b)

Da diese internationalen Normen eine Reihe von Justierungen der Überprüfung fordern, können diese über Parameter in einer Nutzereingabe vorgenommen und anschließend über eine Datenstruktur an die fest implementierten Überprüfungsfunktionen des *SMC* übergeben werden. Der eigentliche Überprüfungsprozess auf Seiten des *SMC* bleibt dabei weiterhin in einem geschlossenen System und nur die Ergebnisse werden für den Anwender visualisiert (siehe Abbildung 45).

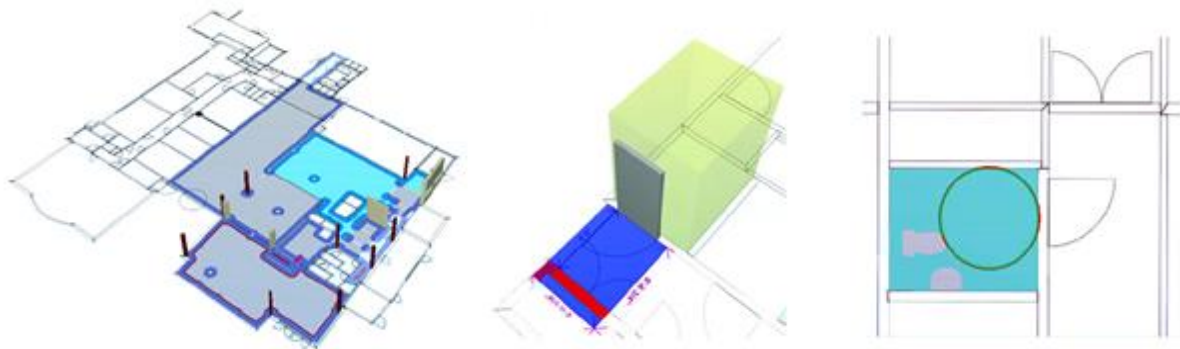


Abbildung 45: Ergebnisse einer Überprüfung zu Barrierefreiheit mit dem SMC auf der HITOS-Plattform
links – Überprüfung der Zugänglichkeit von Räumen
zentral – Überlappung der Türblätter beim Öffnen
rechts – Wenderadius eines Rollstuhlfahrers in einem Raum
(Eastman, Lee, et al. 2009b)

3.1.4.5 Ausblick für weitere Entwicklungen

Trotz der Vielzahl mittlerweile etablierter Plattformen für das *Building Information Modeling* gibt es auch aktuell noch immer wieder neue Bestrebungen, zentrale Datensysteme für Bauprojekte zu entwickeln. Dieses ist darauf zurückzuführen, dass viele der existierenden

Softwarelösungen zwar umfassende und vielseitige Möglichkeiten bieten, Prozesse im Bauwesen zu optimieren, jedoch durch verwendete Komponenten, wie z.B. das Datenmodell, dem externen Anwender starre Rahmenbedingungen aufzwingen und ihn so in seiner Freiheit einschränken. Um dieses Problem zu umgehen, entscheiden sich viele Kunden für maßgeschneiderte Lösungen, die sie entweder aus bereits existierenden Produkten zusammenstellen oder vollständig selbst entwickeln.

Zu diesen Kunden gehört unter anderem auch die Norwegischen Regierungsbehörde *Norwegian Building Authority* (2014), welche die Unternehmensberatung *Holte Consulting* beauftragt hat, eine Studie zu etablierten Plattformen durchzuführen und insbesondere deren Funktionalitäten im Bereich des *Code Compliance Checking* zu bewerten. Die Ergebnisse dieser Studie sind in dem Report *ByggNett Status Survey* (Holte Consulting 2014) festgehalten.

3.2 Struktur des Code Compliance Checking

Obwohl sich die in Abschnitt 3.1 vorgestellten Ansätze jeweils durch ihre eigenen Methoden zur automatisierten Konformitätsüberprüfung auszeichnen, lässt sich eine gemeinsame Struktur identifizieren. Eastman (2009b) definiert diese als Ablauf und Interaktion einzelner Prozessschritte.

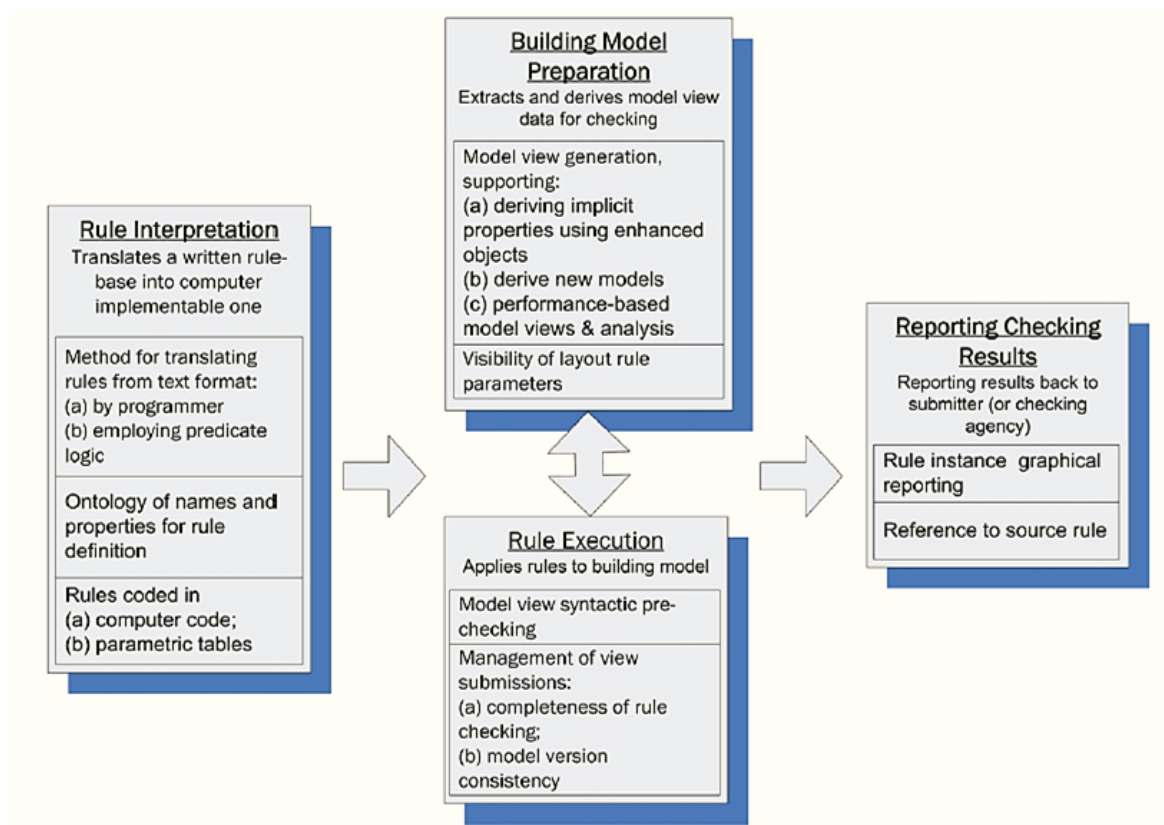


Abbildung 46: Prozessschritte einer Konformitätsüberprüfung (Eastman, Lee, et al. 2009b)

Wie in Abbildung 46 dargestellt unterteilt er den Gesamtprozess des *Automated Code Compliance Checking* in vier Bestandteile:

– *Rule Interpretation*

Um ein Regelwerk überhaupt erst für eine Maschine lesbar zu machen, muss dieses in einem ersten Schritt in eine maschineninterpretierbare Sprache übersetzt werden. Wie bereits bei den bisher entwickelten Ansätzen zu erkennen ist, stellt dieser Überprüfungsprozess die Basis der jeweiligen Methodik dar, da mit der Festlegung der Art der Übersetzung viele Möglichkeiten aber auch Grenzen für die folgenden Prozessschritte vorgegeben werden.

– *Building Model Preparation*

Nachdem die Inhalte des entsprechenden Regelwerks definiert und niedergelegt sind, ist es notwendig, das verwendete Gebäudemodell für die entsprechende Überprüfung vorzubereiten. Die in der vorliegenden Arbeit behandelten Ansätze haben gezeigt, dass es sehr schnell zu Komplikationen mit Gebäudemodellen aufgrund von Inkonsistenzen, also widersprüchlichen, falschen oder nicht vorhandenen Informationen kommt. Eine direkte Verwendung ist daher

nicht sinnvoll, sondern in einem weiteren Schritt sollte das digitale Modell vor der Überprüfung entweder vor- oder aber aufbereitet werden.

– *Rule Execution*

Mit dem aufbereiteten Modell und den digitalen Regelwerken kann nun der eigentliche Überprüfungsprozess erfolgen. In diesem Schritt werden die Regelwerke von einer ausführenden Instanz ausgelesen, interpretiert und mittels der im Gebäudemodell enthaltenen Informationen verarbeitet.

– *Reporting Checking Results*

In einem letzten Prozessschritt werden die Ergebnisse der Überprüfung schließlich für den Anwender aufbereitet und festgehalten. Dieser Prozessschritt wird gerade in den moderneren Ansätzen zum *Code Compliance Checking* vorrangig graphisch gestaltet.

3.3 Anforderungen an den Überprüfungsprozess

An Hand des in Abschnitt 3.1 vorgestellten Stands der Wissenschaft und der Gliederung von Eastman (2009b) lassen sich nun Anforderungen und Rahmenbedingungen für die einzelnen Prozessschritte definieren, die Grundlagen für die erfolgreiche Entwicklung einer neuen Methode zum *Code Compliance Checking* sind. Diese Anforderungen sollen in den folgenden Unterkapiteln für die jeweiligen Einzelschritte des Gesamtprozesses vorgestellt werden.

3.3.1 Übersetzung und Auslegung der Regelwerke

Die Idee der Digitalisierung von Sprache in mündlicher oder schriftlicher Form existiert seit den Anfängen der Computerwissenschaften und ist auch heute in den verschiedenen Anwendungsbereichen ein relevantes Thema. Es geht prinzipiell darum, den Informationsgehalt von gesprochenem und geschriebenem Wort möglichst präzise in Binärcode zu übersetzen. Das Problem hierbei ist, dass Sprache Informationen in unterschiedliche Formen und Ebenen transportieren kann. Der Sprachwissenschaftler Karl Bühler unterscheidet drei verschiedene Arten von Informationen, welche in Sprache übermittelt werden können (Becker 2013): Ausdruck (*expressiv*), Appell (*illokutionär*, *kommunikativ*) und Darstellung (*deskriptiv*, *kognitiv*). Da sich der Ausdruck der Sprache vor allem auf das Wesen des Senders, also ein Individuum, bezieht, kann dieser Aspekt in der vorliegenden Arbeit außer Acht gelassen werden. Die beiden Elemente Appell und Darstellung hingegen beschreiben sehr präzise die Gestalt der Informationen, wie diese in Vorschriften von Normen enthalten sind. Eine Norm hat die Funktion, dem Bearbeiter einen Sachverhalt darzustellen und anschließend auf dieser Basis die Einhaltung von Regeln und Rahmenbedingungen einzufordern.

Ein allgemeiner Lösungsansatz, um Sprach-Informationen zu digitalisieren, ist die Implementierung einer Schnittstelle, der sogenannten Mensch-Maschine-Kommunikation (Schenk und Rigoll 2010). Ein bekanntes Beispiel für eine solche Schnittstelle ist die Entwicklung von Spracherkennungssoftware, die in den letzten Jahren intensiv vorangetrieben wurde und sich auf eine ausgeprägte Interaktion von Mensch und Maschine konzentriert. Trotz des rasanten technologischen Fortschritts der letzten Jahrzehnte und den daraus resultierenden

Werkzeugen, stellt dieser Bereich immer noch große Herausforderungen an die Entwickler (Becker 2013).

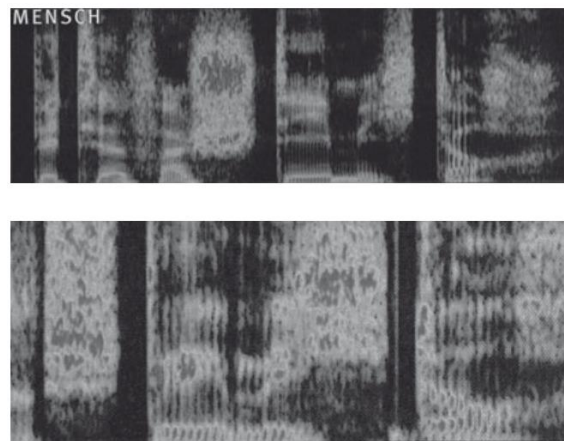


Abbildung 47: Analyse des Satzes "Bitte versteh mich doch", gesprochen von Mensch (oben) und Maschine (unten), mittels eines Sonagramm (Becker 2013)

Diese Herausforderung der Mensch-Maschine-Kommunikation gilt auch für den Übersetzungsprozess des *Code Compliance Checking*.

Da sich Regelwerke im Bauwesen auf den gesamten Lebenszyklus eines Gebäudes und somit auf alle Fachdisziplinen beziehen, enthalten diese eine Vielzahl von Informationen, welche auf vielfältige Art und Weise dargestellt sein können. Diese Darstellungsweisen reichen von einfach und klar strukturierten Tabellen mit Randbedingungen über Zeichnungen bis hin zu in Fließtext beschriebenen Sachverhalten. Viele der in Kapitel 3.1 vorgestellten Methoden setzen für ihren Übersetzungsprozess einen klar definierten Rahmen durch die Verwendung logischer Systeme, welche in unterschiedlicher Form ausgestaltet sein können.

Die Formulierung der Elemente einer Norm mittels einer logischen Syntax fällt für einen Anwender – insbesondere, wenn er keine große Erfahrung im Bereich dieser Systeme hat - mit zunehmender Komplexität des Inhaltes sehr schwer und kann anschließend auch nur mit großem Aufwand nachvollzogen werden. Vor allem die Darstellung von Zusammenhängen einzelner Elemente einer Norm, z.B. von geometrischen Abhängigkeiten, lassen sich mittels logischer Operatoren zwar darstellen, jedoch verlieren diese Beschreibungen sehr schnell an Übersichtlichkeit und blockieren somit die Schnittstelle zwischen Mensch und Maschine.

Dieses bedeutet, dass mit den logischen Systemen in vielen Ansätzen zwar die Basis des Übersetzungsprozesses ausgebaut, die Interaktion aber, welche eine bedeutende Rolle für den Erfolg einer solchen Methode spielt, nicht ausreichend betont wird. Stellt die Syntax aufgrund ihrer Komplexität zu hohe Ansprüche an den Anwender, verliert der automatisierte Überprüfungsprozess seinen ursprünglich geplanten Vorteil, Zeit und Aufwand zu minimieren.

Zusammenfassend lassen sich an dieser Stelle die Anforderungen festhalten, dass grundlegend alle Elemente einer Vorschrift oder Norm des Bauwesens für ein *Automated Code Compliance Checking* innerhalb einer Syntax abbildbar sein müssen. Zugleich muss jedoch innerhalb dieser Syntax die Interaktion von Mensch und Maschine in den Fokus gerückt werden, damit die Automatisierung des Übersetzungsprozesses ihre Vorteile an dieser Stelle nicht einbüßt. Ziel

sollte es sein, jedem Anwender die Möglichkeit zu geben, verhältnismäßig einfach und schnell gewünschte Regelwerke aus den jeweiligen Fachdisziplinen zu definieren.

3.3.2 Vor- und Aufbereitung des Gebäudemodells

Neben dem digitalisierten Regelwerk ist das Gebäudemodell die zweite Informationsquelle für den Überprüfungsprozess. Wie viele der innerhalb der vorliegenden Arbeit vorgestellten Ansätze zeigen, ist die direkte Verwendung eines Gebäudemodells für das *Automated Code Compliance Checking* problematisch.

Während des Überprüfungsprozesses muss wiederholt an geeigneter Stelle auf den Informationsgehalt des Modells zugegriffen werden. Die Komplexität der Struktur eines Datenmodells gibt vor, wie leicht dieser Zugriff und somit auch der Weiterverarbeitungsprozess innerhalb der Syntax gestaltet werden kann. Die Syntax wiederum bestimmt wie gut die Schnittstelle zwischen Mensch und Maschine, also die Interaktion ausgebaut ist.

Des Weiteren gilt, dass die Fehlerfreiheit eines Überprüfungsprozesses direkt mit der Qualität des Gebäudemodells korrespondiert. Enthält eine Informationsquelle der Überprüfung Fehler, macht dieses den gesamten Prozess wertlos.

Viele der in Abschnitt 3.1 vorgestellten Ansätze verwenden das *IFC*-Format, welches seine Stärken insbesondere in der Interoperabilität und in dem Austausch von Daten besitzt. Jedoch zeigen Beetz et al. (2009) an dieser Stelle auch die großen Nachteile dieses offenen Datenstandards auf (siehe Abschnitt 3.1.3.2).

Ebertshäuser und von Both (2013) belegen die allgemeine Unzufriedenheit in der Baubranche mit der Qualität von Gebäudemodellen in einer Umfrage, welche in Abbildung 48 dargestellt ist.

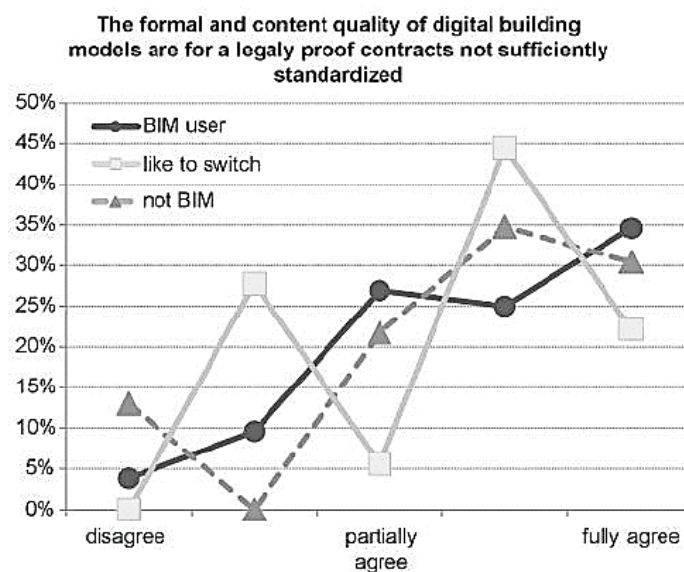


Abbildung 48: Ergebnis einer Umfrage zu der mangelnden Qualität von Gebäudemodellen (Ebertshäuser und von Both 2013)

An gleicher Stelle zeigt Lee (2010), dass diese Probleme gelöst werden können, indem die Informationen des *IFC*-Modells auf ein eigenes Datenmodell abgebildet werden (siehe

Abschnitt 3.1.2.3). In seiner Methode nutzt er nicht nur eine eigene Datenstruktur, an welcher sich seine entwickelte Syntax orientieren kann, sondern er schränkt zusätzlich den Informationsgehalt auf eine Teilmenge ein, auf die sich der anstehende Überprüfungsprozess bezieht. Auf diese Weise kann die Gefahr gemindert werden, dass es zu Inkonsistenzen, Widersprüchen oder aber fehlerhaften Informationen kommt, welche den Überprüfungsprozess letztlich wertlos machen. Gleichzeitig kann mit dieser Aufbereitung des Datenmodells die Syntax der *BERA*-Sprache gerade in Hinsicht auf die Datenzugriffe sehr einfach gehalten werden.

Jedoch muss an dieser Stelle auch beachtet werden, dass durch die Abbildung des Datenmodells zusätzliche Schnittstellen geschaffen werden, welche das Risiko mit sich bringen, dass es hier wiederum zu Datenfehlern oder –verlusten kommen kann.

Zusammenfassend kann festgehalten werden, dass genau abzuwägen ist inwieweit eine Auf- und Vorbereitung des Datenmodells innerhalb des Überprüfungsprozesses sinnvoll ist. Zeichnet sich das für den Überprüfungsprozess gewählte Datenmodell durch einen einfachen und direkten Datenzugriff aus, kann eine Ab- oder Umbildung der Daten auch hinderlich sein.

3.3.3 Ausführung des Überprüfungsprozesses

Die Fehlerfreiheit der Ergebnisse einer Konformitätsüberprüfung liegt in dem Verantwortungsbereich des Anwenders. Daher kommen bei der zurzeit üblichen, manuellen Überprüfung von Regelwerk und Gestaltungsplanung insbesondere die Erfahrung, die Umsicht und die Sorgfalt des Sachbearbeiters zum Tragen. Es ist empfehlenswert und auch weitgehend Praxis, dass jeder einzelne Schritt einer Überprüfung an Hand von Erfahrungswerten oder überschlägigen Rechnungen hinsichtlich seiner Plausibilität überprüft wird. Durch das schrittweise Vorgehen bei der manuellen Bearbeitung lässt sich diese Überprüfung sehr gut in den Arbeitsprozess integrieren. Eine Automatisierung hingegen bringt die Gefahr mit sich, dass diese Kontrolle an Bedeutung verliert und die Verantwortung allein der Maschine übertragen wird, was bereits aus rein rechtlichen Aspekten nicht möglich sein kann. Seit dem Einzug der Computerwissenschaften in das Bauwesen besteht dieses Problem in vielen Bereichen, wie z.B. der Berechnung der Statik, bei dem der Bearbeiter nicht nur auf die erhaltenen Zahlen eines maschinellen Berechnungsprozesses vertrauen kann, sondern in der Pflicht steht, diese Ergebnisse gleichzeitig oder nachlaufend zu überprüfen.

Einige der vorgestellten Ansätze, wie beispielsweise der *SMC* (siehe Abschnitt 3.1.4.3) verarbeiten die Informationen, ohne dass der Nutzer die im Hintergrund laufenden Prozesse einsehen kann. Diese Prozesse, welche lediglich Input- und Outputdaten sichtbar machen, werden auch *Black-Box*-Methoden genannt (von Bertalanffy 1972). Die mangelnde Transparenz und Übersichtlichkeit einer solchen Methode erlauben keine Plausibilitätsüberprüfung während des Prozesses und stellen somit die Ergebnisse grundsätzlich in Frage. Dieses Problem kann lediglich gelöst werden, indem der Überprüfungsprozess so transparent und offen wie möglich gestaltet wird.

Das Gegenmodell zu der *Black-Box* ist die sogenannte *White-Box*-Methode, welche sowohl die Elemente als auch die Prozess- und Zwischenschritte für den Anwender sicht- und nachvollziehbar macht. Grundstein für die Anwendung dieser Methode ist die Lesbarkeit der

digital abgebildeten Informationen für Mensch und Maschine. Auf diese Weise kann der Nutzer gezwungenermaßen in den Überprüfungsprozess integriert werden, damit es so automatisch zu einer Interaktion an Stelle einer rein maschinellen Durchführung des Prozesses kommt. Der Unterschied zwischen den beiden Methoden ist in Abbildung 49 schematisch dargestellt.

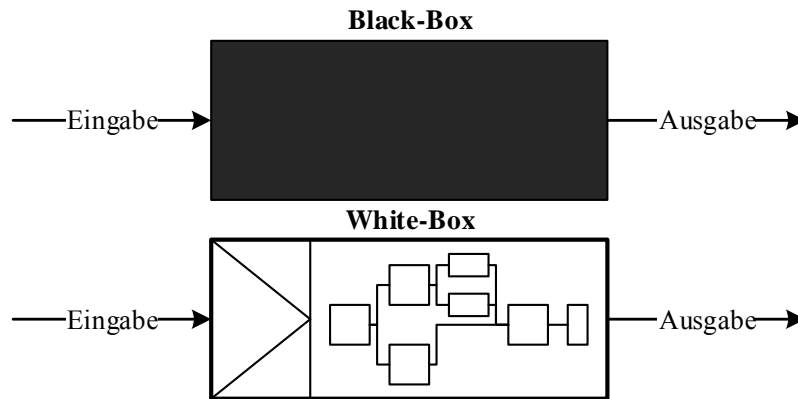


Abbildung 49: Schematische Darstellung einer Black-Box- und White-Box-Methode

Generell gilt für das *Automated Code Compliance Checking*, dass der eigentliche Prozess der Überprüfung soweit wie möglich transparent und offen gehalten werden sollte, damit der Anwender an jeder Stelle des Prozesses die aktuellen Werte einsehen und eine Plausibilitätsüberprüfung vornehmen kann. Diese Überprüfungen könnten bei stark geometrisch basierten Vorschriften auch durch die Visualisierung des aktuellen Sachverhaltes unterstützt werden.

3.3.4 Aufbereitung der Ergebnisse des Prozesse

Bei der endgültigen Aufbereitung der Ergebnisse des Checks spielt die Plausibilitätsüberprüfung der erhaltenen Ergebnisse ebenfalls eine bedeutende Rolle. Stichprobenartige Überprüfungen, Visualisierungen und einfache Plausibilitätsprüfungen können dem Anwender dabei helfen, die erhaltenen Ergebnisse hinsichtlich ihrer Fehlerfreiheit zu überprüfen.

Die in der vorliegenden Arbeit vorgestellten Methoden zeigen, dass an dieser Stelle die Entwicklung aufgrund der Etablierung der *CAD*-Systeme in dem Bauwesen bereits sehr weit vorangeschritten ist und sich die Ergebnisse aussagekräftig darstellen lassen. Im Sinne einer nachhaltigen Interaktion von Mensch und Maschine sollte insbesondere die graphische Aufbereitung der Ergebnisse dynamisch möglich sein, d.h. dass zu jedem einzelnen Schritt einer Überprüfung die aktuellen Elemente und Ergebnisse dargestellt werden können.

4 Code Compliance Checking auf Basis einer visuellen Sprache

Die in Abschnitt 3.3 aufgestellten Anforderungen an die einzelnen Schritte der Konformitätsüberprüfung führen zu der Fragestellung, mit welcher Methode alle Anforderungen innerhalb eines Ansatzes erfüllt werden können. In den folgenden Unterkapiteln sollen Methodik und Aufbau eines neuen Ansatzes zum *Code Compliance Checking* auf Basis einer visuellen Sprache vorgestellt werden, welcher im Zuge der vorliegenden Arbeit entwickelt wurde.

4.1 Methodik

Die Grundlage aller bisherigen Ansätze ist die Abbildung des Informationsgehalts einer Norm innerhalb eines digitalen Informationssystems. Zu der Vielzahl von Möglichkeiten zur Ausgestaltung eines solchen Systems gehört unter anderem auch die sogenannte *Visual Programming Language (VPL)*, deren Syntax nicht mit textuellen sondern mit graphischen Elementen beschrieben wird.

In den nachfolgenden Unterkapiteln soll zunächst ein Überblick über die Entwicklung des *Visual Programming* mit Hilfe ausgewählter Forschungsansätze gegeben werden. Anschließend werden die Grundlagen sowie Vor- und Nachteile zu dieser Methode aufgezeigt, um so weitere Rahmenbedingungen für die anschließende Entwicklung eines neuen Ansatzes zum *Code Compliance Checking* vorzugeben.

4.1.1 Historische Entwicklung des Visual Programming

Erste Forschungen im Bereich der visuellen Programmiersprachen lassen sich bis in die 1950er Jahre zurückführen, in denen die Idee aufkam, dass ein Anwender auch ohne fundierte Programmierkenntnisse in der Lage sein sollte, Verarbeitungssysteme zu erstellen, zu steuern und zu bedienen. Myers (1990) beschreibt, dass der Bedarf an einer Lösung dieses Problems zur damaligen Zeit durch den rasanten Anstieg der Zahl der Computernutzer größer geworden war und daher die Forschung in dem Bereich der *VPL* intensiviert wurde. Als ersten markanten Forschungsansatz nennt er die Entwicklung des *Graphical Program Editor* von William R. Sutherland im Jahre 1966 am *Massachusetts Institute of Technology* im Zuge seiner Dissertation zum Thema „*On-Line Graphical Specification of Computer Procedures*“. Der *Graphical Program Editor* war darauf ausgelegt, Schaltpläne für Hardwarekomponenten zu visualisieren und diese anschließend auch zu interpretieren. Somit kann diese Entwicklung als erstes visuelles Programmiersystem bezeichnet werden.

Durch den technischen Fortschritt in den folgenden Jahrzehnten und die damit verbundene Verbesserung von Soft- und Hardwarekomponenten konnten auch im Bereich der *VPL* große Fortschritte erzielt werden. Als weiterer Meilenstein gilt die Entwicklung der Programmierumgebung *Pygmalion*, welche von David C. Smith an der *Stanford University* im Jahre 1975 entwickelt wurde (Schiffer 1998). Die grundlegende Überlegung von Smith war, dass es für das kreative Denken eines Anwenders hilfreich sei, mit Bildern, also z.B. mit Piktogrammen, anstatt mit Fließ- oder Codetextbeschreibungen zu arbeiten, weshalb er seine Entwicklung in einem Rückblick auch als *Executable Electronic Blackboard* bezeichnete. Eine

wichtige Grundlage für die Entwicklung von *Pygmalion* war die damalige, quasi parallel verlaufende Einführung von Benutzerschnittstellen, sogenannter *User Interfaces (UI)*, in den Computerwissenschaften. In Abbildung 50 ist die Benutzeroberfläche von *Pygmalion* dargestellt, mit welcher es möglich ist, einfache Fakultätsfunktionen visuell zu formulieren und anschließend zu berechnen.

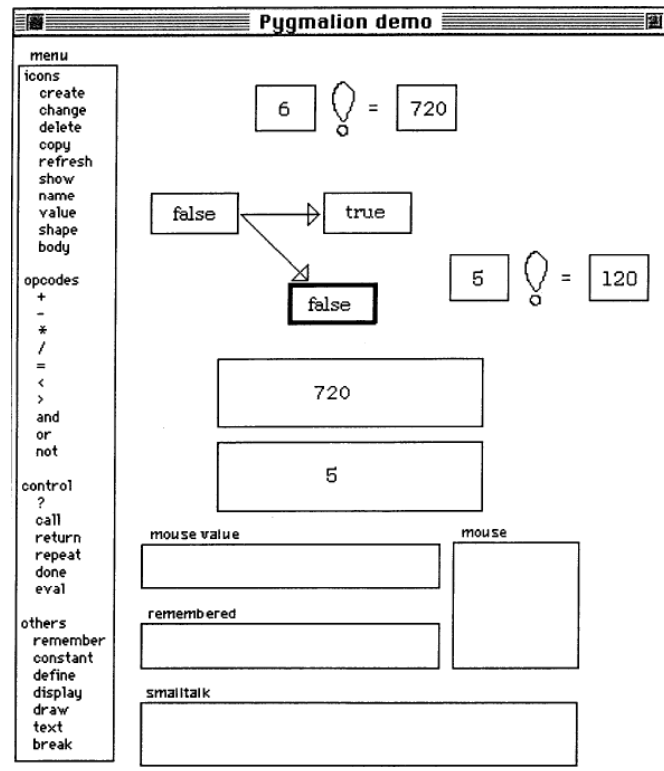


Abbildung 50: Pygmalion Userinterface für Fakultätsfunktionen (Schiffer 1998)

Ein erstes kommerzielles Tool auf Basis einer VPL wurde in den 1990er Jahren von der US-amerikanischen Firma *National Instruments* (2014) mit *LabVIEW* entwickelt, welches auch heute noch weiterentwickelt und verwendet wird. Es handelt sich hierbei um eine Software im Bereich der Elektro- und Messtechnik, welche Blockdiagramme mit Hilfe von graphischen Elementen erstellen und anschließend auswerten kann (siehe Abbildung 51).

