Lehrstuhl für Realzeit-Computersysteme

Automatische Komponentenbestimmung am Beispiel einer aktiven Realzeitdatenbank

Marcel Birkhold

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. techn. Joachim Swoboda

Prüfer der Dissertation:

1. Univ.-Prof. Dr.-Ing. Georg Färber
2. Univ.-Prof. Bernd Brügge, Ph.D.

Die Dissertation wurde am 21.11.2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 16.09.2002 angenommen.

Zusammenfassung

Zunehmend werden industrielle Softwaresysteme aus Komponenten zusammengebaut. Die Entstehung eines Komponentendesigns ist bisher nicht immer durch Objektivität geprägt, sondern sie wird hauptsächlich durch die folgenden Faktoren beeinflußt:

- die fachliche und soziale Kompetenz (die Erfahrung, Qualifikation und Kommunikationsfähigkeit der Mitarbeiter)
- die Organisationsstruktur
- die Wiederverwendung bereits vorhandenen Codes

Designs, die unter diesen Umständen entstehen, sind nicht optimal. Aus derselben Spezifikation entsteht durch unterschiedliche Designer ein breites Spektrum von Komponentendesigns.

In der vorliegenden Arbeit wird ein Verfahren entwickelt, durch das ein Komponentendesign objektiver entsteht. Bisherige Spezifikationssprachen (z. B. die UML) sind nicht ideal zur automatischen Komponentenbestimmung, z. B. sind einige dringend benötigte Informationen nicht verfügbar. Es wird daher zunächst eine neue Spezifikationssprache vorgestellt, das *Graphical Concept Network (GCN)*. Softwaresysteme werden im Graphical Concept Network durch Begriffe und deren Beziehungen untereinander beschrieben. Die Modellierung sollte möglichst intuitiv erfolgen. Eine Beschränkung auf wirklich notwendige Konstrukte soll dies unterstützen, z. B. wird innerhalb des Modells auf Typen (wie class, long, ...) verzichtet. GCN Modelle sind allgemein verwendbar und nicht auf eine spezielle Anwendungsdomäne beschränkt.

Aus dieser Systemspezifikation werden dann – nach einigen Vorverarbeitungsschritten – automatisch die Komponenten des Systems bestimmt. Die Komponentenbestimmung stützt sich auf Kriterien, die dem aktuellen Stand von gutem Softwaredesign entsprechen. Die Kriterien sind sowohl der Fachliteratur entnommen, als auch das Ergebnis langjähriger Erfahrung in der industriellen Softwareentwicklung. Sie werden in Form von Regeln formuliert. Aufbauend auf den Regeln werden Kosten definiert, die dann optimiert werden. Das Ziel ist die Minimierung der Gesamtkosten für das Systems.

Das Verfahren ist nicht ohne Werkzeugunterstützung vorstellbar. Ein Tool zur Eingabe und Auswertung eines GCN Modells wurde prototypisch implementiert. Am Beispiel einer aktiven Realzeitdatenbank wurde das Verfahren untersucht und die Praxistauglichkeit getestet.

Nomenklatur

a	ein Aktor (siehe 4.2)
$BOOL$	Menge der boolschen Werte, $BOOL = \{true, false\}$
c	ein Begriff (concept) (siehe 4.2)
C	Menge von Begriffen (siehe 4.2)
ci	eine Begriffsinstanz (concept instance) (siehe 5.2)
CI	Menge von Begriffsinstanzen (siehe 5.2)
$CONREF$	die Menge der passenden Verfeinerungen; $CONREF = \{fg_i\}$ (siehe 4.2)
$cond$	eine Bedingung (condition) (siehe 4.2)
$COND$	Menge von Bedingungen (siehe 4.2)
$ctype$	Typ eines Begriffs (siehe 4.2)
$dtype$	Datentyp eines Begriffs (siehe 4.2)
$etype$	Endtyp eines Begriffs (siehe 4.2)
$exProp$	Existenzeigenschaften einer Begriffsinstanz (siehe 5.2)
fg	ein funktionaler Teilgraph (siehe 5.1)
fg_c	ein vollständiger funktionaler Teilgraph (siehe 5.1)
FG	eine Menge von funktionalen Teilgraphen (siehe 5.1)
ID	Menge von Identifiern (siehe 4.2)
$impl$	die Implementierung einer Nachricht (siehe 4.2)
l	eine Schleife (loop) (siehe 4.2)
L	Menge von Schleifen (siehe 4.2)
m	eine Nachricht (message) (siehe 4.2)
M	Menge von Nachrichten (siehe 4.2)
mth	eine Methode einer Begriffsinstanz (siehe 5.2)
MTH	Menge von Methoden (siehe 5.2)
N	Menge der natürlichen Zahlen
p	ein Parameter einer Nachricht (siehe 4.2)
P	eine Menge von Parametern (siehe 4.2)
pi	ein Parameter einer Methode (siehe 5.2)
PI	ein Menge von Parametern (siehe 5.2)
$path$	ein Pfad eines Graphen (siehe 5.1)
$POTREF$	die Menge der potentiellen Verfeinerungen; $POTREF = \{fg_i\}$ (siehe 4.2)
r	eine Rolle (siehe 4.2)
s	ein Sheet (siehe 4.2)
TID	Menge von Typ-Identifiern (siehe 4.2)
VAR	Menge von Variablen (siehe 5.2)

Inhaltsverzeichnis

1. Einleitung

Der Traum jedes Softwareentwicklers ist es, Softwaresysteme zu entwickeln, ohne dabei immer die gleichen Probleme neu lösen zu müssen. Trotz intensiver Bemühungen in der Softwaretechnologie ist dieser Traum immer noch nicht erfüllt, auch wenn es deutliche Fortschritte in der Produktivität und der beherrschbaren Komplexität gibt.

In vielen anderen Ingenieursdisziplinen, wie z.B. der Hardwareentwicklung oder dem Maschinenbau, ist es üblich, neue Dinge zu erschaffen, indem bereits vorhandene Teile wiederverwendet werden. Komplexe Schaltungen werden nicht aus einzelnen Transistoren oder Gattern zusammengebaut. Es werden bereits fertige Bauteile verwendet, die selbst schon eine hohe Komplexität besitzen, z.B. ein Microcontroller, ein Busbaustein etc. Der Vorgang ist auch auf einer höheren Ebene denkbar. Ein Rechnersystem (z.B. bestehend aus PC, Monitor, Drucker) ist dafür ein gutes Beispiel. Die Einzelteile können - herstellerunabhängig - ausgetauscht werden. Dasselbe gilt für die Rechnerhardware ebenso: Festplatten, Grafikkarten oder Soundkarten können bei heutigen PCs problemlos kombiniert oder getauscht werden.

Ein Lösungsansatz, die Wiederverwendung in der Softwareentwicklung zu verbessern, ist das objekt-orientierte Paradigma. Daten und Funktionen werden zu Objekten zusammengefaßt und als Einheit begriffen, um sie einfacher wiederverwenden zu können. Die Vorteile, die man sich durch diese Art der Wiederverwendung erhofft hatte, sind hinter den Erwartungen zurückgeblieben. Was innerhalb von Firmen (oder meist nur Abteilungen) noch teilweise funktioniert, war firmenübergreifend zum Scheitern verurteilt. Die Granularität der meisten Objekte ist einfach zu klein, ihre Abhängigkeit von anderen Objekten zu groß.

Komponentenbasierte Softwareentwicklung setzt nun hier an. Die Grundidee dabei ist, Softwaresysteme aus fertigen Komponenten zusammenzusetzen. Diese Komponenten können zusätzlich für den speziellen Einsatz über Properties konfiguriert werden. Sie erlauben es, Komponenten so flexibel zu gestalten, daß sie in unterschiedlichen Umgebungen eingesetzt werden können. Über Properties kann z.B. die Farbe eines Buttons, die Größe eines Fensters oder ein spezieller Algorithmus zur Sortierung festgelegt werden.

Zunächst waren bei der komponentenbasierten Softwareentwicklung prinzipielle technische Probleme zu lösen. Wie arbeiten Komponenten, die sich zur Entwicklungszeit nicht kennen, überhaupt zusammen? Wie finden sie sich? Wie kommunizieren sie miteinander? Diese Fragen werden durch Komponentensysteme adressiert, wie sie heute schon häufig im Einsatz sind. Beispiele sind hier COM oder JavaBeans. Sie legen den technischen Rahmen für den Entwurf und die Entwicklung von Komponenten fest.

Neben diesen technischen Problemen stellt sich auch die Frage, was ein optimales Komponentendesign ausmacht. Welche Größe sollen die Komponenten beispielsweise haben? Diese Fragen zu beantworten ist nicht trivial. Im Prinzip kann ein einzelnes Objekt eine Komponente sein, aber - als anderes Extrem - auch eine ganze Anwendung. Meist werden diese Fragen während des Designs eines Systems beantwortet, allerdings ohne Anwendung objektiver Kriterien. Häufig kommt es zwischen den Entwicklern dadurch zu heftigen Diskussionen oder sogar Streit. Welche

Lösung dann wirklich realisiert wird, wird nicht selten durch ein Machtwort ('Management-Entscheidung') bestimmt.

Wenn man solche Designfragen objektiver beantworten will, muß die Spezifikation eines Softwaresystems auf andere Weise erfolgen als heute üblich. Es wird eine höhere Abstraktionsebene benötigt. Heute gängige objekt-orientierte Spezifikationssprachen, z.B. die Unified Modeling Language (UML), erzwingen frühzeitig Aussagen, die besser nicht erfolgen sollten. So werden Eigenschaften des Systems festgelegt, die sich bei Anwendung von objektiven Kriterien nicht immer als vorteilhaft erweisen. Nach Abschluß der Spezifikation könnten diese Eigenschaften mit besserem Überblick festgelegt werden.

Die Entscheidungen werden objektiver, wenn die Entscheidungsfindung nicht dem Menschen alleine überlassen wird, sondern automatisiert durch die Anwendung von Regeln erfolgt. Der Entscheidungsprozeß wird dadurch anhand der Regeln nachvollziehbar und wiederholbar. Die Auswirkungen, die sich durch Änderungen in der Spezifikation ergeben, können sofort im Design überprüft werden. Man gewinnt dadurch eine sehr effiziente Möglichkeit, Varianten der Spezifikation durchzuspielen, um sie iterativ zu verbessern.

Um objektive Aussagen fundiert treffen zu können, werden allerdings auch zusätzliche Informationen über das System benötigt, die normalerweise nicht durch die meist statische Modellierung (z.B. mit Klassendiagrammen) verfügbar sind. Dies betrifft u.a. die tatsächlich verwendete Zahl von Objekten (im Sinne von Instanzen von Klassen) oder die Anzahl der wirklich stattfindenden Aufrufe zwischen Objekten.

In dieser Arbeit wird untersucht, ob und wie objektive Kriterien angewendet werden können, um Komponentendesign besser zu machen. Ziel ist die automatische Bestimmung von Komponenten. Um das Ziel zu erreichen, ist eine neue Spezifikationssprache notwendig. Sie basiert auf Begriffen und den Beziehungen zwischen den Begriffen und heißt 'Graphical Concept Network'.

Durch die ständig steigende Leistungsfähigkeit von Prozessoren und den Preisverfall bei Speicherbausteinen sind objekt-orientierte Ansätze und Komponententechnologien auch für einige eingebettete Systeme verwendbar. Besonders geeignet sind dazu Systeme, die viele Daten verwalten müssen, sogenannte *daten-dominierte Systeme*. Sie sind meist so komplex, daß ohnehin leistungsfähige Hardware benötigt wird. Hierzu zählen z.B. Meßgeräte oder Flugüberwachungssysteme.

Die vorliegende Arbeit ist wie folgt gegliedert: Kapitel 2 beschäftigt sich mit moderner Softwareentwicklung, im speziellen mit Komponenten, Komponentenmodellen und Datenbanken. Kapitel 3 gibt einen Überblick über das *Graphical Concept Network* (GCN). Es erfolgt eine Einordnung in das Umfeld und ein einleitendes Beispiel zeigt die Erstellung eines GCN Modells. Anschließend werden spezielle Syntaxelemente genauer vorgestellt. In Kapitel 4 wird das Graphical Concept Network formal definiert. Kapitel 5 beschäftigt sich mit der Abbildung eines abstrakten GCN Modells auf eine konkrete Realisierung: Die Umformung in einen funktionalen Graphen, die Bildung der Begriffsliste, die Klassifikation und die Komponentenbestimmung. Kapitel 6 beschreibt als reales Anwendungsbeispiel die Modellierung einer Datenbank mit einem Graphical Concept Network und die Ergebnisse des Compilevorgangs. Das letzte Kapitel faßt die wesentlichen Erkenntnisse zusammen und gibt einen Ausblick auf mögliche Erweiterungen des Systems.

2. Moderne Softwareentwickung

2.1. Komponentenbasierte Softwareentwicklung

Die moderne Softwareentwicklung ist gekennzeichnet durch die Forderung nach immer komplexeren Softwaresystemen bei gleichzeitig sinkenden Entwicklungszeiten (Kostendruck). Die komponentenbasierte Softwareentwicklung kann einen möglichen Ausweg darstellen. Sie gewinnt immer mehr an Bedeutung und kann als logische Fortsetzung bzw. Erweiterung des objektorientierten Programmierens verstanden werden [Szy97].

Die Vorteile des komponentenorientierten Softwareentwurfs sind u.a.:

- Fehlervermeidung durch weniger Neucodierung; d. h. höhere Softwarequalität, Zuverlässigkeit, Sicherheit, Performance durch ausgereifte Software.
- Wiederverwendbarkeit: kürzere Entwicklungszeiten, geringere Kosten, geringerer Testaufwand; d. h. Verbesserung der 'Time-To-Market'.
- Vereinfachung der 'Make-Or-Buy'-Entscheidung.

Ziel ist es, Anwendungsprogramme aus Komponenten, d. h. vorgefertigten, in sich abgeschlossenen binären Software-Bausteinen[1] mit definierter Aufgabe zusammenzusetzen („LEGO[2]-Prinzip"). Diese werden für die jeweilige Anwendung nur noch entsprechend konfiguriert. Das Zusammenspiel der Komponenten stellt dann die gewünschte Funktionalität der Applikation zur Verfügung. Ein konsequent komponentenbasierter Entwicklungsprozeß trägt wesentlich dazu bei, Entwicklungszeiten zu verkürzen und qualitativ bessere Software zu geringeren Kosten bereitzustellen.

Damit Komponenten ohne zusätzliche Codeentwicklung ('Gluecode') kombiniert werden können, wurde im Rahmen des Forschungsverbundes FORSOFT der Bayrischen Forschungsstiftung eine datenbankbasierte Architektur [MBFW99] [BMW00a] entwickelt (siehe Abbildung 2.1). Zu sehen ist die Infrastruktur der Komponententechnologie, der direkte Austausch von Kontrollinformationen zwischen den Komponenten und die davon getrennte Übertragung von Daten über die Zustandsdatenbasis. Das Kernstück dieser Architektur, die Zustandsdatenbasis, wird realisiert durch eine aktive Realzeit-Datenbank, die die Kommunikation der Komponenten untereinander ermöglicht.

Zum Aufbau dieses Kapitels: Zunächst wird der Begriff der Komponente genauer betrachtet und den Rahmen dieser Arbeit definiert. Anschließend erfolgt eine kurze Darstellung praxisrelevanter Komponentenmodelle. Eine Vorstellung von aktiven Realzeit-Datenbanken soll einen Eindruck über den Stand der Technik in diesem Bereich vermitteln. Eine Problembeschreibung rundet das Kapitel ab.

[1] d. h. die Bausteine liegen vollständig fertig kompiliert vor
[2] LEGO ist ein eingetragenes Warenzeichen

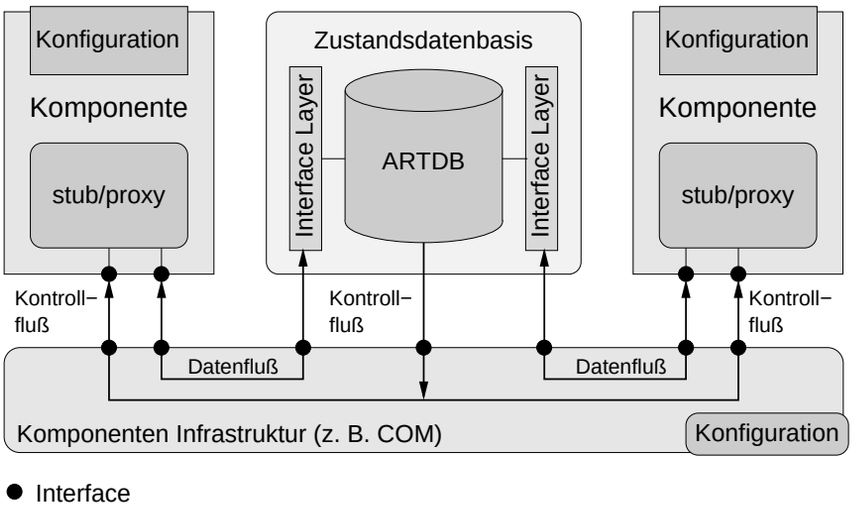


Abbildung 2.1.: Architekturmodell: Abgebildet ist die getrennte Übertragung von Kontrollfluß (z. B. über COM) und Datenfluß (über die sog. Zustandsdatenbasis mit konfigurierbarem Interfacelayer).

2.1.1. Definitionen

Wie durch die Einleitung bereits deutlich geworden ist, wird der Begriff der Softwarekomponente vielseitig verwendet. Es gibt eine Vielzahl von Definitionen, von denen nun einige exemplarisch genannt werden:

- 'Components are software units that are context independent both in the conceptual and the technical domain. [CS96]'
- 'Eine Softwarekomponente ist ein Baustein mit vertraglich spezifizierten Schnittstellen und nur ausdrücklichen Kontextabhängigkeiten. Eine Softwarekomponente kann unabhängig verwendet werden und leicht mit anderen Komponenten integriert werden [Szy97].'
- 'Eine Komponente ist ein Stück Software, das klein genug ist, um es in einem Stück zu erzeugen und pflegen zu können, das groß genug ist, um eine sinnvolle Funktionalität zu bieten und eine individuelle Unterstützung zu rechtfertigen sowie mit standardisierten Schnittstellen ausgestattet ist, um mit anderen Komponenten zusammenzuarbeiten [Gri98]'

Wie man sieht, legt jede Definition Wert auf unterschiedliche Aspekte. Eine einheitliche Definition des Begriffs *Komponente* ist deswegen schwierig. Der Verlauf einer Diskussion von Wissenschaftlern aus Forschung und Industrie zu diesem Thema ist zusammengefaßt in [B+98]. Einigkeit besteht darin, daß der Umfang einer Komponente sehr unterschiedlich sein kann. Der eine Extremfall besteht darin, daß ein einzelnes Objekt als Komponente aufgefaßt wird, der andere, daß auch eine komplette Anwendung, die durch ein Interface angesprochen werden kann, eine Komponente darstellt. Im Verlauf der Diskussion wurde ein formales Modell vorgestellt [Bro98], das auf alle bisher gängigen Komponentensysteme paßt. Eine Komponente besteht in diesem Modell aus Input- und Outputkanälen (Channels). Die Komponente beschreibt ihr Verhalten durch eine Funktion F, die die Inputkanäle auf die Outputkanäle abbildet. Kritisiert wurde an diesem Modell allerdings die sehr große Allgemeinheit. Ein formales Modell für Componentware im allgemeinen, das auch die Interaktionen zwischen Komponenten berücksichtigt, findet man in [BRS+99].

Inzwischen konvergiert das Verständnis einer Softwarekomponente. Dies zeigt sich auch an den gängigen Komponentenmodellen (siehe Abschnitt 2.1.3), auch wenn ihre technischen Implementierungen unterschiedlich sind. Eine Komponente realisiert bestimmte Funktionalitäten, die sie nach außen als Dienste bekannt macht. Diese Dienste können von anderen Komponenten benutzt werden. Logisch zusammenhängende Dienste werden in Schnittstellen (Interfaces) zusammengefaßt. Eine Komponente kann mehrere Schnittstellen haben, wobei ein Dienst auch von mehreren Komponenten angeboten werden kann.

In dieser Arbeit wird davon ausgegangen, daß eine Komponente mindestens die im folgenden beschriebenen Eigenschaften erfüllt.

Eine *Komponente* besteht aus:

- einem eindeutigen Namen,
- einer Menge von Interfaces (im allgemeinen > 1),
- einer Menge von internen Klassen, die die Komponente implementieren und
- einer Menge von Properties (Konfigurationsdaten), um die Komponente für das aktuelle Einsatzgebiet konfigurieren zu können.

Schnittstellen sind der zentrale Punkt bei Komponenten. Sie erlauben den Zugriff auf die Funktionalität einer Komponenten nur an genau definierten Stellen. Eine *Schnittstelle* weist deshalb folgende Eigenschaften auf:

- Sie besitzt einen eindeutigen Namen.
- Sie besteht aus einer Menge von Methoden mit definierter Syntax und Semantik.
- Sie bildet einen Vertrag, der für immer unveränderlich festgelegt ist.

Da in dieser Arbeit nur das objekt-orientierte Paradigma betrachtet wird, werden Interfaces und Komponenten auf Implementierungsebene durch Klassen realisiert[3].

Eine *Klasse* besteht aus:

- einem eindeutigen Namen.
- einer Menge von Methoden.
- einer Menge von Attributen.

Zusätzlich wird erwartet, daß eine Komponente ein in sich fertiges, abgeschlossenes, binäres[4] Stück Software [Szy97] ist. Nur so kann eine Komponente einfach wiederverwendet werden; auch bei einem Wechsel der Entwicklungsumgebung oder Programmiersprache.

Eine Komponente ist aber immer für die Verwendung innerhalb eines *Komponentenmodells* ausgelegt. Leider können Komponenten, die für ein Komponentenmodell entwickelt wurden, in der Regel nicht im Rahmen eines anderen Komponentensystems eingesetzt werden.

Analog zum objekt-orientierten Paradigma, bei dem ein Objekt eine Instanz einer Klasse ist, ist eine Komponenteninstanz eine Instanz einer Komponente. In der Literatur wird der Begriff 'Komponente' häufig sowohl für Komponente als auch Komponenteninstanz verwendet.

[3] COM erlaubt z.B. auch prozedurale Sprachen zur Implementierung von Komponenten
[4] kann auch ein just-in-time compilierbares Format sein, wie es z.B. Java verwendet

Komponenten können neben dem Binärcode weitere zugehörige Informationen enthalten, die während der Entwicklung oder zur Laufzeit hilfreich oder notwendig sind. Dazu zählen z.B. Dokumentation, die Spezifikation des Input-/Output-Verhaltens [BRS+99], Factories zur Erzeugung von Instanzen [Mic97], Pre- und Postconditions [Mey94] etc. Im Rahmen dieser Arbeit werden diese Zusätze nicht näher betrachtet.

Neben den bisher beschriebenen *Black-Box-Komponenten* (die internen Details einer Komponente bleiben verborgen), werden in der Literatur [BW97] auch sogenannte *Gray-Box Komponenten* favorisiert. Gray-Box Komponenten veröffentlichen Teile ihrer Interna, um z.B. anzugeben, unter welchen Umständen sie andere Komponenten benutzen oder in welchem Zustand sie sich befinden.

Das Resultat der hier beschriebenen automatischen Komponentenbestimmung sind aber ausschließlich Black-Box-Komponenten – auch wenn das System selbst die Innereien der Komponenten kennt. Nach der automatischen Codegenerierung liegen binäre Komponenten vor, die nur über ihre Schnittstellen angesprochen werden können.

2.1.2. Willkürlichkeit eines Komponentendesigns

Der Zugriff auf die Funktion der Komponente erfolgt ausschließlich über Interfaces. Daher müssen auch alle Verwaltungsfunktionen über Interfaces ausgelöst werden (z.B. in COM: IUnknown, IClassFactory).

Durch die Implementierung der veröffentlichten Methoden an Interfaces durch Klassen werden Methoden an Interfaces zu Methoden einer Klasse.

Daraus folgt: Wenn ein Interface aus seinen Methoden besteht und jede Methode ihrerseits aus ihrer Syntax und Semantik, dann bildet ein Interface die eigentliche Funktion einer Komponente nach außen ab. Da eine Komponente nur über ihre Interfaces angesprochen werden kann, kann die Komponente auch nicht mehr Funktionalität haben, als über die Methoden der Interfaces angesprochen werden kann.

Dies führt zu folgender Aussage:

Die Funktion einer Komponente besteht aus der Summe ihrer Interfaces.

Da ein Interface nicht an eine Komponente gebunden ist, ist es die Funktion genauso wenig. Eine Komponente ist daher nichts weiter als eine Zusammenfassung verschiedener Funktionen an einem (speziellen) Ort. Die Zusammenstellung der Funktion in einer Komponente ist somit willkürlich, da es keine einschränkenden Regeln gibt.

Als Beispiel dienen drei Klassen (in Abbildung 2.2 dargestellt), $class1$, $class2$ und $class3$. Sie haben jeweils eine Methode, bezeichnet mit $m1$, $m2$ bzw. $m3$. Alle drei Methoden sollen von außen angesprochen werden können. Eine Aufteilung der einzelnen Methoden auf *mehrere* Interfaces an einer Komponente wird in diesem Beispiel nicht betrachtet.

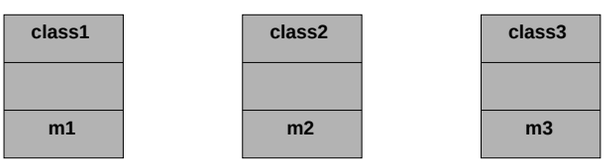


Abbildung 2.2.: Diese drei Klassen mit jeweils einer Methode sollen auf Komponenten verteilt werden.

Die Verteilung der drei Klassen auf Komponenten ist in fünf Varianten möglich. Die beiden extremen Möglichkeiten sind eine Komponente mit 3 Klassen bzw. 3 Komponenten mit jeweils einer Klasse. Die restlichen drei Varianten ergeben jeweils zwei Komponenten. Abbildung 2.3 zeigt drei der möglichen Varianten. Dieses Beispiel zeigt schon für wenige Klassen die Vielzahl

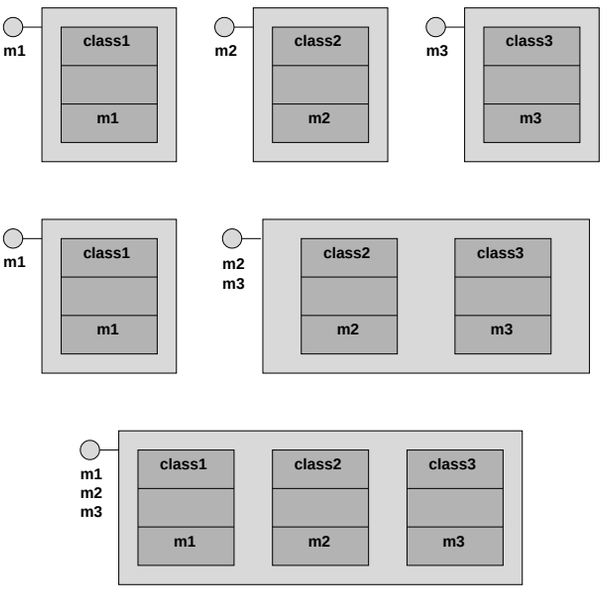


Abbildung 2.3.: Drei Varianten, drei Klassen auf Komponenten zu verteilen. Die Methoden sind unterhalb der Interfaces (kleine Kreise, auch 'Lollipops' genannt) dargestellt.

der denkbaren Designs. Welche Möglichkeit wirklich die beste Variante ist, ist schwierig zu entscheiden. Die Entscheidung wird durch viele Faktoren beeinflußt, z.B.:

- welche Beziehungen existieren zwischen den Klassen ?
- welche Daten und welche Menge an Daten werden zwischen den Klassen ausgetauscht ?
- wie groß werden die entstehenden Komponenten ?

Die getroffenen Entscheidungen hängen bisher u.a. sehr stark von der Qualifikation, der Erfahrung und den persönlichen Ansichten des Designers bzw. des Teams ab. Ob das so entstehende Ergebnis dann aber wirklich gut bzw. optimal ist, ist ungewiß.

2.1.3. Aktuelle Komponententechnologien

Dieses Kapitel stellt einige aktuelle Komponententechnologien vor. Ziel dieser Darstellung ist eine Beschreibung der unterschiedlichen technischen Realisierungen von Komponentenmodellen. COM definiert einfach ein binäres Format zum Aufruf von Methoden an Interfaces, bei JavaBeans werden die Eigenschaften der Sprache Java benutzt und CORBA verwendet eine zentralistische Struktur (ORB).

Eine vergleichende Bewertung der Modelle findet hier absichtlich nicht statt. Mehr technisch orientierte Vergleiche von Komponentenmodellen sich in der Literatur zu finden (z.B. [HJS01] [Raj98]). Der praktische Einsatz und die dabei auftretenden Probleme werden z.B. in [GT00] ausführlich dargestellt.

COM, DCOM

Das Component Object Model (COM) von Microsoft bildet die Grundlage für die Interaktion zwischen allen Arten von Software. Die meisten COM-gestützten Technologien werden heute zwar mit *ActiveX* bezeichnet, die Grundlage ist aber immer COM [Mic97]. COM ist inzwischen nicht mehr auf Windows-Betriebssysteme beschränkt. Es gibt z.B. für den Apple Mac eine Implementierung von Microsoft selbst und von der Software AG [Sof01] u.a. für das Betriebssystem Linux.

Jedes COM-Objekt unterstützt beliebig viele Schnittstellen, die jeweils eine Reihe von Methoden umfassen. Clients von COM-Objekten können nur über die Methoden der Schnittstellen auf die Dienste und Daten zugreifen. Abbildung 2.4 stellt ein COM-Objekt mit seinen Schnittstellen dar.

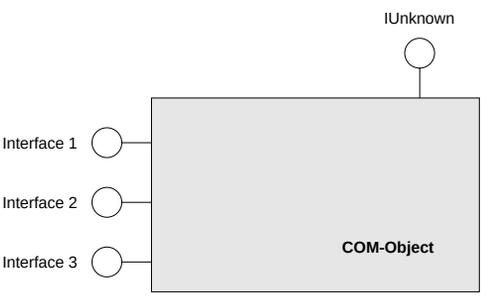


Abbildung 2.4.: Darstellung eines COM-Objekts mit seinen Schnittstellen.

Eine Schnittstelle ist ein Vertrag zwischen dem Objekt und seinen Clients. COM fordert, daß eine einmal definierte Schnittstelle sich nie wieder ändert. Falls eine neue Version eines COM-Objektes erweiterte Dienste anbieten will, definiert es einfach eine neue Schnittstelle. Alte Clients sehen keine Änderungen an den von ihnen verwendeten Schnittstellen. Neue Clients benutzen die alten und neuen Schnittstellen.

Um Objekte und Schnittstellen referenzieren zu können, müssen sie eindeutige Namen besitzen. COM verwendet dazu in Raum und Zeit eindeutige 128 Bit GUIDs (Global Unique IDentifier).

COM definiert einen *binären* Standard. Es spielt daher keine Rolle, in welcher Programmiersprache ein COM-Objekt implementiert wird. Es muß nicht einmal eine objekt-orientierte Sprache sein. Wichtig ist nur, daß die Schnittstellen des COM Objekts die richtige binäre Struktur hat.

Die Implementierung eines COM-Objekts kann durch eine einzige Klasse erfolgen, es können aber auch viele, verschiendene Klassen sein. Abbildung 2.5 zeigt ein COM-Objekt mit einer Schnittstelle. Der Schnittstellenzeiger, den ein Client auf eine Schnittstelle besitzt, hat er sich zunächst vom Komponentensystem besorgt. Der Zeiger zeigt auf eine Struktur, die sogenannte VTable. Für jede Methode der Schnittstelle gibt es einen Eintrag in der VTable, der auf die Methode verweist. Diese Struktur entspricht dem Vorgehen eines C++ Compiler für eine Klasse. Somit ist die Implementierung eines COM-Objektes in C++ etwas direkter als in anderen Sprachen.

Jedes Interface in COM ist vom *IUnknown* Interface abgeleitet. *IUnknown* umfaßt die drei Methoden $QueryInterface()$, $AddRef()$ und $Release()$. Damit hat jedes andere Interface auch diese drei Methoden. Durch das Erzeugen eines COM-Objektes bekommt ein Client einen Zeiger auf ein beliebiges Interface. $QueryInterface()$ dient dann dazu, weitere Schnittstellen des Objektes zu erfahren. Da jedes Interface von *IUnknown* erbt, reicht ein Zeiger auf eine beliebige Schnittstelle, um die anderen Schnittstellen zu bekommen.

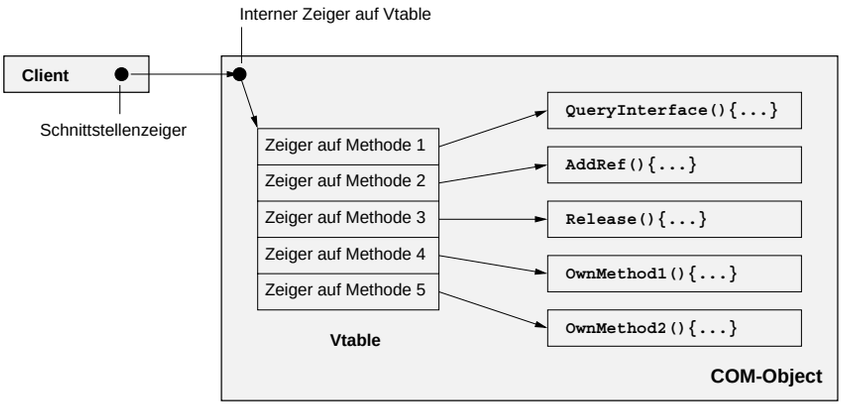


Abbildung 2.5.: Darstellung eines COM-Objekts mit seinen Schnittstellen (aus [Cha96]).

AddRef() und *Release*() realisieren eine Referenzzählung, um zu bestimmen, wann ein Objekt nicht mehr benötigt wird und freigegeben werden kann. Probleme kann es hier allerdings bei zyklischen Abhängigkeiten geben.

Die Schnittstellen in COM werden mit einer IDL (Interface Definition Language) beschrieben [Mic98][5]. An Schnittstellen ist nur einfache Vererbung erlaubt, für die Implementierung des Objektes selbst hängt es nur von der Programmiersprache ab.

Bei COM sind die Interfaces der zentrale Punkt. Da beliebig viele Interfaces unterstützt werden, ist Mehrfachvererbung an Schnittstellen nicht notwendig.

DCOM (Distributed COM) ist die verteilte Variante von COM. Aus Sicht des Client ist das Vorgehen völlig transparent. Die Kommunikation mit entfernten Objekten wird mit RPC (Remote Procedure Calls) über Proxy/Stub Verbindungen durchgeführt. Marshalling bzw. Unmarshalling dient der Verpackung von Daten in ein Standardformat beim Transport über Prozeß- oder Rechnergrenzen hinweg.

Ein Wort zur Performance vom COM: Aufrufe innerhalb des gleichen Prozeßes sind (nach der Herstellung aller Zeigerverbindungen) nichts weiter als direkte Methodenaufrufe. Die Verwendung von COM hat also keinen Einfluß auf die Geschwindigkeit. Anders sieht es natürlich bei der Prozeß- bzw. Rechnerübergreifenden Kommunikation aus. Hier fordert das Marshalling und der Transport seinen Tribut. Im Vergleich zu anderen Komponentenmodellen ist die Performanz allerdings sehr hoch [GT00].

Diese Darstellung von COM ist bei weitem nicht vollständig. Für mehr Informationen sei hier z.B. auf [Cha96] verwiesen.

JavaBeans

JavaBeans bauen auf der Programmiersprache Java auf, und ihre Eigenschaften sind ohne Kenntnisse von Java nicht zu verstehen. Aus diesem Grund wird zunächst ein Überblick über Java gegeben. Die folgenden Ausführungen sind so oder in ähnlicher Form in vielen Publikationen, z.B. [Szy97] [Mor97], zu finden oder in den Spezifikationen von Java [Sun99] und JavaBeans [Sun97].

Java ist eine objekt-orientierte Programmiersprache, die sich an C++ anlehnt, aber gestrafft

[5] um sie von der CORBA IDL zu unterscheiden, spricht mal auch von MIDL (Microsoft Interface Definition Language)

wurde. Ein wichtiger Unterschied ist das Fehlen von Zeigern (Pointern). Die Mehrfachvererbung vom Klassen wurde ebenfalls gestrichen. Dafür kam eine automatische Garbage-Collection hinzu.

Der große Vorteil von Java ist die Plattformunabhängigkeit. Sie wird durch eine Übersetzung des Javaprogramms in Bytecode erreicht. Der Bytecode wird dann von einer *virtuellen Maschine* abgearbeitet. Compilierte Java Klassen werden zu *Packages* zusammengefaßt.

JavaBeans ist das Komponentensystem zu Java. Die JavaBeans-Spezifikation fordert als Hauptmerkmal einer JavaBeans-Komponente, daß sie in einer visuellen Programmierumgebung manipuliert werden kann. Dies bedeutet nicht, daß JavaBeans nur sichtbare Komponenten für graphische Benutzeroberflächen sein können. Die Forderung bezieht sich nur auf die 'Design Time', zur Laufzeit können die JavaBeans durchaus unsichtbar sein.

Realisiert werden JavaBeans im Grunde durch ein zusätzliches Package für Java. Die JavaBeans API stellt einige Funktionen zur Verfügung, die in einem Komponentensystem benötigt werden, z.B. für das visuelle Toolhandling (*introspection*) oder die Persistenz für JavaBeans. Beans können *Properties* besitzen und dadurch konfiguriert werden.

Da JavaBeans direkt auf Java aufsetzen, stehen alle Möglichkeiten von Java zur Verfügung. Mehrfachvererbung von Klassen (Implentierungsvererbung) ist in Java nicht möglich, mehrfache Interfacevererbung dagegen schon. Interfaces sind in Java eigentlich pure virtuelle Klassen; so werden die typischen Probleme bei Mehrfachvererbung vermieden (Diamond-Problem [Szy97]).

Durch die Mehrfachvererbung von Interfaces können Javaklassen beliebige Interfaces implementieren. Ob eine Klasse ein bestimmtes Interface implementiert (also unterstützt) kann zur Laufzeit durch *instanceof* geprüft werden:

```
Bean1 myBean;

if (myBean instanceof Interface1)
{
 Interface1 myIf=(Interface1) myBean;
 ...
}
```

Die Speicherverwaltung von JavaBeans nutzt die Mechanismen von Java. Objektreferenzen werden transparent gezählt und nicht mehr referenzierte Objekte werden automatisch freigegeben. Selbst Referenzzyklen werden erkannt und korrekt behandelt. Als Programmierer braucht man sich um die Referenzzählung nicht zu kümmern.

JavaBeans selbst bieten für eine verteilte Implementierung keine eigene Infrastruktur. Entweder man benutzt die Java eigene *Remote Method Invocation* (RMI), oder man verwendet DCOM oder CORBA.

CORBA

CORBA steht für *Common Object Request Broker Architecture*. Die Wurzeln von CORBA liegen im Jahr 1989, damals wurde die OMG (Object Management Group) gegründet.

Das urspüngliche Ziel der OMG war es, ein generelles Problem zu lösen: wie können verteilte objekt-orientierte Systeme interagieren, wenn sie in unterschiedlichen Sprachen auf unterschiedlichen Rechnern laufen. Das Ergebnis war CORBA 1.1 im Jahr 1991. Es folgte CORBA 2.0,

veröffentlicht im Jahr 1995 und verfeinert 1996 [OMG96]. Die Version 2.4 der CORBA Spezifikation aus dem Jahr 2000 wurde im wesentlichen um die Anforderungen für Realzeit-Systeme erweitert.

Das Ziel hinter CORBA war die Integration einer breiten Masse von Sprachen, Implementierungen und Plattformen. Daher konnte sich die OMG nie auf binäre Standards einlassen: Alles wird sorgfältig standardisiert, um viele verschiedene Implementierungen zu ermöglichen. Der Nachteil dieser offenen Methode ist, daß individuelle CORBA Implementierungen nicht (effizient) binär miteinander arbeiten können. Sie müssen immer auf (teure) high-level Protokolle ausweichen (z.B. das Internet Inter-ORB Protokoll (IIOP)).

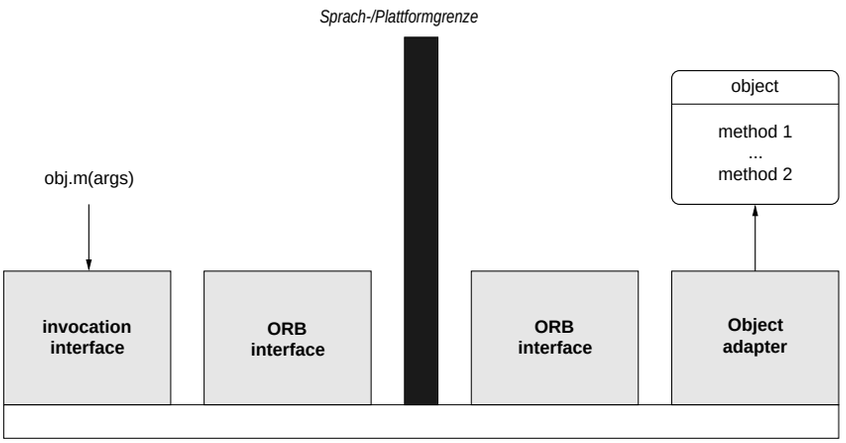


Abbildung 2.6.: schematische Darstellung CORBA Architektur (aus [Szy97]).

Der zentrale Baustein der CORBA-Architektur (Abbildung 2.6) ist der Object Request Broker (ORB). Der ORB dient als zentraler Objekt Bus, über den alle Anforderungen von Clients transparent für diese abgewickelt werden. Jedes CORBA Server Objekt hat eine *einzige* Schnittstelle und stellt damit einen Satz von Methoden zur Verfügung.

Um einen Service anzufordern, besorgt sich ein Client vom ORB eine Objektreferenz auf das Serverobjekt. Damit kann der Client scheinbar Methodenaufrufe ausführen, als ob der CORBA Server im Adressraum des Clients wäre. Der ORB hat die Aufgabe die Implementierung des Serverobjektes zu finden und die Kommunikation zwischen dem Client und dem Server zu unterhalten. Ein CORBA Objekt interagiert mit dem ORB entweder über das ORB Interface oder durch einen *Objekt Adapter* - es gibt *Basic Objekt Adapter (BOA)* (eigentlich veraltet, da abhängig vom verwendeten ORB) und *Portable Objekt Adapter (POA)*.

Die Schnittstellen werden auch in CORBA über eine Interface Definition Language (IDL) beschrieben, die einen vollständig anderen Sprachumfang als die COM IDL besitzt. Aus der IDL Beschreibung werden dann für die gewünschte Zielsprache (u.a. C++ oder Java) die stub (bzw. proxy)/skeleton[6] Dateien erzeugt.

Da CORBA nur eine Spezifikation ist, gibt es für (fast) alle Plattformen eine ORB Implementierung. Bekannt sind z.B. VisiBroker von Borland [Cor01] oder Orbix von Iona [Ion01].

CORBA verwendet wie COM eine Referenzzählung, die eine Mithilfe des Programmierers erfordert. Nicht mehr benötigte Referenzen müssen explizit freigegeben werden.

Die Einschränkung auf eine Schnittstelle wird durch Mehrfachvererbung an Interfaces entschärft.

[6] Die Clientseite heißt in CORBA *stub oder proxy*, die Serverseite *skeleton*. In DCOM heißt die Clientseite *proxy* und die Serverseite *stub*.

Allerdings erreicht CORBA damit nicht die Flexibilität von COM oder JavaBeans, die Interfaces als eigenständige Einheiten betrachten. Dies ist nicht erstaunlich, denn im Grunde ist CORBA eher als *Middleware* zu klassifizieren, als ein vollständiges Komponentensystem [OH98]. Abhilfe soll hier *CorbaComponents* schaffen, das u.a. mehrere Schnittstellen pro Serverobjekt erlaubt. Allerdings ist CorbaComponents bis heute (10/2001), trotz mehrjähriger Diskussion, nicht verabschiedet.

2.2. Aktive Realzeitdatenbanken

Aktive Datenbanken [WC96] bieten die Möglichkeit, mit bestimmten Situationen, die in einer Datenbank auftreten können (z. B. ein bestimmter Zustand oder Zustandswechsel in der Datenbank; eine bestimmte Operation auf der DB), Aktionen zu verbinden, die nur dann ausgeführt werden, falls eine vorher festgesetzte Bedingung erfüllt ist (Event-Condition-Action-Prinzip, ECA-Prinzip). Bedingungen betreffen in der Regel Aussagen über den augenblicklichen Zustand der Datenbank. Aktionen können z.B. das Auslösen von Vorgängen in der Datenbank oder der Aufruf von Funktionen und Programmen sein. Typische Vertreter von aktiven Datenbanken sind z. B. POSTGRES [SRH90], SQL-3 (relational), SAMOS [GD92] oder Ode [GJ91] (objekt-orientiert).

Realzeitdatenbanken [Ram96] bieten die Möglichkeit, neben der logischen Konsistenz auch die rechtzeitige Bearbeitung von Transaktionen sicherzustellen. Dies ist insbesondere dann notwendig, wenn beispielsweise auf der Grundlage des Datenbankinhaltes Steuer- und Regelentscheidungen getroffen werden müssen.

Aktive Realzeitdatenbanken [HB99] verbinden die Eigenschaften beider Datenbanktypen und stellen aktive Datenbankfunktionalität unter Berücksichtigung von Zeitbedingungen zur Verfügung.

Eines der ersten Projekte dieser Art war HiPAC [CBB+89], allerdings war dort das Thema Realzeit nur für zukünftige Arbeiten vorgesehen. Neuere Forschungsprojekte sind REACH [BBKZ93], STRIP [AKGM96] und DeeDs [AHE+96]. REACH ist ein Prototyp einer diskbasierten aktiven Realzeitdatenbank. STRIP ist dagegen hauptspeicherbasiert und unterstützt weiche Zeitanforderungen. DeeDs basiert auf dem objekt-orientierten Ansatz und wurde für die Realisierung der ZDB des Architekturmodells näher untersucht. Leider zeigte sich, daß sie nicht alle Anforderungen erfüllen konnte. Zum einen ist sie sehr groß (ohne Daten benötigt sie ca. 2 MB Speicher), zum anderen unterstützt sie aktuell keine dynamischen Regeln, so daß Clients sich nicht dynamisch an- und abmelden können. Dies war aber eine wesentliche Voraussetzung für die ZDB.

Es gibt inzwischen auch kommerzielle Implementierungen von aktiven Realzeitdatenbanken. Es zeigte sich allerdings, daß ein Einsatz aufgrund der vorhandenen Unzulänglichkeiten z.Zt. nicht zu empfehlen ist [Blö00].

2.3. Probleme/Motivation/Ziele der Arbeit

Es wurde bereits dargelegt (siehe Abschnitt 2.1.2), daß viele Entscheidungen, die ein Komponentendesign bestimmen, willkürlich getroffen werden müssen. Dies gilt sogar, wenn bekannte Designkriterien beachtet werden. Heutige Designs sind einfach zu komplex, um jederzeit alle

Kriterien beachten zu können. Ein weiteres Problem ist, daß bereits Entscheidungen getroffen werden, bevor alle für die Entscheidungen relevanten Informationen vorhanden sind.

Man befindet sich so in einen Zustand, der nicht zufriedenstellend ist. Er führt dazu, daß Komponentendesigns häufig keinen Bestand haben (Verlust der Wiederverwendbarkeit), weil sie einfach durch bessere Designs ersetzt werden können.

Das Ziel kann deshalb nur sein, die Entscheidungen nicht alleine dem Menschen zu überlassen, sondern den Entscheidungsprozeß zu automatisieren. Das optimale Ergebnis muß also automatisch mit Hilfe von vorgegeben Kriterien berechnet werden. Durch den Ausschluß des Menschen an diesen Entscheidungen wird das Ergebnis reproduzierbar.

Die Kriterien werden in Form von Regeln definiert. Die Regeln repräsentieren aktuell bekannte und bewährte Design- und Implementierungskriterien (z.B. [VB00]). Sie umfassen u.a. auch bekannte objekt-orientierte Designheuristiken (z.B. [Rie96]). Die hier aufgestellten Regeln liefern für jedes korrekte Design ein optimales Ergebnis.

Ziel der Arbeit ist es nicht, einen optimalen Satz von Regeln aufzustellen. Es werden zwar Regeln aufgestellt, aber keine Aussagen über die Qualität der Regeln gemacht. Die angegebenen Regeln erfüllen aber natürlich ihren Zweck. Die Qualität der Regeln ist zunächst auch nicht entscheidend. Wichtig ist nur, daß sie konsequent automatisch angewendet werden und dadurch die Willkürlichkeit heutiger Designs reduziert wird.

Das Ziel ist also, zu zeigen, daß aus einem abstrakten Design alle Implementierungsentscheidungen automatisch abgeleitet werden können und die resultierende Implementierung des Softwaresystems optimal im Sinne des verwendeten Regelsatzes ist. Andere, bessere Regeln führen sicher zu anderen Implementierungen, die aber wiederum optimal in Bezug auf diese Regeln sind.

