Teoribaseret udvikling af pålidelige webbaserede systemer

Theory based development of reliable web based systems

Tue Becher Jensen
Terkel Kristian Tolstrup

LYNGBY 2003
EKSAMENSPROJEKT
NR. 19

IMM

Trykt af IMM, DTU

Forord

Denne rapport er et eksamensprojekt. Projektet er udarbejdet ved Institut for Informatik og Matematisk Modellering på Danmarks Tekniske Universitet.

Tue Becher Jensen	Terkel Kristian Tolstrup

Resume

Der findes allerede mange webbaserede systemer på Internettet. På trods af dette, eksisterer der ikke en udbredt metode til udvikling af webbaserede systemer, der kan hjælpe en udvikler til at producere pålidelige systemer, som det f.eks. er tilfældet ved konstruktion af oversættere. Med pålidelige systemer, menes systemer der garanterer browsersikre sider, navigationssikkerhed mv.

Vi giver en faseopdelt metode til konstruktion af pålidelige webbaserede systemer, herunder udvikling af begrebsdannelser, teorier og værktøjer. Under projektet har vi blandt andet udviklet en måde at formalisere brugerens interaktion med systemet. Der gives værktøjer der tillader hurtig udvikling, vedligeholdelse og dokumentation af pålidelige systemer. Metoden sikrer en stærk opdeling af funktionalitet, navigation og layout der tillader separat test.

KEYWORDS: Webbaseret system, udviklingsmetode, værktøjsbaseret udvikling, funktionssignaturer, database modellering, navigationsdiagram, kodegenerering.

Abstract

There are already many web based systems on the Internet. In spite of this there does not exist a widespread method for developing web based systems that will aid a developer create reliable systems, like it is the case e.g. for construction of compilers. By reliable systems is meant systems guaranteeing browsersafe pages, navigationsafety etc.

We present a step by step method for construction of reliable web based systems, including development of concept, theories and tools. During this project we have among other developed a way of formalizing the user interactions with the system. There will be presented tools allowing a fast development, maintenance and documentation of reliable systems. The method guarantee a strong separation of functionality, navigation and layout this allow separate testing.

KEYWORDS: Web based system, development method, tools based development, function signatures, database modelling, navigation diagram, code generation.

Indhold

Figurer

Kapitel 1

Indledning

Denne rapport omhandler udviklingen af webbaserede systemer. Der er udarbejdet teorier for udviklingen af et webbaseret system, og disse er samlet i en metode. Metoden giver retningslinier for udvikling af pålidelige webbaserede systemer og baserer sig i høj grad på en række værktøjer. Metoden er opdelt i en række faser, og for hver fase er der udviklet værktøjer der i stor udstrækning assisterer gennemførslen af de forskellige trin i fasen.

1.1 Problemstilling

Antallet af webbaserede systemer, der bliver udviklet, vokser hastigt, og sammen med antallet øges også kompleksiteten i systemerne. En struktureret udviklingsmetode og værktøjer til samme er derimod en mangel. Mange eksisterende webbaserede systemer er bygget op uden faste procedurer for vedligeholdelse. En stor del af disse er, grundet udokumenterede lappeløsninger og begrænset afprøvning, med tiden blevet meget uoverskuelige og fejlbehæftede. Dette betyder at vedligeholdelse og administration bliver unødigt besværligt. Resultatet er i sidste ende upålidelige webbaserede systemer. Problemet beskrives til tider som værende en reel web-krise [7].

Ved udvikling af webbaserede systemer anvendes sjældent en specifik udviklingsmetode. Dette kan undre, da udviklingsmetoder generelt anvendes mere og mere ved udviklingen af systemer, der ikke skal være webbaserede.

Ved udvikling af webbaserede systemer er det ofte minimering af tiden fra idé til produkt, der har højeste prioritet. Dette betyder, at analyse og design af systemet ofte springes over for at komme direkte til implementeringen. Systemer, der på denne måde er udviklet efter devisen “Udgiv hurtigt, ret fejl senere”, er ofte monolitiske i opbygningen og svære at få overblik over. En tilgangsvinkel hvor der benyttes et programmeringssprog med modulbegreb, ville lette vedligeholdelsen af systemet betragteligt.

1.2 Relateret arbejde

Vi vil i dette afsnit kort beskrive en række metoder og tilgangsvinkler, som tidligere er foreslået i forbindelse med ovenstående problemstilling. Der har været arbejdet i flere forskellige retninger.

Opbygning af konceptmodeller.

En retning er at analysere de eksisterende webbaserede systemer for at bygge en konceptmodel op, og derefter teste systemet i et testmiljø genereret ud fra konceptmodellen. For eksempel findes der to værktøjer **ReWeb** og **TestWeb**,der kan analysere og teste et webbaseret system [14]. Disse værktøjer giver dokumentation og testmiljø, men de er kun brugbare på eksisterende systemer og giver ingen direkte retningslinier for konstruktion af webbaserede systemer.

Udvidelse af eksisterende metoder.

En anden tilgang er at give en metode til udvikling af webbaserede systemer. Der findes flere eksempler på udvidelser af metoder til generel systemudvikling, så de passer bedre til webudwikling. Der er ændret på specifikke dele af metoden, men fundamentet er stadig generelt for mange typer systemer og ikke specifikt for webbaserede systemer. Flere har f.eks. forsøgt at udvide *Unified Modeling Language* (UML) med konstruktioner til design af navigation, heriblandt [16]. Det er ofte et problem at funktionaliteten og navigationen ikke skilles ad, da de så ikke kan implementeres og testes hver for sig. Dette resulterer i forringet vedligeholdelsesvenlighed.

Webforespørgselssprog.

Andre koncentrerer sig om udviklingen af webbaserede systemer for allerede eksisterende databaser. Ofte har det været med henblik på at skabe et forespørgselssprog, der kan præsentere semistruktureret data.

Der er to generationer af webforespørgselssprog [6], der kan lave forespørgsler på dokumenter i semistrukturerede databaser eller eksisterende webbaserede systemer:

Første generation af webforespørgselssprogene gør det muligt at lave forespørgsler der finder dokumenter der indeholder ønskede oplysninger. De tillader således ikke egentlig omstrukturering.

Anden generation af webforespørgselssprogene kaldes webmanipulationssprogene. De går videre i to væsentlige retninger.

- Det er muligt at udtrække specifikt indhold fra fundne dokumenter.
- Resultater af forespørgsler kan kombineres til nye dokumenter.

Fælles for begge generationer af webforespørgselssprog er at de giver god mulighed for at modellere navigationen i et system oven på informationen i en database. Ingen af dem tager dog stilling til implementering af funktionalitet og test heraf.

Udviklingsplatforme.

Vi vil her kort beskrive relevante egenskaber for en række udviklingsplatforme til webbaserede systemer. Valget af platform beskrives og begrundes i næste afsnit.

Flere platforme vidner om, at der er lagt stor vægt på hurtig implementering. Platforme som *PHP Hypertext Preprocesor* (PHP) og *Active Server Pages* (ASP) er sprog, der er målrettet til udvikling af webbaserede systemer. Det betyder, at der sammen med platformen kommer et omfattende funktionsbibliotek, der i vid udstrækning letter implementeringen af de webbaserede systemer. Implementeringen af det enkelte dokument foregår monolitisk da indhold, navigation og funktionalitet er blandet sammen i én fil. Derudover er der er ingen kontrol eller test af systemet fra platformens side.

Anvendes en objekt-orienteret metode til udviklingen, er det naturligt at anvende en platform, der understøtter objekter. Platformen *Java Server Pages* (JSP) giver alle mulighederne fra Java, og således også objektstrukturen.

Platformen <bigwig> er ligesom PHP og ASP udviklet specielt til webbaserede systemer [3]. Platformens styrke er, at der ved oversættelse af systemet, foretages en række tests, der sikrer en gyldig struktur af de enkelte dokumenter.

Status.

Alle de nævnte platforme og tilgangsvinkler tillader på forskellig vis opbygning af et webbaseret system. Ingen af dem giver dog en specifik opskrift på hvorledes et velstruktureret webbaseret system, kan opbygges fra bunden.

På samme vis er der ingen opskrift på hvorledes man kan opnå en fornuftig opdeling af navigation, funktionalitet og indhold.

Der mangler altså en egentlig *software engineering* disciplin for hvordan det skal gribes an, når der skal konstrueres et webbaseret system.

1.3 Formål

Vores formål med denne rapport er at komme med en værktøjsunderstøttet *software engineering* metode til konstruktion af pålidelige webbaserede systemer.

Med pålideligt menes at arkitekturen bl.a. sikrer at følgende egenskaber er opfyldt:

- Integritet sikres i database.
- Implementeringen af funktionelle krav foretages i typesikre omgivelser.
- Navigationssikkerhed — ingen døde links eller sider og ingen manglende argumenter.
- Browsersikre sider — standarder for HTML overholdes.
- Brugergruppers rettigheder sikres og overholdes.
- Typecheck af parametre.
- Alle valideringer er statiske, dvs. at de kan foretages før ibrugtagning af systemet.

Hovedtesen bag vores arbejde er at store dele af webbaserede systemer kan genereres ud fra specifikationer af:

1. funktionelle krav, der definerer den logik systemet indeholder.
2. navigationen, der definerer brugerinteraktion med systemet.

Disse to beskrivelsesformer er desuden tidligere vist at være velegnet som grundlag for dokumentation af webbaserede systemer, se [2] og [11].

Vi ønsker at opnå en arkitektur med en klar adskillelse af navigation, funktionalitet og indhold. Dette indebærer, at der skal benyttes et modulopdelt programdesign der muliggør separat afprøvning af de enkelte dele af systemet.

Endeligt ønsker vi at danne os et overblik over hvor langt man kan nå med udgangspunkt i de to beskrivelsesformer.

1.4 Oversigt

I denne rapport beskriver vi en metode til udvikling af pålidelige webbaserede systemer. En designer kan ved at benytte metoden hurtigt nå fra en specifikation af de påkrævede funktionaliteter til en prototype. Samtidig gives der værktøjer, der hjælper ved udviklingen, samt klare retningslinier for test og dokumentation af systemet.

Benyttes den foreslåede metode, vil resultatet være et webbaseret system bestående af: en relationsdatabase, et funktionsbibliotek og et navigationssystem. Systemet vil således være opsplittet i tre naturlige dele: indhold, funktionalitet og navigation. Hver af de tre dele kan testes adskilt. Layout af de enkelte sider foretages separat med *style sheets*.

Baseret på specifikation af de funktionelle krav og navigationen, genereres bl.a.:

- Forslag til databasedesign. (E/R-diagram)
- Implementering af databasedesign. (SQL-skema)
- Signatur for funktionsbibliotek.
- Prototypeimplementering af funktionsbibliotek.
- Konverteringsbibliotek.
- Serversider, der implementerer navigationen.
- Adgangsbegrænsninger på databasetabeller, funktioner og sider.

Der er gives værktøjer til at sikre at alle brugerkonfigurerbare dele af systemet er gyldige *før* de benyttes ved generering af det endelige system.

Valg af platform.

Vi vil her begrunde valget af platform, til implementering af de med metoden modellerede, webbaserede systemer.

Vi har valgt at benytte funktionssproget *Standard ML* (SML) [8] som platform for implementering af de funktionelle krav. Implementeringen af navigationen genereres automatisk i *ML Server Pages* (MSP) der gør brug af indlejret ML. ML er kendetegnet ved at være et funktionsbaseret sprog med meget stærke typer, et godt modulsystem og understøttelse af højereordens funktioner.

Da alle serversider genereres automatisk, vinder metoden ikke noget ved funktionsbibliotekerne i ASP eller PHP. Desuden tillader hverken ASP eller PHP en skarpt modulopdelt arkitektur.

JSP tillader fint en modulopdelt arkitektur baseret på objekter.

JSP har ikke et typebegreb, der er lige så stærkt som ML, og så fokuseres på objekter frem for funktionalitet. Da vores metode tager udgangspunkt i en specifikation af systemets funktionalitet, er det mest naturligt at anvende et funktionsbaseret sprog. Ved at anvende et funktionsbaseret sprog fås et håndtag til at udnytte specifikationen af systemets funktionalitet. Da ML understøtter sammensatte typer kan specifikationen af systemets funktionalitet næsten benyttes direkte ved implementering af funktionsbiblioteket.

<bigwig> validerer struktur og indhold af de enkelte dokumenter, men ikke sammenhængen mellem flere dokumenter i et webbaseret system. Vores metode vil automatisk sikre de egenskaber <bigvig> kan sikre. Derudover har ML igen en fordel med hensyn til stærke typer og et godt modulsystem.

Rapportens struktur.

Rapporten vil hovedsageligt gennemgå vores metode til udvikling af webbaserede systemer. Undervejs vil vi beskrive de værktøjer, der er udviklet.

I kapitel 2 gives en generel beskrivelse af de elementer der indgår i arkitekturen for webbaserede systemer. Vi afslutter kapitlet med en beskrivelse af den arkitektur der benyttes i vores metode.

I kapitel 3 introduceres begreberne *linkelementer*, *statiske elementer* og *dynamiske elementer*.

I kapitel 4 gives en oversigt over metoden. Skridt for skridt gennemgås et eksempel for udviklingen af et system.

I kapitel 5 beskrives hvorledes de funktionelle krav kan udtrykkes ved specifikation af brugergrupper, funktionssignaturer og typer.

I kapitel 6 vises hvordan et databasedesign kan afledes på baggrund af en signatur og hvordan et databaseskema kan genereres på baggrund af et design. Et led i konstruktionen af databasedesignet er en normalisering der bevirker at E/R-diagrammet kan blive væsentligt større end signaturen.

I kapitel 7 indføres begrebet navigationsdiagram som en model af brugerens interaktion med systemet. Bestanddelene i et navigationsdiagram er begrænset af signaturen. Dette giver en klar fordel, da vi så formelt kan beskrive ting som type- og navigationsikkerhed. Da navigationsdiagrammet i stor udstrækning skal defineres af en designer, gives et værktøj der kan sikre at de formelt beskrevne egenskaber er overholdt.

I kapitel 8 beskrives den kode der kan genereres på baggrund af signaturer og navigation. Dette inkluderer hvorledes et navigationsdiagram kan benyttes til at generere en implementation af navigationen, hvorledes der kan genereres en prototypeimplementering af de funktionelle krav samt hvorledes et konverteringsbibliotek kan opbygges.

I kapitel 9 beskrives hvordan de enkelte dele af et genereret webbaseret system kan testes. Dette inkluderer separat test af funktionsbibliotek, navigation og layout. Opsplitningen er vigtig da den "rå" fuktionalitet skal testes for korrekthed, mens de andre dele af systemet skal testes for brugervenlighed.

I kapitel 10 beskrives hvorledes de grundlæggende specifikationer af funktionelle krav og navigation, kan benyttes som grundlag for dokumentation af systemet.

I kapitel 11 beskrives en række idéer til udvidelse af metoden, nye værktøjer og alternative tilgangsvinkler.

I kapitel 12 afrundes rapporten med en konklusion på projektet.

Forudsætninger for at læse rapporten.

Kapitel 4 indeholder en oversigt over de forskellige faser i metoden, og i kapitlerne 5 – 8 gives en beskrivelse af de enkelte skridt i metoden. Læseren vil her have behov for generelt kendskab til HTML, programmering i ML samt databasemodellering, herunder E/R-diagrammer.

Kapitel 2

Webbaserede systemers arkitektur

I dette kapitel gives en generel beskrivelse af de elementer der indgår i arkitekturen for et webbaseret system. Kapitlet afsluttes med et afsnit der beskriver den arkitektur der benyttes i vores metode.

Når man taler om webbaserede systemer, menes der systemer, der på applikationsniveau er distribuerede. Det vil sige, at de kan gøre information tilgængelig, uafhængigt af brugerens fysiske placering og platform. Dette under antagelse af at forud bestemte protokoller anvendes til kommunikationen.

Internettet er i dag det største og mest udbredte netværk. Det er således mest naturligt at anvende dette som grundlag for et webbaseret system. Webbaserede systemer udvikles til den del af internettet, der kaldes *World Wide Web* (WWW). På WWW benyttes protokollen HTTP til kommunikation. Vi vil i det følgende se på konsekvenserne af, at WWW benyttes til konstruktion af webbaserede systemer. Alle større platforme har applikationer til kommunikation via HTTP. Protokollen HTTP benyttes også på mange andre netværk, f.eks. intranet.

2.1 Terminologi

I det følgende fastslås terminologien bag en række essentielle ord, der benyttes gennem rapporten.

Hypertekst

En tekst der indeholder links (hyperlinks) til anden tekst i samme dokument eller andre dokumenter.

> "Let me introduce the word hypertext to mean a body of written or pictorial material interconnected in such a complex way that it could not conveniently be presented or represented on paper" –Ted Nelson, [12].

Hyperlinks

En metode til at forbinde et dokument til et andet. Hyperlinks skaber forbindelser mellem hypertekstdokumenter, så det, ved at "springe" mellem dokumenterne, virker som et samlet dokument.

Klient

Et program der sender forespørgsler til en server, og modtager svar fra serveren. En web-browser er et eksempel på en klient.

Server

Et program der modtager forespørgsler fra en klient, for derefter at udføre den forespurgte opgave (evt. via andre programmer), for så til sidst at returnere resultatet af forespørgslen. En web-server er et eksempel på en server, der returnerer forespurgte hypertekstdokumenter til en web-browser.

Web-browser

En web-browser er en klient, der kan tolke og vise et hypertekstdokument. En bruger kan benytte en web-browser til at manøvrere rundt i et webbaseret system. Web-browseren sørger for at sende forespørgsler til de nødvendige servere, når brugeren vil følge et hyperlink.

2.2 HTTP

I dette afsnit beskrives begrænsninger som følge af den protokol der ligger til grund for komunikation på WWW.

Protokollen *Hypertext Transfer Protocol* (HTTP) anvendes på WWW til overførsel af hypertekst mellem klient og server. Ifølge protokollen skal klienten sende en forespørgsel til serveren, der angiver, hvor hypertekstdokumentet kan findes. Placeringen af hypertekstdokumenter angives med en *Uniform Resource Identifier* (URI).

Da HTTP definerer de regler, som det webbaserede system skal kommunikere i henhold til, skal det webbaserde system anvende URIs til at linke hypertekstdokumenter sammen.