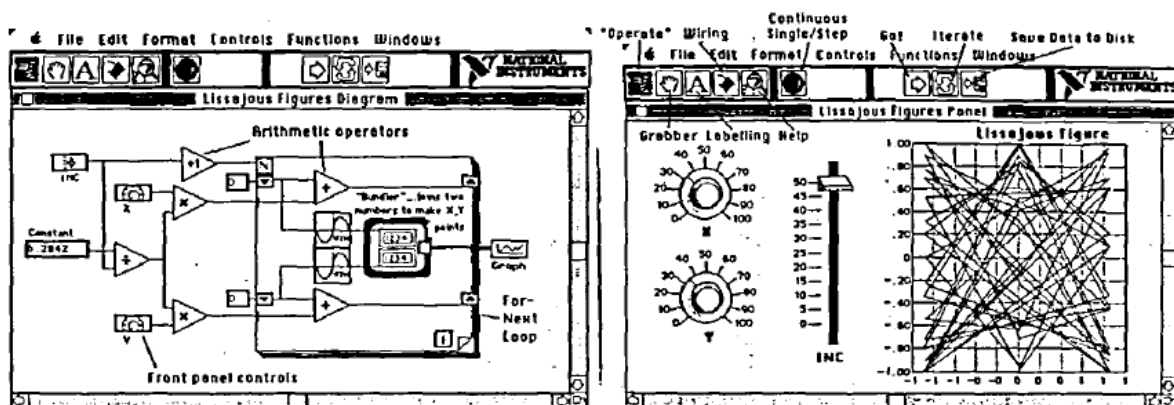


Abbildung 51:
links: graphische Eingabe des Blockdiagramms in LabVIEW
rechts: resultierendes Userinterface für die Auswertung in LabVIEW
(Myers 1990)

Auch im Bereich der Prädikatenlogik (siehe Abschnitt 3.1.1) wurden Forschungen durchgeführt, um Formulierungssysteme für logische Aussagen mittels der *VPL* zu unterstützen. Mit dem *Visual Logic Programming* entwickelten Pau und Olason (1991) eine Möglichkeit, prädikatenlogische Aussagen mit Hilfe graphischer Elemente zu formulieren, diese anschließend automatisch in die *PROLOG*-Sprache (siehe Abschnitt 3.1.1) zu übersetzen und so zu interpretieren. Hierfür entwickelten sie einen graphischen Editor (siehe Abbildung 52 links) als Benutzerschnittstelle, in welchem der Nutzer über Knoten- und Kantenobjekte einen Graphen zeichnen konnte, welcher wiederum die logische Aussage repräsentierte.

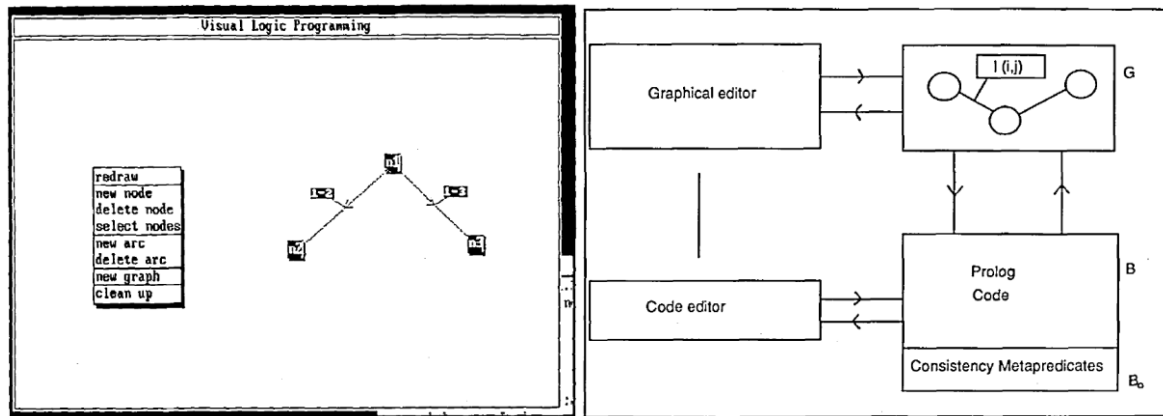


Abbildung 52:
rechts: Softwarearchitektur der Visual Logic Programming
links: Graphischer Editor der Visual Logic Programming
(Pau und Olason 1991)

Ein neuerer Ansatz war die Entwicklung von *Scratch* im Jahre 2007 am *Massachusetts Institute of Technology*. Maloney et al. (2010) verfolgten das Ziel eine Entwicklerumgebung für jüngere Anwender zu schaffen und in dieser das Erlernen von Programmiersprachen visuell zu unterstützen. Daher kann *Scratch* als Erziehungs- und Lehrmittel betrachtet werden.

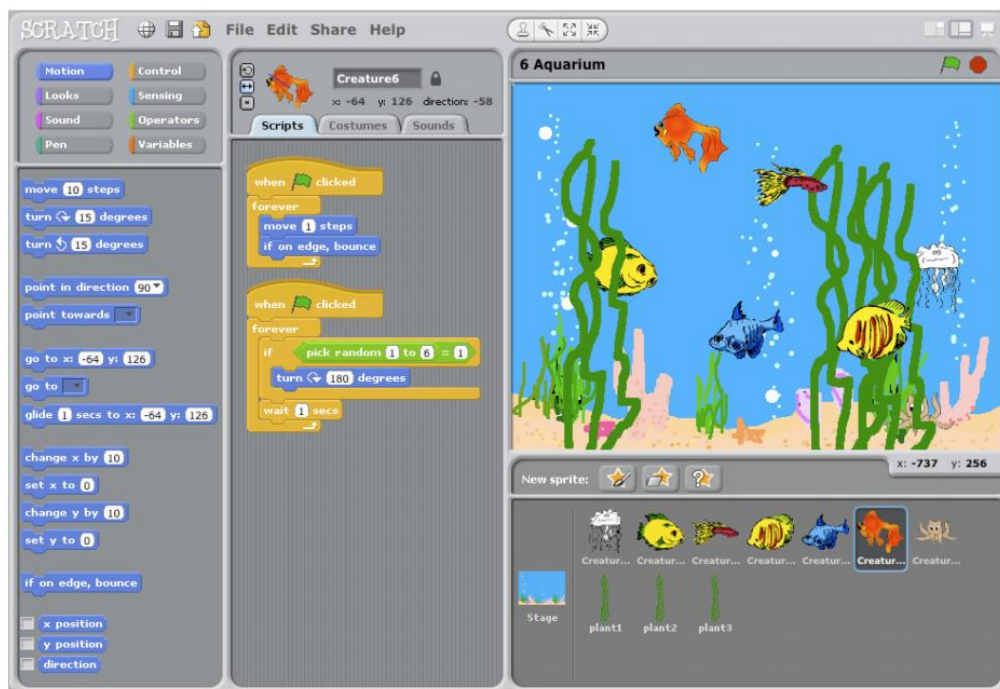


Abbildung 53: Userinterface von Scratch (Maloney, et al. 2010)

Wie in Abbildung 53 zu sehen ist, ist die Oberfläche von *Scratch* auf die junge Klientel ausgelegt und arbeitet stark mit visuellen Elementen. Die einzelnen Objekte der Syntax geben dem Anwender - ähnlich wie Puzzleteile – die vorhandenen Entwicklungsmöglichkeiten vor. Gleichzeitig wird jeder Schritt der Entwicklung in einem separaten Fenster visualisiert, damit nicht nur das Ergebnis kontrolliert werden kann, sondern auch die Motivation des Anwenders steigt.

Das visuelle Programmieren spielt auch heute noch eine Rolle, wobei sich die Entwicklungen vermehrt auf den jeweiligen Anwendungsbereich fokussieren und sich von den Programmiersprachen entfernen. Die *VPL* wird zurzeit verstärkt als Steuerungs- und Modifikationswerkzeug für Informationssysteme unterschiedlichster Software eingesetzt und dient heute weniger als visuelles Hilfsmittel für Entwicklungsumgebungen.

Als Beispiel sei an dieser Stelle die 3D-Modellierung-Software *Rhinoceros* (McNeel 2014) genannt, welches im Zusammenhang mit dem Plug-In *Grasshopper* (2014) um eine *VPL*-Funktionalität erweitert wird. Dem Anwender ist es mittels der graphischen Programmiersprache möglich, Anforderungen und Randbedingungen mit Hilfe eines Graphen zu definieren, und so eine Geometrie zu erstellen und zu steuern. Über Modifikation und Manipulation der Parameter können auch sehr komplexe Geometrien - wie sie in Abbildung 54 beispielhaft dargestellt ist - kontrolliert werden.

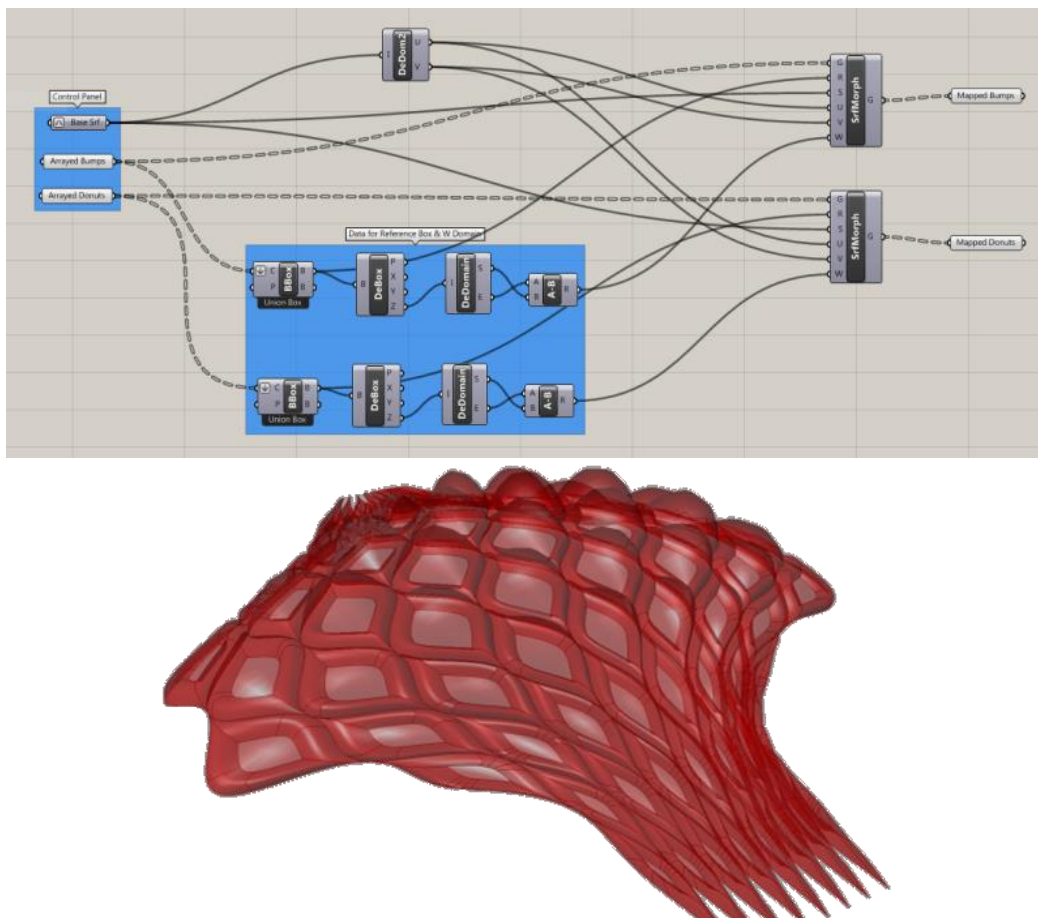


Abbildung 54:
oben: Zeichenfläche für die Erstellung von Randbedingungen in Grasshopper
unten: Beispiel-Geometrie, die in Rhinoceros mit Grasshopper erstellt wurde

Auch im Bereich des Bauwesens und *Building Information Modeling* gibt es bereits Tools und Plug-Ins, welche die vorhandenen *CAD*-Programme um *VPL*-Funktionalitäten erweitern. Ein Beispiel hierfür ist das Tool *Dynamo* (2014), welches die *BIM*-Software *Autodesk Revit* (2014) um eine *VPL*-Funktionalität ergänzt. Auch hier übernimmt die *VPL* die Aufgabe eines Steuerungs- und Modifikationswerkzeugs (siehe Abbildung 55).

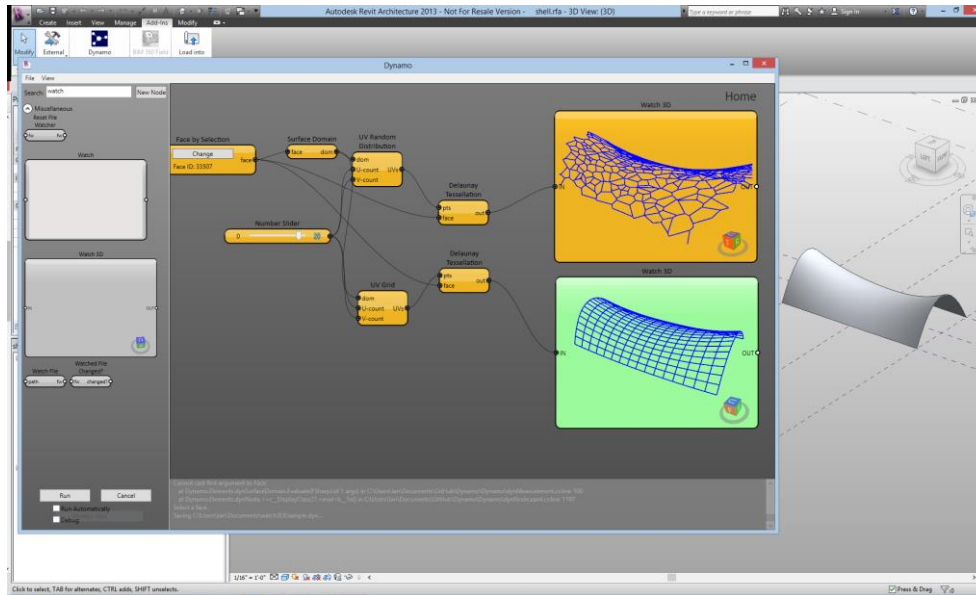


Abbildung 55: VPL-Werkzeug Dynamo in Autodesk Revit (Dynamo 2014)

4.1.2 Grundlagen einer visuellen Sprache

Schiffer (1998) definiert den Begriff „visuelle Sprache“ als „eine formale Sprache mit visueller Syntax oder visueller Semantik und dynamischer oder statischer Zeichengebung“.

Der Begriff „visuell“ bedeutet „das Sehen oder den Gesichtssinn betreffend“. Er beschreibt alles was über die Sehfähigkeit des Menschen aufgenommen und verarbeitet werden kann. Zu diesen Wahrnehmungen gehören nicht nur rein graphische Elemente, wie beispielsweise Bilder oder Piktogramme, sondern insbesondere auch die Schrift (siehe Abbildung 56).

```

schweigen schweigen schweigen
schweigen schweigen schweigen
schweigen           schweigen
schweigen schweigen schweigen
schweigen schweigen schweigen

```

Abbildung 56: Visuelle Dichtung im Gedicht »Schweigen« von Eugen Gomringer, 1969 (Schiffer 1998)

Auch nicht-graphische Programmiersprachen arbeiten mit visuellen Elementen, da es z.B. weitverbreitete Praxis in den Computerwissenschaften ist, Programmiercode nach bestimmten Maßgaben mittels farblicher und stilistischer Merkmalen zu formatieren. Als Beispiel hierfür können sämtliche Codepassagen in der vorliegenden Arbeit oder aber das folgende Beispiel in Code 17 dienen. Mit Hilfe einer solchen Formatierung fällt es dem Programmierer deutlich leichter, einzelne Elemente, wie z.B. Signalwörter, und deren Zusammenhang untereinander zu erkennen.

Code 17: Beispiel für eine Formatierung von Programmcode zur Erleichterung der Lesbarkeit

```

1  if (A == true)
2  {
3      A = false;
4  }
5  else if (A == false)
6  {
7      A = true;
8  }

```

Eine Antwort auf die Frage, warum Informationen in visueller Gestalt vom Menschen im Vergleich zu anderen Formen besser aufgenommen werden können, gibt die sogenannte Kognitionspsychologie. Mehrere Studien innerhalb dieser Wissenschaft belegen diesen Umstand und führen ihn auf eine Spezialisierung von Prozessen innerhalb des menschlichen Gehirns zurück. Es ist anerkannter Stand der Forschung, dass die linke Hemisphäre des Gehirns überwiegend für logisches und analytisches Denken verantwortlich ist, während in der rechten Gehirnhälfte vorwiegend gefühlsbezogene und ästhetische Prozesse sowie auch die Verarbeitung bildlicher und räumlicher Informationen ablaufen. Dabei können analytische Prozesse nur sequentiell, bildliche Prozesse hingegen parallel verarbeitet werden, wodurch sich ein höheres Potential ergibt. Dieses erklärt, warum es dem Menschen deutlich leichter fällt die Informationen visuell als rein textuell aufzunehmen (Schiffer 1998).

Obwohl dieses der VPL als positive Eigenschaft zu Gute kommt, gibt es auch kritische Stimmen, welche die Verwendung einer visuellen Sprache im Bereich der Softwareentwicklung grundsätzlich ablehnen. Kritiker dieser Methode betonen insbesondere, dass die Entwicklung von Programmen zu komplex sei, als dass diese alleine mit Hilfe graphischer Elemente visualisiert und verständlich gestaltet werden könne. Als Beispiel führt der Informatiker Tony Hoare zwei visuelle Systeme an, die sich durch nur eine kleine Erweiterung unterscheiden und gemäß einem mathematischen Beweis äquivalent sind (Schiffer 1998). Doch durch diese eine Erweiterung hat das System bereits deutlich an Komplexität und Größe gewonnen, was in Abbildung 57 verdeutlicht wird.

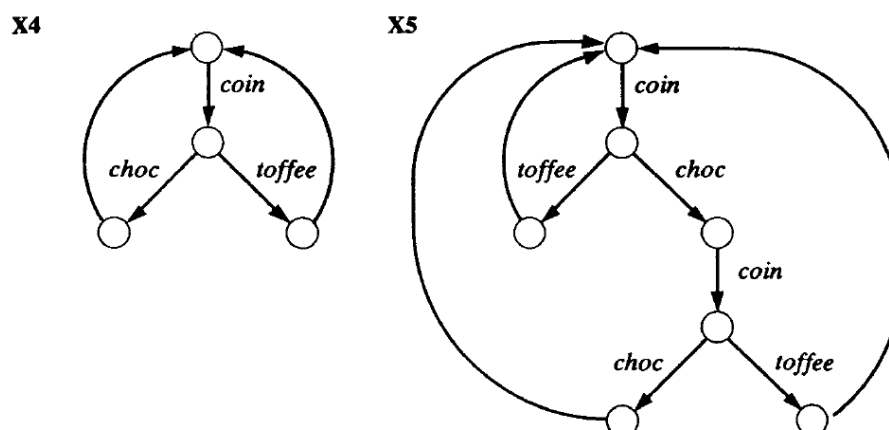


Abbildung 57: Veranschaulichung der Komplexität eines visuellen Systems nach Hoare (Schiffer 1998)

Eine ausführliche Diskussion über weitere Theorien der Kognitionspsychologie sowie zu Pro und Contra der visuellen Programmiersprachen führt Schiffer (1998) in seinem Buch *Visuelle Programmierung* aus.

Für den Begriff „visuell“ muss an dieser Stelle eine klare Unterscheidung getroffen werden, da dieser in den vergangenen Jahrzehnten von der Softwareindustrie in verschiedenen Bereichen mit unterschiedlicher Bedeutung verwendet wurde. Viele Programmiersprachen werden als visuelle Werkzeuge bezeichnet (z.B. *Visual Basic* und *Visual C++* von *Microsoft*), weil diese für den Bau von Benutzerschnittstellen eingesetzt werden können, nicht aber weil sie selbst graphische Programmiersprachen sind. Oftmals wird auf diese Unterscheidung sowohl in der Fachliteratur, als auch von den Herstellern kein großer Wert gelegt, so dass es zu Missverständnissen kommen kann. Daher wird im Folgenden der Begriff „visuell“ ausschließlich im Zusammenhang mit graphischen Programmiersprachen verwendet.

Zur Beschreibung der *VPL* muss nun noch der Begriff „Sprache“ definiert werden, welcher im Allgemeinen ein „System von Zeichen und Regeln, das einer Sprachgemeinschaft als Verständigungsmittel dient“ beschreibt (Scholze-Stubenrecht 2004). In diesem System können drei unterschiedliche Ebenen betrachtet werden (Schiffer 1998):

- *Syntax*:
die Verknüpfung von Zeichen, d.h. die Beziehung der Zeichen untereinander.
- *Semantik*:
die Bedeutung von Zeichen, d.h. die Beziehung der Zeichen zu den bezeichneten Dingen.
- *Pragmatik*:
die Wirkung von Zeichen, d.h. Beziehung der Zeichen zu den betroffenen Personen.

In den Computerwissenschaften wird die Sprache zumeist auf die beiden Ebenen Syntax und Semantik reduziert und als *formale Sprache* bezeichnet, welche eine mathematische Annäherung des Informationsgehaltes einer Aussage zum Ziel hat (vgl. auch Abschnitt 3.1.2 und 3.3.1). Diese Annäherung ist jedoch nur möglich, indem der Sprache eine syntaktische Struktur und jedem einzelnen Element eine definierte Bedeutung, wie z.B. eine Anweisung an eine Maschine, zugewiesen wird.

Mit der sprachlichen Form, Funktion und Gesetzmäßigkeit der Elemente einer Sprache beschäftigt sich die sogenannte *Grammatik*, welche für jede Sprache klar definiert sein muss und somit auch für die visuelle Sprache gilt (Scholze-Stubenrecht 2004). Ein Beispiel für die Implementierung einer solchen *Grammatik* ist das Compilergenerator-System *VLCC* (*Visual Language Compiler*), mit dessen Hilfe es möglich ist, alle Komponenten einer Grammatik für eine visuelle Sprache zu definieren (siehe Abbildung 58). In diesem können die verschiedenen vorhandenen Elemente, deren mögliche Zusammenhänge und vor allem deren Funktion genau definiert werden, um so den Rahmen für eine visuelle Sprache zu schaffen.

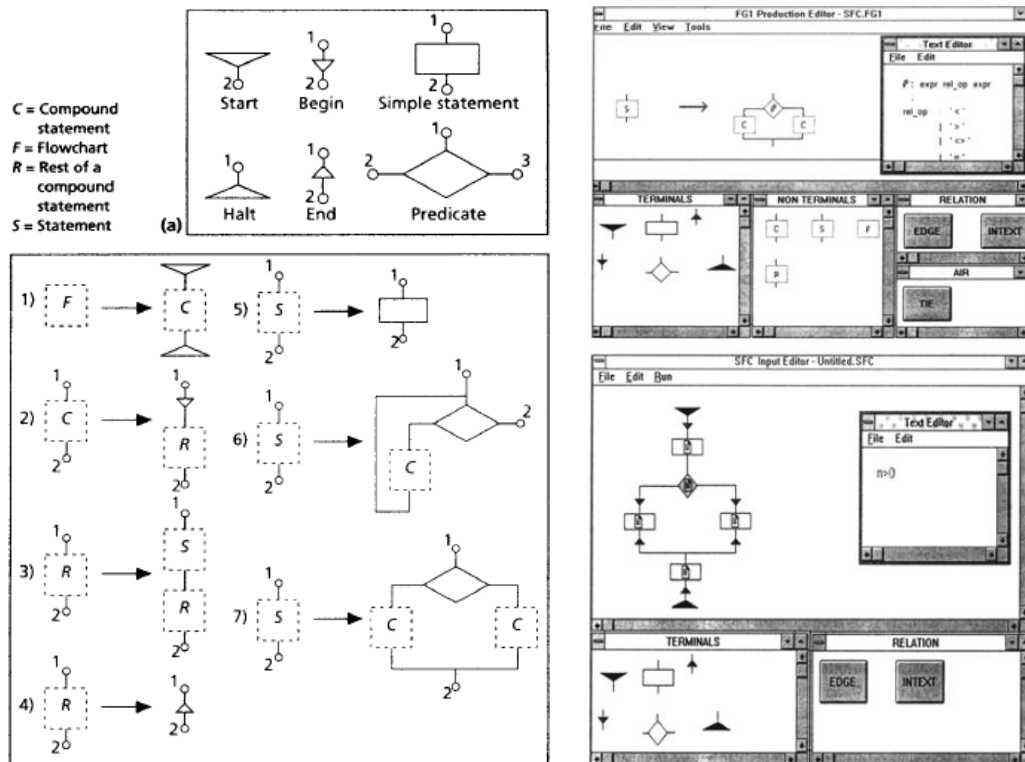


Abbildung 58:
 links: Grammatik: des VLCC
 rechts oben: Grammatikeditor des VLCC
 rechts unten: Diagrammeditor des VLCC
 (Schiffer 1998)

Schiffer (1998) nennt in seiner Definition der visuellen Sprache einen weiteren wichtigen Aspekt graphischer Informationssysteme. Sprache tritt unabhängig von der übermittelten Information in zwei unterschiedlichen Erscheinungsformen auf: der statischen und der dynamischen Zeichengebung.

Ein statisches Informationssystem, welches beispielsweise mit einer formalen, textuellen Sprache formuliert wurde, dient vorrangig der Fixierung von Wissen und arbeitet unabhängig von Raum und Zeit. Zwar erhält der Anwender für eine bestimmte Eingabe einen Ausgabewert, jedoch kann er mit den Elementen des Systems nicht interagieren. Das Senden und Empfangen von Informationen kann in diesem System zeitlich versetzt geschehen. Ein dynamisches Informationssystem hingegen ist offener gestaltet und reagiert auf äußere Einflüsse, wie z.B. ein Signal. Dieses System beinhaltet zwar ebenfalls Wissen, mit welchem es auf eine Eingabe antworten kann, jedoch lässt es selbst Interpretationsspielraum für den Anwender, welcher in direkte Interaktion mit dem System tritt. Das Ausmaß und die Freiheit dieser Interaktion zwischen System und Anwender werden ebenfalls in der Grammatik definiert, indem für die einzelnen Elemente der Sprache Variablen deklariert werden, welche in einem Informationssystem modifiziert werden können. Das Senden und Empfangen von Informationen in einem dynamischen System geschieht immer gleichzeitig.

Die vorangegangenen Ausführungen zeigen die Relevanz und Notwendigkeit der Definition einer Grammatik für die Anwendung einer visuellen Sprache im Bereich des *Automated Code Compliance Checking*.

4.2 Visual Code Checking Language

Im Rahmen der vorliegenden Arbeit soll gezeigt werden, dass mit Hilfe einer visuellen Sprache, wie sie in Abschnitt 4.1 vorgestellt wurde, die in Abschnitt 3.3 formulierten Anforderungen für ein *Automated Code Compliance Checking* erfüllt werden. Mittels der Vernetzung graphischer Elemente kann ein Verarbeitungsprozess so formuliert werden, dass eine beliebige Konformitätsüberprüfung von Regelwerk und Gebäudemodell beschrieben wird, ohne dabei an Übersichtlichkeit und Transparenz für den Anwender zu verlieren. Im Folgenden wird diese visuelle Sprache als *Visual Code Checking Language (VCCL)* bezeichnet.

4.2.1 Elemente der VCCL

Um der *VCCL* einen festen Rahmen zu geben, sollen die beiden graphischen Objekte **Knoten** und **Kante** verwendet werden. Auf dieser Basis lassen sich die folgenden grundlegenden Elemente der *VCCL* und deren graphische Darstellung definieren:

- *Objektknoten (graphische Darstellung: rechteckiger Knoten)*

Ein Objektknoten beschreibt einen Gegenstand, welcher in einem real existierenden System individuell und eindeutig identifiziert werden kann. Mit Hilfe eines solchen Knotens wird somit ein Objekt eines bestimmten Datentyps repräsentiert.

- *Operatorknoten (graphische Darstellung: rautenförmiger Knoten)*

Zur Beschreibung von Verarbeitungsprozessen zwischen den Objektknoten dienen die sogenannten Operatorknoten. Diese wenden eine wohldefinierte Operation auf eine festgelegte Anzahl von Eingangsgrößen, die sogenannten Operanden, an und erzeugen daraus ein entsprechendes Ergebnis.

- *Verknüpfung (graphische Darstellung: Kante)*

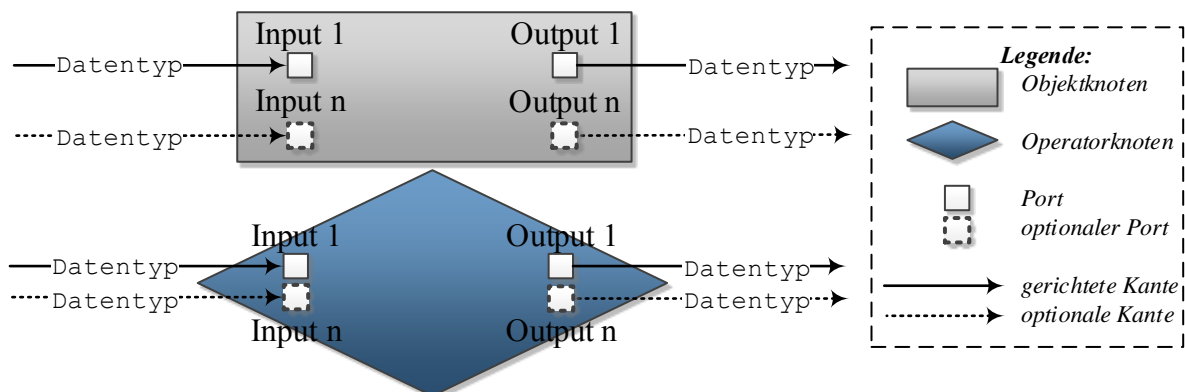
Grundlegend verknüpft eine Kante zwei Knoten der *VCCL* und bildet so das Verarbeitungssystem in Form eines Graphen ab. Eine Kante muss eine definierte Richtung besitzen, um das Gesamtsystem als eindeutig gerichtete Verarbeitungskette darzustellen. Die Verbindung der einzelnen Knoten der *VCCL* mit Hilfe der Kanten hat keine weiter definierte Funktion, sondern bestimmt ausschließlich, welche Knoten der *VCCL* miteinander in Interaktion treten.

- *Schnittstellen (graphische Darstellung: Input- und Output-Ports)*

Um die Verbindung zwischen den Knoten der *VCCL* so eindeutig und präzise wie möglich zu beschreiben, besitzen die Knoten sogenannte Schnittstellen, welche graphisch als Input- und Output-Ports innerhalb des jeweiligen Knotens dargestellt werden. Diese Schnittstellen definieren für das jeweilige Element der *VCCL*, welche Informationen über die Verknüpfung mittels der Kanten in das Element eingehen und anschließend weitergegeben werden können. Einem Knoten können unabhängig von seinem eigenen Informationsgehalt und seiner Funktion eine beliebige Anzahl solcher Schnittstellen zugeordnet werden. Damit ein Knoten in das System integriert werden kann, muss dieser jedoch mindestens eine Schnittstelle besitzen. In

Syntaxdiagramm 1 ist die graphische Darstellung der Schnittstellen als sogenannte Ports für Objekt- und Operatorknoten schematisch abgebildet.

Syntaxdiagramm 1: Schematische Darstellung der Ports für Daten- und Operatorknoten innerhalb der VCCL



Um den Informationsfluss der resultierenden Verarbeitungskette zu veranschaulichen, werden Input-Ports grundsätzlich auf der linken Seite, Output-Ports hingegen auf der rechten Seite eines VCCL-Knoten angetragen. Für jede einzelne Schnittstelle ist zudem eindeutig festgelegt, welcher Datentyp über diese übertragen werden kann. Somit ist die Übermittlung von Informationen in dem gesamten Verarbeitungssystem präzise definiert und eine Übertragung von Fehlinformationen zwischen den einzelnen Knoten ausgeschlossen. Wie in Syntaxdiagramm 1 dargestellt wird der jeweils festgelegte Datentyp der Schnittstelle zur Veranschaulichung an der verknüpften Kante angegeben. Da hierdurch jede einzelne Schnittstelle eindeutig gekennzeichnet wird, kann die Bezeichnung der einzelnen Schnittstellen, wie diese zur Veranschaulichung in Syntaxdiagramm 1 dargestellt ist, in den folgenden Diagrammen entfallen.

In den folgenden Unterkapiteln sollen nun die einzelnen Elemente der VCCL und deren Funktionsweise im Detail erläutert werden.

4.2.1.1 Objektknoten

Prinzipiell können innerhalb der *VCCL* beliebige Datentypen deklariert werden, welche anschließend für die Definition der Objektknoten verwendet werden können. Im Rahmen dieser grundlegenden konzeptionellen Vorstellung der *VCCL* sollen zunächst ausgewählte Datentypen verwendet werden, welche in Abbildung 59 dargestellt sind.