Nun stellt sich die Frage, wie das Design eines Softwaresystems spezifiziert werden muß, damit überhaupt Regeln angewendet werden können. Damit willkürliche, 'gemeinte' Entscheidungen umgangen werden können, ist es sinnvoll, den Designvorgang dort zu beenden, wo konkrete Entscheidungen notwendig werden. Dies bedeutet, daß das Design des Systems möglichst abstrakt erfolgen muß. Gleichzeitig müssen aber alle für die Anwendung der Regeln notwendigen Informationen vorhanden sein.

Aus diesem Grund wurde die Spezifikationssprache *Graphical-Concept-Network (GCN)* entwickelt. Ein höheres Abstraktionsniveau konnte dabei erreicht werden durch:

- eine Beschränkung auf wirklich notwendige Konstrukte,
- Verzicht auf Typen,
- Verzicht auf Klassendefinitionen.

Ein Graphical Concept Network modelliert ein Begriffsbild. Das Begriffsbild wird beschrieben durch Begriffe und die Beziehungen zwischen den Begriffen. Die Erstellung eines Designs mit GCN Modellen hilft aber nicht, das Begriffsbild zu finden. Es muß vorher bereits existieren. Um das Begriffsbild des zu lösenden Problems zu finden, können aber bekannte und gängige objekt-orientierte Techniken eingesetzt werden, beschrieben z.B. in [BD00] [Boo94]. Zu betonen ist, daß die Qualität des Begriffsbildes auch bei GCN Modellen direkten Einfluß auf die Qualität des Designs hat. Ein schlechtes Begriffsbild ergibt trotz aller Regeln ein schlechtes Design.

In der Praxis stellt es häufig ein großes Problem dar, von der Anforderungsdefinition (Use-Cases [Fow97]) zu einem guten objekt-orientierten Design (Klassendiagramm) zu kommen. Diese Aufgabe ist meist sehr komplex und deswegen sind die notwendigen Zusammenhänge nicht im

ersten Anlauf vollständig erfaßbar. Die Folge sind meist mehrere, zeitaufwendige Versuche, bis das endgültige Modell fertiggestellt ist.

Das Design mit GCN Modellen ist dagegen Use-Case getrieben, d.h. ein Design entsteht Stück für Stück anhand der Use-Cases aus der Anforderungsanalyse. Es wird so immer ein kleinerer Ausschnitt und damit ein überschaubarer Teil modelliert. Die Summe der Use-Cases ergibt dann das gesamte Design des Systems. Dadurch ist ein direkterer Übergang von der Anforderungsdefinition zum Design möglich. Das System setzt dann automatisch die einzelnen Teildesigns zusammen und leitet daraus das Gesamtdesign ab.

Als Grundlage für die bereits vorgestellte Architektur zur komponentenbasierten Entwicklung von Anwendungen ohne Glue-Code wird eine aktive Realzeitdatenbank benötigt. Existierende kommerzielle Datenbanken bzw. Forschungsprototypen sind dafür allerdings nicht geeignet.

Um die Architektur zu realisieren und in der Praxis testen zu können, ist der Entwurf und die Implementierung einer aktiven Realzeitdatenbank notwendig. Daher bot es sich an, diese Datenbank als Anwendungsbeispiel zu wählen. Dieses Beispiel ist auch umfangreich genug, die Praxistauglichkeit des Verfahrens nachzuweisen.

3. Graphical Concept Network

In der Softwareentwicklung werden heute verstärkt Komponententechnologien eingesetzt, um durch Wiederverwendung die Kosten und Entwicklungszeiten zu senken. Um Komponenten effektiv einsetzen und wiederverwenden zu können, darf ihr Entwurf nicht willkürlich geschehen. Der Designvorgang muß so gestaltet werden, daß Entscheidungen, die den Umfang und die Aufteilung der Komponenten bestimmen, automatisiert durchgeführt werden können. Der Designer darf deshalb keine Entscheidungen vorwegnehmen können, die die automatische Bestimmung der Komponenten beeinflußt. Ein Design auf dem Level von Programmiersprachen ist zu konkret und nimmt alle wichtigen Designentscheidungen bereits vorweg.

Das in diesem Kapitel informell vorgestellte *Graphical Concept Network (GCN)* [BW01] ist eine abstrakte Spezifikationssprache, die keine impliziten Designentscheidungen fordert. Sie ist allgemein verwendbar und ist nicht beschränkt auf eine spezielle Anwendungsdomäne. Dem Entwurf der Sprache lagen folgende Anforderungen zugrunde:

- sie sollte intuitiv verständlich sein,
- sie sollte keine frühzeitigen Designentscheidungen erzwingen,
- sie soll vollständig sein (Schleifen, Bedingungen, arithmetische Operationen),
- sie sollte alle notwendigen Details enthalten, um aus dem Modell ablauffähigen Code generieren zu können.

Nach einer Einordnung wird die Syntax von GCN Modellen an einem kleinen Beispieldesign Schritt für Schritt eingeführt. Anschließend werden einzelne Syntaxelemente und Besonderheiten des GCN Modells genauer betrachtet.

3.1. Einordnung

Der Softwareentwickungsprozeß wird gewöhnlich in vier Hauptaktivitäten eingeteilt: die *Analyse*, das *Design*, die *Implementierung* und den *Test*. Diese Arbeit beschäftigt sich fast ausschließlich mit der zweiten Phase, dem eigentlichen Design eines Softwaresystems.

Ein Softwareentwicklungsprozeß, z.B. das *V-Modell* [V-M01] oder der *Rational Unified Process* [Kru00], dient dazu, die einzelnen Aktiviäten bei der Softwareentwicklung zu koordinieren. Es ist allerdings nicht Gegenstand dieser Arbeit, einen geeigneten Entwicklungsprozeß für GCN Modelle zu entwickeln. Das GCN ist (zumindest bisher) ausschließlich eine Beschreibungssprache.

In der Analyse werden die wesentlichen Begriffe des zu lösenden Problems extrahiert [Boo94], sie liefert das Begriffsbild der Anwendungsdomaine. Diese Begriffe bilden die Grundlage des Designs.

Ein Design setzt sich aus vielen Begriffen zusammen, die miteinander verknüpft werden. Begriffe in einem Design interagieren mit anderen Begriffen, sie werden z.B. als Parameter übergeben und kommunizieren über Nachrichten.

Wie bereits erwähnt, stammen einige Begriffe aus der Analyse, (viele) weitere stammen aus dem Designvorgang selbst. Zusammen ergeben sie eine Sammlung von Begriffen, die miteinander in Beziehung gesetzt werden, das *Begriffsbild*. Dabei ist nicht von Anfang an klar, welche Rolle die Begriffe im Sinne eines Softwaredesigns haben. Ob ein Begriff eine Variable, Klasse oder eine Komponente ist, ist beim Design zunächst nicht wichtig, sondern ergibt sich im Laufe des Designs automatisch.

In gängigen objekt-orientierten Spezifikationssprachen muß man sich hier jedoch frühzeitig festlegen. Die *Unified Modelling Language* [OMG00][Fow97] (UML) kann heute als Standardnotation für objekt-orientierte Analyse und Design angesehen werden. Sie soll daher hier stellvertretend betrachtet werden. Die UML besteht aus verschiedenen Diagrammarten, die unterschiedliche Sichtweisen auf ein Softwaresystem erlauben. Ziel eines Komponentendesigns ist eine möglichst vollständige Beschreibung durch *Klassendiagramme*.

Klassendiagramme modellieren die Klassen und Interfaces, ihre Eigenschaften und ihre Beziehungen untereinander. Sie werden im Laufe des Designs immer detaillierter und führen zu einer sehr detailreichen Spezifikation des Systems. Es sind allerdings viele Entscheidungen zu treffen: Was wird zu einer Komponente? Welche Interfaces gibt es und wie sieht das Klassenmodell genau aus? Ein fertiges Design enthält alle Details und Entscheidungen, um daraus eine Implementierung abzuleiten. Es ist bereits entschieden, welche Klassen es gibt, welche Attribute und Methoden sie besitzt, welche Interfaces von welcher Klasse implementiert werden und welche Klassen zu welcher Komponente gehören [Boo94].

All diese Entscheidungen sind in weiten Teilen aber willkürlich. Sie werden getroffen, weil der Designer sie für richtig hält. Es ist daher sinnvoll, den Designvorgang dort zu beenden, wo die konkreten Entscheidungen notwendig werden.

Hier setzt das Graphical Concept Network an. Es fordert keine konkreten Designentscheidungen. Es modelliert nur Begriffe und ihre Beziehungen untereinander. GCN Modelle haben dadurch Ähnlichkeiten zu *Mindmaps* [Car00] oder zu *semantischen Netzen* [Min75]. Mindmaps werden oft eingesetzt, um zusammengehörende Begriffsgruppen zu finden und zu visualisieren. Sie sind nicht auf die Informatik beschränkt und werden in vielen Gebieten eingesetzt. Semantische Netze dagegen werden hauptsächlich im Bereich der Künstlichen Intelligenz (KI) eingesetzt, um Wissen zu repräsentieren. Ein semantisches Netz ist charakterisiert durch Knoten (eigentlich auch Begriffe), die durch Kanten verknüpft werden. Die Kanten tragen die semantische Information (z.B. *isA*, *has*, *belongsTo*, etc.). Semantische Netze haben die gleiche Ausdrucksmächtigkeit wie die Prädikatenlogik. Dies ist aber nicht immer ausreichend zur Erstellung von Softwaredesigns. Es fehlt u.a. die Möglichkeit, Bedingungen oder Schleifen auszudrücken.

GCN Modelle besitzen auch gewisse Ähnlichkeiten zu *Sequenzdiagrammen* der UML. Sie werden in UML verwendet, um den Nachrichtenfluß zwischen einer Menge von Objekten zu visualisieren. Häufig werden sie bereits in der Analyse verwendet. Sequenzdiagramme sind allerdings ohne explizite Erweiterung nicht geeignet, mathematische Operationen und damit Algorithmen auszudrücken. Eine Darstellung von Übergabe-Parametern innerhalb von Sequenzdiagramme führt leider zu sehr unübersichtlichen Diagrammen, v.a. wenn diese weitere Beziehungen haben. Die Modellierung von Iterationen ist zwar möglich, aber nicht detailliert genug, um daraus Code generieren zu können (es fehlen z.B. Abbruchkriterien). Obwohl es sicher möglich wäre, Sequenzdiagramme so zu erweitern, daß sie alle gestellten Anforderungen an die Spezifikationssprache erfüllen könnten, erscheint dies nicht sinnvoll.

GCN Modelle sind auch verwandt mit Konzepten der visuellen Programmierung (ein guter Überblick findet sich in [Sch98]). Dieser Begriff umfaßt ein sehr breites Gebiet, z.B. auch die Modellierung von graphischen Benutzeroberflächen. Wir beschränken uns hier jedoch auf visuelle Programmiersprachen. Folgt man der Definition aus [Sch98][1], so ist das GCN Modell eine visuelle Programmiersprache. Es ist sogar eine Universalprogrammiersprache, denn sie hat die Ausdruckskraft von WHILE-Programmen, die bekanntlich die Menge der Touring-berechenbaren Algorithmen abdecken (siehe z.B. [Sch92])[2].

Bisherige Ansätze zur visuellen Programmierung sind meist ungeeignet zur Bestimmung eines optimalen Komponentendesigns. Entweder werden wie in UML Designentscheidungen impliziert, wie z. B. in *Vista* [Sch98], das auf Smalltalk aufbaut. Oder sie eignen sich hauptsächlich für spezielle Anwendungsgebiete, wie z. B. *VisaVis* [Pos96], das einen streng funktionalen Charakter hat (es fehlen u.a. Schleifen). Oder sie eignen sich nicht zum objekt-orientierten Design, wie z. B. *LabView* [Nat01].

Aus dem GCN Modell wird durch einen Compilevorgang (sh. Kapitel 5) ein konkretes Design, wobei alle notwendigen Designentscheidungen vom System auf Grund von *Regeln* getroffen werden. Das resultierende Design kann vollständig z. B. in UML dargestellt und direkt implementiert werden.

Das System stützt sich auf eine Menge von *Regeln*. Regeln begründen immer dann ein Vorgehen, wenn eine Annahme nicht durch formale Herleitung begründet werden kann. Sie geben den Rahmen vor, wie aus einem GCN Modell das bestmögliche konkrete Design bestimmt wird. Die Regeln entstanden aus der Erfahrung langjähriger Softwareentwicklung und sind meist intuitiv verständlich.

Im nächsten Kapitel werden anhand eines Beispiels die Grundlagen eines GCN Modells erläutert. Es folgen Abschnitte mit einer genaueren Betrachtung ausgewählter Syntaxelemente. Die Syntaxbeschreibung ist aber keineswegs vollständig. Für eine vertiefte Betrachtung aller Syntaxelemente und ihrer Anwendung sei auf [WB01] verwiesen.

3.2. Beispiel

Dieses Kapitel dient der Einführung der graphischen Modellierungssprache. Die Beschreibung ist nicht vollständig, da es nur einen Eindruck vermitteln soll. Schritt für Schritt wird die Syntax anhand eines nicht zu komplexen Beispiels eingeführt: Der Startvorgang eines Motors (Diesel und Benziner) soll modelliert werden[3].

Die Komplexität von realen Designs kann sehr groß sein. Daher besteht ein graphisches GCN-Modell aus beliebig vielen *Sheets*, die hierarchisch gruppiert werden können. Ein Sheet wird in einem eigenen Window dargestellt und modelliert einen kleinen Ausschnitt des Gesamtdesigns. Man kann sich eine Sheet-Sammlung als eine Art Use-Case vorstellen. Das Beispiel besteht aus den Sheets 'StartEngine', 'StartPetrolEngine', 'StartDieselEngine', 'GetSecond' und 'RunEngine'.

Die Grundlage eines GCN-Modells sind Begriffe und Verbindungen zwischen den Begriffen. Verbindungen zwischen Begriffen können u.a. durch Nachrichten, Attribute, Alternativen oder

[1] Eine visuelle Programmiersprache ist eine visuelle Sprache zur vollständigen Beschreibung der Eigenschaften von Software. Sie ist entweder eine Universalsprache oder eine Spezialprogrammiersprache [S.64].

[2] der Beweis ist einfach: für jedes Konstrukt eines WHILE-Programms gibt es ein direktes Ausdrucksmittel in GCN

[3] Das Beispiel ist natürlich sehr vereinfacht. Motorenentwickler mögen mir verzeihen.

Übergabeparameter entstehen. Im Beispiel kommen alle diese Verbindungsarten vor.

Der Motor soll (aus Sicht des Modells von außen) mit der Nachricht “Start“ gestartet werden. Der Motor (’Engine’) hat zwei Alternativen, ’DieselEngine’ und ’PetrolEngine’. Der Sachverhalt wird in Abbildung 3.1 in GCN-Syntax dargestellt.

Der Aufruf von außen (an den Systemgrenzen) wird durch einen *externen Aktor* eingeleitet (er wird durch das Symbol ’Person’[4] dargestellt). Ein Begriff wird als Knoten mit dem Namen des Begriffes dargestellt (hier ’Engine’, ’DieselEngine’ und ’PetrolEngine’).

Begriffe müssen vor der ersten Benutzung generiert werden. Dazu dient ein *Generatoraufruf*, dargestellt durch eine gestrichelte Linie vom externen Aktor zum Begriff ’Engine’.

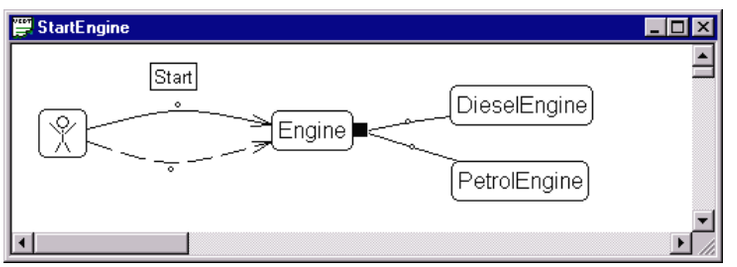


Abbildung 3.1.: Beispiel: Senden der Nachricht “Start“ von außerhalb des Modells.

Damit ist das erste Sheet bereits fertig.

Es ist offensichtlich, daß das Modell an dieser Stelle nicht vollständig ist. Die Nachricht “Start“ bewirkt selbstverständlich etwas. Dies wird in einem weiteren Sheet modelliert. Das neue Sheet stellt eine *Verfeinerung* der Nachricht dar. Auch der Generator kann verfeinert werden, dies wird hier allerdings nicht weiter betrachtet.

Diesel- und Benzinmotoren müssen beim Start unterschiedlich behandelt werden. Der Dieselmotor muß beim Kaltstart vorglühen, der Benziner nicht. Betrachten wir daher zunächst den einfacheren Fall des Benzinmotors (Abbildung 3.2).

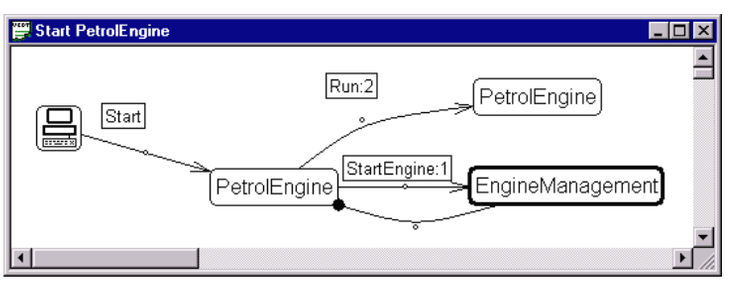


Abbildung 3.2.: Beispiel: Verfeinerung der Nachricht “Start“ für den Benzinmotor.

Eingeleitet wird eine Verfeinerung durch einen *internen Aktor* (Symbol ’Computer’). Es folgt die Nachricht, die verfeinert werden soll, hier also “Start“. Der Benzinmotor hat als Attribut ein Steuergerät (’EngineManagement’). Attribute werden mit einem kleinen, schwarz gefüllten Kreis angeschlossen. In welchem Sheet Attribute zu einem Begriff definiert werden ist beliebig. Die Definition wäre auch im ersten Sheet möglich gewesen. Es wäre sogar erlaubt, einen Teil der Attribute im ersten Sheet und weitere Attribute in anderen Sheets anzulegen.

[4]die Ähnlichkeit zu Aktoren in UML Use-Cases ist beabsichtigt

Das Starten des Benzinmotors reduziert sich auf das Schicken der Nachricht "StartEngine" an das Steuergerät und das anschließende Senden der "Run"-Nachricht an den Motor. Da die Realisierung des Steuergerätes als gegeben vorausgesetzt wird und nicht Teil des Designmodells ist, ist der Begriff 'EngineManagement' als *Endbegriff* ausgeführt (dicke Umrandung). Endbegriffe zeigen an, daß das Modell an dieser Stelle nicht weiter verfeinert wird. Sie dienen als Schnittstelle zur realen Programmierumgebung (z.B. Klassenbibliotheken) und besitzen daher einen konkreten Typ (in diesem Fall z.B. *class EngineManagement*).

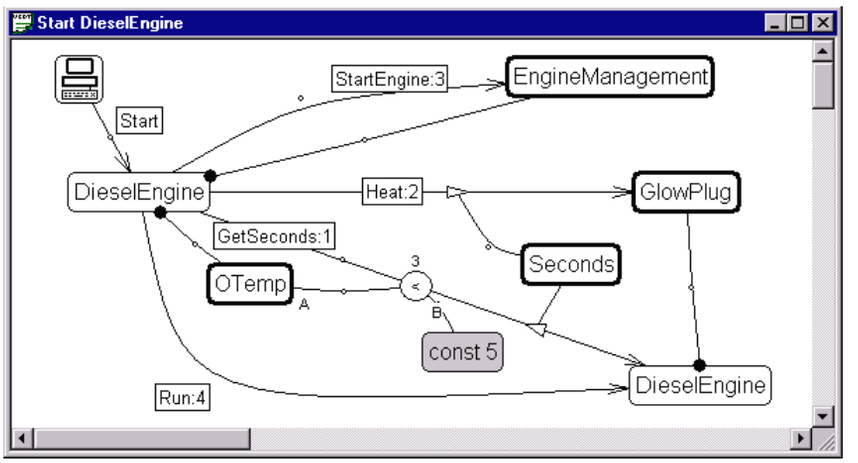


Abbildung 3.3.: Beispiel: Verfeinerung der Nachricht "Start" für den Dieselmotor.

Nun zur Verfeinerung des Startvorgangs beim Dieselmotor (Abbildung 3.3). Im Unterschied zum Benzinmotor wird der Dieselmotor vor dem eigentlichen Starten noch vorgeglüht, wenn die Außentemperatur unter 5 Grad beträgt.

Der Vorgang setzt sich aus vier Nachrichten zusammen, "GetSeconds:1", "Heat:2", "StartEngine:3" und "Run:4". Sie sind mit Nummern versehen, die die zeitliche Reihenfolge der Aufrufe festlegen. Wenn die Reihenfolge der Nachrichten innerhalb eines Sheets nicht beliebig ist, muß sie eindeutig festgelegt werden. Hierzu erhalten die Nachrichten Zeitmarken der Art ":< *number* >".

Zunächst wird die Vorglühdauer bestimmt, aber nur wenn die Außentemperatur (Begriff 'OTemp') unter 5 Grad (der Begriff '5' ist als Konstante definiert) beträgt. Eine Bedingung ist als kleiner Kreis dargestellt, der in seiner Mitte das Vergleichssymbol trägt. Sie sichert das Senden einer Nachricht (hier "GetSeconds") ab, auf der sie angebracht ist. Die Nachricht wird nur gesendet, wenn die Bedingung erfüllt ist. Über der Bedingung sieht man (falls angegeben) den ELSE-Fall, hier die '3'. Die Zahl besagt, daß bei Nichterfüllung der Bedingung mit der Nachricht 3 fortgefahren wird.

Die Reihenfolge der Operanden bei Vergleichen spielt fast immer eine Rolle, daher werden sie grundsätzlich mit A und B bezeichnet. Das Ergebnis des Vergleichs ist immer $A \otimes B$, wobei '$\otimes$' für die Vergleichsoperation steht.

Die Zeitdauer des Vorglühens hängt von der Außentemperatur ab. Sie wird explizit durch die Nachricht "GetSeconds" berechnet. Die Nachricht hat einen *Parameter*, den Rückgabewert "Seconds". Er ist ein Endbegriff mit Typ *short*. Nachrichten können jeweils beliebig viele Input-, Output- oder Input/Output-Parameter besitzen. Dabei wird ausschließlich die Richtung des Datenflusses festgelegt, nicht wie die Übergabe technisch realisiert wird (z.B. ob als Rückgabewert, by reference oder by value).

Der Begriff 'Seconds' wird dann als Input-Parameter für die Nachricht "Heat:2" verwendet. Der Empfänger der Nachricht "Heat" ist die Glühkerze ('GlowPlug'), die als Endbegriff wieder als gegeben vorausgesetzt wird, und als Attribut zum Dieselmotor gehört.

Die Nachrichten "StartEngine" an den Begriff 'EngineManagement' und "Run" an den Begriff 'DieselEngine' sind bereits vom Benzinmotor bekannt.

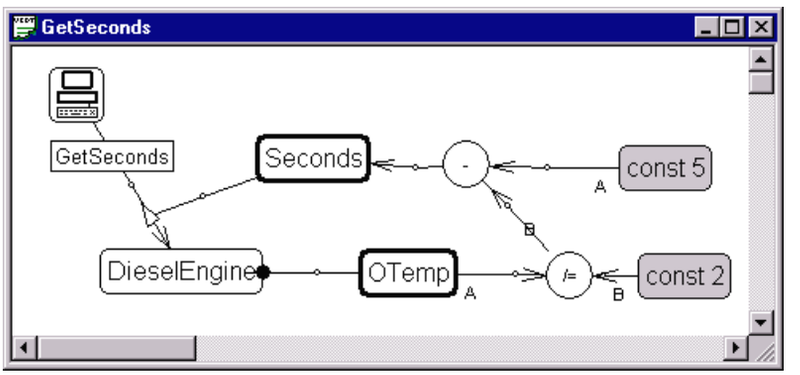


Abbildung 3.4.: Beispiel: Verfeinerung der Nachricht "GetSeconds" durch mathematische Operationen.

"GetSeconds" wird nicht an einen Endbegriff geschickt und muß daher in einem weiteren Sheet noch verfeinert werden (Abbildung 3.4). Die Dauer der Vorglühzeit ist hier linear von der Temperatur abhängig, die Formel lautet: $Seconds = 5 - (OTemp/2)$. Je kühler es wird, desto länger muß vorgeglüht werden.

In GCN Modellen werden Berechnungen mit mathematischen Operationen beschrieben. Diese werden durch Kreise dargestellt, in deren Mitte sich das Operationszeichen befindet. Pfeile verbinden die Kreise mit den Operanden. Wie bei Bedingungen werden die Operanden mit A und B gekennzeichnet. Es gilt dabei immer:

$$\text{'Begriff'} = \text{'Begriff an A'} \otimes \text{'Begriff an B'}$$

Kapitel 3.7 widmet sich ausführlicher den arithmetischen Operationen.

Zur Fertigstellung des Beispielmodells fehlt noch die Verfeinerung der "Run" Nachricht. In diesem Beispiel sei die Verfeinerung dieser Nachricht für Diesel- und Benzinmotoren identisch und sehr einfach. In einer Schleife wird ständig Kraftstoff eingespritzt, bis ein Flag zur Beendigung des Vorgangs gesetzt wird.

Abbildung 3.5 zeigt den Vorgang in GCN-Syntax. Interessant sind hier vor allem drei Dinge:

- Die Verfeinerung der Nachricht gilt nicht explizit für Diesel- oder Benzinmotoren, sondern allgemein für Motoren ('Engine').
- Es wird eine Nachricht innerhalb einer *Schleife* gesendet. Eine Schleife umfaßt eine Menge von Nachrichten (hier nur die Nachricht "Inject"), die durch Angabe einer Schleifenbezeichnung (hier '1') - abgetrennt durch ein Komma - gruppiert werden. Falls die angegebene Bedingung nicht erfüllt ist, wird die Schleife beendet ('break' als ELSE).
- Durch eine Zuweisung wird einem Begriff (hier 'Running') ein bestimmter Wert (hier 'TRUE', wieder eine Konstante) zugewiesen. Die Zuweisung hat zusätzlich eine explizite Zeitangabe.

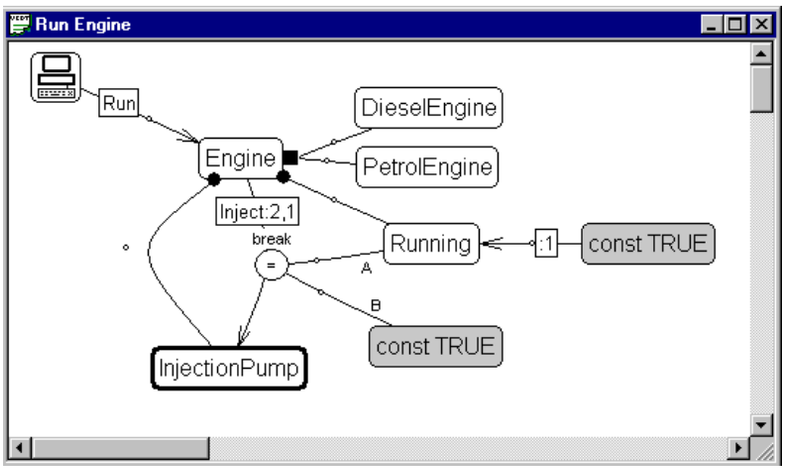


Abbildung 3.5.: Beispiel: Verfeinerung der Nachricht “Run“.

An dieser Stelle ist das Beispielmodell vollständig. Alle Methoden sind entweder Libraryaufrufe oder genau genug spezifiziert.

Ein Design entsteht normalerweise nicht im freien Raum, sondern fügt sich in eine schon bestehende Umgebung ein. Aus diesem Grund gibt es im Design neben den Endbegriffen auch Interfaces, um vorhandene Komponenten einzubeziehen. Sie werden vom System als gegeben hingenommen und ohne Änderung übernommen.

Tabelle 3.1 zeigt die Syntaxelemente eines GCN-Modells. Hier erscheinen auch Elemente, die im Beispiel keinen Platz fanden. Alle Syntaxelemente zusammen erlauben es, ohne Einschränkung beliebige Modelle zu erstellen.

3.3. Alternativen und Identitäten

Alternativen und Identitäten haben eine sehr enge Beziehung. Identitäten wurden im Beispielmodell nicht verwendet. Dieses Kapitel stellt diese beiden sehr mächtigen Syntaxelemente detaillierter vor. Trotz der Verwandschaft sollen sie zunächst getrennt voneinander vorgestellt werden.

Identitäten

In der Umgangssprache werden häufig Synonyme verwendet. Begriffe, die anders genannt werden, aber eigentlich dasselbe aussagen. Der Gebrauch von Synonymen ist dem Menschen so geläufig, daß er nur durch große Anstrengung verhindert werden kann. Es wird daher bei GCN Modellen erst gar nicht versucht, dies zu verhindern. Die Identität wird deshalb explizit als Syntaxelement eingeführt. Sie wird dargestellt durch eine direkte Linie zwischen den beiden Begriffen. Die Identität hat keine Richtung und gilt gegenseitig.

Die Wirkung der Identität wird durch folgende Regel sehr gut charakterisiert:

Regel 1 *Zwei Begriffe teilen ihre* gemeinsamen *Erfahrungen, d.h. ihre Attribute, Nachrichten und Alternativen. Unter gemeinsamer Erfahrung werden alle Eigenschaften verstanden, die sie während der Identität der Begriffe erfahren.*

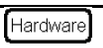
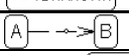
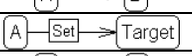
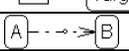
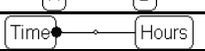
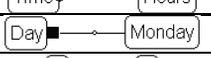
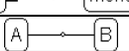
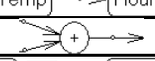
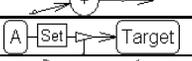
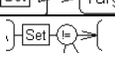
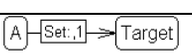
Syntaxelement	Name	Bedeutung
	Externer Aktor	Repräsentiert die Außenwelt, die das Modell stimuliert.
	Interner Aktor	Repräsentiert einen modellinternen Begriff, der im aktuellen Sheet verfeinert werden soll.
	Begriff	Begriff des Modells. Ein Begriff hat keinen Typ und ist in seiner Anwendung nicht eingeschränkt.
	Instanz	Instanz eines Begriffes in einen Sheet.
	Array	Begriffe können Array bilden (hier: 4 Elemente) mit fester oder variabler Länge.
	Endbegriff	Modell-externer Begriff. Endbegriffe werden nicht verfeinert und haben einen Typ.
	Konstante	Begriff mit konstantem Wert.
	Konfigurationsdatum	Begriff wird aus Konfigurationsdatei gelesen.
	Interface	Definiert ein gegebenes Interface.
	Generator	Erzeugt einen Begriff (Konstruktor).
	Nachricht	Es wird eine Nachricht an “target“ geschickt.
	Release	Zerstört einen Begriff (Destruktor).
	Attribute	Der Begriff “Hours“ ist Attribut des Begriffes “Time“.
	Alternative	Ein Begriff kann beliebige viele Alternativen haben.
	Identität	Zwei Begriffe sind gleich.
	Zuweisung	Ein Begriff wird einem anderen zugewiesen.
	Operation	Mathematische Operation mit den Werten der Begriffe.
	Parameter	Übergibt einen Parameter vom linken an den rechten Begriff.
	Bedingung	Die Nachricht “Set“ wird nur geschickt, wenn die Bedingung erfüllt ist (hier: ungleich).
	Schleife	Eine mit Komma abgetrennte Bezeichnung definiert eine Schleife. Verschachtelte Schleifen sind durch weitere Kommata möglich.

Tabelle 3.1.: Syntax Elemente des GCN Modells.

Zur Erläuterung dieser Regel betrachten wir Abbildung 3.6. Sie zeigt eine Identität von zwei Begriffen *Concept A* und *Concept B*. Die Wirkung der Identität zwischen den Begriffen ist in Abbildung 3.7 dargestellt. Alternativen, Attribute und Nachrichten auf einen Begriff gelten durch die Identität auch für den jeweils anderen Begriff. Erfahrungen, die sie einzeln vor der Identität gemacht haben, werden im Moment der Identitätsbildung *nicht* übertragen.

Werden dieselben Begriffe in einem anderen Sheet nicht identisch erklärt, so teilen sie dort auch nicht ihre Erfahrungen.

Der Typ eines Begriffes (z.B. Konstante, Interface oder Endbegriff) wird durch eine Identität nicht berührt, also nicht abgeglichen. Konflikte werden erst bei der Typbestimmung aufgelöst.

Alternativen

Alternativen sind gewöhnliche Begriffe, die über eine gerichtete Beziehung zu einer Alternative eines anderen Begriffes (Oberbegriff) werden. Sie ordnen einem Begriff eine Menge von Repräsentanten zu. Alternativen werden in der graphischen Beschreibungssprache durch ein kleines Quadrat dargestellt (siehe Beispieldesign).

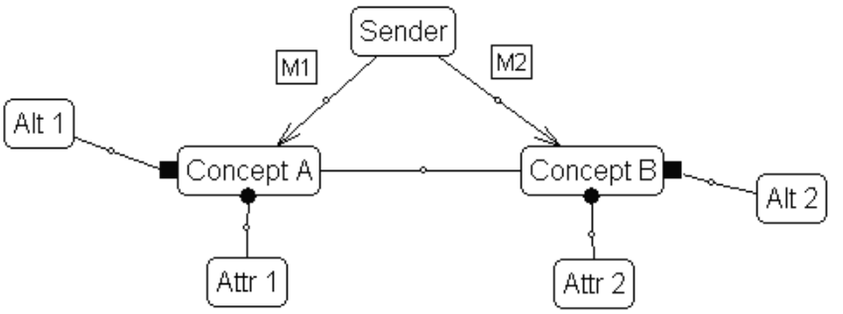


Abbildung 3.6.: Die Begriffe Concept A und Concept B sind identisch.

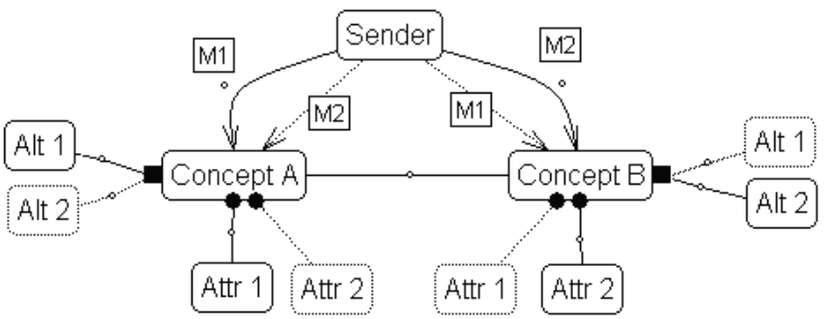


Abbildung 3.7.: Wirkung der Identität aus Abbildung 3.6. Die gestrichelt dargestellten Elemente entstehen durch die Identität.

Alternativen können in ihrer Wirkung sehr unterschiedlich sein. Im einfachsten Fall sind sie als Elemente einer Menge (*enum* in 'C'; *SET* in 'Modula-2') aufzufassen. In diesem Fall müssen alle Alternativen Konstanten sein. Arrays sind als Alternativen (auch wenn alle Elemente konstant sind) nicht erlaubt.

Wenn die Alternativen nicht konstant sind, dann sind sie die Elemente eines *Platzhalters*. Daraus folgt dann die Regel:

Regel 2 *Eine Nachricht an einen Begriff mit Alternativen wird aufgefaßt als eine Nachricht an alle Alternativen.*

Sobald der Oberbegriff eine Nachricht empfängt, werden die Alternativen interpretiert. Der Oberbegriff wird durch einen *Splitter* ersetzt, der an alle Alternativen die Nachricht verschickt. Mit anderen Worten: eine Nachricht an einen Begriff (den Oberbegriff) wird in n parallele Nachrichten gesplittet, die jeweils die Alternative als Empfänger besitzen. Als Beispiel dient das bereits bekannte Sheet "Start Engine"; Abbildung 3.8 zeigt den Begriff *Engine* mit seinen zwei Alternativen Benzinmotor (*PetrolEngine*) und Dieselmotor (*DieselEngine*), der die Nachricht *Start* empfängt und die gedankliche Interpretation dieser Konstruktion. Wenn der Oberbegriff ein Endbegriff ist, gibt es diese Interpretation nicht. Endbegriffe werden per Definition als existent betrachtet und eine weitere Verfeinerung ist nicht vorgesehen. Es macht daher auch keinen Sinn, evtl. vorhandene Alternativen zu interpretieren.

Regel 3 *Wird eine Nachricht verfeinert und hat der Empfänger der Nachricht Alternativen, so gilt die Verfeinerung auch für jede Alternative einzeln.*

Alternativen können auch überall dort eingesetzt werden, wo ein (Teil-)GCN-Design auf mehrere Einzelpositionen zutrifft (in diesem Sinne eingesetzt im Beispieldesign in Sheet "Run Engine").

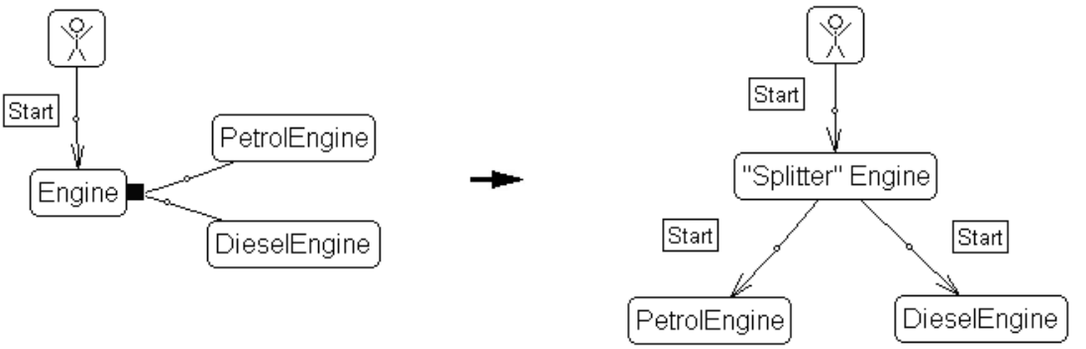


Abbildung 3.8.: Interpretation einer Nachricht an einen Begriff mit Alternativen.

Sie ermöglichen in diesem Fall eine Schreibvereinfachung (besser: Zeichenvereinfachung) und ersparen einem Designer ein mehrfaches Zeichnen eines Sheets mit gleichem Inhalt, aber unterschiedlichen Begriffen. Die Repräsentanten untereinander sind gleichberechtigt und stehen in keiner direkten Beziehung zueinander. Die Interpretation einer Verfeinerung eines Oberbegrif-

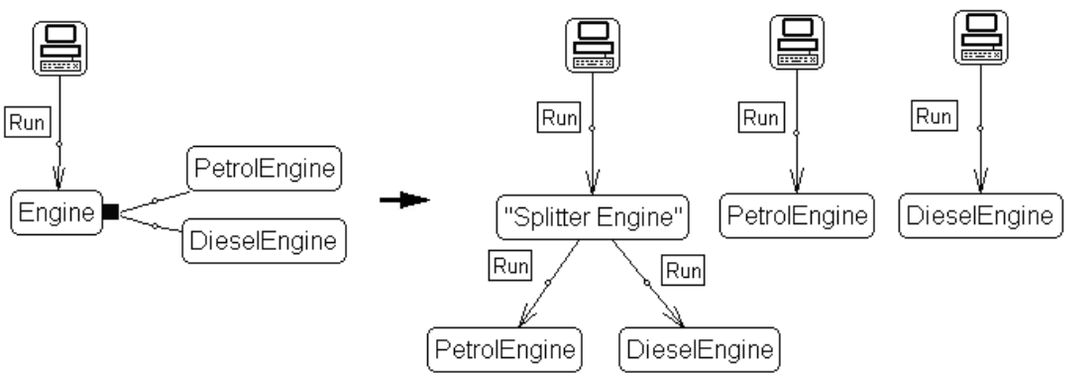


Abbildung 3.9.: Alternativen beim Verfeinern von Nachrichten und ihre Umsetzung.

fes mit Alternativen erfolgt ähnlich wie das Empfangen einer Nachricht an einen Oberbegriff. Der Oberbegriff wird durch einen Splitter ersetzt, der die Nachricht parallel an alle Alternativen sendet. Jede Alternative 'erbt' die Implementierung des Oberbegriffes.

Zusätzlich wird die Verfeinerung interpretiert, als gäbe es für jede Alternative eine eigene Verfeinerung (ohne Oberbegriff). Zur Vereinfachung stelle man sich vor, es gäbe für jede Alternative einen eigenen internen Aktor (Abbildung 3.9) in einem eigenen Sheet.

Am Anfang dieses Kapitels wurde eine enge Beziehung zwischen Alternativen und Identitäten betont. Die Regel, die diese Aussage knapp zusammenfaßt, lautet:

Regel 4 *Jede Identität ist eine potentielle Alternative.*

Dieser Satz bedarf einer Erklärung. Denken wir zunächst nochmal an das Beispiel mit dem Splitter. Dort wurden die Alternativen als Nachrichtenempfänger eingesetzt. Jede dieser Alternativen für sich ist gedanklich identisch zum Oberbegriff. D.h. aber auch, daß eine Verfeinerung für den Oberbegriff verwendet werden kann, falls es keine explizite Verfeinerung für eine oder alle Alternativen gibt.

Immer dann, wenn eine einzelne Alternative an einem Oberbegriff betrachtet wird, kann diese Alternative durch eine Identität mit dem Oberbegriff ausgedrückt werden. Für den Designer ist

dies eine erhebliche Erleichterung, denn er muß bei der Verwendung eines Begriffes nicht wissen, ob der Begriff in Wirklichkeit eine Alternative ist oder nicht. Aber:

Regel 5 *Einmal eingeführte Alternativen werden nicht in Identitäten umgewandelt.*

Eine Alternative ist als Syntaxelement genauer als eine Identität, da sie eine Richtung hat. Es macht daher keinen Sinn die vorhandene Information wieder zu vernichten.

Durch Regel 4 ist auch begründet, daß Attribute eines Oberbegriffes auf die Alternativen übertragen werden, genau so wie es bei den Identitäten beschrieben wurde.

Die folgende Regel dient vor allem einer Zeichenvereinfachung:

Regel 6 *Attribute an einem Oberbegriff gehören allen Alternativen.*

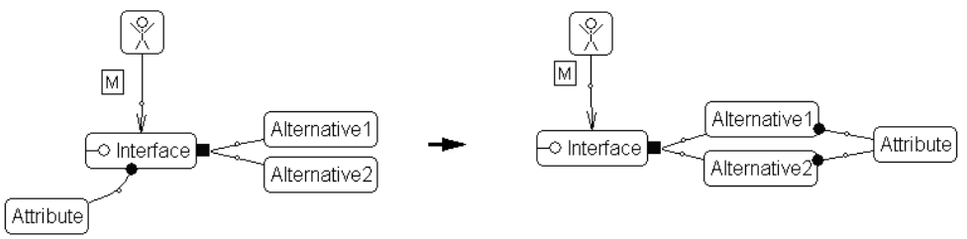


Abbildung 3.10.: Gedankliche Vorstellung von Attributen an einem Oberbegriff.

Durch die GCN Syntax ist es nicht ausgeschlossen, Attribute auch an Interfaces anzuschließen (Abbildung 3.10). Attribute sind an Interfaces zunächst nicht sinnvoll, denn ein Interface ist eine Sammlung von Methoden. Sinnvoll werden sie erst, wenn sie Teil des implementierenden Begriffes werden. Sie ersparen so dem Designer die explizite Angabe der Attribute bei allen implementierenden Begriffen.

3.4. Interfaces

Ein GCN-Modell steht normalerweise nicht alleine im leeren Raum, es muß sich in eine schon vorhandene Umgebung integrieren lassen. Existierende Komponenten mit ihren Interfaces sind meist Bestandteil einer solchen Umgebung. Interfaces können daher in GCN Modellen explizit modelliert werden. Sie werden dann später unverändert übernommen.

Regel 7 *Alternativen an einem Interface werden als Begriffe aufgefaßt, die das Interface implementieren.*

Die Behandlung von Interfaces mit Alternativen unterscheidet sich bis auf eine kleine Ausnahme nicht von Begriffen mit Alternativen. Sie werden beim Empfangen einer Nachricht ebenso durch einen Splitter ersetzt und die Nachricht wird an alle Alternativen geschickt. Dies gilt im Gegensatz zu Endbegriffen auch, wenn das Interface ein Endinterface ist, denn ein Endinterface hat zwar einen festen Typ, aber die Implementierung ist damit noch nicht festgelegt.

Verfeinerungen von Nachrichten an Interfaces werden zunächst auch so behandelt wie Verfeinerungen bei normalen Begriffen. Gedanklich gibt es für jede Alternative einen internen Aktor. Der kleine Unterschied zu normalen Begriffen ergibt sich beim späteren Einhängen der Verfeinerung. Es wird dann zusätzlich das Interface dazwischengeschoben. Die Regel dazu lautet:

Regel 8 *Wenn ein Begriff Alternative eines Interfaces bei einer Verfeinerung ist, wird beim Einhängen der Verfeinerung das Interface zwischen den Aufruf und den Begriff eingeschoben.*

3.5. Generator/Release

Begriffe müssen vor dem Gebrauch bekannt sein. Es gilt daher für ein GCN Modell:

Regel 9 *Jeder verwendete Begriff muß generiert oder definiert sein.*

Um keine Designentscheidung vorwegzunehmen, werden Begriffe durch einen *Generator* erzeugt. Ein Generatoraufruf wird wie eine Nachricht dargestellt, allerdings mit gestrichelter Linie (siehe Abbildung 3.11). Der Generator entspricht im objekt-orientierten Sinne einem Konstruktoraufruf. Auf Parameter wird bewußt verzichtet, obwohl dies den Gebrauch als einfacher Konstruktor einschränkt. Ein Generator kann aber auch komplexere Bedeutung besitzen, wenn ein Begriff z.B. für eine Komponente steht (in COM ist er dann ein *ClassFactory*-Aufruf).

Ein Generator kann wie alle Nachrichten einen Zeitstempel tragen. Er wird dann an der entsprechenden Position ausgeführt. Hat er keinen expliziten Zeitstempel, so wird er als erstes ausgeführt, also als hätte er den kleinsten Zeitstempel im Sheet.

Die nächste Regel beschäftigt sich mit definierten Begriffen:

Regel 10 *Ein Begriff gilt als* definiert, *wenn er*

1. *eine Konstante ist,*
2. *Attribut eines generierten oder definierten Begriffes ist,*
3. *Interface an einem generierten Begriff ist,*
4. *identisch mit einem generierten oder definierten Begriff ist,*
5. *Alternative an einem generierten oder definierten Begriff ist*[5].

Die in Abbildung 3.11 dargestellten Begriffe sind alle generiert oder definiert, was man an Hand obiger Regel leicht nachvollziehen kann. Der Begriff 'Concept 1' wird explizit generiert. Dadurch gilt durch (2) der Begriff 'Attr 1' als definiert und über die Identität (4) auch 'Attr 2'. 'Alt 1' ist definiert auf Grund der 5. Bedingung.

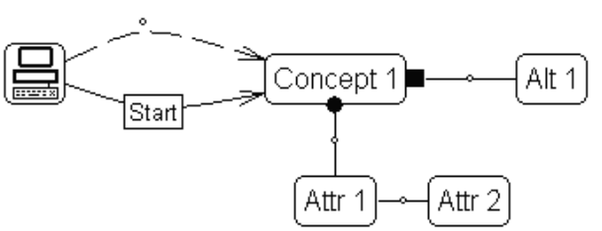


Abbildung 3.11.: Alle Begriffe in diesem Sheet sind generiert oder definiert.

Ein Ausnahme gibt es dennoch: Begriffe an Parameterübergaben von außerhalb des Modells (also bei Nachrichten, die von einem externen Aktor ausgehen) gelten als generiert. Sie müssen schon außerhalb bekannt sein, da durch sie ein gültiger Wert ins Modell gebracht wird. Daraus folgt unmittelbar die nächste Regel:

Regel 11 *Ein Input-Parameter an Systemgrenzen ist existent und muß nicht generiert werden.*

In objekt-orientierten Sprachen gibt es neben dem Konstruktor auch einen Destruktor. Ein Äquivalent zum Destruktor ist auch in GCN-Modellen zu finden, der *Release*. Er wird durch

[5] hier sei auf die Regel 4 verwiesen

eine strichpunktierte Line visualisiert. Ein Release ohne Zeitangabe wird grundsätzlich zuletzt ausgeführt, mit Zeitangabe an der entsprechenden Position im Sheet.

Alle Regeln für das Generieren gelten entsprechend für den Release. Release-Operationen dürfen nur auf Begriffe ausgeführt werden, die bereits generiert wurden.

3.6. Parameterpropagation

Die Aufteilung eines Modells auf viele unterschiedliche Sheets macht den komplexen Designvorgang erst überschaubar. Die Einzelteile werden dann beim Compilevorgang über die Verfeinerungen zu einem Ganzen zusammengefügt.

Parameter transportieren Begriffe (mit ihren Eigenschaften) von einem Begriff zu einem anderen Begriff. Der Transport geschieht nicht nur innerhalb desselben Sheets, sondern auch über Sheetgrenzen hinweg.

Die Eigenschaften eines transportierten Begriffes bestehen im wesentlichen aus seinen Attributen, seinen Alternativen und seinen Identitäten. Diese Eigenschaften sind nicht statisch, sondern können sich ändern.

Damit eine Eigenschaft aus einem übergeordneten Sheet auch im untergeordneten Sheet (der Verfeinerung) erhalten bleiben, müssen transportierte Begriffe *propagiert* werden.