Reglerne for kommunikation via HTTP betyder at serveren ikke kan kommunikere frit med klienten, men må overholde reglerne for forespørgsel på og returnering af hypertekstdokumenter.

2.3 Sider med fast indhold

I dette afsnit beskrives hvordan fast indhold i et hypertekstdokument kan formateres.

Hypertext Markup Language (HTML) indeholder kontruktioner til at udtrykke formatering, billeder og links i hypertekstdokumenter. HTML er *de facto*-standarden for publicering af dokumenter på WWW. Da HTML til stadighed udvikler sig er der lavet en række forskellige versioner af standarden. Hypertekstdokumenter defineret i HTML kan anvende samme elementer som en trykt bog. Det vil sige formateret tekst og billeder. Derudover er det muligt at anvende lyd, videosekvenser og links i dokumentet.

HTML-formaterede hypertekster kan anvende disse elementer til at give overblik og indblik, på samme måde som en figur eller et diagram kan i en bog. Men trods de tilføjede dimensioner i visualiseringselementer, er dokumentet stadig statisk som en bog.

2.4 Sider med skiftende indhold

I dette afsnit beskrives hvorledes tolkning kan benyttes i forbindelse med hypertekstdokumenter.

Normalt ønskes det, at webbaserede systemer kan andet end forevise sider med fast indhold. Ofte er man interesseret i at få udført forskellige funktioner baseret på brugerens interaktion med systemet. Da HTML ikke indeholder nogle dynamiske konstruktorer, er det nødvendigt at anvende et andet sprog, til at udtrykke de dele man ønsker skal være dynamiske. Der er to steder, hvor sådan et sprog kan tolkes, enten på klienten, eller på serveren, der kan tolke sproget og sende resultatet til klienten.

2.4.1 Klient-side tolkning

HTML indeholder elementer, der kan angive, at dele af dokumentet skal tolkes som et andet sprog. En klient, der understøtter sproget, vil så tolke de angivne dele lokalt uden at involvere en server.

De fleste nyere web-browsere kan tolke klient-side sprog. Dog kan der opstå problemer, da ikke alle browsere tolker samme sprog ens eller kræver, at der benyttes specielle dialekter af sprogene.

Fordele ved klient-side tolkning i forbindelse med brugerinteraktion:

- Øget hastighed, da behovet for transmission over netværket mindskes.
- Mindsket arbejdsbyrde på server.

Klient-side sprog er således gode til effektivt at behandle simpel brugerinteraktion. De er ikke på samme måde som servere bundet til kun at udføre funktioner når der kommer en forespørgsel fra en klient. Klient-side sprog kan derimod i vid udstrækning reagere på hændelser som et almindeligt program. Dette kan give effekten af et reaktionsmønster fra et system, der ikke er distribueret.

Da kildekoden til de tolkede dele af dokumentet skal være tilgængelig hos klienten, kan man ikke benytte klient-side tolkning til at håndtere følsomme oplysninger. Hovedsageligt benyttes klient-side tolkning til at sikre, at brugeren har angivet gyldige parametre der kan sendes til serveren hvor det benyttes i en server-side tolkning. Brugere kan omgå klient-side tolkning. Det er altså ikke nok kun at validere parametre på klienten.

Et eksempel på et sprog, der tolkes af alle større web-browsere, er Java Script.

2.4.2 Server-side tolkning

Tolkes et sprog af serveren, er systemet uafhængig af, hvilke sprog klienten understøtter. Kravet er så, at resultatet af server-side tolkningen er et HTML-dokument, der kan sendes til klienten. Der er ikke et tilsvarende krav om at der skal genereres HTML, hvis der tolkes på klient-siden.

Fordele ved server-side tolkning:

- Klienter modtager altid serverens tolkning.
- Den tolkede kildekode hemmeligholdes for brugere.
- Alt efter sprog kan den tolkede kode modulopbygges.

Server-side tolkning er en sikkerhedsmæssig nødvendighed hvis systemet skal behandle følsomme oplysninger. Da den tolkede tekst aldrig forlader serveren, kan det f.eks. tillades, at den indeholder loginoplysninger til en database, uden at sikkerheden for databasen af den grund er kompromitteret.

2.5 Parameteroverførsel

I dette afsnit beskrives hvordan parametre kan overføres fra brugeren til systemet.

For at det webbaserede system kan udføre handlinger baseret på brugerens handlinger, skal der være en grænseflade, der gør det muligt at overføre informationer fra brugeren til systemet. Tolkes de dynamiske dele på klienten, kan dette direkte foranledige mange forskellige handlinger, præcis som ved ikke distribuerede applikationer.
Hvis det derimod er serveren, der skal stå for fortolkningen, er der behov for at overføre informationer fra brugeren til serveren. Protokollen *Common Gateway Interface* (CGI) definerer en sådan grænseflade.

CGI-protokollen gør det muligt for serveren at modtage parametre sammen med en URI. Serverens fortolker har adgang til disse CGI-parametre, og de kan så benyttes i den tekst, der fortolkes. Det betyder, at det er op til klienten at sende de parametre, der skal vedlægges en forespørgsel efter et dokument, der fortolkes på serveren. Dette er altså ikke som i andre applikationer, hvor parametrene til interaktionerne er givet ved implementeringen. Det er altså op til den serverfortolkede tekst at sikre, at de ønskede parametre er til rådighed.

2.6 Lager

I dette afsnit beskrives hvordan informationer kan lagres i systemet.

Webbaserede systemer, der gør brug af server-side tolkning, skal ofte bruge et lager til informationer, der skal gemmes for senere brug.

Der er mange muligheder for valget af sådan et lagermedie. Da mængden af information og antallet af forespørgsler erfaringsmæssigt hurtigt kan eskalere, er det vigtig at benytte en arkitektur, der kan håndtere arbejdsbyrden.

Et *Database Management System* (DBMS) er ofte et fornuftigt valg. De kan normalt skaleres til at håndtere endog meget store mængder information. Til små systemer kan det virke voldsomt, men ulemperne opvejes af fordelene ved opskalering og portering.

Mange af de sprog, der benyttes til implementering af dynamiske dele af hypertekstdokumenter, har metoder til at kommunikere med en databaseserver.

Af sikkerhedsmæssige hensyn er det ofte en fordel at adskille webserver og databaseserver, så det kun er webserveren der kan kontaktes via internettet. Gøres dette er server-side tolkning nødvendigt, da fortolkeren så kan tilgå databaseserveren via det interne netværk.

2.7 Andre servere

I dette afsnit beskrives adgangen til andre servere end web-serveren og databaseserveren.

Webbaserede systemer kan drage nytte af andre servere på lige fod med en databaseserver. Det kan f.eks. være adgang til et internt økonomisystem, eller eksterne servere som transaktionsservere, eller f.eks. en autentifikationsserver, der validerer identifikation af brugere.

I mange tilfælde er det en fordel at benytte en ekstern server, som både brugeren og det webbaserede system stoler på. For eksempel kan man anvende en transaktionsserver hos et pengeinstitut til betalinger. På den måde er det troværdigheden af pengeinstitutet, der skal bære transaktionen.

2.8 Tilstandsvariable

I dette afsnit beskrives mulighederne for at benytte tilstandsvariable på webbaserede systemer.

For at skabe en effekt af at hypertekstdokumenterne hænger sammen som et webbaseret system, ønsker man ofte at bruge en form for tilstandsvariable. Dette kan gøres med CGI-parametre, men *Cookies* og sessioner kan være nemmere at arbejde med. *Cookies* og sessioner kan anbringes på klientens maskine, så de er linkuafhængige.

2.9 Sikkerhed

I dette afsnit beskrives kort de overvejelser der skal gøres under udvikling af webbaserede systemer i forbindelse med sikkerhed.

Det er vigtigt, at sikkerhedsaspekter betragtes i forbindelse med udvikling af webbaserede systemer. Sådanne systemer indeholder ofte følsomme oplysninger, der ikke må kompromitteres.

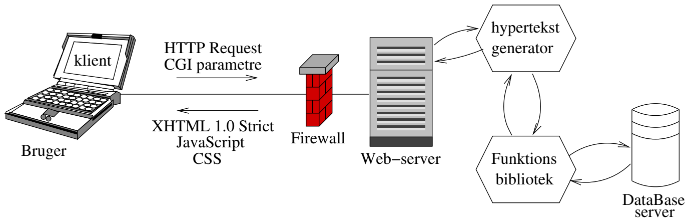
Arkitekturen benyttet til webbaserede systemer gør, at det er vigtigt, at al kommunikation med brugeren kun kan foregå gennem webserveren. Et webbaseret system kan gøre brug af lokale servere. Disse kan blokeres for omgivelserne med en *firewall*. På denne måde kan informationer beskyttes mod tilgang fra andet end de tiltænkte dele af det webbaserde system.

Når et webbaseret system udvikles, er det vigtigt at have i tankerne, at følsomme oplysninger, ikke skal kunne komme uden for den beskyttende *firewall*. Det betyder, at de følsomme oplysninger ikke på noget tidspunkt må sendes til klienten. Hverken i form af hypertekst eller klient-tolkede tekster, da man ikke kan være sikker på håndteringen. Det samme glæder for *Cookies*.

2.10 Den brugte arkitektur

Dette afsnit beskriver den arkitektur der benyttes til webbaserede systemer lavet efter vores metode.

I figur 2.1 illustreres arkitekturen af et webbaseret system som det vil være uden brug af eksterne servere. Klient og web-server kom-



Figur 2.1: Standard systemarkitektur for systemer genereret via vores metode.

munikerer via HTTP protokollen på WWW. Parametre til serveren overføres fra klienten i form af CGI parametre.

Web-serveren returnerer altid HTML i den version der kaldes XHTML 1.0 Strict til klienten. Denne version af HTML er blandt de nyeste, og den ligger op til en opsplitning af indhold og layout. Layout foretages separat ved brug af *Cascading Style Sheets* (CSS).

Sider der tillader at en bruger kan angive parametre der skal sendes til serveren, benytter klient-side tolkning til typevalidering af de angivne parametre. Sproget der benyttes er Java Script, da det kan tolkes af næsten alle nutidige web-browsere.

Alle sider i systemet genereres via tolkning af MSP server-sider. Da MSP tillader men ikke kræver at der skal foretages egentlig tolkning, vil det ikke have indflydelse på sider uden skiftende indhold. Server-side tolkning benyttes bl.a. til databaseadgang og validering af parametre. Sidstnævnte er er nødvendigt da klient-side validering alene, jf. afsnit 2.4.1 ikke er tilstrækkeligt. Da vi har valgt at benytte

MSP til server-side tolkning, benyttes ML's modulsystem til at kalde implementeringen af de funktionelle krav i et funktionsbibliotek.

Som lager benyttes en relationsdatabase der understøtter SQL. Det er funktionsbiblioteket der står for at sende SQL-forespørgsler til databaseserveren.

Ved at benytte en *firewall* adskilles webserver og eventuelle andre interne servere som f.eks. databaseserveren, så det kun er webserveren der kan kontaktes via WWW.

Den benyttede arkitektur forhindrer ikke brug af andre servere, cookies og sessions. Hvis der benyttes en server (udover web-serveren) hvortil både klient og funktionsbibliotek skal have adgang, f.eks. i forbindelse med betaling, bør man sikre at serveren er adskildt fra resten af det webbaserede system med en *firewall*.

Kapitel 3

Karakteristiske træk for hypertekstdokumenter

I dette kapitel introduceres begreber for de forskellige bestanddele af et hypertekstdokument. Disse begreber vil i vid udstrækning blive benyttet gennem resten af rapporten.

3.1 Funktioner og webbaserede systemer

I dette afsnit introduceres begrebet *linkelementer*.

Hvor generelle applikationer kan påbegynde funktioner på mange forskellige grundlag, er webbaserede systemer bundet af, at HTTP kun virker via forespørgselsinteraktion fra brugeren. Det betyder, at hvis vi ønsker at benytte en funktion i det webbaserede system, skal det gøres med et link. Webserveren vil så modtage en forespørgsel, der gør det muligt for fortolkeren at eksekvere funktionen.

Funktionerne i et webbaseret system bliver derfor nødt til, at være tæt bundet til links. Navigationen er bestemmende for, om en funktionsapplikation kan finde sted, men navigationen er ikke bundet til funktionaliteten eller omvendt, de to kan udmærket skilles ad.

Funktionsapplikationer kan altså kun finde sted i forbindelse med et link. I det efterfølgende refereres elementer i hypertekstdokumenter, der linker dokumenter sammen som *linkelementer*.

3.2 Brugerinteraktion med webbaserede systemer

I dette afsnit introduceres begreberne *statiske elementer* og *dynamiske elementer*.

For at kunne foretage en funktionsapplikation, er der ofte brug for informationer fra brugeren. Forrige kapitel beskrev hvorledes disse kunne sendes fra brugeren til det webbaserede system, via CGI-parametre. For at gøre det muligt for brugeren at vide, hvilke parametre det webbaserede system forventer, er det nødvendigt med nogle elementer i hypertekstdokumenter, der bindes til et linkelement, så de kan angive argumenter til funktionerne.

3.2.1 Statiske elementer

CGI-parametre, der skal sendes til webserveren, kan komme fra flere kilder. En er at sætte indsætte en konstant værdi i hypertekstdokumentet, der så sendes tilbage til webserveren sammen med den næste forespørgsel. En anden mulighed er at lade brugeren angive den værdi, der skal sendes.

Set fra serveren, der modtager forespørgslerne, er der ingen forskel på fremgangsmåderne. Derfor kaldes de elementer, der repræsenterer de forskellige fremgangsmåder, for *statiske elementer*.

De *statiske elementer* kan være skjulte værdier, indtastningsfelter, radioknapper, *drop-down* valg, osv. De giver alle, bortset fra skjulte værdier, brugeren en mulighed for at angive eller vælge en parameter til systemet.

3.2.2 Dynamiske elementer

De eneste dele i systemet, der kan ændre sig, er resultater af funktionsapplikationer. Alt andet kan repræsenteres ved statiske elementer som beskrevet ovenfor, samt fast indhold som tekst og billeder. Derfor har vi valgt at kalde resultater af funktionsapplikationer for *dynamiske elementer*.

De dynamiske elementer kan indgå i to sammenhænge i hypertekstdokumentet. En er, at de kan indsættes i og fremvises sammen med resten af dokumentet. Et eksempel på dette kunne være resultatet af en funktion, der laver et opslag i databasen, der bliver forevist brugeren. Den anden sammenhæng, hvori dynamiske elementer kan anvendes, er ved at benytte informationerne i elementerne til at definere skjulte værdier i dokumenter, der sendes til klienterne. På denne måde kan linkelementer i et dokument hos brugeren være parametriseret på baggrund af funktionsapplikationer der udførtes da dokumentet blev tolket.

Kapitel 4

Metoden, en oversigt

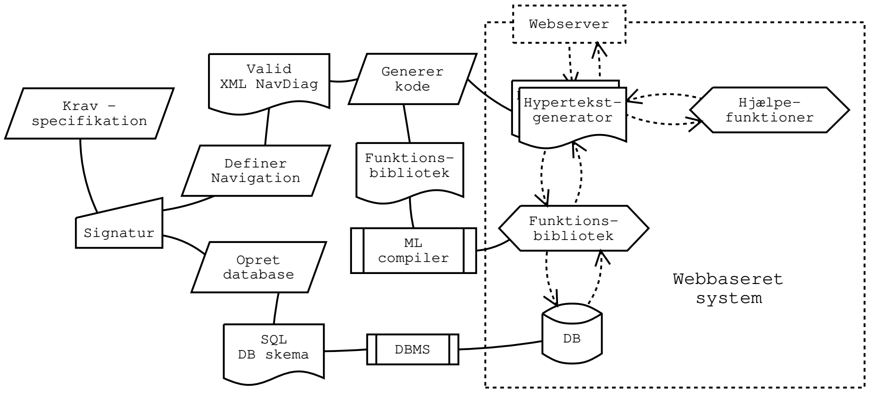
Vores metode til udviklingen af webbaserede systemer bygger på fire hovedfaser. Sammenhængen mellem hovedfaserne er vist i figur 4.1. Vi benytter os af diagrammer baseret på notationen for flowcharts. Et parallellogram illustrerer en hovedfase, en firkant med skrå top er et brugerdefineret input til et værktøj, firkanter med bølget bund er filer, firkanter med dobbelte sidekanter er værktøjer og sekskanter er oversatte ML-biblioteker.

Vores metode starter med fasen *kravspecifikation* hvor brugergrupper, funktionssignaturer og typer specificeres. Resultatet er en *signatur* der benyttes i faserne *opret database* og *definer navigation*.

I fasen *opret database* afledes et databasedesign på baggrund af *signaturen* og der genereres et databaseskema på baggrund af designet.

I fasen *definer navigation* definereres bruger interaktionen i et navigationsdiagram. Her benyttes *signaturen* til at begrænse de byggesten der må benyttes i diagrammet.

I fasen *generer kode* benyttes et navigationsdiagram til at generere kode der implementerer navigationen samt prototypekode der implementerer de funktionelle krav.



Figur 4.1: Oversigt over metoden.

Resultatet af udviklingen er et webbaseret system der er delt op i et hypertekstgenerator modul, der er genereret af et værktøj, et databasesystem, der ligeledes genereres af et værktøj og et funktionsbibliotek der er begrænset af en modul grænseflade der verificeres af ML oversætteren.

I dette kapitel vil vi gennemgå et eksempel på udviklingen af et webbaseret system efter vores metode. Undervejs vil vi illustrere hvorledes egenskaberne der betegner et pålideligt webbaseret system (jævnfør 1.3) er sikret af metodens faser.

Vi vil beskrive udviklingen af et webbaseret system for et universitet. Systemmet skal give studerende informationer om hvilke kurser der er knyttet til universitetets fagområder, f.eks. matematik, informatik osv. Studerende skal yderligere have mulighed for at få forevist informationer om de enkelte kurser. Det skal være muligt for lærerene ved universitetet at rette i informationerne om hvilke kurser der er knyttet til et fagområde.

4.1 Kravspecifikation

Specifikationen af kravene til vores webbaserede system skal centreres omkring funktionaliteten. Det essentielle i denne fase er at identificerer hovedfunktionerne i systemet og beskrive disse ved funktionssignaturer.