Datentyp-Name	Beispiel-Werte	Beschreibung
Basistypen		
Bool	True / False	Wahrheitswert
Float	1,2345	Fließkommazahl
String	„Beispiel“	Zeichenkette
GUID	12-34-56	Globally Unique Identifier
Gebäudemodell		
BuildingElement	BuildingElement1	Überklasse für Bauteile des Gebäudemodells
Wall	Wall1	Wandbauteil
Slab	Slab1	Deckenbauteil
Room	Room1	Raumobjekt
Opening	Opening1	Öffnung
Sonderfall		
List<Datentyp>	{BuildingElement1, BuildingElement2}	Liste von Bauteilen

Abbildung 59: Datentypen der *VCCL*

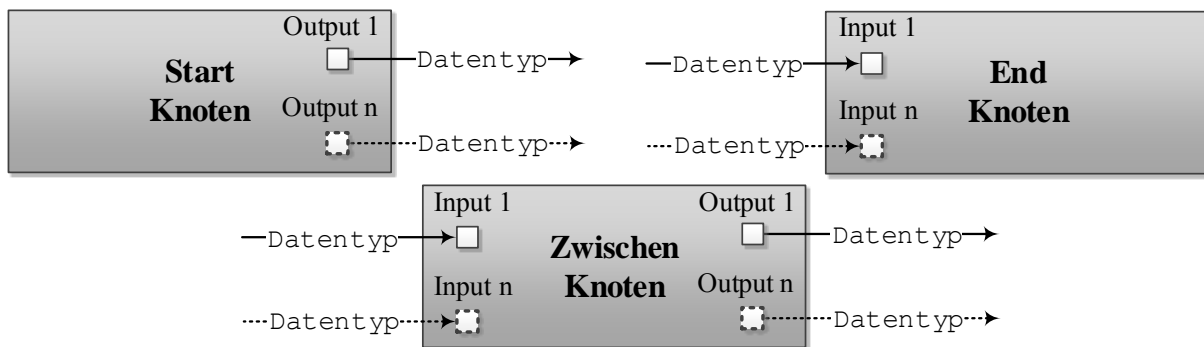
Zur Kennzeichnung des Informationsgehalts besitzt ein Objektknoten eine Beschreibung, welche den Knoten mit Hilfe eines sogenannten Labels kennzeichnet. Diese Kennzeichnung muss grundlegend den Datentyp des Objektknotens beinhalten und erlaubt überdies eine zusätzliche Kurzbeschreibung. Diese Beschreibung kann beispielsweise auch dafür verwendet werden, den Inhalt des Knotens anzugeben. In Syntaxdiagramm 2 ist ein Objektknoten schematisch dargestellt.

Syntaxdiagramm 2: Schematische Darstellung eines Objektknoten



Durch die Definition der sogenannten Input- und Output-Schnittstellen wird für den Objektknoten festgelegt, welche Funktion dieser innerhalb des Verarbeitungssystems einnehmen kann. Ein Objektknoten kann als Start- bzw. Quellknoten (nur Output-Schnittstellen), als Zwischenknoten (Output und Input-Schnittstellen) oder aber als End- bzw. Ergebnisknoten (nur Input-Schnittstellen) fungieren. Die verschiedenen Erscheinungsformen sind in Syntaxdiagramm 3 schematisch dargestellt.

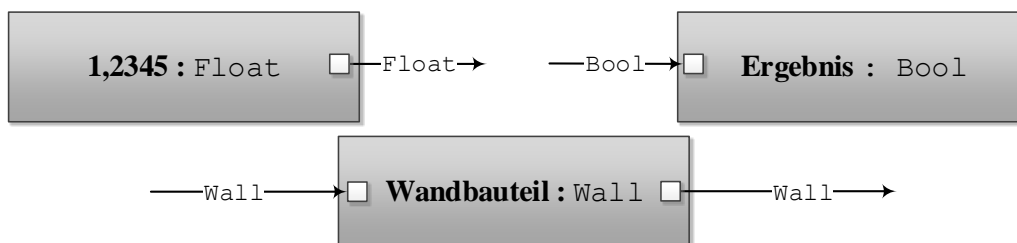
Syntaxdiagramm 3: Schematische Darstellung der unterschiedlichen Erscheinungsformen eines Objektknoten in Abhängigkeit der Schnittstellen



Ein Start-Knoten dient als Informationsquelle für alle mit ihm verknüpften Knoten und muss daher bei der Erstellung innerhalb der *VCCL* direkt mit einem konkreten Wert des jeweiligen Datentyps belegt werden. Dieses kann programmiertechnisch beispielsweise durch eine direkte Nutzereingabe oder einen Ladeprozess umgesetzt werden. Die beiden weiteren Erscheinungsformen, Zwischen- und Endknoten, gliedern sich als verknüpfte Elemente einer Prozedur in eine Verarbeitungskette ein und werden durch den Informationsfluss des Verarbeitungssystems mit Informationen besetzt. Daher werden diese bei der Erstellung nicht mit Werten besetzt.

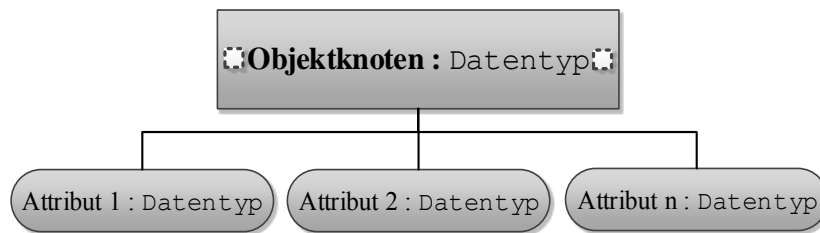
Zur Verdeutlichung sind die verschiedenen Erscheinungsformen in Syntaxdiagramm 4 als beispielhafte Objektknoten dargestellt.

Syntaxdiagramm 4: Beispiele für die unterschiedlichen Erscheinungsformen eines Objektknoten in Abhängigkeit der Schnittstellen

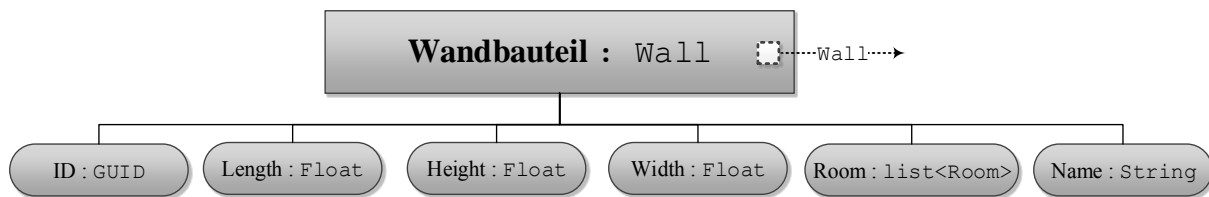


Wie die in Abbildung 59 aufgeführten Datentypen der *VCCL* zeigen, können Objektknoten nicht nur atomare, sondern mehrere Informationen zu einem einzelnen Objekt repräsentieren. Beispielsweise besitzt ein Objekt des Gebäudemodell-spezifischen Datentyps `Wall` mehrere Attribute, die das einzelne Objekt eindeutig beschreiben und innerhalb des Objektknotens gespeichert werden. Zur Veranschaulichung dieser Informationen können innerhalb der *VCCL* sogenannte Attributknoten verwendet werden, welche als ellipsenförmige Knoten über nicht-gerichtete Kanten und ohne Schnittstellen mit dem Objektknoten verbunden sind. Diese Knoten dienen ausschließlich der Veranschaulichung des Informationsgehalts eines Objektknotens und können daher auch entfallen. Jedem Attributknoten muss ein definierter Datentyp (siehe Abbildung 59) zugeordnet werden, damit die beinhaltete Information eindeutig beschrieben wird. Die Darstellungsweise solcher Attributknoten ist in Syntaxdiagramm 5 schematisch dargestellt.

Syntaxdiagramm 5: Schematische Darstellung eines Objektknoten mit den zugehörigen Attributknoten



Ein konkretes Beispiel für die Darstellung der Attributknoten ist in Syntaxdiagramm 6 für einen Objektknoten des Datentyps `Wall` dargestellt.

Syntaxdiagramm 6: Darstellung des Objektknoten vom Datentyp `Wall` und ausgewählter Attributknoten

Durch die definierten Datentypen der einzelnen Attributknoten sind unter anderem auch direkte Rückschlüsse von dem Objektknoten auf das Gebäudedatenmodell möglich. Im dargestellten Beispiel ist eine solche Verknüpfung mit Hilfe des Attributs *Room* umgesetzt, bei der es sich um eine Klasse des Gebäudedatenmodells handelt. Über dieses Attribut kann für das Objekt *Wandbauteil* ein direkter Rückschluss auf dessen Position innerhalb des räumlichen Modells gezogen werden. Da ein Wandbauteil den Bezug zu mehreren Räumen haben kann, ist dem Attribut *Room* als Datentyp eine Liste `list<Room>` zugeordnet, welche mehrere Objekte des Datentyps `Room` beinhalten kann.

Damit die Übersicht in den nachfolgenden Kapiteln gewahrt bleibt, werden Attributknoten von Datenobjekten nur dann dargestellt, wenn sich der Anwendungsfall explizit auf diese bezieht.

Wie bereits in dem Beispiel in Syntaxdiagramm 6 zu sehen ist, kann ein Objektknoten eine Sammlung von Objekten repräsentieren, indem diesem als Datentyp eine Liste zugeordnet wird. Eine solche Liste kann mit einem beliebigen Datentyp der *VCCL* verwendet werden und wird nur zur Veranschaulichung des Konzepts der *VCCL* innerhalb der vorliegenden Arbeit ebenfalls gesondert dargestellt. In Syntaxdiagramm 7 ist die schematische Darstellung einer solchen Liste abgebildet. Wie bereits bei den Attributknoten gilt, dass dieses ausschließlich der Veranschaulichung des Informationsgehalts eines Objektknotens dient und daher nicht zwingend dargestellt sein muss.

Syntaxdiagramm 7: Veranschaulichung eines Objektknotens vom Datentyp Liste



Da durch die Definition des Objektknotens bereits ein Datentyp für die Elemente der Liste vorgegeben ist, muss der Typ nicht mehr gesondert für jedes einzelne Element dargestellt werden, sondern eine einfache Kurzbeschreibung der einzelnen Listenobjekte ist ausreichend. Ein konkretes Beispiel für einen solchen Listen-Objektknoten ist in Syntaxdiagramm 8 dargestellt.

Syntaxdiagramm 8: Beispiel eines Objektknotens vom Datentyp list<Wall>



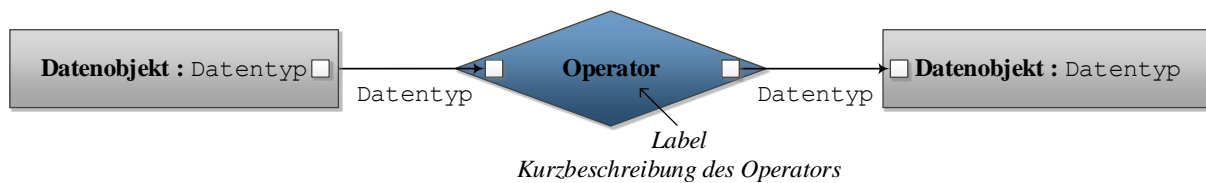
Durch die Definition eines Objektknotens als Repräsentant eines Datentyps können eine Vielzahl von Gegenständen, welche in einem real existierenden System eindeutig identifiziert werden können, dargestellt werden. Mögliche weitere Einsatzbereiche von Objektknoten können beispielsweise sein:

- Räumliche Objekte: Abschnitt, Geschoss, Teilgebiet, ...
- Zeitliche Objekte: Projektstart, Termin, Meilenstein, ...
- Projektspezifische Objekte: Auftraggeber, Projektbeteiligter, ...
- Wirtschaftliche Objekte: Kosten, Aufwandswert, ...

4.2.1.2 Operatorknoten

Ein Operatorknoten beschreibt eine definierte Operation, die die über die Schnittstellen verknüpften Eingangsinformationen zu einem Ergebnis verarbeitet und die anschließend über die Ausgangsschnittstellen ausgegeben wird. Daher ist ein solcher Operatorknoten in der VCCL als zwischenstehender Verarbeitungsprozess ausgebildet und muss mindestens eine Eingangs- und eine Ausgangsschnittstelle besitzen, da er sonst nicht in die Verarbeitungskette integriert werden kann. Wie bereits bei den Objektknoten erklärt, kann auch dem Operatorknoten durch ein Label eine Kurzbeschreibung hinzugefügt werden. In Syntaxdiagramm 9 ist eine schematische Darstellung der Funktionsweise eines Operatorknotens in der VCCL dargestellt.

Syntaxdiagramm 9: Schematische Darstellung eines Operatorknoten



Im Prinzip repräsentiert ein Operatorobjekt eine Funktion, welche die über die Inputschnittstellen erhaltenen Informationen zu den Ausgabedaten verarbeitet. In der programmiertechnischen Umsetzung hat dieses Vorgehen den praktischen Vorteil, dass der Inhalt einer solchen Operation, wie beispielhaft in Code 18 aufgeführt, sehr einfach formuliert werden kann.

Code 18: Beispielhafte Verarbeitungsfunktion eines Operators in Pseudocode

```

1 // Pass over
2 Process (Input inputObject)
3 {
4     Object outputObject = inputObject;
5     return outputObject;
6 }
  
```

Da auf diese Weise auch logische Operatoren abgebildet werden können, ist die VCCL in der Lage, alle logischen Systemen, wie diese insbesondere in den Abschnitten 3.1.1 und 3.1.1.3 vorgestellt wurden, zu verwenden.

Mögliche Einsatzbereiche von Operatorknoten können sein:

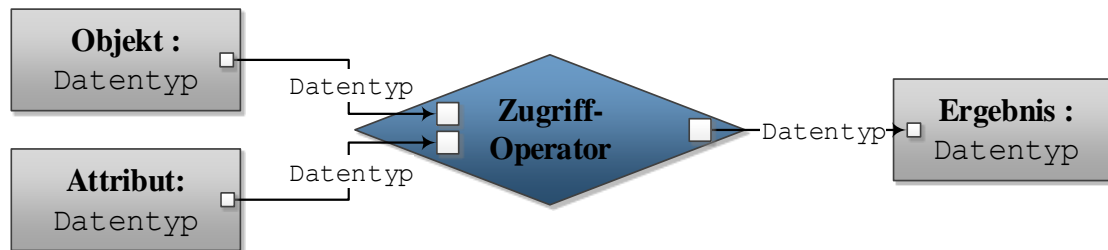
- Logische Operatoren (Aussagen-, Prädikaten- und deontische Logik)
- Operatoren zum Datenzugriff, z.B. Zugriff auf Attribute von Datenobjekten
- Filter-Operatoren
- Verarbeitende Operatoren z.B. Berechnung von (fehlenden) Attributen
- Vergleich-Operatoren
- Geometrische Operatoren
- Operatoren zur Überprüfung der Einhaltung von Randbedingungen

Die Umsetzung und Anwendung solcher Operatoren soll nun an Hand von einigen Beispielen demonstriert werden:

– Datenzugriff

Ein Datenzugriff ist in der VCCL als verarbeitender Prozess definiert, da einem Operator-knoten mit Hilfe von verknüpften Objektknoten sämtliche Informationen übergeben werden können, um einen präzisen Zugriff auf die Daten eines Objektknotens zu beschreiben. In Syntaxdiagramm 10 ist der schematische Ablauf eines solchen Zugriffs dargestellt.

Syntaxdiagramm 10: Schematische Darstellung einer Zugriffsoperation



Über die beiden Input-Werte *Objekt* und *Attribut* sind sämtliche notwendigen Informationen gegeben, welche für den Datenzugriff erforderlich sind. Hierzu muss das *Attribut* über seinen Wert lediglich die Information des *Objekts*, auf welche zugegriffen werden soll, eindeutig beschreiben. Prinzipiell ist es in der VCCL freigestellt, wie das *Attribut* diese Information beschreibt und ist daher nicht an einen definierten Datentyp gebunden.

An dieser Stelle müssen zwei Arten der Zugriffsoperation unterschieden werden, der sogenannte Lese- (*Get*) und der Schreibe-Zugriff (*Set*). Je nachdem, ob eine Information eines Datenobjekts gelesen oder aber überschrieben werden soll, muss der jeweilige Operator gewählt werden.

Eine Ausformulierung der schematischen Operation in Syntaxdiagramm 10 als Lesezugriff ist in Code 19 dargestellt.

Code 19: Funktion eines Lese-Zugriffsoperators in Pseudocode

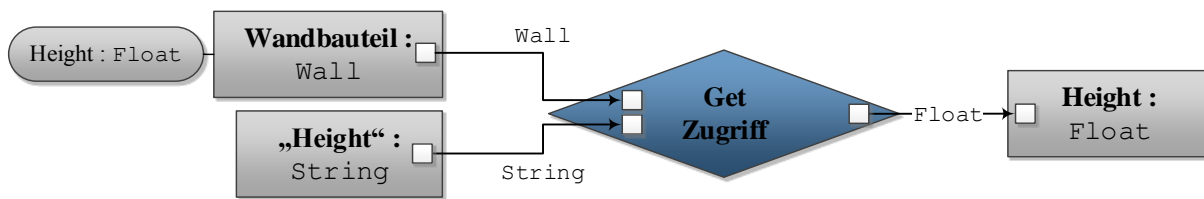
```

1 // Data Access
2 Process (Input Objekt, Input Attribut)
3 {
4     if (Objekt.Attribut != NULL)
5         return Objekt.Attribut;
6     else (Objekt.Attribut == NULL)
7         return NULL;
8 }

```

Ein konkretes Beispiel für einen Lese-Zugriff auf die Daten eines Objektknotens ist in Syntaxdiagramm 11 dargestellt. In diesem Fall stellt das Objekt *Attribut* einen *String*-Wert dar, welcher eindeutig das Attribut *Height* des Datenobjekts *Wandbauteil* über den Namen des Attributs identifiziert. Mit diesen Informationen ist der Zugriffsoperator in der Lage, auf das Attribut zuzugreifen und kann dieses anschließend über die Output-Schnittstelle mit dem entsprechenden Datentypen wieder ausgeben.

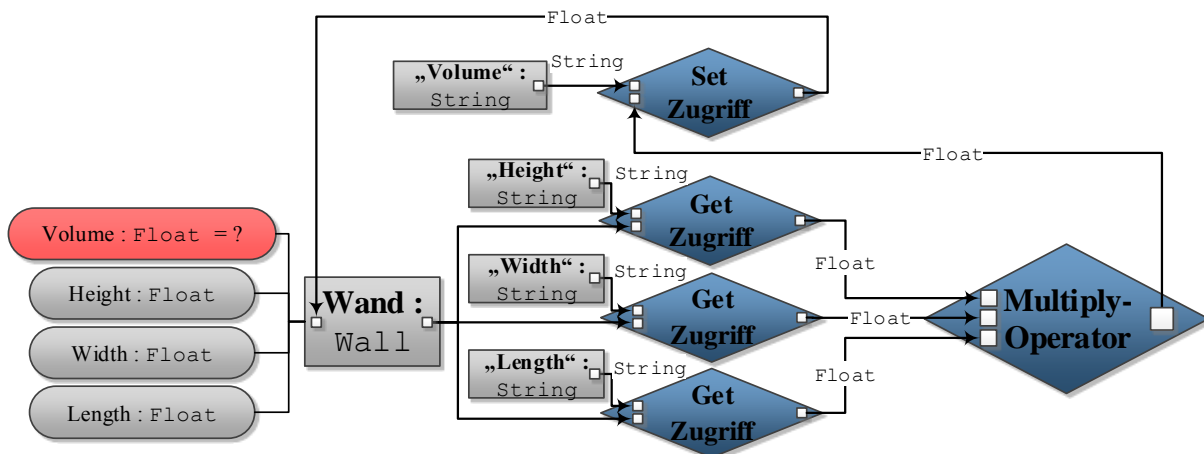
Syntaxdiagramm 11: Schematische Darstellung einer Zugriffsoperation auf das Attribut eines Datenobjektes



– Berechnung eines fehlenden Attributs

Sollte ein Objektknoten eines Gebäudemodell-spezifischen Datentyps eine gewünschte Information nicht enthalten, ist es hilfreich, sich diese aus dem Datenmodell oder dem Objektknoten selbst abzuleiten. Auch diese Aufgabe kann ein Operator-knoten übernehmen, indem er auf die vorhandenen Informationen zugreift und diese weiterverarbeitet. In dem Beispiel in Syntaxdiagramm 12 wird zunächst auf die Attribute *Height*, *Width* und *Length* des Objektknoten *Wand* zugegriffen, um diese anschließend in einer Multiplikations-Operation zu dem nicht besetzten Attribut *Volume* weiter zu verarbeiten.

Syntaxdiagramm 12: Beispiel zu der Berechnung eines fehlenden Attributes eines Objektknotens



Die Funktion, welche die Operanden innerhalb des Multiplikations-Operators weiterverarbeitet, ist allgemeingültig gehalten und in Code 20 dargestellt.

Code 20: Funktion des Multiplikations-Knoten in Pseudocode

```

1 // Multiply
2 Process (Input a, Input b, Input c)
3 {
4     return a * b * c;
5 }

```

Nach dem gleichen Prinzip des Multiplikations-Operators lassen sich sämtliche weitere arithmetische Operationen innerhalb eines *VCCL*-Operator-knoten abbilden.

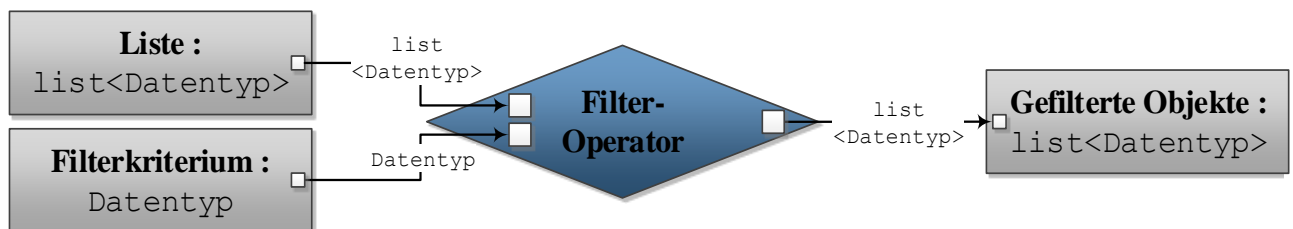
– Filteroperation

Über Operator-knoten können auch Informationen eines Objektknoten gefiltert und somit für die anschließende Verarbeitung aufbereitet werden. In diesem Falle fungiert ein sogenanntes

Filterkriterium dazu, eine Maßgabe zu definieren, nach welcher in dem Filteroperator schließlich die Informationen eines Objekts gefiltert werden können. In der programmiertechnischen Umsetzung stellt das Kriterium einen Ausdruck (Prädikat) dar, dessen Auswertung in dem Verarbeitungsprozess einen Booleschen Wert (wahr/falsch) ergibt, nach welchem anschließend die Objekte sortiert werden können.

Dabei ist zu beachten, dass eine Filteroperation in der Regel auf Objektknoten angewandt wird, welchen eine Liste von Objekten als Datentyp zugeordnet ist. Der schematische Ablauf dieser Operation ist in Syntaxdiagramm 13 und der zugehörige Pseudocode der Funktion in Code 21 dargestellt.

Syntaxdiagramm 13: Schematische Darstellung eines Operatorobjekts



Code 21: Funktion des Filteroperators in Pseudocode

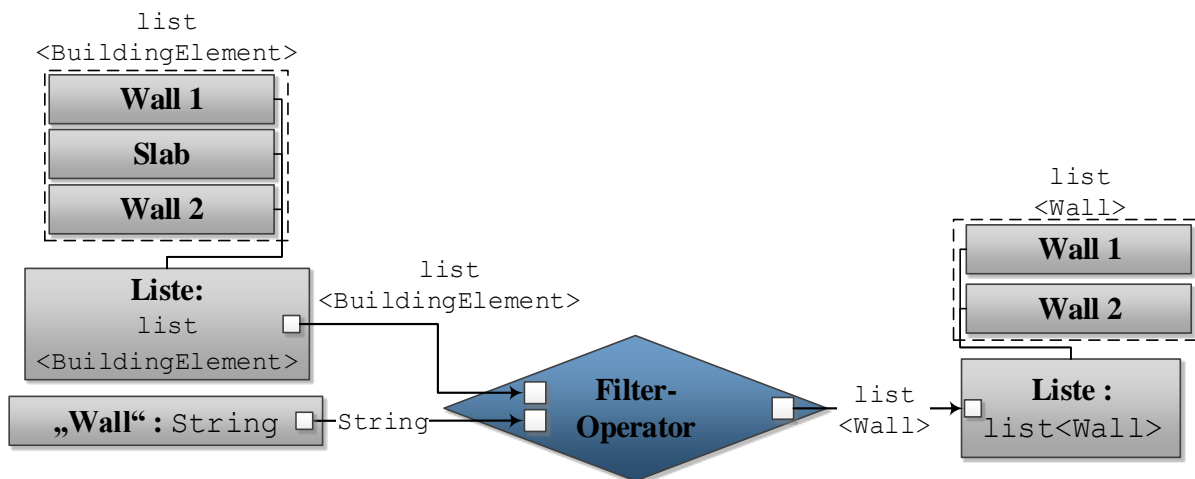
```

1 // List Filter
2 Process (Input list<Datentyp>, Input Filterkriterium)
3 {
4     Output outputList;
5     foreach(item in list<Datentyp>)
6     {
7         if (item.Filterkriterium == Filterkriterium)
8         {
9             outputList.Add(item);
10        }
11    }
12    return outputList;
13 }

```

Ein konkretes Beispiel für eine solche Filteroperation ist in Syntaxdiagramm 14 dargestellt. In diesem Falle wird ein Objektknoten mit einer Liste von Bauteilen nach dem jeweiligen Datentyp gefiltert und alle Elemente des Datentyps `Wall` werden ausgegeben. Das Filterkriterium wird in diesem Fall durch einen einfachen `String`-Wert definiert, welcher den Datentyp `Wall` eindeutig beschreibt.

Syntaxdiagramm 14: Schematische Darstellung einer Filteroperation nach dem Datentyp Wall



Nach dem gleichen Prinzip können innerhalb der Filter-Operation auch andere Kriterien, wie z.B. geometrische Attribute eines Objektes als `Float`-Wert, in die Filteroperation eingehen und für Vergleichsoperationen innerhalb der Verarbeitungsfunktion verwendet werden.

– Geometrische Operatoren

Da sehr viele Regelungen im Bauwesen geometrisch oder an dem Raummodell orientiert sind, müssen auch diese Informationen innerhalb der *VCCL* abgebildet werden. An dieser Stelle können ebenfalls Operatorknoten genutzt werden, um diese Prozesse zu beschreiben. Grundlage für diese Operationen sind die geometrischen Informationen, welche in dem Gebäudemodell zur Verfügung stehen. Diese können über einen Datenzugriff abgerufen und in dem Operator über geometrische Algorithmen weiterverarbeitet werden, so dass am Ende die gewünschte Information ausgegeben werden kann.

Generell haben die geometrischen Operationen zum Ziel, das räumliche Verhältnis zwischen zwei oder mehreren geometrischen Objekten zu überprüfen. Im dreidimensionalen Raum gibt es eine Vielzahl von Möglichkeiten, wie zwei Objekte in einem räumlichen Verhältnis zueinander stehen können. In Abbildung 60 ist schematisch eine Auswahl von topologischen Beziehungen dargestellt.

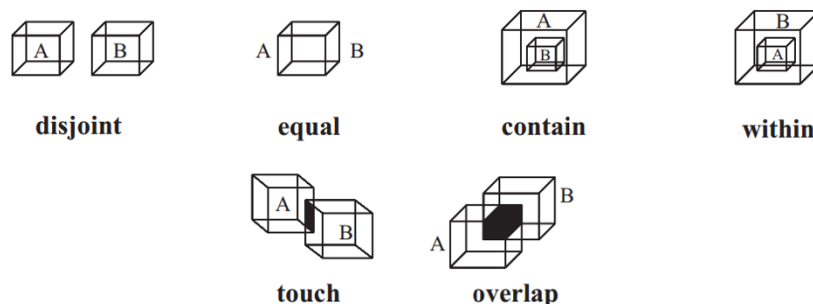
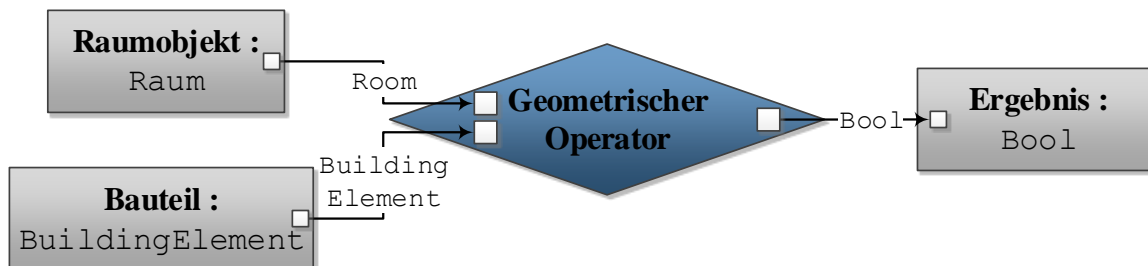


Abbildung 60: Schematische Darstellung einiger geometrischer Verhältnisse zwischen zwei geometrischen Objekten (Paul und Borrman 2008)

Mittels der *VCCL*-Operatoren ist es möglich, diese geometrischen Verhältnisse zu erfassen, was an dieser Stelle durch ein Beispiel zu räumlichen Nachbarschaftsbeziehungen von Bauteilen aufgezeigt werden soll.

In vielen Datenmodellen sind zwar grundlegende Informationen zu der Geometrie eines Objekts enthalten (z.B. Höhe, Breite, Tiefe), jedoch wird in der Regel keine Aussage darüber getroffen, wie diese räumlich zueinander im Verhältnis stehen. An dieser Stelle ist es nötig zu überprüfen, wie dieses Verhältnis konkret aussieht. Ein beispielhafter geometrischer Überprüfungsprozess ist in Syntaxdiagramm 15 mit Hilfe der *VCCL* dargestellt.

Syntaxdiagramm 15: Schematische Darstellung eines geometrischen Operators in VCCL



In dem vorliegenden Beispiel soll das topologische Verhältnis zwischen einem Raumobjekt und einem beliebigen Bauteil des Typus `BuildingElement` überprüft werden. Hierzu werden zunächst die beiden Objekte selbst über die Input-Schnittstellen an den geometrischen Operator übergeben.

An dieser Stelle kann die Annahme getroffen werden, dass beide Objekte, *Raumobjekt* und *Bauteil*, über geometrische Informationen, z.B. in Form von *Boundary-Representation*-Daten (*BREP*), verfügen und dass auf diese innerhalb des geometrischen Operators zugegriffen werden kann. Auf Basis dieser Informationen kann innerhalb des Operators berechnet werden, welches räumliche Verhältnis zwischen den beiden Objekten vorliegt. Grundlage für solche Berechnungen können geometrische Bibliotheken sein, wie diese auch bereits bei den *FORNAX*-Objekten verwendet wurden (siehe Abschnitt 3.1.4.1).

Im vorliegenden Beispiel wird in Code 21 überprüft, ob sich die beiden Objekte schneiden oder aber berühren. Das Ergebnis der Berechnung wird als `Bool`-Wert ausgegeben.

Code 22: Funktion des geometrischen Operators in Pseudocode

```

1 // Geom. Operation
2 Process (Input Raum, Input Bauteil)
3 {
4     if (Intersects(Raum.BREP, Bauteil.BREP) == TRUE)
5     {
6         return TRUE;
7     }
8     else if (Touching(Raum.BREP, Bauteil.BREP) == TRUE)
9     {
10        return TRUE;
11    }
12    else
13    {
14        return FALSE;
15    }
16 }

```

Auf gleiche Art und Weise können weitere geometrische Operatoren formuliert werden, um so schließlich verschiedene geometrische Verhältnisse zu prüfen.