Beispiel: Ein Begriff C besitzt bei der Übergabe zwei Alternativen, im untergeordneten Sheet aber keine. Nach dem Verbinden der Sheets muß er auch im zweiten Sheet beide Alternativen besitzen.

Parameter können in zwei Richtungen propagiert werden: beim Verbinden von zwei Sheets werden alle Parameter vom übergeordneten Sheet ins untergeordnete Sheet (nach unten) propagiert. Beim Rückweg werden zusätzlich Output-Parameter nach oben propagiert.

Propagieren heißt konkret, daß ein übergebener Begriff seine Eigenschaften auf alle Begriffe mit gleichem Namen (genauer: gleicher Identifier) überträgt, d.h. gedanklich den ursprünglichen Begriff ersetzt.

3.7. Mathematische Operationen

Neben den 'herkömmlichen' Operationen gibt es einen Satz von speziellen String-Operationen, die hier ebenfalls (etwas ungenau) unter den mathematischen Operationen zusammengefaßt sind. Sie erleichtern den in vielen Programmiersprachen umständlichen Umgang mit Strings und helfen so, VCDT Modelle kompakt zu halten. Eine ausführliche Erläuterung der Stringoperationen findet sich in [WB01]. Tabelle 3.2 faßt die möglichen arithmetischen Operationen zusammen.

Die einfachste Form einer mathematischen Operationen ist die Zuweisung. Sie ist zugleich Bestandteil jeder anderen Operation, denn das Resultat jeder Operation wird einem Begriff zugewiesen.

Eine Zuweisung $A = B$ kann auch aufgefaßt werden als ein aktives Holen von B durch A, und damit als Nachricht *_get*[6] an B. Wie alle Nachrichten kann eine Zuweisung daher eine Positionsnummer und Bedingungen besitzen.

[6] vom System generierte Namen haben einen vorangestellten Unterstrich, um nicht mit Modellnamen zu kollidieren

Operator	Bedeutung	Operator	Bedeutung
@	explizite Typkonvertierung	?	Speichergröße eines Begriffes
??	Anzahl der Indices eines Arrays	′	Transposition
~	bitweises Komplement	!	NOT
$*=$	Expansion	$/=$	Reduktion
$+=$	Increment	$-=$	Dekrement
<<	Shift nach links	>>	Shift nach rechts
<	kleiner	<=	kleiner, gleich
>	größer	>=	größer, gleich
==	gleich	$!=$	ungleich
+	Addition	−	Subtraktion
*	Multiplikation	/	Division
%	Modulo	^	bitweises XODER
&	bitweises UND	|	bitweises ODER
&&	logisches UND	||	logisches ODER
$\|x\|$	ABS()	min	MIN()
$@	Stringformatierung	max	MAX()
$+	Stringaddition	$ <	String, kleiner
$−	Stringsubtraktion	$ >	String, größer
$:	Stringkürzung	$ <<	String, Shift links
$−	Stringsubtraktion	$ >>	String, Shift rechts

Tabelle 3.2.: Mathematische Operationen des GCN Modells.

Ein Problem bleibt dabei noch: die *_get* Nachricht muß zum richtigen Zeitpunkt ausgelöst (getriggert) werden. Dazu ist ein künstliches Ereignis notwendig, eine *_trigger*-Nachricht an den ergebniserhaltenden Begriff (hier A).

Der Startzeitpunkt für eine Berechnung ist dabei der kritische Punkt. Prinzipiell muß das Ergebnis einer Rechnung bekannt sein, bevor der Wert benutzt wird. Der Triggerzeitpunkt hängt so von der Einsatzart des Ergebnisbegriffs ab. Es können sechs verschiedene Fälle unterschieden werden, die nach der Erweiterung der Zuweisung auf allgemeine mathematische Operationen vorgestellt werden.

Stellvertretend für allgemeine Operationen wird nun eine einfache Addition ($Summe = A + B$), wie in Abbildung 3.12 dargestellt, betrachtet. Sie kann umgeformt werden zur äquivalenten

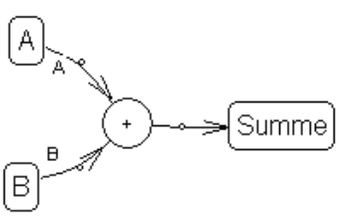


Abbildung 3.12.: Eine Addition als einfache mathematische Operation: $Summe = A + B$.

nachrichtenbasierten Darstellung in Abbildung 3.13. Das Ergebnis einer arithmetischen Operation wird immer einem (einzigen) Begriff zugewiesen (hier *Summe*). Nach der Umformung startet eine Berechnung von diesem Begriff. Die Operationen selbst werden zu Begriffsknoten der Form '*_op* < *symbol* >', die gedanklich das Ergebnis der jeweiligen Operation speichern. Die notwendigen Eingangsdaten besorgt sich die Operation durch *_getA* bzw. *_getB* Nachrichten.

Selbst komplexe Operationen können durch dieses Verfahren problemlos in eine Folge von Nachrichten umgewandelt werden.

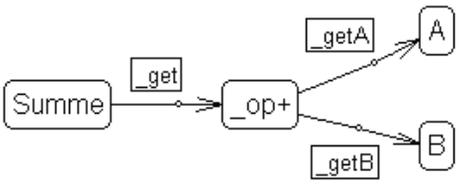


Abbildung 3.13.: Nachrichtenbasierte Umformung der Addition: $Summe = A + B$.

Allerdings muß noch bestimmt werden, wann die Rechnung ausgelöst wird. Es wird dabei unterschieden:

- Der Begriff ist Übergabe-Parameter.
- Der Begriff ist Rückgabe-Parameter.
- Der Begriff ist Teil einer Bedingung.
- Der Begriff ist ein Attribut.
- Der Begriff ist Empfänger einer Nachricht.
- Es gibt eine Zeitangabe (Position) bei der Zuweisung.

Die folgenden Abbildungen zeigen immer links den Ausgangszustand und in der rechten Hälfte die Transformation inklusive der Trigger-Nachricht.

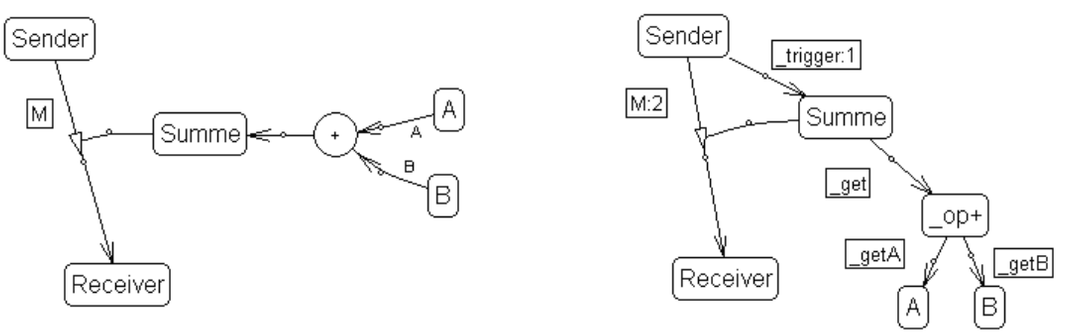


Abbildung 3.14.: Nachrichtenbasierte Umformung einer Berechnung an einem Übergabe-Parameter.

Der aktuelle Wert eines Übergabeparameters muß vor der Übergabe bekannt sein. Die Berechnung erfolgt deswegen genau vor der Nachricht, die den Parameter übergibt (Abbildung 3.14).

Bei einem Rückgabeparameter ist die Situation genau umgekehrt. Ein Rückgabewert wird so spät wie möglich berechnet. Die Berechnung wird deshalb vom Empfänger der Nachricht als letzte Aktion ausgelöst (Abbildung 3.15).

Begriffe an Bedingungen müssen bekannt sein, bevor die Bedingung ausgewertet wird. Der Fall gleicht somit der Parameterübergabe. Auch hier wird die Berechnung vom Sender ausgelöst, bevor die Nachricht mit der Bedingung gesendet wird (Abbildung 3.16).

Ein Attribut gehört einen Begriff. Es ist deswegen die Aufgabe des Besitzers, das Attribut rechtzeitig zu berechnen. Der Attributbesitzer löst die *_trigger* Nachricht sofort nach dem Empfang einer Nachricht aus (Abbildung 3.17).

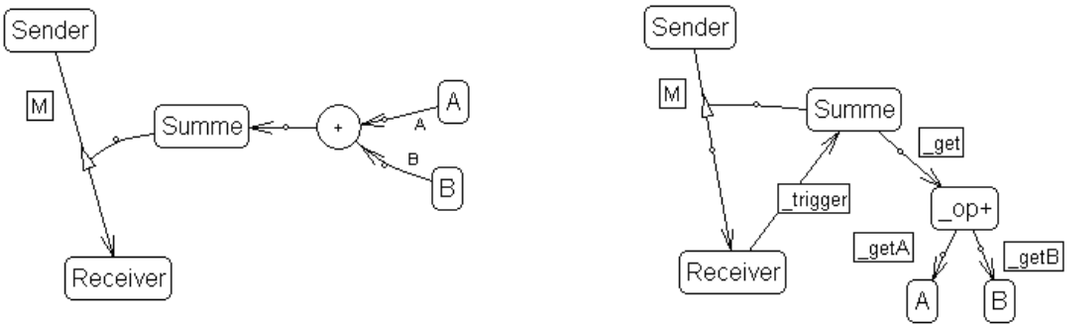


Abbildung 3.15.: Nachrichtenbasierte Umformung einer Berechnung an einem Rückgabe-Parameter.

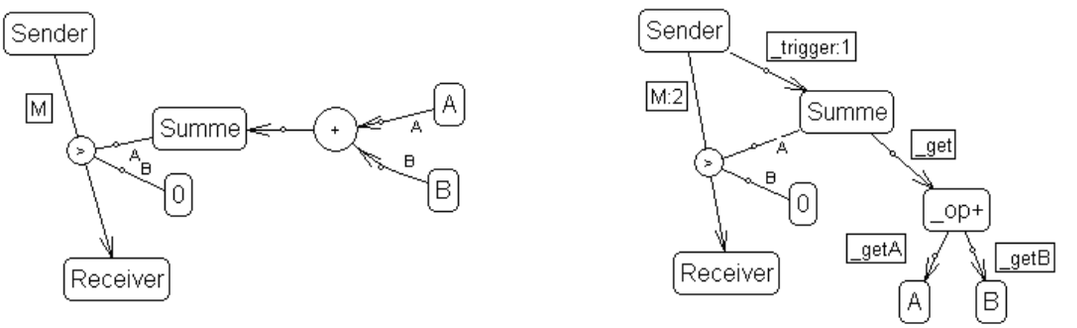


Abbildung 3.16.: Nachrichtenbasierte Umformung einer Berechnung an einer Bedingung.

Falls der Empfänger einer Nachricht selbst Ergebnis einer Operation ist, ist es ausreichend, sich selbst als erste Aktion zu berechnen. In diesem Fall wird nicht einmal eine explizite *_trigger* Nachricht benötigt, die erste *_get* Nachricht reicht aus (Abbildung 3.18).

Für Arrayzugriffe bekommen die Nachrichten einen bzw. zwei Parameter, die den *source*- bzw. *dest*-Index angeben.

Haben Zuweisungen eine explizite Zeitangabe, so werden sie (bzw. die entsprechende *_trigger* Nachricht) genau dort eingereiht, wo sie laut Zeitangabe sein sollen.

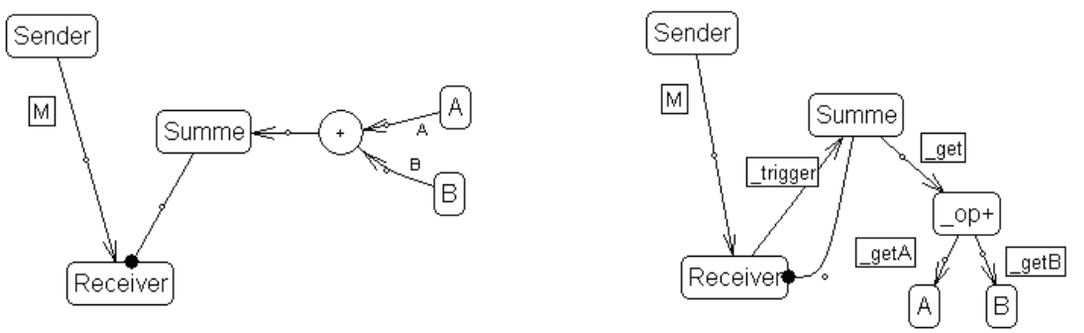


Abbildung 3.17.: Nachrichtenbasierte Umformung einer Berechnung an einem Attribut.

Abbildung 3.18.: Nachrichtenbasierte Umformung einer Berechnung an einem Nachrichtenempfänger.

4. Formale Definition des GCN-Modells

Das folgende Kapitel definiert das in dieser Arbeit verwendete GCN-Modell. Dies geschieht in formaler Form, um Mehrdeutigkeiten zu vermeiden. Die Implementierung einer Toolunterstützung wird so ebenfalls erleichtert. In Kapitel 5 wird dieses Modell dann dem Compilevorgang unterzogen. Im ersten Abschnitt dieses Kapitels wird zunächst die Notation eingeführt. Den zentralen Punkt bilden dann die Beschreibungselemente des GCN-Modells (siehe auch Tabelle 3.1). Im letzten Abschnitt werden dann noch einige häufig benötigte Operationen definiert.

4.1. Notation

In dieser Arbeit werden Mengen groß geschrieben. Mit N wird die Menge der natürlichen Zahlen einschließlich der 0 bezeichnet, $N = \{0, 1, 2, 3, \ldots\}$. Die Menge $N^+ = N \setminus \{0\}$ bezeichnet die Menge der positiven natürlichen Zahlen. Die Menge $BOOL = \{true, false\}$ definiert die boolschen Werte $true$ und $false$, sie werden im bekannten Sinn eingesetzt.

Sei M eine Menge, dann bezeichnet $|M|$ die *Mächtigkeit* der Menge, d.h. die Anzahl der Elemente in der Menge.

Um die Übersichtlichkeit zu wahren, werden nicht betrachtete Teile eines Tupels mit '...' dargestellt. Dies bedeutet gleichzeitig, daß die so ausgeblendeten Elemente nicht benötigt bzw. bei Operationen nicht geändert werden.

4.2. Beschreibungselemente

Das Graphical-Concept-Network stellt die in Tabelle 3.1 dargestellten Syntaxelemente bereit. Jedes Syntaxelement wird in diesem Kapitel formal definiert.

GCN Modelle bestehen aus einzelnen Sheets, ein Sheet definiert sich als:

Definition 1 (Sheet) *Ein Sheet ist das 5-Tupel*

$$s = (name_s, a_s, C_s, M_s, L_s)$$

mit folgenden Eigenschaften:

- $name$ *ist die* Bezeichnung *des Sheets.*
- a_s *ist der* Aktor *des Sheets.*
- C_s *ist eine endliche Menge, die Menge der* Begriffe (concept).
- M_s *ist eine endliche Menge, die Menge der* Nachrichten (message).
- L_s *ist eine endliche Menge, die Menge der* Schleifen (loop).

Ein Sheet beschreibt einen Ausschnitt des Gesamtdesigns. Startpunkt eines Sheets ist ein Aktor. Ausgehend vom Aktor kommunizieren Begriffe über Nachrichten miteinander. Es gibt zwei verschiedene Typen von Aktoren, externe (external) und interne (internal) Aktoren. Aktoren sind wie folgt definiert:

Definition 2 (Aktor (actor)) *Ein Aktor ist das Tupel*

$$a = (type, m, r)$$

mit folgenden Eigenschaften:

- $type \in \{external, internal\}$ *ist der* Typ *des Aktors.*
- m *ist die vom Aktor* gesendete Nachricht.
- r *ist die* Rolle.

Ein *externer Aktor* schickt Nachrichten, die außen (außerhalb des Modells) sichtbar sind. Jeder externe Aktor hat eine Rolle; eine Rolle definiert eine bestimmte *Einsatzart*. Ein Aktor repräsentiert dabei einen speziellen Benutzer des Systems, z.B. 'Administrator' oder 'All Users'.

Ein *interner Aktor* dient zur Einleitung einer *Verfeinerung* genau einer Nachricht. Er ist damit ein Platzhalter; er steht stellvertretend für alle Sender dieser Nachricht. Ein interner Aktor definiert keine bestimmte Einsatzart, denn er wird nur innerhalb des Modells verwendet; er hat daher keine eigene Rolle.

Definition 3 (Rolle) *Ein Rolle ist das Tupel*

$$r = (user, task)$$

mit folgenden Eigenschaften:

- *user ist der* Name *des Aktors.*
- *task ist die* Aufgabe *der gesendeten Nachricht.*

Mehrere externe Aktoren können dieselbe Aufgabe haben (z.B. 'Startup System'). Die Aufgabe bezieht sich dann auf alle Nachrichten, die im Zuge des Vorgangs angewendet werden.

Zentrales Syntaxelement eines GCN Modells sind Begriffe; hier ihre Definition:

Definition 4 (Begriff (concept)) *Ein Begriff ist definiert durch das Tupel*

$$c = (name, ctype, etype, dtype, ATR, ALT, IDENT, m_r, impl_r, ID, TID)$$

Hierbei sind:

- *name der* Name *des Begriffes.*
- $ctype = \{normal, config, const, interface, splitter, ifsplitter\}$ *der* Typ *des Begriffes.*
- $etype = \{normal, end, fromEnv\}$ *der* Endtyp *des Begriffes.*
- *dtype der* Datentyp *des Begriffes.*
- ATR *eine endliche Menge von Begriffen, die Menge der* Attribute.
- ALT *eine endliche Menge von Begriffen, die Menge der* Alternativen.
- $IDENT$ *eine endliche Menge von Begriffen, die Menge der* Identitäten.

- m_r *die* empfangene (receive) Nachricht.
- $impl_r$ *die* Implementierung *der empfangenen Nachricht*
- $ID = \{id_i\}$ *die endliche Menge der* Identifier. $id_i \in N$
- $TID = \{tid_i\}$ *die endliche Menge der* Typ-Identifier. $tid_i \in N$

Ein Begriff hat einen Namen *name*, der nur innerhalb eines Sheets, aber nicht im gesamten Modell eindeutig sein muß.

Jeder Begriff hat einen Begriffstyp *ctype*. Die möglichen Typen sind:

- *Normaler Begriff.* Jeder Begriff, der nicht explizit einen anderen Typ erhält, ist ein normaler Begriff.
- *Konfigurationsdatum.* Ein Begriff wird als Konfigurationsdatum definiert, wenn sein Wert aus einer Konfigurationsdatei eingelesen werden soll (Properties einer Komponente). Ein Konfigurationsdatum muß einen Default-Wert besitzen.
- *Konstante.* Ein Begriff kann als Konstante definiert werden. Sein Wert ist nicht änderbar.
- *Interface.* Ein Begriff kann als Interface definiert werden. Ein Interface im Design wird als gegeben angesehen und ohne Änderung übernommen.
- *Splitter.* Ein Begriff wird dann zum Splitterbegriff, wenn er eine Nachricht empfängt und gleichzeitig Alternativen besitzt. Statt die Nachricht selbst zu verarbeiten, schickt er sie an alle seine Alternativen (siehe Kapitel 3.3).
- *Interface-Splitter.* Ein zum Splitter umgebautes Interface.

Jeder Begriff hat einen von drei möglichen Endtypen *etype*. Ein Begriff kann ein Endbegriff sein, der nicht weiter verfeinert wird und einen festgelegten Typ hat (*end*). Er kann zusätzlich global bekannt sein ($fromEnv$), d.h. er stammt vom Komponentenenvironment oder aus einer Library. Dadurch wird eine Fehlerausgabe bei der späteren Existenzprüfung unterdrückt. Oder er ist ein normaler Begriff, der erst später klassifiziert wird (*normal*).

Jedem Begriff kann ein Datentyp zugeordnet werden. Bei Endbegriffen geschieht dies durch den Designer, bei allen anderen Begriffen später durch die Klassifikation.

Definition 5 (Datentyp (datatype)) *Ein Datentyp ist das Tupel*

$$dtype = (btype, btypename, arraysize)$$

mit folgenden Eigenschaften:

- $btype \in \{int, long, ..., char, enum, component, struct, class, ...\}$ *der* Basistyp.
- *btypename ist der* Basistyp *der Programmierumgebung (als String).*
- $arraySize \in N \cup \{variable\}$ *die* Arraygröße.

Einem *Endbegriff* muß ein realer Typ zugeordnet werden. Um ein Typchaos bei der Klassifikation zu vermeiden, können nur *Basistypen (BaseTypes)* definiert werden. Die Basistypen richten sich nach der RPC-Definition [Maj99]. Bei einfachen Typen wie z.B. $long, float$ oder *char* ist der *basetypename* leer. Bei komplexeren Typen wie *class* oder *struct* enthält *basetypename* den Namen des realen Datentyps (z.B. *struct Date*).

Für *Konfigurationsdaten* kann der Typ abstrakter bestimmt werden. Es stehen nur die Typen $\{boolean, integer, fraction, string, byte, uniqueId, alternative\}$ zur Verfügung. Sie werden erst durch einen Abgleich mit Endbegriffen zu konkreten Typen nach der RPC-Spezifikation.

Ein Begriff kann als Array definiert sein. *arraySize* gibt die Größe des Arrays an; der Wert 0 bedeutet, daß der Begriff kein Array ist, der Wert *variable*, daß die Arraygröße dynamisch bestimmt wird.

Attribute, Alternativen und Identitäten sind Begriffe. Sie werden der entsprechenden Menge des Begriffstupels zugeordnet. Identitäten sind richtungslos, sie werden daher wechselseitig eingetragen, d.h. wenn zwei Begriffe c_1 und c_2 identisch erklärt wurden, gilt: $c_1 = (..., IDENT_1 = \{c_2\}, ...)$ und $c_2 = (..., IDENT_2 = \{c_1\}, ...)$.

Alle Begriffe eines Sheets werden für jede Verwendung als eigenes Tupel c_i angelegt. Falls ein Begriff gleichen Namens in einem Sheet mehrere Nachrichten empfängt, wird für jede empfangene Nachricht ein eigenes Begriffstupel definiert. Die einzelnen Begriffstupel unterscheiden sich dann nicht im Namen, Typ oder den Attributen, sondern nur in der empfangenen Nachricht und der Implementierung der Nachricht. Zu jeder empfangenen Nachricht gehört eine Implementierung *impl*. Ist die Implementierung innerhalb des Sheets nicht gegeben, so ist die Implementierung zunächst nicht bekannt ($impl = \emptyset$).

In einem GCN Modell werden in verschiedenen Sheets Begriffe mit gleichem Namen verwendet. Der Name alleine rechtfertigt noch nicht, daß diese Begriffe als identisch angesehen werden. Man stelle sich hier nur einen Begriff 'MyContainer' vor. In einer Stelle im Design ist er eine Liste, an einer anderen Stelle eine Hashtabelle. Würden beide Vorkommen von 'MyContainer' identisch erklärt, so müßten sie denselben Typ haben (Typproblem). Ein anderes Problem ergibt sich, wenn Begriffe gleichen Names unterschiedliche Instanzen (mit demselben Typ) bilden. Auch in diesem Fall dürfen die Begriffe nicht identisch erklärt werden. Für die Komponentenbestimmung ist es wesentlich, die wirkliche Anzahl der Instanzen eines Begriffe (entspricht den Objekten im OO-Sinne) zu kennen (Instanzproblem). Um diese Probleme zu lösen, werden eindeutige *Identifier* verwendet. Für das Instanzproblem werden die Identifier ID benutzt, für das Typproblem die Typidentifier TID.

Es muß bei der Id-Vergabe folgende Regel beachtet werden:

Regel 12 *Im gleichen Sheet sind Begriffe mit gleichem Namen gleich gemeint. Sie bekommen daher denselben Identifier.*

Begriffe innerhalb eines Sheets werden grundsätzlich nur an ihrem Namen unterschieden. Unterschiedliche Begriffe bekommen unterschiedliche, Modell-eindeutige Identifier. Begriffe gleichen Namens bekommen den selben Identifier. Es gibt jedoch die Möglichkeit, explizit mehrere Instanzen eines Begriffes (mit gleichem Namen) in einem Sheet anzulegen. Jede Instanz dieses Begriffes bekommt dann einen eigenen, eindeutigen Identifier.

Die Mengen der Identifier enthalten zunächst jeweils nur ein Element. Die Identifier treffen sich während des Compilevorganges und bilden dadurch Mengen.

Begriffe können über Ident-Verbindungen identisch zu anderen Begriffen gesetzt werden. Sobald eine Identität zwischen zwei Begriffen besteht, teilen sie ihre gemeinsamen Erfahrungen, d.h. eine Alternative und ein Attribut für einen Begriff wird allen dazu identischen Begriffen hinzugefügt.

Bisher wurde viel von Nachrichten gesprochen, hier nun ihre Definition:

Definition 6 (Nachricht (message)) *Eine Nachricht ist definiert durch das Tupel*

$$m = (mname, mtype, pos, sender, rcv, P, COND)$$

Dabei sind:

- *mname der* Name *der Nachricht.*
- $mtype \in \{generator, release, normal, sysgen\}$ *der* Typ *der Nachricht.*
- $pos \in N$ *die* Reihenfolge (Position) *der Nachricht innerhalb des Sheets.*
- $sender \in \{a_s, c\}$ *der* Sender *der Nachricht. sender kann auch ein Aktor sein.*
- *rcv der* Empfänger *der Nachricht. Der Empfänger ist immer ein Begriff.*
- *P die endliche Menge der* Parameter.
- *COND die endliche Menge der* Bedingungen (Conditions)*, die erfüllt sein muß, damit die Methode ausgeführt wird.*

Die Parameter einer Nachricht sind wie folgt definiert:

Definition 7 (Parameter) *Ein Parameter ist ein 2-Tupel*

$$p = (c_p, d)$$

mit folgenden Eigenschaften:

- c_p *ist der zu transportierende Begriff.*
- $d \in \{in, out\}$ *ist die* Richtung (direction).

Ein Parameter dient dem Transport eines Begriffes c_p von einem Begriff zu einem anderen Begriff über Nachrichten. Parameter haben eine Richtung, sie können Input- oder Output-Parameter sein. Im Gegensatz zu Programmiersprachen wie z.B. C++ kann es beliebig viele Parameter jeder Richtung geben.

Definition 8 (Bedingung (Condition)) *Eine Bedingung ist das 4-Tupel*

$$cond = (symbol, a, b, else)$$

mit folgenden Eigenschaften:

- *symbol repräsentiert die* Operation. $symbol \in \{<, >, <=, >=, ==, !=, ...\}$.
- *a die _get − Nachricht an den linken* Operand.
- *b die _get − Nachricht an den rechten* Operand.
- $else \in N \cup \{break, continue, return\}$ *ist die* Aktion *bei Nichterfüllung der Bedingung.*

Eine Bedingung sichert das Versenden einer Nachricht ab. Nur wenn die Bedingung erfüllt ist, wird die Nachricht geschickt. Die beiden Operanden werden vor der Auswertung der Bedingung über die *_get*-Nachrichten geholt. So können sie vor der Auswertung auch durch eine mathematische Operation bestimmt werden.

Wenn die Bedingung nicht erfüllt ist, wird mit derjenigen Nachricht fortgefahren, die in *else* angegeben wurde. Für $else = return$ gilt: die aktuelle Nachricht wird nicht gesendet und die Abarbeitung wird an den Sender zurückgegeben.

Die Werte *break* und *continue* sind nur sinnvoll, falls die Bedingung (für die Nachricht) innerhalb einer Schleife auftritt. Schleifen sind:

Definition 9 (Schleife (loop)) *Eine Schleife wird beschrieben durch das 3-Tupel*

$$l = (lname, pos, impl_l)$$

mit folgenden Eigenschaften:

- $lname \in N$ *ist die* Bezeichnung *der Schleife. Gewöhnlich werden sie Sheet-lokal durchnummeriert, es ist aber auch eine textuelle Benennung zulässig.*
- $pos \in N$ *die* Position *der Schleife innerhalb des Sheets.* $pos = \min(m_i = (..., pos_i, ...) \in impl_l)$
- $impl_l$ *die* Implementierung *der Schleife.*

Eine Schleife gilt nur innerhalb eines Sheets. Die Schleifenbezeichnung dient der Identifikation, da mehrere Schleifen (auch geschachtelt) innerhalb eines Sheets definiert werden können. Eine Schleife umfaßt eine oder mehrere Nachrichten, die prinzipiell unendlich lange gesendet werden. Der Abbruch (*break*) bzw. die weitere Ausführung (*continue*) einer Schleife wird über Bedingungen gesteuert, die bei den Nachrichten innerhalb der Schleife angegeben werden.

Wir rufen uns in Erinnerung, daß zu einem Begriff die Implementierung einer empfangenen Nachricht gehören kann. Die Definition fehlt noch und soll nun nachgetragen werden:

Definition 10 (Implementierung) *Eine Implementierung ist eine geordnete Menge von Methoden und Schleifen:*

$$impl = \{M_{impl} \cup L_{impl}\}$$

Hierbei sind:

- M_{impl} *eine endliche Menge, die Menge der* implementierenden Nachrichten.
- L_{impl} *die Menge der Schleifen.*
- *es muß gelten:* $pos_i <= pos_{i+1}\ \forall z_i = (..., pos_i, ...), z_{i+1} = (..., pos_{i+1}, ...) \in impl$

Die Implementierung umfaßt im Grunde eine Menge von Nachrichten. Manche Nachrichten werden in einer Schleife geschickt.

Beispiel

Ein Beispiel soll die bisherigen Ausführungen veranschaulichen. Abbildung 4.1 zeigt das bereits bekannte Sheet für die Verfeinerung der "Run"-Nachricht aus Kapitel 3. Das Sheet enthält einen internen Aktor, fünf Begriffe 'Engine', 'DieselEngine' 'PetrolEngine', 'InjectionPump', und 'Running', die Konstante 'TRUE', die (expliziten) Nachrichten "StartEngine" und "Inject", die impliziten Nachrichten (aus der Zuweisung generiert) "_trigger" und "_get", eine Bedingung und eine Schleife. Formal betrachtet enthält dieses Sheet damit die Elemente:

- das Sheet $s = ('RunEngine', a1, \{c1, c2, c3, c4, c5, c6, c7, c8, c9\}, \{m1, m2, m3, m4\}, \emptyset)$
- den Aktor $a1 = (\{internal\}, m1, (',,'))$
- den Begriff $c1 = ('Engine', normal, normal, dtype_1, \{c5, c6\}, \{c2, c3\}, \emptyset, m1, \{l1\}, \{1\}, \{1\})$
- den Begriff $c2 = ('DieselEngine', normal, normal, dtype_2, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{2\}, \{2\})$
- den Begriff $c3 = ('PetrolEngine', normal, normal, dtype_3, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{3\}, \{3\})$
- den Begriff $c4 = ('InjectionPump', end, normal, dtype_4, \emptyset, \emptyset, \emptyset, m2, \emptyset, \{4\}, \{4\})$

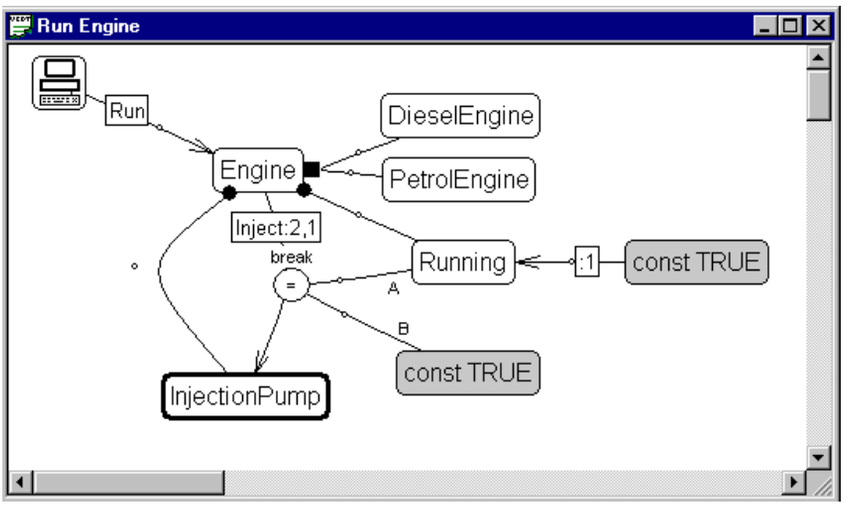


Abbildung 4.1.: Verfeinerung der "Run"-Nachricht des Beispielmodells.

- den Begriff $c5 = ('InjectionPump', end, normal, dtype_4, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{4\}, \{4\})$
- den Begriff $c6 = ('Running', normal, normal, dtype_5, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{5\}, \{5\})$
- den Begriff $c7 = ('Running', normal, normal, dtype_5, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{5\}, \{5\})$
- den Begriff $c8 = ('TRUE', constant, normal, dtype_6, \emptyset, \emptyset, \emptyset, m3, \emptyset, \{6\}, \{6\})$
- den Begriff $c9 = ('TRUE', constant, normal, dtype_6, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{6\}, \{6\})$
- den Datentyp $dtype_4 = (class, 'InjectionPump', 0)$
- die Nachricht $m1 = ('StartEngine', method, 0, a1, c1, \emptyset, \emptyset)$
- die Nachricht $m2 = ('Inject', method, 2, c1, c4, \emptyset, \{cond1\})$
- die Nachricht $m3 = ('\_trigger', method, 1, c1, c7, \{m4\}, \emptyset)$
- die Nachricht $m4 = ('\_get', method, 1, c7, c9, \emptyset, \emptyset)$
- die Schleife $l1 = ('1', 1, \{m2\})$
- die Bedingung $cond1 = (c7, c8, '=', 'break')$

Zu beachten ist, daß jeder Begriff im Sheet *pro Einsatz* ein eigenes Begriffstupel erhält. Deutlich erkennbar ist dies beim Begriff 'InjectionPump', der einmal als Attribut (Tupel $c4$) und einmal als Empfänger der Nachricht 'Inject' verwendet wird (Tupel $c5$). Beide Tupel haben aber dieselben Identifier (hier: 4) und denselben Typ $dtype_4$. Gleiches gilt für die Begriffe 'Running' und 'TRUE'.

'InjectionPump' ist ein Endbegriff und hat daher einen Typ, der bereits hier konkret angegeben werden kann ($dtype_4$).

4.3. Vereinfachungen

Im formalen Modell wurden gegenüber der Modelleingabe im Tool VCDT einige Vereinfachungen vorgenommen. Sie werden in diesem Kapitel motiviert und begründet.

Value

Der Wert (value) eines Begriffes ist im objekt-orientierten Sinne ein spezielles, implizites Attribut (genannt 'Value'), das zu jedem Objekt gehört (Abbildung 4.2). Eine Konstante mit dem Namen *pi* hat z.B. näherungsweise den Wert 3.1415.

Ursprünglich stammt die Idee, jedem Objekt einen Value zuzuordnen, aus der Programmiersprache Smalltalk [Mit89]. Die Werte werden daher in Smalltalk konsquent durch Methoden manipuliert. Dieses Vorgehen geht sogar soweit, daß z.B. die Addition zweier Integerwerte über Methodenaufrufe realisiert wird.

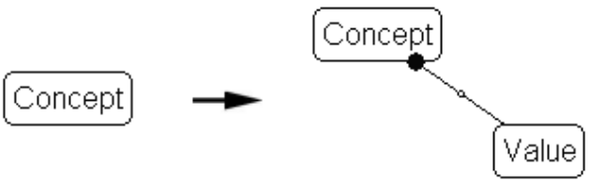


Abbildung 4.2.: Ein Begriff und seine Darstellung mit implizitem Value.

Die Situation bei Begriffen mit Attributen ist in Abbildung 4.3 dargestellt. Jedes Attribut besitzt wiederum einen eigenen Value.

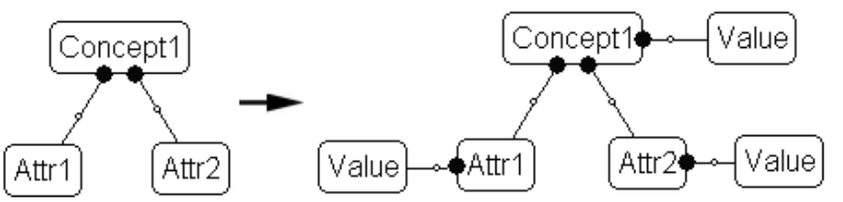


Abbildung 4.3.: Values an Begriffen mit Attributen.

Bei Zuweisungen oder mathematischen Operationen auf Begriffen wird immer das Attribut 'Value' manipuliert. Abbildung 4.4 zeigt anschaulich, wie eine Zuweisung zu begreifen ist. Neben

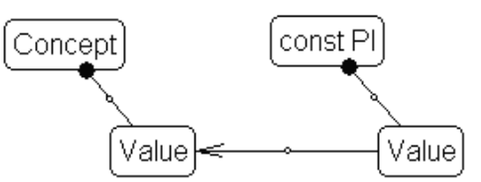


Abbildung 4.4.: Eine Zuweisung zwischen Begriffen ist zu begreifen als eine Zuweisung zwischen ihren 'Values'.

dem Wert besitzen Begriffe ein weiteres implizites Attribut *Default*, das den Defaultwert eines Begriffes repräsentiert. Der Defaultwert kann bei der Definition des Begriffes explizit angegeben werden. Weitere implizite Attribute wie $MinValue$ oder $MaxValue$ sind vorstellbar.

Ein Begriff steht in einem GCN Modell sowohl für sich als auch für seinen Value. Um im Modell auf den Defaultwert zuzugreifen, wird das Attribut $Default$ explizit modelliert.

Values tauchen bei der formalen Definition des GCN Modells nicht auf. Das Ziel der hier vorgestellten Verfahren ist es primär, allen Begriffen in einem Modell einen konkreten Typ zuzuordnen und Komponenten mit ihren Interfaces zu bestimmen. Für diese Betrachtungen ist

der Wert eines Begriffes während seines Lebens im Modell nicht interessant. Bei einer Zuweisung ist es z.B. aus den gerade beschriebenen Blickwinkel nur interessant, ob die beteiligten Begriffe denselben Typ haben oder zumindest kompatibel sind. Der konkrete Wert ist nur zur Laufzeit des Systems wichtig.

Arrays

Begriffe können Arrays bilden. Ein Array ist an sich nichts weiter als eine Gruppierung von typgleichen Begriffen, die einfach ansprechbar sind.

Auch bei Arrays gibt es das spezielle Attribut Value, allerdings nicht nur einmal, sondern für jeden Index genau einen (Abbildung 4.5). Bei Arrays gilt per Definition, daß jeder Wert eines

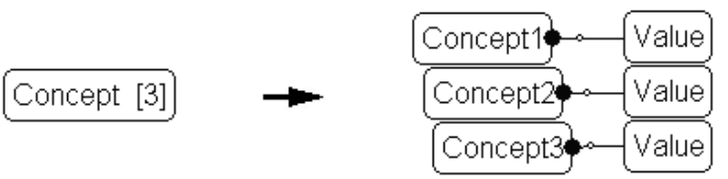


Abbildung 4.5.: Values bei Arrays.

Arrays zwar einzeln ansprechbar ist, aber jeder Wert denselben Typ hat.

Bei Bedingungen, Zuweisungen und mathematischen Operationen werden einzelne Arrayindices und damit einzelne Werte angesprochen. Die konkreten Arrayindices müssen bei der Spezifikation des Modells angegeben werden, sonst wäre das Modell unvollständig. Für die Werte in einem Array gilt im Grunde dasselbe wie für den Wert eines normalen Begriffes. Der konkrete Inhalt ist nur zur Laufzeit interessant. Für die Behandlung des Modells im Sinne der Typfindung kann ein Array dann aber wie ein gewöhnlicher Begriff behandelt werden, denn ein Array hat im Grunde auch nur einen einzigen Typ.

Wenn die Werte eines Array aber nicht interessant sind, ist die Addressierung eines speziellen Wertes, also eines speziellen Arrayindexes, im formalen Modell nicht notwendig.

Für die Codegenerierung ist es natürlich unbedingt notwendig zu wissen, auf welchen Index wo zugegriffen wird. Aus diesem Grund werden im Tool die gesamten Array-relevanten Informationen aus dem graphischen Modell in den funktionalen Graphen übernommen und 'mitgeschleift'. Für die formale Betrachtung im Rest der Arbeit spielen diese Daten aber keine Rolle und wurden daher aus dem formalen Modell herausgelassen.

Alias

Für Begriffe, Parameter und Attribute kann ein sogenannter *Aliasname* vergeben werden. Ein Aliasname dient gewöhnlich der Verbindung des Modells zur realen Programmierumgebung. Bereits existierende Klassen und Methoden verwenden festgelegte Bezeichnungen, die im Modell nicht sinnvoll oder nicht aussagekräftig genug sind. Mit Aliasnamen können die realen Bezeichnungen referenziert werden, ohne das Modell damit zu belasten.

Aliasnamen werden allerdings nicht im formalen Modell berücksichtigt, sondern vorher in eine Identität umgewandelt. Bei der Umwandlung wird ein neuer Begriff angelegt, der den Namen des Aliasnamen bekommt und einen eigenen Identifier. Alle anderen Eigenschaften werden vom Begriff geerbt, der den Aliasnamen besitzt.

In/Out-Parameter

Parameter können in der graphischen Modellierungssprache als In/Out-Parameter modelliert werden; sie sind dann sowohl Input- als auch Output-Parameter. Bei der Abbildung von der graphischen Darstellung in die formale Darstellung werden In/Out-Parameter umgewandelt in zwei Parameter: einen Input- und einen Output-Parameter, beide mit demselben Namen und demselben Identifier (Abbildung 4.6).

Abbildung 4.6.: Ein InOut-Parameter wird aufgefaßt als ein Input- und ein Output-Parameter.

Bei der graphischen Eingabe kann ein In/Out-Parameter in einem Sheet als ein Parameter und in einem anderen Sheet durch zwei Parameter dargestellt werden. Beide Darstellungsvarianten sind gleichwertig.

Die konkrete Realisierung eines Parameters (z.B. als 'call by reference') bleibt dem Codegenerator überlassen.

Zusammenfassung

Die beschriebenen Vereinfachungen - dazu gehören auch die in Kapitel 3.7 beschriebenen mathematischen Operationen - ermöglichen es, das formale Modell für GCN Modelle kompakt zu halten. Besondere Beschreibungsformen werden auf die Basiselemente von GCN Modellen zurückgeführt, also Nachrichten und Begriffe. Sie integrieren sich so nahtlos in die formale Beschreibung.

4.4. Allgemeine Operationen/weitere Begriffe

Dieses Kapitel definiert einige weitere Begriffe und Operationen auf den bisherigen Definitionen.

Für ein gegebenes Modell G ist:

- IA_G die Menge aller internen Aktoren, $ia_{G,i}$ das i-te Element der Menge.
- EA_G die Menge aller externen Aktoren, $ea_{G,i}$ das i-te Element der Menge.
- C_G die Menge aller Begriffe, $c_{G,i}$ das i-te Element der Menge.
- M_G die Menge aller Methoden, $m_{G,i}$ das i-te Element der Menge.

Falls klar ist, welches Modell gemeint ist, wird der Index G einfach weggelassen.

Man benötigt im weiteren Verlauf häufig einen Test, ob zwei Begriffe gleich (genauer 'namensgleich') sind. Zunächst die Definition:

Definition 11 (namensgleich) *Zwei Begriffe*

$$c_1 = (name_1, \ldots, ALT_1, IDENT_1, \ldots)$$

und

$$c_2 = (name_2, \ldots, ALT_2, IDENT_2, \ldots)$$

sind namensgleich (Symbol '$\equiv$'), wenn eine der folgenden Bedingungen erfüllt ist:

1. $name_1 = name_2$
2. $\exists id_i = (name_i, ...) \in IDENT_1 : name_i = name_2$
3. $\exists id_i = (name_i, ...) \in IDENT_2 : name_i = name_1$
4. $\exists id_i = (name_i, ...) \in IDENT_1, id_j = (name_j, ...) \in IDENT_2 : name_i = name_j$

Zwei Begriffe gelten als namensgleich, wenn ihre Namen gleich sind (1). Namensgleich sind zwei Begriffe auch, wenn der Name des ersten Begriff gleich dem Namen eines Begriffes ist, der mit dem zweiten Begriff identisch gesetzt wurde (2,3) oder wenn es einen Begriff gibt, der in den Ident-Mengen beider Begriffe vorkommt (4).

Zwei namensgleiche Begriffe können ihre Eigenschaften übertragen, auch 'erben' genannt. Die Abbildung dazu lautet:

Definition 12 (MergeProperties) *Die Abbildung*

$$MergeProperties : c \times c_m \rightarrow c'$$

überträgt die Eigenschaften eines Begriffes $c_m = (name_m, ctype_m, etype_m, dtype_m, ATR_m, ALT_m, IDENT_m, \ldots, ID_m, TID_m)$ *auf einen Begriff* $c = (name, ctype, etype, dtype, ATR, ALT, IDENT, \ldots, ID, TID)$. *Das Ergebnis ist der geänderte Begriff* $c' = (name, ctype', etype', dtype', ATR', ALT', IDENT', \ldots, ID', TID')$ *mit:*

- $ctype' = \begin{cases} ctype, & falls \quad ctype_m \neq splitter \wedge ctype_m \neq ifsplitter \\ ctype_m, & falls \quad ctype_m = splitter \vee ctype_m = ifsplitter \end{cases}$
- $etype' = \begin{cases} etype, & falls \quad etype_m = normal \\ etype_m, & sonst \end{cases}$
- $dtype' = \begin{cases} dtype, & falls \quad dtype_m = (btype = notype, ...) \\ dtype_m, & sonst \end{cases}$
- $ATR' = ATR \cup ATR_m$
- $IDENT' = IDENT \cup IDENT_m$
- $ID' = ID \cup ID_m$
- $TID' = TID \cup TID_m$

Es wird an dieser Stelle absichtlich keine Aussage über Alternativen gemacht. Sie müssen je nach Fall gesondert behandelt werden.

Eine ähnliche Abbildung kann für Nachrichten definiert werden:

Definition 13 (MergeMessageProperties) *Die Abbildung*

$$MergeMessageProperties : m \times m_m \rightarrow m'$$

überträgt die Eigenschaften einer Nachricht $m_m = (name_m, type_m, ..., COND_m)$ *auf eine Nachricht* $m = (name, type, \ldots, COND)$. *Dabei muß gelten:* $name = name_m \vee type = type_m$. *Das Ergebnis ist die geänderte Nachricht:*

$$m' = (name, type, \ldots, COND \cup COND_m).$$

Parameter werden erst im Umfeld der Anwendung der Abbildung behandelt.

In Kapitel 3.3 wurde erläutert, daß Identitäten in Alternativen umgewandelt werden können. Es wird deshalb hier die Abbildung *IdentToAlt* definiert, die eine Umwandlung von Identitäten in Alternativen durchführt.

Definition 14 (IdentToAlt) *Ein Begriff* $c = (\ldots, ALT, IDENT = \{ident_1\}, \ldots)$ *mit genau einer Identität* $ident_1$ *wird durch die Abbildung*

$$IdentToAlt : c \rightarrow c'$$

umgewandelt in den Begriff $c' = (\ldots, ALT', IDENT', \ldots)$ *mit:*

- $ALT' = ALT' \cup \{ident_1\}$
- $IDENT' = \emptyset$

Die Umwandlung eines Begriffs mit Alternativen in einen Splitter wird mehrfach benötigt. Sie soll daher hier als Abbildung *ConceptToSplitter* definiert werden:

Definition 15 (ConceptToSplitter) *Ein Begriff* $c = (\ldots, ctype, ATR, ALT, IDENT, m_r, impl, \ldots)$ *mit Alternativen,* $|ALT| > 0$, *der eine Nachricht* m_r *empfängt, wird durch die Abbildung*

$$ConceptToSplitter : c \rightarrow c'$$

umgewandelt in den Begriff $c' = (\ldots, ctype', ATR, ALT', IDENT, m_r, impl' \ldots)$ *mit:*

- $ctype' = \begin{cases} splitter, & falls \quad ctype = normal \\ ifsplitter, & falls \quad ctype = interface \end{cases}$
- $\forall alt_i \in ALT : impl' = impl' \cup \{m_i\}$ *mit* $m_i = (mname_r, mtype_r, pos_r, sender = ctype', rcv = alt_i, P_r, COND_r)$
- $\forall alt_i \in ALT : alt_i = (..., ATR, ..., m = m_i, impl, ...)$
- $ALT' = \emptyset$

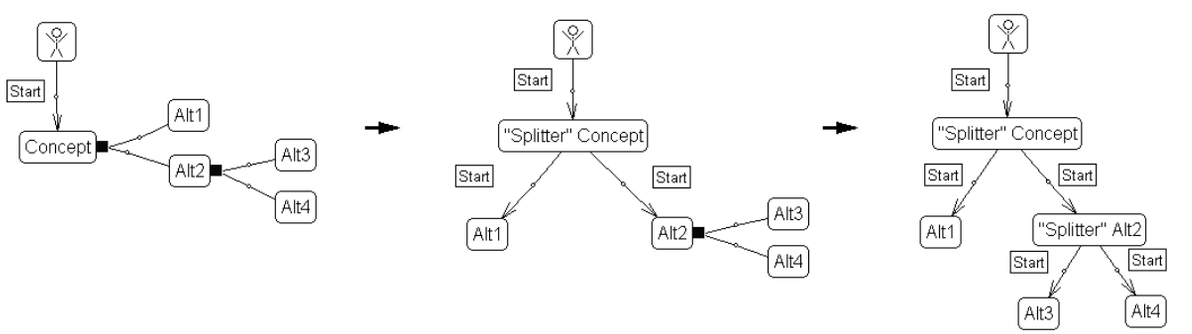


Abbildung 4.7.: Zweistufige Alternativen und der schrittweise Umbau mit Splittern.