Vi har identificeret følgende funktioner:

```
showCurr [student, teacher] -
  Viser en beskrivelse af et fagområde.

showCourse [student, teacher] -
  Viser informationen knyttet til et kursus.

addCourse [teacher] -
  Tilføj et kursus til et fagområde.
```

Funktionerne er annoteret med de brugergrupper der har rettighed til at anvende funktionen.

Det er nødvendigt at nå en hierarkisk opdeling af brugergrupperne, fordi vi senere i metoden ønsker at kunne identificerer det laveste niveau en bruger må være på, når funktionen anvendes. Kender vi den laveste brugergrupper er det muligt at generaliserer funktionerne, og sikre brugerrestriktionerne ét sted, istedet for, for hver brugergruppe.

Da `student` kun findes i forbindelse med `teacher`, siges `teacher` at inkluderer rettighederne fra `student`. Den laveste brugergruppe for funktionerne `showCurr` og `showCourse` er derfor `student`, mens den er `teacher` for funktionen `addCourse`.

Vi specificerer funktionssignaturer for funktionerne:

```
showCurr   : Name            -> Curr   [student]
showCourse : CourseId        -> Course [student]

addCourse  : Name * CourseId -> unit   [teacher]
```

Hvor en signatur består af funktionens navn, typer for argumenter, type for resultat og en annotering af brugergruppe. Læg mærke til funktionen `addCourse` der har typen `unit` som resultat, det betyder at funktionen ikke har en værdi som resultat, men derimod en sidevirkning. I vores tilfælde er sidevirkningen, at der tilføjes et kursus til beskrivelsen af et fagområde i databasen.

Typeomgivelserne der bruges i funktionssignaturerne udspecificeres:

```
Name       = string
CourseId   = int
CourseDesc = string
Course     = CourseId * CourseDesc
Curr       = Name * Course list
```

Her er typerne `Name`, `CourseId` og `CourseDesc` erklæret ved simple typer (`int` er heltal og `string` er tekststrenge). `Course` og `Curr` er erklæret ved typekonstruktorer på de andre typeerklæringer.

Vi har nu en kravspecifikation for det webbaserede system vi ønsker at udvikle. Der er givet korte tekstuelle beskrivelser af funktionaliteten og funktionssignaturer der specificerer argumenter og resultateter for funktionerne, dette kan samles til en dokumentation for det udviklede system. Mere herom i kapitel 10.

De næste faser i metoden har funktionssignaturerne som grundlag for de værktøjer, der kan generere dele af det færdige system. Nærmere gennemgang af hvorledes kravspefikationen skal udarbejdes findes i kapitel 5.

4.2 Opret database

Når kravspecifikationen er på plads er næste fase at opnå et database design. Vi ønsker at nå et databasedesign der er beskrevet ved SQL-sætninger. For at nå dette mål normaliserer vi først typeomgivelserne. Dette er nødvendigt for at kunne generere et E/R-diagram.

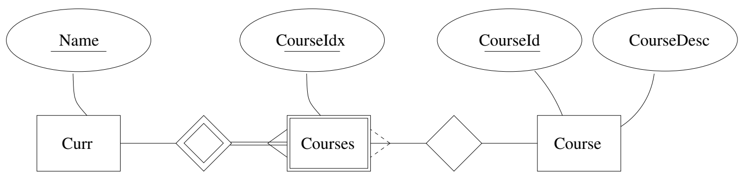
Det genererede E/R-diagram indeholder markeringer hvor der skal foretages en undersøgelse af relationen mellem to typer. Nøglerne i E/R-diagrammet genereres ud fra funktionssignaturerne.

Typeomgivelserne der blev specificeret i kravspecifikationen normaliseres:

```
Name       = string
CourseId   = int
CourseDesc = string
Course     = CourseId * CourseDesc
Courses    = Course list
Curr       = Name * Courses
```

Typen `Courses` introduceres for at alle typeerklæringer er på normaliseret form. Filen for funktionssignaturer og type omgivelser efter normaliseringen findes i appendiks A.1.

Værktøjet *ergen* kan ud fra de normaliserede typer generere et E/R-diagram med markeringer, se figur 4.2 (markeringer angives grafisk med stiplede linier).



Figur 4.2: Genereret E/R-diagram.

Læg mærke til at der ikke er en markering på den svage relation. Vores metode benytter kun svage entiteter til at repræsentere typeerklæringer indeholdende en `list`-konstruktion. En nøgle der angiver placeringen i listen føjes til de svage entiteter. En reference til en typeerklæring indeholdende list vil således altid være en svag *én-til-mange* relation.

Entiteter relateret til en svag entitet vil altid være annoteret *én*, da en *mange* annotering ville betyde at entiteten kunne have to værdier på samme plads i listen.

Markeringen på relationen mellem `Courses` og `Course` skal ikke ændres da et kursus kan forekomme på flere forskellige fagområder.

Nøglerne er genereret ud fra funktionssignaturerne. I vores eksempel skal der ikke tilføjes nogle nøgler.

SQL-sætninger kan genereres med værktøjet *sqlgen* :

```
CREATE TABLE Curr(
      Name TEXT PRIMARY KEY
);

CREATE TABLE Courses(
      CourseIdref INT REFERENCES Course(CourseId),
      CourseIdx INT PRIMARY KEY,
      Nameref TEXT PRIMARY KEY REFERENCES Curr(Name)
);

CREATE TABLE Course(
      CourseId INT PRIMARY KEY,
      CourseDesc TEXT
);


CREATE USER student;
CREATE USER teacher;

GRANT SELECT ON Curr, Courses TO student;
GRANT SELECT ON Courses, Course TO student;
GRANT INSERT, DELETE, UPDATE ON Courses, Curr, Course TO
    teacher;
```

Der er genereret en tabel for hver entitet. Alle relationer mellem tabeller er begrænset med referencer til de passende nøgler.

Værktøjet *sqlgen* genererer, ud over tabeller i database, også SQL-sætninger der opretter brugere og tildeler rettigheder til tabellerne. Dette gøres ud fra annoteringerne af brugergrupper på funktionssignaturerne. Læg mærke til at der ikke gives læseadgang (`SELECT`) på nogle tabeller til brugergruppen `teacher`, da denne indeholder brugerrettighederne fra `student` og anvender de samme implementeringer af funktionerne `showCurr` og `showCourse`.

Metoden for generering af E/R-diagram og SQL-sætninger beskrives nærmere i kapitel 6.

4.3 Definer navigation

Brugernes interaktion med det webbaserede system specificeres i et navigationsdiagram. Når navigationen er specificeret har vi udviklet værktøjer der kan sikre at de egenskaber vi har opstillet for et pålideligt webbaseret system overholdes.

Et navigationsdiagram skal angive en model for brugerens interaktion med systemet og indeholder en række elementer som beskrevet i kapitel 3. Navigationsdiagrammet er begrænset af funktionssignaturerne vi bestemte i kravspecifikationen. Det betyder at funktionssignaturerne angiver hvilke linkelementer der kan anvendes i diagrammet. Der kan kun forekomme dynamiske elementer, der er resultater af en funktion anvendt på et linkelement, der leder til siden.

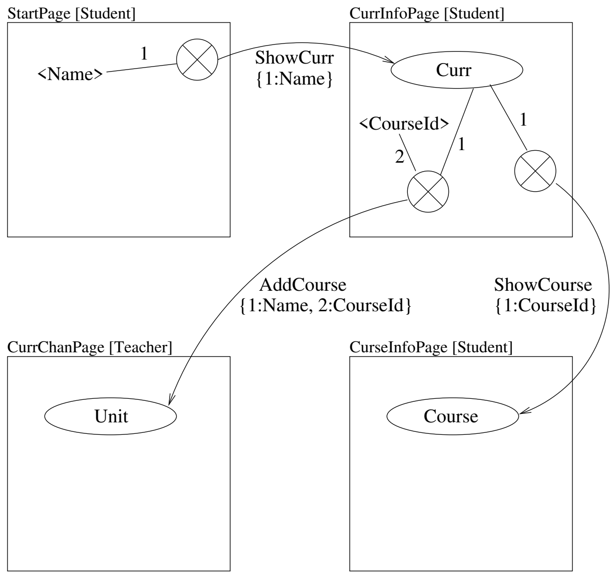
For vores eksempel er der funktionerne `showCurr`, `showCourse` og `addCourse`, der henholdvis giver anledning til dynamiske elementer med typerne `Curr`, `Course` og `unit`. Disse elementer kan anvendes når der skal findes argumenter til en efterfølgende funktionsapplikation. Hvis der ikke er en værdi af samme type som argumentet til en funktion, er det nødvendigt med et statisk element.

Navigationsdiagrammet i figur 4.3 angiver interaktionen med brugeren i vores webbaserede system.

I navigationsdiagrammet illustreres hvorledes resultatet af en funktion, kan benyttes som argumenter til funktioner på linkelementer i siden. På siden `CurrInfoPage` findes argumentet til `showCourse` i listen af `CourseId` som er indeholdt i resultatet af ShowCurr.

Værktøjet *CheckNavDiag* sikrer at egenskaber for navigationssikkerhed er overholdt i det specificerede navigationsdiagram. Det vil sige at der ikke er nogle døde links, at der er værdier tilstede for alle argumenter til funktioner, at der er dynamiske elementer for alle resultater af funktioner på links til siden og at alle sider er forbundet.

Kapitel 7 beskriver opbygningen af navigationsdiagrammer, formaliserer disse og beskriver verifikationen. Navigationsdiagrammet i figur 4.3 findes på XML-form i appendiks A.4.



Figur 4.3: Navigationsdiagram for universitetssystemet.

4.4 Generer kode

Ved implementeringen af det webbaserede system kan dele af koden genereres ud fra funktionssignaturerne og navigationsdiagrammet. Der genereres en signaturfil for funktionsbiblioteket, en prototype af funktionsbiblioteket og hele navigationsdelen af det endelige webbaserede system.

For at generere en signaturfil for funktionsbiblioteket, samt en prototype, anvendes værktøjet *ProjGen* . For vores eksempel giver det følgende prototypeimplementering:

```
(*automatically generated test implemetation*)
type Name    = string
type CourseId = int
type CourseDesc = string
type Course  = ( CourseId * CourseDesc )
type Curr    = ( Name * Course list )

fun ShowCurr (NameArg) = (NameArg, [(1, "string:CourseDesc"
     )] )
fun ShowCourse (CourseIdArg) = (CourseIdArg, "string:
    CourseDesc" )
fun AddCourse (NameArg, CourseIdArg) = ()
```

Alle funktionerne returnerer standard værdier der svarer til typen angivet i funktionssignaturen. Funktionsbiblioteket kan anvendes til at teste navigationen med. Funktionaliteten og navigationen er delt op af signaturfilen der angiver grænsefladen. Den genererede signaturfil findes i appendiks A.5.

Implementeringen af navigationsdelen af det webbaserede system foregår ved først at angive konfigurationen af elementerne fra navigationsdiagrammet. Et værktøj sikrer at de konfigurerbare filer overholder XHTML 1.0 Strict og at alle statiske elementer repræsenterer en værdi af den angivne type. Tilslut genereres koden for navigationen, indeholdende typecheck af CGI parametre, konvertering til grænsefladerne for bruger og funktionsbibliotek og funktionsapplikering.

Konfigurationsfilerne genereres af værktøjet *CreateSta* , for siden `CurrInfoPage`:

```
<sta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="" xsi:noNamespaceSchemaLocation="sta.xsd">
      <submit id="link2" destination="CurrChanPage">
            <input type="submit" value="CurrChanPage"/>
      </submit>
      <submit id="link3" destination="CourseInfoPage">
            <input type="submit" value="CourseInfoPage"/>
      </submit>
</sta>

<demandsta xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns="" xsi:noNamespaceSchemaLocation="demand
    .sta.xsd">
      <elem id="s2">
            <character type="CourseId">
                  <input type="text" name="s2"/>
            </character>
      </elem>
</demandsta>
```

I `<sta>` er linkelementerne angivet. Kode genereringen sørger for at finde værdier fra statiske og dynamiske elementer. Vi kan tilføje fremvisning af dynamisk genererede værdier eller statisk indhold i denne fil.

Repræsentationen af statiske elementer er angivet i `demandsta`. Standard er alle statiske felter indtastningsfelter, men de kan ændres til at benytte alle former for elementer til videresendelse af CGI parametre.

Værktøjet *CheckSta* foretager valideringen af konfigurationsfilerne. Valideringen foregår som beskrevet i 8.4.2.

Kodegenereringen indsætter de statiske konfigurationer i en hypertekstgenerator, der indeholder validering og konvertering af parametre. De genererede hypertekstgeneratorer for siderne findes i appendiks A.13 - A.16.

Det webbaserede system kan testes som beskrevet i 9, og dokumenteres som beskrevet i 10.

Som beskrevet kan vi nå en prototype direkte fra specifikationerne af funktionssignaturer og navigationsdiagram. Vejen fra kravspecifikation til implementering lettes af værktøjer. Det betyder at vi når en hurtig implementering. Ikke ved at springe kravspecifikation over, derimod ved at lægge vægt på denne og automatiserer de efterfølgende faser.

4.5 Andre værktøjer

For at lette arbejdet for designeren af det webbaserede system har vi udviklet følgende værktøjer:

sig2xml

Oversætter repræsentationen af en signaturfil til skelettet for et navigationsdiagram. Se kapitel 7.3.

tagmerger

Gør det muligt at inkluderer andre filer og konstanter i filerne med statiske informationer. Se kapitel 8.4.3.

distfiles

Kopierer hjælpefunktionsbiblioteket og klientsidecheck-biblioteket så de er tilgængelige for alle brugergrupper angivet i navigationsdiagrammet.

makefile

Kan konfigureres med hvilket bibliotek der skal indeholde det webbaserede system. Anvendes programmet *make* med parametren `buildall` bygges en prototype af det webbaserede system ud fra et navigationsdiagram. Prototypen kan anvendes ved test af navigationen. Ændres funktionsbiblioteket eller konfigurationsfilerne kan *make* anvendes med parametren `build` der bygger et webbaseret system uden at overskrive disse.

Kapitel 5

Kravspecifikation for webbaserede systemer

I dette kapitel beskrives hvorledes de funktionelle krav kan udtrykkes på en form der er kort og præcis, og som lægger op til de følgende faser i metoden, herunder dokumentation af systemet. Dette opnås ved specifikation af brugergrupper, funktionssignaturer og typer.

Når et system skal designes, er det som det første altid en god idé at fastslå og specificere kravene til systemet. Dette gælder naturligvis også for webbaserede systemer. Indsamling, klasificering og validering af krav kan f.eks. gøres som beskrevet i [18].

5.1 Identifikation af funktioner og brugergrupper

Vores metode er centreret omkring funktionaliteten af systemet. Når kravene til systemet er specificeret, skal de gennemgås og funktionerne udpeges. Hver funktion gives et sigende navn, og angives på formen:

```
Funktionsnavn -
  En beskrivelse af funktionaliteten
```

Beskrivelsen kan være på uformel tekstuel form og ligge tæt op ad kravspecifikationen. Det vigtige er at opdele kravspecifikationen efter funktionalitet da vores metode tager udgangspunkt i denne.

At vi starter med at udpege funktionaliteten, adskiller sig fundamentalt fra objektorienterede (OO) designmetoder, der f.eks. bygger systemet op omkring en konceptmodel [15]. Ved typisk OO design fastlægges funktionaliteten først efter at objektstukturen er designet. Altså bliver uddelegering af funktioner lavet på baggrund af implementeringsbetragtninger. Vi viser i kapitel 6, at der med udgangspunkt i en funktionel beskrivelse kan foretages en række implementeringsnære valg. I dette tilfælde udmunder det sig i et konceptuelt desig ved et E/R-diagram.

Vi vil senere vise hvordan metoden giver mulighed for separat test af funktionaliteten mod en given kravspecifikation.

Funktionsbeskrivelserne deles op efter brugerne af systemet. Hver funktionsbeskrivelse annoteres med en eller flere brugergrupper.

```
Funktionsnavn [brugergrupper] -
  En beskrivelse af funktionaliteten
```

Metoden vil senere sikre, at funktionen kun kan anvendes af de angivne brugergrupper, så det er vigtigt at overveje annoteringerne nøje.

5.1.1 Hierarkisk opdeling af brugergrupper

I dette afsnit beskrives hvordan brugergrupper der benyttes til rettighedsannoteringer opdeles i et hieraki.

Når brugergrupperne skal anvendes, ved implementering af databaseadgang i funktionsbiblioteket, er det nødvendigt at have en hierarkisk opdeling. Det betyder at de skal klassificeres i forhold til hinanden.

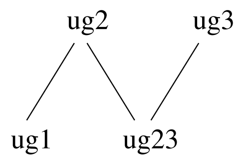
Når brugergrupperne skrives op for et webbaseret system, kan en gruppe angives at have alle rettigheder, en anden gruppe har. Man kan analysere funktionsbeskrivelserne efter følgende regler:

1. Hvis en brugergruppe altid er anvendt sammen med en anden brugergruppe, skal denne angives som værende indeholdt i den anden brugergruppe. For hver af de funktioner grupperne før havde rettigheder til, fjernes herefter den af de to brugergrupper, der inkluderer den anden.
2. Hvis to brugergrupper skal have rettighed til samme funktion, og den ene ikke kan indeholde den anden, skal en ekstra brugergruppe oprettes. Den nye brugergruppe skal så have rettighed til funktionen, og den skal være indeholdt af alle de brugergrupper, der oprindeligt skulle have rettighed til funktionen.

Eksempel:
Vi starter med følgende funktionsnavne og brugergrupper:

```
fun1 [ug1, ug2]
fun2 [ug2, ug3]
fun3 [ug3]
```

I figur 5.1 ses den hierarkiske opdeling der opnås ved at følge ovenstående regler. For funktionen `fun1`, benyttes regel nummer 1, da



Figur 5.1: Brugerhieraki.

brugergruppen `ug1` altid er anvendt sammen med brugergruppen `ug2`. For funktionen `fun2`, benyttes regel nummer 2, hvor en ny brugergruppe kaldet `ug23` oprettes. For funktionen `fun3` er der ingen ændringer. Samlet fås følgende brugergrupper som bund for funktionerne.

```
fun1 [ug1]
fun2 [ug23]
fun3 [ug3]
```

Hver funktion kan således annoteres med en entydig bund, for hvilken brugergruppe der har rettighed til at anvende funktionen.