4.2.1.3 Weitere Elemente der VCCL

Zwar lassen sich mit Hilfe der bereits vorgestellten Elemente der *VCCL* viele Informationen und Gegenstände einer Vorschrift im Bauwesen abbilden, jedoch gibt es weitere Darstellungsweisen von Informationen in gebräuchlichen Normen, welche sich nur sehr schwer auf dieser Grundlage beschreiben lassen. Für diese Fälle können innerhalb der *VCCL* spezielle Knoten definiert werden, welche sich insbesondere für die Speicherung des Informationsgehalts dieser Darstellungsweisen anbieten.

Als Beispiel können an dieser Stelle Datentabellen genannt werden. Für diesen Anwendungsfall wurde in der *VCCL* ein eigener Knotentyp definiert, welcher geeignet ist, mit Hilfe konkreter Inputdaten auf tabellarische Daten zuzugreifen und anschließend den korrekten Antwortwert über die Ausgabeschnittstelle weiterzugeben.

Syntaxdiagramm 16: Schematische Darstellung eines Datentabellen-Knoten



Da solche Datentabellen sehr häufig in Normen des Bauwesens auftreten, ist es sinnvoll diesen Knoten möglichst allgemeingültig zu konzipieren, so dass dieser in der Lage ist, viele Arten von Tabellen darzustellen. Entsprechende konkrete Beispiele für die Umsetzung solcher Datentabellenknoten sind in den Abschnitten 4.3 und 5.2 beschrieben.

4.2.2 Grundsätze und Eigenschaften der VCCL

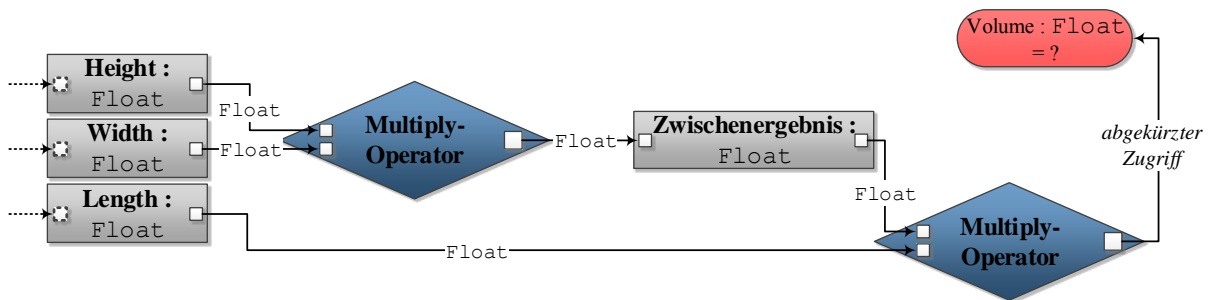
Mittels der bereits eingeführten Knotentypen können Datenzugriff, -filter und Analysefunktionen ausreichend formuliert werden. Jedoch sollen an dieser Stelle Grundsätze für die *VCCL* formuliert werden, welche der Syntax und Grammatik einen konkreten Rahmen setzen und so zu der Konsistenz und Fehlerfreiheit des gesamten Prozesses beitragen.

Die Verwendung der Operatorknoten birgt aufgrund der hohen Flexibilität und der Möglichkeit zur freien Formulierung von Verarbeitungsprozessen die Gefahr, dass große Teile des Prozesses innerhalb einer einzigen Operation und somit viele spezielle Knoten für Einzelfälle definiert werden. Dieses ist jedoch nicht sinnvoll, da es sich bei Knotentypen immer um allgemeingültige Datenobjekte handeln sollte, welche für alle Regelwerke und Vorschriften eingesetzt werden können. Eine Ausnahme bilden hier lediglich die Knotentypen, welche für eine spezielle Darstellungsweise von Informationen in Vorschriften, z. B. Datentabellen, deklariert werden dürfen.

Daher gilt für alle Knotentypen der *VCCL* der **Grundsatz der Generizität** (Allgemeingültigkeit), also einer möglichst universellen Anwendungsmöglichkeit jedes einzelnen Knoten innerhalb eines Verarbeitungssystems, ab. Damit jedoch gleichzeitig jeder gewünschte Prozess vom Anwender abgebildet werden kann, gilt über dies für alle Knotentypen der **Grundsatz der feinsten Granularität**, also einer maximal möglichen Zerlegbarkeit des Prozesses in die Einzelemente und -schritte.

Als Beispiel für diese beiden Grundsätze kann der Multiply-Operator in Syntaxdiagramm 12 dienen, welcher drei Eingabewerte miteinander multipliziert. Hierbei handelt es sich um einen speziell für den vorliegenden Fall definierten Knoten, denn dieser Prozess kann, wie in Syntaxdiagramm 17 dargestellt, in zwei einzelne Schritte zerlegt werden. Die vorausgehenden Datenzugriffe auf die Attribute der Objektknoten, wie diese im vorhergehenden Beispiel dargestellt wurden, sind an dieser Stelle nicht zusätzlich abgebildet.

Syntaxdiagramm 17: Darstellung einer Verarbeitungsfunktion nach dem Grundsatz der feinsten Granularität

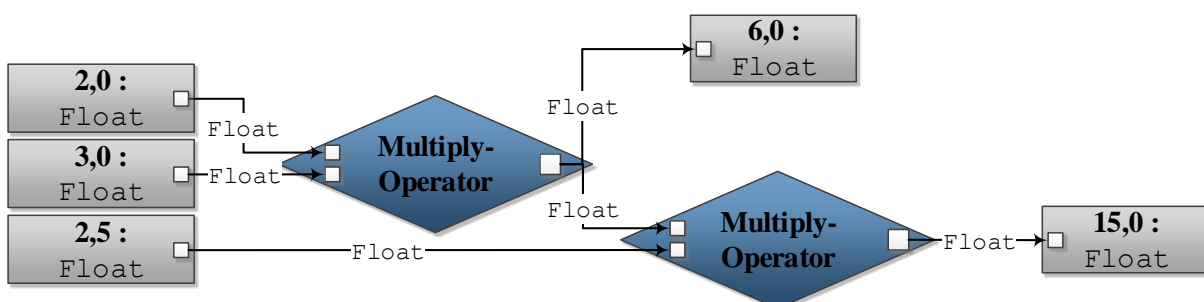


Zwar geht dieses auf den ersten Blick zu Lasten der Übersichtlichkeit, jedoch wird in der vorliegenden Arbeit in erster Linie darauf geachtet, dass die Abbildbarkeit jedes möglichen Prozesses gegeben ist. Des Weiteren ist es für den Nutzer übersichtlicher, wenn es nur eine begrenzte Anzahl an Knotentypen und nicht für jeden Spezialfall ein eigenes Element der VCCL gibt.

Als weitere Voraussetzung gilt innerhalb der VCCL der **Grundsatz der maximalen Flexibilität**, welche dem Anwender durch die Struktur der Knoten zur Verfügung stehen muss. Gemäß den Anforderungen aus Abschnitt 3.3 muss der Nutzer an jeder geeigneten Stelle in der Lage sein, Stichproben vorzunehmen, damit der Überprüfungsprozess seine Transparenz bewahrt. Durch die feine Granularität der Elemente des Verarbeitungssystems und der allgemeingültig bzw. generisch formulierten Verarbeitungsprozesse ergibt sich eine maximale Freiheit des Nutzers bei der Formulierung, was insbesondere die Überprüfungsprozesse mit einschließt.

Wie in Syntaxdiagramm 17 dargestellt, kann eine solche Stichprobe innerhalb der VCCL sehr einfach mit einem End-Objekt-knoten (vgl. Syntaxdiagramm 3) dargestellt werden, welcher ein Zwischenergebnis veranschaulicht. So kann der beinhaltete Wert eines solchen Knotens beispielsweise über das Label des Knotens angezeigt werden. Auf diese Weise wird dem Anwender eine konkrete Einsicht in den Informationsfluss der Verarbeitungsprozedur gegeben. Ein Beispiel einer solchen Stichprobe ist in Syntaxdiagramm 18 dargestellt.

Syntaxdiagramm 18: Veranschaulichung von Zwischenschritten durch Objekt-knoten



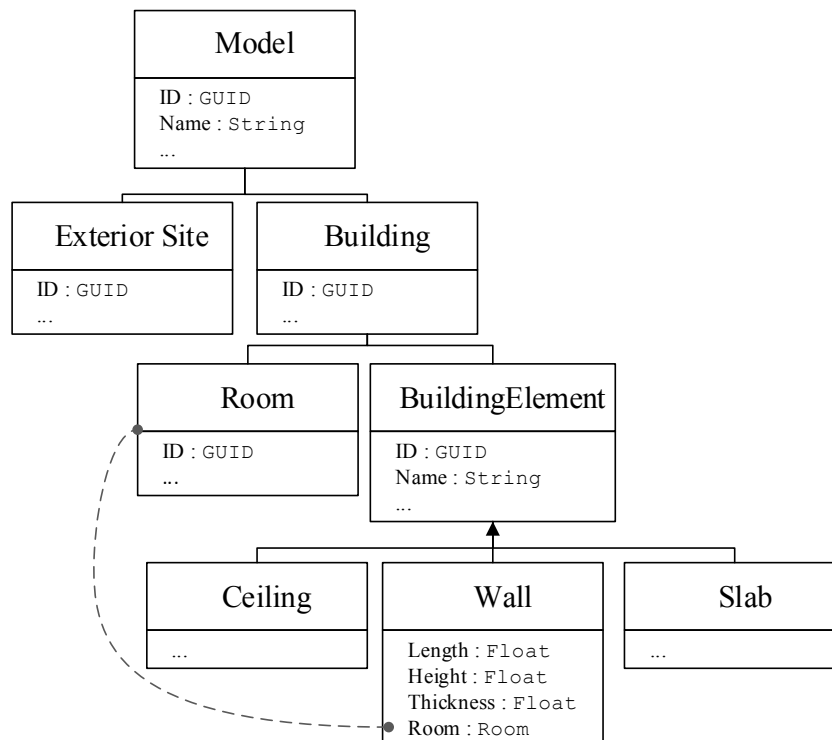
Wie bereits in Abschnitt 4.1.2 beschrieben, basiert die visuelle Sprache auf der dynamischen Zeichengebung, welche auf äußere Einflüsse, wie z.B. eine Modifikation oder Manipulation des Anwenders, direkt reagieren kann. Diese Eigenschaft unterstützt den Anwender bei dem Umgang mit der *VCCL*, da jeder einzelne Prozessschritt mit seinem aktuellen Informationsgehalt eingesehen werden kann. Im Gegensatz zu einer geschlossenen Verarbeitungsmethode, handelt es sich hierbei also um ein **dynamisches System und Vorgehen**. Dieses kann insbesondere verstärkt und unterstützt werden, wenn die Informationen eines Knotens auch visuell für den Anwender aufbereitet werden, so dass dieser direkt das Ergebnis seiner vorangegangenen Manipulationen einsehen kann.

4.2.3 VCCL und das Gebäudemodell

Bei der Festlegung der Knotentypen der *VCCL* spielt die Struktur und Taxonomie des zu Grunde liegenden Gebäudemodells eine wesentliche Rolle, da die Verarbeitungsprozesse der visuellen Sprache eng mit den Daten des Modells verknüpft sind. Ein Datenzugriff kann nur so einfach gestaltet werden, wie es das Datenmodell zulässt. Daher ist es sinnvoll, sich bei der Festlegung von Knoten- und Datentyp an der Struktur des Gebäudemodells zu orientieren (vgl. Abschnitt 4.2.1.1).

In der vorliegenden Arbeit soll ein Datenmodell mit einer einfachen Struktur als theoretische Grundlage für die *VCCL* dienen, da es zunächst vorrangig um die Funktionsweise der visuellen Sprache geht. In Syntaxdiagramm 19 ist exemplarisch ein vereinfachter Teilausschnitt dieses fiktiven Gebäudedatenmodells, welches der *VCCL* innerhalb dieser konzeptionellen Vorstellung zu Grunde liegt, in einem *Unified-Modeling-Language*-Diagramm dargestellt.

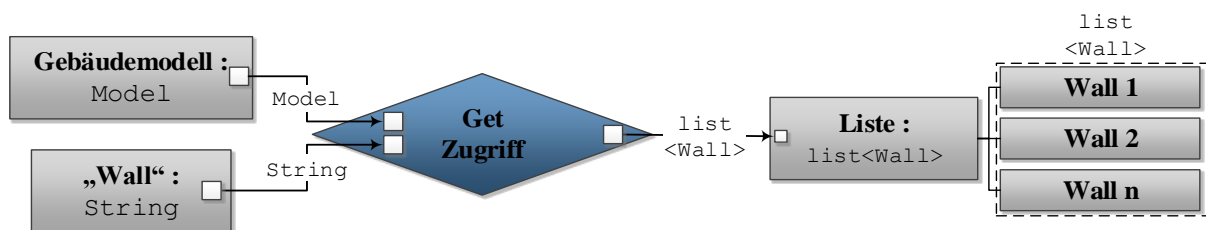
Syntaxdiagramm 19: Struktur des Gebäudemodells als Grundlage für die *VCCL*



Wie in der Struktur zu sehen ist, besitzt das Gebäudedatenmodell sowohl eine räumliche als auch eine elementbasierte Hierarchie. Eine Rückbeziehung zwischen diesen beiden Hierarchien kann jedoch über Attribute geschlossen werden. So kann beispielsweise ein Objekt des Typus *Wall* über das Attribut *Room* in die räumliche Struktur des Gebäudemodells eingeordnet werden. Diese Eigenschaft spielt bei der Anwendung von geometrisch-topologischen Verarbeitungsprozessen eine wesentliche Rolle, da die direkte Beziehung zwischen diesen Elementen den Prozess deutlich vereinfacht.

Das Gebäudemodell fungiert innerhalb des Verarbeitungssystems als Informationsquelle und kann daher als eigener Start-Objektknoten (vgl. Syntaxdiagramm 3) vom Datentyp `Model` dargestellt werden. Es kann an dieser Stelle vorausgesetzt werden, dass nach der Erstellung dieses Knotens sämtliche Informationen des Gebäudedatenmodells in dem Objekt zur Verfügung stehen.

Syntaxdiagramm 20: Zugriff auf das Datenmodell innerhalb mittels eines Zugriff-Operators

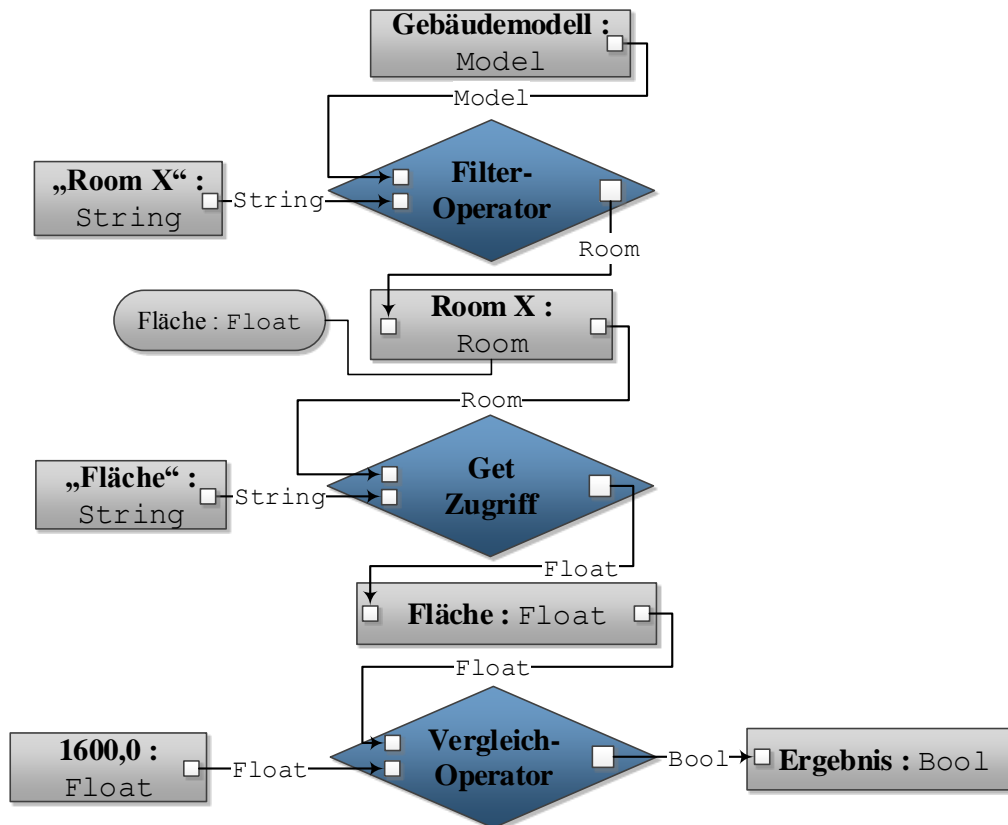


Anschließend kann dieser Objektknoten, wie bereits in Abschnitt 4.2.1.2 vorgestellt, mit Hilfe von Operatorknotten weiterverarbeitet werden. In Syntaxdiagramm 20 ist ein beispielhafter Zugriff auf die Objekte des Datentyps `Wall` über einen Zugriffoperator dargestellt.

Eine wichtige Grundlage für die Konzeption der *VCCL* stellt die Fehlerfreiheit und Konsistenz des Gebäudemodells dar. Da in der vorliegenden Arbeit ein fiktives Gebäudedatenmodell verwendet wird, müssen diese beiden Eigenschaften vorausgesetzt werden.

An dieser Stelle muss von dem Bearbeiter die Bedeutung dieser Passage interpretiert und anschließend in die *VCCL* übersetzt werden. Wenn man voraussetzt, dass die Struktur der Räume eines Gebäudemodells mit der Aufteilung in die Rauchabschnittsflächen korrespondiert, kann diese Passage, wie in Syntaxdiagramm 21 dargestellt, direkt formuliert werden.

Syntaxdiagramm 21: Darstellung einer Fließtext-Passage einer Vorschrift in *VCCL*



In dem abgebildeten *VCCL*-Graphen wird zunächst ein einzelnes Raumobjekt mit Hilfe eines Filterkriteriums innerhalb des Gebäudedatenmodells identifiziert und in einem separaten Objektknoten gespeichert. Der Raum wird in dem vorliegenden Beispiel eindeutig über das Attribut *Name*, also den *String*-Wert „Room X“, erkannt. Da die Speicherung der Zwischenschritte in einzelnen Objektknoten nur als Veranschaulichung des Informationsflusses innerhalb des Graphen dient, kann auf diese auch verzichtet und die Verknüpfung direkt mit dem nachfolgenden Knoten hergestellt werden.

Anschließend wird über einen Datenzugriff der Wert des Attributs *Fläche* abgefragt. Mit Hilfe dieser Information kann in einem Vergleich-Operatorknoten überprüft werden, ob die Anforderung, dass die Fläche kleiner als 1600 m² ist, eingehalten wird. Die Ausformulierung dieser Vergleichsoperation ist in Code 23 dargestellt.

Code 23: Abschließende Vergleichsoperation einer Überprüfung in Pseudocode

```

1 // Vergleichsoperator
2 Process (Input Wert, Input Vergleichswert)
3 {
4     if (Wert <= Vergleichswert)
5     {
6         return TRUE;
7     }
8     else
9     {
10        Return FALSE;
11    }
12 }

```

Die zentrale Anforderung der *DIN 18232-2:2007-11* ist die Einhaltung der notwendigen Rauchabzugsfläche. Diese Fläche stellt einen Grenzwert dar, welcher in Abhängigkeit einiger Parameter durch eine Datentabelle, die ausschnittsweise in Abbildung 62 dargestellt ist, beschrieben wird.

Raumhöhe ^a	Höhe der Rauchschiicht	Höhe der raucharmen Schicht	Notwendige Rauchabzugsfläche A_w in m ²				
			Bemessungsgruppe				
h in m	z in m	d in m	1	2	3	4	5
3,0	0,5	2,5	4,8	6,2	8,2	11,0	15,4
3,5	1,0	2,5	3,4	4,4	5,8	7,8	10,9
	0,5	3,0	6,7	8,7	11,3	15,0	20,4
4,0	1,5	2,5	2,8	3,6	4,7	6,4	8,9
	1,0	3,0	4,8	6,2	8,0	10,6	14,4
4,5	2,0	2,5	2,4	3,1	4,1	5,5	7,7
	1,5	3,0	3,9	5,0	6,5	8,7	11,8
	1,0	3,5	5,9	8,4	10,7	13,9	18,6

Abbildung 62: Ausschnitt der Tabelle für die notwendige Rauchabzugsfläche A_w in m² je Rauchabschnitt [DIN 18232-2:2007-11]

Der Soll-Wert der Rauchabzugsfläche für einen Raum ergibt sich über die Höhe des Raumes, die Höhe der Rauchschiicht und der jeweiligen Brandbemessungsgruppe, d.h. der Stärke des Brandes. Bei der Höhe des Raumes handelt es sich um das Attribut des Raumobjekts, dessen Wert direkt aus dem Gebäudedatenmodell bezogen werden kann. Die Höhe der Rauchschiicht und die Brandbemessungsgruppe hingegen beziehen sich auf die Brandschutzklasse und die Intensität des Brandes, welche für den Raum überprüft werden soll, und erfordern eine Nutzereingabe.

Der Ist-Wert der Rauchabzugsfläche muss direkt aus dem Gebäudemodell bezogen werden, denn dabei handelt es sich um einen Wert, der von den vorhandenen Öffnungen des Raumes abhängig ist. Dabei ist zu beachten, dass innerhalb der Norm eine Vielzahl von Korrektur- und Abminderungswerten angegeben wird, welche die Beschaffenheit und Eigenschaften des Raumes mit einberechnet.

Zum Beispiel beschreibt die Norm, dass die Öffnungsfläche eines Fensters durch den Fenstertypus und den Öffnungswinkel des Fensters beeinflusst wird. Dieser Umstand wird innerhalb einer Tabelle in Abhängigkeit von Fenstertypus und Öffnungswinkel als Korrekturfaktor c_z angegeben (siehe Abbildung 63).

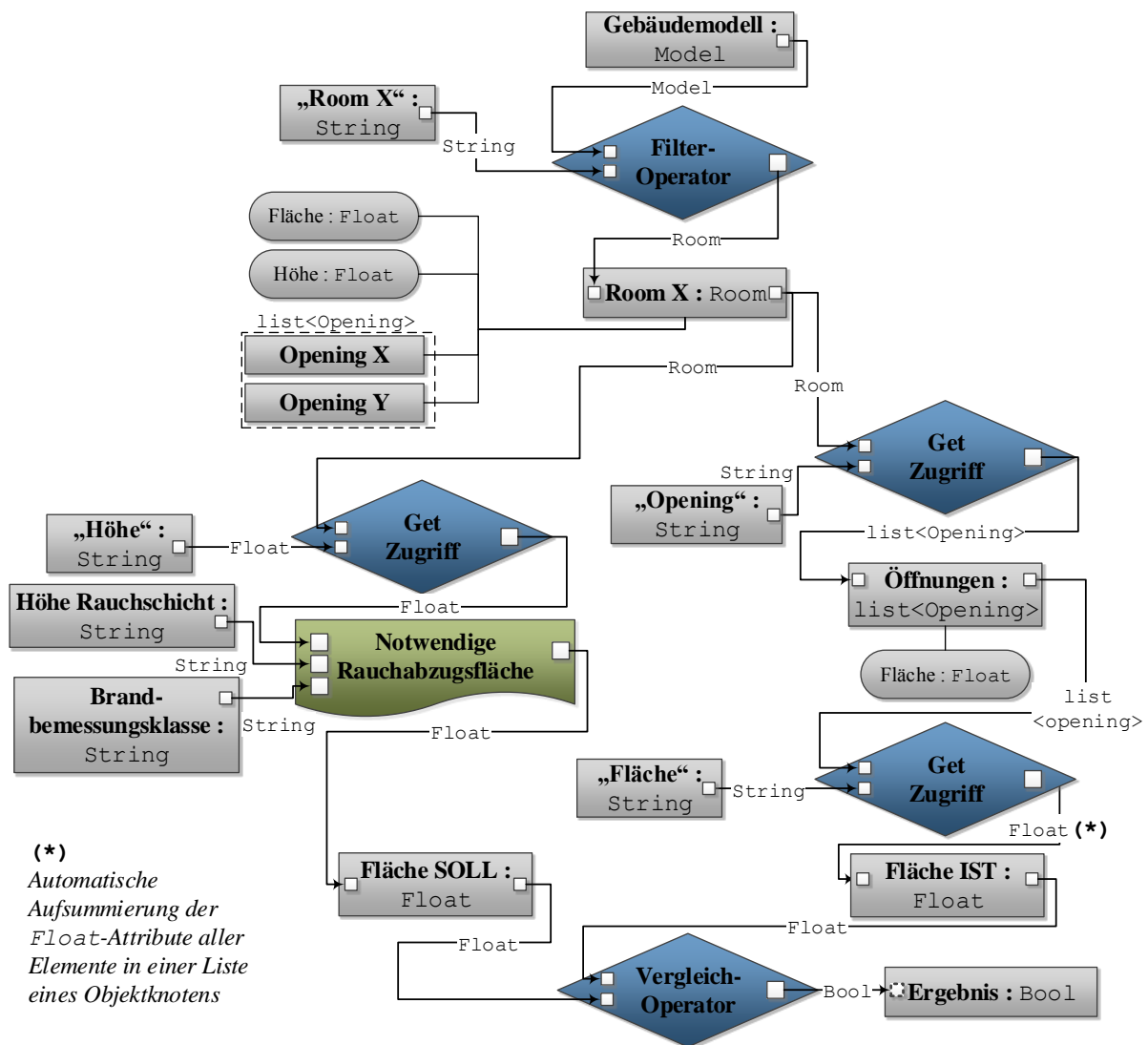
Öffnungsart	Öffnungswinkel	Korrekturfaktor c_z
Tür- oder Toröffnungen, Maschengitter		0,7
öffenbare Jalousien	90°	0,65
Dreh- oder Kippflügel	90°	0,65
	≥ 60°	0,5
	≥ 45°	0,4
	≥ 30°	0,3

Abbildung 63: Korrekturfaktor c_z für unterschiedliche Öffnungswinkel von Öffnungsflächen [DIN 18232-2:2007-11]

Diese Korrekturen können ebenfalls in der VCCL abgebildet werden, sofern die dafür notwendigen Informationen auch in dem Gebäudedatenmodell enthalten ist. In dem obigen Beispiel würde dieses bedeuten, dass für das Objekt *Fenster* die Attribute *Typ* und *Öffnungswinkel* definiert sein müssen, damit der Korrekturfaktor über einen Datentabellenknoten ausgelesen werden kann. In der vorliegenden Arbeit sind diese Anforderungen der Vollständigkeit halber erwähnt, um den Abbildungsprozess transparent zu beschreiben. In dem nachfolgenden Gesamtbeispiel werden sie dann jedoch nicht in den Graphen mit einbezogen.

Abschließend lässt sich nun mit Hilfe der VCCL eine Verarbeitungsprozedur formulieren, welche den Überprüfungsprozess zu der Einhaltung der Rauchabschnittsfläche eines Raumes beschreibt. Hierzu wird auf der einen Seite der Ist-Wert aus den Informationen des Gebäudemodells und auf der andere Seite der Soll-Wert mit Hilfe der Datentabelle bestimmt. Die beiden erhaltenen Werte können abschließend in einem Vergleich-Operatorknoten einander gegenüber gestellt werden.

Syntaxdiagramm 22: Schematischer Ablauf des gesamten Überprüfungsprozesses



Der gesamte Überprüfungsprozess, welcher in Syntaxdiagramm 22 dargestellt ist, beginnt zunächst wiederum mit einer Identifikation eines Raumobjekts. Anschließend wird in dem ersten Zweig des Prozesses das Attribut *Höhe* des Raumobjekts über einen Zugriff-Operator abgefragt und zusammen mit den beiden Nutzereingaben, welche durch die Objektknoten *Höhe der Rauchschicht* und *Brandbemessungsklasse* dargestellt sind, in den Datentabellenknoten eingegeben. Das Ergebnis dieser Operation ist der Soll-Wert für die notwendige Rauchabzugsfläche, welcher in dem Objektknoten *Fläche SOLL* abgelegt wird.

In dem anderen Prozess-Zweig werden alle Objekte des Datentyps `Opening` aus dem Gebäudedatenmodell gefiltert und in einem Objektknoten mit dem Datentyp `list<Opening>` abgelegt, da an dieser Stelle vorausgesetzt wird, dass das Objekt *Raum* ein zugehöriges Attribut besitzt, welches die Öffnungen des Raumes beinhaltet. Die gesamte Öffnungsfläche des Raumes, welche den Ist-Wert der Überprüfung darstellt, kann aus dem Attribut *Fläche* des Objektknotens *Öffnungen* abgefragt werden. Hierbei ist zu beachten, dass der *Get-Zugriff* des Operatorknottes bei der Anwendung auf einen Objektknoten mit einer Liste die Werte der `Float`-Attribute automatisch aufsummiert.

Abschließend kann die Einhaltung der Vorschrift durch den Vergleich von Soll- und Ist-Wert in einer letzten Operation überprüft werden.

Da es sich bei dem aufgeführten Beispiel zu der *VCCL* bisher nur um eine rein theoretische Anwendung handelt, muss die Tragfähigkeit des vorgestellten Ansatzes noch nachgewiesen werden. Daher soll nun in dem folgenden Abschnitt eine praktische Implementierung der *VCCL* vorgestellt werden, welche die Eignung der visuellen Sprache für die praktische Anwendung unter Beweis stellt.

5 Nachweis der Tragfähigkeit

Im Folgenden wird das Tool *Code Builder* vorgestellt, das in der vorliegenden Arbeit entwickelt wurde. Es soll die Tragfähigkeit des in Kapitel 4 vorgestellten Ansatzes nachweisen. In den folgenden Unterkapiteln wird die Entstehung und Funktionsweise dieser Softwarelösung beschrieben.

5.1 bim+

Code Builder wurde in enger Zusammenarbeit mit der *bim+ GmbH*, einem Tochterunternehmen der *Nemetschek AG*, entwickelt.

Die *Nemetschek AG* wurde im Jahre 1963 von Dipl.-Ing. Georg Nemetschek als *Ingenieurbüro für das Bauwesen* in München gegründet. Als eines der ersten Ingenieurbüros setzte es auf die Unterstützung von Computern im Entwurf und in der Konstruktion von Bauprojekten und entwickelte zunächst ausschließlich für den Eigenbedarf Software im Bereich der Tragwerksplanung für Hoch- und Tiefbauten. Heute setzt sie sich aus mehreren Tochterfirmen zusammen und beschäftigt rund 2500 Mitarbeiter. Bekannte Produkte der *Nemetschek AG* sind die CAD-Softwarepakete *Nemetschek Allplan* und *Vectorworks* (Nemetschek AG 2013).

Im Jahr 2013 wurde von der *Nemetschek AG* die Tochterfirma *bim+ GmbH* gegründet, die sich auf die Entwicklung einer *Open-BIM*-Plattform konzentriert. Ziel der Entwicklung ist es, eine zentrale und in alle Richtungen kompatible Softwarelösung für alle Beteiligten eines Bauprojekts zu bieten und so Abstimmungsprobleme und –konflikte während des Bauprozesses zu vermeiden. Dies wird über ein zentrales, online gespeichertes Datenmodell, die *bim+*-Plattform, realisiert. Sie ermöglicht die Speicherung des gesamten Informationsgehalts eines Gebäudemodells sowie eine direkte Kommunikation und einen unmittelbaren Informationsaustausch zwischen allen Projektbeteiligten.



Abbildung 64: Firmenlogo von bim+ (2013)

Im Gegensatz zu anderen *BIM*-Plattformen, welche von weiteren *CAD*-Softwareherstellern in den vergangenen Jahren entwickelt wurden, ist *bim+* nicht ausschließlich auf die Verwendung eines herstellereigenen Formates ausgelegt, sondern soll als Schnittstelle für mehrere etablierte Gebäudemodellformate fungieren und so eine optimale Grundlage für die Entwicklung von externen Softwarelösungen bieten.