Es ist möglich, daß die Alternativen selbst wieder Alternativen besitzen. Die Abbildung *ConceptToSplitter* wird solange angewendet, bis die Empfänger von Nachrichten keine Alternativen mehr besitzen. Abbildung 4.7 zeigt den schrittweisen Umbau mit zwei Splittern.

5. Compilevorgang des GCN-Modells

Das folgende Kapitel beschreibt den Compilevorgang für ein GCN Modell. Der erste Abschnitt beschäftigt sich mit der Erstellung des Strukturgraphen, der den funktionalen Ablauf eines GCN Modells repräsentiert. Aus dem Strukturgraphen wird dann eine Begriffsliste gewonnen (Abschnitt 5.2) und die darin enthaltenen Begriffe werden im darauffolgenden Schritt klassifiziert (Abschnitt 5.3). Abschnitt 5.4 beschreibt aufbauend auf der Klassifikation die Bestimmung der Komponenten und Interfaces. Den Abschluß des Compilevorgangs bildet eine Codegenerierung, die das abstrakte Design automatisch in ablauffähigen Code umsetzt. Der letzte Schritt ist aber nicht Gegenstand dieser Arbeit und wird in Kapitel 5.5 nur angerissen.

Das Beispiel des Motors aus Abschnitt 3.2 wird dabei in den einzelnen Schritten zur Veranschaulichung verwendet; es wird im folgenden kurz *das Beispielmodell* genannt.

5.1. Die funktionale Struktur des Modells

Der funktionale Graph (Strukturgraph) stellt die Summe aller potentiellen Pfade im Programmablauf dar, die durch Stimulation von außen (d.h. durch Stimulation der externen Aktoren) auftreten können. Dabei wird nicht versucht, die tatsächlichen Pfade zu berechnen; Schleifen und Bedingungen werden nicht ausgewertet. Ob und wie oft welche Pfade durchlaufen werden, ist nicht Gegenstand der Betrachtung. Um ein Gefühl für einen funktionalen Graphen zu bekommen, sei auf Abbildung 5.1 verwiesen. Sie zeigt den vollständigen Graphen des Beispielmodells als *TreeView*[1].

Die Knoten des Graphen werden durch schwarze Symbole auf weißem Grund dargestellt. Zusatzinformation zu den Graphknoten befinden sich direkt unterhalb des jeweiligen Knotens, dargestellt durch weiße Symbole auf schwarzem Grund. Zusatzinformationen bei Begriffen sind z.B. die Attribute oder Alternativen, bei Nachrichten die Parameter und Bedingungen.

Die Wurzel eines Baumes bildet immer ein Aktor, hier der externe Aktor. Von ihm geht die gesendete Nachricht aus, gefolgt vom Empfänger der Nachricht. Im Beispielmodell ist die verkürzte Variante bei einem Splitter zu sehen (lt. Regel 2). Die Nachricht “Start“ wird vom Splitter an seine beiden Alternativen gesendet (’PetrolEngine’ und ’DieselEngine’). Auf der nächsten Ebene unterhalb des Empfängers befindet sich die Implementierung der Nachricht, also alle Nachrichten, die von der Nachricht “Start“ ausgelöst werden, usw. Gut zu erkennen ist, daß die Implementierung von “Start“ für den Benzinmotor (’PetrolEngine’) die beiden Nachrichten “StartEngine“ und “RunEngine“ umfaßt, die Implementierung für den Dieselmotor aber die vier Nachrichten “GetSeconds“, “Heat“, “StartEngine“ und “RunEngine“.

Ein funktionaler Graph (Strukturgraph) ist definiert als *gerichteter, antizyklischer Graph*, ein *Baum*:

$$\text{fg} := (\text{V}, \text{E}, \text{root}).$$

[1] Erstellt mit dem prototypischen Tool VCDT. Für eine Vorstellung von VCDT siehe Anhang A.

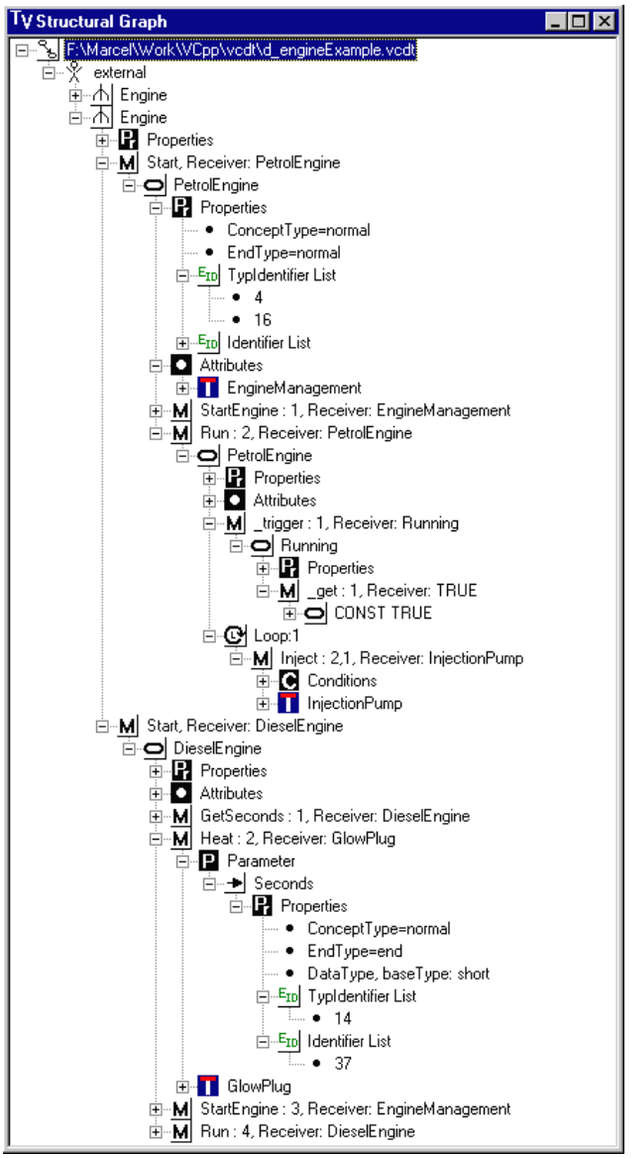


Abbildung 5.1.: Der funktionale Gesamtgraph des Beispielmodells.

Dabei gilt:

- V die endliche Menge der *Knoten (vertix)*, $V = C \cup M \cup A \cup L$.
- E die endliche Menge der *geordneten Kanten (edge)*, $e = (a, b)$ wobei a der *Anfangspunkt* und b der *Endpunkt* genannt wird.
- *root* ist der *Wurzelknoten* des Baumes.

Die Knoten des Baumes bestehen aus Begriffen C, Nachrichten M, Aktoren A und Schleifen L.

Die folgenden Unterkapitel beschreiben die Erzeugung des funktionalen Gesamtgraphen für ein gegebenes Modell $\mathcal{G}$.

5.1.1. Aufbau der Teilgraphen

Der erste Schritt zur Erzeugung des funktionalen Gesamtgraphen ist die Abbildung der einzelnen Sheets s_i in einen funktionalen Teilgraphen fg_i.

Definition 16 (funktionaler Graph eines Sheets) *Der funktionale Graph eines Sheets s_i ist definiert als:*

$$fg(s_i) := fg_i = (V_i, E_i, root_i)$$

Er wird auch als Teilgraph *des Sheets s_i bezeichnet.*

Die Wurzel eines Teilgraphen bildet der Aktor des Sheets. Die Abbildung eines Sheets in den funktionalen Teilgraphen wird mit *makegraph* bezeichnet und ist formal definiert als:

$$makegraph : s_i = (name_i, a_i, C_i, M_i, L_i) \rightarrow fg_i = (V_i, E_i, a_i)$$

mit:

- $V_i = a_i \cup C_i \cup M_i \cup L_i$
- $E_i = \{e_l\}$

Der Aktor, die Begriffe, Nachrichten und Schleifen des Sheets bilden die Knoten des Graphen. Die Kanten des Graphen ergeben sich folgendermaßen:

- für $a_i = (type_i, m_i, r_i)$ eine Kante $e = (a_i, m_i)$ vom Aktor zur Nachricht m_i
- $\forall\ m_j = (..., rcv_j, ...) \in M_i$ gibt es eine Kante $e_j = (m_j, rcv_j)$ zum Empfänger
- $\forall\ c_j = (..., impl_j \neq \emptyset, ...) \in C_i$ gibt es Kanten $e_k = (c_j, z_k)$, $\forall z_k \in impl_j$
- $\forall\ l_l = (lname_l, pos_l, impl_l) \in impl_j$: Kanten $e_k = (loop_l, m_k)$ $\forall m_k \in impl_l$

Die Bezeichnung des Sheets $name_i$ ist während der Eingabe des Modells eine Hilfe für den Benutzer, für den funktionalen Graphen wird er nicht benötigt.

Die weitere Betrachtung eines funktionalen Graphen wird vereinfacht, wenn zunächst einige Begriffe definiert werden:

Definition 17 (Pfad (path) eines Graphen) *Ein Folge von Knoten $path = v_0..v_n$, $(v_0 \neq v_n)$ eines funktionalen Graphen heißt* Pfad (path)*, wenn es Kanten $e_i = (v_{i-1}, v_i)$ für $i = 1, ..., n$ gibt. Da ein funktionaler Graph per Definition keine Zyklen enthält, gilt auch, daß die v_j paarweise verschieden sind.*

Ein Pfad eines funktionalen Graphen hat die immer die Form $amc((l^*)mc)^*$[2]. Der Pfad fängt mit einem Aktor an, gefolgt von einer Nachricht an einen Begriff. Anschließend können beliebig viele Nachrichten an Begriffe folgen, die auch innerhalb von (mehrfach geschachtelten) Schleifen auftreten können.

Definition 18 (Blatt (leaf)) *Ein* Blatt *eines funktionalen Graphen ist ein Knoten vom Typ Begriff $(v_n = c_i = (name_i, ..., impl_i, ...))$, der keine Implementierung $(impl_i = \emptyset)$ und damit keine Nachfolger hat (Grad 1).*

In einem funktionalen Graphen können Blätter nur Begriffsknoten sein, denn eine Nachricht hat immer einen Empfänger, und Schleifen umfassen mindestens eine Nachricht.

[2] wenn Splitter vorkommen, ist dies nicht ganz korrekt, weil dort die eigentlich redundante, ursprüngliche Nachricht entfernt wird, siehe Kapitel 5.1.4. Zum einfacheren Verständnis kann aber zunächst von dieser Darstellung ausgegangen werden.

Definition 19 (abgeschlossener Pfad) *Ein Pfad* $path = v_0..v_{n-1}v_n$ *heißt* abgeschlossen, *wenn ein Blatt* v_n *ein Endbegriff ist oder alle Parameter der Nachricht* v_{n-1} *versorgt sind. Versorgt bedeutet, daß die Parameter entweder dem Empfängerbegriff bekannt sind (z.B. als Attribut) oder durch mathematische Operationen bestimmt werden.*

Abbildung 5.2 zeigt mögliche Formen von abgeschlossenen Pfaden.

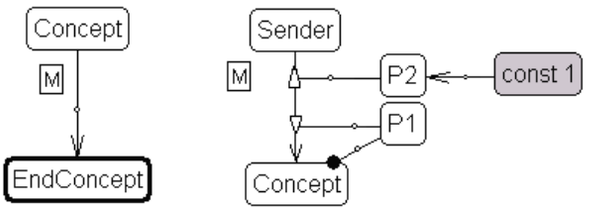


Abbildung 5.2.: Diese Pfade sind abgeschlossen.

Definition 20 (vollständiger Teilgraph) *Ein* vollständiger (complete) Teilgraph fg_c *ist ein Teilgraph, dessen Pfade nach Definition 19 alle abgeschlossen sind.*

Um später (beim Bau des Gesamtgraphen) die Fallunterscheidungen zu minimieren, werden die Teilgraphen bereits jetzt so vorverarbeitet, um sie später alle gleich behandeln zu können.

Die Umsetzung von Regel 3 geschieht statisch nach dem Aufbau eines Teilgraphen. Die Regel besagt, daß eine Verfeinerung für eine Nachricht an einen Begriff mit Alternativen sowohl für den Begriff selbst als auch für alle Alternativen gilt.

Die folgende Betrachtung gilt für alle Teilgraphen $fg_i = (V_i, E_i, a_i = (internal, ...))$. O.B.d.A. sei fg genau ein solcher Teilgraph mit dem Pfad $path = a_i m c \ldots$ mit $c = (..., ALT, ...)$, $j = |ALT|, j > 0$.

Der Teilgraph wird zunächst j mal dupliziert. Aus einem Teilgraphen fg werden dadurch $j + 1$ Teilgraphen; einer für jede Alternative und einer für den Oberbegriff.

Für den Teilgraphen des Oberbegriffs, genannt fg_0, gilt: der Empfänger c wird durch die Abbildung $ConceptToSplitter$ umgeformt.

Für die Teilgraphen (Kopien!) fg_1 bis fg_j gilt:

- sei $alt = (name_a, \ldots, ATR_a, ALT_a, \ldots, IDENT_a, \ldots, ID_a, TID_a) \in ALT$
- c wird umgeformt zu $c = (name_a, ..., ATR \cup ATR_a, ALT_a, IDENT_a, ..., ID_a, TID_a)$

Beachte: Attribute an Begriffen mit Alternativen gelten für alle Alternativen.

Bemerkung: Für Splitter wird der funktionale Graph etwas gestrafft. Die Nachricht, die vom Splitter empfangen wird, ist redundant, da die Nachricht unverändert vom Splitter an die Alternativen gesendet wird. Sie wird daher im funktionalen Graphen entfernt. Es gibt dann nur eine Kante direkt vom ursprünglichen Sender zum Splitter. Diese Straffung gilt immer dann, wenn ein Splitter eingeführt wird.

Es gibt zwei unterschiedliche Arten von funktionalen Teilgraphen, sie unterscheiden sich durch den Aktortyp der Wurzel. Man kann deshalb definieren:

Definition 21 (Menge der Teilgraphen mit externen Aktoren) *Die Menge aller Teilgraphen mit einem externen Aktor als root wird mit* FG_{ea} *bezeichet:*

$$FG_{ea} = \{fg_i = (V_i, E_i, root_i = a_i) \mid a_i = (type_i, ...), type_i = external\}$$

Falls keine Verwechslungsgefahr besteht, wird diese Menge kurz mit Menge der externen Aktoren *bezeichnet. Ein Element fg_{ea} der Menge bezeichnen wir einfach als* externer Aktor.

Genauso kann man eine Menge für Teilgraphen mit interne Aktoren definieren:

Definition 22 (Menge der Teilgraphen mit internen Aktoren) *Die Menge aller Teilgraphen mit einem internal Aktor als root wird mit FG_{ia} bezeichet:*

$$FG_{ia} = \{fg_i = (V_i, E_i, root_i = a_i) \mid a_i = (type_i, ...), type_i = internal\}$$

Falls keine Verwechslungsgefahr besteht, wird diese Menge kurz mit Menge der internen Aktoren *bezeichnet. Ein Element fg_{ia} der Menge bezeichnen wir auch als* internen Aktor.

Das Ziel dieses Schrittes war der Aufbau aller Teilgraphen eines GCN Modell, also die Bildung der Mengen FG_{ea} und FG_{ia}.

Für das Sheet 'Run Engine' des bekannten Beispiels ergibt die Abbildung *makegraph* den funktionalen Graphen in Abbildung 5.3. Die funktionalen Graphen des Modells sind bezeichnet mit $FG_{ea} = \{fg_{startEngine}\}$ und $FG_{ia} = \{fg_{startDieselEngine}, fg_{startPetrolEngine}, fg_{runEngine}, fg_{getSeconds}\}$.

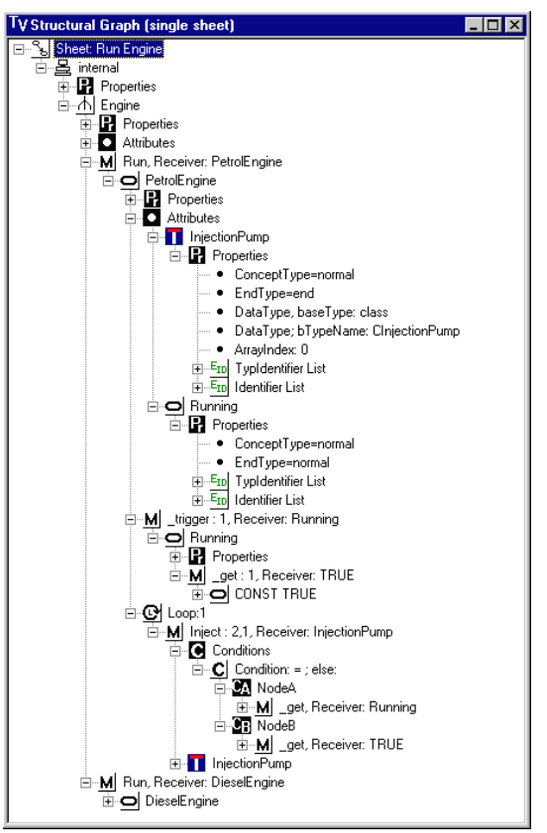


Abbildung 5.3.: Der funktionale Teilgraph für das Sheet 'RunEngine' des Beispielmodells (incl. Umsetzung von Regel 3)

5.1.2. Bau des Gesamtgraphen

Nach dem Aufbau aller Teilgraphen wird daraus der gesamte funktionale Graph erzeugt. Die Definition für den Gesamtgraphen eines GCN Modells lautet:

Definition 23 (funktionaler Gesamtgraph eines GCN Modells) *Der funktionale Gesamtgraph eines GCN Modells G besteht aus der Summe aller vollständigen Teilgraphen $f_{ea,c}$, also aus der Menge FG_{ea} nach der Verfeinerung.*

$$fg(\mathcal{G}) = \{fg_{ea,c} | fg_{ea} \in FG_{ea}\}$$

Ein GCN Modell hat üblicherweise mehr als einen externen Aktor. Der funktionale Gesamtgraph eines Modells ist daher ein *Wald* von vollständigen, funktionalen Teilgraphen.

Da der Gesamtgraph aus der Summe der externen Aktoren besteht, wird o.B.d.A. für die folgende Beschreibung nur noch ein externer Aktor betrachtet: $fg_{ea} \in FG_{ea}$. Für alle weiteren externen Aktoren wird analog verfahren.

Ein unvollständiger funktionalen Teilgraph fg_{ea} soll zu einem vollständigen Teilgraphen werden. Dazu werden alle nicht abgeschlossenen Pfade des Teilgraphen behandelt und vervollständigt, wodurch fg_{ea} wächst.

Regel 13 *Der funktionaler Gesamtgraph eines GCN Modells besitzt nur abgeschlossene Pfade.*

O.B.d.A. wird im folgenden genau ein nicht abgeschlossener Pfad des funktionalen Teilgraphen betrachtet und eine Implementierung dafür gesucht.

Implementierungen von Nachrichten werden in Sheets mit internen Aktoren definiert, also in der Menge FG_{ia}. Die Aufgabe lautet nun, denjenigen funktionalen Teilgraphen zu finden, der die Nachricht am besten implementiert. Die Implementierung einer Nachricht wird auch als *Verfeinerung (refinement)* der Nachricht bezeichnet.

Der Ablauf des gesamten Vorgangs ist in Abbildung 5.4 dargestellt. Die einzelnen Schritte werden in den folgenden Abschnitten ausgeführt.

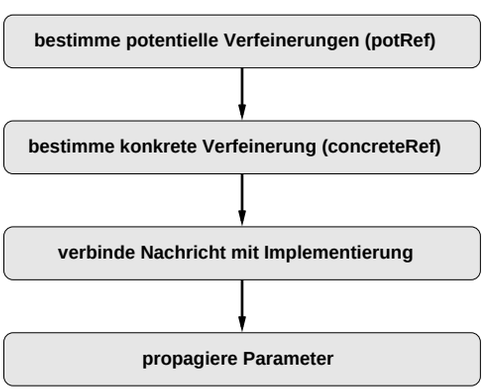


Abbildung 5.4.: Schritte bei der Verfeinerung eines nicht abgeschlossenen Pfades eines funktionalen Graphen.

Bestimmung der potentiellen Verfeinerungen

Dieser Abschnitt beschreibt die Lösung der folgenden Aufgabe: finde für einen unvollständigen Pfad $path = ...mc$ die internen Aktoren, die die Nachricht $m = (mname, \ldots, P, \ldots)$ an den Begriff c verfeinern.

Die Menge dieser potentiell passenden Verfeinerungen wird mit $POTREF$[3] bezeichnet. Sie enthält alle funktionalen Teilgraphen, die die Nachricht mit Namen *mname* und

[3] für 'potential refinement'

Empfängerbegriff c implementieren. Sei $fg_i = (V_i, E_i, a_i) \in FG_{ia}$ ein Teilgraph, $e_1 = (a_i, m_i), e_2 = (m_i, c_i) \in E_i$ zwei Kanten des Graphen und $m_i = (mname_i, ..., rcv_i = c_i, P_i, ...)$ die Nachricht an den Begriff c_i, die vom internen Aktor verfeinert wird, dann gilt:

$$POTREF = \{fg_i \in FG_{ia} \mid mname_i = mname, c_i \equiv c\}$$

Die Menge $POTREF$ enthält jetzt alle internen Aktoren, die weiter untersucht werden müssen. Die Auswahlbedingung für die Menge $POTREF$ ist leicht verständlich: die Nachrichtennamen müssen identisch sein und der Empfänger der Nachricht muß namensgleich zum Begriff sein, der die Nachricht implementiert.

Regel 14 *Entsprechend Regel 13 gilt, daß es für jeden nicht abgeschlossenen Pfad in einem funktionalen Graphen eine Verfeinerung geben muß.*

Ist $|POTREF| = 0$, so ist dies ein Fehler im Modell, denn dann kann ein Pfad nicht abgeschlossen werden. Ein Modell ist aber nur dann vollständig, wenn es keine nicht abgeschlossenen Pfade gibt.

Bestimmung der konkreten Verfeinerung

Die Bestimmung der konkreten Verfeinerung aus der Menge der möglichen Verfeinerungen geschieht über die Parameter der ursprünglich gesendeten Nachricht.

Für alle internen Aktoren $a_i = (internal, m_i, ...)$ der Menge $POTREF$ werden die Parameter P_i der Nachricht m_i untersucht. Sie müssen zu den Parametern P der zu verfeinernden Nachricht m passen, d.h. für jeden Parameter $p \in P$ muß ein passendes Element $p_{k,i} \in P_i$ gefunden werden. P_i kann allerdings mehr Elemente besitzen als P, wenn folgende Regel eingehalten wird:

Regel 15 *Bei einer Parameterübergabe an Nachrichten müssen nicht übergebene Parameter ein Default-Attribut besitzen.*

Diese Regel erhöht die Übersichtlichkeit von GCN-Modellen und entlastet den Designer. Sie ist in ähnlicher Form in Programmiersprachen wie z.B. C++ realisiert.

Die Menge der konkret passenden Teilgraphen $CONREF$[4] für einen Pfad $path = ...mc$ mit $m = (..., P, ...)$ ergibt sich somit zu:

$$CONREF = \{fg_i \mid fg_i \in POTREF, P \text{ match } P_i\}$$

mit $fg_i = (V_i, E_i, a_i = (internal, m_i = (..., P_i, ...), role_i))$

Die Abbildung $match$ bestimmt, ob zwei Parametermengen zueinander passen. Bevor $match$ formal definiert wird, betrachten wir zwei Mengen von Parametern $P = \{p_1, p_2, \ldots, p_n\}$ und $P_r = \{p_{r1}, p_{r2}, \ldots, p_{rm}\}$[5]. Die beiden Mengen dürfen wegen Regel 15 unterschiedlich groß sein $(n <= m)$.

Der Vergleich zweier Parametermengen kann reduziert werden auf die einzelnen Elemente der beiden Mengen. Für jedes Paar $(p_i; p_{rj})$ wird geprüft, ob die Parameter zueinander passen.

Seien $p = (c, d)$ und $p_r = (c_r, d_r)$ zwei Parameter mit den Begriffen $c = (name, \ldots, ALT, \ldots)$ und $c_r = (name_r, \ldots, ALT_r, \ldots)$. Sie passen (match), falls folgende Bedingungen erfüllt sind:

[4] für 'concrete refinement'
[5] P_r für $P_{refinement}$

1. $d = d_r$ (gleiche Richtung)
2. $c \equiv c_r \vee c \in ALT_r \vee c_r \in ALT(namensgleich)$
3. für $ALT \neq \emptyset \wedge ALT_r \neq \emptyset$ gilt: $\begin{cases} ALT \cap ALT_r \neq \emptyset & \text{falls} \quad d = output \\ ALT \subseteq ALT_r \neq \emptyset & \text{falls} \quad d = input \end{cases}$

Die Menge $matched = \{(p_i, p_{rj})\}$ bekommt einen Eintrag, falls die Bedingungen erfüllt sind.

Bedingung 1 ist selbsterklärend. Bedingung zwei besagt, daß die Begriffe entweder namensgleich sind oder gegenseitig als Alternative vorkommen, was wegen Regel 4 einer Identität entspricht.

Alternativen an Begriffen müssen für Input- und Output-Parameter unterschiedlich behandelt werden. Informell betrachtet, darf durch einen Input-Parameter nicht mehr von einer Verfeinerung verlangen werden, als die Verfeinerung selbst anbietet.

Für Output-Parameter hingegen können nicht mehr Informationen (von der Verfeinerung) zurückgegeben werden als beim Senden der Nachricht bekannt sind. Die Bildung der Schnittmenge ist im Moment des Mergevorgangs die einzige Möglichkeit, Output-Parameter zu behandeln, da die *konkrete Menge* der Alternativen für Output-Parameter erst später beim rückwärts propagieren feststeht. Sollte diese Annahme falsch sein, so werden im weiteren Verlauf der Verfeinerung Fehler generiert.

Nach diesem Einschub zurück zur Funktion *match* für zwei Parametermengen, die formal als Abbildung dargestellt werden kann:

$$match : P \times P_r \rightarrow BOOL$$

Zwei Parametermengen P und P_r passen zueinander, wenn folgende Bedingungen erfüllt sind:

1. $\forall i : i <= n : \exists mp_i \in matched$ mit $mp_i = (p_i, p_j)$
2. $p(i) \neq p(k)\ \forall k \neq i$ mit $p(i) = p_j$ für $mp_i = (p_i, p_j)$
3. $\forall j : n < j <= m : p_j = (d_j, c_j = (..., ATR, ...)) : \exists$ 'Default' $\in ATR$

Die erste Bedingung besagt, daß es für jeden Parameter der zu verfeinernden Nachricht *genau einen* Parameter (Bedingung 2) in der Verfeinerung gibt. Für alle übrig gebliebenen Parameter muß es einen Defaultwert geben (Bedingung 3).

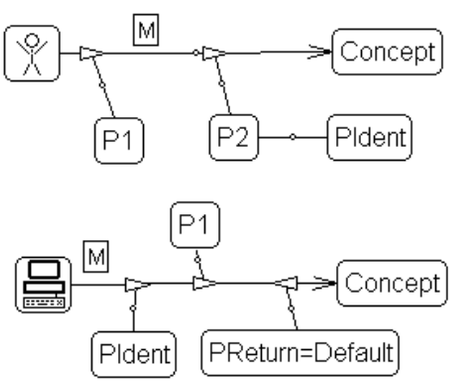


Abbildung 5.5.: Eine konkrete Verfeinerung für eine Nachricht an den Begriff *Concept*.

Abbildung 5.5 zeigt eine Nachricht M mit zwei Parametern an den Empfänger *Concept* und eine passende Verfeinerung dazu. Es paßt dort der Parameter $P1$ direkt und der Parameter $P2$ über

die Identität mit dem Begriff $PIdent$. Parameter $PResult$ der Verfeinerung ist ein zusätzlicher, mit einem Defaultwert versehener Parameter.

An dieser Stelle hat $CONREF$ im einfachsten Fall genau ein Element, im folgenden kurz fg_{ia} genannt. Für $|CONREF| > 1$ ist eine Sonderbehandlung nötig. Kapitel 5.1.3 widmet sich diesem und weiteren Ausnahmen. Die komplexeren Fälle werden dort auf den hier gezeigten, einfachen Fall reduziert.

Ist $|CONREF| = 0$, so ist dies ein Fehler im Modell (siehe Regel 14).

Der Mergevorgang

Das Ziel des Mergevorgangs[6] ist es, den nicht abgeschlossenen Pfad $path = ...mc$ des Teilgraphen fg_{ea} und damit die Nachricht m an den Begriff $c = (name, ...)$ zu verfeinern (zu implementieren). Die passende Verfeinerung in Form des internen Aktors $fg_{ia} \in CONREF$ wurde im letzten Abschnitt bestimmt. Sie implementiert die Nachricht m_i an den Begriff $c_i = (name_i, ctype_i, etype_i, dtype_i, ATR_i, ALT_i, IDENT_i, m_i, impl_i, ID_i, TID_i)$.

Die gesuchte Implementierung der Nachricht m findet sich in der Implementierung $impl_i$ des Begriffes c_i. Der nicht abgeschlossene Pfad wird durch den Zusammenbau mit der Implementierung zu einem fortgesetzten Pfad; es entstehen dadurch u.U. neue, nicht abgeschlossene Pfade.

Die Abbildung $mergegraphs$ beschreibt den Vorgang formal:

$$mergegraphs : fg_{ea} \times fg_i \in CONREF \rightarrow fg'_{ea}$$

Die benötigten Bezeichnungen sind hier noch einmal zusammengefaßt. Die zu verfeinernde Nachricht und der dazu gehörende funktionale Graph besteht aus:

- der externe Aktor $fg = (V, E, root) \in FG_{ea}$
- der nicht abgeschlossene Pfad $path = ...mc$ mit:
- der Nachricht $m = (mname, mtype, pos, sender, rcv = c, P, COND)$
- ihren Parametern $P = \{p_i = (d_i, c_i)\}$
- dem ursprünglichen Empfänger $c = (name, ctype, etype, dtype, ATR, ALT, IDENT, m, impl = \emptyset, ID, TID)$

Der interne Aktor zur Verfeinerung:

- $fg_{ia} = (V_{ia}, E_{ia}, a_{ia})$ mit:
- dem Aktor $a_{ia} = (internal, m_{ia}, role_{ia})$
- der Nachricht $m_{ia} = (mname_{ia}, mtype_{ia}, pos_{ia}, a_{ia}, rcv_{ia}, P_{ia}, COND_{ia})$
- ihren Parametern $P_{ia} = \{p_{ia,j} = (d_{ia,j}, c_{ia,j})\}$
- dem Empfänger mit der Implementierung $rcv_{ia} = (name_{ia}, ctype_{ia}, etype_{ia}, dtype_{ia}, ATR_{ia}, ALT_{ia}, IDENT_{ia}, m_{ia}, impl_{ia}, ID_{ia}, TID_{ia})$
- funktionalem Teilbaum mit Wurzel rcv_{ia}: $fg_{rcv} = (V_{rcv} \subset V_{ia}, E_{rcv} \subset E_{ia}, rcv_{ia})$

[6] der Name 'Merge' kommt vom Verschmelzen funktionaler Teilgraphen, fg_{ea} und fg_{ia}

Um das bereits beschriebene Instanzproblem zu addressieren, werden die Identifier ID (nicht die Typidentifier !) für alle Begriffe unterhalb des internen Aktors (auch Parameter und Conditions) neu vergeben. Dadurch wird erreicht, daß mehrfaches Einhängen derselben Verfeinerung an unterschiedlichen Stellen im Graphen nicht zur Zusammenführung von Begriffen führt, die eigentlich unterschiedliche Instanzen darstellen. Die Menge ID aller Begriffe des internen Aktors enthält zu diesem Zeitpunkt genau einen Identifier, der bei der Neuvergabe ersetzt wird.

Der Vorgang der Verfeinerung verbindet den (ursprünglichen) Empfänger c mit der Implementierung $impl_{ia}$ des Empfängers rcv_{ia}.

Im neu entstehenden funktionalen Graphen wird der Empfänger rcv_{ia} überflüssig. Damit seine Eigenschaften nicht verloren gehen, werden sie vereinigt mit den Eigenschaften des Begriffes c, der sich ergibt zu: $c = MergeProperties(c, rcv_{ia})$. Beide Begriffe haben wegen Regel 2 bzw. Regel 3 keine Alternativen (mehr).

Die Nachricht muß, wie der Empfänger, abgeglichen werden, da auch hier nur eine weiter existiert: $m = MergeMessageProperties(m, m_{ia})$

Auch mit den Parametern der Nachricht wird ähnlich verfahren: nur einer von beiden bleibt übrig und sie müssen abgeglichen werden. Zusätzlich müssen allerdings noch die Alternativen pro Paar gesondert betrachtet werden; es gelten dieselben Regeln wie für das Bestimmen der passenden Paare.

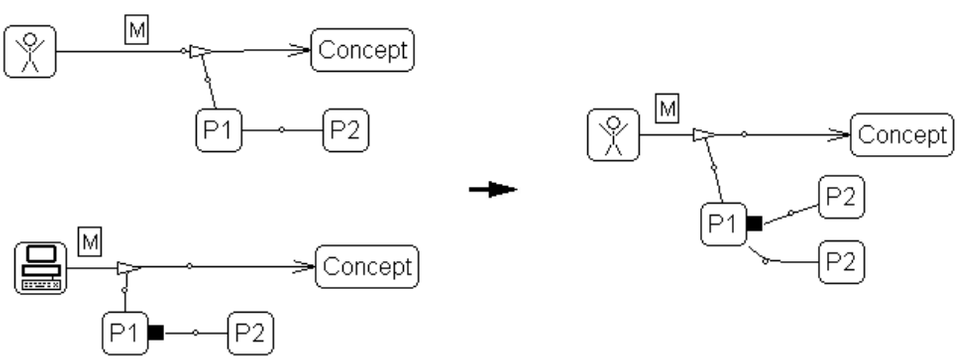


Abbildung 5.6.: Nach dem Abgleich erscheint P2 sowohl als Identität als auch als Alternative.

Durch Regel 4 kann es nach dem Abgleich der Eigenschaften vorkommen, daß ein Begriff sowohl identisch ist, als auch als Alternative vorkommt (Abbildung 5.6). Der Begriff wird dann aus der Menge der Identitäten entfernt.

Es gilt: $\forall mp_i = (p_{i1}; p_{i2}) \in matched$, mit $p_{i1} = (d_{i1}, c_{i1})$ und $p_{i2} = (d_{i2}, c_{i2})$:

- $c_{i1} = MergeProperties(c_{i1} = (..., ALT_{i1}, IDENT_{i1}...), c_{i2} = (..., ALT_{i2}, ...))$ und
- für $ALT_{i1} \neq \emptyset \wedge ALT_{i2} \neq \emptyset$ gilt: $ALT_{i1} = \begin{cases} ALT_{i1} \cap ALT_{i2} & \text{falls} \quad d_{i1} = output \\ ALT_{i1} \subseteq ALT_{i2} & \text{falls} \quad d_{i1} = input \end{cases}$
- für $ALT_{i1} = \emptyset$ gilt: $ALT_{i1} = ALT_{i2}$
- $\forall c_x : c_x \in IDENT_{i1} \wedge c_x \in ALT_{i1} : IDENT_{i1} = IDENT_{i1} \setminus \{c_x\}$

Dem funktionalen Graphen $fg' = (V', E', root) \in FG_{ea}$ werden die Knoten des funktionalen Teilgraphen fg_{ia} hinzugefügt, und die Kanten vom Begriff rcv_{ia} zur Implementierung ersetzt durch Kanten von c zur Implementierung. Formal:

- $V' = V \cup (V_{rest} \setminus rcv_{ia})$

- $E' = E \cup (E_{rest} \setminus e_k = (rcv_{ia}, z_k), \forall z_k \in impl_{ia})$
- $E' = E' \cup \{e_k = (c, z_k), \forall z_k \in impl_{ia}\}$

Parameterpropagation

Alternativen und Identitäten werden zum einen gebraucht, um konkrete Verfeinerungen für nicht abgeschlossene Pfade zu bestimmen und zum anderen, um als Ziel von Nachrichten durch einen Splitter aufgelöst zu werden. Sie müssen daher nach einem Mergevorgang nach unten in den neu angehängten funktionalen Teilgraphen propagiert werden:

Regel 16 *Alternativen und Identitäten an Parameterübergaben müssen (mit der Richtung der Übergabe) propagiert werden, damit sie bei der nächsten Auflösung eines nicht abgeschlossenen Pfades zur Verfügung stehen.*

Bei der Parameterpropagation wird der Teilgraph mit Wurzel $root = c$ betrachtet, also der Teilgraph unterhalb des Nachrichtenempfängers. Die abgeglichenen Input-Parameter der Nachricht werden nach unten propagiert:

- $\forall p_i = (d_i, c_i = (name_i, ...ALT_i, ..., ID_i)) \in P$
- $\forall c = (name_c = name_i, ..., ID_c \cap ID_i \neq \emptyset, ...) \in fg' = (V, E, c)$ gilt: $MergeProperties(c, c_i)$

Output-Parameter werden beim Rückweg nach demselben Prinzip *eine Ebene* nach oben propagiert.

Für das Beispielmodell ist der vollständige Strukturgraph bereits in Abbildung 5.1 dargestellt worden.

5.1.3. Komplexere Auswahlregeln

Der vorangegangene Abschnitt hat den Aufbau des funktionalen Graphen für den einfachsten Fall beschrieben, bei dem es genau eine passende Verfeinerung für eine Nachricht gibt. Im folgenden werden die komplexeren Regeln erörtert. Es wird jeweils ein Fall für sich betrachtet. Dabei darf aber nicht vergessen werden, daß manche Fälle gleichzeitig auftreten können und sie dann schrittweise bearbeitet werden.

Besserer Match von Parametern

Bei der Bestimmung der konkreten Verfeinerung $CONREF$ wurde bei der Auswahl der passenden Parameter als zweite Bedingung gefordert, daß die Begriffe namensgleich sind. Es kann nun einen Parameter p geben, der auf zwei Parameter p_1 und p_2 paßt, also für beide die richtige Richtung hat, namensgleich ist und eine kompatible Alternativenmenge hat. Abbildung 5.7 zeigt ein konkretes Beispiel.

Regel 17 *Gibt es für einen Parameter mehrere mögliche Bindungen, so wird diejenige Bindung ausgewählt, bei der die Namen der Parameter gleich (*nicht *nur namensgleich!) sind.*

Es gilt dann für $p = (name, ...) \equiv p_1 = (name_1, ...)$ und $p = (name, ...) \equiv p_2 = (name_2, ...)$:

$$\text{binde} \begin{cases} p_1 & \text{falls} \quad name_1 = name \\ p_2 & \text{falls} \quad name_2 = name \end{cases}$$

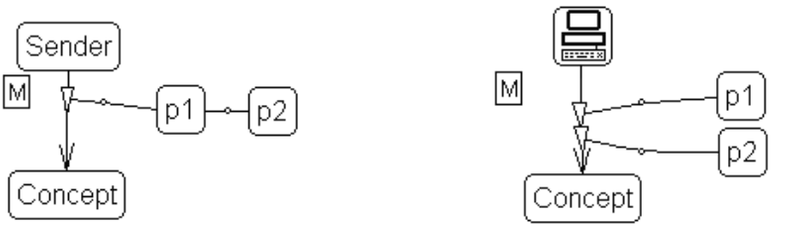


Abbildung 5.7.: Besserer Match von Parametern: p1 links wird an Parameter p1 rechts gebunden und nicht an p2.

Der Fall für zwei Parameter kann leicht erweitert werden auf n Parameter. Die gleichen Kriterien gelten, falls zwei (oder mehr) Parameter auf einen Parameter am internen Aktor passen.

Absicherung über Bedingungen

Wenn es zwei (oder mehr) passende Verfeinerungen gibt, liegt normalerweise ein Fehler im Modell vor, es sei denn, es gilt die Regel:

Regel 18 *Sind alle passenden Verfeinerungen über Bedingungen abgesichert, werden* alle *Implementierungen parallel benutzt.*

Wenn für zwei Nachrichten $m_1 = (..., COND_1)$ und $m_2 = (..., COND_2)$ zweier Verfeinerungen gilt $COND_1 \neq \emptyset$ und $COND_2 \neq \emptyset$, dann werden *beide* Verfeinerungen verwendet. Voraussetzung ist, daß beide Bedingungen nicht gleichzeitig erfüllt sein können[7]. An den Sender der

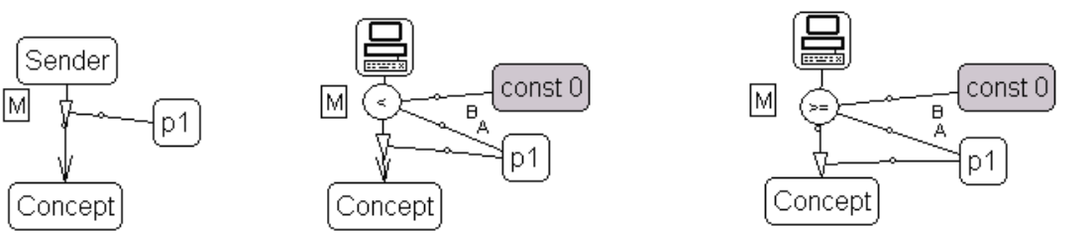


Abbildung 5.8.: Zwei interne Aktoren passen für die Nachricht M (links) und sind über Bedingungen abgesichert.

zu verfeinernden Methode werden beide Möglichkeiten angehängt. Um den Mergevorgang wie gewohnt durchführen zu können, wird die ursprüngliche Nachricht (und der Empfänger) verdoppelt, d.h. der Sender[8] schickt die Nachricht zweimal (Abbildung 5.8. Der Pfad $path = ...smc$ wird zu zwei Pfaden $path_1 = ...sm_1c_1$ und $path_2 = ...sm_2c_2$ mit $m_1 = m$, $c_1 = c$ und $m_2 = m$, $c_2 = c$ (Abbildung 5.9).

Jeder dieser neu entstandenen, nicht abgeschlossenen Pfade wird mit einer per Bedingung abgesicherten Verfeinerung gemergt. Der Mergevorgang selbst erfolgt für jeden Pfad wie bekannt.

Der Fall mit zwei passenden Verfeinerungen kann leicht auf den allgemeinen Fall mit n Bedingungen und damit n Verfeinerungen erweitert werden.

[7] VCDT wertet die Bedingungen aber nicht aus, es wird davon ausgegangen, daß sich beide Pfade durch die Bedingungen gegenseitig ausschließen.

[8] Der Sender darf kein externer Aktor sein. Ein Aufruf an einem externen Aktor darf nur bedingungslos ausgeführt werden.

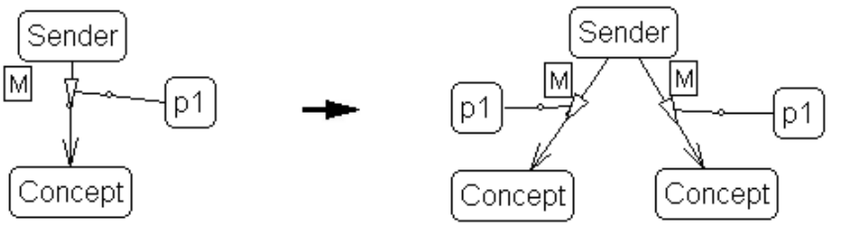


Abbildung 5.9.: Verdopplung der Nachricht bei zwei Verfeinerungen, die über Bedingungen gesichert sind.

5.1.4. Komplexere Merge-Regeln

Expansion von Alternativen beim Empfänger von Nachrichten

Wenn ein Begriff c Empfänger einer Nachricht m ist und er Alternativen ALT_c, $|ALT_c| > 0$, besitzt, dann wird der Begriff als Platzhalter für die Alternativen aufgefaßt. Jede Alternative ist damit Empfänger der Nachricht m (Regel 2). Der ursprüngliche Begriff c dient als Verzweigungspunkt im funktionalen Graphen.

Sei $c = (name, ctype, ..., ATR, ALT, IDENT, impl, ...)$ der Empfänger und $m = (mname, mtype, pos, sender, rcv, P, COND)$ die empfangene Nachricht. Im funktionalen Graph $fg = (V, E, root)$ geschieht folgendes:

1. $\forall alt_i \in ALT : V = V \cup alt_i \cup m_i$ mit $m_i = (mname, mtype, pos, c, alt_i, P, COND)$.
2. $\forall alt_i \in ALT : E = E \cup (c, m_i) \cup (m_i, alt_i)$
3. $E = E \cup (sender, c)$
4. $E = E \setminus (m, c)$ (Entsorgung der Kante (m,c))
5. $V = V \setminus m$
6. $\forall alt_i \in ALT : alt_i = (..., impl, ...)$ (Implementierung wird für alle Alternativen übertragen)

Der Typ des Begriffs ändert sich; er wird zu einem Splitterbegriff (1). Vom Splitterbegriff wird die empfangene Nachricht an alle Alternativen, die jetzt zu eigenen Graphknoten werden (2), weitergeleitet (3). Um den Graphen möglichst klein zu halten, wird die ursprüngliche Nachricht aus dem Graphen entfernt und eine direkte Verbindung zwischen dem Sender und dem Splitter eingefügt (5,6). Die ursprüngliche Verbindung ist redundant. Abbildung 5.10 zeigt den umgebauten funktionalen Graphen für das Beispielmodell vor und nach dem Umbau. Nach der Expansion des Graphen werden alle (neu entstandenen) unvollständigen Pfade wie gehabt verfeinert.

Bei der Suche nach Verfeinerungen wird wie folgt vorgegangen:

1. zunächst: wie bereits bekannt.
2. anschließend: für $CONREF = \emptyset$: suche fg_{ia} für Nachrichtenname und Splittername.

Bei mehrfach geschachtelten Splitterknoten wird Schritt 2 für alle Splitterknotennamen ausgeführt, bis es eine Verfeinerung gibt. Gilt am Ende immer noch $|POTREF| = 0$, so ist das Modell unvollständig nach Regel 14.

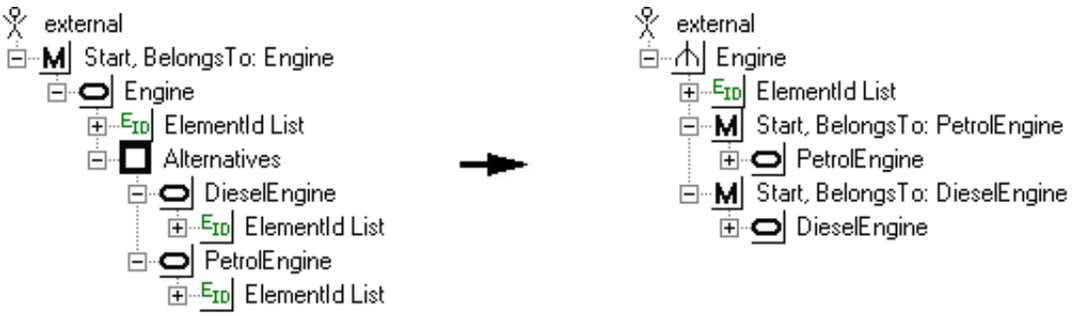


Abbildung 5.10.: Funktionaler Graph für einen Begriff mit Alternativen vor (links) und nach dem Umbau (rechts).

Splitter und Identitäten

Treffen Identitäten auf Splitter an internen Aktoren ergibt sich aus Regel 4[9] ein besonderes Vorgehen. Die beiden Modellierungen in Abbildung 5.11 sind für den Designer äquivalent. Für die Bestimmung der Verfeinerung bisher aber nicht.

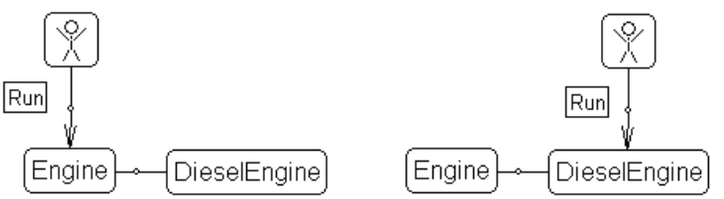


Abbildung 5.11.: Eine Nachricht an einen Begriff mit Identität (links) kann auch modelliert werden als Nachricht an den identischen Begriff (rechts).

Gegeben seien die Verfeinerungen für die Nachricht *Run* in Abbildung 5.12, bekannt aus Kapitel 3.3 und bereits umgeformt, wie dort beschrieben.

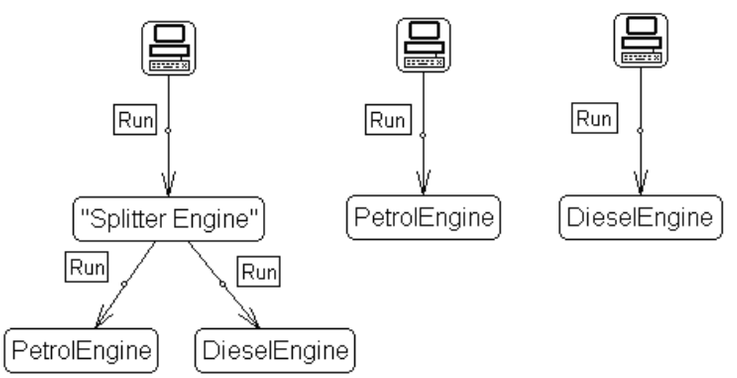


Abbildung 5.12.: Gegebene Verfeinerungen für "Run", bereits umgewandelt.