5.2 Funktionssignaturer

Specifikationen af funktionelle krav foretages ved angivelse af funktionssignaturer. Dette er inspireret af de signaturer der anvendes i specifications sprog som VDM og RAISE, samt i funktionelle programerings sprog som SML og Haskell.

Funktionssignaturerne angives som:

$$f : \mathit{typeNavn}_1 * \cdots * \mathit{typeNavn}_n \rightarrow \mathit{typeNavn}$$

Her angiver f funktionens navn, blev som bestemt i funktionsspecifikationen. Argumenter til funktionen er en n-tuple, hvor det i-te argument har typen $\mathit{typeNavn}_i$, resultatet af funktionen har typen *typeNavn*. Typerne er abstrakte og beskrives indtil videre kun ved navne.

Funktionssignaturerne er begrænset, idet argumenterne kun må være type navne der refererer simple typer. Simple typer er beskrevet nedenfor. Resultatet må have en vilkårlig type.

Hvis en funktion enten ikke tager nogle argumenter, eller resulterer i en ren sideeffekt, kan den specielle type `unit` anvendes. En funktion der ikke giver et resultat vil således have en signatur af formen:

$$f : \mathit{typeNavn}_1 * \cdots * \mathit{typeNavn}_n \rightarrow \texttt{unit}$$

Det kunne f.eks. være en funktion der ændrer indholdet i databasen.

5.3 Specifikation af typer

Når der er specificeret funktionssignaturer for samtlige funktioner, skal typerne konkretiseres. Alle typenavne der er brugt i funktionssignaturer skal konkretiseres ved typeerklæringer.

De simple typer er `int`, `string` og `bool`, der angiver at der er tale om henholdsvis heltal, tekststrenge og sandheds værdier (sand eller falsk). Komplekse typer sammensættes med typekonstruktorerne for kartesiske produkter '`*`' og lister '`list`'.

En typeerklæring skrives på formen:

$$typeNavn = typeExpr$$

Hvor *typeNavn* er navnet på typen der erklæres, og *typeExpr* er et type udtryk der enten er en simpel eller kompleks type. Derfor kalder vi typeerklæringerne for simple og komplekse.

Tilsammen kaldes alle typeerklæringerne for typeomgivelsen. En typeomgivelse kan indholde typeerklæringer som f.eks.:

- $T_1 = \texttt{int}$, simpel typeerklæring,
- $T_2 = T_1\ \texttt{list}$, kompleks type med list konstruktoren,
- $T_3 = T_1 * T_2$, kompleks type med kartesisk produkt,
- $T_4 = T_1 * T_1\ \texttt{list}$, kompleks type med flere kontruktorer.

Følger man den beskrevne metode, har man et godt udgangspunkt til både dokumentation og udvikling af systemet. Dette gælder specielt for de følgende dele af design fasen, samt når der skal laves en funktionel test af det webbaserede system.

Kapitel 6

Databasedesign

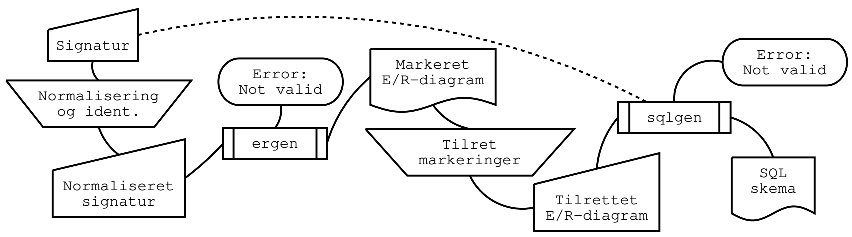
I dette kapitel beskrives hvorledes vores viden om funktioner, brugergrupper og datatyper kan anvendes til at give retningslinier for designet af databasen, der skal ligge til grund for det webbaserede system.

Vi vil vise, at det er muligt at generere en stor del af databasestrukturen efter klassificering af funktionssignaturer og normalisering af typeomgivelserne. De nødvendige normaliseringer bevirker at typeomgivelserne kan blive væsentligt større end de oprindeligt var. Normaliseringen er nødvendig for at opnå en form der er ækvivalent med komponenterne i et E/R-diagrammet.

Som indledning til kapitlet vil vi kort beskrive den process der fører til databasen. I de efterfølgende afsnit vil de enkelte trin blive forklaret nærmere, og vi vil forklare hvorledes databasens normalform kan verificeres.

Databasen designes efter metoden illustreret i figur 6.1.

Udgangspunktet for databasegenereringen er en *signatur* der indeholder specifikation af brugergrupper, funktionssignaturer og typer på en form, som beskrevet i kapitel 5. I første skridt, *normalisering og*



Figur 6.1: Fremgangsmåde for at generere databasen.

ident., identificeres front-end funktioner og typeomgivelserne normaliseres. Resultatet er en *normaliseret signatur*, som af værktøjet *ergen* benyttes til at generere et *markeret E/R-diagram*. Markeringerne i E/R-diagrammet står hvor der er usikkerhed angående relationers kardinalitet. I næste skridt, *tilret markeringer*, skal markeringer samt nøgler verificeres og eventuelt tilrettes af designeren, hvilket resulterer i et *tilrettet E/R-diagram*. Til sidst genereres, på baggrund af det tilrettede E/R-diagram, et *SQL-skema* der indeholder de SQL-udtryk, der skal til for at oprette databasen i et passende DBMS.

6.1 Identifikation af front-end funktioner

For at kunne generere et E/R-diagram bliver vi nødt til at identificere de funktioner der fortæller noget om opbygningen. Funktioner der kun fungerer som en interaktionsflade mellem brugeren og databasen, kaldes *front-end* funktioner.

Eksempler på front-end funktioner er funktioner, der slår op i en tabel og henter en tupel, eller funktioner der indsætter eller sletter tupler i en tabel. Dette uden at manipulere de involverede informationer.

Det gælder for front-end funktioner, at deres funktionelle afhængiheder kan overføres dirrekte til relationelle afhængigheder for attributterne i databasen. Dette benyttes som beskrevet i afsnit 6.3.2 til identifikation af nøgler i E/R-diagrammet.

6.2 Normalisering af typeerklæringer

Normalisering af typer er nødvendigt da frit sammensatte typeudtryk, kan gøres væsentligt mere kompakte end repræsentationen er mulig i et tilsvarrende E/R-diagram.

Typeomgivelserne skal altså være på en normaliseret form, for at de efterfølgende skridt er mulige. Det betyder at typeerklæringerne skal omskrives til en mere begrænset form. I den begrænsede form er følgende typeudtryk tilladt:

- simple typer (`int`, `string` og `bool`)
- typeudtryk på formen: $typeNavn$ `list`
- typeudtryk på formen: $typeNavn_1 * \cdots * typeNavn_n$

Det betyder, at normaliserede typeudtryk ikke kan kombinere flere komplekse kontruktorer i en typeerklæring. Samme virkning kan dog opnåes ved at introducere nye typer, der kan forbinde de forskellige dele af typeudtrykket. Har man et typeudtryk, der indeholder mere end en kompleks typekonstruktor, skal det normaliseres på følgende måde:

$T_1 = \texttt{int}$
$T_4 = T_1 * \ldots * T_1\ \texttt{list}$

Vi normaliserer ved at introducere en ny type for udtrykket T_1 `list`:

$T_1 = \texttt{int}$
$T_1' = T_1\ \texttt{list}$
$T_4 = T_1 * \ldots * T_1'$

Ligeledes kan komplekse typer, hvor der er anvendt kartesiske produkter af kartesiske produkter, også splittes op. Ved denne normaliseringsprocess vil der på tilsvarende vis opstå flere typer.

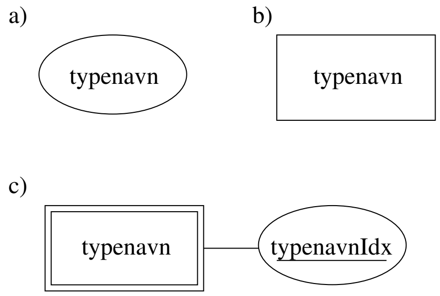
Læg mærke til at en typeerklæring på formen
$typeNavn = typeNavn'$, ikke kan finde sted i de normaliserede typeomgivelser. Typen $typeNavn$ kan fjernes og alle steder, hvor den anvendes, kan man i stedet anvende typen $typeNavn'$

6.3 Automatisk generering af E/R-diagram

Efter vores normalisering af datatyperne er vi i en situation, hvor hver datatype er beskrevet ved en simpel type (`int`, `string`, `bool`), et produkt af andre datatyper eller en liste af en anden datatype. En database kan nu designes ud fra datatyperne. Vi starter med et tomt E/R-diagram (Entity/Relationship-diagram), hvor der genereres elementer for hver type i typeomgivelserne.

Følgende retningslinier er givet til generering af et E/R-diagram ud fra typeomgivelserne:

- Simple typeerklæringer bliver til attributer, se figur 6.2a.
- Komplekse typeerklæringer, hvor konstruktoren for kartesisk produkt er benyttet, bliver til entiteter, se figur 6.2b.
- Komplekse typeerklæringer, hvor `list`-konstruktoren er brugt, bliver til svage (*weak*) entiteter. Ydermere forbindes en attribut, der navngives som *indeks* for listen, denne attribut markeres som nøgle, se figur 6.2c.



Figur 6.2: Elementer i E/R-diagram. a) attribut, b) entitet, c) svag entitet med indeks attribut.

Idéen bag retningslinierne er, at kartesiske produkter svarer til tupler i databasen. Ethvert element i et kartesisk produkt er enten navnet på en simpel eller kompleks type. De simple typer svarer til fields i databasen, og de komplekse typeerklæringer svarer til relationer til andre tabeller.

Komplekse typeerklæringer, der indeholder `list`, er mere komplicerede i databasesammenhæng, da der ikke på samme måde eksisterer en ækvivalent repræsentation i databasen. En liste er en datastruktur indeholdende en række elementer af samme type, hvor rækkefølgen af elementerne har betydning. Derfor opfatter vi listen som en entitet, der har en relation til entiteten eller attributen, for den refererede type, samt en indeks attribut, der angiver det refererede elements placering i listen.

Hvis en typeerklæring indeholder konstruktoren `list`, og typen ikke er refereret i en anden typeerklæring, skal entiteten ikke være svag. Det skyldes, at listen ikke er forbundet til en anden værdi, men derimod "global". Ellers skal entiteten være svag, da listen er en del af den type, der refererer den. Antag at to værdier indeholder en liste, disse lister er adskildt og har hver deres første element, andet element osv.

Relationerne i E/R-diagrammet genereres ud fra de typeerklæringer der er komplekse. Hvis en typeerklæring indeholder typenavnet på en anden type, er entiteten forbundet til elementet for den anden type i E/R-diagrammet.

Typeerklæringen indeholder ikke nødvendigvis tilstrækkelig information til at fastslå alle relationers kardinalitet. Hvor der kan være tvivl, indsættes markeringer. Når en relation er markeret, er det således et fingerpeg til designeren om, at relationen skal konkretiseres manuelt. En markeret *mange-til-en* kardinalitet skal muligvis ændres til *en-til-en*.

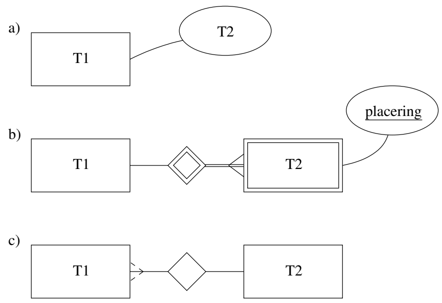
Der er to tilfælde for erklæringer, der kan generere en relation:

- $typeNavn_1 = typeNavn_2$ `list`
- $typeName_1 = \cdots * \ typeName_2 \ * \cdots$

Hver relation repræsenteres i E/R-diagrammet, som følger, hvor den refererede type er $typeNavn_2$:

- Hvis den refererede type er simpel, forbindes attributen for typen direkte til entiteten, se figur 6.3a.
- Hvis den refererede type er erklæret med en `list`, er der jævnfør ovenstående oprettet en svag (*weak*) entitet. Derfor skal der anvendes en svag relation. Disse skal være *én-til-mange*, da hver liste er bundet til den refererende type, se figur 6.3b.
- Hvis den refererede type er erklæret med et kartesisk produkt, forbindes entiteterne med en markeret *mange-til-én* relation, se figur 6.3c.

  Grunden til at typeerklæringer med kontruktoren `list` medfører en markeret *mange-til-én* relation, er at der er en attribut for placeringen i listen, og der ikke kan være to elementer på samme plads i en liste.



Figur 6.3: Relationer i E/R-diagram. a) entitet - attribut, b) svag relation, c) markeret relation.

6.3.1 Velformet E/R-diagram

Følges ovenfor beskrevne metode, vil man altid generere et velformet E/R-diagram. I den forstand at følgende ikke kan forekomme:

Attribut - Attribut Dette tilfælde kan ikke genereres af vores metode, da attributer kun genereres af typeerklæringer af simpel form (int, string, ...). Disse kan ikke indeholde referencer til andre typer. Derfor kan en attribut aldrig blive forbundet til en anden attribut.

Entitet - Entitet To entiteter kan aldrig blive forbundet uden brug af en relation. I vores metode indeholder alle regler for at forbinde entiteter, relationer.

Svag relation mellem to entiteter Kan ikke forekomme da svage relationer kun laves, når den refererede type indeholder en `list`, og kun når denne er repræsenteret som en svag entitet.

6.3.2 Nøgler

Ud fra de funktioner, der er bestemt som front-end funktioner, kan man drage nogle funktionelle afhængigheder imellem attributer i E/R-diagrammet. Således kan vi give nogle retningslinier for hvilke attributer, der er nøgler for en entitet.

For hver funktionssignatur analyseres argumenter og resultatet. Da alle argumenter til en funktion er simple typer (se kapitel 5), må de være repræsenteret ved attributer i E/R-diagrammet. Det er ikke påkrævet, at resultatet af funktionen skal være en simpel type. Hvis det er en kompleks type, er der en entitet i E/R-diagrammet, der repræsenterer typen.

Da der kun analyseres front-end funktioner, må typen for resultatet være afhængig af argumenttyperne. Resultattypen kan opfattes som mængden af alle attributerne, der er forbundet til entiteten. Der er altså en funktionel afhængighed mellem de attributer, der repræsenterer argumenter, og alle attributerne, der er forbundet til entiteten, der repræsenterer resultatet.

Således vil nøgler til en entitet der er fremkommet via resultattypen af en funktion, være de attributter, der er fremkommet via funktionens argumenttyper, som er forbundet til entiteten.

Nøglerne markeres i E/R-diagrammet, men de kræver eftersyn, da alle entiteter skal have en nøgle. Grunden til. at der er entiteter, som metoden ikke kan finde en nøgle for, kan være, at der ikke er nok front-end funktioner.

6.4 Tilrettelse af markerede relationer

De relationer, der markeres, når metoden anvendes til at generere et E/R-diagram, skal kontrolleres af designeren. Et eksempel med en typeerklæring baseret på et kartesisk produkt er:

$typeNavn_1 = typeNavn_2 * typeNavn_3$

Her refererer $typeNavn_1$ til $typeNavn_2$ og $typeNavn_3$. Antager vi at $typeNavn_2$ er erklæret ved en kompleks typeerklæring, vil der skabes en markeret *mange-til-én* relation mellem $typeNavn_1$ og $typeNavn_2$. Vi er sikre på at $typeNavn_2$ indgår i relationen med annoteringen *én*.

Der eksisterer altså *én* værdi af typen $typeNavn_2$ for hver værdi af typen $typeNavn_1$. Det er op til designeren på baggrund af kendskab til anvendelsen, at bestemme, om der er *én* eller *mange* værdi(er) af typen $typeNavn_1$ for hver værdi af typen $typeNavn_2$.

Alle nøgler i E/R-diagrammet skal kontrolleres, og der skal identificeres en nøgle for de tabeller, der ikke har en.

6.5 Automatisk generering af SQL-skemaer

Når E/R-diagrammet er tilrettet af designeren, kan der automatisk genereres SQL-sætninger, der kan oprette tabeller i databasen. Vi vil vise at med udgangspunkt i et tilrettet E/R-diagram og funktionssignaturerne, kan følgende genereres i SQL-sætninger:

- Databaseskemaer
- Begrænsninger der sikrer referentiel integritet
- Brugergrupper for databasen
- Adgangsbegrænsninger på databasen

Databaseskema.

Første skridt er databaseskemaet, der kan genereres ud fra E/R-diagrammet. Vi anvender en metode, der er beskrevet i [19]:

- For hver (ikke svag) entitet i E/R-diagrammet dannes en tabel i databaseskemaet. Alle attributer forbundet til entiteten repræsenteres som felter i tabellen.
- For hver *én-til-x* relation i E/R-diagrammet anbringes nøglen for entiteten, der indgår i relationen med *én*, i tabellen for entiteten, der indgår med x.
- For hver *mange-til-mange* relation i E/R-diagrammet, dannes en tabel i databasen, med nøglerne fra de to entiteter der er forbundet af relationen.
- For hver svag entitet i E/R-diagrammet dannes en tabel i databasen. Alle attributter forbundet til entiteten repræsenteres som felter i tabellen. For alle tabeller forbundet til entiteten med svage relationer dannes felter i tabellen for den relaterede entitetets nøgle.

Referentiel integritet.

Alle steder, hvor en nøgle fra en anden tabel benyttes, oprettes en reference til feltet for den brugte nøgle. På denne måde sikres, at en værdi ikke ændres et sted i databasen, mens den forbliver uændret et andet sted i databasen og dermed, at reference integriteten opretholdes.

Det er muligt at sætte politiker op for behandlingen af brud på referetiel integritet, følgende politikker er i SQL2-standarden:

Standard: Afvis en ændring der ikke overholder den referentielle integritet. DBMS kan ikke behandle ændringen og afviser den derfor med en fejlbesked om, at der er refererende felter.

Før videre: Før ændringen videre til alle refererende felter i databasen. Således sikres, at alle felter har samme værdi efter ændringen. Det betyder, at sletter man tuplen med nøglen, slettes af DBMS alle tupler med en værdi, der refererer nøglen.

Sæt NULL: Alle felter, der refererer den ændrede værdi, ændres til den tomme værdi (`NULL`).