In Abbildung 65 ist die grundlegende Struktur der *bim+*-Plattform dargestellt, welche die verschiedenen Ebenen der Softwarearchitektur aufzeigt. Der Kern der Plattform, das sogenannte *bimOS*, arbeitet serverseitig und bietet dem Anwender alle wesentlichen Funktionen eines Cloud-Service, die sogenannten *CRUD* (*Create, Read, Update & Delete*). Zu diesen

Funktionen des zentralen *bimOS* gehören vorrangig die Speicherung, Verwaltung und Organisation der Gebäudemodelle. An dieser Stelle zeigt sich bereits die Offenheit der Plattform, da sowohl weit verbreitete offene Datenmodellstandards, wie das *IFC*-Format, als auch Formate anderer *CAD*-Softwarehersteller in die Plattform importiert werden können (siehe Abbildung 65). Dabei wird der Informationsgehalt der einzelnen Gebäudemodelle nicht über die Datenstruktur des jeweiligen Formates auf die Plattform geladen, sondern in ein *bim+*-eigenes Datenmodell übertragen und innerhalb einer serverseitigen Datenbank gespeichert. Es handelt sich also nicht um einen direkten Import, sondern vielmehr um eine fest implementierte Abbildungsfunktion für jedes einzelne, kompatible Datenformat.

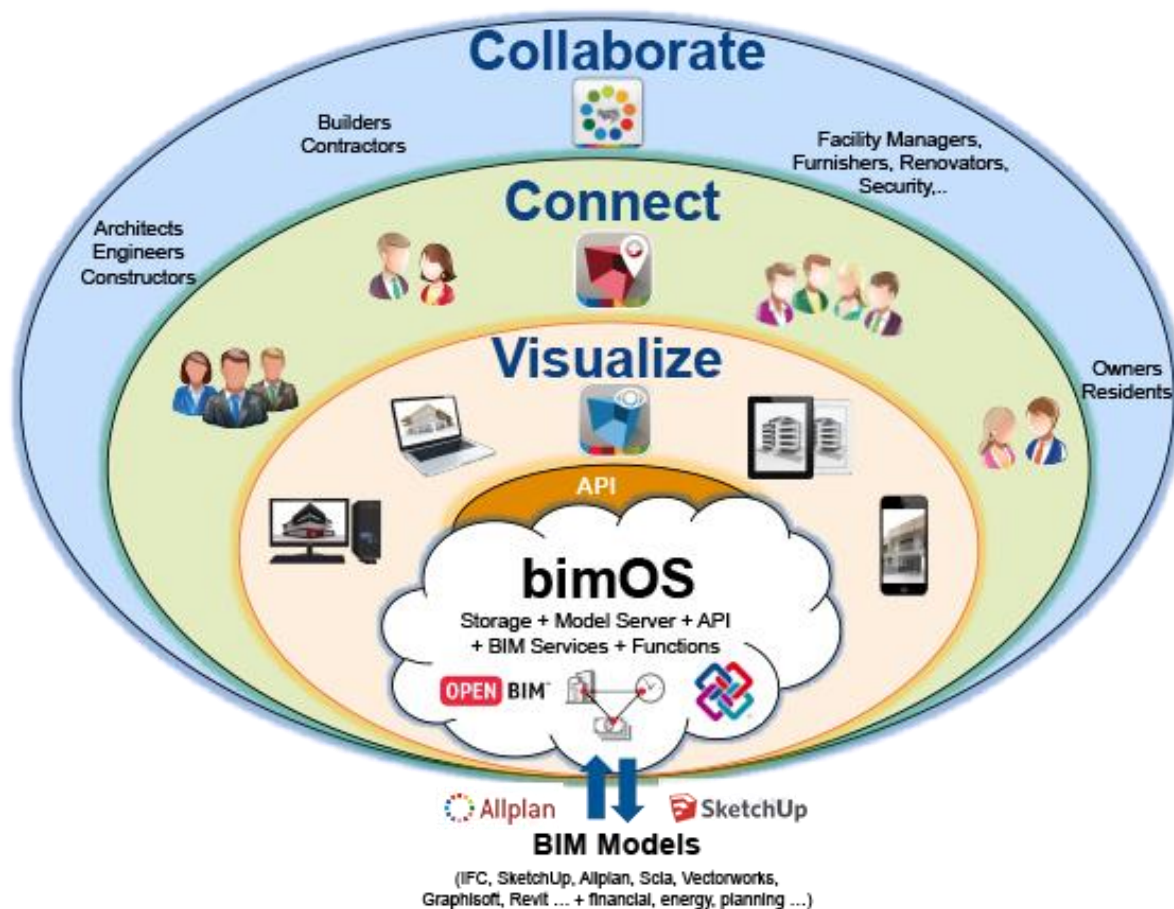


Abbildung 65: Struktur der *bim+*-Plattform (*bim+* 2013)

In den übergeordneten Ebenen bietet *bim+* eine Vielzahl von Grundfunktionalitäten, auf welche nach der Registrierung auf der *bim+*-Plattform zugegriffen werden kann. Das sogenannte *bim+*-Portal ist dafür zuständig, dem Nutzer auf unterschiedliche Art und Weise den Zugriff und die Weiterverarbeitung der Informationen eines Gebäudemodells zu ermöglichen.

Ein wesentlicher Bestandteil dieses Portals ist die Webbrowser-Applikation von *bim+*, welche unter anderem eine Modell-, Projekt- und Teamverwaltung (siehe Abbildung 66 und Abbildung 67) sowie einen 3D-Viewer (siehe Abbildung 68) und somit alle grundlegenden Funktionen für die Verwaltung der Modelle bietet.

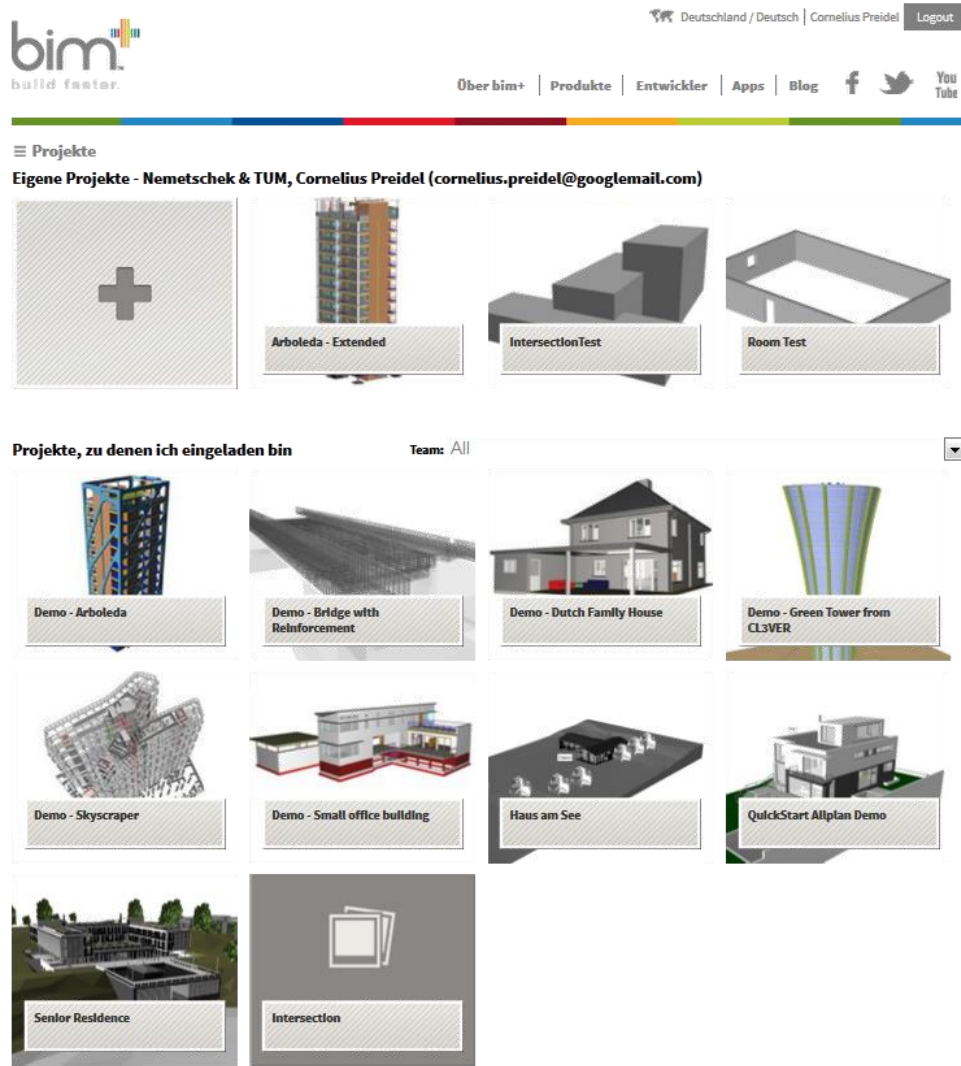


Abbildung 66: Projekt- und Modellauswahl von bim+ ein einem Web-Browser

Jeder registrierte Nutzer der Plattform kann eine beliebige Anzahl von Gebäudemodellen in seinem Konto speichern und diese für andere Projektbeteiligte, welche ebenfalls unter *bim+* registriert sein müssen, freigeben. Hierzu bietet die Plattform eine eigene Projektverwaltung, in welcher die Nutzungsrechte und Funktionen der jeweiligen Beteiligten zugewiesen werden können (siehe Abbildung 67).

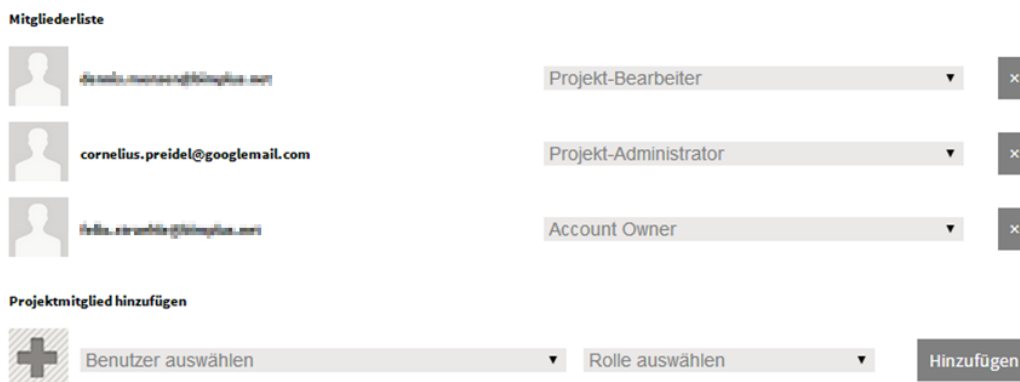


Abbildung 67: Teamverwaltung für ein Modell in bim+

Der geometrische Informationsgehalt eines Gebäudemodells kann ebenfalls direkt in der Browser-Applikation in einem 3D-Viewer angezeigt und detaillierte Informationen zu einzelnen Bauteilen können eingesehen werden (siehe Abbildung 68).

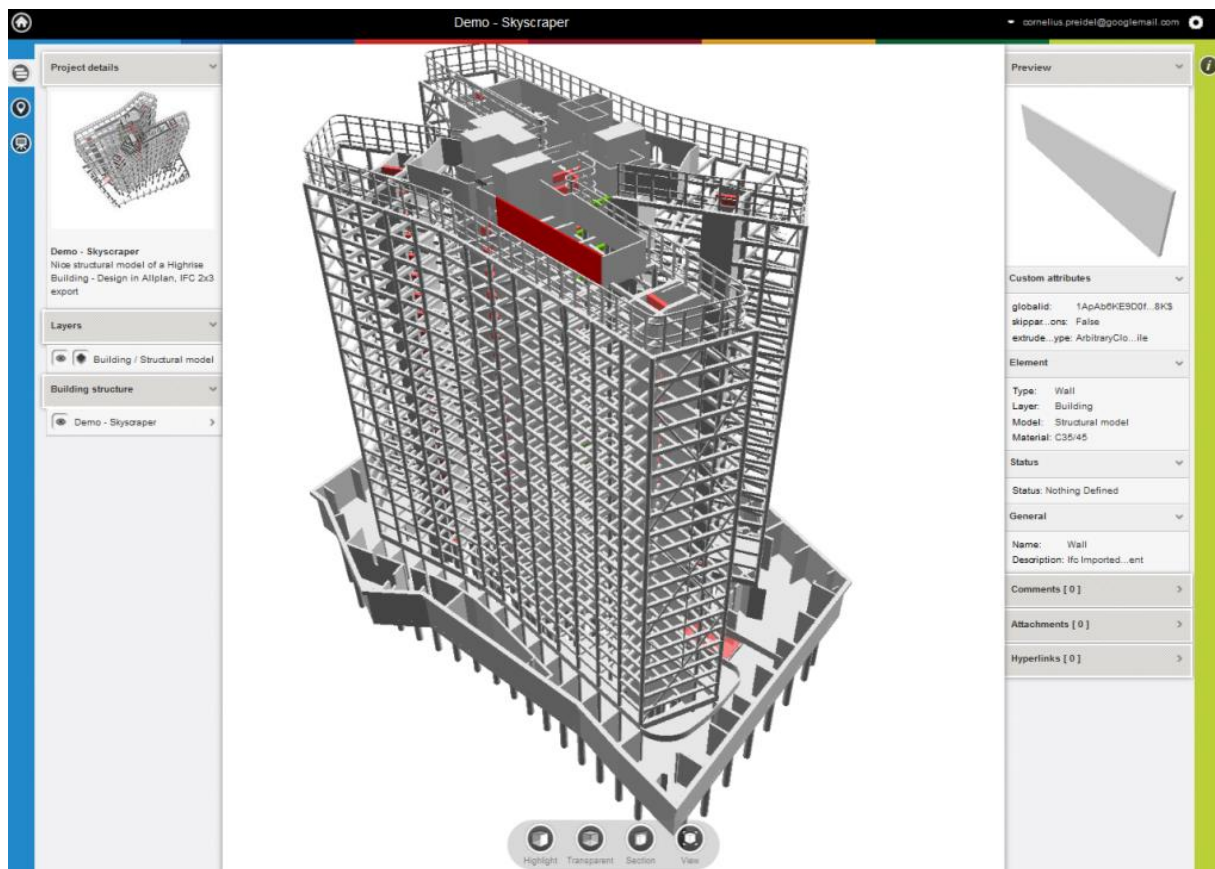


Abbildung 68: Web-Viewer von bim+ im Browser

Die *bim+*-Plattform bietet eine Vielzahl von Möglichkeiten für externe Entwickler, ihre Software in der Plattform zu integrieren oder an diese anzubinden. Damit erfüllt sie den Anspruch einer Entwickler-freundlichen Umgebung für Softwarehersteller der verschiedenen Fachdisziplinen des Bauwesens.

In der *bimOS* stehen mehrere Programmierschnittstellen, sogenannte *Application Programming Interfaces (API)*³, zur Verfügung, über welche von außerhalb auf die Plattform zugegriffen werden kann. Alle verfügbaren *APIs* der *bim+*-Plattform basieren auf dem Programmierparadigma *Representational state transfer (REST)* und verwenden die *JavaScript Object Notation (JSON)* als Datenaustauschprotokoll. Mit Hilfe dieser Programmierschnittstellen kann von externen Softwarelösungen auf die unterschiedlichen Informationsebenen der Plattform zugegriffen werden. Zusätzlich können neue Datensätze erstellt, modifiziert oder auch gelöscht werden.

In Abbildung 69 ist die technische Struktur der *bim+*-Plattform dargestellt, welche sich durch die Verknüpfung der einzelnen Elemente über die vorhandenen *APIs* ergibt.

³ <https://doc.bimplus.net/display/bimpluspublic/Home>

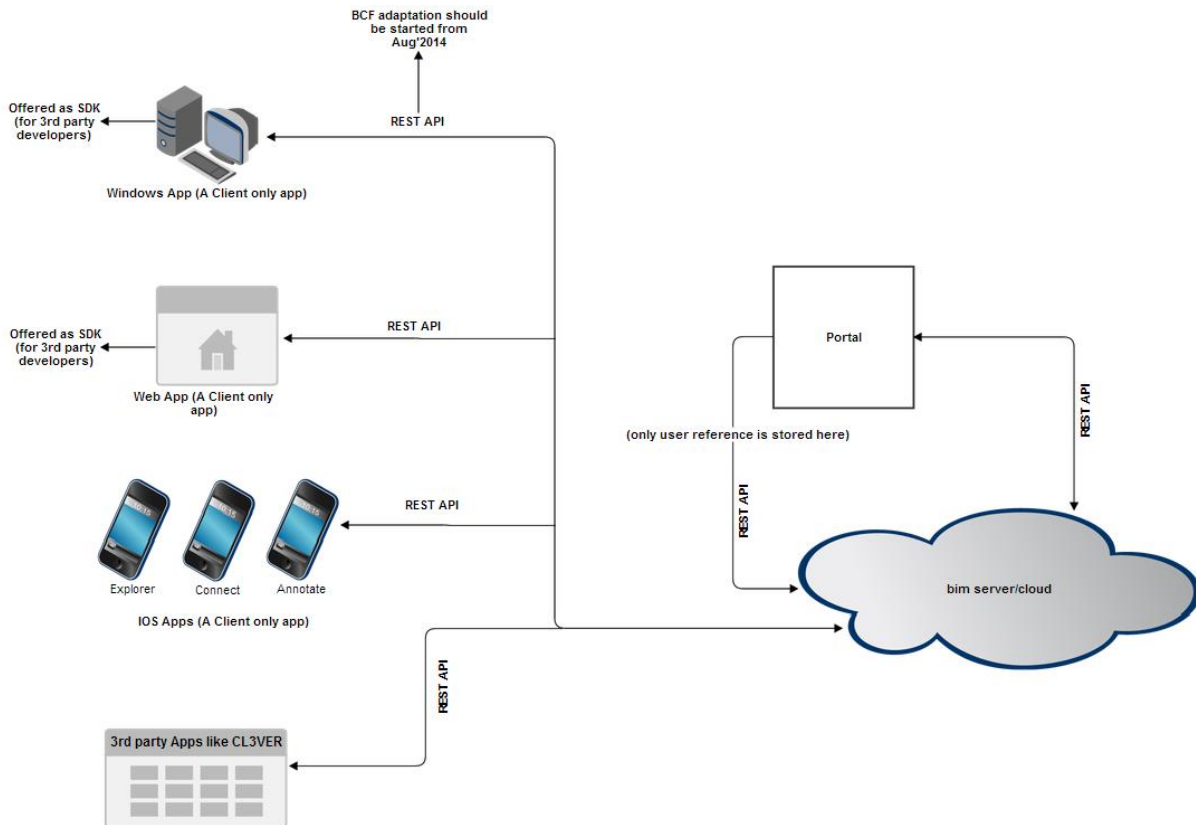


Abbildung 69: Technische Struktur der bim+-Plattform (bim+ 2013)

Darüber hinaus wird die Plattform durch die APIs in sogenannte *REST-Services* gegliedert, die jeweils angeben auf welchen spezifischen Informationsgehalt von der jeweiligen Programmierschnittstelle zugegriffen werden kann (siehe Abbildung 70). Eine ausführliche Dokumentation der jeweiligen Services ist auf der Internetpräsenz von *bim+* zu finden⁴.

Project relevant services

[Project Service](#)
[Model Service](#)
[Object Service](#)
[ElementType Service](#)
[Attachment Service](#)
[Issue Service](#)
[Pin Service](#)
[Comment Service](#)
[Import Service](#)
[Slideshow Service](#)
[Hyperlink Service](#)

Administration relevant services

[Authorization Service](#)
[User Management Service](#)
[Team Management Service](#)
[Membership Management Service](#)
[Rights & Roles Service](#)

Other services

[Message Service](#)

Internal services

[Log Service](#)
[User Activation/Deactivation Service](#)
[Team Activation/Deactivation Service](#)
[DB Statistics Service](#)
[DB MoveService](#)

Abbildung 70: API-Services von bim+

⁴ <https://doc.bimplus.net/pages/viewpage.action?pageId=4459171>

Für die Entwicklung von webbasierten Applikationen steht ein *Web Software Development Kit*⁵ frei zur Verfügung, welches den Zugriff auf die serverseitigen Informationen sowie mehrere Grundfunktionalitäten, wie diese auch in der Webbrowser-Applikation verwendet werden, erlaubt.

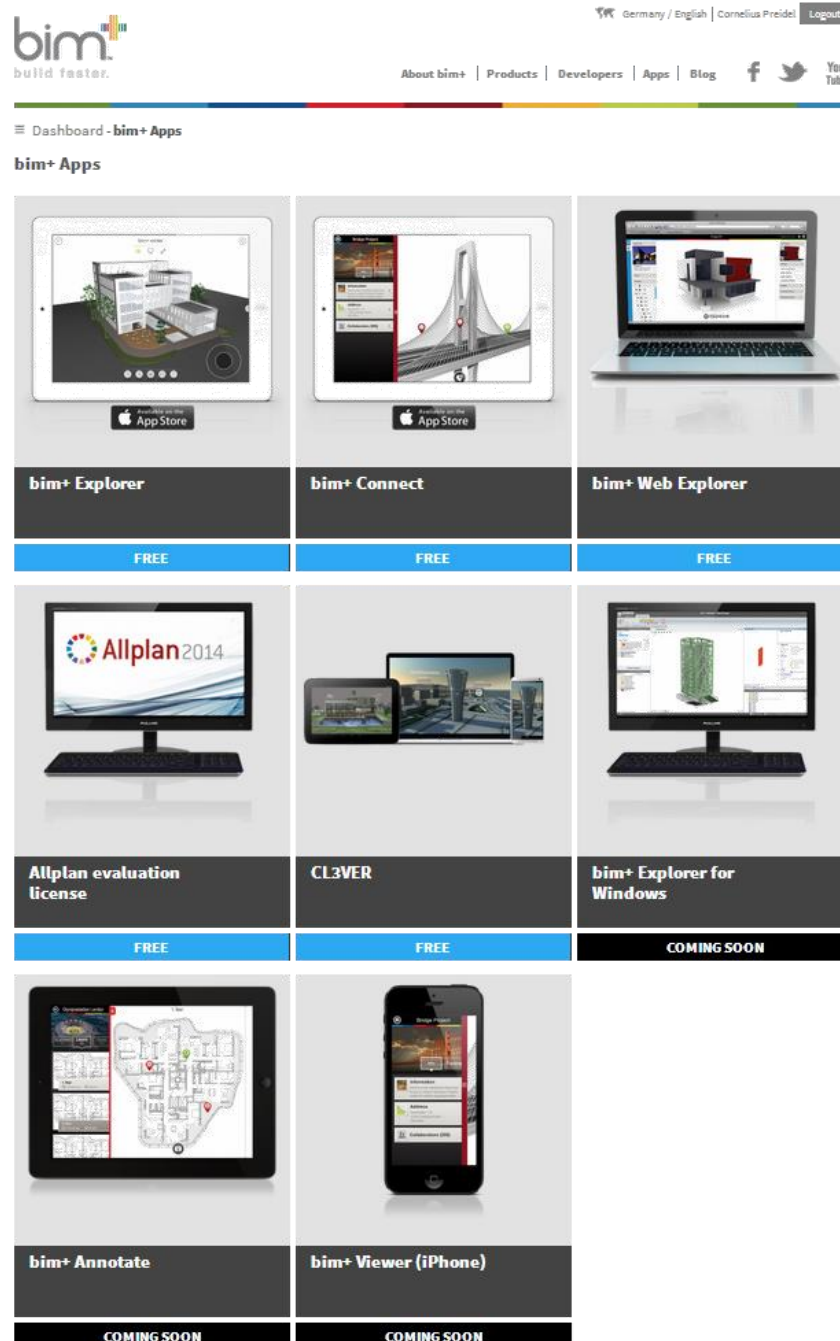


Abbildung 71: bim+ Appstore

Für die Vermarktung und den Verkauf der Software bietet *bim+* den externen Entwicklern einen *Appstore*⁶, der auf der Homepage von *bim+* zu finden ist (siehe Abbildung 71). In diesem

⁵ <https://github.com/Bimplus> & <https://doc.bimplus.net/pages/viewpage.action?pageId=10911766>

⁶ <https://www.bimplus.net/Apps/bim-Apps/>

werden aktuell bereits einige *bim+*-eigene und auch externe Entwicklungen, wie z.B. ein Tool des Softwareherstellers *CL3VER*, kostenfrei angeboten.

Für die Erleichterung der Entwicklung von lokalen Applikationen in der *.Net*-Umgebung steht darüber hinaus die *Windows*-basierte Softwarelösung *bim+ Explorer* zur Verfügung. In dieser Applikation ist der Zugriff auf die serverseitigen Informationen über die proprietäre *API* innerhalb von Funktionen bereits integriert und bietet so eine Entwickler-freundliche und leicht verständliche Umgebung. Eine erste Entwicklerversion des *bim+ Explorer* steht im Internet frei zur Verfügung.⁷

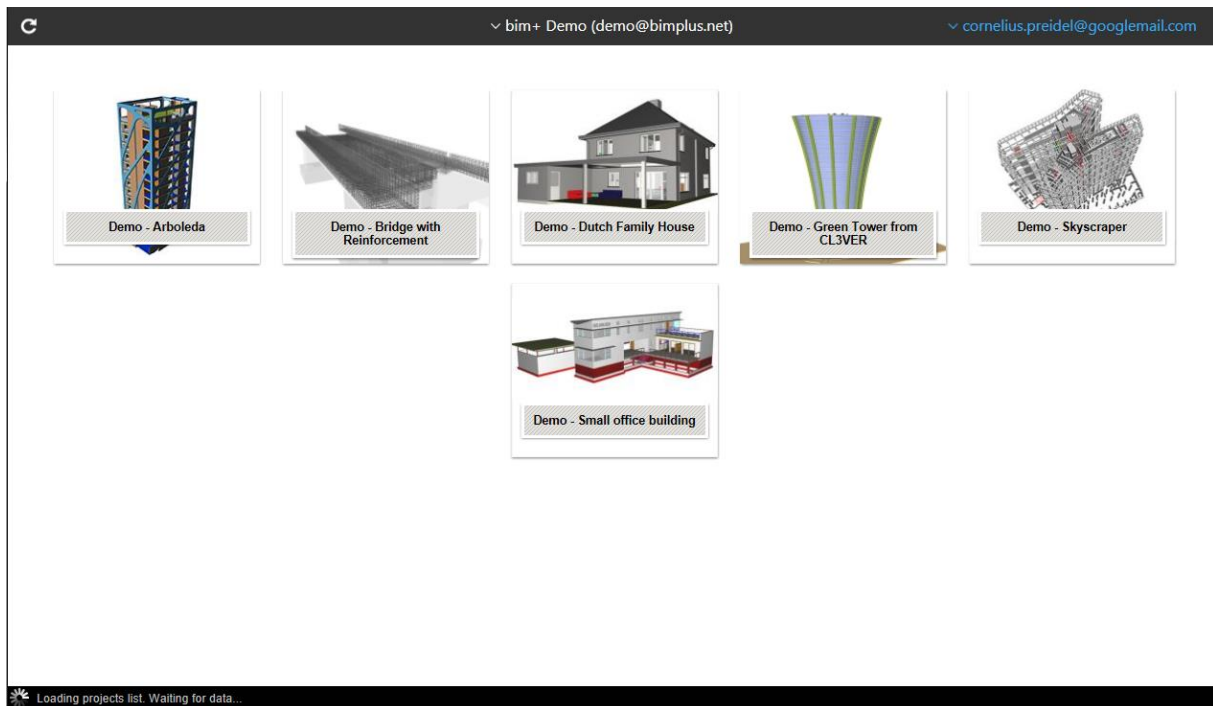


Abbildung: Projektauswahl im bim+ Explorer

Der *bim+ Explorer* ist ähnlich aufgebaut wie die Webbrowser-Version von *bim+* und bietet ebenfalls viele grundlegende Funktionalitäten wie eine Projekt- und Modellauswahl sowie einen 3D-Viewer. Diese Funktionalitäten arbeiten jedoch nicht webbasiert wie die Elemente der *Web SDK*, sondern sind als Module lokal in die Applikation integriert. Von allen Modulen des *bim+ Explorer* ist ein Zugriff auf die proprietäre *API* der *bim+-Plattform* über implementierte Funktionen möglich.

⁷ <https://github.com/Bimplus/Windows-SDK>

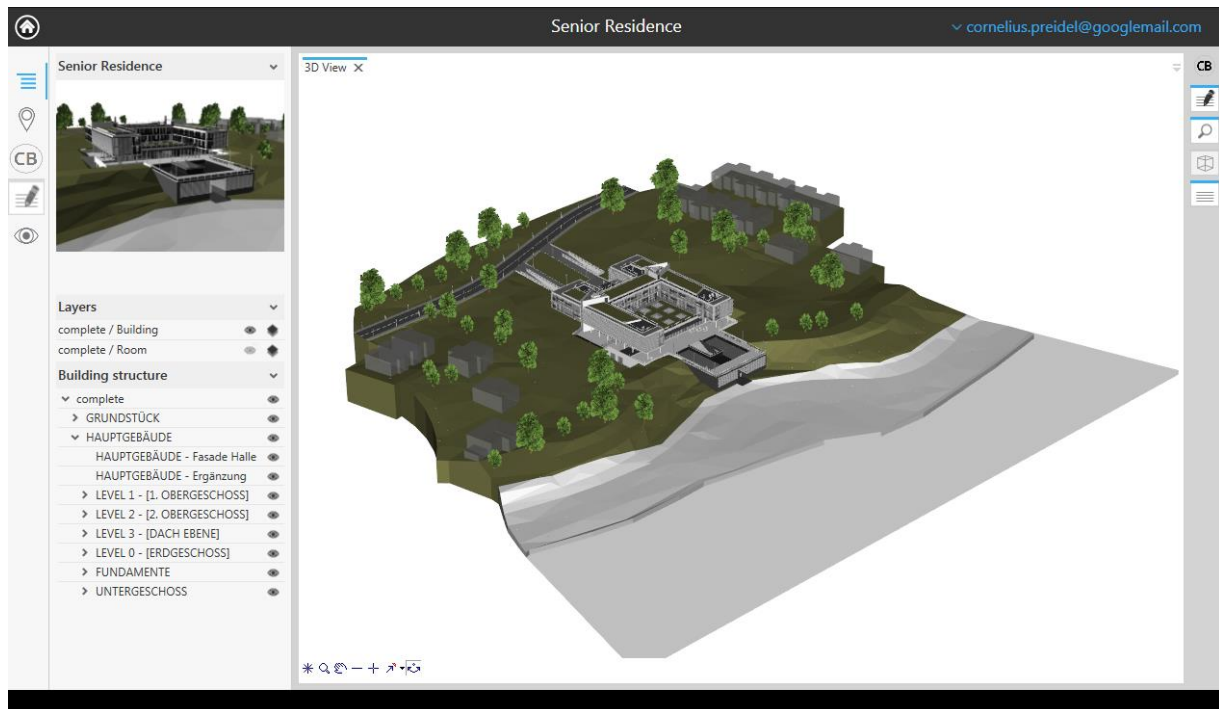


Abbildung 72: 3D-Viewer im bim+ Explorer

Für die Anbindung externer Software ist der *bim+ Explorer* nach dem Prinzip des *Managed Extensibility Framework (MEF)* aufgebaut, so dass externe Module sehr leicht als Plug-Ins integriert werden können. Daher eignet sich diese Applikation in der vorliegenden Arbeit als Grundlage für den Nachweis der Tragfähigkeit der *VCCL*.