Es ist offensichtlich, daß sowohl die Verfeinerung mit dem Splitter als auch die Verfeinerung für *DieselEngine* für beide Modellierungen aus Abbildung 5.11 passen. Im linken Fall (Nachricht an *Engine*) wurde bisher die Verfeinerung mit Splitter gewählt, im rechten Fall die Verfeinerung

[9] Identitäten sind potentielle Alternativen

für *DieselEngine*. Es wird im Zweifelsfall immer der direkten Modellierung der Vorzug gegeben. Dies ist aber im konkreten Fall nicht korrekt. In beiden Fällen müßte die Verfeinerung für *DieselEngine* ausgewählt werden, denn dies ist die *speziellere* Verfeinerung.

Dies führt zu folgendem Vorgehen, wenn der Aufrufer eine Identitäten besitzt und es interne Aktoren mit Splittern gibt:

- Gibt es eine Verfeinerung mit Splitter und gibt es eine Verfeinerung für den identischen Begriff: Verfeinere ohne Splitter
- Gibt es eine Verfeinerung mit Splitter und keine Verfeinerung für den identischen Begriff: Verfeinere mit Splitter.

Die Verfeinerung selbst erfolgt wie bereits beschrieben.

Rekursion bei Splitterknoten

Einfache Rekursionen der Form $m_1c_1(cm)^*m_1c_1$[10] werden im funktionalen Graphen einfach abgebrochen, da sie für die Struktur des Modells keine neuen Informationen liefern. Durch die Rekursion entstehen keine neuen Kriterien zur Typfindung und Komponentenbestimmung. Eine Rekursion gilt als abgeschlossener Pfad und wird nicht weiter verfeinert.

Ein Splitterknoten schickt eine Nachricht an alle Alternativen (siehe Regel 2). Alternativen an Splitterknoten werden solange betrachtet, bis sie nichts neues mehr beitragen, d.h.:

Regel 19 *An einem Splitter muß es für alle alternativen Nachrichtenziele eine konkrete Verfeinerung geben. Solange diese Forderung nicht erfüllt ist, wird eine evtl. auftretende Rekursion ausgeführt.*

Eine Rekursion mit Beteiligung von Splitterknoten wird daher nicht abgebrochen, sondern solange ausgeführt, bis für *alle* alternativen Nachrichtenziele mindestens in einer Rekursionsstufe eine Verfeinerung gefunden werden konnte[11].

Sei d die aktuelle Rekursionstiefe, d_{max} die maximal zu untersuchende Rekursionstiefe. Die Zahl der noch zu verfeinernden Alternativen sei mit $altToRefine_d$ bezeichnet, der Zusatz d gibt die Rekursionstiefe an, es gilt $altToRefine_0 = |ALT|$.

Eine Rekursion mit Splitter wird abgebrochen, wenn:

- $d = d_{max}$ (maximale Rekursionstiefe erreicht)
- $altToRefine_{d-1} = altToRefine_d$ (keine Änderung in der letzten Stufe)
- $altToRefine_d = 0$ (alle Alternativen verfeinert)

Expansion von Alternativen bei Interfaces

Begriffe können explizit als Interface definiert werden, um vorhandene Komponenten zu integrieren. Die Begriffe, die das Interface implementieren, werden als Alternativen an das Interface angeschlossen.

[10] Schleifen sind hier absichtlich weggelassen

[11] nur bis zu einer vorgegebenen maximalen Rekursionstiefe, um durch Fehler im Modell die Verfeinerung nicht endlos zu probieren.

Alternativen an Interfaces werden wie normale Begriffe beim Empfang von Nachrichten per Splitter expandiert. Dies gilt auch, wenn die Alternativen zu einem Empfänger einer Verfeinerung gehören.

Nach Regel 3 gibt es nach der Expansion für jede Alternative einen eigenen internen Aktor, der den Oberbegriff (hier das Interface) nicht mehr enthält. Wird jetzt eine dieser Verfeinerungen verwendet, fehlt das (eigentlich explizit modellierte) Interface. Es wird daher im funktionalen Graphen beim Anhängen der Verfeinerung zwischengeschoben.

Beim Expandieren der Alternativen wird daher für jede Nachricht ein Element in der Menge $ifCalls$ angelegt, um die Information nicht zu verlieren, daß es sich eigentlich um einen Aufruf an ein Interface handelt:

$$ifCall_i = (mname, tname, c_{if})$$

Dabei gilt:

- $mname$, der Nachrichtenname der Verfeinerung
- $tname$, der Name des Zielbegriffs (target) nach der Expansion
- $c_{if} = (..., ctype = interface, ...)$, das explizit modellierte Interface

Beim eigentlichen Einhängen der Verfeinerung wird dann nachgeschaut, ob für die Verfeinerung ein Interface vorgesehen war.

Es gibt bei Interfaces noch einen Spezialfall zu beachten: ist ein Interface identisch mit einem normalen Begriff, so wird der Begriff wegen Regel 4 zu einer Alternative des Interfaces. Formal betrachtet gilt:

Sei $if = (name_{if}, ctype_{if} = interface, ..., IDENT_{if} = \{c\}, ...)$ ein Interface und $c = (name, ctype = normal, ..., IDENT, ...)$ ein normaler Begriff. Dann gilt:

$$if' = (name, ..., ALT_{if} \cup \{c\}, IDENT_{if} \setminus \{c\}, ...) \text{ und } c = (..., IDENT \setminus \{if\}, ...).$$

5.2. Die Begriffsliste

Die Begriffsliste enthält nach ihrem Aufbau die Informationen über alle Begriffe eines GCN Modells.

Voraussetzung für einen erfolgreichen Aufbau der Begriffsliste ist ein vollständiger funktionaler Graph, im folgenden mit fg bezeichnet. Nur durch den Aufbau des funktionalen Graphen werden Begriffe aus dem lokalen Kontext eines Sheets in den globalen Kontext des Systems transformiert.

Ziel ist dabei, festzustellen, welche Begriffe gleichen Namens sich tatsächlich treffen. Dies führt zur Definition von *Begriffsinstanzen*.

Begriffe gleichen Names vereinigen bei Aufbau des funktionalen Graphen ihre Identifier. Begriffe gleichen Namens mit einer Überschneidung ihrer Id-Menge ID werden als identisch betrachtet. Sie bilden eine eigene *Begriffsinstanz*.

Die Begriffsliste wird Stück für Stück (schrittweise) beim Durchlaufen des funktionalen Graphen gebildet. Es ist zu einem beliebigen Zeitpunkt nicht klar, welche Begriffe gleichen Namens zusammengehören werden und welche nicht. Es gibt daher für jeden Namen eines Begriffes nicht eine einzige Ausprägung, sondern eine Menge von Begriffsinstanzen.

Die Menge aller Begriffsinstanzen bildet die *Begriffsliste* eines GCN Modells:

$$\text{clist} := \{\text{ci}\}.$$

Eine Begriffsinstanz ist definiert als:

Definition 24 (Begriffsinstanz (concept instance)) *Eine Begriffsinstanz ist das Tupel*

$$ci = (name, ID, TID, ctype, dtype, exProp, MTH, VAR, ATR, ALT, SALT, IDENT)$$

mit:

- *name der* Instanzname.
- ID *die endliche Menge der* Identifier
- TID *die endliche Menge der* Typ-Identifier
- $ctype \in \{normal, config, const, interface\}$ *der* Typ *der Begriffsinstanz*
- $dtype = (btype, btypename, arraysize)$ *der* Datentyp *der Instanz.*
- $exProp$ *die* Existenzeigenschaften *der Instanz.*
- MTH *eine endliche Menge, die Menge der* Methoden.
- VAR *eine endliche Menge von Referenzen, die Menge der* impliziten Variablen
- ATR *eine endliche Menge von Referenzen, die Menge der* Attribute.
- ALT *eine endliche Menge von Referenzen, die Menge der* Alternativen.
- $SALT$ *eine endliche Menge von Referenzen, die Menge der* Splitteralternativen.
- $IDENT$ *die endliche Menge von Referenzen, die Menge der* Identitäten.

Der Instanzname *name* entspricht dem Namen des Begriffes. Eine Begriffsinstanz ist neben dem Namen durch seine Menge von Identifiern, ID, eindeutig bestimmt. Die Typ-Identifier, TID, werden später zur Bestimmung des Typs verwendet. Beide Mengen sind bereits von der Definition eines Begriffes bekannt.

Der Begriffstyp der Begriffsinstanz *ctype* und der Datentyp *dtype* sind auch bereits von der Definition eines Begriffes bekannt (siehe Kapitel 4.2). Sie werden für die Begriffsinstanz ungeändert übernommen. Alle Begriffe einer Begriffsinstanz müssen den gleichen Datentyp haben, sonst liegt ein Fehler im Modell vor[12].

Die Existenzeigenschaften $exProp$ einer Begriffsinstanz enthalten Informationen über die Existenz, die Gültigkeit und die Sichtbarkeit der Instanz.

Die Menge M enthält die Methoden eines Begriffes. Die an einen Begriff gesendeten Nachrichten werden zu Methoden seiner Begriffsinstanz. Die Implementierung der Methode selbst ist für die Begriffsinstanz nicht von Bedeutung.

Die Menge VAR enthält die impliziten Variablen für einen Begriff. Implizite Variablen sind Attribute, die automatisch vom System generiert werden, um der Begriffsinstanz bekannt gemachte Begriffe zu speichern. Die einzelnen Elemente der Menge sind *Referenzen* auf Begriffsinstanzen, die neben der Begriffsinstanz selbst eine Menge von referenzierenden Begriffsinstanzen enthält.

[12] das Tool VCDT stellt dies schon bei der Eingabe sicher

Die Mengen der Attribute ATR, Alternativen ALT und Identitäten $IDENT$ sind bereits geläufig (aus der Definition eines Begriffes). Sie bestehen hier allerdings nicht aus Begriffen, sondern - wie die Variablen - aus Referenzen auf Begriffsinstanzen.

Die Menge $SALT$ enthält die Menge der an einem Splitter aufgelösten Begriffsinstanzen, d.h. alle Empfänger einer Nachricht, die vom Splitter gesendet wird. Vor der Auflösung waren diese Begriffe Alternativen des Splitterbegriffs (siehe Kapitel 3.3).

Es wird hier explizit zwischen (normalen) Alternativen und aufgelösten Alternativen unterschieden, um bei der Klassifikation bessere Aussagen machen zu können.

Bleibt noch zu klären, was unter einer Referenz, den Methoden (mit ihren Parametern) und den Existenzeigenschaften genau zu verstehen ist:

Definition 25 (Referenz) *Eine* Referenz *ist definiert durch das Tupel*

$$ref = (ci, CIREF)$$

mit:

- *ci die Begriffsinstanz.*
- $CIREF = \{ci_i\}$ *die Menge der Begriffsinstanzen, die ci benutzen/referenzieren, die endliche Menge der* Begriffsinstanzreferenzen.

Für die in diesem Kapitel besprochenen Schritte erscheint es übertrieben, eine *Menge* von Begriffsinstanzreferenzen zu verwenden. Die Information ist später bei der Komponentenbestimmung aber sehr hilfreich und wird deshalb hier bereits gesammelt.

Definition 26 (Methode) *Eine* Methode *ist definiert durch das Tupel*

$$mth = (mname, R, PI, CIREF)$$

mit:

- *mname der* Name *der Methode.*
- *R die endliche Menge der* Rollen *der Methode.*
- *PI die endliche Menge der* Parameter *der Methode.*
- $CIREF = \{ci_i\}$ *die Menge der Begriffsinstanzen, die die Methode aufrufen, die endliche Menge der* Begriffsinstanzreferenzen.

Die Rollen einer Methode ergeben sich als Summe der Rollen aller externen Aktoren, die diese Methode aufrufen.

Die Parameter einer Methode sind analog zu Parametern einer Nachricht definiert, nur wird der Begriff c_p durch eine Begriffsinstanz ci_p ersetzt:

Definition 27 (Parameter einer Methode) *Ein Parameter einer Methode ist ein 2-Tupel*

$$pi = (ci_p, d)$$

mit folgenden Eigenschaften:

- ci_p *ist die zu transportierende Begriffsinstanz.*
- $d \in \{in, out\}$ *ist die* Richtung (direction).

Definition 28 (Existenzeigenschaften (existence properties)) *Die Existenzeigenschaften einer Begriffsinstanz werden durch das Tupel*

$$exProp = (generated, released, validValue, defined, visExternal)$$

beschrieben, dabei gilt:

- $generated \in N$ *gibt an, wie oft die Instanz* generiert *wurde*
- $released \in N$ *gibt an, wie oft Instanz* freigegeben *wurde*
- $validValue \in BOOL$ *gibt an, ob die Instanz einen* gültigen Wert *besitzt*
- $defined \in BOOL$ *gibt an, ob die Instanz* existiert *(z.B. als Attribut oder als* $fromEnv$*)*
- $visExternal \in BOOL$ *gibt an, ob* externe Sichtbarkeit *gegeben ist.*

Wird eine Instanz generiert, wird der Zähler *generated* um eins erhöht. Gleiches gilt für den Zähler *released*, der bei jedem Release erhöht wird. Das Verhältnis von *generated* zu *released* gibt darüber Auskunft, ob eine Begriffsinstanz zu einem bestimmten Zeitpunkt existiert oder nicht.

Ein gültiger Wert (Value) wird durch $validValue = TRUE$ angezeigt.

Manche Begriffe (Begriffsinstanzen) werden nicht explizit generiert, sondern sind einfach existent und damit *definiert* (z.B. als Attribut). In einem solchen Fall wird $defined$ auf $TRUE$ gesetzt.

Begriffsinstanzen können außerhalb des Modells (an externen Aktoren) sichtbar sein. Ist dies der Fall, wird die $visExternal = TRUE$ gesetzt.

Zum vollständigen Aufbau der Begriffsliste wird der funktionale Graph zweimal durchlaufen. Einmal, um alle Begriffsinstanzen zu sammeln, ein zweites Mal zur Bestimmung der impliziten Variablen, zur Existenzuntersuchung und zur Bestimmung unreferenzierter Objekteigenschaften.

Sammlung der Begriffe

Begriffe mit gleichem Namen haben bei ihrer Begegnung im funktionalen Graphen ihre ID Mengen vereinigt (vgl. $MergeProperties()$, Kapitel 4.4). Dadurch gilt für einen Begriff die Regel:

Regel 20 *(Zuordnungsregel) Ein Begriff wird durch seinen Namen und seine ID Menge genau einer Begriffsinstanz zugeordnet. Existiert sie nicht, wird eine neue Begriffsinstanz angelegt.*

Da mehrere Begriffe einer Begriffsinstanz zugeordnet werden, gilt:

Regel 21 *(Summenregel) Die Eigenschaften einer Begriffsinstanz bestehen aus der Summe der Eigenschaften aller Begriffe, die dieser Instanz zugeordnet werden.*

Regel 21 bedeutet informell, daß

- die Alternativen der Begriffe vereinigt werden.

  Begründung: Begriffe haben explizit Alternativen zugewiesen bekommen. Bei einer Verfeinerung werden der Parameterbindung unpassende Alternativen entfernt, daher bleiben im funktionalen Graphen nur Alternativen bestehen, die benötigt werden.

- die Attribute der Begriffe vereinigt werden.

  Begründung: Es ist egal, an welcher Stelle im Design ein Begriff Attribute erhält. Sie dürfen daher auch hinterher vereinigt werden.

- Eigenschaften wie der Datentyp bei Endbegriffen etc. sich nicht widersprechen dürfen. Es gelten dabei die Regeln von $MergeProperties$ analog.

Die Begriffsliste gebildet durch die Anwendung der beiden folgenden Abbildungen für jeden Begriff des funktionalen Graphen:

Definition 29 *(ConceptToCInstance) Die Abbildung*

$$ConceptToCInstance : c \rightarrow ci$$

ordnet einen Begriff $c = (name_c, ..., ID_c)$ *seiner Begriffsinstanz* $ci = (name_{ci}, ID_{ci}, ...)$ *zu. Dabei muß gelten:*

- $name_c = name_{ci}$
- $ID_c \subseteq ID_{ci} \neq \emptyset$

Definition 30 *(AddCProperties) Die Abbildung*

$$AddCProperties : ci \times c \rightarrow ci'$$

addiert die Eigenschaften des Begriffes $c = (name, ctype, etype, dtype, ATR, ALT, IDENT, m, ..., ID, TID)$ *mit* $m = (..., rcv, ...)$ *zu seiner Begriffsinstanz* $ci = (name_{ci}, ID_{ci}, TID_{ci}, ctype_{ci}, dtype_{ci}, ..., M_{ci}, ..., ATR_{ci}, ALT_{ci}, IDENT_{ci}, ...)$ *durch:*

- $\forall atr_j \in ATR : ATR_{ci} = ATR_{ci} \cup (ConceptToCInstance(atr_j), \emptyset)$
- $\forall alt_j \in ALT : ALT_{ci} = ALT_{ci} \cup (ConceptToCInstance(alt_j), \emptyset)$
- $\forall ident_j \in IDENT : IDENT_{ci} = IDENT_{ci} \cup (ConceptToCInstance(ident_j), \emptyset)$
- $ID_{ci} = ID_{ci} \cup ID_c$
- $TID_{ci} = TID_{ci} \cup TID_c$
- $ctype_{ci} = \begin{cases} ctype, & falls \quad ctype \neq splitter \wedge ctype \neq ifsplitter \\ ctype_{ci}, & sonst \end{cases}$
- $dtype_{ci} = \begin{cases} dtype, & falls \quad etype = end \vee etype = fromEnv \\ dtype_{ci}, & sonst \end{cases}$
- *AddMProperties(m, ConceptToCInstance(rcv))*

Die Definition der Abbildung $AddMProperties$ lautet:

Definition 31 *(AddMProperties) Die Abbildung*

$$AddMProperties : m \times ci_r \rightarrow ci'_r$$

addiert die Eigenschaften der Nachricht $m = (mname, mtype, pos, sender, rcv, P, COND)$ *zur Begriffsinstanz* $ci_r = (name_r, ..., existenceProp_r = (generated_r, released_r, ...), ..., MTH_r, ...)$ *und* $ci_{sender} = (name_s, ..., SALT_s, ...)$ *durch:*

- $MTH_r = MTH_r \cup (mname, R, PI, \emptyset)$, *falls* $mtype = normal$
- $\forall c_i \in PI = \{(c_i, d_i)\} : AddCProperties(ConceptToCInstance(c_i), c_i)$
- $\forall a = (..., rcv_a, ...) \in COND = \{(symbol, a, b, else)\} :$ $AddMProperties(a, ConceptToCInstance(rcv_a))$

- $\forall b = (..., rcv_b, ...) \in COND = \{(symbol, a, b, else)\}$:
  $AddMProperties(b, ConceptToCInstance(rcv_b))$
- *falls* $sender = (..., ctype_s, ...)$ *mit* $ctype_s = splitter \vee ctype_s = if splitter$ *gilt:*
  $SALT_s = SALT_s \cup (ConceptToCInstance(ci_r), \emptyset)$
- $generated_r = generated_r + 1$, *falls* $mtype = generator$
- $released_r = released_r + 1$, *falls* $mtype = released$

Es werden nur normale Nachrichten zu Methoden einer Begriffsinstanz. Generatoren, Releases und vom System erzeugte Nachrichten für die mathematischen Operationen werden nicht zu Methoden der Begriffsinstanz. Sie werden vom Codegenerator behandelt.

Der Aufbau der Begriffsliste für einen funktionalen Graphen fg=(V,E,root) erfolgt durch:

$$\forall c_i \in V : AddCProperties(ConceptToCInstance(c_i), c_i)$$

Nach dem Aufbau der Begriffsliste werden Begriffsinstanzen zusammengefaßt, deren ID Mengen sich überschneiden. Das mehrfache Anlegen eigentlich zusammengehöriger Begriffsinstanzen kann vorkommen, wenn die Begriffe in ungünstiger Reihenfolge in die Begriffsliste eingetragen werden.

Auch bei Begriffsinstanzen kann das von einzelnen Begriffen bereits bekannte Problem auftreten, daß eine Begriffsinstanz sowohl in der IDENT-, als auch in der ALT-Menge vorkommt (siehe auch Abbildung 5.6 in Kapitel 5.1.2). Die Auflösung des Problems erfolgt analog, mit derselben Begründung: die Begriffsinstanz wird aus der Menge IDENT entfernt und erscheint nur noch in der Menge ALT.

Die Begriffsliste für das Beispielmodell nach der Sammlung der Begriffe ist in Abbildung 5.13 dargestellt.

Bestimmung impliziter Variablen

Übergebene Parameter müssen von den Empfängern gespeichert werden, denn nach der Übergabe ist ein Zugriff auf die übergebenen Begriffe jederzeit möglich. Zur Speicherung der Begriffe (genauer: Begriffsinstanzen) werden implizite Variablen angelegt.

Der funktionale Graph wird dazu durchlaufen und jede Nachricht betrachtet. Input-Parameter einer Nachricht legen eine Variable beim Empfängerbegriff an, Output-Parameter beim Sender der Nachricht. Als Name der Variablen wird der Name der zu speichernden Begriffsinstanz gewählt.

Definition 32 *(AddVariable) Die Abbildung*

$$AddVariable : c \times c_p \rightarrow ci'$$

fügt der zum Begriff $c = (name, ..., ID, ...)$ *gehörenden Begriffsinstanz* $(ci = (name_{ci}, ID_{ci}, ..., VAR_{ci}, ...))$ *eine Variable für die zu* c_p *gehörenden Begriffsinstanz* (ci_p) *hinzu. Es gilt:*

$$VAR_{ci} = VAR_{ci} \cup (ci_p, \emptyset)$$

Zur Bestimmung der impliziten Variablen werden alle Pfadstücke $path_i = ...c_{i-1}l^*m_ic_i...$ eines funktionalen Graphen betrachtet mit $m_i = (..., mtype_i, ..., P_i, ...)$.

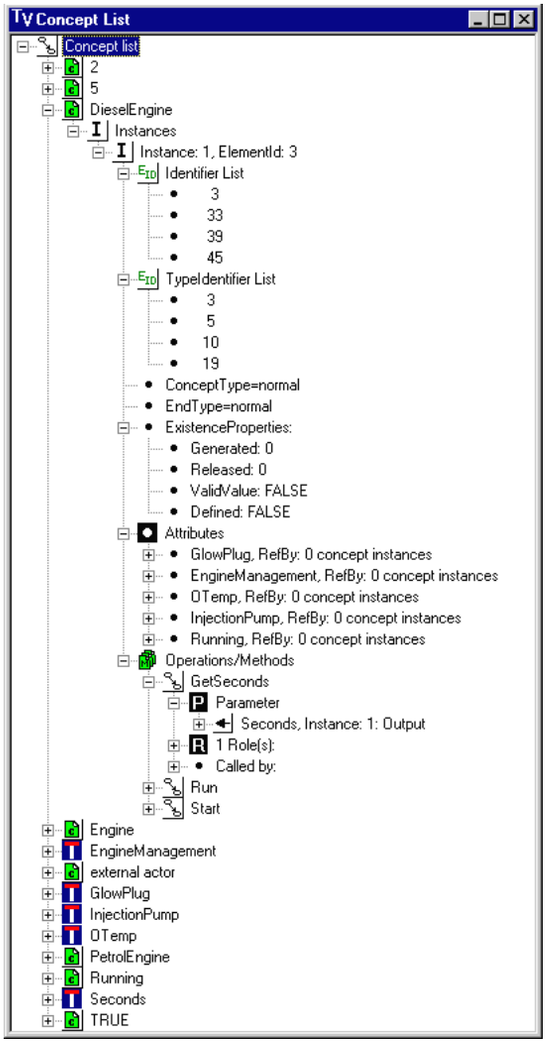


Abbildung 5.13.: Die Begriffsliste des Beispielmodells nach dem Aufsammeln der Begriffe aus dem funktionalen Graphen.

Es gilt $\forall p_j = (c_j, d_j) \in P_i$: $\begin{cases} AddVariable(c_{i-1}, c_j) & \text{falls} \quad d_j = output \\ AddVariable(c_i, c_j) & \text{falls} \quad d_j = input \end{cases}$

Es gilt zusätzlich für $mtype_i = generator$: $AddVariable(c_{i-1}, c_i)$

Nachdem alle Variablen angelegt wurden und eine Begriffsinstanz Variablen von mehreren Begriffsinstanzen besitzt, die denselben Namen haben, werden die ID-Mengen dieser Begriffsinstanzen vereinigt. Damit werden die Begriffsinstanzen zu *einer* Begriffsinstanz verschmolzen. Es werden auch die dann evtl. mehrfach vorhandenen Variablen (bzw. die Begriffsinstanzen, die sie speichern) vereinigt.

Regel 22 *Begriffsinstanzen, die sich über Variablen oder Attribute treffen, werden zusammengefaßt.*

Nicht alle impliziten Variablen werden tatsächlich benötigt. Nur Variablen, die außerhalb der aktuellen Methodenimplementierung (aktueller Kontext) referenziert werden, werden zu Attributen der Begriffsinstanz. Die Informationen für diese Entscheidung werden im nächsten Schritt, der *Existenzprüfung*, gesammelt.

Existenzprüfung

Die Existenzprüfung dient der frühzeitigen Aufdeckung von Designfehlern. Nicht initialisierte Objekte sind dabei eines der Hauptprobleme der Softwareentwicklung. Ein Fehler im Design ist immer dann vorhanden, wenn eine Nachricht an einen (lokal) unbekannten Begriff geschickt wird oder ein (noch) nicht gültiger (bzw. unbekannter) Wert per Parameter übergeben wird.

Nach Regel 9 müssen alle Begriffe generiert oder definiert sein. Die folgenden Regeln legen fest, wann eine Begriffsinstanz als generiert bzw. definiert gilt:

Regel 23 *Begriffsinstanzen sind* generiert, *falls einer der zugehörigen Begriffe*

- *explizit durch einen Generator erzeugt wird (released < generated).*
- *identisch zu einem generierten Begriff ist.*
- *von einem generierten Interface aufgerufen wird (durch die Generierung eines Interface wird die dazu gehörende Komponente auch generiert).*
- *ein Interface mit generierter Alternative oder generierter Splitteralternative ist.*

Regel 24 *Begriffsinstanzen sind* definiert, *falls für einer der zugehörigen Begriffe Regel 10 erfüllt ist. Zusätzlich gilt ein Begriff als definiert, falls er*

- *mit* fromEnv *gekennzeichnet ist.*
- *Output-Parameter einer Methode eines Endbegriffes ist.*
- *Input-Parameter an einem externen Aktor ist.*
- *Variable eines definierten/generierten Begriffes wird.*

Regel 25 *Begriffsinstanzen haben einen gültigen Wert* (validValue), *sobald einer der zugehörigen Begriffe*

- *eine Konstante ist.*
- *eine Identität zu einer Konstanten besitzt.*
- *einen Default-Wert hat.*
- *eine Zuweisung (damit auch eine mathematische Operation) besitzt (d.h. eine _get Nachricht auslöst) und alle Input-Begriffe der Operationen einen gültigen Wert besitzen.*
- *identisch zu einem Begriff ist mit validVaue = TRUE.*
- *Output-Parameter einer Methode eines Endbegriffes ist.*
- *Input-Parameter an einem externen Aktor ist.*
- *eine Alternative hat, die eine Konstante ist.*

Für alle Parameterübergaben wird geprüft, ob der ausführende Begriff (über seine Begriffsinstanz) die Begriffsinstanz des übergebenen Begriff kennt.

Für alle Nachrichten wird geprüft, ob der ausführende Begriff (über seine Begriffsinstanz) die Begriffsinstanz des Empfängers kennt. Kennen bedeutet dabei:

Regel 26 *Eine Begriffsinstanz* $ci_1 = (..., ATR_1, VAR_1, ALT_1, SALT_1, IDENT_1)$ kennt *eine andere Begriffsinstanz* ci_2, *wenn...*

- *sie identisch sind* ($ci_1 = ci_2$),
- *sie ein Attribut ist* ($ci_2 \in ATR_1$),
- *sie eine implizite Variable ist* ($ci_2 \in VAR_1$).
- *sie eine Alternative ist* ($ci_2 \in ALT_1$),
- *sie eine Ident-Verbindung haben* ($ci_2 \in IDENT_1$),

Zu beachten ist, daß diese Eigenschaften auch rekursiv gelten, z.B. eine gesuchte Begriffsinstanz kann Attribut von einem Attribut sein oder Attribut einer impliziten Variable, etc.

Fehler im Design liegen vor, wenn im Augenblick der Überprüfung

- $generated <= released$,
- Begriffsinstanzen keinen gültigen Wert besitzen,
- Begriffsinstanzen nicht bekannt sind,
- Begriffsinstanzen nicht definiert sind.

Am Ende der Existenzprüfung müssen die Zähler *generated* und *released* für alle generierten Begriffsinstanzen gleich sein. Falls nicht, gibt es nicht paarweise Aufrufe von Generatoren und Releases und eine Begriffsinstanz wird entweder mehrfach freigegeben oder nach dem Beenden des Systems nicht freigegeben.

Für das Beispielmodell wird zu Recht eine Fehlermeldung für das Attribut 'OTemp' erzeugt. 'OTemp' wird zwar verwendet, enthält aber keinen gültigen Wert. Im Beispiel ist ja auch nicht modelliert, wie die Außentemperatur gesetzt wird (siehe auch Kapitel 3.2).

Referenzierung von Objekteigenschaften

Unter den Objekteigenschaften werden impliziten Variablen, Attribute, (Splitter-) Alternativen und Methoden einer Begriffsinstanz verstanden. Es wird protokolliert, welche Begriffsinstanz welche anderen Begriffsinstanzen benutzt und welche Begriffsinstanz welche Methode aufruft.

Zur Bestimmung der referenzierten Objekteigenschaften werden alle Pfadstücke cl^*mc eines funktionalen Graphen betrachtet. Die Begriffsinstanz des Senders muß dabei - hier sei an die Existenzprüfung erinnert - die Begriffsinstanzen der Parameter und des Zielbegriffes kennen (siehe Abbildung 5.14). Wenn dies nicht der Fall ist, liegt ein Fehler im Design vor.

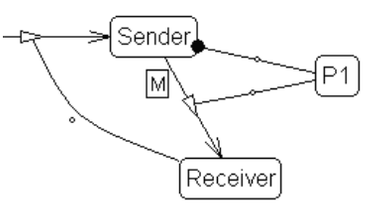


Abbildung 5.14.: Der Begriff 'Sender' kennt 'Receiver' als lokale Variable und den Parameter P1 als Attribut.

Sei c_s ein Sender und $ci_s = (..., VAR, ATR, ALT, SALT, IDENT)$ seine Begriffsinstanz. c_j bzw. ci_j bezeichnen die Begriffsinstanz eines Parameters oder des Empfängers. Es muß dann gelten:

$$\exists ref_j = (ci_j, CIREF_j) \in ATR \cup VAR \cup ALT \cup SALT \cup IDENT$$

Die folgende Regel legt fest, welche Referenzinfomationen aus einem Pfadstück des funktionalen Graphen gewonnen werden können:

Regel 27 *Aus jedem Pfadstück* $path_i = ...c_{i-1}l^*m_ic_i...$ *mit* $ci_{i-1} = ConceptToCInstance(c_{i-1})$, $ci_i = ConceptToCInstance(c_i)$ *und* $m_i = (..., mtype_i, ..., P_i, ...)$ *(*$mth_i = (..., PI_i, ...)$ *die passende Methode dazu) können die folgenden drei Aussagen über die Referenzierung von Begriffsinstanzen abgeleitet werden:*

1. *der Sender referenziert den Empfänger der Nachricht:* $CIREF_i = CIREF_i \cup ci_{i-1}$
2. *der Sender referenziert die Parameter der Nachricht:* $\forall pi_j \in PI_i$: *mit* $pi_j = (ci_j, di_j)$: $CIREF_j = CIREF_j \cup ci_{i-1}$
3. *die Methode wird vom Sender aufgerufen:* $mth_i = (..., CIREF_m)$*:* $CIREF_m = CIREF_m \cup ci_{i-1}$

Falls die Referenz eine Variable ist ($ref_j \in VAR$), wird die Referenz nur dann eingetragen, wenn die Referenz sich *nicht* im lokalen Scope befindet.

Ein lokaler Scope ist:

Definition 33 (lokaler Scope) *Als lokaler Scope einer Nachricht m=(...,rcv,...) wird der Teilgraph eines funktionalen Graphen* $fg_{ls} = (V, E, rcv)$ *bezeichnet, dessen Wurzel der Begriff rcv ist.*

Das Ergebnis dieses Schrittes ist eine Aussage, welche Objekte tatsächlich benutzt werden. Nicht referenzierte Variablen sind als Methoden-lokal zu betrachten. Sie werden aus den Begriffsinstanzen entfernt. Es ist Aufgabe des Codegenerators diese Variablen ggf. zu erzeugen. Alle anderen Variablen - *globale* Variablen genannt - werden zu Attributen der Begriffsinstanz. Alle weiteren Beziehungen bzw. nicht vorhandenen Beziehungen werden später ausgewertet.

Die vollständige Begriffsliste für das Beispielmodell ist in Abbildung 5.15 dargestellt. Man erkennt, daß im Beispielmodell einige Begriffe (z.B. 'Dieselengine') zwar generiert, aber nicht wieder freigegeben werden. Das liegt daran, daß der Shutdown nicht explizit modelliert wurde. Gut zu sehen ist auch die implizite Variable 'Seconds' der Begriffsinstanz 'Dieselengine', die als globale Variable klassifiziert wurde.

5.3. Die Klassifikation der Begriffe

In diesem Kapitel wird häufig von *Typ-Begriffsinstanzen* gesprochen. Nur sie werden für die Klassifikation verwendet. Mehrere Begriffsinstanzen können dieselbe Typ-Begriffsinstanz besitzen.

Die Klassifikation hat die Aufgabe, allen Typ-Begriffsinstanzen (und damit allen Begriffen) einen konkreten *Typ* zu geben. Dazu werden die Beziehungen der Typ-Begriffsinstanzen untereinander ausgewertet. Jede Art von Beziehung (z.B. Identitäten oder Alternativen) ist unterschiedlich zu werten. Dieser Abschnitt wird deshalb die einzelnen *Beziehungsarten* nacheinander betrachten und beschreiben, welche Erkenntnisse aus den Beziehungen für die Typbestimmung gewonnen werden können.

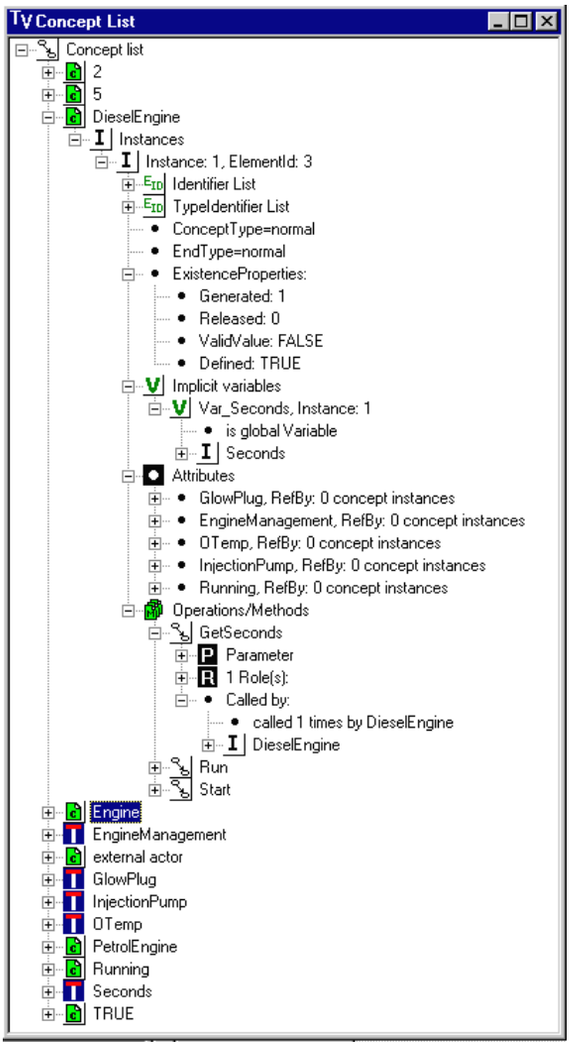


Abbildung 5.15.: Die vollständig aufgebaute Begriffsliste des Beispielmodells.

Abbildung 5.16 zeigt zur Erinnerung die Beziehungen zwischen Begriffsinstanzen, den Typ-Begriffsinstanzen und den Typen. Die Typ-Begriffsinstanzen sind dabei Begriffsinstanzen mit derselben Definition wie die Begriffsinstanzen. Sie werden aus den Begriffsinstanzen gewonnen, indem alle Begriffsinstanzen zusammengefaßt werden, deren Typ-Identifier sich überschneiden.

Bevor der genaue Ablauf der Klassifikation besprochen wird, wird ein Blick auf die verwendeten Typen geworfen. Es werden dabei interne Typen, Konfigurationstypen und externe Typen unterschieden.

5.3.1. Typen

Interne Typen

Während der Klassifikation der Typ-Begriffsinstanzen werden nur die *internen Typen* verwendet. Sie umfassen die in Tabelle 5.1 dargestellten Varianten.

Für viele Typen wird unterschieden zwischen dem Typ und demselben Typ als Konstante. Dies erleichtert die Klassifikation, denn es macht häufig einen Unterschied in der Schlußfolgerung, ob

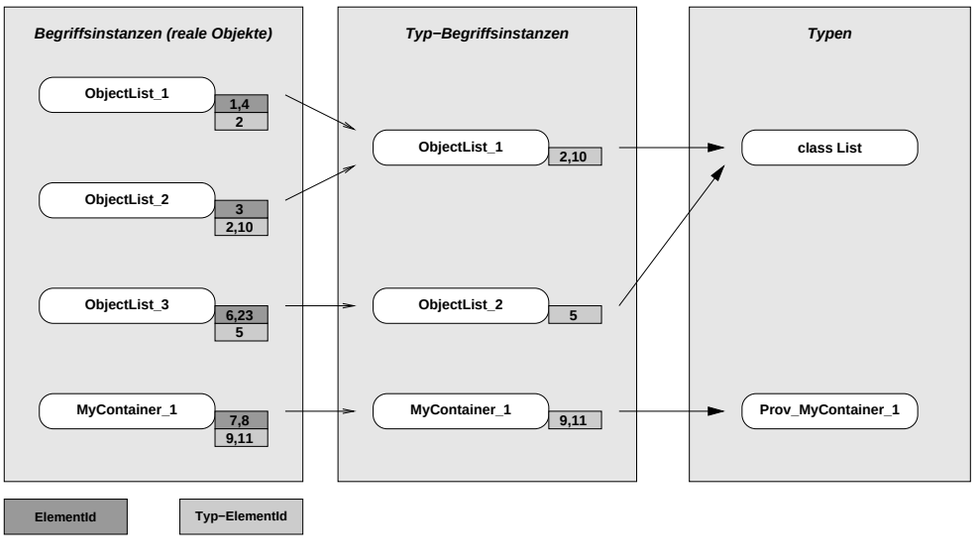


Abbildung 5.16.: Die Zusammenhänge zwischen Begriffsinstanzen, Typ-Begriffsinstanzen und Typen.

eine Konstante auf einen nicht konstanten Begriff oder eine andere Konstante trifft.

Es fällt auf, daß es intern nur noch einen numerischen Typ gibt: *numeric*. Numerische Werte, die von außen (z.B. als Rückgabewert einer externen Methode) ins Modell gebracht werden, haben ein festes Format (z.B. 16 Bit integer, also in C++ *short*). Innerhalb des Modells wird der Wert dann für einige Berechnungen benutzt, um ihn schließlich als *long* (32 Bit Integer) einer anderen externen Methode als Parameter zu übergeben.

Es stellt sich die Frage, welches Format dieser numerische Wert innerhalb des Modells haben soll[13]? Wenn auch Gleitkommazahlen erlaubt sind, wird die Frage noch komplexer.

Eine Festlegung auf einen konkreten numerischen Typ sollte nicht zu früh geschehen. Numerische Werte werden deshalb nicht durch unterschiedliche Typen (short, long, float etc.) repräsentiert, sondern nur durch den Typ *numeric*, der eine Anzahl Bits für die Mantisse und den Exponenten besitzt. Erst am Ende der Klassifikation wird daraus wieder ein konkreter Typ bestimmt.

Bei Gleitkommazahlen nach dem IEEE Standard [IEE] hat die Mantisse bei einfacher Genauigkeit (*float* in C) eine Länge von 23-Bit und ist normiert auf 0,1xxxx wobei die 1 in der Darstellung als redundant entfällt. Der Exponent hat eine Länge von 8 Bit. Wegen des Vorzeichens bleiben 7 Bit von Bedeutung.

Für einen Integerwert (als Beispiel dient ein 32-Bit *long* Wert) gilt dann folgendes: Er wird intern dargestellt mit einem Exponenten von 0 Bit ($2^0 = 1$) und einer Mantisse von 31 Bit (1 Bit wird für das Vorzeichen benötigt).

Begriffe, die von außen (also über Endbegriffe) ins Modell gebracht werden (also Output-Parameter sind), können innerhalb des Modells keinen kleineren Wertebereich als außen besitzen, einen größeren aber durchaus. Analog können Begriffe, die vom Modell auf die Außenwelt wirken, keinen größeren Wertebereich als die Außenwelt haben. Numerische Werte besitzen deshalb in einem GCN Modell nicht einen einzigen, sondern einen minimalen und einen maximalen Exponenten und eine minimale und maximale Mantisse. Für Parameter, die in beide Richtungen wirken, ist der numerische Typ eindeutig festgelegt (minimaler Wert ist gleich dem maximalen

[13]Man würde wohl intuitiv *long* wählen, denn *long* umfaßt auch den Wertebereich von short.

Typ	*konstanter Typ*	*Name*	*Beschreibung*
numeric	const_numeric	nein	numerische Werte
byte	const_byte	nein	nicht interpretiertes Byte (8 Bit)
boolean	const_boolean	nein	boolscher Wert
string	const_string	nein	Zeichenkette
void	const_void	nein	kein spezieller Typ
uniqueId	const_uniqueId	ja	eindeutiger Identifier
enum	-	ja	Aufzählungstyp
enumElement	-	nein	Element eines Aufzählungstypen
struct	-	ja	Struktur
class	-	ja	Klasse
reference	-	ja	Referenz
interface	-	ja	Interface einer Komponente
component	-	ja	existierende Komponente
library	-	ja	externe Bibliothek
none	const none	nein	noch kein Typ

Tabelle 5.1.: Interne Typen für die Klassifikation. Die Spalte Name enthält ein *ja*, falls der Typ einen qualifizierenden Namen benötigt.

Wert).

Durch Beziehungen zu anderen Begriffen (z.B. durch mathematische Operationen) werden der minimale, maximale oder beide Wertebereiche eingeschränkt. Am Ende besitzt jeder Begriff numerischen Typs einen Bereich, den er sicher annehmen kann.

Jeder Typ besitzt eine Deklaration, der seine Eigenschaften genauer spezifiziert. Für einen numerischen Typ lautet sie:

Definition 34 (Deklaration eines numerischen Typs (declaration of numeric type))
Die Deklaration *eines numerischer Typs ist das Tupel*

$$decl_{nt} = (minExp, maxExp, minMant, maxMant)$$

mit:

- *minExp der minimale Exponent des numerischen Typs.*
- *maxExp der maximale Exponent des numerischen Typs.*
- *minMant die minimale Mantisse des numerischen Typs.*
- *maxMant die maximale Mantisse des numerischen Typs.*

Für die anderen Typen soll hier auf eine genaue Beschreibung der Deklaration verzichtet werden.

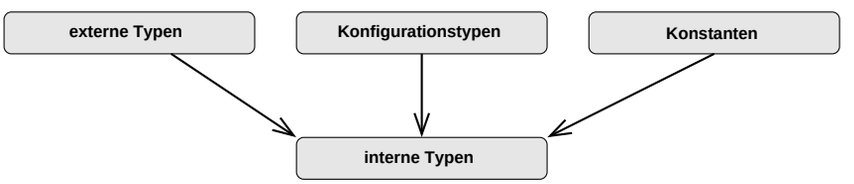


Abbildung 5.17.: Abbildung auf interne Typen.

Die folgenden Abschnitte beschreiben, was externe Typen und Konfigurationstypen sind und wie sie und Konstante in *interne Typen* überführt werden (Abbildung 5.17).

externe Typen		interne Typen		
Typ (btype)	*Bits*	*Typ*	*Mantisse*	*Exponent*
small	8	numeric	7	0
short	16	numeric	15	0
long	32	numeric	31	0
hyper	64	numeric	63	0
float	32	numeric	24	7
double	64	numeric	53	10

Tabelle 5.2.: Numerische externe Typen eines GCN Modells und die interne Darstellung.

externer Typ	*interner Typ*	*Beschreibung*
byte	byte	nicht interpretiertes Byte (8 Bit)
boolean	boolean	boolscher Wert
w_char_t	string	Zeichenkette
void	void	kein spezieller Typ
uniqueId	uniqueId	eindeutiger Identifier
enum	enum	Aufzählungstyp
Konstante[16]	enumElement	Element eines Aufzählungstyp
struct	struct	Struktur
class	class	Klasse
interface	interface	Interface einer Komponente
component	component	existierende Komponente

Tabelle 5.3.: Nicht numerische externe Typen eines GCN Modells und ihre interne Darstellung.

Externe Typen

Die Typen aus der realen Programmierumgebung, wie sie durch die Endbegriffe (siehe Abschnitt 4.2) ins GCN Modell gebracht werden, werden als *externen Typen* bezeichnet. Um ein Typchaos zu vermeiden, werden nur die die Typen der RPC[14]-Spezifikation [Maj99] verwendet. Durch diese Straffung werden die Probleme vermieden, die (z.B. in C++) durch frei definierbare Typen entstehen können (z.B. `typedef DWORT long`) [15].

Der externe Typ eines Endbegriffs (einer Begriffsinstanz) setzt sich zusammen aus:

- dem Basistypen *btype*
- dem quaifizierenden Namen *btypeName*; einige Basistypen benötigen keinen zusätzlichen Namen

Tabelle 5.2 faßt die numerischen Typen zusammen, Tabelle 5.3 die nicht numerischen Typen. Sie zeigen gleichzeitig die Abbildung auf die internen Typen. Für die numerischen Typen ist jeweils der Exponent und die Mantisse angegeben.

[14] Remote Procedure Call

[15] Um den Codegenerator zu unterstützen, kann im Tool zusätzlich ein Deklarartionsstring (z.B. `DWORT *`) angegeben werden.

externer Typ	*interner Typ*	*Beschreibung*
integer	numeric	ganze Zahl
fraction	numeric	Gleitkommazahl
byte	byte	nicht interpretiertes Byte (8 Bit)
boolean	boolean	boolscher Wert
string	string	Zeichenkette
uniqueId	uniqueId	eindeutiger Identifier
alternative	enum	Aufzählungstyp

Tabelle 5.4.: Typen für Konfigurationsdaten eines GCN Modells und ihre interne Darstellung.

Konfigurationstypen

Konfigurationsbegriffe sind Properties von (potentiellen) Komponenten. Sie werden vor dem Gebrauch der Komponente konfiguriert. Um nicht immer alle Properties einstellen zu müssen, hat ein Konfigurationsbegriff einen sinnvollen Defaultwert.

Konfigurationsbegriffe verhalten sich häufig wie Konstanten. Ihre Typen können daher nicht immer vollkommen automatisch bestimmt werden. Sie müssen deshalb bei der Eingabe grob klassifiziert werden. Die Auswahl an möglichen Typen ist allerdings im Vergleich zu Endbegriffen eingeschränkt (Tabelle 5.4).

Typen wie *class*, *struct*, *component* oder *library* sind für Konfigurationsbegriffe nicht sinnvoll und deswegen nicht vorgesehen.

Um den System einen Anhaltspunkt zu geben, welchen Exponenten ein numerisches Konfigurationsdatum annehmen kann, müssen die Typen *numeric* (Integerwerte) und *fraction* (Gleitkommazahlen) explizit unterschieden werden. Für beide Typen können sogenannte *Constraints* (OCL [IBM98], Teil der UML ab Version 1.1) angegeben werden. Bei der Umrechnung in interne Typen werden die Constaints benutzt, um die Mantisse und den Exponenten zu bestimmen.

Typen von Konstanten

Konstante Begriffe können grundsätzlich fünf interne Typen annehmen: *const none*, *numeric*, *string*, *boolean* und *uniqueId*. Sie werden nicht explizit vom Designer klassifiziert, sondern automatisch. Es gelten dabei die Regeln:

Regel 28 *Eine Konstante wird als* string *klassifiziert, wenn der Wert in Anführungszeichen eingeschlossen ist.*

Regel 29 *Eine Konstante wird als* boolean *klassifiziert, wenn der Wert 'true' oder 'false' ist. Dabei wird Groß- und Kleinschreibung nicht berücksichtigt.*

Regel 30 *Eine Konstante wird als* numeric *klassifiziert, wenn der Wert eine Zahl ist. Aus dem Defaultwert wird die Mantisse und der Exponent bestimmt.*

Konstanten, die UniqueIds bezeichnen, können nicht sicher bestimmt werden, da ihre Form vom verwendeten Komponentenmodell abhängt (in COM haben die GUIDs z.B. das Format "{...{...}}"). Sie werden deshalb zunächst als *const none* klassifiziert.