Alle typeerklæringer med `list`, der er refereret i en anden typeerklæring, er repræsenteret som svage entiteter i E/R-diagrammet. Det betyder, at tabellen kommer til at indeholde en referencebegrænsning til nøglen for den refererende tabel. Det er her klart at listen er ubrugelig uden resten af værdien. Derfor kan politiken "før videre" vælges i disse tilfælde.

Brugergrupper.

Brugergrupperne kan læses af funktionssignaturernes annoteringer. De kan derefter direkte oprettes i databasen.

Adgangsbegrænsning.

Brugergruppernes rettigheder til tabellerne i databasen kan defineres ud fra funktionssignaturerne. Hvis en brugergruppe har rettighed til at anvende en funktion, skal den have lov til at læse fra de tabeller, der indeholder de angivne argumenter og resultat. Hvis en funktion ikke giver et resultat (resultat af typen `unit`) har den en sidevirkning. En funktion, der giver en sidevirkning på systemet, må nødvendigvis skulle skrive til databasen, derfor gives der skriveadgang til alle tabeller indeholdende en attribut blandt argumenterne til funktionen.

Det er nu vist at det er muligt at generere en database med referentiel integritet og brugerbegrænset adgang.

6.6 Verifikation af databasens normalform

Verifikation af en normalform for databasen overlades til designeren. Normalformerne er opfyldt hvis:

Første normalform Alle rækker i en tabel har samme antal felter med simple værdier.

Anden normalform Intet felt er afhængig af en delmængde af nøglen for tabellen.

Tredje normalform Intet felt er afhængigt af et andet felt, der ikke er nøgle for tabellen.

Alle normalformer er kun opfyldt, hvis de foregående også er opfyldt. Yderligere forklaring af databasers normalform kan findes i [10].

Er fremgangsmåden beskrevet i dette kapitel brugt til at generere databasen, er det garanteret at databasen er på første normalform, da alle attributer er på en simpel form, og derfor genererer ensartede tupler i databasetabellerne.

Anden normalform er verificeret for alle tabeller, hvor nøglen er en enkelt attribut. Dette, fordi andre attributer ikke kan være afhængige af dele af en attribut. Det samme kan altså ikke garanteres for tabeller, der er genereret af svage entiteter. Disse tabeller indeholder en sammensat nøgle, nemlig referantens nøgle og placeringen i listen.

Hvis den refererede type i listeerklæringen er simpel, er der netop en attribut, der ikke er afhængig af nøglen for den refererende type, eller af placeringen i listen. Hvis den refererede type derimod er en kompleks type, er der ikke nogle attributter i tabellen, der kan være afhængige af en delmængde af nøglen.

Det betyder at den eneste måde, brud på anden normalform kan forekomme, er, når en funktionssignatur genererer en delt nøgle.

Tredje og højere normalforme skal verificeres af designeren, da det kommer an på specifik viden om sammenhængen af typerne.

6.7 Værktøj til generering af database

Der er udviklet et værktøj, *ergen*, der ud fra funktionssignaturer kan generere et E/R-diagram med de omtalte markeringer. Funktionssignaturerne skal angives i en fil, i følgende format:

```
funNavn : typeNavn1 * ... * typeNavnn -> typeNavn
                                          [brugerGruppe]
...
#
typeNavn1 = typeErkl
...
```

Når det genererede E/R-diagram er tilrettet, kan der genereres SQL-sætninger, der kan anvendes direkte på DBMS'et. Hertil har vi udviklet værktøjet *sqlgen*, der efter den tidligere beskrevne metode indlæser E/R-diagrammet og genererer SQL-sætningerne.

Kapitel 7

Design af navigation

I dette kapitel beskrives hvorledes navigationen for et webbaseret system kan modelleres.

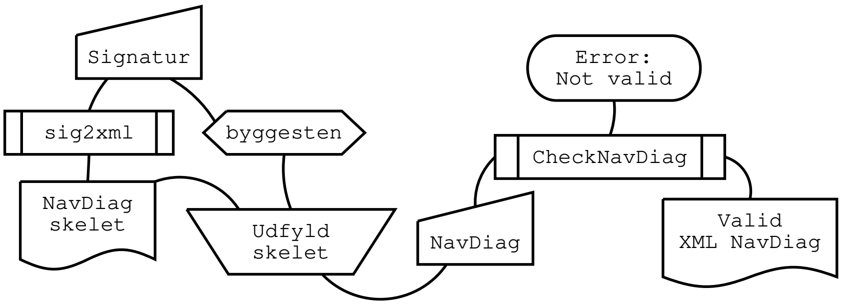
Vi indfører begrebet navigationsdiagram som en model af brugerens interaktion med systemet. Bestanddelene i et navigationsdiagram er begrænset af signaturen. Dette giver en klar fordel, da vi så formelt kan beskrive ting som type- og navigationsikkerhed.

Da navigationsdiagrammet i stor udstrækning skal defineres af en designer, gives et værktøj der kan sikre at de formelt beskrevne egenskaber er overholdt.

Vi vil anvende diagrammer, der oprindeligt er inspireret af [2], men vores fremgangsmåde anvender ikke *frames*. Først og fremmest tillader XHTML 1.0 Strict ikke *frames*. Dermed undgår man problemer forbundet med at web-browserne på markedet viser *frames* forskelligt. Desuden mener vi ikke, at *frames* er nødvendigt for udviklingen af webbaserede systemer, da der ikke er en navigation (eller et layout), der ikke kan laves uden *frames*.

Vores navigationsdiagrammer indeholder alle de elementer, vi gennemgik i afsnit 3. Disse elementer karakteriserer alle elementer i et hypertekstdokument, der kan anvendes ved navigation mellem dokumenterne i systemet.

Som indledning til kapitlet vil vi kort beskrive den process der fører til et valideret navigationsdiagram. Metoden der benyttes til specificering af navigationen er illustreret i figur 7.1. beskrevet nedenfor er forbundet.



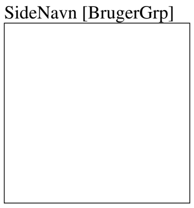
Figur 7.1: Fremgangsmåde for at definere navigation.

Udgangspunktet for specificering af navigationen er en *signatur*, der indeholder specifikation af brugergrupper, funktionssignaturer og typer på en form, som beskrevet i kapitel 5. For at lette implementeringen af validerings- og kodegenereringsværktøjer repræsenteres et navigationsdiagram i en XML-fil, der ud over selve navigationen også indeholder funktionssignaturer og typer. Mere om dette i afsnit 7.3. For at lette overgangen har vi implementeret værktøjet *sig2xml*, der genererer et skelet til en XML-repræsentation af navigationen (*NavDiag skelet*), baseret på en *signatur*. I næste trin, *udfyld skelet*, skal brugerens interaktion med systemet bestemmes. De *byggesten* der kan benyttes til beskrivelsen er begrænset af *signaturen*. Resultatet er et navigationsdiagram (*NavDiag*) som kan valideres af værktøjet *CheckNavDiag*. Hvis *CheckNavDiag* ikke raporterer fejl, har man et valideret navigationsdiagram (*valid XML NavDiag*) der kan benyttes til generering af kode.

7.1 Navigationsdiagrammer

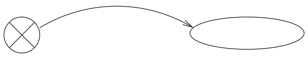
Et navigationsdiagram bygges begrebsmæssigt op ved hjælp af *sider*, *statiske* elementer, *dynamiske* elementer og *link*-elementer.

En side indeholder *statiske* og *dynamiske* elementer og er forbundet til sig selv eller andre *sider* via *link*-elementer. En *side* angives i et navigationsdiagram grafisk ved en simpel firkant annoteret med navn og brugergruppe, se figur 7.2.



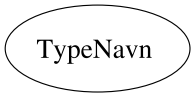
Figur 7.2: Grafisk repræsentation af en *side*.

Et *link*-element består af et anker, der peger på et *dynamisk* element. Dette angives i et navigationsdiagram grafisk ved en cirkel med et kryds som anker, og en pil til det dynamiske element, se figur 7.3. Et link kan som det eneste element annoteres med funktionsapplikationer. Dette gøres ved at angive funktionsnavn og en nummereret liste af typenavne for argumenterne, over pilen.



Figur 7.3: Grafisk repræsentation af et *link*.

Et *dynamisk* element er et element, der er fremkommet som resultat af en funktionsapplikation. Det kan således kun eksistere i forbindelse med destinationen af et *link*-element. Typen af funktionsresultatet definerer typen på det dynamiske element. Et *dynamisk* element angives i et navigationsdiagram grafisk ved en ellipse indeholdende et typenavn, se figur 7.4.



Figur 7.4: Grafisk repræsentation af et *dynamisk* element.

Et *statisk* element indsættes, hvor et argument til en funktion ikke er afhængigt af andre funktionsresultater. Således er statiske elementer faste for siderne. Her menes faste i den forstand, at det kan være skjulte værdier, indtastningsfelter, radioknapper etc., der er faste for siden. Et *statisk* element er således ikke nødvendigvis bundet til en fast værdi, men derimod til en funktionsuafhængig definition af hvorledes værdien fremkommer, se afsnit 3.2.1. Grafisk angives et *statisk* element i et navigationsdiagram ved et typenavn omgivet af vinkler, se figur 7.5.

< TypeNavn >

Figur 7.5: Grafisk repræsentation af et *statisk* element.

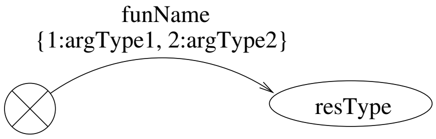
Statiske elementer er de eneste elementer, hvor det er muligt at lade slutbrugeren have direkte indflydelse på værdien af CGI-parametre.

7.1.1 Begrænsning af byggesten

Tidligere nævnte vi at vores udviklingsmetode hovedsageligt er baseret på funktionsspecifikationen. Funktionsspecifikationen benyttes også til begrænsning af byggestenene. En funktionssignatur som:

$funName : argType1 \times argType2 \rightarrow resType$

giver anledning til et *link* som vist på figur 7.6. Det skal bemærkes at det altid er tilladt at benytte links uden funktionsapplikation. Det er ikke tilladt at benytte dynamiske elementer uden tilsvarende funktionsapplikation.



Figur 7.6: *Link* begrænset af funktionssignatur.

7.1.2 Opbygning af et navigationsdiagram

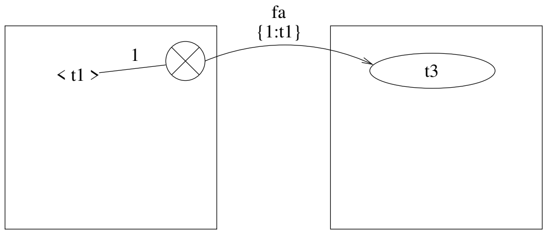
Det er op til designeren at beslutte hvor mange sider, der er nødvendige i systemet, og hvordan de skal være forbundet. Desuden skal det angives, hvorfra argumenterne til eventuelle funktionsapplikationer skal komme. Dette gøres med en streg fra ankeret til elementet, hvorfra argumentværdien skal hentes.

Eksempel:
Givet en signatur som:

$fa : t1 \rightarrow t3$
$fb : t2 \times t4 \rightarrow t5$
$t1 = string$
$t2 = int$
$t3 = t1 \times t2$
$t4 = int$
$t5 = string$

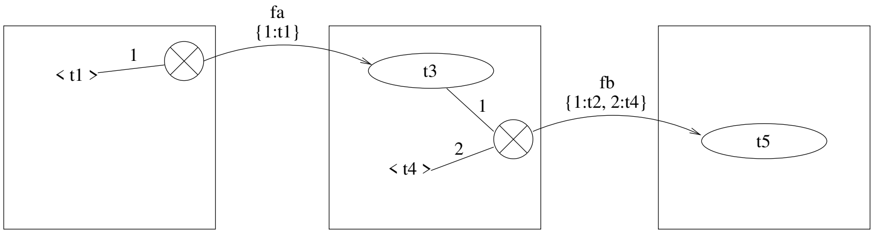
Starter man f.eks. med to sider og forbinder dem med et *link*, der kalder funktionen fa, så skal funktionens argument være at finde på siden med ankeret. Da der ikke er nogle *dynamiske* elementer på denne side, skal der være et *statisk* element med typen $t1$, se figur 7.7. Læg mærke til at alle argumenter skal være til rådighed.

Tilføjes nu et *link*, der kalder funktionen fb fra destinationssiden til en ny side, kan det første argument, der skal være af typen $t2$, findes på 2 måder. Enten ved at tilføje et nyt *statisk* element med typen $t2$, eller ved at udnytte at det *dynamiske* element på siden har typen $t3$, og derfor indeholder en værdi af typen $t2$. Det andet argument



Figur 7.7: To sider forbundet med et *link*.

har typen $t4$ og kan således kun tages fra et nyt *statisk* element. Se figur 7.8 hvor første argument til funktionen fb skal udtages fra det *dynamiske* element.



Figur 7.8: Argumenter tages fra *statisk* og *dynamisk* element.

Der må gerne være flere forskellige *links*, der peger på samme *dynamiske* element, forudsat at de alle indeholder funktionsapplikationer, hvor funktionerne har samme resultat.

I tilfælde, hvor det er angivet, at et argument skal udtrækkes fra et *dynamisk* element, og den del der skal udtrækkes er indeholdt i en *list*-konstruktion, genereres et link for hvert element i listen. Er der tale om, at værdien skal udtrækkes i en type, der indeholder lister af lister, eller skal flere argumenter udtrækkes af typer, der indeholder lister, genereres et link for hver mulig kombination, værdierne kan udtrækkes på.

7.2 Formel model

Dette afsnit indeholder en mere formel definition af et navigationsdiagram. Modellen inkluderer ydermere typer og funktionsdefinitioner fra signaturen og brugergrupper samt adgangsrettigheder.

Vores formelle model er baseret på modeller der beskriver endelige automater [9], men er udviddet med informationer om f.eks. brugergrupper.

Et navigationsdiagram kan defineres ved:

K en oversættelse fra navne på sider til tilstande (siderne).
s den initielle tilstand (startsiden).
$TypeEnv$ en oversættelse fra typenavne til typer.
Sig en oversættelse af funktionsnavne til funktionssignaturer.
U en oversættelse fra navne på brugergrupper til en mængde af indeholdte brugergrupper og en mængde af sider.

Det vil sige at den formelle model for et navigationsdiagram:

$$Diag \quad = \quad (K, s, TypeEnv, Sig, U) \tag{7.1}$$

Modeller for siderne K, startsiden s, typeomgivelserne $TypeEnv$, funktionssignaturerne Sig og brugergrupperne U gives nedenfor.

Bemærk at Id i det følgende er en unik identifikator, der aldrig gentages, og at $IdRef$ er en reference til en af de eksisterende Id-identifikatorer.

Først kigger vi på funktionssignaturerne og typeomgivelserne, der er forklaret i kapitel 5. En funktionssignatur har formen:

$$f : S_1 \times \ldots \times S_n \rightarrow C\ [U]$$

hvor f er navnet på en funktion, S_i er et typenavn der definerer typen på det i'te argument til funktionen, C er et typenavn der definerer typen på funktionsresultatet og U er navnet på den brugergruppe med lavest rettighedsniveau der skal kunne eksekvere funktionen. Vi erindre at alle funktionsargumenter skal have simple typer og at funktionsresultater er komplekse typer. Således er $S_1, \ldots, S_n$ typenavne der refererer simple typer og C er et typenavn der refererer en kompleks type.

I den formelle model repræsenteres en funktionssignatur:

$$
\begin{aligned}
TN &= \{N_1, \ldots, N_k\} && \text{(Typenavne)} && (7.2)\\
FN &= \{N_1, \ldots, N_p\} && \text{(Funktionsnavne)} && (7.3)\\
UN &= \{N_1, \ldots, N_u\} && \text{(Brugergrupper)} && (7.4)\\
Sig &= FN \mapsto (TN - list \times TN \times UN) && && (7.5)\\
f &: S_1 \times \ldots \times S_n \rightarrow C\ [U], && &&\\
& \quad \text{betyder at } Sig(f) = ([S_1, \ldots, S_n], C, U) && &&
\end{aligned}
$$

TN er en endelig mængde typenavne, FN er en endelig mængde funktionsnavne, og Sig er en oversættelse fra et funktionsnavn til en liste typenavne for argumenterne, et typenavn for resultatet og brugergruppen med rettighed til at anvende funktionen.

Typeomgivelserne kan ligeledes formaliseres:

$$
\begin{aligned}
SType &= \{A_1, \ldots, A_n\}, && \text{(Simple typer)} && (7.6)\\
CType &= TN \mid SType \mid CType_1 \times \ldots \times CType_n && &&\\
& \quad \mid CType\ list && && (7.7)\\
TypeEnv &= TN \mapsto CType && && (7.8)
\end{aligned}
$$

Hvor de simple typer $SType$ er `int`, `string` og `bool`, og typekonstruktorerne for kartesisk produkt og lister er til rådighed for komplekse typeerklæringer $CType$. Typeerklæringerne må dog ikke være cykliske. Mere om kontrollen af dette senere.

Navigationsdelen indeholder:

$$
EType = \text{“}Static'' \mid \text{“}Dynamic'' \qquad \text{(Elementtyper)} \qquad (7.9)
$$

$$Elem = Id \mapsto (TN \times EType) \quad (7.10)$$

$$\begin{aligned} SN &= \{N_1, \ldots, N_q\} \qquad \text{(Sidenavne)} && (7.11)\\ Link &= Id \mapsto (FunApp - set \times SN) && (7.12)\\ Side &= Elem \times Link \times UN && (7.13) \end{aligned}$$

Hvor $Elem$ er en oversættelse fra identifikatorer til typenavne og elementets type (statisk eller dynamisk) for elementer i et hypertekstdokument. SN er en endelig mængde af sidenavne. $Link$ er en oversættelse fra identifikatorer til mængden af funktionsapplikationer og navnet på destinationssiden. En $Side$ indeholder elementer $Elem$, links $Link$ og en rettighedsannotering.

På et linkelement kan der foretages nul eller flere funktionsapplikationer, disse har formen:

$$\begin{aligned} Arg &= IdRef \times TN && (7.14)\\ FunApp &= FN \times Arg - list && (7.15) \end{aligned}$$

Arg er identifikatoren for hvilket element, der indeholder værdien, TN er typen. Det er nødvendigt med typen, da man godt kan anvende en undertype af elementet, og værdien kan derfor bestemmes ud fra typen. $FunApp$ er en endelig mængde af funktionsnavne og argumentlister.