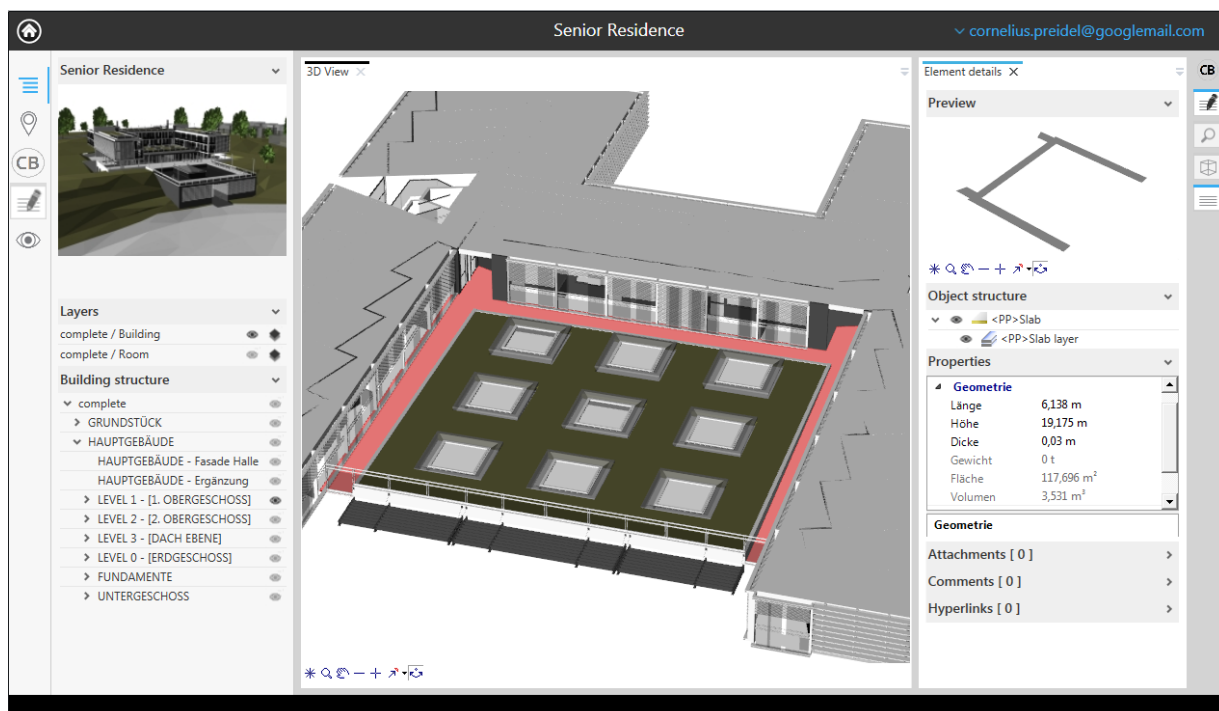


Abbildung 73: Detailansicht im bim+ Explorer

5.2 Code Builder

5.2.1 Aufbau und Funktionsweise

Wie in Abbildung 74 dargestellt wurde das Tool *Code Builder* in die Umgebung des *bim+ Explorer* über die Plug-In-Schnittstelle integriert.

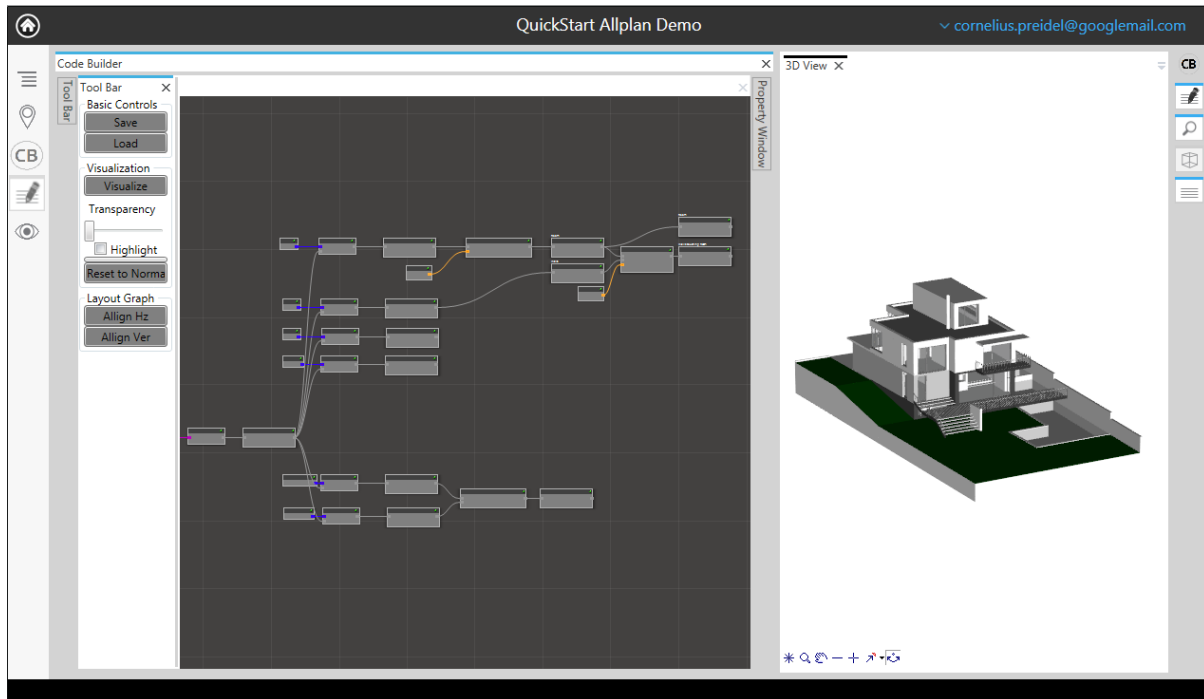


Abbildung 74: Userinterface des Code Builder im bim+ Explorer

Für die Umsetzung der visuellen Sprache *VCCL* innerhalb des *Code Builder* wurde die Klassen-Bibliothek *NodeGraph* verwendet. In dieser ist eine Auswahl von vorgefertigten Strukturen und graphischen Elementen für die Entwicklung einer visuellen Sprache enthalten. *NodeGraph* bietet eine Zeichenfläche, grundlegende Knotentypen, deren graphische Darstellung und die Vernetzung der visuellen Objekte mit Hilfe von Kanten. Die Bibliothek ist auf eine Entwicklung in der *.NET*-Umgebung ausgelegt und steht im Internet frei zur Verfügung⁸.

Mit Hilfe dieser Elemente konnten mehrere Knotentypen für die *VCCL* definiert werden, um grundlegende Operationen und Prozesse darzustellen. Eine Auswahl deklarerter Knotentypen ist in Abbildung 75 dargestellt.

⁸ <https://nodegraph.codeplex.com/>

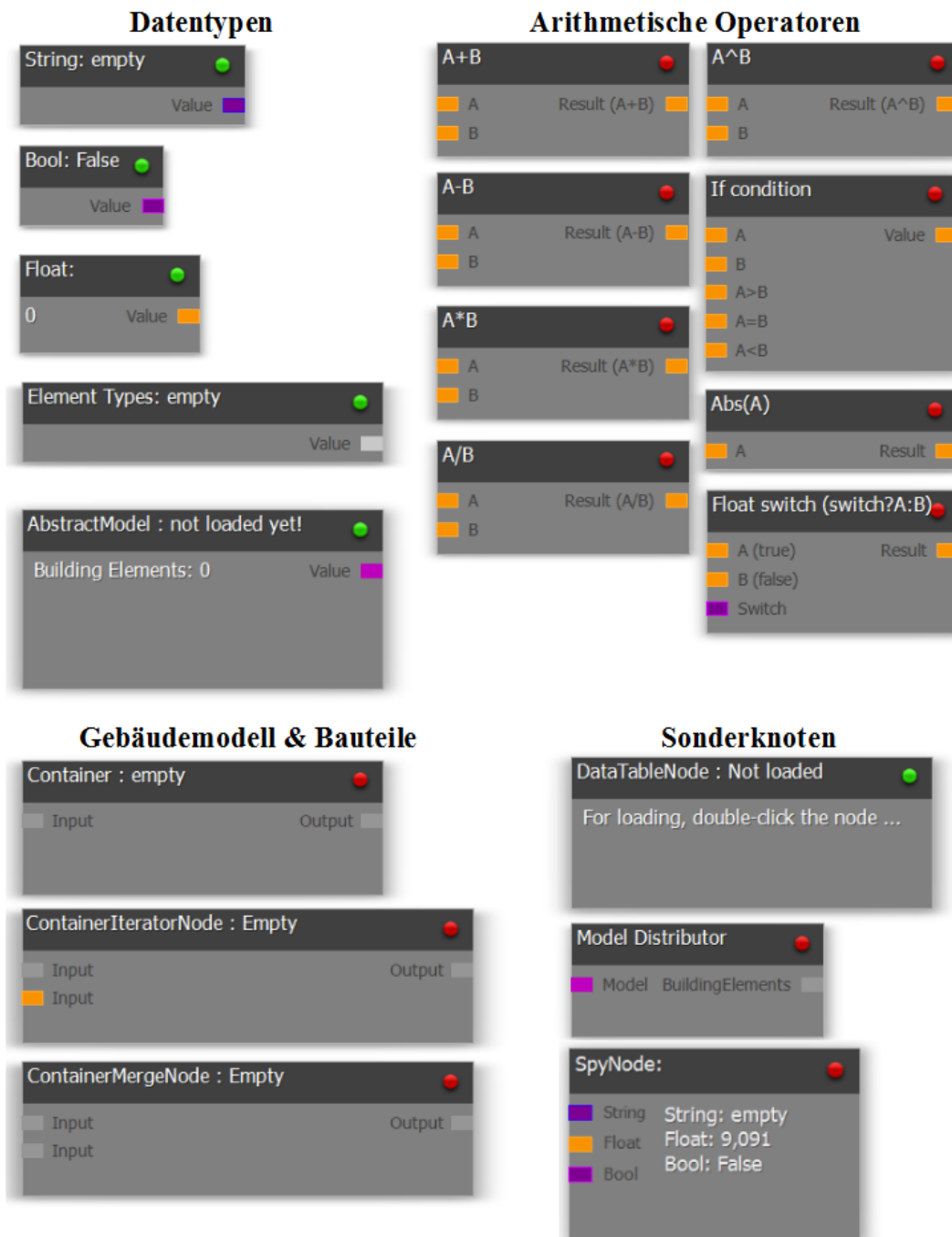


Abbildung 75: Knotenobjekte des Code Builder

Dabei ist zu beachten, dass es aufgrund programmiertechnischer Einschränkungen einige Unterschiede zu der Konzeption der *VCCL*, wie diese in Abschnitt 4.2 vorgestellt wurde, gibt:

- Alle Knotentypen des *Code Builder* werden als rechteckige Knoten dargestellt. Somit gibt es keine graphische Unterscheidung zwischen Operator- und Datenknoten.
- Die graphische Veranschaulichung von Attribut- und Listenobjekten, wie diese bei der Vorstellung des Konzepts der *VCCL* zur Veranschaulichung der Informationen verwendet wurde, ist in *Code Builder* nicht umgesetzt worden.
- Die Kennzeichnung der einzelnen Knotentypen mit Hilfe der Labels ist in *Code Builder* möglich, jedoch wie in Abbildung 76 dargestellt leicht verändert worden.

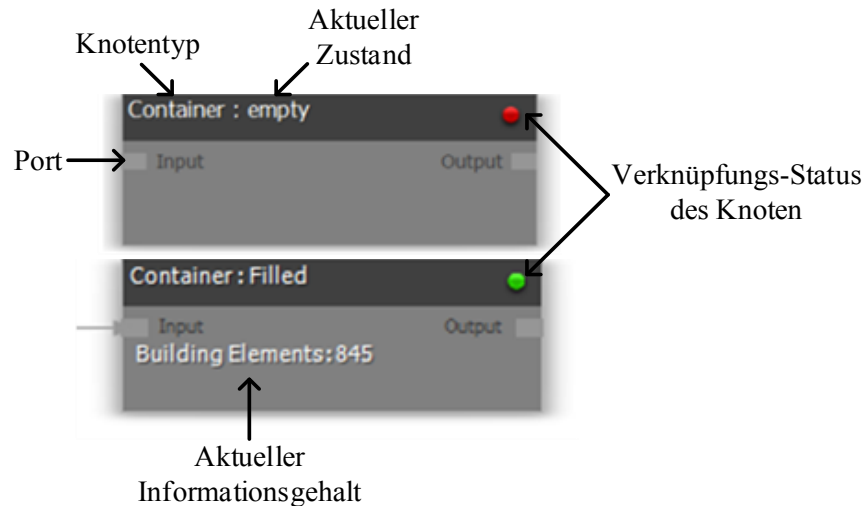


Abbildung 76: Darstellung eines Objektknoten in unterschiedlichen Zuständen in Code Builder

Bei der Definition aller Knotentypen wurde insbesondere auf die Grundsätze der *VCCL* geachtet, welche in Abschnitt 4.2.2 erläutert wurden.

Jede der Input- und Output-Schnittstellen eines Knotens - ein sogenannter *Port* - besitzt einen eindeutigen Datentyp, welcher durch eine Farbe gekennzeichnet wird. Auf diese Weise ist eine Übermittlung von falschen Informationen zwischen den Knoten gemäß der Konzeption der *VCCL* ausgeschlossen.

Die textuelle Kennzeichnung der übertragenen Datentypen auf den Kanten, wie diese bei der Vorstellung der *VCCL* in Abschnitt 4.2 verwendet wurde, wurde ebenfalls durch eine farbige Markierung der gerichteten Kanten, den sogenannten *Connectors*, ersetzt.













Datentyp-Name	Pendant in VCCL	Beispiel / Instanziierung	Beschreibung	Port-Farbe	Connector-Farbe
Basistypen					
Bool	<i>Bool</i>	True, False	Wahrheitswert		
Float	<i>Float</i>	1,2345	Fließkommazahl		
String	<i>String</i>	„Beispiel“	Zeichenkette		
Gebäudemodell					
AbstractModel	<i>Model</i>	-	Gebäudemodell		
ElementType	-	Wall, Opening, Room	Element-Typen		
GenericElement Container	<i>list</i> < <i>BuildingElement</i> >	{Wall1, Wall2, Slab1}	Liste von Bauteilen		

Abbildung 77: Datentypen in Code Builder

In Abbildung 77 sind sämtliche Datentypen des *Code Builder*, deren Pendants in der *VCCL* (siehe Abschnitt 4.2.1) sowie die farbliche Kennzeichnung von *Ports* und *Connectors* dargestellt.

Nach dem Grundsatz der Generizität wurden alle Knotentypen so allgemeingültig wie möglich gehalten. Zur Speicherung von Bauteilen dient beispielsweise ein sogenannter *Container* (siehe Abbildung 75), welcher durch seinen Datentyp `list<BuildingElement>` in der Lage ist, eine beliebige Anzahl von Bauteilen zu speichern. Durch die Weiterverarbeitung dieses Objektknotens mit Hilfe von Operator-knoten, welche die beinhalteten Informationen filtern und weiterverarbeiten, kann darauf verzichtet werden, für jeden einzelnen Gebäudemodell-

spezifischen Datentyp einen eigenen Knotentyp zu definieren. So hat der Anwender alle Möglichkeiten und die größtmögliche Flexibilität, um die Informationen den individuellen Bedürfnissen anzupassen, ohne gleichzeitig die Übersicht über die verschiedenen Knotentypen der *VCCL* zu verlieren.

Durch das *MEF*-Design des *bim+ Explorer* hat ein Plug-In direkten Zugang auf die *API* von *bim+* und kann daher über sämtliche *Services* auf die serverseitigen Informationen zugreifen. Durch diese direkte Anbindung an die *BIM*-Plattform kann der Datenzugriff auf das Gebäudemodell sehr einfach gestaltet werden. Bei dem Design der Knotentypen wurde jeder Datenzugriff so formuliert, dass die Informationen erst dann von dem Server geladen werden, wenn der Zugriff in dem Verarbeitungsprozess erreicht wird. Dadurch werden so wenige Informationen wie möglich lokal gespeichert, und die aktuellen serverseitigen Informationen der Plattform werden in Echtzeit genutzt.

Zur Einbindung der Informationen des Gebäudedatenmodells wurde das Konzept der *VCCL* grundlegend beibehalten, und so steht ein Objektknoten des Datentyps *AbstractModel* zur Verfügung, welcher als Informationsquelle dient. Der Anwender kann je nach Bedarf für diesen Knotentyp über eine Nutzereingabe den Anstoß geben, dass das aktuell geladene Gebäudedatenmodell des *bim+ Explorer* in diesen Objektknoten geladen wird.

Alle weiteren Knotentypen, welcher innerhalb des *Code Builder* als Informationsquellen dienen (siehe Knoten *String*, *Bool* und *Float* in Abbildung 75), können über eine Nutzereingabe mit einem Wert belegt werden.

Mit Hilfe der vorgestellten Knotentypen und der Vernetzung der *Ports* mittels der *Connectors* kann nun eine Verarbeitungsprozedur abgebildet werden. In Abbildung 78 ist eine einfache Rechenoperation innerhalb von *Code Builder* dargestellt.



Abbildung 78: Visuelle Darstellung einer einfachen Rechenoperation im Code Builder

Wie bereits in Abschnitt 3.3.3 erläutert ist die Plausibilitätsüberprüfung ein wesentlicher Bestandteil des *Code Compliance Checking*. Die dynamische Zeichenumgebung beschreibt die Eigenschaft einer Verarbeitungsprozedur, die mit einer visuellen Sprache formuliert ist. Auf diese Weise kann auf äußere Einflüsse reagiert werden. Dadurch wird es möglich, dem Nutzer direkte Einblicke in die Zwischenprozesse zu geben und somit Stichproben während des Prozesses zu machen. Die Knoten der *VCCL* sind so gestaltet worden, dass diese zu jedem Zeitpunkt des Verarbeitungsprozesses ihre Ausgabewerte anzeigen können, wie dieses beispielsweise auch bei dem Endknoten *SpyNode* in Abbildung 78 zu sehen ist.

Durch die Integration des *Code Builder* in den *bim+ Explorer* kann diese Funktionalität für geometrische Prozesse auch visuell ausgestaltet werden. Dieses soll an Hand des *VCCL*-Graphen in Abbildung 79 gezeigt werden, welcher ein beliebiges Gebäudemodell hinsichtlich der enthaltenen Bauteiltypen filtert. Die gefilterten Objekte werden nicht nur als Ergebnis durch die resultierende Anzahl in den jeweiligen Objektknoten dargestellt, sondern können darüber hinaus auch in dem 3D-Viewer des *bim+ Explorer* visualisiert werden.

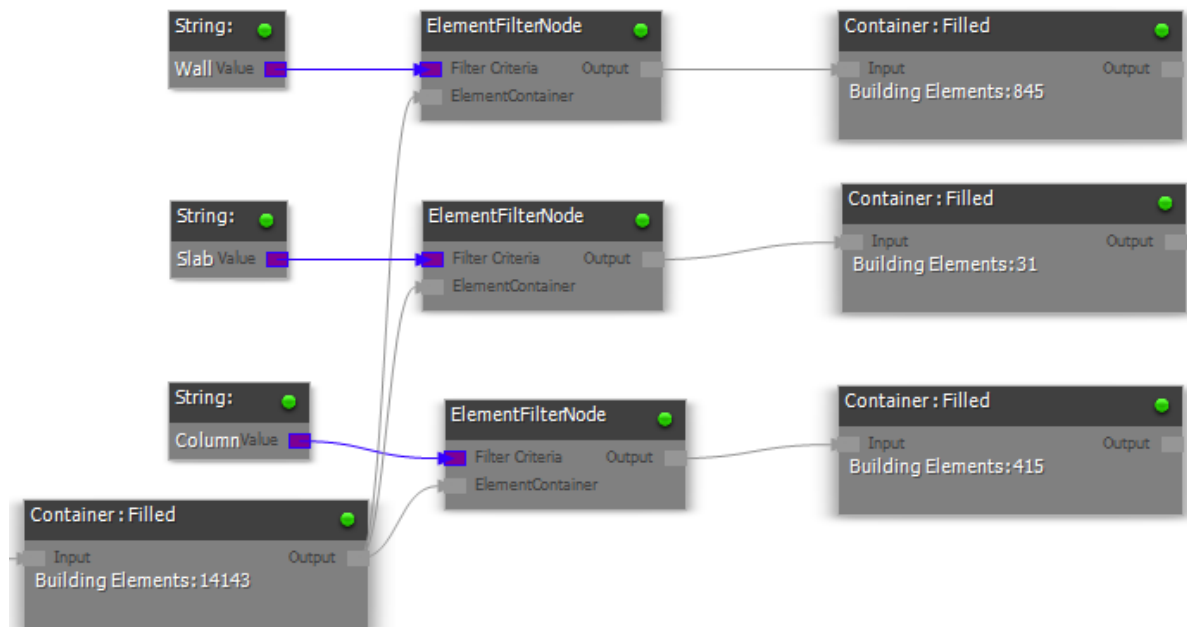


Abbildung 79: VCCL-Graph zur Filterung verschiedener Bauteiltypen

Die Visualisierung der einzelnen Objektknoten in dem obigen Beispiel ist in Abbildung 80 dargestellt.

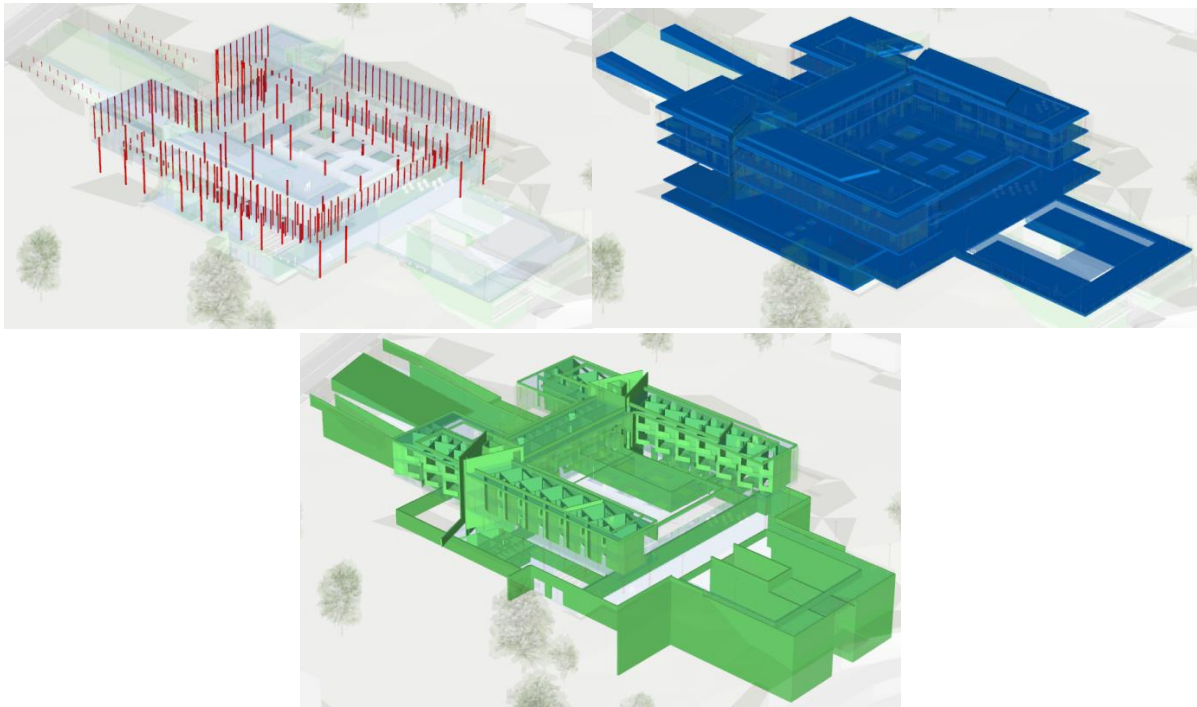


Abbildung 80: Visualisiertes Ergebnis der Filteroperation in bim+
oben links: Säulenbauteile, oben rechts: Deckenbauteile, unten: Wandbauteile

Da der *VCCL*-Graph keine Daten speichert, sondern lediglich die Struktur eines Verarbeitungssystems und den Informationsfluss des Prozesses abbildet, lässt sich dieser sehr leicht in Form einer Auszeichnungssprache speichern. Daher wurde eine Speicher- und Ladefunktionalität für *VCCL*-Graphen innerhalb des *Code Builder* entwickelt. In Code 24 ist der *VCCL*-Graph aus Abbildung 78 in der abgewandelten Auszeichnungssprache **.xng* beispielhaft dargestellt. In diesem Format werden zunächst die enthaltenen Knotenobjekte und anschließend deren Verknüpfung über die Schnittstellen festgehalten.

Code 24: Darstellung eines *VCCL*-Graphen in einer Auszeichnungssprache

```

1  <NodeGraphControl>
2    <NodeGraphControl.NodeGraphView>
3      <NodeGraphNodeCollection>
4        <NodeGraphLayoutEdit.FloatConstNode Name="Float: 100"/>
5        <NodeGraphLayoutEdit.FloatConstNode Name="Float: 100"/>
6        <NodeGraphLayoutEdit.AdditionNode Name="A+B" />
7        <NodeGraphLayoutEdit.SpyNode Name="SpyNode: />
8      </NodeGraphNodeCollection>
9      <NodeGraphLinkCollection>
10     <NodeGraphControl.NodeGraphLink InputNodeId="2" OutputNodeId="3"/>
11     <NodeGraphControl.NodeGraphLink InputNodeId="0" OutputNodeId="2"/>
12     <NodeGraphControl.NodeGraphLink InputNodeId="1" OutputNodeId="2"/>
13   </NodeGraphLinkCollection>
14 </NodeGraphControl.NodeGraphView>
15 </NodeGraphControl>

```

5.2.2 Abbildung einer Norm mit Hilfe des Code Builder

Mit diesen vorgestellten Elementen der *VCCL* soll nun die Tragfähigkeit der visuellen Sprache durch die Umsetzung des Überprüfungsprozesses gemäß den Vorschriften der *DIN 18232-2:2007-11*, welche bereits in Abschnitt 4.3 vorgestellt wurde, nachgewiesen werden. Die Definition der Knotentypen (siehe Abbildung 75) wurde darauf ausgelegt, dass dieses Regelwerk im Folgenden abgebildet werden kann.

An dieser Stelle sollen jedoch zunächst noch Knotentypen, welche für die Umsetzung der Norm von besonderer Bedeutung sind, näher erläutert werden.

Wie bereits in Abschnitt 4.3 beschrieben handelt es sich bei den Anforderungen der *DIN 18232-2:2007-11* um raumbezogene Vorschriften, welche die Gliederung des Gebäudemodells in eine Raumstruktur zur Voraussetzung haben. Diese Information kann – sofern diese in dem Gebäudemodell definiert wurde – aus dem Datenmodell von *bim+* abgerufen werden und ist in Abbildung 81 für ein Beispielgebäude dargestellt.

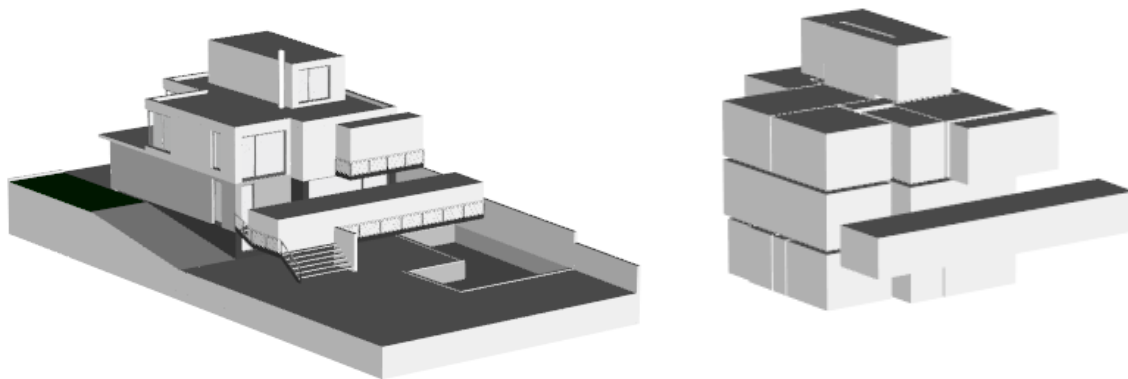


Abbildung 81: Visualisierung der Raumstruktur in *bim+*

In einem ersten Schritt soll zunächst ein einzelner Raum des gesamten Gebäudemodells für die Anwendung der Norm ausgewählt werden. In der Vorstellung des Konzepts der *VCCL* wurde davon ausgegangen, dass ein Raumobjekt eindeutig durch ein Attribut identifiziert werden und somit über eine Operation aus dem Gebäudedatenmodell gefiltert werden kann. Da dieses in dem Modell von *bim+* aufgrund des mangelnden Attributs auf diese Weise nicht möglich ist, wird dieser Prozessschritt mit Hilfe einer Kombination aus einer Filter- und einer Iterationsoperation bewerkstelligt, welche als *VCCL*-Graph in Abbildung 82 dargestellt ist.

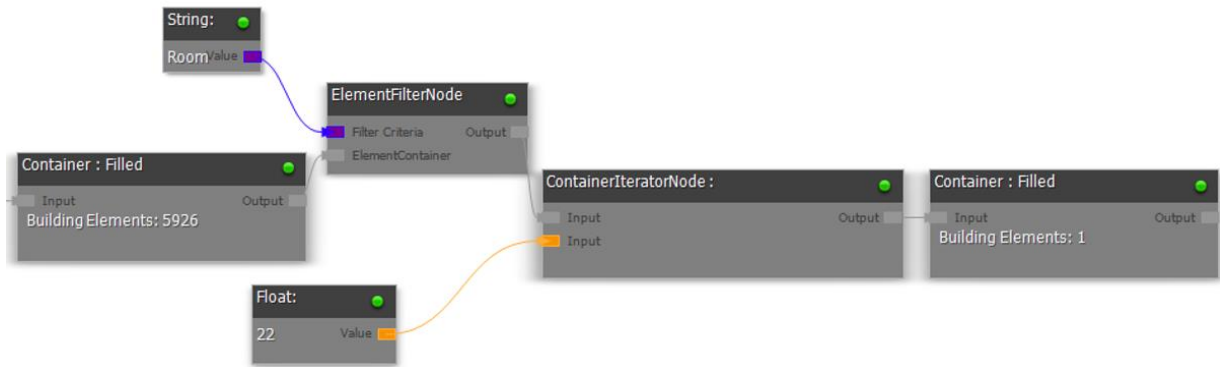


Abbildung 82: Auswahl eines einzelnen Raumes im Gebäudemodell mittels VCCL

Nach dem Filtern aller Raumobjekte aus dem Gebäudedatenmodell kann mit Hilfe des Knotens *ContainerIterator* (siehe Abbildung 75) ein einzelnes Objekt über seine Position in der Bauteilliste des *Container* gewählt werden. Um zu kontrollieren, welches Raummodell ausgewählt wurde, ist es möglich dieses wie in Abbildung 83 dargestellt zur Kontrolle zu visualisieren.

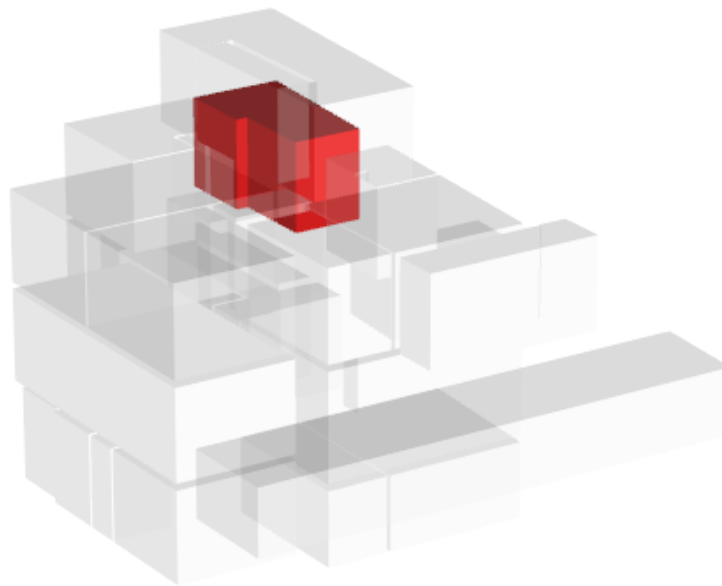


Abbildung 83: Visualisierung des betroffenen Raumes

Die zugehörigen Öffnungsflächen dieses Raumes für die Berechnung des Ist-Wertes der Rauchabzugsfläche erhält man, indem die geometrische Beziehung zwischen den Bauteilen und dem einzelnen Raummodell definiert wird. Voraussetzung hierfür ist die Identifikation, aller Bauteile, welche dem Raumobjekt zugeordnet sind. Diese Zuordnung ist im Gegensatz zu dem fiktiven Gebäudemodell, welches in Abschnitt 4.2.3 als theoretische Grundlage der *VCCL* vorgestellt wurde, in dem Datenmodell von *bim+* nicht enthalten. Daher muss an dieser Stelle diese Beziehung über einen geometrischen Operator berechnet werden.