[16] nur Konstante, die Alternative an einem Begriff sind

interner Typ	*Name in Typliste*
string, const string	*string*
byte, const byte	*byte*
boolean, const boolean	*boolean*
uniqueId, const uniqueId	*uniqueId*
void	*void*
numeric	*numeric_<name>_<instNo>*
const numeric	*const_numeric_<value>*
none	*Prov_<name>_<instNo>*
const none	*const_none_<value>*
struct	*struct <btypeName>*
enum	*enum <btypeName>*
class	*class <btypeName>*
component	*component <btypeName>*
interface	*interface <btypeName>*

Tabelle 5.5.: Aufbau der Typnamen für alle internen Typen. *name* steht für den Begriffsinstanznamen, *instNo* für die Instanznummer der Begriffsinstanz. *Value* gibt den Wert der Konstanten an und *btypeName* ist der qualifizierende Name eines externen Typs.

Konstanten, die einen symbolischen Wert repräsentieren (z.B. `MACHINE_RESET`), werden ebenso als *const none* klassifiziert.

5.3.2. Typliste

Alle Typen eines GCN Modells werden in einer Typliste abgelegt. Die Menge aller Typen bildet die *Typliste* eines GCN Modells:

$$\text{tlist} := \{\text{type}\}.$$

Ein Typ ist definiert als:

Definition 35 (Typ (type)) *Ein Typ ist das Tupel*

$$type = (name, inttype, dtype, decl)$$

mit:

- *name der* Typname.
- *itype der* interne Basistyp*; siehe Tabelle 5.1.*
- *dtype der* externe Typ *(bei Endbegriffen).*
- *decl die* Deklaration *des Typs.*

Jeder Typ hat einen eindeutigen Namen *name*. Verschiedene Begriffe bzw. Typ-Begriffsinstanzen können denselben Typ haben. Der Typname wird in Abhängigkeit des internen Basistypen *itype* gebildet (siehe Tabelle 5.5). Die entstehenden Typnamen sollten dabei möglichst intuitiv verständlich sein.

Die einfachen konkreten Typen wie *string*, *void* oder *uniqueId* erhalten den Namen des Typs (also z.B. 'string') ohne weitere Zusätze. Die entsprechenden Konstanten erhalten denselben Namen wie ihr nicht konstantes Äquivalent.

Ein numerischer Typ erhält zunächst einen vorläufigen provisorischen Namen der Form *numeric_<instname>_<instno>*, denn der konkrete numerische Typ wird erst durch die Beziehungen zu anderen Begriffen bestimmt. Für numerische Konstanten gilt diese Aussage entsprechend. Die minimale Mantisse und der minimale Exponent werden zwar durch den Wert der Konstanten festgelegt, die jeweiligen Maximalwerte aber nicht. Sie ergeben sich erst durch die Beziehungen zu anderen Begriffen. Daher bekommen numerische Konstanten ebenfalls einen provisorischen Namen der Form *const_numeric_<value>* (z.B. *const_numeric_3.1415*).

Da jede Begriffsinstanz ohne konkreten Typ (*itype* = *none*) potentiell einen neuen Typ definieren kann (man denke hier nur an Klassen), wird er in der Typliste unter einem provisorischem Namen angelegt, der sich aus dem Begriffsnamen und der Instanznummer (um den Namen eindeutig zu machen) zusammensetzt: Prov_<instname>_<instno>.

Eine Konstante ohne Typ (sie haben den internen Typ *const none*) bekommt den provisorischen Namen *const_none_<value>*.

Die komplexeren Typen *struct*, *enum*, *class*, *component* und *interface* werden unter ihrem *vollständigen Namen* in die Typliste eingetragen. Er setzt sich aus dem Namen des Typs und dem qualifizierenden Namen zusammen (z.B. *class CHardware* oder *enum Farben*).

Die Deklaration *decl* enthält alle Fakten, die einen Typen bestimmen. Bei Klassen sind dies z.B. die Attribute und Methoden, bei Aufzählungstypen die Elemente der Menge und bei Konstanten ihr Wert. Für numerische Werte wurde die Deklaration bereits genau definiert, für die anderen Typen soll dies, wie bereits erwähnt, hier nicht erfolgen.

5.3.3. Darstellung der Beziehungen zwischen Typen

Die Klassifikation der Typen geschieht über die Auswertung von Beziehungen zwischen den Begriffen. Jeder Begriff hat einen internen Typ (zumindest *none*) und so treffen über die Beziehungen eigentlich interne Typen aufeinander.

Für jede mögliche Kombination von Typen muß daher eine Aussage getroffen werden. Diese Aussagen werden in Form einer zweidimensionalen Matrix (Tabelle) dargestellt. Für jede Art von Beziehung (Alternativen, Identitäten etc.) wird eine eigene Matrix aufgestellt.

Jedes Matrixelement enthält eine Aussage über die beteiligten Typen. Die grundsätzlichen Aussagen, hier Aktionen genannt, sind in Tabelle 5.6 dargestellt.

Die Aktionen *ok* und *error* sind selbsterklärend. Manche Aktionen können durch eine Angabe in runden Klammern näher klassifiziert werden. Dadurch wird bestimmt, auf welchen der Eingangstypen die Aktion angewandt wird. Die Aktion *type (A)* weist z.B. dem Eingangstypen A den internen Typ *type* zu.

Die Aktion *SizeCheck* gleicht zwei numerische Typen bzgl. ihrer Mantisse und ihres Exponenten ab:

Definition 36 (SizeCheck) *Die Abbildung SizeCheck*

$$SizeCheck : type_A \times type_B \rightarrow type'_A \times type'_B$$

Aktion	*Beschreibnung*
type (A)	Der Eingangstype A wird zu $< type >$.
type (B)	Der Eingangstype B wird zu $< type >$.
ok	Typen sind kompatibel, keine Aktion
error	Typen sind inkompatibel, Fehlermeldung
SizeCheck	Abgleich der numerischen Typen
Merge	Abgleich von *struct* und *enum*
IFCheck	Abgleich von Interfaces
BaseIF (A)	Der Eingangstyp A wird zum $Basisinterface$
BaseIF (B)	Der Eingangstyp B wird zum $Basisinterface$
V(A)	Wiederhole Aktion für den Valuetyp von A
V(B)	Wiederhole Aktion für den Valuetyp von B
V(A)(B)	Wiederhole Aktion für den Valuetyp von A und B

Tabelle 5.6.: Mögliche Aktionen in einer Beziehungsmatrix.

gleicht zwei numerische Typen $type_A = (minExp_A, maxExp_A, minMant_A, maxMant_A)$ *und* $type_B = (minExp_B, maxExp_B, minMant_B, maxMant_B)$ *ab:*

- $minMant'_{A,B} = \begin{cases} minMant_A, & falls \quad minMant_A > minMant_B \\ minMant_B, & sonst \end{cases}$
- $maxMant'_{A,B} = \begin{cases} maxMant_A, & falls \quad maxMant_A < maxMant_B \\ maxMant_B, & sonst \end{cases}$
- $minExp'_{A,B} = \begin{cases} minExp_A, & falls \quad minExp_A > minExp_B \\ minExp_B, & sonst \end{cases}$
- $maxExp'_{A,B} = \begin{cases} maxExp_A, & falls \quad maxExp_A < maxExp_B \\ maxExp_B, & sonst \end{cases}$

Die Aktion *Merge* dient dazu, bei Strukturen und Aufzählungen Typen zusammenzufassen, die sich über Identitäten treffen. Unterschieden wird dabei, ob die beteiligten Typen von außen kommen (aus Endbegriffen) oder intern entstanden sind. Sie werden im folgenden mit *Endtyp* und *provisorischer Typ* bezeichnet. Für Strukturen werden die Attribute betrachtet, für Aufzählungen die Alternativen. Es sind drei Fälle zu unterscheiden:

- Beide Typen sind provisorisch: Vereinige die Alternativen bzw. Attribute; vereinige beide Typen.
- Beide Typen sind Endtypen: Fehlerausgabe, falls sie nicht in allen Alternativen bzw. Attributen übereinstimmen.
- Ein Typ ist Endtype und der andere ein provisorischer Typ: der provisorische Typ muß eine Untermenge besitzen; er wird zum Endtyp.

Die Aktion *CheckIF* behandelt die komplexe Interface-Problematik. Sie entsteht, weil unterschiedliche Interfaces nicht kompatibel sind. Sie sind aber alle von einem Basisinterface (in COM z.B. *IUnknown*) abgeleitet. Man muß nun feststellen, ob ein Begriff als Typ ein *spezielles* Interface ist, oder ein *Basisinterface*. In diesem Zusammenhang macht auch die Aktion *BaseInterface* Sinn. Sie macht aus einem (unspezifischen) Typ *Interface* den konkreten Typ *Basisinterface* (in COM also *interface IUnknown*).

Eine Besonderheit gibt es zu beachten, wenn Begriffe als Klassen klassifiziert werden. In Abschnitt 4.3 wurde beschrieben, daß ein Begriff auch ein implizites Attribut *Value* besitzt. Für die meisten Typen hat dies keine Auswirkung, da der Typ identisch mit dem Typ des Values (im weiteren Verlauf *Valuetyp* genannt) ist. Bei Klassen ist dies anders. Hier wird explizit zwischen dem Typ des Begriffs (also *class*) und dem Valuetyp (z.B. *numeric*) unterschieden. Bei einem Treffen von Klassen auf andere Typen ist eine Überprüfung des Valuetyps notwendig. Die Aktion *V* fordert daher eine Wiederholung der aktuellen Auswertung unter Ersetzung eines oder beider Eingangstypen durch den entsprechenden Valuetyp. Welche Eingangstypen ersetzt werden sollen, wird wieder in runden Klammern qualifiziert.

5.3.4. Klassifikation

Ein GCN Modell enthält viele verschiedene Typen, von denen am Anfang nur die Typen der Endbegriffe bekannt sind. Die anderen Typen ergeben sich durch die Beziehungen der Begriffe untereinander. Grundlage dieses Ansatzes ist die aus der Systemtheorie [Unb93] bekannte *Beobachtbarkeit* und *Steuerbarkeit*. Dies bedeutet, daß jeder Begriff entweder durch seine Wirkung am Ausgang (Beobachtbarkeit) oder durch die Wirkung am Eingang (Steuerbarkeit) bestimmt werden kann. Falls er weder beobachtbar noch steuerbar ist, ist er überflüssig (toter Code) oder es liegt ein Fehler im Modell vor.

Die Klassifikation selbst umfaßt die Schritte:

1. Initiales Anlegen der Typliste
2. Auswertung der Beziehungen
3. Zuammenfassung der Typen

Initiales Anlegen der Typliste

Zunächst wird jeder Begriffsinstanz ein Typ (mit internem Basistyp und Typname) zugeordnet. Dies geschieht für Endbegriffe, Konfigurationsbegriffe und Konstanten wie bereits beschrieben. Typ-Begriffsinstanzen mit Methoden bekommen den internen Typ *class*. Typ-Begriffsinstanzen mit Attributen, aber ohne Methoden werden als *struct* klassifiziert[17]. Alle anderen Typ-Begriffsinstanzen bekommen den internen Typ *none*. Die Namen ergeben sich unmittelbar aus den internen Basistypen.

Abbildung 5.18 zeigt die initiale Typliste des Beispielmodells. Man erkennt die Typen der Endbegriffe (z.B. class CEngineManagement), die Konstanten und provisorischen Typen. Typen mit dem internen Basistypen *none* sind durch die dunkle Einfärbung leicht zu erkennen (*Engine*, *Running*).

Auswertung der Beziehungen

Nachdem die initiale Typliste erstellt wurde, werden die Beziehungen der Typ-Begriffsinstanzen untereinander ausgewertet. Die Auswertung wird solange fortgesetzt, bis es keine Änderungen an der Typliste mehr gibt.

[17] Strukturen lassen sich nicht vermeiden, wenn eine existierende Außenwelt eingebunden werden muß. An einigen Stellen läßt sich (trotz aller Bemühungen um Allgemeinheit) nicht verbergen, daß als Sprache C++ und als Komponentensystem COM verwendet wurde.

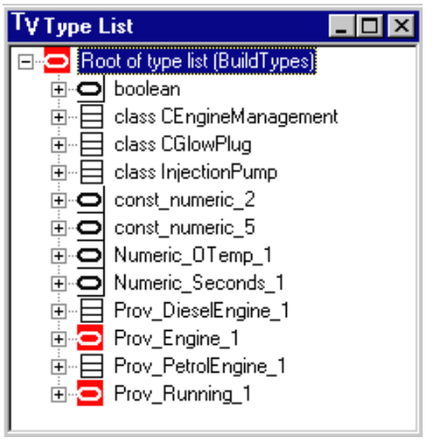


Abbildung 5.18.: Die initiale Typliste des Beispielmodells.

Für alle Typ-Begriffsinstanzen der Begriffsliste werden alle Beziehungen über die jeweiligen Matrizen ausgewertet. Für jeweils zwei betrachtete Typ-Begriffsinstanzen (genannt A und B) wird eine Aussage über die beteiligen Typen ($type_A$, $type_B$) getroffen.

Die ausgewerteten Beziehungen, die im folgenden näher erläutert werden, sind:

1. Identitäten
2. Alternativen
3. Splitteralternativen
4. Kombinationen von Alternativen
5. Mathematische Operationen

Zu (1), Identitäten:

Es hat sich gezeigt, daß die Identität zweier Typ-Begriffsinstanzen für die Klassifikation sehr wertvoll ist. Die Tabelle 5.7 zeigt die Matrix für die Behandlung von Identitäten. Die Matrix ist symmetrisch, da Identitäten keine Richtung haben. Es soll hier nicht die ganze Tabelle detailliert behandelt werden, sondern anhand von einigen Beispielen das Prinzip erläutert werden.

Nehmen wir zunächst die Zeile mit $type_B = none$, denn sie enthält viele Aktionen. Man erkennt sofort die logische Tatsache, daß keine Aussage über B getroffen werden kann, wenn auch A keinen Typ ($type_A = none$) hat. Hat A allerdings einen Typ, wird dieser Typ in den meisten Fällen auf B übertragen. Aus $type_A = string$ folgt die Aktion $string(B)$, somit $type_B = string$. Das selbe gilt z.B. für $type_A = uniqueId$.

Sind die beteiligten Typen Klassen ($type = class$) so stehen in der Matrix häufig die Aktionen *V(A)*, *V(B)* oder *V(A)(B)*. Sie bedeuten, daß sich die beiden beteiligten Typen nicht widersprechen, aber ein weiterer Vergleich beider Valuetypen (bei *V(A)(B)*), des Valuetyps von A (bei *V(A)*) mit dem Typ $type_B$ oder des Valuetyps von B (bei *V(B)*) mit dem Typ $type_A$ notwendig ist.

Das Aufeinandertreffen zweier numerischer Typen ist erlaubt, erfordert aber einen Abgleich ihrer Deklarationen durch die Abbildung *SizeCheck*.

Zu (2), Alternativen:

(↓ B) type (A →)	numeric	const numeric	string	const string	bool	const bool	byte	const byte	uniqueID	const uniqueID
numeric	SizeCheck	ok	error	error	error	error	error	error	error	error
const numeric	ok	error	error	error	error	error	error	error	error	error
string	error	error	ok	ok	error	error	error	error	error	error
const string	error	error	ok	error	error	error	error	error	error	error
bool	error	error	error	error	ok	ok	error	error	error	error
const bool	error	error	error	error	ok	error	error	error	error	error
byte	error	error	error	error	error	error	ok	ok	error	error
const byte	error	error	error	error	error	error	ok	error	error	error
uniqueID	error	error	error	error	error	error	error	error	ok	ok
const uniqueID	error	error	error	error	error	error	error	error	ok	error
enum	error	error	error	error	error	error	error	error	error	error
enumElement	error	error	error	error	error	error	error	error	error	error
struct	error	error	error	error	error	error	error	error	error	error
void	error	error	error	error	error	error	error	error	error	error
none	numeric (B)	numeric (B)	string (B)	string (B)	bool (B)	bool (B)	byte (B)	byte (B)	uniqueID (B)	uniqueID (B)
const none	numeric (B)	numeric (B)	string (B)	string (B)	bool (B)	bool (B)	byte (B)	byte (B)	uniqueID (B)	uniqueID (B)
class	V(B)	error	V(B)	error	V(B)	error	V(B)	error	V(B)	error
component	error	error	error	error	error	error	error	error	error	error
interface	error	error	error	error	error	error	error	error	error	error
reference	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok

(↓ B) type (A →)	enum	enum Element	struct	void	none	const none	class	component	interface	reference
numeric	error	error	error	error	numeric (A)	numeric (A)	V(A)	error	class (B)	ok
const numeric	error	error	error	error	numeric (A)	numeric (A)	V(A)	error	error	ok
string	error	error	error	error	string (A)	string (A)	V(A)	error	class (B)	ok
const string	error	error	error	error	string (A)	string (A)	V(A)	error	error	ok
bool	error	error	error	error	bool (A)	bool (A)	V(A)	error	class (B)	ok
const bool	error	error	error	error	bool (A)	bool (A)	V(A)	error	error	ok
byte	error	error	error	error	byte (A)	byte (A)	V(A)	error	class (B)	ok
const byte	error	error	error	error	byte (A)	byte (A)	V(A)	error	error	ok
uniqueID	error	error	error	error	uniqueID (A)	uniqueID (A)	V(A)	error	class (B)	ok
const uniqueID	error	error	error	error	uniqueID (A)	uniqueID (A)	V(A)	error	error	ok
enum	Merge	ok	error	error	enum (A)	enumEl (A)	V(A)	error	class (B)	ok
enumElement	ok	error	error	error	enum (A)	enumEl (A)	V(A)	error	error	ok
struct	error	error	Merge	error	struct (A)	error	error	error	error	ok
void	error	error	error	ok	ref (A)	error	nop	nop	nop	ok
none	enum (B)	enum (B)	struct (B)	ref (B)	nop	nop	V(A)	component (B)	ref (B)	ref (B)
const none	enumEl (B)	enumEl (B)	error	error	nop	nop	V(A)	NullOnly	NullOnly	nop
class	V(B)	error	error	nop	V(B)	V(B)	V(A)(B)	error	ref (B)	ok
component	error	error	error	nop	component (A)	NullOnly	error	ok	error	interface (A)
interface	error	error	error	nop	ref (A)	NullOnly	error	error	CheckIf	ok
reference	ok	ok	ok	ok	ref (A)	nop	ok	interface (B)	ok	ok

Tabelle 5.7.: Aktionsmatrix für Identitäten.

(↓ B) type (A →)	numeric	const numeric	string	const string	bool	const bool	byte	const byte	uniqueID	const uniqueID
numeric	SizeCheck	error	error	error	error	error	error	error	error	error
const numeric	SizeCheck	error	error	error	error	error	error	error	error	error
string	error	error	ok	error	error	error	error	error	error	error
const string	error	error	ok	error	error	error	error	error	error	error
bool	error	error	error	error	ok	error	error	error	error	error
const bool	error	error	error	error	ok	error	error	error	error	error
byte	error	error	error	error	error	error	ok	error	error	error
const byte	error	error	error	error	error	error	ok	error	error	error
uniqueID	error	error	error	error	error	error	error	error	ok	error
const uniqueID	error	error	error	error	error	error	error	error	ok	error
enum	error	error	error	error	error	error	error	error	error	error
enumElement	error	error	error	error	error	error	error	error	error	error
struct	error	error	error	error	error	error	error	error	error	error
void	error	error	error	error	error	error	error	error	error	error
none	numeric (B)	error	string (B)	error	bool (B)	error	byte (B)	error	uniqueID (B)	error
const none	numeric (B)	error	string (B)	error	bool (B)	error	byte (B)	error	uniqueID (B)	error
class	V(B)	error	V(B)	error	V(B)	error	V(B)	error	V(B)	error
component	error	error	error	error	error	error	error	error	error	error
interface	error	error	error	error	error	error	error	error	error	error
reference	error	error	error	error	error	error	error	error	error	error

(↓ B) type (A →)	enum	enum Element	struct	void	none	const none	class	component	interface	reference
numeric	error	error	error	ok	numeric (A)	error	V(A)	error	error	ok
const numeric	error	error	error	error	numeric (A)	error	V(A)	error	error	error
string	error	error	error	ok	string (A)	error	V(A)	error	error	ok
const string	error	error	error	error	string (A)	error	V(A)	error	error	error
bool	error	error	error	ok	bool (A)	error	V(A)	error	error	ok
const bool	error	error	error	error	bool (A)	error	V(A)	error	error	error
byte	error	error	error	ok	byte (A)	error	V(A)	error	error	ok
const byte	error	error	error	error	byte (A)	error	V(A)	error	error	error
uniqueID	error	error	error	ok	uniqueID (A)	error	V(A)	error	error	ok
const uniqueID	error	error	error	error	uniqueID (A)	error	V(A)	error	error	error
enum	error	error	error	ok	error	error	error	error	error	ok
enumElement	ok	error	error	error	enum (A)	error	V(A)	error	error	error
struct	error	error	error	ok	error	error	error	error	error	ok
void	error	error	error	ok	nop	error	V(A)	error	error	ok
none	error	error	error	nop	nop	error	V(A)	error	class (B)	nop
const none	enumEl (B)	error	error	nop	nop	error	V(A)	error	error	error
class	V(B)	error	error	ok	ref (A)	error	ok	error	ok	ok
component	error	error	error	ok	interface (A)	error	error	error	ok	interface (A)
interface	error	error	error	ok	ref (A)	error	error	error	CheckIf	ok
reference	error	error	error	error	error	error	error	error	error	error

Tabelle 5.8.: Aktionsmatrix für Alternativen.

(↓ B) (A →)	none	class	component	interface	reference
class	ref (A)	ok	error	ok	ok
component	interface (A)	error	ok	ok	interface (A)
interface	ref (A)	ok	ok	CheckIf	ok
reference	error	error	error	error	error

Tabelle 5.9.: Aktionsmatrix für Splitteralternativen. Alle fehlenden Teile enthalten die Aktion *error*.

Unter Alternativen-Beziehung wird verstanden, daß ein Begriff A eine Alternative in Form eines Begriffes B hat (5.19). Die Matrix für Alternativen-Beziehungen (siehe Tabelle 5.8) ist

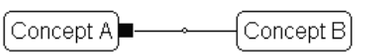


Abbildung 5.19.: *Concept A* besitzt die Alternative *Concept B*.

nicht symmetrisch, denn Alternativen haben im Gegensatz zu Identitäten eine Richtung. Man erkennt in der Matrix sehr schön, daß der Begriff A keine Konstante sein darf (z.B. Spalte *const numeric*), denn eine Konstante hat per Definition ein festen, unveränderbaren Wert. Der Oberbegriff (Begriff A) und seine Alternativen müssen in der Regel auch denselben Grundtyp haben. Die Typen *numeric* und *const numeric* haben z.B. denselben Grundtyp. Wie bereits von den Identitäten bekannt, wird für eine Klasse die Prüfung mit dem Valuetyp der Klasse wiederholt.

Zu (3), Splitteralternativen

Die Matrix für Splitter-Alternativen-Beziehungen ist in Tabelle 5.9 abgebildet. Dabei wurden zur Steigerung der Übersicht alle Zeilen und Spalten weggelassen, die ausschließlich die Aktion *error* enthalten. Dies betrifft alle konstanten Typen (z.B. *const numeric*) und die Typen *component*, *enum* und *struct*. Splitteralternativen entstehen aus aufgelösten Alternativen an Begriffen mit Methodenaufrufen. Methoden gehören aber immer zu Klassen. Es bleiben deshalb nur Typen in der Matrix, die direkt (z.B. Interface) oder indirekt (Value) Klassen betreffen. Es ist deswegen nicht erstaunlich, daß die einzigen Aktionen in dieser Matrix *class(A)*, *class(B)* und *ok* sind.

Zu (4), Kombinationen von Alternativen

Unter den Kombinationen von Alternativen wird das in Abbildung 5.20 dargestellte Konstrukt verstanden. Die Begriffe *Concept A* und *Concept B* sind beide Alternativen eines weiteren Begriffs *Concept C*. Betrachtet werden hier alle Kombinationen von Alternativen untereinander. Tabelle 5.10 stellt die passende Matrix dar. Sie ist wiederum symmetrisch, denn auf den Al-

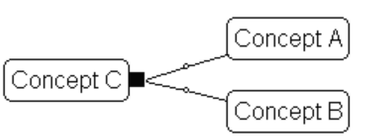


Abbildung 5.20.: Kombination der Alternativen *Concept A* und *Concept B* an einem Begriff *Concept C*.

ternativen untereinander ist keine Reihenfolge definiert. Die grundsätzliche Regel lautet: alle Alternativen an einem Begriff müssen denselben Typ haben. Dadurch wird auch verständlich, daß gilt: hat eine Alternative einen Typ, so wird er fast immer auf eine Alternative ohne Typ

(↓ B) type (A →)	numeric	const numeric	string	const string	bool	const bool	byte	const byte	uniqueID	const uniqueID
numeric	SizeCheck	SizeCheck	error	error	error	error	error	error	error	error
const numeric	SizeCheck	ok	error	error	error	error	error	error	error	error
string	error	error	ok	ok	error	error	error	error	error	error
const string	error	error	ok	ok	error	error	error	error	error	error
bool	error	error	error	error	ok	ok	error	error	error	error
const bool	error	error	error	error	ok	ok	error	error	error	error
byte	error	error	error	error	error	error	ok	ok	error	error
const byte	error	error	error	error	error	error	ok	ok	error	error
uniqueID	error	error	error	error	error	error	error	error	ok	ok
const uniqueID	error	error	error	error	error	error	error	error	ok	ok
enum	error	error	error	error	error	error	error	error	error	error
enumElement	error	error	error	error	error	error	error	error	error	error
struct	error	error	error	error	error	error	error	error	error	error
void	error	error	error	error	error	error	error	error	error	error
none	numeric (B)	numeric (B)	string (B)	string (B)	bool (B)	bool (B)	byte (B)	byte (B)	uniqueID (B)	uniqueID (B)
const none	numeric (B)	numeric (B)	string (B)	string (B)	bool (B)	bool (B)	byte (B)	byte (B)	uniqueID (B)	uniqueID (B)
class	V(B)	V(B)	V(B)	V(B)	V(B)	V(B)	V(B)	V(B)	V(B)	V(B)
component	error	error	error	error	error	error	error	error	error	error
interface	error	error	error	error	error	error	error	error	error	error
reference	error	error	error	error	error	error	error	error	error	error

(↓ B) type (A →)	enum	enum Element	struct	void	none	const none	class	component	interface	reference
numeric	error	error	error	error	numeric (A)	numeric (A)	V(A)	error	error	error
const numeric	error	error	error	error	numeric (A)	numeric (A)	V(A)	error	error	error
string	error	error	error	error	string (A)	string (A)	V(A)	error	error	error
const string	error	error	error	error	string (A)	string (A)	V(A)	error	error	error
bool	error	error	error	error	bool (A)	bool (A)	V(A)	error	error	error
const bool	error	error	error	error	bool (A)	bool (A)	V(A)	error	error	error
byte	error	error	error	error	byte (A)	byte (A)	V(A)	error	error	error
const byte	error	error	error	error	byte (A)	byte (A)	V(A)	error	error	error
uniqueID	error	error	error	error	uniqueID (A)	uniqueID (A)	V(A)	error	error	error
const uniqueID	error	error	error	error	uniqueID (A)	uniqueID (A)	V(A)	error	error	error
enum	Merge	error	error	error	enum (A)	error	V(A)	error	error	error
enumElement	error	ok	error	error	error	enumEl (A)	error	error	error	error
struct	error	error	Merge	error	struct (A)	error	error	error	error	error
void	error	error	error	ok	nop	error	V(A)	error	error	error
none	enum (B)	error	struct (B)	nop	nop	nop	V(A)	nop	error	error
const none	error	enumEl (B)	error	error	nop	nop	V(A)	error	error	error
class	V(B)	error	error	V(B)	V(B)	V(B)	ok	ok	error	error
component	error	error	error	error	nop	error	ok	ok	error	error
interface	error	error	error	error	nop	error	error	error	nop	error
reference	error	error	error	error	error	error	error	error	error	error

Tabelle 5.10.: Aktionsmatrix für Kombinationen von Alternativen.

übertragen (siehe z.B. Zeile mit $type_B = none$). Bei numerischen Werten wird wie immer ein Abgleich durchgeführt (*SizeCheck*).

Zu (5), mathematische Operationen

Das Ergebnis mathematischer Operationen kann nur drei Typen annehmen: *boolean* (z.B. Vergleiche), *numeric* (z.B. +) oder *string* (z.B. $+). Es werden daher drei Gruppen gebildet und jede mathematische Operationen wird einer dieser Gruppen zugeordnet.

Jeder Begriff ohne Typ (d.h. interner Typ *none*), der das Resultat einer mathematischen Operation aufnimmt, bekommt den Typ der Operation (z.B. *numeric* bei '+'). Begriffe mit Typ dürfen dem Resultattyp nicht widersprechen. Treffen zwei numerische Werte aufeinander, wird die bereits vorgestellte Aktion *SizeCheck* ausgeführt.

Die direkte Zuweisung (Assign) ist die einfachste Form der mathematischen Operationen. Falls einer der beiden Partner einen Typ hat, wird er vom anderen Partner übernommen (der Typ-Zusatz *const* geht dabei verloren).

Bisher können nur für Konfigurationsbegriffe Constraints vergeben werden. Es ist denkbar, für alle Begriffe Constraints einzuführen. Es wäre dann möglich, bei mathematischen Operationen auch die Mantisse und Exponenten entsprechend der Operation anzupassen.

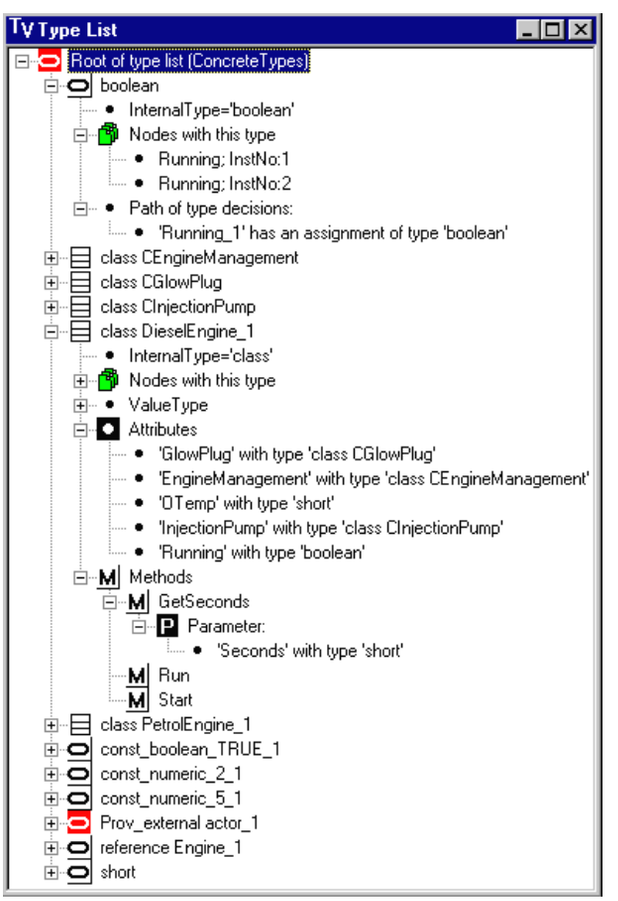


Abbildung 5.21.: Die Typliste des Beispiels nach der Auswertung aller Beziehungen.

Sobald die Typliste sich nicht mehr ändert, sollte kein Typ mehr den internen Typ *none* haben.

Zusammenfassung der Typen

Nach dem Ende der Auswertung gibt es viele provisorische Typen. Sie werden nun umgewandelt in konkrete Typen.

Numerische Typen (auch Konstante) erhalten aufgrund ihrer Deklaration (minimale und maximale Mantissen und Exponenten) einen konkreten Typ. Um den Speicherplatzbedarf zu optimieren, wird der kleinste mögliche Typ genommen, der sich aus dem minimalen Exponenten und Mantisse ergibt.

Provisorische Typnamen werden durch konkrete Typnamen ersetzt. Provisorische Typen, die durch die Klassifikation zu einem einfachen Typ (z.B. string) geworden sind, werden durch den einfachen Typ ersetzt. Bei provisorischen Typen, die zu neuen Klassen oder Aufzählungen geworden sind, wird Prefix 'Prov_' und die Instanznummer entfernt und der Typ vorangestellt (aus *Prov_Dieselengine_1* wird *class Dieselengine*). Gibt es danach namensgleiche Typen, die nicht identisch sind, wird die Instanznummer für den konkreten Typ beibehalten. Die Identität von Klassen nachzuweisen ist nicht trivial. Es reicht nicht, daß sie dieselben Attribute und Methoden haben, denn die Implementierung der Methoden kann dann immer noch unterschiedlich sein (es können zwei verschiedene Sheets dafür existieren). Es wird daher hier auf eine Zusammenfassung von namensgleichen Klassen verzichtet.

In Abbildung 5.21 ist die Typliste des Beispieldesigns nach der Auswertung aller Beziehungen und der Umbenennung der Typen dargestellt. Man sieht, daß es dort keine unklassifizierten Typen mehr gibt (abgesehen vom Typ des externen Aktors, der nur aus technischen Gründen existiert).

5.4. Die Komponentenbestimmung

Nachdem alle Begriffe bzw. Begriffsinstanzen klassifiziert sind und einen konkreten Typ besitzen, kann die *Bestimmung der Komponenten* erfolgen.

Es geht bei der Komponentenbestimmung darum, eine optimale Anzahl an Komponenten zu finden und die Verteilung der Objekte auf die einzelnen Komponenten zu bestimmen. Objekte sind dabei alle diejenigen Begriffsinstanzen, die als Klasse identifiziert wurden.

Zunächst werden allgemeine Regeln erläutert, die der Entscheidungsfindung zugrunde liegen. Die Regeln führen dann zu einem Kostenmodell und einer Kostenfunktion. Die Kostenfunktion wird anschließend optimiert.

5.4.1. Bildung von Interfaces

Die Interfaces von Komponenten werden auf zwei Arten gebildet. Auf der einen Seite können sie im Design fest vorgegeben werden (siehe Kapitel 4). Diese Interfaces werden ohne Änderung mit all ihren Methoden in die Komponente übernommen.

Auf der anderen Seite entstehen neue, vom System generierte Interfaces. Jedes Objekt gehört zu einer Komponente und damit gehört auch jede Methode zu einer Komponente. Eine Methode, die außerhalb ihrer Komponente aufgerufen wird, wird zu einer Methode an einem Interface. Die Namen dieser Interfaces werden aus der Rolle der externen Aktoren gebildet. Zur Erinnerung: eine Rolle wird definiert durch einen *Benutzer* (user) und eine *Aufgabe* (task). Jede Methode 'erbt' die Rolle des externen Aktors unter der sie aufgerufen wird.

Der Name des Interfaces ergibt sich für eine Rolle $r = (user, task)$ zu:

$$name_{if} =< task > \_ < user >$$

Alle Methoden, die diese Rolle haben, werden Teil dieses Interfaces. Methoden mit mehreren Rollen werden Teil jedes durch die Rollen definierten Interfaces.

5.4.2. Allgemeine Kriterien

Die Bestimmung der optimalen Aufteilung der Begriffsinstanzen auf Komponenten stützt sich auf Kriterien, die nun als Regeln formuliert werden sollen. Die Regeln werden hier zunächst informell vorgestellt und begründet. Sie werden später benutzt, um verschiedene Kostenarten abzuleiten.

Die hier vorgestellten Kriterien sind nicht unbedingt neu. Es werden zum Teil gängige Heuristiken für objekt-orientiertes Design verwendet bzw. angepaßt und auf Komponenten übertragen.

Regel 31 *Jedes Programm ist als ganzes und dem Benutzer gegenüber eine Komponente.*

Das Ziel des Verfahrens ist es, ein optimales Komponentendesign zu bestimmen. Damit muß es zumindest eine Komponente geben, die sich dem Benutzer (auch im Sinne eines anderen Systems) präsentiert. Dabei ist es egal, wie groß diese Komponente ist und wie sie sich zusammensetzt.

Regel 32 *Alle Nachrichten vom Benutzer an das System sind Aufrufe von Interfaces.*

Diese Regel folgt automatisch, denn das System präsentiert sich dem Benutzer gegenüber als Komponente (siehe Regel 31). Komponenten bieten ihre Dienste per Definition nur über ihre Interfaces an. Somit werden alle Methoden, die von außen aufgerufen werden, zu Methoden an Interfaces.

Regel 33 *Eine Komponente darf nicht zu viele Interfaces implementieren (ca. 10).*

Dies ist keine technische Randbedingung, sondern dient der Übersichtlichkeit bei der Verwendung einer Komponente. Psychologische Untersuchungen zeigen, daß Menschen im Durchschnitt ca. 7 Dinge im Kurzzeitgedächtnis behalten können [Mil56]. Weil diese Heuristik in der Softwareentwicklung häufig verwendet wird und es keine besseren Erkenntnisse gibt, wird sie auch hier eingesetzt. Wenn man annimmt, daß einige Interfaces fast immer vorhanden sein müssen (z.B. IUnknown, IPersitStream in COM), so sind ca. 10 Interfaces pro Komponente noch vertretbar.

Regel 34 *Eine Methode eines Interfaces darf nicht von zwei Objekten in der Komponente realisiert werden.*

Abbildung 5.22 zeigt die Entstehung eines solchen Falles. Sowohl Komponente A als auch Komponente B besitzen ein Interface *Interface1* mit der Methode *m1*. Die Methode *m1* wird in Komponente A durch das Objekt A implementiert (gestrichelte Linie), in Komponente B von Objekt B. Wenn beide Komponenten zur Komponente AB zusammengefaßt würden, so ist nicht entscheidbar, wer die Methode nun tatsächlich implementiert bzw. ausführt, wenn sie am Interface aufgerufen wird. Diese Konstellation ist deswegen nicht erlaubt.

Eine andere Situation ergibt sich, wenn es zwei namensgleiche Methoden an unterschiedlichen Objekten gibt und nicht beide Methoden über dasselbe Interface nach außen angeboten werden müssen. Diese Situation stellt kein Problem dar (Abbildung 5.23).

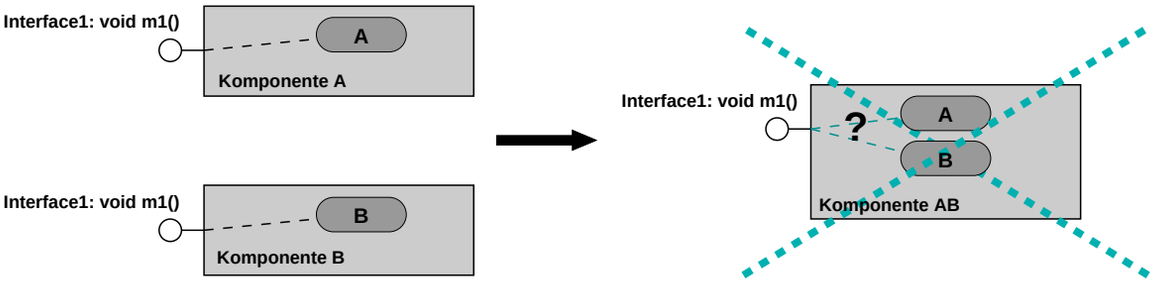


Abbildung 5.22.: Komponente A und B haben dasselbe Interface *Interface1*. Ein Methode des Interfaces wird aber in beiden Komponenten von unterschiedlichen Objekten realisiert. Eine Zusammenfassung der beiden Komponenten ist daher nicht möglich.

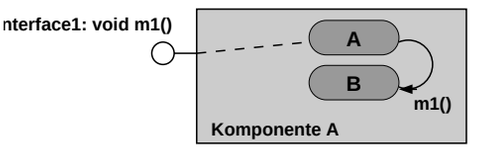


Abbildung 5.23.: Komponente A enthält zwei Objekte A und B mit jeweils der Methode m1. Diese Konstellation ist erlaubt, wenn nicht beide Methoden am selben Interface veröffentlicht werden.

Regel 35 *Eine Komponente soll nicht zwei namensgleiche Interfaces besitzen.*

Diese Regel ist nur eine 'Soll'-Regel, denn es gibt eine technische Lösung für dieses Problem ('Tear-Off-Interfaces'). Allerdings ist diese Lösung mit Aufwand verbunden und sollte vermieden werden.

Regel 36 *Ein Interface an einer Komponente muß von ihr vollständig implementiert werden.*

Ein Interface besteht i.d.R. aus mehreren Methoden. Jede Methode kann von einem anderen Objekt implementiert werden. Eine Komponente muß daher genügend Objekte besitzen, um alle Methoden eines Interfaces zu implementieren. Es dürfen keine Methoden am Interface angeboten werden, die keine Implementierung besitzen.

Regel 37 *Die Zahl der Aufrufe zwischen Komponenten sollte minimiert werden.*

Analog zur Heuristik '*Minimize the number of message sends between a class and its collaborator*' (4.3 in [Rie96]), übertragen auf Komponenten. Bei Komponenten sind die negativen Auswirkungen noch größer als bei Klassen, denn Komponenten können auf verteilten Rechnern laufen.

Regel 38 *Die Menge der zwischen Komponenten ausgetauschten Daten sollte minimiert werden.*

Diese Regel läßt sich sehr ähnlich zur letzten Regel begründen. Die Daten, die zwischen Komponenten ausgetauscht werden, müssen meist über Prozeß- und/oder über Rechnergrenzen transportiert werden. Dazu müssen sie zum Transport bei Sender eingepackt und beim Empfänger wieder ausgepackt werden (Marshaling). Hinzu kommt die Übertragungsdauer über ein Netzwerk.

Regel 39 *Attribute einer Klasse, die selbst Klassen sind, werden sofort in dieselbe Komponente gezogen.*

Es ist aus Performancegründen nicht sinnvoll, Attribute einer Begriffsinstanz, die selbst vom Typ Klasse sind, auf andere Komponenten zu verteilen, auch wenn dies theoretisch möglich ist. Es werden deswegen, und um die initiale Anzahl von Komponenten zu verringern, alle Begriffsinstanzen, die Attribute einer anderen Begriffsinstanz sind, in deren Komponente hineingezogen. Eine Analogie dazu findet sich in Heuristik 7.1 in [Rie96]: '*When given a choice [...] between a containment and an aggregation, choose the containment relationship.*'.

Regel 40 *Eine Komponente darf nicht zu viele Konfigurationsdaten besitzen (ca. 30).*

Regel 41 *Eine Komponente sollte Konfigurationsdaten besitzen.*

Die Anzahl der Konfigurationsdaten sollte 30 nicht überschreiten. Diese obere Schranke hat sich in der Praxis bewährt. Mehr Konfigurationsdaten machen den Konfigurationsvorgang unübersichtlich (z.B. bei der Darstellung in einem Konfigurationstool). Jede Komponente sollte allerdings mindestens ein Konfigurationsdatum besitzen, denn reine Dienstleistungskomponenten sind nicht förderlich.

Regel 42 *Eine Komponente, die eine 1:N Beziehung zu einer Komponente B hat, sollte nicht mit einer Komponente B zusammengefaßt werden, wenn Komponente B Konfigurationsdaten hat.*

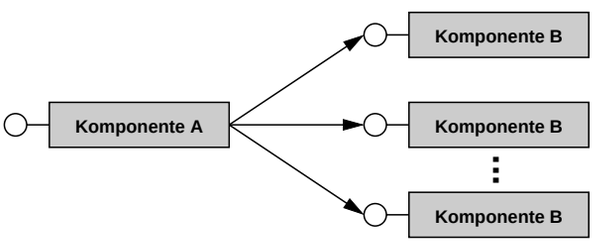


Abbildung 5.24.: Eine 1:N Beziehung zwischen der Komponente A und der Komponente B.

Eine 1:N Beziehung liegt vor, wenn eine Komponente A zu einer unbekannten Anzahl von Instanzen von Komponente B eine Beziehung hat (siehe Abbildung 5.24). Eine Zusammenfassung von Komponente A und B bietet sich so eigentlich an. Besitzt Komponente B allerdings Konfigurationsdaten, so bringt eine Zusammenfassung Probleme beim konfigurieren. In der gemeinsamen Komponente existiert dann für jede Instanz von Komponente B ein eigener Satz von Konfigurationsdaten (Abbildung 5.25). Eine solche Konstellation läßt sich sehr schwer konfigurieren, denn die genaue Anzahl der Instanzen von Komponente B entsteht dynamisch. Diese Situation sollte deswegen vermieden werden.

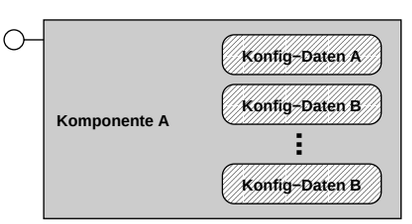


Abbildung 5.25.: Konfigurationsdaten nach Zusammenfassung zweier Komponenten mit 1:N Beziehung.

Die Regeln zur Komponentenbestimmung müssen nicht immer genau eingehalten werden. Durch die Abbildung der Regeln in Kosten und anschließender Optimierung der Kosten, ist es durchaus erlaubt, eine Regel nicht strikt zu befolgen, wenn die Kosten dadurch insgesamt sinken.

5.4.3. Kosten und Kostenfunktion

Zur Optimierung des Systems werden bestimmte Eigenschaften des Systems mit Kosten belegt. Die genauen Werte der einzelnen Kosten sind zunächst nicht wichtig. Im Kapitel 6.2.2 werden zwei unterschiedliche Kostensätze untersucht.

Die Kosten für das System (auch *Gesamtkosten* genannt) setzten sich zusammen aus den *Komponentenkosten* und den *Verbindungskosten*.

Die Komponentenkosten sind diejenigen Kosten, die durch die Existenz einer (speziellen) Komponente entstehen. Sie setzten sich aus mehreren einzelnen Kosten zusammen:

- den Grundkosten (Basecosts) zur Verwaltung der Komponente,
- den Klassenkosten,
- den Interfacekosten und
- den Konfigurationskosten.

Die Verbindungskosten entstehen durch das Zusammenspiel der Komponenten als System. Sie umfassen die Kosten für den Datenaustausch zwischen den Komponenten, sie werden bestimmt durch:

- die Anzahl der Methodenaufrufe,
- die Anzahl der übertragenen Parameter und
- die Art und Größe der übertragenen Parameter.

Alle hier angesprochenen Kostenarten werden in den folgenden Abschnitten genauer vorgestellt. Ein Überblick über die Kostenarten, ihre Zusammenhänge und ihre Einflußgrößen sind in Abbildung 5.26 dargestellt.

Grundkosten einer Komponente

Eine Komponente stellt für das System einen gewissen Aufwand dar. Eine Komponente muß z.B. gefunden und erzeugt werden, die Komponenten müssen sich untereinander kennenlernen. Dieser Aufwand wird mit den Grundkosten einer Komponente berücksichtigt. Die Grundkosten sind für jede Komponente gleich und werden mit $compBaseCost$ bezeichnet.

Klassenkosten

Eine Komponente besteht aus mindestens einem Objekt, also mindestens einer Instanz einer Klasse. Alle Instanzen einer Klasse teilen sich den Codebereich, d.h. die Implementierung der Methoden. Der Speicherplatzbedarf für ein Objekt kann daher unterteilt werden in einen statischen und dynamischen Teil. Der statische Teil fällt für jede verwendete Klasse unabhängig von der Anzahl der Instanzen in jeder Komponente genau einmal an. Der dynamische Teil fällt für jede Instanz erneut an und besteht aus den Attributen (abgesehen von statischen Attributen). Der dynamische Speicherverbrauch muß bei der Kostenberechnung nicht berücksichtigt

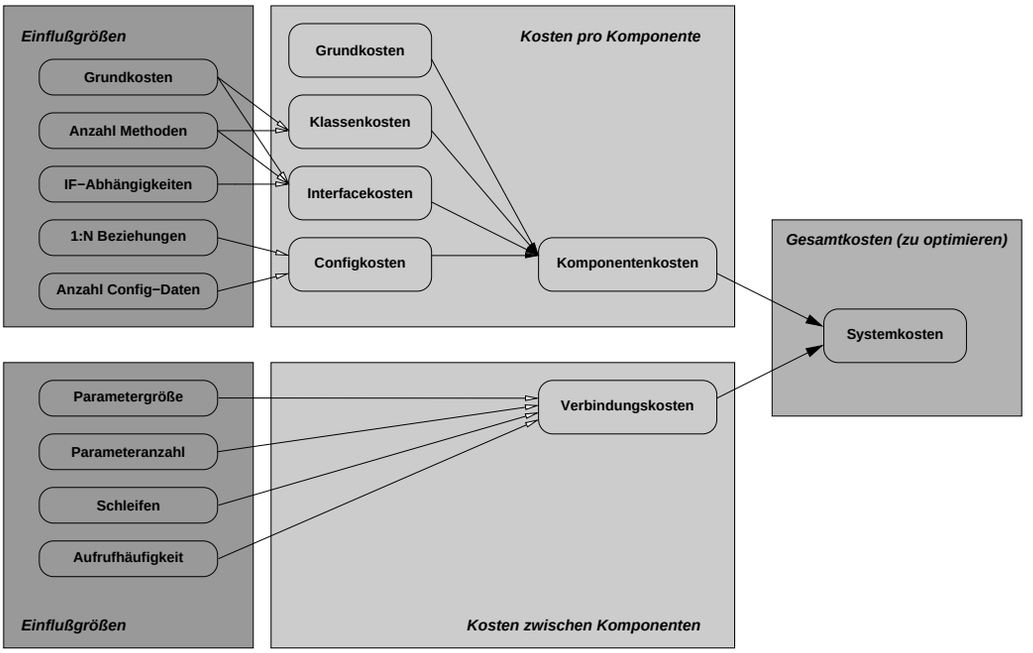


Abbildung 5.26.: Die unterschiedlichen Kostenarten, die bei der Berechnung eines Komponentendesigns verwendet werden und ihre Einflußgrößen.

zu werden. Die Anzahl der Instanzen im Gesamtsystem ist immer gleich, unabhängig davon, in welcher Komponente eine Instanz angesiedelt ist.