Vi kan nu samle definitionen af et navigationsdiagram:
($Diag$ (7.1), Sig (7.5) og $TypeEnv$ (7.8) gentages.)

$$\begin{aligned} K &= SN \mapsto Side && (7.16)\\ s &\in SN && (7.17)\\ TypeEnv &= TN \mapsto CType &&\\ Sig &= FN \mapsto (TN - list \times TN \times UN) &&\\ U &= UN \mapsto (UN - set \times SN - set) && (7.18)\\ Diag &= (K, s, TypeEnv, Sig, U) && \end{aligned}$$

Vi har nu formaliseret en model for navigationsdiagrammer. Dette gør det muligt at beskrive egenskaber ved navigationsdiagrammer. Vi vil anvende dette til at definere kravene for velformede navigationsdiagrammer.

7.2.1 Krav for velformethed

Baseret på ovenstående model, har vi defineret en række udtryk, der skal være overholdt for, at der er tale om et velformet navigationsdiagram.

Når et navigationsdiagram er velformet, vil det sige, at der er navigationssikkerhed i systemet. Det betyder at følgende egenskaber skal være gældende for systemet:

- Startsiden skal være blandt siderne i systemet.
- Links er velformede, der eksisterer signaturer for funktionerne med tilsvarende navne og argumenter.
- Der er ingen døde links.
- Der er værdier for alle argumenter i elementer på siden.
- Resultatet af en funktion har samme type som det element, der skal indeholde værdien.
- Alle sider er forbundet.

Nedenfor gennemgår vi egenskaberne en for en og formaliserer dem i modellen for navigationsdiagrammer.

Funktioner benyttet i fremstillingen:

$\mathcal{D}(SideNavn)$ finder den side, som $SideNavn$ mappes til, og returnerer funktionen $Id \mapsto (TN \times EType)$, hvor de Id, der mappes til et element, der har $EType =$ "*Dynamic*", er defineret.

$\mathcal{S}(SideNavn)$ finder den side, som $SideNavn$ mappes til, og returnerer funktionen $Id \mapsto (TN \times EType)$, hvor de Id, der mappes til et element, der har $EType =$ "*Static*", er defineret.

$\mathcal{L}(SideNavn)$ finder den side, som $SideNavn$ mappes til, og returnerer en mængde indeholdende $(FunApp-set \times SN)$ for alle de Id, der er defineret for siden.

Domænet, en oversættelse er defineret for, er givet af *dom*.

Startsiden skal være i mængden af alle sider indeholdt i diagrammet:

$$s \in dom\ K \tag{7.19}$$

Alle link skal være velformede, det vil sige, at der eksisterer en signatur med passende funktionsnavn og argumenttyper for hver funktionsapplikation.

$$\begin{aligned}\forall s' \in dom\ K &: \forall(fapps, \_) \in \mathcal{L}(s')\\ &: \forall(fn, [(\_, TN_1), \ldots, (\_, TN_j)]) \in fapps\\ &: ([TN_1, \ldots, TN_j], \_, \_) = Sig(fn)\end{aligned} \tag{7.20}$$

Der må ikke være døde links, det vil sige, at alle links skal pege på en side i navigationsdiagrammet.

$$\begin{aligned}\forall s' \in dom\ K &: \forall(\_, dest) \in \mathcal{L}(s')\\ &: dest \in dom\ K\end{aligned} \tag{7.21}$$

Alle argumenter er velformede, det vil sige, at alle argumenter benyttet i en funktionsapplikation, skal findes via *statiske* eller *dynamiske* elementer på den side, hvor *link*-elementet har sit anker. Dette kan være direkte, eller udtaget fra et komplekst *dynamisk* element.

$$\begin{aligned}\forall s' \in dom\ K &: \forall(fapps, \_) \in \mathcal{L}(s')\\ &: \forall(\_, args) \in fapps\\ &: OccursIn(args, (\mathcal{S}(s') \cup \mathcal{D}(s')))\end{aligned} \tag{7.22}$$

hvor, $ExistsIn(TN_a, TN_b)$ afgør, om den navngivne type TN_a er lig med eller indeholdt i den navngivne type TN_b, og

$$\begin{aligned}OccursIn(args, maps) &= \forall(IdRef, TN) \in args\\ &: (TN_m, \_) = maps(IdRef)\\ &\quad \wedge ExistsIn(TN, TN_m)\end{aligned} \tag{7.23}$$

Alle *dynamiske* elementer skal have korrekt type. Det vil sige at alle sider indeholder dynamiske elementer med samme type som resultatet af de funktionsapplikationer, der udføres via links, der peger på siden:

$$\begin{aligned}\forall s' \in dom\ K &: \forall(Fapps, Dest) \in \mathcal{L}(s') \\ &: \{res_i | (fn_i, \_) \in Fapps \wedge (\_, res_i, \_) \in Sig(fn_i)\} \\ &\quad = \{(\mathcal{D}(Dest))(Id) | Id \in dom\ \mathcal{D}(Dest)\} \qquad (7.24)\end{aligned}$$

Alle sider er forbundet, det vil sige, at alle sider kan nås fra startsiden ved at følge links på de enkelte sider.

$$\forall s' \in dom\ K \quad : \quad Connected(s, s') \qquad (7.25)$$

hvor $Connected(s1, s2) =$
$(s1 = s2) \vee \exists(\_, Dest) \in \mathcal{L}(s1) : Connected(Dest, s2)$

7.3 Navigationsdiagram i XML

Vi har valgt at benytte XML til at repræsentere navigationsmodellen. Verifikationsværktøjet er implementeret i Java og gør brug af Apaches Xerces Parser.

Ved at bruge XML til at repræsentere navigationsmodellen, har vi mulighed for at benytte XML Schema [1] til at definere en stor mængde af de nødvendige begrænsninger på modellen. Vi kan benytte Xerces-parseren til at validere XML-dokumentet mod XML-Schemaet. Dette gør, at kun de mere komplekse krav kræver egentlig implementering i Java. Et navigationsdiagram, der overholder den formelle model, kan skrives i en XML-fil. Hovedopdelingen af en sådan fil skal være, som følger:

En eller flere *webpage*-elementer, der hver især indeholder statiske elementer, dynamiske elementer og linkelementer.

Et *alfabet*-element, der indeholder definitioner af alle navngivne typer.
Et *funsigs*-element, der indeholder definitioner af alle funktionssignaturer.
Et *startpage*-element der indeholder navnet på startsiden.
Et *accessspec*-element, der indeholder definitioner af brugergrupper og deres rettigheder til at tilgå de enkelte sider i systemet.

Vi har beskrevet den fulde struktur af en XML-fil indeholdende navigationsdiagram, signatur og adgangsbegrænsning i et XML-Schema kaldet *navdiag.xsd* .

For at lette arbejdet med at udfærdige gyldige navigationsdiagrammer i XML har vi implementeret et værktøj, kaldet *sig2xml* . Det tager samme argument som E/R-generator værktøjet, se afsnit 6.7 og genererer *alfabet* og *funsigs* elementerne, der skal være i XML-filen.

7.4 Værktøj til verifikation

Vi har på baggrund af ovenstående model implementeret et værktøj kaldet *CheckNavDiag*, der kan validere et navigationsdiagram.

Dette indebærer at sikre at alle krav for velformethed, se afsnit 7.2.1, er opfyldt. F.eks. sikres det at der ikke forekommer cykliske typedefinationer ved at benytte en farvende dybdeførstsøgning.

Derudover undersøges det at brugergruppers rettigheder til funktioner og sider, sikres og overholdes, samt at der ikke kan snydes med funktionsrettigheder.

Se appendiks B for en nærmere gennemgang af de tests *CheckNavDiag* udfører, for at validere et navigationsdiagram.

Kapitel 8

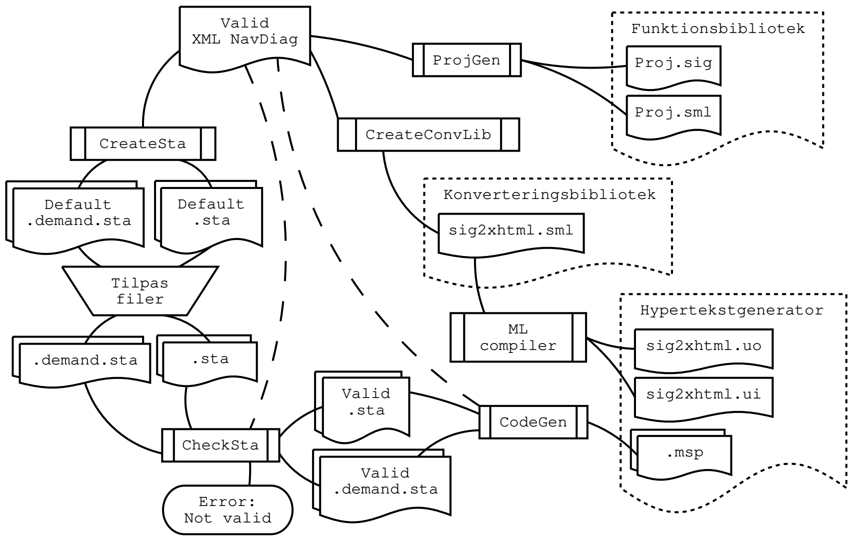
Generering af kode

I dette kapitel beskrives hvorledes et navigationsdiagram, se kapitel 7, kan benyttes som grundlag for en implementering af navigationen.

Vi giver værktøjer til generering af prototypeimplementering af de funktionelle krav i et funktionsbibliotek, generering af færdig implementering af konverteringsfunktioner i et konverteringsbibliotek, generering af konfigurerbare filer for siderne i navigationsdiagrammet, validering af konfigurerbare filer og generering af færdige MSP-filer der, baseret på de konfigurerbare filer, implementerer navigationen og binder denne sammen med funktionsbibliotek og konverteringsbibliotek.

Som indledning til kapitlet vil vi kort beskrive den process der fører til ovenstående arkitektur. Processen er illustreret i figur 8.1.

Udgangspunktet for processen er et valideret navigationsdiagram (*Valid XML NavDiag*), se kapitel 7. Der er tre separate dele i processen, generering af *funktionsbibliotek*, *konverteringsbibliotek* og *.msp*-filer. Sammen med det oversatte *konverteringsbibliotek* udgør *.msp*-filerne den færdige *hypertekstgenerator*.



Figur 8.1: Oversigt over generering af kode for et webbaseret system.

Funktionsbibliotek Værktøjet *ProjGen* kan direkte generere en færdig ML-signatur (*Proj.sig*), samt en prototypeimplementering af ML-signaturen (*Prog.sml*).

Konverteringsbibliotek Værktøjet *CreateConvLib* kan direkte genere en implementering af *konverteringsbiblioteket*. Den genererede fil kaldes *sig2xhtml.sml*.

.msp Genereringen af *.msp* filerne foregår i en række trin. I første trin genererer værktøjet *CreateSta* to konfigurerbare filer for hver side i navigationsdiagrammet (*Default .demand.sta* og *Default .sta*). I næste skridt, *Tilpas filer*, har designeren mulighed for at tilføje fast indhold og justere *statiske* elementer. Herefter kan de ændrede konfigurerbare filer (*.demand.sta* og *.sta*) valideres med værktøjet *CheckSta*. Til sidst kan værktøjet *CodeGen* givet de velformede konfigurationsfiler (*Valid .demand.sta* og *Valid .sta*) generere en *.msp*-fil for hver side i systemet.

Udover de her nævnte værktøjer har vi implementeret et værktøj der kan hjælpe med håndtering af konfigurationsfiler i store systemer. Mere om dette i afsnit 8.4.3.

8.1 Generator, en oversigt

Resultatet af kodegenereringsfasen er en arkitektur hvor:

- Implementering af de funktionelle krav foregår i et stærkt typet miljø.
- Navigationen er skarpt separeret fra den funktionelle del.
- Hypertekstgeneratorerne altid giver valid XHTML 1.0 Strict. Se [13].
- Brugerindtastede værdier bliver typetestet på klient og server for at opnå fordelene beskrevet i [4]

Funktionsbibliotek.

For at opnå fuld funktionalitet er det kun nødvendigt at udskifte prototypeimplementeringen med en implementering, der tilgår databasen på passende vis.

Ved at benytte ML og dettes modulsystem opnår vi en modulær arkitektur i vores webbaserede system. Desuden er ML's stærke typesystem, sammen med understøttelsen af sammensatte værdier og højereordens funktioner, med til at gøre ML til en glimrende platform til konstruktion af webbaserede systemer via applikationer, der involverer strukturerede værdier. Se [11].

Konverteringsbibliotek

Konverteringsbiblioteket *sig2xhtml.sml* indeholder funktioner til at konvertere typede ML-værdier til gyldig XHTML. Det benyttes til at visualisere funktionsresultater. Da konverteringsbiblioteket genereres maskinelt på passende vis, sikres det at funktionsresultater altid visualiseres på en måde der er gyldig i XHTML 1.0 Strict.

Konfigurerbare filer

Brugen af konfigurerbare filer muliggør validering af de faste informationer der skal være på de enkelte sider. Dette sikrer at de kun indeholder gyldig XHTML 1.0 Strict, men inkluderer også f.eks. typetest af skjulte værdier i *statiske* elementer.

Vi ender altså med en arkitektur, hvor alle konfigurerbare dele kan valideres før generering af det endelige system. Dette betyder at vi med fornuftigt konstruerede værktøjer kan garantere at det færdige system altid vil generere XHTML 1.0 Strict.

.msp

Filerne implementerer tilsammen navigationen og binder denne sammen med funktionsbibliotek og konverteringsbibliotek.

Da *.msp*-filerne genereres maskinelt på passende vis, sikres at funktionsbibliotek og konverteringsbibliotek benyttes korrekt.

Filerne genereres så der sikres en korrekt typet interaktion med brugere af systemet. Dette sikres ved automatisk typetest, både hos klienter og server. Specielt hos serveren er dette nødvendigt, da systemet ellers ville være åben for utypet brug af funktionsbiblioteket. Dette kunne opstå hvis en bruger omgår klientens typetest og sender CGI-parametre direkte til MSP-filen.

I de tilfælde hvor der i konfigurationsfilerne er aktiveret visualisering af *dynamiske* elementer, er det også de genererede MSP-filer, der står for at kalde de nødvendige funktioner fra konverteringsbiblioteket.

De genererede MSP-filer skal ikke ændres, og placeres i en biblioteksstruktur der afspejler brugergruppernes adgangsbegrænsninger. Således muliggøres brugen af standard validering af brugere, via webserverens restriktionsfunktionalitet.

8.2 Funktionsbibliotek

Som tidligere beskrevet, er det i sidste ende op til de autogenererede MSP-filer, der beskriver navigationen, at kalde de funktioner, der implementer de funktionelle krav. MSP-filerne forventer at kunne kalde disse funktioner via et funktionsbibliotek, der er realiseret som et ML-modul kaldet *Proj*. På denne måde udnytter vi ML's modulsystem til at opnå en streng opdeling mellem navigation og funktionalitet.

Værktøjet *ProjGen* kan, givet en XML-repræsentation af et navigationsdiagram, automatisk generere en ML-signatur der beskriver de påkrævede funktioner, samt deres typer. Desuden genereres en prototypeimplementering af ML-signaturen.

Denne fremgangsmåde giver flere fordele. For det første tillader den automatiske generering af en prototypeimplementering, at man kan afprøve navigationen uafhængigt af den bagvedliggende funktionalitet. For det andet sikrer ML-signaturen, at enhver implementering af de funktionelle krav altid bliver typemæssigt overholdt, og at strukturen af sammensatte værdier overholdes.

Brugen af ML sikrer derudover, at implementeringen af de funktionelle krav foregår i et stærkt typet miljø, og modulstrukturen muliggør, at afprøvning af de implementerede funktioner kan foretages fuldstændig uafhængigt af navigationen.

Endeligt er det kun nødvendigt at udskifte prototypeimplementeringen af de funktionelle krav for at opnå et fuldt funktionelt system. Under implementering af de funktionelle krav, skal man være opmærksom på at benytte de korrekte brugernavne i forbindelse med databaseforespørgsler. Her er det navnet på brugergruppen, der er tilknyttet den enkelte funktion, der skal benyttes som brugernavn til databasen. Ved at benytte de korrekte brugernavne til databasen sikres, at funktionerne kun kan benytte de dele af databasen, de pågældende brugere har adgang til. Således reduceres mulighederne for fejlagtig implementering af de funktionelle krav.

8.2.1 Værktøj: ProjGen

Herunder beskrives, hvorledes værktøjet *ProjGen*, på baggrund af en valideret XML-repræsentation af et navigationsdiagram, genererer ML-signatur og prototypeimplementering. Her skal vi som tidligere nævnt minde om at en valideret XML-repræsentation af et navigationsdiagram, inkluderer informationer om typer og funktioner.

Proj.sig

Signaturfilen består af to hoveddele: signaturer for typer, og signaturer for funktioner.

Alle typesignaturer udtrækkes fra navigationsdiagramfilen, og de ordnes så sekventiel definition af typerne er mulig i ML. Det vil sige at typerne defineres i en rækkefølge, så der aldrig benyttes udefinerede typenavne til at definere nye typer med. Herefter udtrækkes alle signaturer for funktioner fra navigationsdiagrammet. Informationerne skrives med ML-syntaks for en signatur, i en fil kaldet *Proj.sig* .

Proj.sml

Ligesom signaturfilen, består den genererede prototypeimplementering af to hoveddele: definition af typer, og definition af funktioner.

I ML er syntaksen for at definere typer den samme som for at angive deres signatur. Således kan typedefinitionen laves præcist, som det var gældende for signaturen af typerne i *Proj.sig* -filen.

Prototypedefinitionen af funktionerne genereres ud fra signaturen for funktionerne. Strukturen af typerne benyttes til at generere sammensatte værdier. Hvor listdefinitionen er benyttet, genereres en værdi indeholdende en liste med 1 element.