Wie in Abbildung 84 dargestellt wurde im *Code Builder* ein allgemeingültiger geometrischer Operator-Knoten definiert, welcher das geometrische Verhältnis zwischen zwei beliebigen Listen von Bauteilen identifizieren kann.

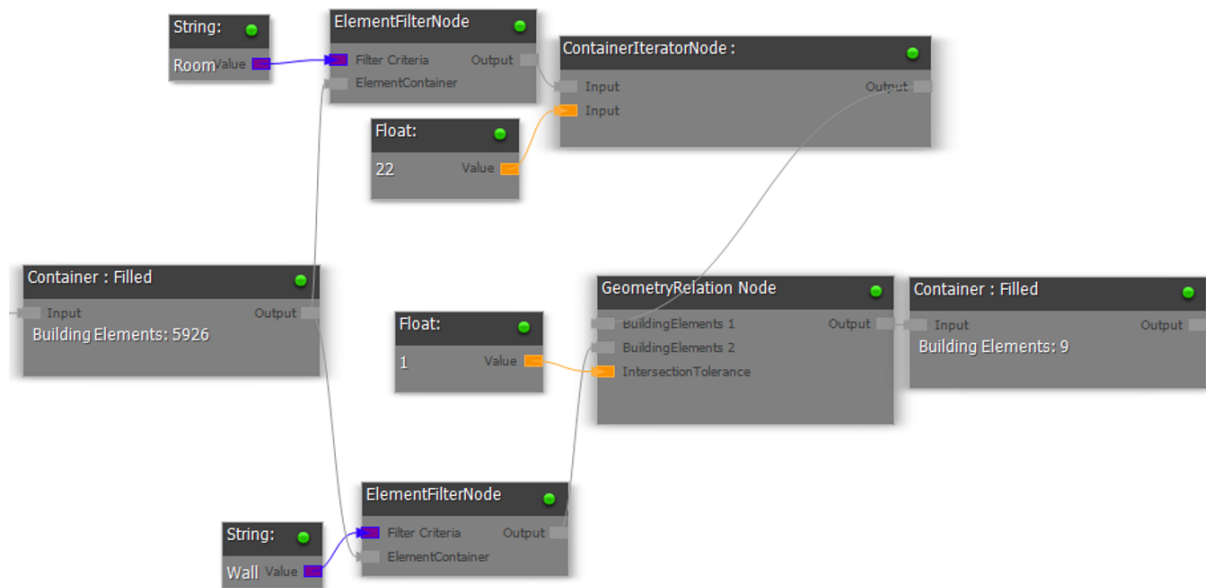


Abbildung 84: VCCL-Graph zur Berechnung von geometrischen Relationen

Innerhalb des Verarbeitungsprozesses dieses geometrischen Operators werden die *BREP*-Daten einzelner Bauteile abgerufen. Gleichzeitig wird überprüft, ob sich diese an einer bestimmten Stelle schneiden oder berühren. Sollte dieses der Fall sein, besteht eine geometrisch-topologische Beziehung zwischen den beiden Objekten und als Ergebnis werden alle positiv getesteten Bauteile über die Output-Schnittstelle ausgegeben. Das Ergebnis in dem resultierenden Objektknoten kann anschließend zur Kontrolle visualisiert werden. In Abbildung 85 ist das Ergebnis der Überprüfung der geometrischen Abhängigkeit eines Raumes mit allen Wandbauteilen des Gebäudemodells dargestellt.

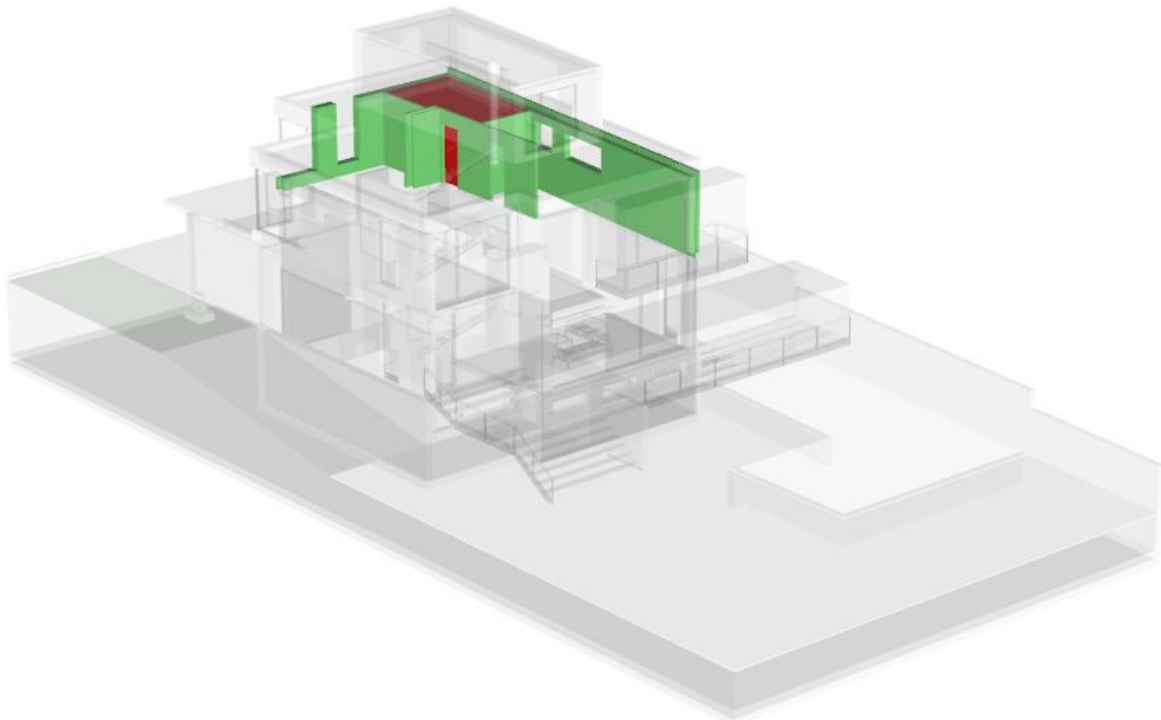


Abbildung 85: Ergebnis des geometrischen Operators für ein Raumobjekt und die Wandbauteile des Gebäudemodells

Ein weiteres wichtiges Element für den Überprüfungsprozess der *DIN 18232-2:2007-11* ist die Abbildung des Informationsgehalts der Tabelle für die geforderte und notwendige Rauchabzugsfläche. Wie bereits in Abschnitt 4.2.1.3 beschrieben handelt es sich hierbei um einen gesondert definierten Knoten. Daher wurde innerhalb des *Code Builder* ein Ansatz gewählt, welcher eine externe Datentabelle in den *VCCL*-Graphen integriert. Der *DataTableNode* ist in der Lage, eine Tabelle im **.xlsx*-Format auszulesen und die einzelnen Parameter, welche für die Ermittlung des Ausgabewertes notwendig sind, als Inputschnittstellen anzunehmen. Auf diese Weise wird ein generischer Zugriff auf Datentabellen realisiert, bei welchem der Knoten flexibel auf die Eingaben des Nutzers reagiert und gleichzeitig in der eigenen Funktionsweise unangetastet bleibt.

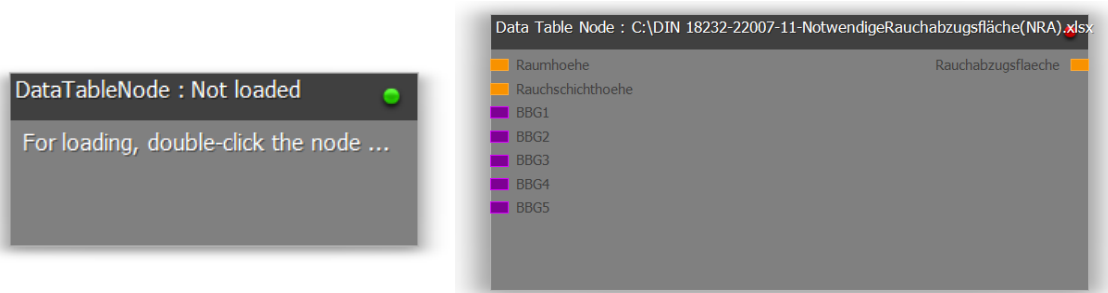


Abbildung 86: Datentabellenknoten im ungeladenen (links) und geladenen (rechts) Zustand

Das Prinzip hinter der Tabelle ist die Klassifizierung von Spalten und Reihen, so dass die Verarbeitungsfunktion des Datentabellenknoten die relevanten Grenzwerte an dem jeweiligen Ort in der Tabelle finden kann. Hierzu ist die Tabelle in zwei Teile gegliedert, welche einerseits die *Ports* des *DataTableNode* und andererseits die tabellarischen Ausgabewerte vorgeben.

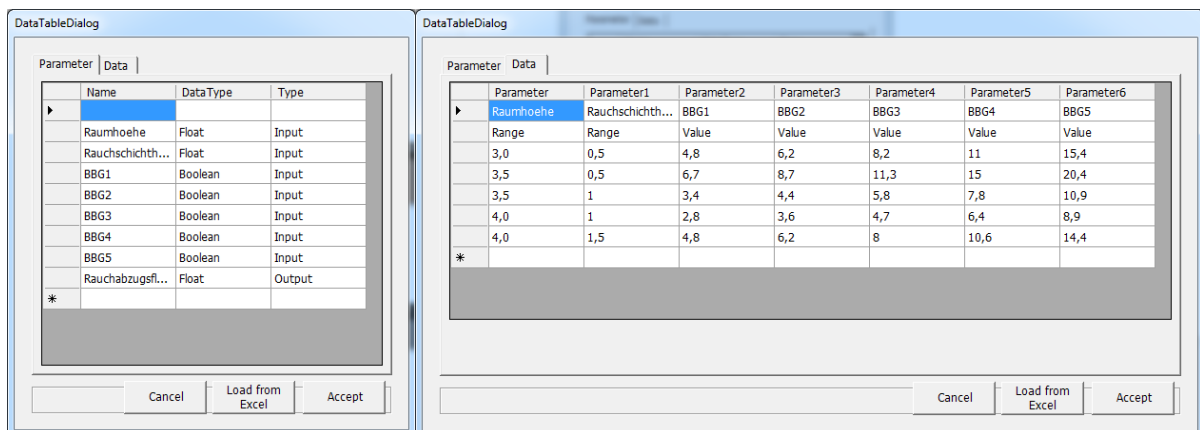


Abbildung 87: Abbildung einer Datentabelle in einem VCCL-Knoten

Möchte der Anwender eine eigene Datentabelle innerhalb eines solchen Knoten abbilden, muss er lediglich die Werte nach dem Schema der in Abbildung 87 dargestellten Tabellen eintragen und legt somit die Metadaten für den Knoten innerhalb des *VCCL*-Graphen fest.

In Abbildung 88 ist ein Beispiel für die Ausgabe der notwendigen Rauchabzugsfläche für konkrete Eingabewerte dargestellt. Wie bereits in Abschnitt 4.3 beschrieben, erfordert die Datentabelle neben der *Raumhöhe*, welche aus dem Gebäudemodell bezogen werden kann, die beiden Nutzereingaben *Rauchschiechthöhe* und *Brandschutzklasse*. In dem dargestellten

Beispiel kann die *Rauchschichthöhe* vom Anwender frei gewählt werden, die vorhandenen *Brandschutzklassen* hingegen müssen mit `Bool`-Werten gewählt werden, da diese gemäß der Norm auf 5 verschiedene Klassen begrenzt sind und dem Anwender auf diese Weise eine eindeutige Auswahl angeboten wird.

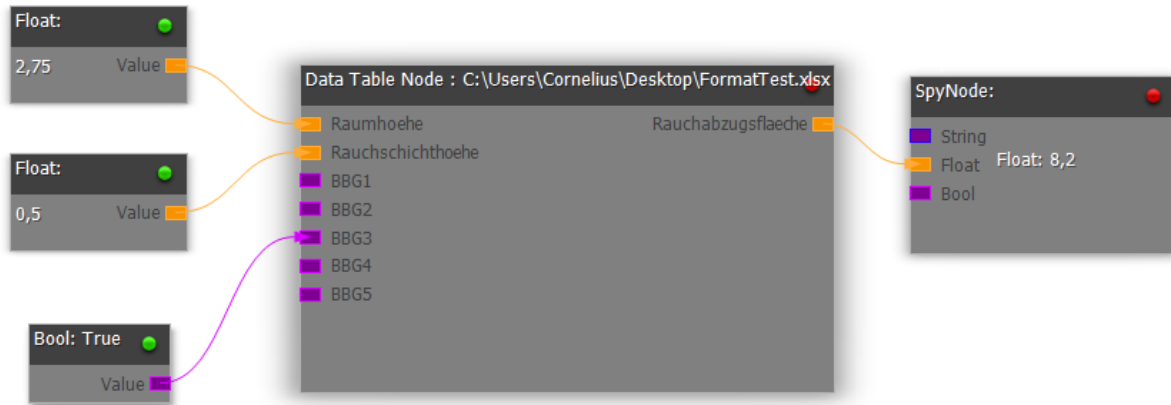


Abbildung 88: Ausgabe eines Grenzwerts vom Datentabellenknoten für beispielhafte Eingabedaten

Mit Hilfe der vorgestellten Elemente der *VCCL*, kann nun der gesamte Überprüfungsprozess gemäß der Richtlinien der *DIN 18232-2:2007-11* dargestellt werden (siehe Abbildung 89).

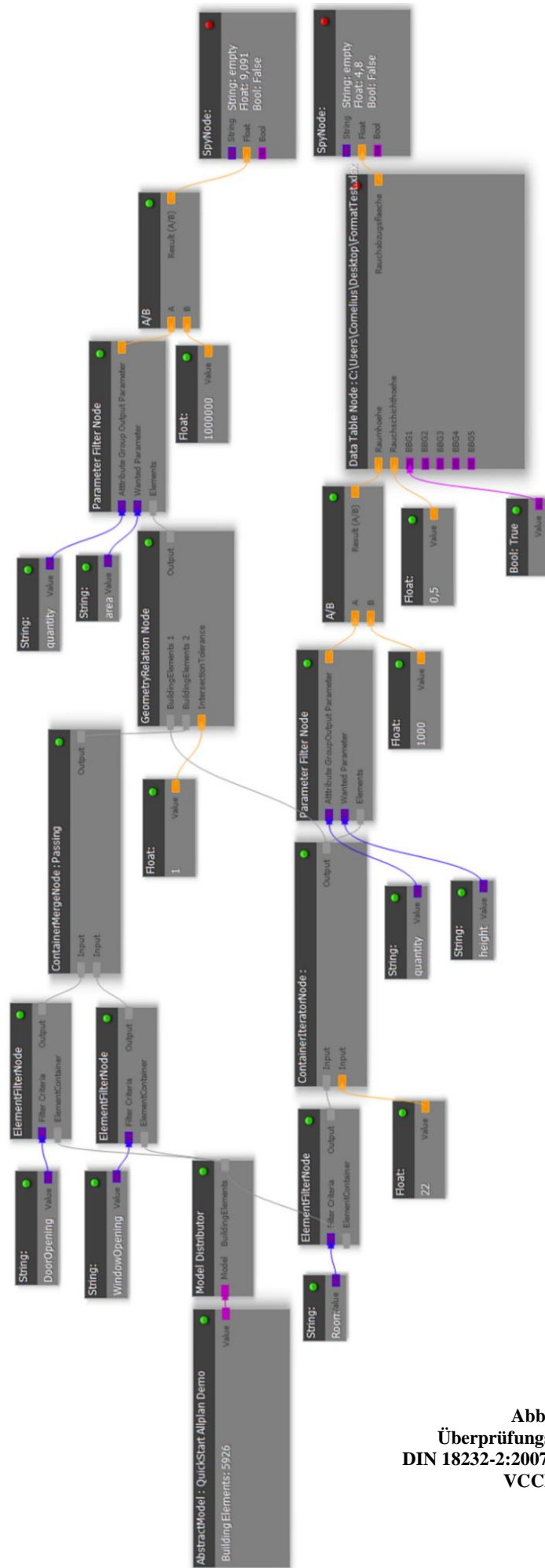


Abbildung 89:
Überprüfungsprozess gemäß der
DIN 18232-2:2007-11 dargestellt in einem
VCCL-Graphen

Das textuelle Ergebnis der Überprüfung kann direkt im Graphen an dem Endknoten ausgelesen werden. Bisher ist eine weitere Aufbereitung der Ergebnisse, z.B. in Form eines Nachweisdokumentes, noch nicht implementiert.

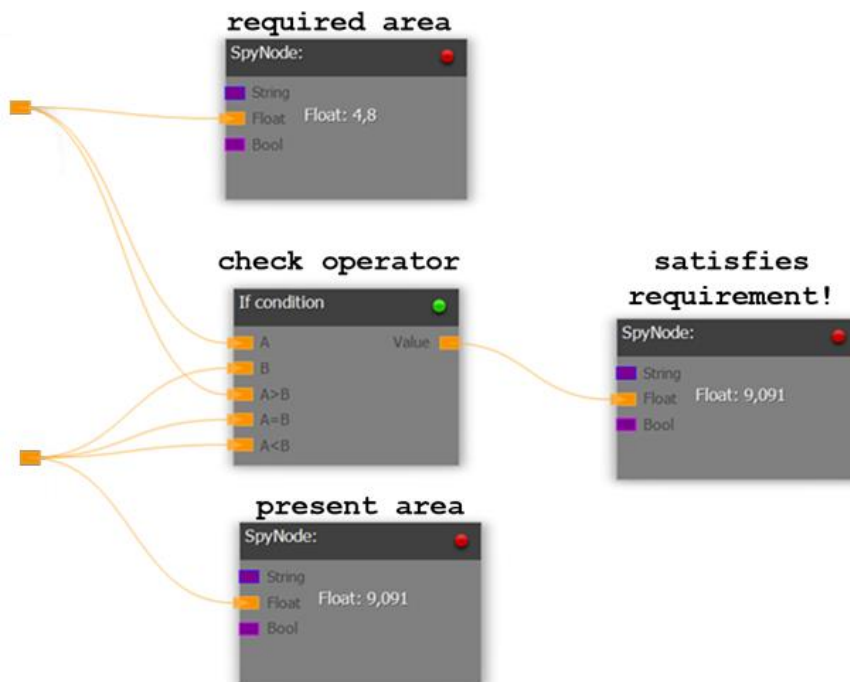


Abbildung 90: Interpretation des Ergebnisses des Überprüfungsprozesses im VCCL-Graphen

Darüber hinaus kann das Ergebnis der Überprüfung für den betroffenen Raum, wie in Abbildung 91 dargestellt, visualisiert werden. Hier sind sämtliche Bauteile zu sehen, welche zu dem betroffenen Raum gehören und eine Öffnungsfläche beinhalten, die an den Raum grenzt.

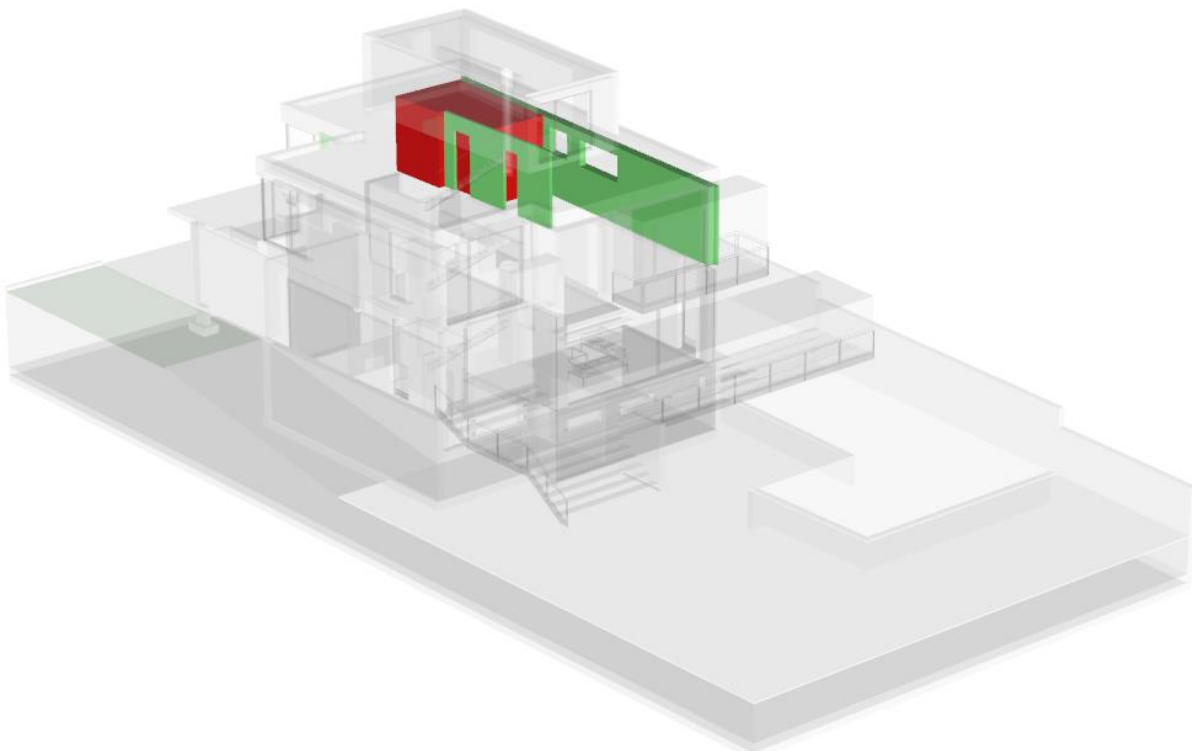


Abbildung 91: Visualisierung des Ergebnisses des Überprüfungsprozesses

Um nun den gleichen Überprüfungsprozess für einen anderen Raum durchzuführen, muss lediglich der betroffene Raum im Graphen und somit auch im Gebäudemodell gewechselt werden. Dieses kann wiederum mit Hilfe des *ContainerIterator* bewerkstelligt werden, welcher bereits bei der Auswahl des Raummodells vorgestellt wurde. Bisher wurde eine automatische Iteration einer Liste von Objekten innerhalb des *Code Builder* noch nicht umgesetzt.

Im Prinzip kann jedoch auf diese Weise jeder einzelne Raum des Gebäudedatenmodells vollautomatisch überprüft werden. In Abbildung 92 ist beispielhaft das Ergebnis für einen weiteren Raum des Gebäudemodells visualisiert.

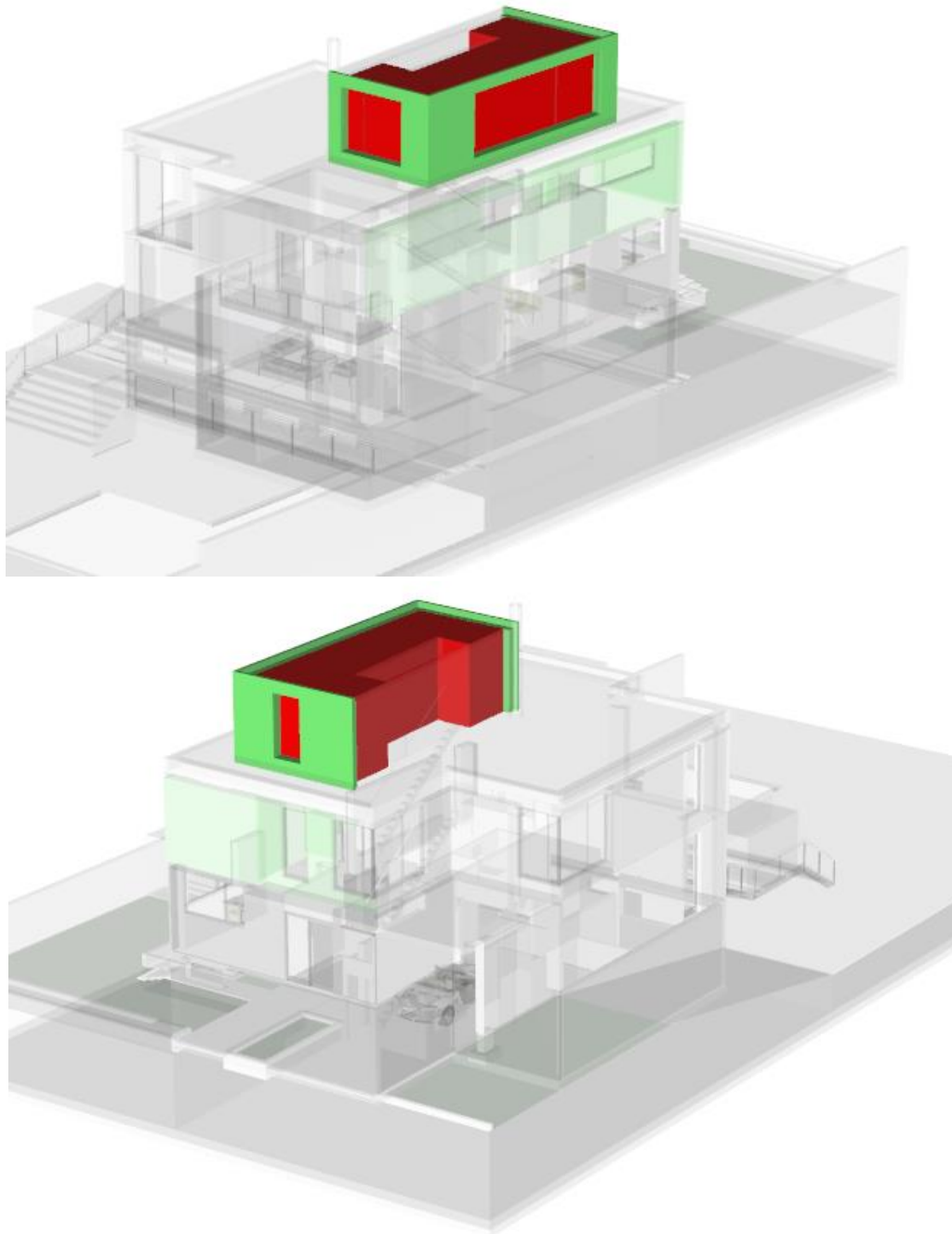


Abbildung 92: Visualisierung des Ergebnisses des Überprüfungsprozesses für einen weiteren Raum

6 Zusammenfassung und Ausblick

Mit dem Überblick zu den bisherigen Ansätzen des *Automated Code Compliance Checking* wird in der vorliegenden Arbeit die außerordentliche Relevanz und Bedeutung der Automatisierung der Konformitätsüberprüfung von Regelwerk und Gebäudedatenmodell für die Bauindustrie aufgezeigt. Auf Grundlage der vorgestellten Methoden kann eine gemeinsame Struktur des Überprüfungsprozesses identifiziert sowie eine Untersuchung der Vor- und Nachteile der Ansätze für jeden einzelnen Prozessschritt durchgeführt werden. Bei einer nachfolgenden Betrachtung wird eine Vielzahl von Anforderungen formuliert, welche als Basis für die Entwicklung einer neuen Methode zum *Automated Code Compliance Checking*, die insbesondere auch von Anwendern ohne fundierte Programmierkenntnisse genutzt werden kann, dienen.

Ein zentrales Kriterium für die Qualität eines Überprüfungsprozesses ist die Ausgestaltung der Mensch-Maschine-Kommunikation, also das Maß der Integration des Anwenders in den Überprüfungsprozess. Da viele der bislang entwickelten und angewendeten Methoden genau an dieser Stelle unzulänglich sind, wird in der vorliegenden Arbeit eine visuelle Sprache eingeführt, welche in den vergangenen Jahrzehnten bereits in anderen Fachbereichen erfolgreich zur Verbesserung der Informationsübermittlung zwischen Mensch und Maschine eingesetzt wurde. Hierbei wird der Vorteil genutzt, dass visuell dargestellte Prozeduren aufgrund der Prozessverarbeitung im menschlichen Gehirn vom Anwender effektiver und schneller als textuelle Informationen verarbeitet werden. In diesem Ansatz liegt ein großes Potential, das zur Optimierung und Effizienzsteigerung des Überprüfungsprozesses ausgeschöpft werden kann.

In der vorliegenden Arbeit wird die sogenannte *Visual Code Checking Language (VCCL)* vorgestellt, für welche eine Grammatik und Syntax speziell für den Anwendungsbereich der automatisierten Konformitätsüberprüfung formuliert wird. Mit der praktischen Umsetzung der *VCCL* und der erfolgreichen Überprüfung eines geltenden Regelwerks innerhalb des *Plug-In Code Builder* kann die Tragfähigkeit dieses Ansatzes nachgewiesen werden.

In der Baupraxis gibt es eine Vielzahl von Regelwerken und Normen mit weiteren Darstellungsweisen von Informationen, die prinzipiell durch Elemente der *VCCL* abbildbar sind, welche jedoch noch entwickelt werden müssen. Daher ist eine weitgehende Analyse weiterer Regelwerke notwendig, um Methoden und Ausdrucksweisen zu entwickeln, welche diese Inhalte schließlich in der Knoten-Bibliothek der *VCCL* erfassen.

Durch die Einführung einer visuellen Sprache in den Bereich des *Code Compliance Checking* wird die Grundlage eines neuen Genres für eine automatisierte Umsetzung von Prozessen des Bauwesens geschaffen. Diese Vorgehensweise eröffnet eine Vielzahl von Möglichkeiten für Entwicklungen in weiteren Bereichen des Bauwesens.

Nicht nur Normen, sondern alle Vorschriften aus den verschiedenen Anwendungsbereichen des Bauwesens können mit Hilfe der *VCCL* erfasst werden. So kann beispielsweise auch die Automatisierung von Kalkulationsprozessen realisiert werden, indem die Regelungen für die

Mengenermittlung bei einer Kostenrechnung von Bauprojekten innerhalb der *VCCL* abgebildet werden.

Wie die verschiedenen Ansätze zu der *VPL* in Abschnitt 4.1.1 zeigen, eignet sich eine visuelle Sprache insbesondere als Steuerungs-, Modifikations- und Zugriffswerkzeug von Verarbeitungssystemen. Da die *VCCL* sehr eng mit dem Gebäudemodell interagiert, bieten sich auch in diesem Bereich weitere Einsatzmöglichkeiten. Eine Integration von *VCCL*-Verarbeitungsprozessen kann dem Gebäudedatenmodell die Eigenschaft der dynamischen Zeichengebung verleihen und damit ein Reaktionsvermögen des Datenmodells auf äußere Einflüsse ermöglichen. Dieses wäre ein erster Schritt in Richtung eines intelligenten Gebäudemodells.

Über einen *VCCL*-Graphen könnte beispielsweise eine automatisierte Kontrolle des Informationsgehalts eines Gebäudemodells abgebildet werden, die automatisch gestartet wird, sobald eine Änderung an dem Datenmodell vorgenommen wird. Auf Grundlage einer umfassenden Bibliothek von *VCCL*-Knoten und vorgefertigten Graphen könnten diese Kontrollprozesse sehr einfach vom Anwender erstellt und bei Bedarf angepasst werden.

Zusammenfassend lässt sich festhalten, dass die Entwicklung der *VCCL* einen Schritt in Richtung der Automatisierung vieler Prozesse des Bauwesens darstellt. Um den Rahmen dieser Möglichkeiten und Grenzen zu definieren, sind weitere Analysen und vor allem ein Ausbau der theoretischen Grundlagen der *VCCL* erforderlich.