Der statische Speicherplatzbedarf einer Klasse wird bestimmt durch die Codegröße der Methoden. Da die konkrete Codegröße nicht verfügbar ist, wird als Näherung zur Kostenbestimmung nur die Anzahl der Methoden einer Klasse (hier mit n_{cl} bezeichnet) berücksichtigt. Jede Methode verursacht dieselben Kosten, $classMethodCost$. Zusätzlich wird für jede Klasse ein Konstruktor, ein Copykonstruktor und ein Destruktor berücksichtigt. Für jede Klasse werden zusätzlich Grundkosten angenommen, $classBaseCost$. Die Klassenkosten ergeben sich dann zu:

$$c_{cl} = classBaseCost + (n_{cl} + 3) * classMethodCost$$

Da Komponenten über ihre Interfaces angesprochen werden, ist es unerheblich wie sie im Inneren realisiert sind. Die Anzahl der Methoden einer Klasse, die innerhalb einer Komponente eingesetzt wird, sind damit nicht relevant. Die Einhaltung der Heuristik 4.7 in [Rie96]: '*Classes should not contain more objects than a developer can fit in his or her short-term memory*' wird an dieser Stelle absichtlich nicht erzwungen.

Die Klassenkosten können vor der Optimierungsrechnung statisch bestimmt werden.

Interfacekosten

Da alle Dienste von Komponenten nur über Interfaces angesprochen werden können, müssen Beziehungen zu Objekten außerhalb der eigenen Komponente immer über Interfaces abgewickelt werden. Die Interfacekosten repräsentieren diese Tatsache. Für jedes Interface einer Komponente gibt es Grundkosten ($ifBaseCosts$) und pro Methode des Interfaces Methodenkosten ($ifMethodCost$). Die Interfacekosten für ein Interface mit n Methoden ergeben sich so zu:

$$c_{if} = ifBaseCosts + n * ifMethodCost$$

Eine Komponente kann beliebig viele Interfaces besitzen. Um eine Komponente übersichtlich zu halten, ist eine Beschränkung auf 10 Interfaces zu empfehlen. Die Interfacekosten steigen deswegen ab dem 11. Interface. Für jedes weitere Interface gibt es einen Aufschlag in Höhe von $ifOverCost$. Falls es ein Interfaces mehrfach gibt, erfolgt eine Bestrafung durch Kosten in Höhe von $costSameIF$.

Die Interfacekosten für eine Komponente ergeben sich aus der gewichteten Summe der Kosten für jedes einzelne Interface, wobei $count_{if}$ die Anzahl der Interfaces angibt.

$$c_{if,comp} = \begin{cases} \sum\limits_{\forall if \in comp} c_{if}, & \text{falls} \quad count_{if} <= 10 \\ \sum\limits_{\forall if \in comp} c_{if} + (count_{if} - 10) * ifOverCost, & \text{falls} \quad count_{if} > 10 \end{cases}$$

Konfigurationskosten

Die Konfigurationskosten bewerten die Flexibilität einer Komponente und die Komplexität ihrer Handhabung. Nach Regel 41 wird eine Komponente ohne Konfigurationsdaten bestraft, d.h. mit Kosten ($costNoConfig$) belegt.

Mehr als 30 Konfigurationsdaten (Regel 40) machen die Konfiguration einer Komponente unübersichtlich. Die Kosten steigen deshalb danach stark linear. Die Konfigurationskosten für eine Komponente mit n Konfigurationsdaten ergeben sich zu:

$$c_{config,comp} = \begin{cases} costNoConfig, & \text{falls} \quad n = 0 \\ 0, & \text{falls} \quad 1 <= n <= 30 \\ (n - 30) * costOverConfig, & \text{falls} \quad n > 30 \end{cases}$$

Bei den Konfigurationskosten werden auch die *1:N Beziehungen* zwischen Komponenten berücksichtigt. Eine solche Konstellation wird mit sehr hohen, aber immer gleichen Kosten belegt ($cost1toN$).

1:N Beziehungen zwischen Begriffsinstanzen können in GCN Modellen nur durch Schleifen entstehen. Um diese Beziehung auszuwerten, werden die Begriffsinstanzen (Objekte) der Komponenten untersucht.

Die Informationen, ob zwischen zwei Begriffsinstanzen, die sich in zwei verschiedenen Komponenten befinden, eine 1:N Beziehung vorliegt, wird aus dem funktionalen Graphen gewonnen. Ein einfacher Fall ist in Abbildung 5.27 dargestellt.

Komponente A besteht aus einem Objekt, A1, und ruft zwei weitere Komponenten, Komponente B und Komponente C, auf. Komponente B besteht aus zwei Objekten B1 und B2 und besitzt Konfigurationsdaten. Komponente C besteht aus einem Objekt C1. Die Situation im funktionalen Graphen ist im rechten Teil der Abbildung 5.27 dargestellt[18].

Begriff A1 ruft sowohl den Begriff B1 als auch den Begriff C1 in einer Schleife auf. Trotzdem können die Komponenten A und C problemlos zusammengefaßt werden. Anders sieht es bei Komponente A und Komponente B aus. Wenn beide zusammengefaßt werden, so erhöhen sich die Konfigurationskosten der neuen Komponente um $cost1toN$:

[18] Die Begriffe A1, B1, B2 und C1 sind nach der Klassifikation zu Objekten geworden.

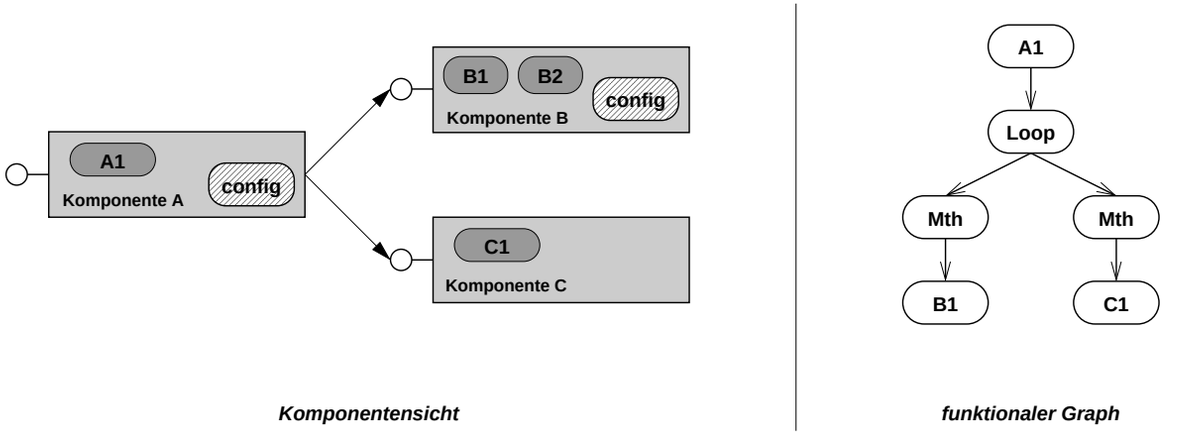


Abbildung 5.27.: Bestimmung einer 1:N Beziehung zwischen Komponenten. Links die Sicht in Komponentendarstellung, rechts die Lage im funktionalen Graphen.

$$c_{config,comp} = c_{config,comp} + cost1toN$$

Die Verhältnisse werden etwas komplexer, wenn sich ein Objekt selbst innerhalb einer Schleife aufruft (Objekt *A1* in Abbildung 5.28).

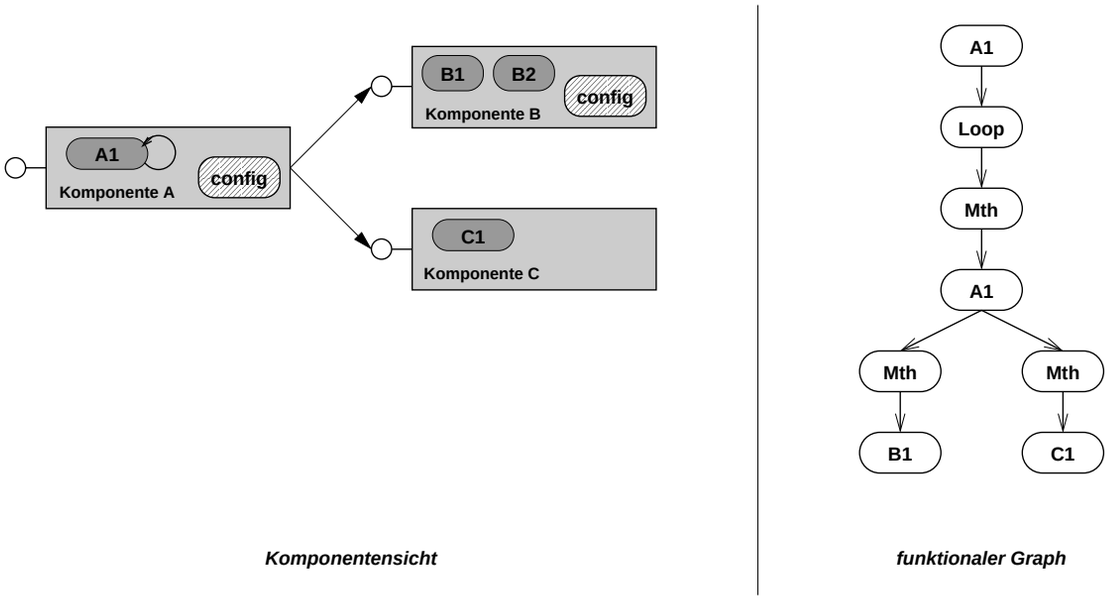


Abbildung 5.28.: Bestimmung einer komplexen 1:N Beziehung zwischen Komponenten. In der Schleife ruft sich ein Objekt selbst auf (*A1* in Komponente A). Die Betrachtung wird dann ausgeweitet auf die Objekte, die anschließend aufgerufen werden (hier *B1* und *C1*).

Das Objekt selbst gehört dann bereits zu seiner Komponente und die Konfigurationsdaten dieser Komponente sind durch die Schleife nicht betroffen. Allerdings können in der Schleife weitere Aufrufe an andere Objekte (und damit Komponenten) stattfinden. In Abbildung 5.28 sind dies die Objekte B1 und C1 (entsprechend Komponente B und Komponente C). Diese Kom-

Parametertyp	paramBaseCost
Klasse	classTypeCost
Struktur	structTypeCost
einfache Typen	simpleTypeCost
Interface	interfaceTypeCost
Strings	stringTypeCost

Tabelle 5.11.: Die unterschiedenen Parametertypen und die jeweils verwendeten Kosten.

ponenten müssen nun ebenfalls auf Konfigurationsdaten untersucht werden und evtl. bei einer Zusammenfassung mit den Kosten $cost1toN$ belegt werden.

Komponentenkosten

Es wurden nun alle Kostenarten besprochen, aus denen sich die Komponentenkosten zusammensetzen, also Grundkosten, Interfacekosten, Konfigurationskosten und Klassenkosten. Die Kosten für eine Komponente ergeben sich damit zu:

$$c_{comp} = compBaseCost + c_{if,comp} + c_{config,conn} + \sum_{\forall cl \in comp} c_{cl,comp}$$

Parameterkosten

Jede Methode einer Klasse besitzt eine feste Zahl an Parametern (Menge P). Bei jedem Aufruf der Methode müssen diese Parameter transportiert werden. Dadurch entstehen Kosten, die innerhalb einer Komponente vernachlässigt werden können, zwischen Komponenten aber sehr teuer sein können.

Die Parameterkosten werden durch den Typ des Parameters bestimmt. Je komplexer der Typ ist, desto höher sind die Kosten. Es wird unterschieden zwischen einfachen Typen (z.B. numerische Werte), Strukturen, Klassen, Interfaces und Strings. Die Werte für $paramBaseCost$ sind in Tabelle 5.11 zusammengefaßt. Es muß zusätzlich berücksichtigt werden, ob die Parameter Arrays sind. Die Parameterkosten $c_{p,mth}$ werden berechnet als:

$$c_{P,mth} = \sum_{\forall p \in P_{mth}} arrayIndexCount_p * paramBaseCost_p$$

mit

$$arrayIndexCount_p = \begin{cases} 1, & \text{falls Parameter kein Array ist} \\ n, & \text{falls Parameter ein Array der Größe } n \text{ ist} \\ 10, & \text{falls Parameter ein Array der variabler Größe} \end{cases}$$

Sie können ebenfalls vor der Optimierungsrechnung statisch für jede Methode bestimmt werden.

Verbindungskosten

Verbindungskosten entstehen durch die Kommunikation über Komponentengrenzen hinweg. Komponenten müssen sich nicht im selben Adressraum befinden, sondern können verteilt ausgeführt werden. Jede Kommunikation zwischen Komponenten führt so zu Kosten. Verbindungskosten werden immer zwischen genau zwei Komponenten berechnet. Die Verbindungskosten

zwischen Objekten innerhalb einer Komponente werden vernachlässigt (sie sind null). Die Menge aller Methoden, die zwischen beiden Komponenten A und B aufgerufen wird, wird mit $MC_{a,b}$ bezeichnet. Kosten zwischen zwei Komponenten A und B berechnen sich durch:

$$c_{conn,A,B} = \sum_{\forall mth \in MC_{a,b}} c_{P,mth} * callCount_m$$

$$\text{mit } callCount_m = \begin{cases} mc_{a,b}, & \text{falls Methodenaufruf nicht innerhalb Schleife} \\ loopAvgCount^{lc} * mc_{a,b}, & \text{falls Methodenaufruf innerhalb Schleife} \end{cases}$$

Dabei bedeutet die Zahl *loopAvgCount* die durchschnittlich angenommene Zahl von Schleifendurchläufen und *lc* die Anzahl der geschachtelten Schleifen.

Die Anzahl der Aufrufe für eine spezielle Methode wird aus den Referenzen der Methode (siehe Abschnitt 5.2) ermittelt. Die Information, ob der Aufruf innerhalb von (geschachtelten) Schleifen stattfindet, kann dem funktionalem Graphen entnommen werden.

Systemkosten

Die Systemkosten ergeben sich aus der Summe der Komponentenkosten und der Summe der Verbindungskosten zwischen den Komponenten:

$$c_{system} = \sum_{\forall comp} c_{comp} + \sum_{\forall conn} c_{conn}$$

5.4.4. Die Optimierung

Die Komplexität zur Bestimmung des optimalen Komponentendesigns mit n Komponenten ist eigentlich O($n!$), und damit nicht wirklich in vertretbarer Zeit zu berechnen. Der Aufwand zu Lösung des Problems kann aber deutlich reduziert werden, wenn nur diejenigen Kombinationen betrachtet werden, die überhaupt sinnvoll sind. Komponenten, die keine Beziehung zueinander haben, sich also nicht aufrufen, brauchen auch nicht zusammengefaßt werden. Diese Komponenten-Kombinationen brauchen deswegen für die Berechnung überhaupt nicht betrachtet werden. Der Berechnungsaufwand wird dadurch deutlich reduziert. Um einen Eindruck zu gewinnen, ist in Abbildung 5.29 eine Matrix gezeigt, die die realen Beziehungen zwischen Anfangskomponenten (Teil des Datenbankdesigns) darstellt. Die Matrix ist in der Regel sehr dünn besetzt. Jeder schwarze Punkt repräsentiert eine Verbindung zwischen zwei Komponenten. Da hier nicht die Richtung des Aufrufs unterschieden wird, ist die Matrix symmetrisch.

Ausgangssituation

Bevor mit der Optimierung begonnen werden kann, muß zunächst die Ausgangssituation hergestellt werden. Dazu wird aus jeder Begriffsinstanz jeweils eine eigene Komponente (Abbildung 5.30). Bereits im Design explizit vorgesehene Interfaces werden den einzelnen Komponenten zugeordnet. Diese Interfaces können nicht mehr verändert werden und müssen von der Komponente implementiert werden.

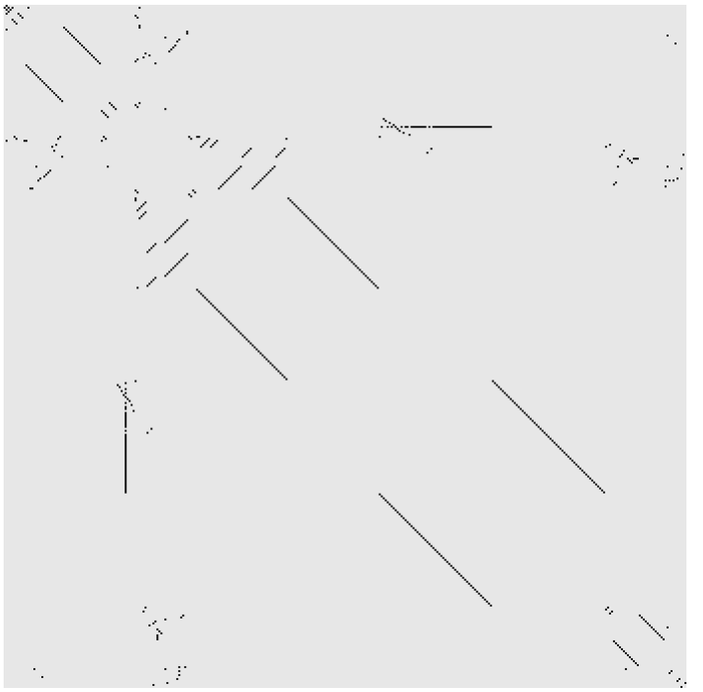


Abbildung 5.29.: Eine reale Beziehungsmatrix von ca. 350 Komponenten (Teil des Datenbankdesigns). Jeder schwarze Punkt repräsentiert eine Beziehung zwischen zwei Komponenten.

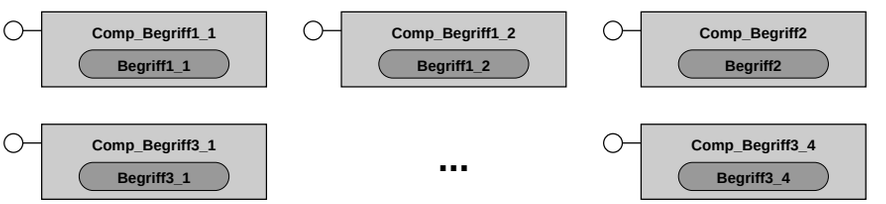


Abbildung 5.30.: Ausgangspunkt zur Komponentenbestimmung. Jedes Objekt wird zu einer eigenen Komponente.

Der Algorithmus

Trotz der schwachen Besetzung der Matrix ist die Komplexität aber immer noch zu groß, um alle Kombinationen zu berechnen. Es wird deswegen ein Greedy-Algorithmus verwendet [Sch97]. Die Idee dabei ist, in jedem Schritt genau zwei Komponenten zusammenzufassen. Es werden die beiden Komponenten zusammengefaßt, die die größte Reduktion der Gesamtkosten bringen. Die Komplexität des Problems wird dadurch von $O(n!)$ auf $O(n^3)$ gesenkt.

Die Reduktion der Kosten zwischen zwei Komponenten i und j wird berechnet durch:

$$red_{ij} = c_{comp,i} + c_{comp,j} + c_{conn,ij} - c_{comp,ij}$$

Dabei bedeuten:

- $red_{i,j}$ Reduktion der Kosten bei Zusammenfassung von Komponente i und Komponente j
- $c_{comp,i}$ die Kosten für Komponente i
- $c_{comp,j}$ die Kosten für Komponente j

- $c_{conn,ij}$ die Verbindungskosten zwischen Komponente i und Komponente j
- $c_{comp,ij}$ die Kosten für die zusammengefaßte Komponente

Um die Kosten für die zusammengefaßte Komponente zu bestimmen, werden beide Komponenten mit allen Konsequenzen virtuell zusammengefaßt und die Kosten für diese virtuelle Komponente bestimmt. Die Verbindungskosten zwischen beiden Komponenten sind nach der Zusammenfassung null. Sie erscheinen deswegen in der obigen Formel nicht mehr.

Die Namensgebung der zusammengefaßten Komponente ergibt sich nach der Regel:

Regel 43 *Die zusammengefaßte Komponente bekommt den Namen der aufrufenden Komponente.*

Der verwendete Algorithmus in seiner einfachen Form lautet in Pseudosprache:

Vorbereitungen:
1. jedes Objekt wird zu einer eigenen Komponente
2. fasse Attribute eines Objektes mit seiner Komponente zusammen

Hauptteil:
1. do {
2. for (alle sinnvollen Paare (i;j)) {
3. berechne $red_{i,j}$
4. bestimme red_{max}; merke Paar $(i_{max}; j_{max})$
5. }
6. if $(red_{max} > 0)$ {
7. fasse Komponente i_{max} und j_{max} zusammen
8. }
9. } while $(red_{max} > 0)$

Das Ende des Vorgangs ist dann erreicht, wenn eine Zusammenfassung von zwei Komponenten keine Verbesserung der Gesamtkosten mehr bringt.

Es werden bei der Komponentenfindung nicht nur Klassen, sondern Objekte betrachtet. Deswegen führt der bisher beschriebene einfache Ansatz generell zu mehr Komponenten als notwendig, denn ähnliche Komponenten werden nicht zusammengefaßt. Man stelle sich als Ausgangspunkt

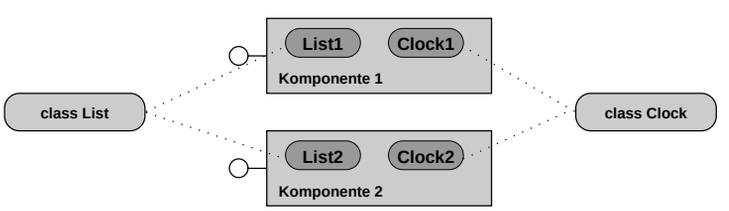


Abbildung 5.31.: Zwei Komponenten mit Objekten vom selben Typ.

z.B. die in Abbildung 5.31 dargestellt Situation vor. Es gibt zwei Komponenten, die aus denselben Klassen bestehen, allerdings mit jeweils verschiedenen Instanzen. Bisher werden sie zu zwei unterschiedlichen Komponenten, nicht zu einer Komponente mit zwei Komponenteninstanzen. Dies ist aber nicht unbedingt die beste Lösung. Es werden daher Komponententypen eingeführt und der Algorithmus im folgenden erweitert.

Ein *Komponententyp* ist bestimmt durch die Typen der Objekte in der Komponente. Er besteht aus der Summe der Klassen innerhalb der Komponente. Nach Herstellung der Ausgangssituation

haben z.B. alle Komponenten denselben Komponententyp, deren Objekte denselben Typ haben (z.B. eine Liste vom Typ *class List*) (siehe Abbildung 5.32).

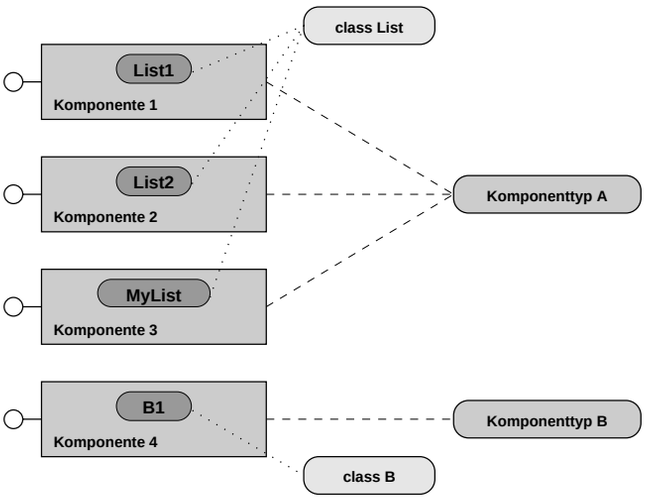


Abbildung 5.32.: Zusammenhang zwischen Komponenten und Komponententypen. Die Komponenten 1 bis 3 haben jeweils ein Objekt vom selben Type.

Die Erweiterung besteht darin, daß nach der Zusammenfassung von zwei Komponenten A und B überprüft wird, ob es weitere Komponentenpaare gibt, die dieselben Typen der gerade zusammengefaßten Komponenten haben. Für all diese Paare wird geprüft, ob die Zusammenfassung der Komponenten eine Reduzierung der Kosten bringt. Es wird allerdings zusätzlich erwartet, daß die Reduktion der Kosten ähnlich gut ist wie bei der Zusammenfassung von Komponente A und B. Dazu wird ein Schwellwert verwendet; in dieser Arbeit wird mindestens die Hälfte der ursprünglichen Reduktion verlangt.

Der erweiterte Algorithmus zur Komponentenbestimmung in Pseudonotation lautet damit:

Vorbereitungen:

1. jedes Objekt wird zu einer eigenen Komponente
2. fasse Attribute eines Objektes mit seiner Komponente zusammen

Hauptteil:

```
1.  do {
2.      for (alle sinnvollen Paare (i;j)) {
3.          berechne red_{i,j}
4.          bestimme red_max; merke Paar (i_max; j_max)
5.      }
6.      if (red_max > 0) {
7.          fasse Komponente i_max und j_max zusammen
8.          for (alle Kombinationen mit type_k == type_i und type_l == type_j) {
9.              if (red_{k,l} >= 0.5 * red_max)
10.                 fasse Komponente k und l zusammen
11.         }
12.     }
13. } while (red_max > 0)
```

Für das Beispieldesign ergeben sich zwei Komponenten, *Comp_DieselEngine_1* und

Comp_PetrolEngine_1 (Abbildung 5.33). Aufgrund der Attributregel (siehe Regel 39) ist diese Situation nach Herstellung der Ausgangslage bereits erreicht und ändert sich dann nicht mehr. Es bleibt anzumerken, daß auch ohne Anwendung der Attributregel dasselbe Ergebnis entsteht. Es werden dann die Attribute einzeln in die jeweilige Komponente gezogen.

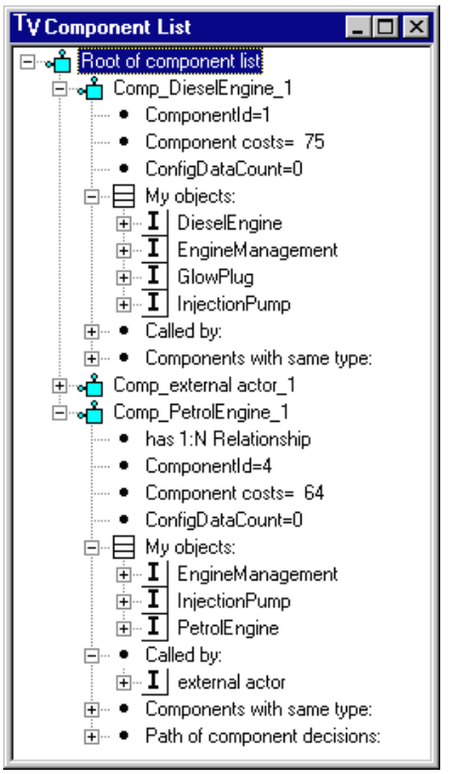


Abbildung 5.33.: Das Ergebnis der Komponentenbestimmung für das Beispieldesign. Die Komponente 'external Actor' ist keine richtige Softwarekomponente, sondern steht stellvertretend für einen beliebigen externen Benutzer.

5.5. Codegenerierung

Die automatische Codegenerierung aus einem GCN Modell ist relativ problemlos möglich. Alle notwendigen Daten sind nach dem Ende des Compilevorganges vorhanden. Die Codegenerierung im Detail ist nicht Teil dieser Arbeit. Es soll nur die prinzipielle Vorgehensweise dargestellt werden. Abbildung 5.34 zeigt die wesentlichen Schritte bei der Codegenerierung.

Sie erfolgt in drei Schritten. Im ersten Schritt werden die Klassendefinitionen mit ihren Methoden und Member-Variablen sowie allen benötigten Typdeklarationen erstellt. Die Daten dazu stammen aus der Klassifikation. Anschließend wird der Strukturgraph verwendet, um die Methoden der Klassen zu implementieren.

Im zweiten Schritt entstehen aus den Klassendeklarationen heraus die Komponenten, indem die Ergebnisse der Komponentenbestimmung benutzt werden. Dabei werdem sowohl Interfaceklassen wie IDL-Files für die Interfacebeschreibung erstellt. Abgeschlossen wird dieser Schritt durch die Definition der Einzelprojekte mit der nachfolgenden Erstellung der Makefiles.

Im dritten Schritt werden die noch rein funktionalen Komponenten in ein organisatorisches Umfeld gesetzt, mit dessen Hilfe die Komponenteninstanzierung wie auch der Startup und Shut-

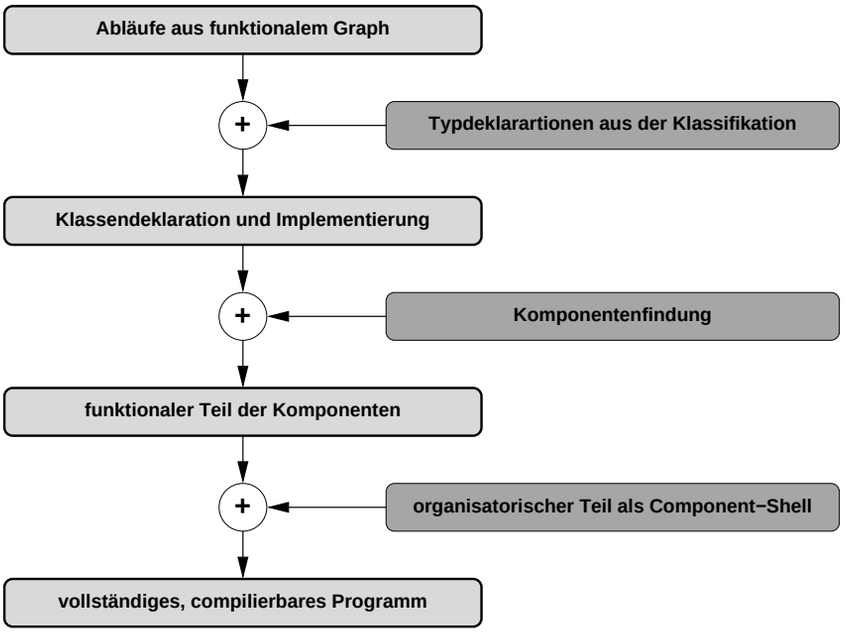


Abbildung 5.34.: Schritte der Codegenerierung aus einem GCN Modell

down, die Konfiguration der Properties und der Test und das Debugging abgewickelt werden können.

Das organisatorische Umfeld ist abhängig von der Komponententechnologie. Wird COM / DCOM von Microsoft zugrunde gelegt, so besteht es aus der ATL[19] sowie einer selbsterstellten Component-Shell, mit deren Hilfe das explizite Schreiben von Organisationscode (z.B. für den Startup) der Komponente vermieden wird. Das Ergebnis der Codegenerierung sind eine Reihe von Projekten mit allen benötigten Sourcen, fertig für den konventionellen Compilevorgang. Das Ergebnis kann vollständig eingesehen werden und theoretisch auch noch von Hand modifiziert werden.

Die Codegenerierung macht einen weiteren Rationalisierungsgewinn des VCDT-Systems sichtbar. Neben den üblichen Vorteilen einer automatischen Codegenerierung (z.B. weniger Programmierfehler), wird dem Entwickler der expliziten Umgang mit der Codierungsmechanik von Komponenten erspart. Als Vorteile ergeben sich dann u.a.:

- kein Wissen über die ATL notwendig
- die Referenzzählung geschieht automatisch
- IDL-Files werden automatisch generiert
- partielle Instanzierung zur Speicherplatzeinsparung geschieht implizit
- Tearoff-Interfaces werden - wenn notwendig - automatisch generiert

[19] ActiceX Template Library, Bestandteil von MS Visual C++

6. Anwendungsbeispiel: Datenbankdesign

Als komplexes Anwendungsbeispiel zur Untersuchung der Praxistauglichkeit von GCN-Modellen wurde eine aktive Realzeit-Datenbank (ARTDB) gewählt. Das Design entstand im Rahmen des Forschungsprojektes FORSOFT II der Bayrischen Forschungsstiftung, Teilprojekt HRS, beim Industriepartner Rohde&Schwarz.

Dieses Kapitel beschreibt zunächst die Anforderungen, die an die ARTDB gestellt werden und erläutert die wesentlichen Eigenschaften der Datenbank.

Anschließend wird das GCN-Modells der Datenbank vorgestellt und untersucht. Die wesentlichen Ergebnisse der Schritte *Aufbau funktionaler Graph*, *Begriffsbestimmung* und *Klassifikation* werden dargestellt. Dem letzten Schritt des Compilevorganges, der *Komponentenbestimmung*, wird ein eigenes Unterkapitel gewidmet. Eine Diskussion der aufgetretenen Probleme schließt das Kapitel ab.

6.1. Anforderungen an die Datenbank

Im FORSOFT-Projekt wurden anhand von Fallbeispielen die wesentlichen Anforderungen an eine aktive Realzeitdatenbank für eingebettete Systeme untersucht [BMW00b]. Es wurde ein Transaktionskonzept erarbeitet, das für diese Anforderungen besonders geeignet ist. Die wichtigsten Ergebnisse werden nun kurz dargestellt.

Datenstruktur

Die Datenstruktur für datendominierte Realzeitsysteme unterscheidet sich erheblich von der Datenstruktur in Informationssystemen. Der Replikationsgrad einzelner Datensätze in der Datenbank ist gering. Komplexe Suchalgorithmen und Anfragesprachen wie SQL (bzw. RT-SQL [PFW97]) sind eher von untergeordneter Bedeutung.

Hauptspeicherdatenbank

Die Datenbank ist hauptspeicherbasiert, denn dadurch ergeben sich einige wesentliche Vorteile [GMS92]:

- Verbesserte Performance durch Verzicht auf langsame Plattenzugriffe.
- Bestimmbare und garantiertbare Zugriffszeiten.
- Keine Notwendigkeit für komplexen Datenstrukturen (z.B. B^+-Baum [Com79]) zur Datenverwaltung.

Eine Hauptspeicherdatenbank kann zwar weniger Daten verwalten als Disk-basierte Systeme; für fast alle Anwendungen im Bereich der eingebetteten Realzeitsysteme stellt dies keine Einschränkung dar.

Aktive Funktionalität

Aktive Datenbanken erleichtern häufig die Anwendungsentwicklung ganz erheblich. Einzelne Daten müssen nicht mehr regelmäßig vom Benutzer abgeholt werden (polling). Die Datenbank informiert betroffene Clients automatisch über Änderungen oder führt *selbständig* (*aktiv*) vorher festgelegte Aktionen durch. Die aktive Funktionalität wird gewöhnlich durch ECA-Regeln festgelegt. ECA steht dabei für *Event*, *Condition* und *Action*: eine Action wird ausgeführt, falls ein Event eingetreten ist und eine bestimmte Bedingung (Condition) erfüllt ist.

Die hier vorgestellte Datenbank implementiert ECA Regeln nur eingeschränkt. Als Event ist ausschließlich die Änderung eines Datums vorgesehen.

Die möglichen Aktionen sind beschränkt auf:

- Information von Clients über das Eintreten von Ereignissen (Notification). Notifications sind dynamisch, d.h. sie können zur Laufzeit vom Client hinzugefügt oder entfernt werden.
- Automatischer Abgleich von abhängigen Daten (Dependence-Rechnung). Zur Beschreibung der Dependence-Rechnungen wurde eine graphische Regelspezifikation definiert [Sch99].

TA Konzept

Verfahren, die die Einflüsse paralleler Transaktionen kontrollieren, heißen *concurrency-control* Verfahren. Es gibt Sperr- [EGLT76], Zeitstempel-basierte [BSR80], optimistische [KR81] (hauptsächlich für Systeme mit überwiegenden Nur-Lesetransaktionen) und Multi-Versions [BG83] Verfahren.

Hier wurde ein Sperrverfahren gewählt, denn Untersuchungen zeigen, daß der Durchsatz bei Sperrverfahren bei beschränkten Ressourcen i.d.R. höher ist [LS96]. Dies gilt insbesondere für Hauptspeicherdatenbanken. Die Performance hängt dort stark von der Granularität der Sperren ab. Optimal ist ein komplettes Sperren der Datenbank [GMS92]. In der Praxis ist dies allerdings nicht möglich, denn eine langdauernde Transaktion könnte dann alle anderen Transaktionen blockieren.

In dieser Datenbank wird als Granularität für die Sperren die *Lockgruppe* verwendet. Eine Lockgruppe besteht aus einer Menge von Daten und wird aus den Abhängigkeiten der Daten berechnet. Wird ein Datum der Lockgruppe in der Datenbank benötigt, wird die ganze Gruppe gesperrt. Dies ist sinnvoll, da weitere Daten der Lockgruppe im Zuge der Dependence-Rechnung sehr wahrscheinlich ebenso benötigt werden. Die genaue Bestimmung der Lockgruppen wird in [BMW00b] beschrieben.

Es werden fünf *Transaktionstypen* unterschieden (siehe Tabelle 6.1). Alle Transaktionen bis auf die explizite Transaktion können im Konfliktfall vom System abgebrochen und (ohne Mitwirkung der Anwendung) neu aufgesetzt werden. Explizite Transaktionen dürfen nicht abgebrochen werden, da sie z.T. mit der Umwelt interagieren und die Folgen nicht ohne weiteres rückgängig gemacht werden können.

TA-Typ	Eigenschaften
Implicit-TA	Schreiben eines einzelnen Datums
Read-TA	eine Menge von Daten wird nur gelesen
Place-TA	eine Menge von Daten wird nur geschrieben
Open-TA	Änderungen werden sofort in der DB sichtbar
Explicit-TA	gewöhnliche Schreib/Lese-Transaktion

Tabelle 6.1.: Die von der Datenbank verwendeten Transaktionstypen.

Ein prioritätsgesteuertes Scheduling dient der Unterstützung von Realzeitanforderungen. Die Datenbank ist kein eigener Task und ist jeweils Teil der Anwendungstasks. So wird die Priorität der Anwendungstask zum Scheduling der Datenbankzugriffe genutzt. Dies spart unnötige Kontextwechsel zwischen der Datenbank und den Anwendungstasks.

6.2. GCN Modell der Datenbank

Das GCN Modell der Datenbank besteht insgesamt aus mehr als 400 Sheets. Hier soll aber nur ein Ausschnitt des Modells betrachtet werden, der *Start der Datenbank*.

Der Start der Datenbank wird eingeleitet durch die Nachrichten 'SetEnvironment' und 'StartOperation' (Abbildung 6.1). Beide Nachrichten gehen von einem externen Aktor aus, der als Rolle den Benutzer 'SystemControl' und die Aufgabe 'Runtime' hat. Der zweite sichtbare Aktor dient dem 'Herunterfahren' der Datenbank.

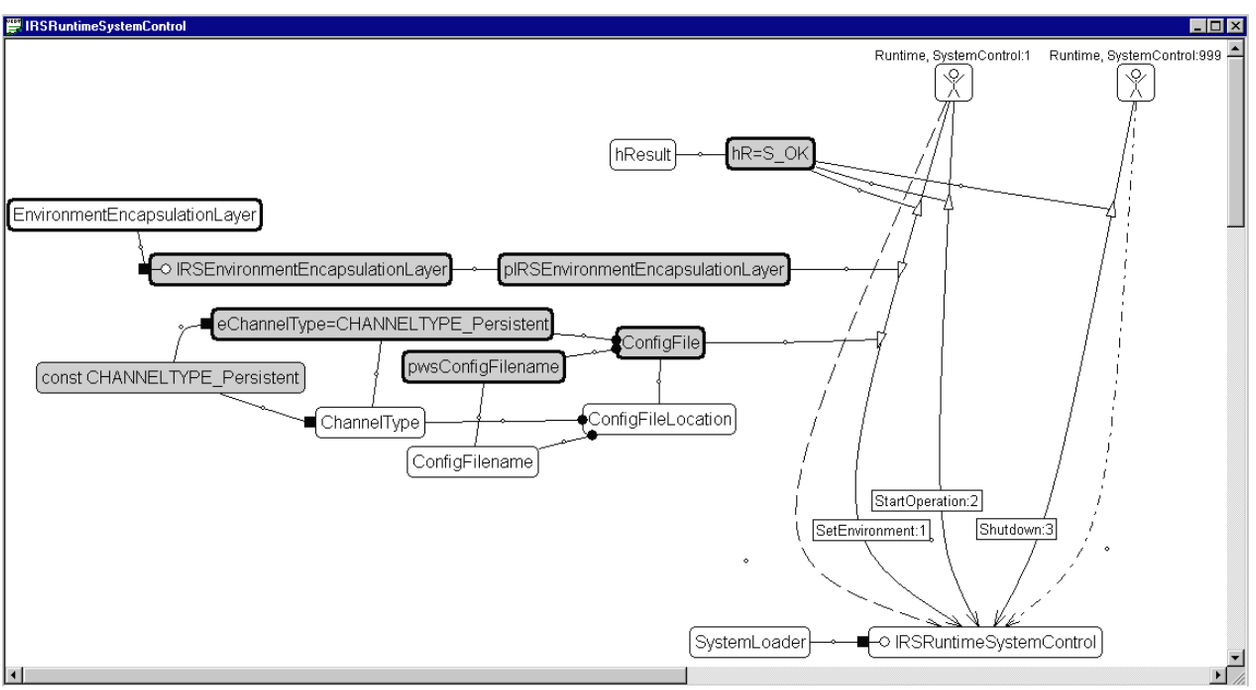


Abbildung 6.1.: Der Start der Datenbank ausgehend von einem externen Aktor.

Zu erkennen ist, daß die Nachrichten nicht direkt an den Begriff 'Systemloader' gehen, sondern an das Interface 'IRSRuntimeSystemControl'. Diese Vorgabe ist notwendig, da die Datenbank sich an gewisse interne Richtlinien von Rohde&Schwarz halten muß. Da das Interface bereits explizit modelliert ist, muß es später nicht vom System generiert werden.

Die Nachricht 'StartOperation' ist der zentrale Punkt beim Systemstart. Ihre Verfeinerung ist recht komplex (Abbildung 6.2).

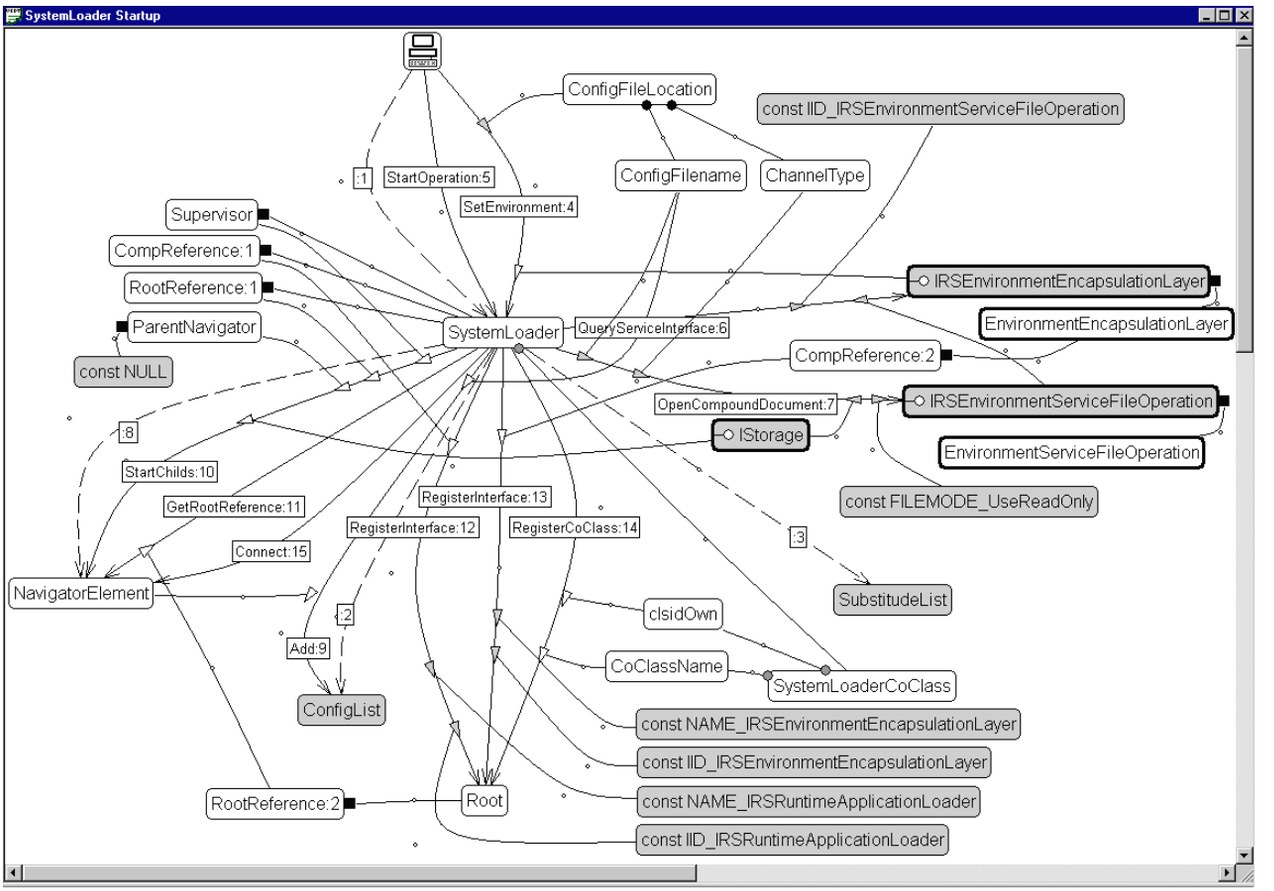


Abbildung 6.2.: Der Start der Datenbank ausgehend von einem externen Aktor.

Der gesamte Startvorgang der Datenbank umfaßt ca. 200 Sheets und wird hier nicht weiter dargestellt.

6.2.1. Ergebnisse des Compilevorgangs

Dieser Abschnitt beschreibt die wichtigsten Ergebnisse der Schritte *Aufbau des funktionalen Graphen*, *Begriffsbestimmung* und *Klassifikation*. Alle angegebenen Zeiten beziehen sich auf einen handelsüblichen PC mit folgenden Eigenschaften:

- AMD Duron-Prozessor, 800MHz
- 512MB RAM
- Betriebssystem Windows NT 4.0, Servicepack 6

Der verfügbare Hauptspeicher hat einen wesentlichen Einfluß auf die Ausführungszeiten des Systems. Sobald Daten in virtuellen Speicher ausgelagert werden müssen, verlängern sich die Ausführungszeiten gegenüber den hier angegebenen Zeiten deutlich.

Alle Messungen wurden mit dem Tool VCDT (Version 1.1) vorgenommen.

Funktionaler Graph

Der Aufbau des funktionalen Graphen benötigt von allen Schritten des Verfahrens den meisten Speicherplatz. Dies liegt v.a. an der Kopienbildung beim Zusammenbau der Teilgraphen. Für das betrachtete Teilmodell (den Startvorgang) der Datenbank werden ca. 165MB Speicher benötigt und der Vorgang dauert ca. 25 Sekunden. Die Komplexität des Designs ist daran zu erkennen, daß der längste Pfad des funktionalen Graphen aus 25 geschachtelten Methodenaufrufen besteht. Die Anzahl der Graphknoten, nach Typen aufgeschlüsselt, ist Tabelle 6.2 zu

Knotentyp	Anzahl
Begriff	13111
Nachricht	13260
Schleife	302
Operation	754

Tabelle 6.2.: Die Anzahl der Knoten des funktionalen Graphen für das Datenbankmodell.

entnehmen. Zusammen sind es mehr als 27000 Knoten. Dazu kommen noch Parameter und Bedingungen als Teil der Nachrichten.

Begriffsbestimmung

Die initiale Begriffsliste umfaßt 14419 Begriffsinstanzen[1] bei 612 verschiedenen Begriffsnamen. Über die Variablen bzw. Attribute werden viele initiale Begriffsinstanzen vereinigt, so daß sich deren Zahl am Ende der Begriffsbestimmung auf 3636 Instanzen verringert.

Der komplette Aufbau der Begriffsliste inklusive der Existenzprüfung dauert ca. 22 Sekunden und benötigt ca. 32 MB Speicherplatz.

Die Existenzprüfung erkennt korrekt, daß fast alle generierten Begriffsinstanzen nicht wieder freigegeben werden und gibt entsprechende Fehlermeldungen aus. Die gemeldeten Fehler sind eine Folge der ausschließlichen Betrachtung des Startvorgangs.

Klassifikation

Die initiale Typliste enthält 678 Typen, 238 davon provisorisch, d.h. ohne konkreten Typ (Einzelheiten siehe Tabelle 6.3). Der Aufbau der Typliste und die anschließende Auswertung der Beziehungen zwischen den Begriffsinstanzen dauert ca. 10 Sekunden. Der Bedarf an Hauptspeicher ist vernachlässigbar, er beträgt unter 1 MB.