Simple værdier produceres efter følgende regler:
Hvis argumenterne til funktionerne indeholder typer, der indgår i typen på resultatet, videreføres den del af argumentet direkte til resultatet. For de dele af resultatet, der ikke kan gives direkte fra argumentet, indsættes standardværdier, som følger:
– Typen *int* får værdien `1`
– Typen *string* får værdien `"string:typeNavn"`
– Typen *bool* får værdien `false`

F.eks. vil ML-signaturen

```
type TN1 = int
type TN2 = string
type TN3 = ( TN1 * TN2 )
val FN1 : TN1 -> TN3
```

resultere i en prototypeimplementering som

```
type TN1 = int
type TN2 = string
type TN3 = ( TN1 * TN2 )
fun ( TN1Arg )= ( TN1Arg, "string:TN2" )
```

Når der er genereret prototyper for hver funktion i signaturen, skrives disse sammen med typedefinitionerne, i en fil kaldet *Proj.sml* .

8.3 Konverteringsbibliotek

Konverteringsbiblioteket, *sig2xhtml*, benyttes i de tilfælde, hvor et *dynamisk* element skal visualiseres. Det skal indeholde en funktion for hver af de navngivne typer. Hver funktion skal kunne konvertere en given værdi til en XHTML-repræsentation.

Værktøjet, *CreateConvLib*, kan givet en valideret XML-repræsentation af et navigationsdiagram, fuldt generere den færdige ML-kildekode til konverteringsbiblioteket. Den genererede kildekode skal så oversættes med en ML-oversætter, og det færdige ML-konverteringsbibliotek er klart til brug.

Funktionerne i konverteringsbiblioteket kaldes kun fra de autogenererede MSP-filer.

Ved at benytte denne fremgangsmåde, sikres at visualiseringen af funktionsresultater, aldrig vil kunne kompromittere validiteten af de XHTML 1.0 Strict dokumenter, der i sidste ende sendes til klienten.

For at muliggøre definition af den endelige præsentation via CSS, genereres funktionerne så:

- Alle typer pakkes ind i `div`-elementer med `class`-attributten sat til den enkelte types navn.
- Sammensatte typer pakkes ind i `div`-elementer, der afspejler strukturen af typerne.

For hver funktion, der skal konvertere typer bestående af sammensatte værdier, genereres passende hjælpefunktioner.

F.eks. vil typer med ML-signaturen

```
type a = int
type b = string
type c = ( a * b list )
```

resultere i generering af et konverteringsbibliotek med funktionerne:

```
fun a2xhtml(x) =
  "<div␣class=\"a\">\n"^(Int.toString(x))^"</div
     >\n"

fun b2xhtml(x) =
  "<div␣class=\"b\">\n"^(Msp.htmlencode x)^"</div
     >\n"

fun listHelpFun1(lc2) = b2xhtml(lc2)
fun carprodHelpFun0(a0,a1) =
   a2xhtml(a0)^
  "<div␣class=\"list\">\n"^
    (String.concat(List.map listHelpFun1 a1))^
  "</div>\n"
fun c2xhtml(x) =
  "<div␣class=\"c\">\n"^
   ("<div␣class=\"carprod\">\n"^
     (carprodHelpFun0 x)^
    "</div>\n")^
  "</div>\n"
```

Funktioner, der skal foretage konvertering af sammensatte typer, kalder konverteringsfunktionerne for de enkelte deltyper. For at sikre at alle funktioner, der kaldes er defineret før kaldet, benyttes ordningsmekanismen fra *ProjGen*-værktøjet.

8.4 Konfigurerbare filer

For at tilgodese et ønske om at have et webbaseret system hvor man kan tilføje fast tekst, billeder og lignende så har vi valgt at benytte konfigurerbare filer. Der skal laves 1 eller 2 konfigurationsfiler for hver side i systemet. Der skal altid være en konfigurationsfil, der for den enkelte side, beskriver det faste synlige indhold, samt rækkefølgen af samme. Det være sig tekst, billeder, placering og udseende af linkankre og placering af dynamiske elementer, hvis de skal visualiseres. De sider i navigationsdiagrammet, der indeholder *statiske* elementer, skal hver have yderligere en konfigurerbar fil. Denne skal definere de XHTML-konstruktioner, der er nødvendige for at indsamle de funktionsargumenter, der ikke udtages fra *dynamiske* elementer.

Ved at benytte denne opdeling sikres en adskillelse af indhold og funktionalitet af det faste XHTML.

De to typer konfigurerbare filer kaldes hhv. *.demand.sta* og *.sta* og er baseret på et XML-format.

.sta-filer.

De konfigurationsfiler, der beskriver det faste synlige indhold af en side, navngives med "efternavnet" *.sta* .

En *.sta* -fil må indeholde:

- `submit` tags der hver for sig definerer placering af et link, samt udseende af linkets anker. Med hensyn til udseende af et anker må der med enkelte begrænsninger benyttes alt, hvad man normalt må have inde i et XHTML 1.0 Strict anker.
- `dynamic` tags der hver for sig aktiverer visualisering og definerer placering af et *dynamisk* element.
- Alt fast XHTML-indhold der ikke kan ændre på eller blokere for funktionaliteten af systemet. Det vil sige at man kan benytte alle tags fra *XHTML 1.0 Strict BlockGroup*[1], bort set fra forms og ankre.

Den fulde struktur af en *.sta* -fil er defineret i XML Schema filerne *sta.xsd* og *include.xsd* .

[1] *XHTML 1.0 Strict BlockGroup* indeholder tags som paragraffer, overskrifter, lister, tabeller mm.

Som standard indeholder en *.sta* -fil kun en liste med de forskellige links på siden, og hver af disse er sat til at kunne blive aktiveret med en knap, der har titlen på linkets destinationsside som tekst.

.demand.sta-filer.

De konfigurationsfiler der beskriver hvorledes *statiske* argumenter skal tilvejebringes, navngives med "efternavnet" *.demand.sta* .

En *.demand.sta* -fil skal indeholde en definition af hvert *statisk* element på den side filen beskriver. Følgende tags er lovlige, når XHTML-funktionaliteten af det enkelte *statiske* element skal specificeres:

- `input` tags hvor typen er begrænset til indtastningsfelter, password indtastningsfelter, radio knapper, filer og skjulte værdier.
- `select` tags der muliggør et rullegardin valg mellem faste værdier.
- `textarea` tags der tillader indtastningsarealer.
- `label` tags der kan benyttes til at give labels for de øvrige tags.

Derudover er der en række begrænsninger knyttet til brugen af de forskellige tags. Det er f.eks. ikke tilladt at have flere `input` tags der definerer indtastningsfelter for samme *statiske* element, hvorimod det er i orden med flere `input` tags hvis de alle definerer radioknapper. Det er heller ikke lovligt at tillade samtidigt valg af flere værdier i en `select` konstruktion, da et *statisk* element altid skal være af simpel type.

Se afsnit 8.4.2 for en nærmere beskrivelse af begrænsningerne for indholdet af *.demand.sta* -filer.

Den fulde struktur af en *.demand.sta* -fil er defineret i XML Schema filerne *demand.sta.xsd* og *include.demand.sta.xsd* .

Alle *statiske* elementer genereres som standard til at være indtastningsfelter, men kan ændres til skjulte værdier, drop-down bokse og lignende med forudbestemte værdier.

8.4.1 Værktøj: CreateSta

Værktøjet *CreateSta* kan, givet en valideret XML-repræsentation af et navigationsdiagram, generere standard versioner af alle de konfigurerbare filer i systemet.

For hver side i navigationsdiagrammet, oprettes en *.sta* -fil. Som standard indeholder den genererede fil kun en `submit` tag for hvert *statisk* element på siden. I hver af disse `submit` tags er linkets anker defineret som en knap, med en tekst, der er titlen på siden for linkets destination.

For hver side i navigationsdiagrammet der indeholder *statiske* elementer, oprettes en *.demand.sta* -fil. En *.demand.sta* -fil skal som tidligere nævnt, indeholde en definition af hvert *statisk* element på den side filen beskriver. Som standard defineres alle *statiske* elementer som tekst indtastningsfelter.

Således genererer *CreateSta* værktøjet konfigurationsfiler, der tillader hurtig generering af navigerbare prototyper af systemet.

8.4.2 Verificering af konfigurationsfiler

Da konfigurationsfiler i sagens natur er til for at blive ændret af en bruger, er der også mulighed for at introducere fejl. Med værktøjet *CheckSta* kan de konfigurerbare filer valideres mod en given XML-repræsentation af et navigationsdiagram.

Det sikres at en række egenskaber er opfyldt, bl.a. at de konfigurerbare filer indeholder specifikationer af alle de *statiske* elementer i navigationsdiagrammet. Herunder at dele af en specifikation der angiver CGI værdier som ikke kan ændres hos klienten, har korrekt type.

Verificeringen af konfigurationsfiler kan således afsløre problemer i de filer der vil medføre fejl, før generering af *.msp* -filer foretages.

De to typer konfigurerbare filer valideres hver for sig, og har hver deres XML Schema fil, der inkluderer begrænsede versioner af det fulde XML Schema for XHTML 1.0 Strict. Navigationsdiagrammet benyttes til at fastlægge hvilke filer der skal være i systemet, og hver af disse valideres som beskrevet i appendiks C.

8.4.3 Håndtering af konfigurations filer i store systemer

Ved større webbaserede systemer med mange forskellige sider, vil der ofte opstå redundant information i *.sta* -konfigurationsfilerne. Eksempler på dette kunne være et fast sidehoved der skal kunne findes på alle sider, eller kontakt information der skal være på en mængde sider. Skal der i disse eksempler ændres i sidehovedet eller kontaktinformationen skal ændringerne foretages i alle de berørte *.sta* -filer.

Her ville genbrugelighed, af informationer der ikke lagres i database, være en stor fordel, i forbindelse med vedligeholdelse af systemet.

Til dette formål har vi udviklet værktøjet *tagmerger*, der kan benyttes som en preprocessor til at bygge *.sta* -filer med.

tagmerger tager to argumenter. Navnet på den fil der skal læses fra, og navnet på den fil der skal skrives til. Ved eksekvering kopieres indhold fra inputfilen til outputfilen. Under kopieringen søges efter kommandoer. Hvis en kommando mødes, udføres denne før kopieringen fortsætter. De to kommandoer *tagmerger* understøtter er:

`<?insert:`*fn*`?>` Indsætter indholdet af filen med navnet fn på dette sted. Der søges efter og udføres kommandoer, i de på denne måde inkluderede filer.

`<?const:`*cn*`?>` Indsætter værdien af konstanten *cn* på dette sted. De enkelte konstanter skal defineres i en separat fil.

Ved at benytte disse kommandoer kan redundant information nu skabes ved at indsætte samme information i flere filer.

I eksemplet hvor der skulle være et fast sidehoved, kunne man f.eks. nøjes med at definere sidehovedet i en enkelt fil kaldet *sidehoved.part* , og derefter indsætte kommandoen `<?insert:sidehoved.part?>` i toppen af alle filer hvor sidehovedet skulle indsættes. Skal der herefter foretages ændringer i sidehovedet, kan *tagmerger* værktøjet sørge for at ændringerne bliver distribueret til alle de berørte filer.

8.5 MSP-kodegenerering

MSP-filerne er rygraden i det færdige webbaserede system. Der skal være en MSP-fil for hver side i navigationsdiagrammet. Grundlæggende genererer en MSP-fil, det hypertekstdokument, der sendes til klienten. Dette inkluderer alle links, og MSP-filerne definerer således navigationen i systemet.

Det er MSP-filerne der står for:

- Modtagelse og validering af CGI parametre.
- Kald af funktioner i funktionsbiblioteket.
- Konvertering af funktionsresultater til XHTML ved kald af funktioner i konverteringsbiblioteket.
- Generering af links, herunder udtrækning af delresultater fra funktionsresultater.

Givet velformede konfigurationsfiler, kan værktøjet *CodeGen*, ud fra navigationsdiagrammet, automatisk generere alle de færdige MSP-filer for systemet.

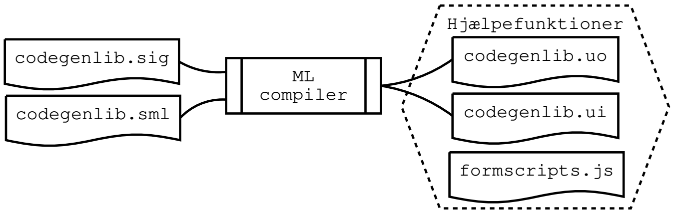
Eksekveringen af MSP-filer foregår sekventielt, og opbygningen er som følger:
Først hentes og typetestes de forventede CGI parametre.
Derefter kaldes funktioner i funktionsbiblioteket, og deres resultater gemmes i variable.

Herefter er indholdet af den til siden hørende *.sta* -konfigurationsfil indsat. Hvor der er linkanker definitioner i *.sta* -filen, er der i stedet indsat MSP-kode, der genererer links. Hvor der er aktiveringer af visualisering i *.sta* -filen, er der indsat funktionskald der producerer HTML ud fra resultaterne fra kaldene af funktionsbiblioteket.

8.5.1 Hjælpefunktioner

De genererede MSP-filer benytter hjælpefunktioner fra ML-biblioteket *codegenlib*. I biblioteket har vi blandt andet implementeret funktioner til at finde det cartesiske produkt af to lister. Biblioteket benyttes kun af MSP-filer.

Derudover er der javascript hjælpefunktioner som benyttes ved typevalidering af værdier hos klienten.



Figur 8.2: Klargøring af hjælpefunktioner.

Hjælpefunktionerne er faste for metoden. I figur 8.2 kan det ses hvorledes hjælpefunktionerne skal klargøres til brug.

8.5.2 CGI parametre

I dette afsnit vil vi beskrive de konventioner der benyttes ved navngivning af CGI parametre.

Der skal være et sæt CGI parametre for hvert *dynamisk* element på en side. Disse parametre skal angive hvilken funktion der skal kaldes for at generere det *dynamiske* element, samt de nødvendige argumenter til funktionen.

Formatet for hvert sæt CGI parametre er som følger:

- Den parameter der skal angive hvilken funktion der skal køres, for at generere et *dynamisk* element med identifikatoren *dynid*, skal navngives "*dynid*`fun`". Værdien af "*dynid*`fun`" parameteren skal være navnet på den funktion der skal køres.
- En parameter der skal angive argument nummer i til funktionen med navnet $funname$, skal navngives "$funname$`arg`i". Værdien af "$funname$`arg`i" parameteren skal være det som skal benyttes ved kald af funktionen.

Eksempel:
Et *dynamisk* element med identifikatoren `d1`, skal genereres ved kald af funktionen `myFun`, med argumenterne `1` og `A`.
dette vil give et CGI parameter sæt som
`d1fun=myFun&myFunarg0=1&myFunarg1=A`

Denne fremgangsmåde tillader at forskellige links kan benytte forskellige funktioner til at generere det samme *dynamiske* element. Samtidigt tillader det også at der kan være flere funktionsapplikationer på ét link.

8.5.3 Server-side validering, konvertering og funktionsapplikering

Når CGI parametrene modtages af en MSP-fil, så skal de valideres.

Først skal det sikres at alle de forventede CGI parametre eksisterer. Dette afgøres ved, for det enkelte *dynamiske* element på siden, at hente den CGI parameter der angiver hvilken funktion der skal køres for at generere værdien af elementet. Dernæst afgøres om funktionsnavnet er blandt de funktioner der må benyttes til generering af netop det *dynamiske* element. Hvis det er en "lovlig" funktion sikres at de nødvendige argumenter kan hentes. Ved en "lovlig" funktion forstås en funktion der i navigationsdiagrammet forekommer på et af de links der resulterer i det specifikke dynamiske element.

Ved at undersøge om der er tale om en "lovlig" funktion, sikres det at der ikke kan kaldes utilsigtede funktioner fra funktionsbiblioteket, selvom en bruger forsøger at snyde systemet ved at sende fabrikerede CGI parametre direkte til MSP-filen.

Næste trin er at sikre at alle funktionsargumenter kan valideres som værende af den korrekte type. Dette er nødvendigt for at sikre at funktionsbiblioteket ikke benyttes med utilsigtede værdier.

Når funktionsargumenternes typer er valideret, konverteres de til de korrekte ML-typer, så de er klar til at blive brugt ved kald af funktionsbiblioteket.

Endeligt benyttes de konverterede argumenter til at kalde funktionen i funktionsbiblioteket. Resultatet af funktionskaldet gemmes i en variabel der senere kan benyttes til udtrækning af argumenter til nye links, og visualiseres via kald til konverteringsbiblioteket.

Med de typebegrænsninger der på nuværende tidspunkt er i vores system, vil alle argumentværdier der kan konverteres, også kunne valideres. Dette vil dog ikke nødvendigvis være tilfældet, hvis metoden f.eks. udviddes til at kunne håndtere simple typer begrænset af regulære udtryk. Mere om dette i kapitel 11 der beskriver forslag til videre arbejde.

Når *CodeGen* værktøjet genererer MSP der skal udføre ovenstående process, udnyttes navigationsdiagrammet til opbygning af en `case` konstruktion der hurtigt kan afgøre om et givet funktionsnavn er "lovligt", og i så fald hvor mange argumenter der skal kunne læses, samt hvad deres type skal være.

Eksempel:
Antag, at et *dynamisk* element med identifikatoren `d1`, ifølge navigationsdiagrammet kun kan genereres ved kald af funktionen `myFun`, der tager to argumenter af typerne *int* og *string*.

Ud fra de ovenstående informationer vil *CodeGen* værktøjet generere følgende erklæring af variablen `d1`.

```
val d1 = let val funName = % "d1fun"
  in case funName of
      "myFun" =>
          let
            val arg0 = case ( Int.fromString(%(funName^"arg0
                ")) )of
                          SOME(i) => i
                        | NONE => raise badtypedCGIParameter
            val arg1 = % (funName^"arg1")
          in
            myFun(arg0,arg1)
          end
    | _ => raise illegalFunction("d1:"^funName)
  end
```

Funktionen `myFun` forventes her at være implementeret i *Proj.sml* funktionsbiblioteket, der er åbnet før variabelerklæringen.

8.5.4 Linkelementers repræsentation

Hvis der i en *.sta* -fil er en definition af et linkanker, skal den tilsvarende MSP-fil indeholde kode der genererer de tilsvarende links.

Et linkelement repræsenteres således i MSP-filen, ved MSP-kode der genererer links opbygget efter følgende fremgangsmåde:

1. Byg MSP-funktion der returnerer et formlink.
   Dette inkluderer kald der sikrer klient-side typetest.
2. Udtræk funktionsargumenter fra de dynamiske elementer.
3. Kald af linkgenerator funktionen.