A Abkürzungsverzeichnis

AEC	Architecture, Engineering and Construction
AISC	American Institute of Steel Construction
BBR	Bundesamt für Bauwesen und Raumordnung
BCA	Building Construction Authority
BERA	Building Environment Rule and Analysis
BIM	Building Information Model(ing)
BOM	BERA Object Model
BREP	Boundary Representation
C#	Objektorientierte Programmiersprache
C++	Objektorientierte Programmiersprache
C3R	Conformance Checking in Construction with the help of Reasoning
CAD	Computer Aided Design
DAT	Design Assessment Tool
EDM	Express Data Manager
EN	Europäische Normen
F&E	Forschung und Entwicklung
FCA	Fire-Code Analyzer
GSA	U. S. General Service Administration
ICC	International Code Council
IFC	Industry Foundation Classes
ISO	International Organization for Standardization
Java	Objektorientierte Programmiersprache
LINQ	Language Integrate Query
LSC	Life Safety Code
ML	Markup Language
NIST	National Institute of Standards and Technology
OCL	Object Constraints Language
OWL	Ontology Web Language
PMQL	Partial Model Query Language
RDF	Resource Description Framework
SASE	Standards Analysis, Synthesis and Expression
SICAD	Standards Interface for Computer Aided Design
SMC	Solibri Model Checker
SPARQL	Protocol And RDF Query Language
SPEX	Standards Processing Expert
SQL	Structured Query Language
STEP	Standard for the exchange of product model data
UI	User Interface
VCCL	Visual Code Checking Language
VLCC	Visual Language Compiler
VPL	Visual Programming Language
WC3	World Wide Web Consortium
XML	Extensible Markup Language

B Anhang

Der vorliegenden Arbeit ist eine CD angefügt, welche die folgenden Informationen enthält:

- Video zur Präsentation der Funktionalitäten des *Code Builder*
- Aktueller Entwicklungsstand des *Code Builder* als Plug-In für den *bim+ Explorer*
- Digitale Version der vorliegenden Arbeit

C Literaturverzeichnis

- Adachi, Yoshinobu. „Overview of Partial Model Query Language (PMQL).“
Technical Research Center of Finland. 2002. <http://cic.vtt.fi/projects/ifcsvr/tec/VTT-TEC-ADA-12.pdf> (Zugriff am 17. 06 2014).
- Administrative Office of the U.S. Courts. „U.S. Courts Design Guide.“
U.S. General Services Administration. 2007.
http://www.gsa.gov/graphics/pbs/Courts_Design_Guide_07.pdf
(Zugriff am 24. 06 2014).
- AEC3. 2014. <http://www.aec3.com/> (Zugriff am 23. 06 2014).
- American Institute of Steel Construction. *AISC*. 2014. <https://www.aisc.org/>
(Zugriff am 28. 06 2014).
- Autodesk, Inc. 2014. <http://www.autodesk.de/> (Zugriff am 10. 07 2014).
- Becker, Jörg. *Die Digitalisierung von Medien und Kultur*. Wiesbaden: Springer, 2013.
- Beetz, Jakob, Jos van Leeuwen, und Bauke de Vries. „IfcOWL: A case of transforming EXPRESS schemas into ontologies.“ *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, Vol. 23, 02 2009: 89 - 101.
- bim+. 2013. <https://www.bimplus.net/> (Zugriff am 05. 12 2013).
- Building & Construction Authority Singapore. *CORENET*. 2006. <http://www.corenet.gov.sg/>
(Zugriff am 04. 06 2014).
- buildingSMART. *ifcXML*. 2014. <http://www.buildingsmart-tech.org/specifications/ifcxml-releases> (Zugriff am 30. 06 2014).
- buldingSMART. *Standards*. 2014. <http://www.buildingsmart.de/bim-know-how/standards>
(Zugriff am 16. 06 2014).
- CRC Construction Innovation. 2006. <http://www.construction-innovation.info/>
(Zugriff am 11. 06 2014).
- Delis, E. A., und A. Delis. „Automatic Fire-Code Checking using Expert-System Technology.“ *Journal of Computing in Civil Engineering*, Vol. 9, No. 2, April 1995: 141-156.
- Digital Alchemy. 2014. <http://www.digitalalchemy.com.au/> (Zugriff am 23. 06 2014).
- Dimyadi, Johannes, und Robert Amor. *Automated Building Code Compliance Checking - Where is it at?* Auckland, New Zealand: University of Auckland, 2013.
- DIN. *Deutsches Institut für Normung*. 2014. <http://www.din.de> (Zugriff am 16. 07 2014).

- Ding, L., R. Drogemuller, J. Jupp, M. Rosenman, und J. Gero. „Automated Code Checking.“ *CRC for Construction Innovation, Clients Driving Innovation International Conference, 25–27 October*. Surfers Paradise: Qld., 2004.
- Ding, Lan, Robin Drogemuller, Mike Rosenman, Mike Marchant, und John Gero. „Automating code checking for building designs - DesignCheck.“ *Clients Driving Innovation: Moving Ideas into Practice*. Wollongong: University of Wollongong, 2006. 1 - 16.
- dRofus. 2014. <http://www.drofus.no/> (Zugriff am 16. 06 2014).
- Dynamo. *DynamoBIM*. 2014. <http://dynamobim.org/> (Zugriff am 10. 07 2014).
- Eastman, C. „Automated Assessment of Early Concept Designs.“ In *Architectural Design, Vol. 79 Issue 2*, von Richard Garber, 52 - 57. UK: John Wiley & Sons, Ltd., 2009.
- Eastman, C., J. Lee, Y. Jeong, und J. Lee. „Automatic rule-based checking of building designs.“ *Automation in Construction, Vol.18 Issue 8*, 30. Juli 2009b: 1011 - 1033.
- Eberg, Ernst, Turi Heieraas, Jan Olsen, und Svein-Helge Eidissen. *Experiences in development and use of a digital Building Information Model (BIM) according to IFC standards from the building project of Tromsø University College (HITOS) after completed Full Conceptual Design Phase*. Report, Oslo: Statsbygg - Norwegian Agency of Public Construction and Property, 2006.
- Ebertshäuser, Sebastian , und Petra von Both. „ifcModelCheck - A Tool for configurable rule-based Model Checking.“ In *Computation and Performance - Proceedings of the 31st eCAADe Conference, Vol. 2*, 525-534. Delft, The Netherlands: Delft University of Technology, 2013.
- Furrer, Frank J. „Eine kurze Geschichte der Ontologie.“ *Informatik-Spektrum, Volume 37, Issue 4* , 30. August 2014: 308 - 317.
- Gallagher, M.P., A.C. O'Connor, J.L. Dettbarn, und L.T. Gilday. „Cost Analysis of Inadequate Interoperability in the U.S. Capital Facilities Industry. NIST.“ 2004.
- Garrett, James H., Mark E. Palmer, und Demir Salih. „Delivering the Infrastructure for Digital Building Regulations.“ *Journal of Computing in Civil Engineering, Vol. 28, Issue 2*, 2014: 167-169.
- gbXML. 2014. www.gbxml.org (Zugriff am 01. 07 2014).
- Genesereth, M., und N. Nillson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 1987.
- GIT College of Architecture. *GSA Preliminary Concept Design BIM Guide*. Georgia, USA: Georgia Insititue of Technology, 2008.
- Grasshopper3D. 2014. <http://www.grasshopper3d.com/> (Zugriff am 08. 07 2014).

- Gruber, T.R. „A translation approach to portable ontology specifications.“
Gaines B.R. & Boose J.H.: Knowledge Acquisition, Volume 5., 1993: 199 - 220.
- Guarino, N, und P Giarette. *Ontologies and Knowledge Bases: Towards a Terminological Clarification*. Padova, Italy: National Research Council, 1995.
- Hjelseth, E., und Nick Nisbet. „Capturing normative constraints by use of the semantic mark-up RASE methodology.“ *Proceedings of the CIB*. Sophia Antipolis, France, 2011.
- Holte Consulting. „ByggNett Status Survey.“ 31. 01 2014.
http://www.dibk.no/globalassets/byggnett/byggnett_rapporter/byggnett-status-survey.pdf (Zugriff am 26. 06 2014).
- International Code Council. 2014. <http://www.iccsafe.org/> (Zugriff am 23. 06 2014).
- ISO. „ICC/ANSI A117.1-2003 - Accessible and Useable Buildings and Facilities.“ 2003.
<http://webstore.ansi.org/RecordDetail.aspx?sku=ICC%2FANSI+A117.1-2003>.
- . „ISO/CD21542, Building construction - Accessibility and usability of built environment.“ 2011.
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50498.
- Joo-Sung, Lee, Min Kyung-Min , Kim Ju-Hyung , Lee Yoon-Sun , und Kim Jae-Jun.
„Building Ontology to implement the BIM focused on pre-design stage.“ *The 25th International Symposium on Automation and Robotics in Construction*. Vilnius, Lithuania: Institute of Internet and Intelligent Technologies, Vilnius University, 2008. 350 - 354.
- Jotne EPM Technology. *Express Data Manager, Vol. 1, Issue 6*. 2004.
<http://www2.epmtech.jotne.com/download/EDM47.pdf> (Zugriff am 07. 08 2014).
- Jotne IT. *About Jotne IT*. 2014. <http://www.epmtech.jotne.com/about-jotne-it>
(Zugriff am 04. 06 2014).
- Kammholz, Karsten. „So werden Baudesaster wie der BER künftig vermieden.“ *Die Welt*, 14. 05 14.05.2014: <http://www.welt.de/politik/deutschland/article127986675/So-werden-Baudesaster-wie-der-BER-kuenftig-vermieden.html>.
- Kost, Martin. „Grundlagen des Semantic Web: Web Ontology Language (OWL).“
<http://www.is.inf.uni-due.de/>. 2013. http://www.is.inf.uni-due.de/courses/db_ws04/fohlen/owl.pdf (Zugriff am 22. 06 2013).
- Lee, Jin Kook. *Building Environment Rule and Analysis (BERA) Language*. Georgia, USA: Georgia Institute of Technology, 2010.
- Liu, Ling, und Tamer Özsu. *Encyclopedia of Database Systems*. Atlanta, USA: Springer Science+Business Media, 2009.

- Lopez, L. A., und R. N. Wright. „Mapping Principles for the Standards interface for Computer Aided Design.“ 1985.
- Maloney, John, Mitchel Resnick, Natalie Rusk, Brian Silverman, und Evelyn Eastmond. „The Scratch Programming Language and Environment.“ In *ACM Transactions on Computing Education*, Vol. 10 Issue 4, Article No. 16. New York: ACM, 2010.
- McGraw Hill Construction. *SmartMarket Report - The Business Value of BIM*. Report, New York: McGraw-Hill Companies, 2009.
- McNeel. *Rhinoceros*. 2014. <http://www.rhino3d.com/> (Zugriff am 08. 07 2014).
- Mohus, F., O. K. Kvarvik, und M. Lie. „The HITOS project - A full scale IFC test.“ In *eWork and eBusiness in Architecture, Engineering and Construction*, von Martinez und Scherer, 191 - 195. London: Taylor & Francis Group, 2006.
- Myers, Brad A. „Taxonomies of Visual Programming.“ In *Journal of Visual Languages and Computing*, Vol. 1 Issue 1, 97 - 123. Pittsburgh, USA: Elsevier, 1990.
- National Fire Protection Association. *NFPA 101 - Life Safety Code*. USA, 2005.
- National Instruments. *LabVIEW*. 2014. <http://www.ni.com/labview/d/> (Zugriff am 09. 07 2014).
- Nawari, N. O. „Automating Codes Conformance.“ *Journal of Architecture Engineering*, Vol. 18, Issue 4, Dezember 2012: 315 - 323.
- Nawari, N. O. „The Challenge of Computerizing Building Codes in BIM Environment.“ *Computing in Civil Engineering*, Vol., 2012: 285-292.
- Nemetschek AG. *Nemetschek Historie: Die Anfänge*. 2013. http://www.nemetschek.com/home/unternehmen/historie/die_anfaenge.html (Zugriff am 05. 12 2013).
- Norwegian Building Authority. *Direktoratet for byggkvalitet*. 2014. <http://www.dibk.no/> (Zugriff am 26. 06 2014).
- Norwegian Statsbygg*. 2014. <http://www.statsbygg.no/> (Zugriff am 16. 06 2014).
- novaCITYNETS. 2014. <http://www.novacitynets.com/fornax/> (Zugriff am 11. 06 2014).
- OASIS. *LegalXML*. 2014. <http://www.legalxml.org/> (Zugriff am 29. 06 2014).
- Open CASCADE Technology. 2014. <http://www.opencascade.org/> (Zugriff am 11. 06 2014).
- Palmirani, M., G. Governatori, A. Rotolo, S. Tabet, H. Boley, und A. Paschke. „LegalRuleML: XML-Based Rules and Norms.“ In *Rule - Based Modeling and Computing on the Semantic Web, 5th International Symposium RuleML*, von M. Palmirani und D. Scottara, 298 - 312. Fort Lauderdale, USA: Springer, 2011.

- Pan, Jeff Z. „Resource Description Framework.“ In *Handbook of Ontologies*, von Steffen Staab und Rudi Studer, 71-90. Springer, 2009.
- Pau, L. F., und H. Olason. „Visual Logic Programming.“ In *Journal of Visual Languages and Computing, Vol. 2, 3* - 15. Elsevier, 1991.
- Paul, N., und A. Borrmann. „Using Geometrical and Topological Modeling Approaches in Building Information Modeling.“ In *Proceedings of ECPPM 2008*, von Alain Zarli und Raimar Schrer, 117 - 127. CRC Press, 2008.
- Pauwels, P., et al. „A semantic rule checking environment for building performance checking.“ *Automation in Construction, Vol. 20, Issue 5*, 13. November 2010: 506 - 518.
- RuleML. 2014. http://wiki.ruleml.org/index.php/RuleML_Home (Zugriff am 29. 06 2014).
- Salama, D., und N. El-Gohary. „Automated Compliance Checking of Construction Operation Plans using a Deontology for the Construction Design.“ *Journal of Computing in Civil Engineering, Vol. 27, Issue 6*, November / Dezember 2013: 681 - 689.
- Schenk, Joachim, und Gerhard Rigoll. *Mensch-Maschine-Kommunikation*. Wiesbaden: Springer, 2010.
- Schenke, Michael. *Logikkalküle in der Informatik*. Wiesbaden: Springer Vieweg, 2013.
- Schiffer, Stefan. *Visuelle Programmierung - Grundlagen und Einsatzmöglichkeiten*. Bonn : Addison-Wesley, 1998.
- Scholze-Stubenrecht, Werner. *Duden - Die deutsche Rechtschreibung*. Mannheim: Dudenverlag, 2004.
- Solibri. *Solibri Model Checker v9.1*. 05. 05 2014. <http://www.solibri.com/solibri-model-checker-v9-1-available-today/> (Zugriff am 11. 06 2014).
- Spatial Corp. 2014. <http://www.spatial.com/products/3d-acis-modeling> (Zugriff am 11. 06 2014).
- Staab, Steffen. „Grading Knowledge - Extracting Degree Information from Texts.“ In *Lecture Notes in Computer Science, Vol. 1744* , 169 - 170. Springer, 1999.
- Tang, X., A. Hammad, und P. Fazio. „Automated Compliance Checking for Building Envelope Design.“ *Journal of Computing in Civil Engineering, Vol 24, No. 2*, März / April 2010: 203 - 211.
- U. S. General Services Administration. 2014. <http://www.gsa.gov/> (Zugriff am 24. 06 2014).
- von Bertalanffy , Ludwig. „The History and Status of General Systems Theory.“ In *The Academy of Management Journal, Vol. 15 No. 4*, 407-426. 1972.
- WC3. *World Wide Web Consortium* . 2014. <http://www.w3.org/> (Zugriff am 25. 06 2014).

- Xu, Rong, W. Solihin, und Zhiyong Huang. „Code Checking and Visualization of an Architecture Design.“ *Visualization*. Austin, TX, USA: IEEE, 2004.
- Yurchyshyna, A, und A. Zarli. „An ontology-based approach for formalisation and semantic organisation of conformance requirements in construction.“ In *Automation in Construction. Volume 18*, von M. Skibniewski, 1084 - 1098. Elsevier, 2009.
- Yurchyshyna, A., C. Faron-Zucker, N. Thanh, und A. Zarli. „Towards an Ontology-enabled Approach for Modeling the Process of Conformity Checing in Construction.“ *Proceedings of the Forum at the CAiSE'08 conference*. Montpellier, France: University of Montpellier, 2008. 21 - 24.
- Zhang, Le, und R.A. Issa Raja. *Developement of IFC-based Construction Industry Ontology fro Information Retrieval from IFC Models*. Florida, USA: University of Florida, 2010.

D Abbildungsverzeichnis

Abbildung 1: Chronologische Entwicklung des Code Compliance Checking (Dimiyadi und Amor 2013)	10
Abbildung 2: Formale Symbole der Aussagenlogik (Schenke 2013)	11
Abbildung 3: Entscheidungstabelle zur Überprüfung des Wärmedurchgangswiderstandes einer Außenwand in Toronto (Tang, Hammad und Fazio 2010).....	13
Abbildung 4: Datenobjekt Tür mit den zugehörigen Parametern (Source = „INP“) und geometrischen Algorithmen (Source = „CALC“) im FCA (Delis und Delis 1995).....	15
Abbildung 5: Beispiel für eine Übersetzung einer Vorschrift mit der Syntax der deontischen Logik (Salama und El-Gohary 2013)	16
Abbildung 6: Hierarchisches Modell für das Element „Norm“ (Salama und El-Gohary 2013)	17
Abbildung 7: Deontisches Modell des Überprüfungsprozesses (Salama und El-Gohary 2013)	17
Abbildung 8: Ablauf einer (Salama und El-Gohary 2013)	18
Abbildung 9: Schematische Darstellung der Ergebnisse der Suchanfrage mit PMQL (Adachi 2002)	21
Abbildung 10: Softwarekonzept für die Abbildung der Inhalte eines Regelwerkes mit Hilfe von XML (Ebertshäuser und von Both 2013)	22
Abbildung 11: Struktur eines Überprüfungsprozesses im SMARTCodes-System (Eastman, Lee, et al. 2009b)	23
Abbildung 12: Verknüpfung zwischen LINQ, SMARTCodes und Gebäudemodell (Nawari, Automating Codes Conformance 2012)	24
Abbildung 13: Klassifizierung von BERA als Programmiersprache (Lee 2010)	25
Abbildung 14: Einordnung von BERA in die unterschiedlichen Entwicklungsstufen der automatisierten Konformitätsüberprüfung (Lee 2010).....	26
Abbildung 15: Abstraktion räumlicher Informationen des IFC-Datenmodells innerhalb des BOM (Lee 2010)	26
Abbildung 16: Integration des BERA Language Tool in der Umgebung des SMC (Lee 2010).....	28
Abbildung 17: Ergebnis der Raumanalyse mit BERA (Lee 2010)	29
Abbildung 18: Ergebnis der Laufwegeanalyse mit BERA (Lee 2010).....	30
Abbildung 19: Auflistung der verschiedenen logischer Familien (Salama und El-Gohary 2013)	30
Abbildung 20: Auflistung logischer Operatoren der OWL Description Logic (Joo-Sung, et al. 2008).....	32
Abbildung 21: Vereinfachtes ontologisches Klassenmodell für ein Gebäudemodell im IFC-Format (Joo-Sung, et al. 2008).....	33
Abbildung 22: Randbedingungen für die Gestaltungsplanung eines Grundrisses (Joo-Sung, et al. 2008).....	34

Abbildung 23: Softwarearchitektur für einen Datenzugriff auf das ontologische Gebäudemodell (Zhang und Raja 2010).....	34
Abbildung 24: Klassenhierarchie des IFC-Datenmodells für IfcOWL (Beetz, van Leeuwen und de Vries 2009).....	36
Abbildung 25: Schematische Abbildung des IFC auf IfcOWL (Beetz, van Leeuwen und de Vries 2009).....	37
Abbildung 26: Schematischer Ablauf in C3R (Yurchyshyna und Zarli 2009).....	39
Abbildung 27: Beispiel für die Funktionsweise eines FORNAX-Objektes (Eastman, Lee, et al. 2009b).....	41
Abbildung 28: Struktur eines CORENET-Projektes (Eastman, Lee, et al. 2009b).....	41
Abbildung 29: Aufbau des EDM (Jotne EPM Technology 2004).....	42
Abbildung 30: Pseudocode-Darstellung einer Norm für DesignCheck (L. Ding, et al. 2006).....	43
Abbildung 31: Graphische Unterstützung der Formulierung von Algorithmen in DesignCheck (Eastman, Lee, et al. 2009b).....	44
Abbildung 32: Aufbau von DesignCheck (L. Ding, et al. 2004).....	45
Abbildung 33: Abschlussbericht des Überprüfungsprozesses in DesignCheck (L. Ding, et al. 2006).....	45
Abbildung 34: Regelsätze-Manager in SMC.....	46
Abbildung 35: Fluchtwegüberprüfung in SMC.....	47
Abbildung 36: Schematischer Ablauf des Preliminary-Design-Tool der GSA (Eastman 2009).....	48
Abbildung 37: Zuordnung der Bezeichnungen von Räumen zur Vorbereitung der jeweiligen Überprüfung (Eastman 2009).....	49
Abbildung 38: Beispiel für eine Richtlinie des Raumkonzeptes der GSA (Eastman 2009).....	49
Abbildung 39: Relationen-Modell der Räume (Eastman 2009).....	50
Abbildung 40: links – Visualisierung der Laufwege im Raummodell rechts – Visualisierung einer fehlerhaften Laufroute (Eastman, Lee, et al. 2009b).....	50
Abbildung 41: Skizze für den Neubau der Universität Tromsø (Mohus, Kvarsik und Lie 2006).....	51
Abbildung 42: Code Compliance Checking im HITOS Projekt (Eastman, Lee, et al. 2009b).....	52
Abbildung 43: oben - Userinterface von dRofus unten – Definition von Anforderungen über Raum-Templates (dRofus 2014).....	53
Abbildung 44: Eingabe der Parameter für die Überprüfung der Barrierefreiheit mit dem SMC auf der HITOS-Plattform (Eastman, Lee, et al. 2009b).....	54
Abbildung 45: Ergebnisse einer Überprüfung zu Barrierefreiheit mit dem SMC auf der HITOS-Plattform links – Überprüfung der Zugänglichkeit von Räumen zentral – Überlappung der Türblätter beim Öffnen rechts – Wenderadius eines Rollstuhlfahrers in einem Raum (Eastman, Lee, et al. 2009b).....	54

Abbildung 46: Prozessschritte einer Konformitätsüberprüfung (Eastman, Lee, et al. 2009b)	56
Abbildung 47: Analyse des Satzes "Bitte versteh mich doch", gesprochen von Mensch (oben) und Maschine (unten), mittels eines Sonagramm (Becker 2013)	58
Abbildung 48: Ergebnis einer Umfrage zu der mangelnden Qualität von Gebäudemodellen (Ebertshäuser und von Both 2013)	59
Abbildung 49: Schematische Darstellung einer Black-Box- und White-Box-Methode	61
Abbildung 50: Pygmalion Userinterface für Fakultätsfunktionen (Schiffer 1998).....	63
Abbildung 51: links: graphische Eingabe des Blockdiagramms in LabVIEW rechts: resultierendes Userinterface für die Auswertung in LabVIEW (Myers 1990)	63
Abbildung 52: rechts: Softwarearchitektur der Visual Logic Programming links: Graphischer Editor der Visual Logic Programming (Pau und Olason 1991)	64
Abbildung 53: Userinterface von Scratch (Maloney, et al. 2010)	64
Abbildung 54: oben: Zeichenfläche für die Erstellung von Randbedingungen in Grasshopper unten: Beispiel-Geometrie, die in Rhinoceros mit Grasshopper erstellt wurde	65
Abbildung 55: VPL-Werkzeug Dynamo in Autodesk Revit (Dynamo 2014)	66
Abbildung 56: Visuelle Dichtung im Gedicht »Schweigen« von Eugen Gomringer, 1969 (Schiffer 1998).....	66
Abbildung 57: Veranschaulichung der Komplexität eines visuellen Systems nach Hoare (Schiffer 1998).....	67
Abbildung 58: links: Grammatik: des VLCC rechts oben: Grammatikeditor des VLCC rechts unten: Diagrammeditor des VLCC (Schiffer 1998).....	69
Abbildung 59: Datentypen der VCCL	72
Abbildung 60: Schematische Darstellung einiger geometrischer Verhältnisse zwischen zwei geometrischen Objekten (Paul und Borrmann 2008).....	80
Abbildung 61: Schematische Darstellung der Wärmefreihaltung [DIN 18232-2:2007-11].....	86
Abbildung 62: Ausschnitt der Tabelle für die notwendige Rauchabzugsfläche A_w in m^2 je Rauchabschnitt [DIN 18232-2:2007-11]	88
Abbildung 63: Korrekturfaktor c_z für unterschiedliche Öffnungswinkel von Öffnungsflächen [DIN 18232-2:2007-11].....	89
Abbildung 64: Firmenlogo von bim+ (2013).....	92
Abbildung 65: Struktur der bim+-Plattform (bim+ 2013)	93
Abbildung 66: Projekt- und Modellauswahl von bim+ ein einem Web-Browser.....	94
Abbildung 67: Teamverwaltung für ein Modell in bim+.....	94
Abbildung 68: Web-Viewer von bim+ im Browser.....	95
Abbildung 69: Technische Struktur der bim+-Plattform (bim+ 2013)	96
Abbildung 70: API-Services von bim+	96
Abbildung 71: bim+ Appstore.....	97
Abbildung 72: 3D-Viewer im bim+ Explorer	99
Abbildung 73: Detailansicht im bim+ Explorer.....	99

Abbildung 74: Userinterface des Code Builder im bim+ Explorer.....	100
Abbildung 75: Knotenobjekte des Code Builder	101
Abbildung 76: Darstellung eines Objektknoten in unterschiedlichen Zuständen in Code Builder	102
Abbildung 77: Datentypen in Code Builder	102
Abbildung 78: Visuelle Darstellung einer einfachen Rechenoperation im Code Builder	103
Abbildung 79: VCCL-Graph zur Filterung verschiedener Bauteiltypen	104
Abbildung 80: Visualisiertes Ergebnis der Filteroperation in bim+ oben links: Säulenbauteile, oben rechts: Deckenbauteile, unten: Wandbauteile	105
Abbildung 81: Visualisierung der Raumstruktur in bim+.....	106
Abbildung 82: Auswahl eines einzelnen Raumes im Gebäudemodell mittels VCCL.....	107
Abbildung 83: Visualisierung des betroffenen Raumes.....	107
Abbildung 84: VCCL-Graph zur Berechnung von geometrischen Relationen.....	108
Abbildung 85: Ergebnis des geometrischen Operators für ein Raumobjekt und die Wandbauteile des Gebäudemodells.....	108
Abbildung 86: Datentabellenknoten im ungeladenen (links) und geladenen (rechts) Zustand	109
Abbildung 87: Abbildung einer Datentabelle in einem VCCL-Knoten.....	109
Abbildung 88: Ausgabe eines Grenzwerts vom Datentabellenknoten für beispielhafte Eingabedaten.....	110
Abbildung 89: Überprüfungsprozess gemäß der DIN 18232-2:2007-11 dargestellt in einem VCCL-Graphen.....	111
Abbildung 90: Interpretation des Ergebnisses des Überprüfungsprozesses im VCCL-Graphen.....	112
Abbildung 91: Visualisierung des Ergebnisses des Überprüfungsprozesses.....	112
Abbildung 92: Visualisierung des Ergebnisses des Überprüfungsprozesses für einen weiteren Raum.....	113

E Codeverzeichnis

Code 1: Abgebildete Norm NFPA 12.3.1 in der FCA-Syntax (Delis und Delis 1995)	14
Code 2: Beispiel für eine Übersetzung einer Regel mit LegalRuleML (Palmirani, et al. 2011)	19
Code 3: Abfrage einer Entität mit der PMQL (Adachi 2002)	20
Code 4: Präzisierung der Abfrage mit SQL-Elementen (Adachi 2002)	20
Code 5: Umfassende PMQL-Abfrage (Adachi 2002)	21
Code 6: Formalisierung der Norwegischen Norm NS 11001-1:2009 gemäß der RASE-Syntax (Hjelseth und Nisbet 2011)	24
Code 7: Vergleich eines IFC- und BOM-orientierten Befehls in BERA (Lee 2010)	27
Code 8: Definition einer Regel in BERA (Lee 2010)	28
Code 9: BERA-Befehl für die Abfrage einer definierten Raumklasse eines Gebäudemodells (Lee 2010)	29
Code 10: BERA-Befehl für Abfrage von Laufwegen innerhalb eines Gebäudemodells (Lee 2010)	29
Code 11: Übersetzung einer Klassenhierarchie in OWL (Kost 2013)	31
Code 12: Beispiel für eine logische Relation im ontologischen Modell und die zugehörige OWL-Übersetzung (Joo-Sung, et al. 2008)	32
Code 13: Beispiel für ein vollständiges ontologisches Modell und die zugehörige OWL-Übersetzung (Kost 2013)	33
Code 14: Vereinfachte Klassendeklaration in EXPRESS (Beetz, van Leeuwen und de Vries 2009)	36
Code 15: Übersetzung der vereinfachte Klassendeklaration in IfcOWL (Beetz, van Leeuwen und de Vries 2009)	37
Code 16: Formulierung einer Vorschrift in SPARQL (Yurchyshyna, et al. 2008)	38
Code 17: Beispiel für eine Formatierung von Programmcode zur Erleichterung der Lesbarkeit	67
Code 18: Beispielhafte Verarbeitungsfunktion eines Operators in Pseudocode	76
Code 19: Funktion eines Lese-Zugriffsoperators in Pseudocode	77
Code 20: Funktion des Multiplikations-Knoten in Pseudocode	78
Code 21: Funktion des Filteroperators in Pseudocode	79
Code 22: Funktion des geometrischen Operators in Pseudocode	81
Code 23: Abschließende Vergleichsoperation einer Überprüfung in Pseudocode	88
Code 24: Darstellung eines VCCL-Graphen in einer Auszeichnungssprache	105

F Syntaxdiagrammverzeichnis

Syntaxdiagramm 1: Schematische Darstellung der Ports für Daten- und Operator-knoten innerhalb der VCCL	71
Syntaxdiagramm 2: Schematische Darstellung eines Objektknoten	72
Syntaxdiagramm 3: Schematische Darstellung der unterschiedlichen Erscheinungsformen eines Objektknoten in Abhängigkeit der Schnittstellen.....	73
Syntaxdiagramm 4: Beispiele für die unterschiedlichen Erscheinungsformen eines Objektknoten in Abhängigkeit der Schnittstellen.....	73
Syntaxdiagramm 5: Schematische Darstellung eines Objektknoten mit den zugehörigen Attributknoten.....	74
Syntaxdiagramm 6: Darstellung des Objektknoten vom Datentyp Wall und ausgewählter Attributknoten.....	74
Syntaxdiagramm 7: Veranschaulichung eines Objektknoten vom Datentyp Liste	75
Syntaxdiagramm 8: Beispiel eines Objektknoten vom Datentyp list<Wall>	75
Syntaxdiagramm 9: Schematische Darstellung eines Operatorknoten.....	76
Syntaxdiagramm 10: Schematische Darstellung einer Zugriffsoperation.....	77
Syntaxdiagramm 11: Schematische Darstellung einer Zugriffsoperation auf das Attribut eines Datenobjektes	78
Syntaxdiagramm 12: Beispiel zu der Berechnung eines fehlenden Attributes eines Objektknoten	78
Syntaxdiagramm 13: Schematische Darstellung eines Operatorobjekts.....	79
Syntaxdiagramm 14: Schematische Darstellung einer Filteroperation nach dem Datentyp Wall.....	80
Syntaxdiagramm 15: Schematische Darstellung eines geometrischen Operators in VCCL.....	81
Syntaxdiagramm 16: Schematische Darstellung eines Datentabellen-Knoten	82
Syntaxdiagramm 17: Darstellung einer Verarbeitungsfunktion nach dem Grundsatz der feinsten Granularität	83
Syntaxdiagramm 18: Veranschaulichung von Zwischenschritten durch Objektknoten	83
Syntaxdiagramm 19: Struktur des Gebäudemodells als Grundlage für die VCCL.....	84
Syntaxdiagramm 20: Zugriff auf das Datenmodell innerhalb mittels eines Zugriff- Operators.....	85
Syntaxdiagramm 21: Darstellung einer Fließtext-Passage einer Vorschrift in VCCL	87
Syntaxdiagramm 22: Schematischer Ablauf des gesamten Überprüfungsprozesses	90