Es bleiben nach der Klassifikation 4 Typen übrig, die nicht klassifiziert werden konnten. In allen Fällen liegt dies daran, daß die zur Klassifikation notwendigen Daten im Teilmodell zum Start der Datenbank nicht vorhanden sind (z.B. ein Attribut, das erst später gesetzt wird).

Nach der Auswertung der Beziehungen gibt es insgesamt 581 verschiedene Typen (siehe Tabelle 6.4), davon 121 Klassen. Es gibt 419 Instanzen dieser Klassen, die als Objekte für die Komponentenbestimmung verwendet werden.

[1] die Diskrepanz zur Anzahl der Knoten im funktionalen Graphen (13111) ergibt sich durch die Parameter und Bedingungen der Methoden; sie sind keine Knoten im Graph, sondern Teil der Nachrichten

Typ	Anzahl
Komponente	3
Klasse	120
Struct	51
Numeric	74
Enum	27
EnumElement	0
Interface	30
Reference	0
Konstante	128
ohne Typ	238

Tabelle 6.3.: Die Ausgangssituation der Klassifikation im Detail. Die einfachen Typen (z.B. *string* oder *boolean*) gibt es jeweils nur einmal und sind hier nicht dargestellt.

Typ	Anzahl
Komponente	3
Klasse	121
Struct	35
Numeric	141
Enum	42
EnumElement	61
Interface	30
Reference	67
Konstante	70
ohne Typ	4

Tabelle 6.4.: Das Endergebnis der Klassifikation im Detail. Die einfachen Typen (z.B. *string* oder *boolean*) gibt es jeweils nur einmal und sind hier nicht dargestellt.

Die numerischen Typen (Anzahl 141) werden bei der Zusammenfassung der Typen aufgrund ihrer Deklaration in konkrete Typen umgerechnet. So reduziert sich ihre Zahl auf 6 Typen (*small*, *short*, *long*, *hyper*, *float*, *double*).

6.2.2. Komponentenbestimmung

Dieser Abschnitt widmet sich dem Vorgang der Komponentenfindung für das Datenbankmodell. Es werden zwei unterschiedliche Kostensätze betrachtet und ihre Auswirkungen auf die Komponentenbestimmung untersucht. Es wird dabei immer der erweiterte Algorithmus verwendet.

Kostenmodelle

Der erste Kostensatz (siehe Tabelle 6.5) betont die Verbindungskosten. Sie sind im Vergleich zu den Komponentenkosten besonders hoch. Man wird man also weniger, aber größere Komponenten erwarten können.

Im Gegensatz dazu sind beim zweiten Kostensatz (Tabelle 6.6) die Verbindungskosten besonders niedrig. Zu erwarten sind hier mehr und damit kleinere Komponenten.

Art	Wert	Art	Wert
compBaseCost	10	classTypeCost	10
classBaseCost	5	structTypeCost	8
classMethodCost	3	simpleTypeCost	2
ifBaseCosts	4	interfaceTypeCost	5
ifMethodCost	3	stringTypeCost	10
costNoConfig	20	loopAvgCount	10
costOverConfig	10	costSameIF	1000
cost1toN	100	ifOverCost	10

Tabelle 6.5.: Kostensatz 1. Hier sind die Verbindungskosten besonders hoch.

Art	Wert	Art	Wert
compBaseCost	5	classTypeCost	2
classBaseCost	5	structTypeCost	1
classMethodCost	3	simpleTypeCost	1
ifBaseCosts	4	interfaceTypeCost	1
ifMethodCost	3	stringTypeCost	1
costNoConfig	20	loopAvgCount	5
costOverConfig	10	costSameIf	1000
cost1toN	100	ifOverCost	10

Tabelle 6.6.: Kostensatz 2 mit besonders niedrigen Verbindungskosten.

Ergebnisse

Ausgangspunkt für die Komponentenbestimmung sind die 419 Objekte aus dem Klassifikationsschritt. Jedes Objekt bildet zusammen mit seinen Attributen zunächst eine eigene Komponente. In Tabelle 6.7 sind alle 386 initiale Komponenten zusammengefaßt. Zu beachten ist, daß es von einigen Komponenten viele 'Instanzen' gibt. Um die Tabelle übersichtlich zu halten, sind diese Komponenten in der Tabelle mit einem Multiplikator versehen (z.B. *Comp_CComponentClass_X*) und die Nummer in ihrem Namen ist durch einen Platzhalter ersetzt.

Unter Verwendung des ersten Kostensatzes liefert der erweiterte Algorithmus 27 Komponenten nach 347 Schritten (Tabelle 6.8). Der zweite Kostensatz liefert dieselben 27 Komponenten (siehe Tabelle 6.9).

Dies zeigt, daß das Verfahren robust gegen Änderungen am Kostensatz ist. Das liegt zum großen Teil am Datenbankdesign selbst. Durch die Beschränkung auf den Startvorgang gibt es nur einen externen Aktor. Dies führt maximal zu einem neuen Interface (das nach außen im Design bereits vorgegeben ist). Die Anzahl der Interfaces und Methoden ist damit kein Kriterium. Wichtigstes Kriterium sind hier die Konfigurationsdaten und die 1:N Beziehungen zwischen den Objekten. Alle gefundenen Komponenten besitzen zwischen 4 und 20 Konfigurationsdaten.

Abbildung 6.3 zeigt die Komponente 'DatabaseManager_28' beispielhaft im Detail. Es sind die Objekte der Komponente und ihre Interfaces mit ihren Methoden dargestellt. Die Interfaces *IPersistStream* und *IRSRegisterInterface* sind bereits durch das Design vorgegeben. Durch das System wurde das Interface *Runtime_SystemControl* hinzugefügt mit den Methoden *Connect*, *InitializeOperation*, etc. Andere Komponenten wie z.B. der 'LoaderSubstitude_6', sprechen den DatabaseManager über dieses Interface während des Startvorgangs an.

Component	Cost	Component	Cost	Component	Cost
71x Comp_AnyDefaultResetFctFormular_X	51	Comp_ApplicationList_1	57	Comp_CApplicList_1	57
36x Comp_AnyIncDecrementFctFormular_X	48	3x Comp_CChildList_X	57	Comp_CConfigList_1	45
8x Comp_AnyDependenceFormular_X	48	5x Comp_CCoClassList_X	51	46x Comp_CConfigClass_X	51
36x Comp_AnyLimitFctFormular_X	54	3x Comp_CInterfaceList_X	48	Comp_CLiList_1	51
20x Comp_CComponentClass_X	57	9x Comp_COM_Library_X	45	Comp_ChildList_3	57
10x Comp_CObjectList_X	54	3x Comp_CSubstitudeList_X	51	6x Comp_ComLibrary_X	45
20x Comp_ComponentClass_X	57	Comp_CoClassList_3	51	Comp_CoClassList_1	51
3x Comp_ConversionLibrary_X	51	46x Comp_ConfigClass_X	45	Comp_ComponentClass_9	45
Comp_DatabaseGroup_30	67	Comp_DatabaseGroup_31	67	Comp_DatabaseGroup_32	67
Comp_DatabaseGroup_45	253	Comp_DatabaseGroup_46	67	Comp_DatabaseManager_28	125
Comp_DefaultResetConst_19	86	Comp_DefaultResetFct_19	86	Comp_Dependence_24	71
Comp_ExtendedNumericScalarRow_18	102	Comp_ErrorManager_11	61	Comp_FWAppController_23	71
Comp_FWAppManager_15	71	Comp_IncDecrementConst_20	86	Comp_IncDecrementFct_20	86
3x Comp_LConversionLibrary_X	51	Comp_InterfaceList_1	51	2x Comp_InterfaceList_X	48
Comp_LIController_27	71	Comp_LIHandler_19	81	Comp_LIManager_13	71
Comp_LiList_1	51	Comp_LimitConst_20	86	Comp_LimitFct_20	86
Comp_LoaderSubstitude_6	84	Comp_MapperGroup_15	71	Comp_MapperManager_19	71
Comp_NavigatorElement_1	80	Comp_NavigatorElement_2	77	Comp_ObjectEnum_24	55
10x Comp_ObjectList_X	57	Comp_Parameter_67	55	Comp_Parameter_74	180
Comp_Root_1	103	Comp_RootSubstitude_83	103	Comp_SubstitudeList_1	51
Comp_SubstitudeList_2	51	Comp_SubstitudeList_5	51	Comp_SystemLoader_2	83

Tabelle 6.7.: Initiale Komponenten mit ihren Komponentenkosten (Kostensatz 1). Um die Tabelle kompakt zu halten, sind einigen Komponenten mit Multiplikatoren versehen. Der Komponentenname enthält dann einen Platzhalter X für die 'Instanznummern'.

Component	Cost	Component	Cost	Component	Cost
Comp_DatabaseGroup_30	101	Comp_DatabaseGroup_31	101	Comp_DatabaseGroup_32	101
Comp_DatabaseGroup_45	362	Comp_DatabaseGroup_46	101	Comp_DatabaseManager_28	195
Comp_DefaultResetFct_19	1127	Comp_DefaultResetConst_19	158	Comp_Dependence_24	184
Comp_ErrorManager_11	89	Comp_ExtendedNumericScalarRow_18	155	Comp_FWAppController_23	121
Comp_FWAppManager_15	149	Comp_IncDecrementConst_20	158	Comp_IncDecrementFct_20	529
Comp_LIController_27	121	Comp_LIHandler_19	165	Comp_LIManager_13	143
Comp_LimitConst_20	158	Comp_LimitFct_20	745	Comp_LoaderSubstitude_6	275
Comp_MapperGroup_15	133	Comp_MapperManager_19	133	Comp_NavigatorElement_2	352
Comp_Parameter_74	301	Comp_RootSubstitude_83	213	Comp_UnitManager_217	105

Tabelle 6.8.: Resultat der Komponentenbestimmung für den ersten Kostensatz.

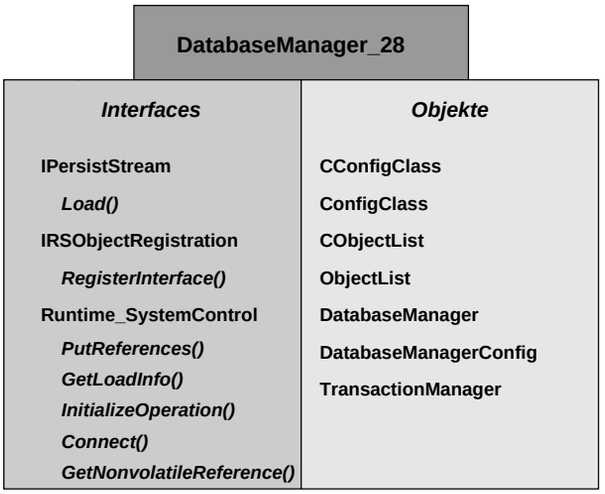


Abbildung 6.3.: Die Komponente *DatabaseManager* im Detail. Die Parameter der Methoden sind hier nicht dargestellt.

Component	Cost	Component	Cost	Component	Cost
Comp_DatabaseGroup_30	96	Comp_DatabaseGroup_31	96	Comp_DatabaseGroup_32	96
Comp_DatabaseGroup_45	357	Comp_DatabaseGroup_46	96	Comp_DatabaseManager_28	190
Comp_DefaultResetFct_19	1122	Comp_DefaultResetConst_19	153	Comp_Dependence_24	179
Comp_ErrorManager_11	84	Comp_ExtendedNumericScalarRow_18	150	Comp_FWAppController_23	116
Comp_FWAppManager_15	144	Comp_IncDecrementConst_20	153	Comp_IncDecrementFct_20	524
Comp_LIController_27	116	Comp_LIHandler_19	160	Comp_LIManager_13	138
Comp_LimitConst_20	153	Comp_LimitFct_20	740	Comp_LoaderSubstitude_6	270
Comp_MapperGroup_15	128	Comp_MapperManager_19	128	Comp_NavigatorElement_2	347
Comp_Parameter_74	296	Comp_RootSubstitude_83	208	Comp_UnitManager_217	100

Tabelle 6.9.: Resultat der Komponentenbestimmung für den zweiten Kostensatz.

Die benötigte Rechenzeit liegt bei ca. 7 Minuten. Dies liegt hauptsächlich an der nicht optimalen Implementierung. In jedem Schritt werden die Verbindungskosten für alle sinnvollen Kombinationen berechnet, auch wenn sie sich zum Schritt davor nicht verändert haben können. Die Berechnung der Verbindungskosten für zwei Komponenten steigt mit der zunehmenden Größe der einzelnen Komponenten deutlich.

Die Entwicklung der Systemkosten während der Komponentenfindung ist in Abbildung 6.4 für den ersten Kostensatz dargestellt. Bei Schritt 245 erkennt man am Sprung das Verhalten des erweiterten Algorithmus. Wie immer werden dort zwei Komponenten (im folgenden mit *Auslösepaar* bezeichnet) zusammengefaßt. Es gibt nun fünf weitere gleichartige Komponentenpaare, die ebenfalls zusammengefaßt werden. Gleichartig bedeutet, daß die erste Komponente des Paars denselben Komponententyp besitzt, wie die erste Komponente des Auslösepaars; für die zweite Komponente gilt dasselbe: ihr Komponententyp ist identisch zu dem der zweiten Komponente des Auslösepaars.

Die Linearität der Systemkosten ist zunächst etwas verblüffend. Sie entsteht hier hauptsächlich, weil das Datenbankdesign sehr symmetrisch aufgebaut ist und die initialen Komponenten ähnlich 'teuer' sind: die Kosten bewegen sich meist zwischen 50 und 100. Bei der Zusammenfassung zweier Komponenten wird deswegen in jedem Schritt auch eine ähnlich Reduktion der Kosten erzielt. Bei genauer Betrachtung der Systemkostenentwicklung sind aber durchaus Unterschiede in der Reduktion pro Schritt zu erkennen, z. B. im Schrittbereich 140 bis 150 (Auschnittvergrößerung siehe Abbildung 6.5).

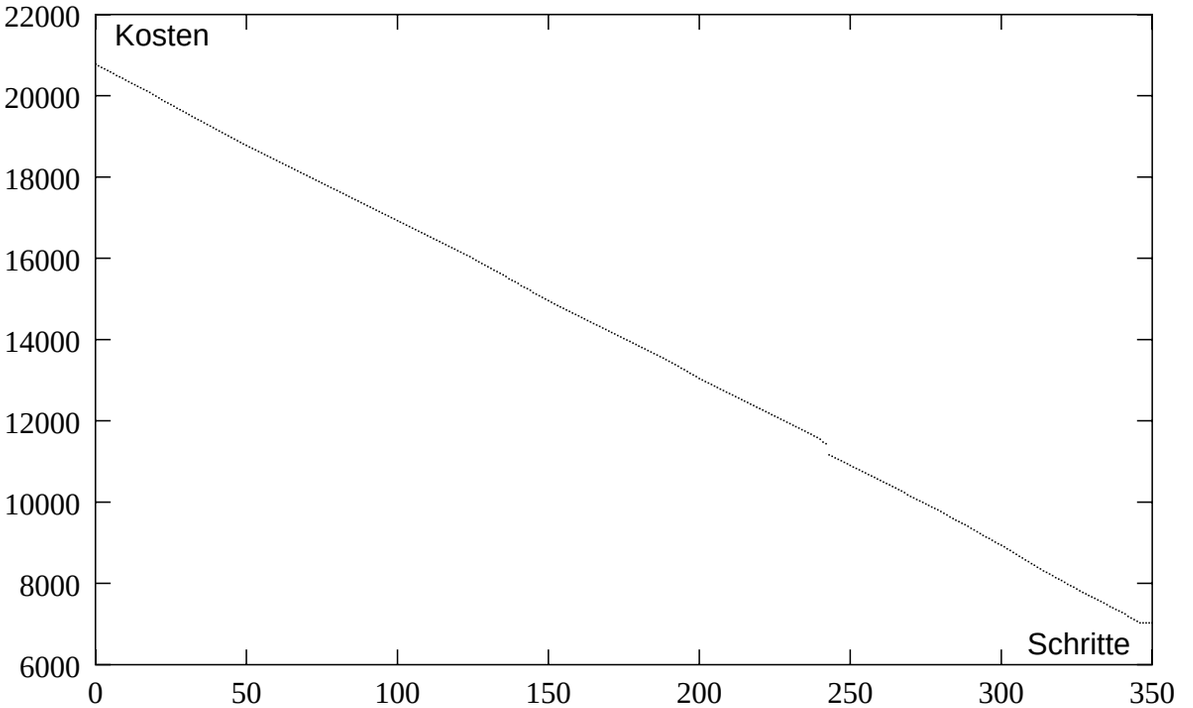


Abbildung 6.4.: Die Entwicklung der Systemkosten bei der Komponentenbestimmung (Kostensatz 1)

6.3. Problemdiskussion

Die Ergebnisse der einzelnen Schritte des Verfahrens belegen, daß das Verfahren funktioniert. Die Klassifikation der Begriffsinstanzen über Beobachtbarkeit und Steuerbarkeit funktioniert sehr zuverlässig. Die Komponentenbestimmung liefert vernünftige Ergebnisse, die im Bereich der Erwartungen liegen. Es werden aber tendenziell mehr Komponenten gebildet, als notwendig sind. Dieses Verhalten hat zwei Ursachen:

- Durch die Unterscheidung von Begriffsinstanzen (zur Vermeidung des Namespace-Problems) treffen sich nur Begriffe gleichen Namens, die sich im Design wirklich begegnen. Begriffsinstanzen mit Methoden, die sich nicht begegnen, bilden neue Klassen, auch wenn sie sehr ähnlich sind (im Datenbankdesign z.B. die unterschiedlichen Instanzen von 'DatabaseGroup'). Dies führt für die Komponentenbestimmung dann schon initial zu unterschiedlichen Komponententypen und damit zu unterschiedlichen Komponenten. Ein Ähnlichkeitsmaß für Klassen würde das Problem reduzieren: Klassen mit gleichem Namen könnten darauf untersucht werden, ob sie gleiche Methoden besitzen. Falls dies der Fall ist, könnten die Klassen zusammengefaßt werden. Die gemeinsame Klasse besitzt dann die Vereinigungsmenge der Methoden und Attribute. Ob Methoden gleich sind, ist aber nicht ganz trivial festzustellen. Es muß auf jeden Fall gelten, daß die Methoden gleich heißen und gleiche Parameter haben. Zusätzlich muß sichergestellt werden, daß die Implementierungen der Methoden gleich sind und dies ist der eigentlich aufwendige Teil.
- Der erweiterte Algorithmus verwendet Komponententypen, um das Problem der sehr ähnlichen, aber doch unterschiedlichen Komponenten zu minimieren. Es zeigte sich allerdings, daß das vorgeschlagene Verfahren nicht ausreichend ist.

  Komponenten, die zunächst denselben Komponententyp besitzen, sind Instanzen derselben

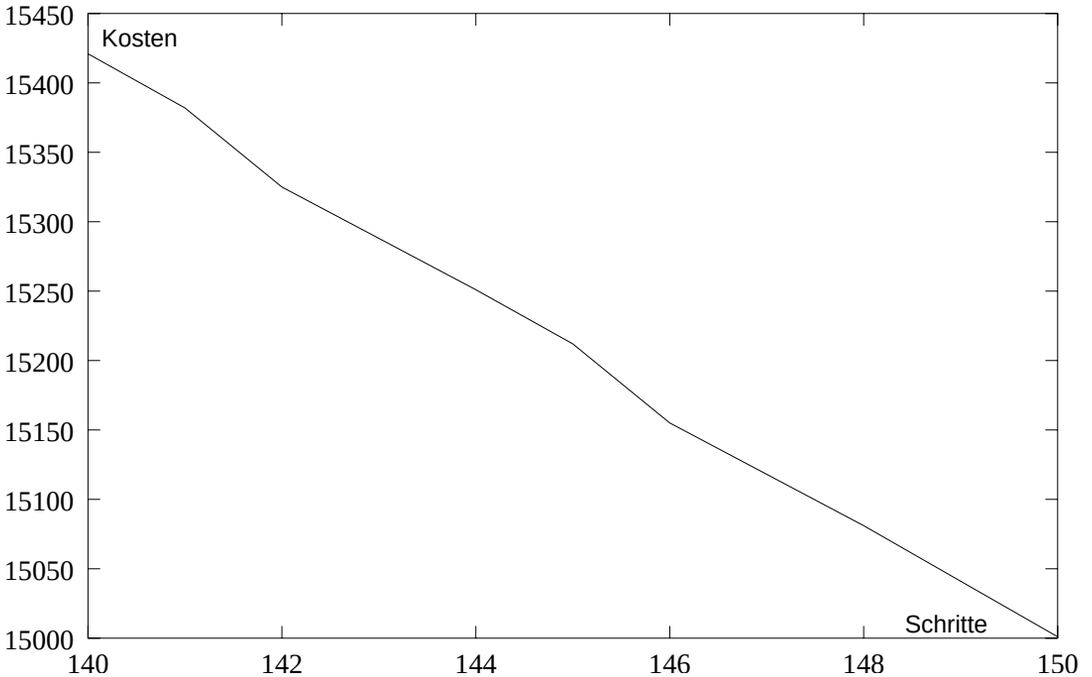


Abbildung 6.5.: Die Entwicklung der Systemkosten im Bereich der Schritte 140 bis 150.

Klasse, also Instanzen derselben Komponente. Am Anfang der Komponentenbestimmung können Komponenteninstanzen gleichen Typs noch recht gut parallel gebildet werden. Sobald allerdings eine der Komponenteninstanzen eine Beziehung zu einem Objekt besitzt, die die anderen Instanzen nicht haben, laufen die Komponententypen auseinander. Dies geschieht häufig aufgrund unterschiedlicher Einsatzgebiete der einzelnen Komponenteninstanzen. Nach dem Auseinanderlaufen der Typen ist eine Zusammenführung der dann unterschiedlichen Komponenteninstanzen zu einer Komponente nicht mehr möglich. An dieser Stelle fehlt ein Ähnlichkeitsmaß für Komponenten. Hierzu müßte die Überschneidung von Objekten und Interfaces zwischen Komponenten gleichen Namens betrachtet werden.

Eine weitere Schwäche zeigte sich bei der Behandlung von Rekursionen. Die Trennung zwischen dem Aufbau des funktionalen Graphen und der Existenzprüfung und damit der Begriffsbestimmung ist nicht in allen Fällen die beste Wahl. Es gibt Designkonstrukte, die erst durch die Erkenntnisse der Existenzprüfung korrekt aufgelöst werden können. Sie entstehen, wenn in einer Rekursionsebene nicht alle Alternativen an einem Splitter erfüllt werden können. Die Information, welche Alternative in welcher Rekursionsstufe erzeugt wurde, geht u.U. verloren, wenn sich Begriffsinstanzen bei der Begriffsbestimmung treffen. In einer weiteren Rekursion im Modell kann es aber notwendig sein, genau die Alternativen zu kennen, die auf der korrespondierenden Stufe der ersten Rekursion benutzt wurden. Ansonsten werden zu viele Begriffsinstanzen erzeugt.

Im Beispieldesign wurden diese Stellen explizit umgangen. Um das Verfahren robust für alle Konstrukte zu bekommen, erscheint es notwendig, nach jeder angehängten Verfeinerung die Existenzprüfung durchzuführen. Dadurch kann festgestellt werden, welche Alternativen zu einem Zeitpunkt wirklich bekannt bzw. gültig sind, um dann nur mit sinnvollen Verfeinerungen fortzufahren.

7. Zusammenfassung und Ausblick

Die vorliegende Arbeit beschreibt ein Verfahren zur automatischen Bestimmung von Komponenten, die dazu notwendige Spezifikationssprache *Graphical Concept Network (GCN)* und als reales Anwendungsbeispiel die Realisierung einer aktiven Datenbank für eingebettete Systeme mit dem Graphical Concept Network.

Ziel ist die Spezifikation eines Softwaresystems ohne Implizierung von konkreten Designentscheidungen. Alle Designentscheidungen werden dem System überlassen, das mit Hilfe von Regeln aus einem gegebenen GCN-Modell ein komponentenbasiertes Design ableitet.

Softwaresysteme werden im GCN durch Begriffe und deren Beziehungen untereinander definiert. Die Spezifikation des Systems besteht dabei aus vielen Einzelteilen, sogenannten *Sheets*. Die Spezifikation erfolgt Use-Case getrieben, dabei umfaßt jeder Use-Case in der Regel eine Menge von Sheets, die hierarchisch gegliedert sein können.

Aus einem vollständig spezifizierten GCN-Modell werden durch einen Compilevorgang die konkreten Komponenten und ihre Interfaces abgeleitet. Der Vorgang besteht aus vier Schritten:

1. Aufbau der funktionalen Struktur des Modells,
2. Bestimmung der Begriffe und Begriffsinstanzen,
3. Klassifikation der Begriffsinstanzen und
4. Bestimmung der Komponenten.

Die funktionale Struktur eines GCN-Modells wird durch einen gerichteten Graphen abgebildet. Dieser Graph dient der internen Repräsentation der Spezifikation. Zur Bestimmung des Graphen werden die einzelnen Sheets zunächst jeweils in einen eigenen Teilgraphen transformiert. Diese Teilgraphen werden dann anhand von Verfeinerungsregeln zu einen Gesamtgraphen zusammengesetzt. Dieser *Strukturgraph* des Modelles enthält alle potentiellen Ausführungspfade inklusive alle potentiellen Verzweigungspunkte des Programms. Durch den Zusammenbau dieses funktionalen Graphen werden bereits viele Fehler im Modell entdeckt, z.B. nicht passende Parameter oder fehlende Verfeinerungen.

Aus dem funktionalen Graphen des Modells werden dann die Begriffe extrahiert. Hier werden *Instanzen* von Begriffen unterschieden, um später bei der Komponentenfindung die wirklich benötigten Objekte (als Instanzen von Klassen) zu kennen. Begriffe werden durch Parameterübergaben durch ein GCN Modell transportiert und in (vom System generierten) Variablen gespeichert. Nur Begriffsinstanzen, die sich im Design begegnen, werden zu einer Instanz zusammengefaßt. Während der Begriffsbestimmung wird zusätzlich eine *Existenzprüfung* durchgeführt. Sie dient zur frühzeitigen Aufdeckung von potentiellen Laufzeitfehlern. Untersucht wird, ob sich Begriffsinstanzen kennen und ob verwendete Begriffsinstanzen definiert sind (Stichwort: nicht initialisierte Variablen). Es wird auch geklärt, ob die vom System generierten Variablen lokal oder global sind. Nur globale Variablen werden später zu Attributen.

Die Begriffsinstanzen werden anschließend klassifiziert. Begriffsinstanzen, die als Endbegriffe definiert wurden, haben bereits einen festen Typ. Für alle übrigen Begriffsinstanzen werden ihre Beziehungen zu anderen Begriffsinstanzen ausgewertet. Über die systemtheoretischen Überlegungen der Beobachtbarkeit und Steuerbarkeit werden die Typen der Endbegriffe ins Modell propagiert. Neue Typen entstehen durch Begriffsinstanzen, die Nachrichten empfangen. Sie bilden neue Klassen.

Nachdem alle Begriffsinstanzen klassifiziert sind, werden für die Komponentenfindung nur noch die Objekte (Instanzen von Klassen) und Interfaces betrachtet. Die Objekte haben untereinander viele Beziehungen. Diese Beziehungen werden mit Kosten versehen und die Summe der Kosten, Gesamtkosten des Systems genannt, sind ein Maß für die Qualität des Systems. Ziel der Komponentenbestimmung ist eine Minimierung der Gesamtkosten. Zunächst wird jedes Objekt zu einer eigenen Komponente. Anschließend werden immer die beiden Komponenten zusammengefaßt, deren Vereinigung die höchste Reduktion der Kosten bringt. Dies geschieht solange, bis die Gesamtkosten nicht mehr sinken.

An diesem Punkt kann ein Codegenerator ein ablauffähiges System erzeugen, daß aus den gefundenen Komponenten besteht. Die Codegenerierung wurde im Rahmen dieser Arbeit allerdings nicht untersucht.

Das Verfahren wurde an einem realen Beispiel getestet. Dazu wurde zum einen das prototypische Tool VCDT entwickelt, das die Eingabe und Verarbeitung eines GCN Modells ermöglicht und zum anderen wurde das GCN Modell einer aktiven Realzeitdatenbank erstellt.

Bei den Untersuchungen zeigten sich auch Schwächen des Verfahrens:

- Namensgleiche Begriffsinstanzen, die bei der Klassifizierung zu neuen Klassen werden, bilden häufig fast gleiche, aber unterschiedliche Klassen. Eine Instanz hat beispielsweise eine Methode mehr als die andere Instanz. Hier fehlt ein Ähnlichkeitsmaß für Klassen.

- Es werden tendenziell mehr unterschiedliche Komponenten erzeugt, als nötig wäre. Dadurch gibt es weniger Instanzen derselben Komponente, sondern mehr Komponenten mit jeweils einer einzigen Instanz. Hier ist ein geeignetes Ähnlichkeitsmaß für Komponenten notwendig.

Ziel der Verwendung von Komponenten ist die Steigerung von Wiederverwendung in der Softwareentwicklung. Komponenten, die durch dieses Verfahren generierte wurden, können nach der Codegenerierung ohne Probleme wiederverwendet werden. Sobald allerdings generierte Komponenten im Einsatz sind, dürfen Verbesserungen an einem GCN Modell nicht zu vollkommen anderen Komponenten führen. Es muß daher eine Möglichkeit vorgesehen werden, um Komponenten bzw. ihre Interfaces 'einzufrieren'. Nur so können Komponenten kompatibel gehalten werden und alte Versionen durch verbesserte Varianten ersetzt werden. Als Syntaxelement könnten dazu z.B. die End-Interfaces verwendet werden. Sie müssen beim 'Einfriervorgang' automatisch vom Tool erzeugt und ins Modell eingefügt werden bzw. in einem Repository abgelegt werden.

In Zuge dieser Arbeit sind längst nicht alle Fragen beantwortet und zusätzlich neue aufgeworfen worden. Weitere Arbeit muß z.B. auf folgenden Gebieten geleistet werden:

- *Erweiterung des GCN um weitere Syntaxelemente.* Die bisherigen Syntaxelemente sind insoweit vollständig, daß das GCN eine allgemeine Programmiersprache ist. Durch eine Ergänzung um z.B. Zustandsautomaten könnte die Erstellung von GCN Modellen aber in bestimmten Anwendungsbereichen noch vereinfacht werden.

- *Mehr und bessere Regeln.* Die in dieser Arbeit vorgestellten Regeln sind u.U. nicht optimal. Es fehlen umfaßende Untersuchung, die belegen, daß die aufgestellten Regeln wirklich sinnvoll sind. Eine Regel z.B. besagt, daß es an einer Komponente maximal 10 Interfaces geben soll. Ihre Begründung stützt sich psychologische Untersuchungen aus dem Jahre 1954. Ist es heute sinnvoller, eine kleinere Zahl anzunehmen? Oder ist eine Beschränkung überhaupt sinnvoll?

  Die Regeln in dieser Arbeit sind sicher auch nicht vollständig. Es sind weitere Regeln denkbar, die sich z.B. mit der nachträglichen Erweiterbarkeit, der Wiederverwendung oder der Anpaßbarkeit der Komponenten beschäftigen. Auch spezielle Regeln für die Verwendung von Komponenten in Frameworks sind vorstellbar.

- *Codegenerierung.* Die Abbildung eines GCN Modells auf verschiedene objekt-orientierte Zielsprachen wie Java oder C++ und verschiedene Komponentensysteme wie COM oder JavaBeans wurde in dieser Arbeit nicht untersucht. Hier ist u.a. zu klären, mit wieviel Aufwand ein GCN Modell Sprach-unabhängig gehalten werden kann (man denke hier z.B. an die Typen der Endbegriffe).

- *Re-engineering.* Ein großes Problem in der Softwareentwicklung ist Legacy-Software. Es ist vorstellbar, den Quellcode, sofern vorhanden, mit einem geeigneten Parser in ein GCN Modell zu übernehmen. Der alte Code kann direkt in einen vollständigen funktionalen Graphen überführt werden. Aus dem funktionalen Graphen kann dann ein komponentenbasiertes Design der Legacy-Software erzeugt werden.

A. VCDT

Das Tool VCDT (Visual Component Design Tool) ist eine Multi-Document MFC (Microsoft Foundation Class) Applikation und läuft unter dem Betriebssystem MS Windows (NT, 2000). Die Oberfläche (Abbildung A.1) bietet links ein Navigationsfenster für die einzelnen Sheets. Jedes Sheet wird in einem eigenen Fenster dargestellt. Mit dem *Operation-Control-Panel*, rechts abgebildet, können die einzelnen Schritte des Compilevorgangs gesteuert werden. Um die Komponentenbestimmung nachvollziehen zu können, kann sie schrittweise durchgeführt werden.

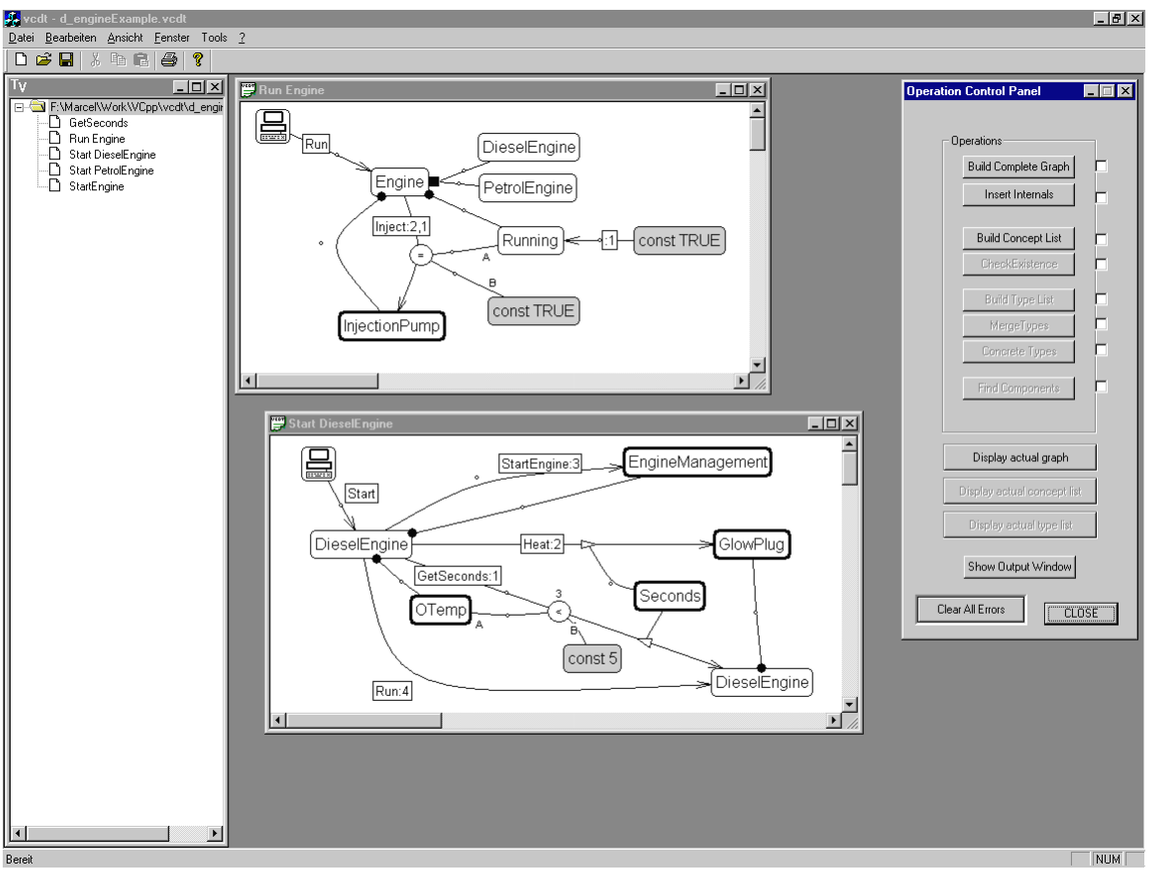


Abbildung A.1.: Oberfläche von VCDT: links das Navigationsfenster, rechts das Operation-Control-Panel.

Das Tool erleichtert die Handhabung eines Modells, indem es u.a. folgende Erweiterungen bzgl. des formalen Modells implementiert:

- Sheets und Begriffe können mit Kommentaren versehen werden.
- Zeichenvereinfachung durch mehrere Aktoren in einem Sheet.
- Eine Menge von Begriffen kann zu einem *Listenbegriff* zusammengefaßt werden und ge-

meinsam angesprochen werden.

- Konstanten können mit 'auto' markiert werden. Ihr Wert wird dann vom Codegenerator automatisch festgelegt.

Literaturverzeichnis

[AHE+96] Andler, Sten ; Hansson, Jörgen ; Eriksson, Joakim ; Mellin, Jonas ; Berndtsson, Mikael ; Eftring, Bengt: DeeDS Towards a Distributed and Active Real-Time Database System. In: *SIGMOD Record 25(1)* (1996), S. 38–40

[AKGM96] Adelberg, B. ; Kao, B. ; Gracia-Molina, H.: An overview of the STanford Real-Time Information Processor (STRIP). In: *ACM SIGMOD Record 25* (1996), S. 37–38

[B+98] Broy, Manfred [u. a.]: What characterizes a (software) component? In: *Software Concept & Tools* 19 (1998), S. 49–56

[BBKZ93] Branding, H. ; Buchmann, A. ; Kudrass, T. ; Zimmermann, J.: Rules in an Open System: The REACH Rule System. In: Paton, N. W. (Hrsg.) ; Williams, M. H. (Hrsg.): *Rules in Database Systems.* Springer, 1993, S. 111–126

[BD00] Brügge, Bernd ; Dutoit, Allen H.: *Object-Oriented Software Engineering; Conquering Complex and Changing Systems.* Prentice Hall, 2000

[BG83] Bernstein, Philip A. ; Goodman, Nathan: Multiversion Concurrency Control - Theory and Algorithms. In: *TODS* 8 (1983), Nr. 4, S. 465–483

[Blö00] Blösch, Jürgen: *Entwicklung einer generischen Test-Bench basierend auf einer aktiven Realzeitdatenbank*, TU-München, Diplomarbeit, 2000

[BMW00a] Birkhold, Marcel ; Münnich, Alexander ; Woitschach, Peter: Eine datenbankbasierte Architektur für Komponentensoftware in eingebetteten Realzeitsystemen. In: *Informationstechnik und Technische Informatik, it+ti* (2000), Juni, S. 40–48

[BMW00b] Birkhold, Marcel ; Münnich, Alexander ; Woitschach, Peter: A real-time transaction concept for active main-memory databases for component architectures / Lehrstuhl für Realzeit Computersysteme (RCS), Technische Universität München. 2000. – Forschungsbericht. http://www.rcs.ei.tum.de/~birkhold/VCDT

[Boo94] Booch, Grady: *Object-oriented Analysis and Design with Applications.* Benjamin/Cunnings Inc., 1994

[Bro98] Broy, Manfred: A uniform mathematical concept of a component. In: *Software Concept & Tools* 19 (1998), S. 57–59

[BRS+99] Bergner, Klaus ; Rausch, Andreas ; Sihling, Marc ; Vilbig, Alexander ; Broy, Manfred: A Formal Model for Componentware. In: *Foundations of Component-Based Systems* (1999)

[BSR80] Bernstein, Philip A. ; Shipman, David W. ; Rothnie, James B.: Concurrency Control in a System for Distributed Databases (SDD-1). In: *ACM Transactions on Database Systems* 5 (1980), Nr. 1, S. 18–51

[BW97] Büchi, Martin ; Weck, Wolfgang: A Plea for Gray-Box Components / Turku Centre for Computer Science. 1997. – Forschungsbericht

[BW01] Birkhold, Marcel ; Woitschach, Peter: Graphical Concept Networks: towards automatic determination of components. In: Bauknecht, K. (Hrsg.) [u. a.]: *Informatik 2001; Tagungsband*, Österreichische Computer Gesellschaft, 2001, S. 817–822

[Car00] Carter, Rita: *Mapping the Mind.* University of Carlifornia Press, 2000

[CBB+89] Chakravarthy, S. ; Blaustein, B. ; Buchmann, A. P. ; Carey, M. ; Dayal, U. ; Goldhirsch, D. ; Hsu, M. ; Jauhari, R. ; Livny, M. ; McCarthy, D. ; McKee, R. ; Rosenthal, A.: HiPACH: A Resaerch Project in Active Time-Constrained Database Management - Final Technical Report. In: *Technical Report XAIT-89-02, Reference Number 187, Xerox Advanced Information Technology* (1989), July

[Cha96] Chappell, David: *ActiveX und OLE verstehen.* Mircosoft Press, 1996

[Com79] Comer, Douglas: The Ubiquitous B-Tree. In: *Computing Surveys, Vol. 11, No. 2* (1979), June, S. 121–137

[Cor01] Corporation, Borland S. *Visibroker ORB.* http://www.borland.com/visibroker/. 2001

[CS96] Ciupke, Oliver ; Schmidt, Rainer: Components as Context-Indeendent Units of Software. In: *Workshop Reader ECOOP* (1996)

[EGLT76] Eswaran, K. P. ; Gray, J. N. ; Lorie, R. A. ; Traiger, I. L.: The Notions of Consistency an Predicate Locks in a Database System. In: *Comm. ACM 19:11* (1976), S. 624–633

[Fow97] Fowler, Martin: *UML Distilled.* Addison-Wesley-Longman, 1997

[GD92] Gatziu, Stelle ; Dittrich, Klaus R.: SAMOS: An Active Object-Oriented Database System. In: *IEEE Data Engineering, Special Issue an Active Databases, 15(1-4):27-30* (1992), December

[GJ91] Gehani, N. H. ; Jagadish, H. V.: Ode as an Active Database: Constraints and Triggers. In: *Proceedings of the 17th International Conference on VLDB* (1991)

[GMS92] Garcia-Molina, Hector ; Salem, Kenneth: Main Memory Database Systems: An Overview. In: *IEEE Transactions on Knowledge and Data Engeneering, Vol. 4, No. 6* (1992), December, S. 509–516

[Gri98] Griffel, Frank: *Componentware. Konzepte und Techniken eines Softwareparadigmas.* dpunkt-Verlag, 1998

[GT00] Gruhn, Volker ; Thiel, Andreas: *Komponentenmodelle: DCOM, JavaBeans, JavaEnterpriseBeans, CORBA.* Addison Wesley, 2000

[HB99] Hansson, Jörgen ; Berendtsson, Mikael: Active Real-Time Database Systems. In: Paton, N. W. (Hrsg.): *Active Rules in Database Systems.* Springer, 1999, S. 405–423

[HJS01] Hofmann, Johann ; Jobst, Fritz ; Schabenberger, Roland: *Programmieren mit COM und CORBA.* Hanser Verlag, München, 2001

[IBM98] IBM. *Homepage zur Object Constraint Language.* http://www.ibm.com/software/ad/library/standards/ocl.html. 1998

[IEE] IEEE: IEEE standard for binary floating point arithmetic. In: *SIGPLAN Notices 22* , S. 9–25

[Ion01] Iona. *Orbix ORB*. http://www.iona.com/products/orbhome.htm. 2001

[KR81] Kung, H. T. ; Robinson, John T.: On Optimistic Methods for Concurrency Control. In: *TODS* 6 (1981), Nr. 2, S. 213–226

[Kru00] Kruchten, Philippe: *The Rational Unfied Process, An Introduction*. Addison Wesley, 2000

[LS96] Lee, Juhnyoung ; Son, Sang H.: Performance of Concurrency Control Algorithms for Real-Time Database Systems. In: *Performance of Concurrency Control Mechanisms in Centralized Database Systems* (1996), S. 429–460

[Maj99] Major, Al: *COM IDL & Interface Design*. Wrox Press Ltd, 1999

[MBFW99] Münnich, Alexander ; Birkhold, Marcel ; Färber, Georg ; Woitschach, Peter: Towards an Active Real-Time Database-Based Architecture for Real-Time Systems using Configurable, Standardized Components. In: *IEEE Proceedings of the IDEAS'99, Montreal, Canada*, 1999, S. 351–359

[Mey94] Meyer, Bertrand: *Object-Oriented Software Construction*. Prentice Hall, 1994

[Mic97] Microsoft Corporation: *The Component Object Model Specification*. 1997. – http://www.microsoft.com/oledev/

[Mic98] Microsoft Corporation. *MIDL Programmer's Guide and Reference*. in: Microsoft Developer's Network (MSDN), Plattform-SDK, VC++6.0 beiliegend, Microsoft. 1998

[Mil56] Miller, G.A.: The magical number seven, plus or minus two: Some limits on our capacity for processing information. In: *Psychological Reviw* 63 (1956), S. 81–97

[Min75] Minsky, Marvin: A framework for representing knowledge. In: P. Winston (Hrsg.): *The Psychology of Computer Vision*. McGraw-Hill, 1975, S. 211–277

[Mit89] Mittendorfer, Josef: *Objektorientierte Programmierung mit C++ und Smalltalk*. Addison-Wesley, 1989

[Mor97] Morrison, Michael: *Java 1.1 für Insider*. SAMS, 1997

[Nat01] National Instruments. *Homepage*. http://www.nationalinst.com. 2001

[OH98] Orfali, Robert ; Harkey, Dan: *Client/Server Programming with Java and CORBA*. 2. John Wiley & Sons, Inc., New York, 1998

[OMG96] Object Management Group, OMG: *The Common Object Request Broker: Architecture and Specification*. 1996. – http://ww.omg.org

[OMG00] Object Management Group, OMG: *UML Specification*. 2000. – http://www.omg.org/technology/documents/formal/unified_modeling_language.htm

[PFW97] Prichard, J.J. ; Fortier, P.J. ; Wolfe, V.F.: RTSQL: Extending the SQL2 Standards to Support Real-Time Databases. In: *Real-Time Database and Information Systems: Research Advances*. Kluwer Academic, 1997, S. 289–310

[Pos96] Poswig, Jörg: *Visuelle Programmierung*. Hanser, 1996

[Raj98] Raj, Gopalan S. *A Detailed comparison of CORBA, DCOM and Java/RMI.* http://www.execpc.com/~gopalan/misc/compare.html. 1998

[Ram96] Ramamritham, Krithi: Real-Time Databases. In: *Journal of Distributed and Parallel Databases, Volume 1, Number 2, pp. 199- 226, invited paper* (1996)

[Rie96] Riel, Arthur: *Object-Oriented Design Heuristics.* Addison-Wesley, 1996

[Sch92] Schöning, Uwe: *Vorlesungsskript Informatik IV (Grundzüge der Theoretischen Informatik).* Uni Ulm - Abteilung Theroretische Informatik, 1992

[Sch97] Schöning, Uwe: *Algorithmen - kurz gefasst.* Spektrum Akademischer Verlag, 1997

[Sch98] Schiffer, Stefan: *Visuelle Programmierung: Grundlagen und Einsatzmöglichkeiten.* Addison-Wesley-Longman, 1998

[Sch99] Schwaiger, Dietmar. *Prototypische Realisierung eines Systems zum datenorientierten Entwurf einer Gerätezustandsdatenbasis, Diplomarbeit, TU-München.* 1999

[Sof01] Software AG. *Homepage.* http://www.software-ag.de. 2001

[SRH90] Stonebraker, Michael ; Rowe, Lawrence ; Hirohama, Michael: The Implementation of POSTGRES. In: *IEEE Transactions on Knowledge and Data Engineering 2(7):125-142* (1990), March

[Sun97] Sun Microsystems. *Specification of JavaBeans API.* http://www.sun.com/beans. 1997

[Sun99] Sun Microsystems. *Java Homepage.* http://www.sun.com/java. 1999

[Szy97] Szyperski, Clemens: *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1997

[Unb93] Unbehauen, Rolf: *Systemtherorie: Gundlagen für Ingenieure.* Oldenburg Verlag, 1993

[V-M01] V-Modell. *Homepage zum V-Modell.* http://www.v-modell.iabg.de. 2001

[VB00] Van Gurp, Jilles ; Bosch, Jan: Design, implementation and evolution of object oriented frameworks: concepts & guidelines. In: *Software - Practice and Experience* (2000)

[WB01] Woitschach, Peter ; Birkhold, Marcel: *Visual Component Design Tool: Syntax und deren Anwendung.* 2001. – Rohde & Schwarz GmbH, http://www.ei.tum.de/~birkhold/VCDT/

[WC96] Widom, Jennifer ; Ceri, Stefano: Introduction to Active Database Systems. In: *Active Databse Systems* (1996), S. 1–41

Danke

Die Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Realzeit-Computersysteme an der Technischen Universität München. Ich möchte mich hier bei allen bedanken, die mich bei der Entstehung dieser Arbeit unterstützt haben. Mein besonderer Dank gilt dabei:

- Herrn Prof. G. Färber für seine Bereitschaft, das Erstgutachten zu übernehmen und seine Begleitung der Arbeit,
- Herrn Prof. B. Brügge für das Zweitgutachten und seine sehr hilfreichen Anregungen.
- allen Mitarbeitern und Kollegen am Lehrstuhl und im FORSOFT Projekt für das angenehme Arbeitsklima und
- Herrn Dipl.-Ing. P. Woitschach von der Firma Rohde & Schwarz, der jederzeit hilfreiche Anregungen hatte und stets ein offener Ansprechpartner für Diskussionen war.

Für ihre Mühe, die Arbeit Korrektur zu lesen, sei Bernd, Carola und insbesondere Ute herzlich gedankt.

Meiner Freundin Astrid danke ich für die liebevolle Begleitung dieses nicht immer einfachen Lebensabschnittes.