De enkelte dele beskrives i det efterfølgende.

Som gennemgående eksempel benyttes følgende del af en signatur:

```
type a = int
type b = int
type c = string
```

```
type d = ( b * c )
type e = ( a * d list )

val myFun : a * b -> d
```

Der beskrives repræsentationen af et link, der er annoteret med funktionen `myFun`.
Følgende antages for eksemplet: På siden hvor linket skal være, er der et *dynamisk* element, med identifikatoren `d1` og typen `e`. Første argument til `myFun` skal komme fra et *statisk* element, der er i den tilhørende *.demand.sta* -fil, er standard defineret som et indtastningsfelt. Det andet argument skal udtrækkes fra det *dynamiske* element. Linkets destination er på en side med navnet `dInfoPage`. Resultatet af funktionsapplikationen er et *dynamisk* element med identifikatoren `d2`.

Byg linkgenerator funktion.

Første del af den kode der genereres, ved opbygning af et linkelements repræsentation, er en MSP-funktion der returnerer et formlink, med tilhørende kald af klient-side typetest funktioner.

Et link kan være annoteret med funktionsapplikationer. Argumenterne til de enkelte funktioner kan komme fra enten *statiske* eller *dynamiske* elementer.

Den MSP-funktion der skal opbygges, skal have et argument, for hvert argument i funktionsapplikationerne, der skal udtrækkes fra *dynamiske* elementer.

For det XHTML-formlink, funktionen skal returnere skal følgende gælde:

- Formens destination er sat til linkets destination.
- Alle MSP-funktionens argumenter indsættes som skjulte værdier.

- Alle definitioner af statiske elementer der vedrører dette link indsættes fra *.demand.sta* -filen
- Ankerdefinitionen indsættes fra *.sta* -filen.

Derudover skal funktionen sikre at der, via javascript hjælpe funktionen, `submitForm`, bliver foretaget klient-side typetest af værdier for statiske elementer.

For det generelle eksempel der benyttes i dette afsnit, genereres følgende MSP-funktion.

```
fun link1FormCont (myFunarg1) = $ (flattenFrags `
  <div id="dInfoPageLink^(getFormCountInc())">
    <form id="dInfoPageLinkForm^(getFormCount())" method="
        get" action="/dInfoPage.msp" onsubmit="return␣
        submitdInfoPageLinkForm^(getFormCount())();">
      <div>
        <input name="d2fun" type="hidden" value="myFun" />
        <input name="myFunarg1" type="hidden" value="^
            myFunarg1" />
        <input name="myFunarg0" type="text" />
        <a href="javascript:void(submitdInfoPageLinkForm^(
            getFormCount())());">
          <input type="submit" value="dInfoPage" />
        </a>
      </div>
    </form>
    <script type="text/javascript">
      function submitdInfoPageLinkForm^(getFormCount())() {
        return submitForm('dInfoPageLinkForm^(getFormCount()
            )', [['myFunarg0','int']]);
      }
    </script>
  </div>
`)
```

Som det kan ses benyttes javascript til at aktivere den genererede form. Dette er valgt da vi således ikke er bundet til knapper, men også kan benytte tekst eller billeder når linkankerets udseende skal specificeres i en *.sta* -fil.

Typetest på klient-siden er ikke vital for systemets funktion, men er derimod indsat som hjælp. Det sikrer at brugeren ikke får lov til at følge et link, før de nødvendige informationer er tilgængelige. Er de ikke det ser brugeren derimod et advarselsvindue, der informerer om de manglende eller ukorrekte typer. Ved at foretage validering på klient-siden, spares serveren også for det arbejde der er forbundet med at skulle returnere fejlbeskeder ved ugyldige indtastninger.

Udtræk funktionsargumenter

Anden del af den kode der genereres, ved opbygning af et linkelements repræsentation, er kode der udtrækker de nødvendige funktionsargumenter fra de dynamiske elementer.

Dette gøres ved at benytte følgende fremgangsmåde:

1. Ved at benytte signaturen for typerne, oprettes funktioner der udtager de ønskede dele af en type.
2. Kald funktionerne med de relevante resultater fra funktionsbiblioteket.
3. Sammensæt alle mulige kombinationer af de fundne argumenter, ved at benytte hjælpefunktionen der genererer cartetiske produkter af lister.

For det generelle eksempel der benyttes i dette afsnit, genereres følgende MSP-kode.

```
val valList =
 let
  fun bFromd1 (x,_) = [Int.toString x]
  fun bFromd0 (_,L) = appfun(List.map bFromd1 L)
 in (bFromd0 d1) end
```

Som det kan ses genereres funktionen `bFromd0`, til at udtrække værdier af typen `b`, fra værdier af typen `d`. Samtidigt foretages konvertering af værdierne til strenge.

Det skal her bemærkes, at der i eksemplet kun skal udtrækkes værdier til ét argument, og at der således ikke benyttes kombinering af de fundne argumenter.

Kald af linkgenerator funktion

Sidste del af den kode der genereres, ved opbygning af et linkelements repræsentation, har ansvaret for at kalde de genererede funktioner.

For det generelle eksempel der benyttes i dette afsnit, genereres følgende MSP-kode.

```
prmap link1FormCont valList
```

Her kaldes `link1FormCont` linkgenerator funktionen, med hvert element i `valList`. Variablen `valList` blev opbygget som beskrevet ovenfor, og indeholder altså alle de forskellige værdier af typen `b`, der kunne udtrækkes af det *dynamiske* element med identifikatoren `d1`.

Resultatet bliver kode der genererer et link for hver af de fundne argumenter.

8.5.5 Visualisering af resultater

Hvis der i en *.sta* -fil er angivet at et dynamisk element skal visualiseres, er det op til den tilsvarende MSP-fil, at kalde de nødvendige funktioner fra konverteringsbiblioteket.

MSP-koden der kalder konverteringsbiblioteket, kan umiddelbart genereres givet det *dynamiske* elements identifikator, og type. Navngivningen af konverteringsfunktionerne er beskrevet i afsnit 8.5.3.

Eksempel:
Et *dynamisk* element med identifikatoren `d1`, og typen `c`, vil føre til følgende MSP-funktionskald:
`c2xhtml(d1)`

Ved at benytte de autogenererede konverteringsfunktioner sikres, som tidligere nævnt, at resultatet af en funktion aldrig vil have indflydelse på validiteten af de hypertekstdokumenter der sendes til klienten.

Kapitel 9

Test af webbaserede systemer

En af fordelene ved at have anvendt vores metode, til udviklingen af et webbaseret system, er at systemet kan testes i mindre dele. Vi har delt testen op i tre dele.

1. Test af funktionsbibliotek
2. Test af navigation
3. Test af layout

I dette kapitel vil vi beskrive hvorledes de enkelte test kan foregå. Vi vil give nogle idéer til ting der bør testes, og hvorledes test suiter kan genereres ud fra specifikationen af funktionelle krav og navigation. Vi vil i hvert afsnit referere, til litteratur, der går videre med de testmetoder vi beskriver.

9.1 Test af funktionsbibliotek

Efter at funktionsbiblioteket er implementeret, er det nødvendigt at teste biblioteket. Dette kan gøres uafhængigt af resten af systemet.

De fejl der kan eksistere i et system, kan deles op i tre kategorier:

Syntaks fejl: Funktionsbiblioteket indeholder en fejl, fordi der forekommer en konstruktion der ikke er i SML syntaksen.

Type fejl: Funktionsbibliotekets syntaks er korrekt, men det forsøges f.eks. at anvende en funktion på en værdi af forkert type (2 + "tekst").

Semantik fejl: Funktionsbibliotekets syntakts er korrekt og type checket, men der er en fejl i f.eks. en udregning.

SML hjælper godt på vej, da oversætteren kan finde fejl af de to første typer. Det er dog op til programmøren at rette fejlene.

Fejl af den tredje type kan oversætteren ikke finde, da denne kategori ikke har med programmets form, men mening at gøre. Der kan foretages strukturel og funktionel test af funktionsbiblioteket, for at finde eventuelle fejl.

Den strukturelle test tilrettelægges ud fra programmets struktur, det vil sige at der skal testes at input kan få programmet til at benytte erklæringerne i alle grene. Testen kan tilrettelægges ud fra viden om hvilke konstruktioner kildeteksten indeholder, og hvordan disse vælger de forskellige grene i programmet. Der skal så vælges et test input for hver vej i programmet.

Da vi startede med at specificere de funktionelle krav, har vi i disse, et potentielt input til en test generator, der kan give retningslinier til en funktionel test af implementeringen.

Målet for en funktionel test er at sikre, at programmet løser det problem det er meningen at det skal løse. Testdata skal laves så de dækker "normale" og "ekstreme" argumentværdier.

At en funktion kun kan tage argumenter af simpel type (`int`, `string` og `bool`) kan udnyttes til, på baggrund af funktionssignaturerne, at generere forslag til argumenter til testen.

Generelt bør alle randværdier testes for de enkelte typer. Det simple tilfælde er argumenter af typen `bool`, der kan testes med værdierne `true` og `false`. Funktionsargumenter af typen `int` har ofte randværdierne opdelt i positive og negative værdier, samt nul. Derudover er der ofte en øvre og en nedre grænse der skal testes. Funktionsargumenter af typen `string` har en randværdi i den tomme streng (“”), og ofte også i strenge der kun indeholder blanktegn. Derudover er der strenge som det vides giver, eller ikke giver, mening for systemet.

Alle værdier fra funktionsbiblioteket bliver *HTML-encoded* af konverteringsbiblioteket, inden de bliver sendt til klienten. Det er altså ikke nødvendigt at teste for ting som “uheldige” tegn[1], eller HTML elementer i strenge der er indeholdt i funktionsresultater.

Udvides metoden, så det tillades at begrænse simple typer, ved at indføre regulære udtryk, kan en testgenerator blive meget mere kraftfuld, da randværdier vil være væsentligt nemmere at fastslå.

For yderligere information om struktural og funktionel test se [17].

9.2 Test af navigation

Formålet med at teste navigationen er at sikre brugervenlig interaktion med det webbaserede system. Det er således en god idé at vurdere hvilke funktioner der skal være lettilgængelige, og for hvilke brugergrupper. Essentielle funktioner ønskes normalt at være lettilgængelige. Dette kan f.eks. undersøges ved at angive en tærskel for hvor mange links brugeren skal følge, for at nå funktionen.

For eksempel er funktionen for gennemførelse af et salg, en essentiel funktion hos en webbaseret forretning. Det kan være at de har en side, der skal tage imod forsendelsesoplysninger før funktionen kan anvendes. Derfor sættes tærsklen for funktionen til to links, fra en vilkårlig side, for kundebrugergruppen.

De angivne tærskler kan så testes med et værktøj der bruger en grafalgoritme, der tæller antallet af links det er nødvendigt at følge. For

[1] Med “uheldige” tegn menes tegn som f.eks “$<$” der for en web-browser angiver at der kommer en HTML-formateringskonstuktion.

at lette problemløsningen kunne værktøjet angive den korteste vej, gennem de sider, der kræver at for mange links skal følges.

9.3 Test af layout

Som tidligere nævnt kan CSS benyttes til at angive layout, af de hypertekstdokumenter der sendes til klienten. CSS giver muligheder for at definere hvorledes forskellige klienter skal præsentere sidernes indhold for brugerne. Dette inkluderer brugen af farver og placering af de forskellige dele på siderne.

W3C giver retningslinier for en række ting man skal overveje når man tester sit layout, se [5]. Herunder at brugerne kan anvende systemet fra mange forskellige platforme. Der findes ikke kun web-browsere til skrivebordscomputere, men også på håndholdte computere, mobiltelefoner osv. Med CSS kan man specificere layout der præsenterer sidernes indhold forskelligt, afhængigt af platformen hvorpå siden skal vises.

De forhold hvorunder systemet skal anvendes, kan også have indflydelse. Hvis f.eks. systemet skal benyttes i meget lyse eller mørke rum, bør det sikres at der ikke benyttes for lyse farver eller farver i tætliggende nuancer.

Det er vigtigt at tage brugerens begrænsninger i betragtning. Hvis man f.eks. udvikler et system til svagtseende eller blinde, er det ikke et visuelt layout man skal bruge, men placering og rækkefølge af indholdet der bør komme i første række.

Ved at følge retningslinierne fra W3C, er det muligt selv at certificere sit webbaserede system i tre niveauer: *Conformance Level* A, Double-A og Triple-A.

De enkelte certificeringer indeholder også krav der er mere indholdsbetonede. F.eks. indeholder første niveau (*Conformance Level* A) krav om at alt visuel information også skal være angivet som tekst, ved at benytte `alt` eller `longdesc` attributerne der er tilrådighed på alle billedelementer.

CSS og *.sta* filer kunne analyseres af et værktøj, der markerer eller giver advarsler om de områder, der indeholder problemer i forhold til en ønsket certificering. En væsentlig del af kravene kan checkes automatisk, og derved lette certificering af et webbaseret system.

Se [5] for yderligere information, om kriterier for godt layout, og A, Double-A og Triple-A certificering.

Kapitel 10

Dokumentation af webbaserede systemer

Det er vigtigt at designet af et system dokumenteres. Dels for at flere kan arbejde sammen om udviklingen, dels for at gøre det muligt at overgive systemet til vedligeholdelse dette gælder både for systemer generelt, og for webbaserede systemer.

Følges vores metode ved udviklingen er der mange informationer at hente til dokumentation for, hvorledes systemet er designet. I dette kapitel vil vi beskrive hvorledes informationerne kan deles op, og bruges til at dokumentere systemet.

Dokumentationen er inddelt i tre afsnit:

- Kravene
- Designet
- Implementeringen

I afsnittet *kravene* dokumenteres kravene til systemets funktionalitet. Det vises hvorledes informationerne samlet i kravspecifikationen kan anvendes hertil.

I afsnittet *designet* dokumenteres de beslutninger der er taget vedrørende designet af systemet. Afsnittet baseres på informationer om hvorledes databasen og interaktionen i form af navigationsdiagrammer er designet.

I afsnittet *implementeringen* dokumenteres implementeringen af systemet. I vores metode foretages dele af implementeringen af værktøjer, derfor består dokumentationen af inputtet til værktøjet.

10.1 Kravene

Ved udvikling af et webbaseret system angives funktionaliteten som det første i kravspecifikationen. Funktionaliteten i systemet er det centrale i udviklingsmetoden og bliver derfor også det centrale i dokumentationen.

Dokumentationen skal kun indeholde funktioner der er tilgængelige for brugerne af det webbaserede system [18]. Dem der skal bruge dokumentationen kan derved lettere overskue systemet, og f.eks. vedligeholdelse af system gøres nemmere.

Kravene til systemet kan dokumenteres på følgende måde:

```
Funktionsnavn [brugergruppe] -
  Funktionsnavn: type_1 * ... * type_n -> type
  En beskrivelse af funktionaliteten
```

Dokumentationen for en funktion indeholder typer for argumenter og resultat. Brugergruppen er angivet og en beskrivelse af funktionaliteten.

Hvis systemet er meget omfattende kan dokumentationen eventuelt inddeles efter brugergrupperne. Dokumentation kan gøres mere overskuelig ved at vedlægge et træ over brugergruppe hierarkiet.

10.2 Designet

Designet af systemet skal dokumenteres ved information om de beslutninger der er taget for at adaptere systemet til arkitekturen for webbaserede systemer.

Dokumentation for design af funktionaliteten er givet i *proj.sig* , der indeholder signaturer for funktionsbiblioteket. Designet af databasen dokumenteres med det tilrettede E/R-diagram. Navigationen er designet i et navigationsdiagram. Det udgør dokumentation for designet af navigationen.

10.3 Implementeringen

Implementeringen af systemet dokumenteres af de valg der er taget ved implementeringen. Vi har angivet metoder til at nå fra de indledende skridt til en implementering, og disse kan ofte genereres automatisk, derfor er kravene og designet de vigtigste dele af dokumentationen.

Den genererede implementation kan suppleres med en beskrivelse af biblioteksstrukturen der er resultatet af de identificerede brugergrupper.

Dokumentationen bør yderligere ledsages af de SQL-sætninger der oprettede databasen. Konfigurationsfiler for repræsentationen af de statiske elementer og opbygningen af siderne er givet i filerne *.demand.sta* og *.sta* . Layout kan angives i et vedlagt CSS. Disse filer er også dem der skal tilrettes for at ændre i det webbaserede system.

10.4 Vedligeholdelse og genbrugelighed

Vi vil her beskrive nogle fordele ved at udvikle et webbaseret system med den beskrevne metode og ved at dokumentere systemet som angivet ovenfor. Ved at følge angivelserne får man et system der er delt op efter funktionalitet, navigation og layout. Hver af disse kan ændres uafhængigt af hinanden.

Systemet er derfor nemt at vedligeholde da en ændring hurtigt og enkelt kan indføres i de dele af systemet hvor det er nødvendigt. Værktøjer der genererer det færdige system køres, og ændringen vil herved brede sig til hele systemet.

Hvis værktøjet *tagmerger* er anvendt fornuftigt, så er de statiske informationer i systemet kun er angivet et sted. Derfor vil det altid være nemt at vende tilbage og rette hvor det er nødvendigt. Dette er specielt rart ved vedligeholdelsen af store systemer.

Vi har hermed vist at vores værktøjsbaserede udviklingsmetode ikke kun genererer et webbaseret system, men også giver væsentlige elementer af dokumentationen.

Kapitel 11

Ideer og videre arbejde

Vi er under uarbejdelsen af vores metode kommet på flere ideer, end vi kunne nå at undersøge relevansen af. I dette kapitel vil vi beskrive de ideer til videre arbejde med udvikling af pålidelige webbaserede systemer, vi føler kan bidrage mest til at forbedre metoden og de udviklede systemer.

Kapitlet er delt i tre dele. Først beskrives nogle udvidelser der vil gøre metoden lettere at arbejde med til praktiske formål. Herefter ideer til værktøjer der kunne generere andre dele af det webbaserede system. Endeligt nogle ideer til andre fremgangsmåder der kunne prøves sammen med vores system.

11.1 Udvidelser af metoden

Skal metoden anvendes til design og implementering af et større webbaseret system, kan nogle af følgende ideer lette arbejdet.

Typeomgivelserne

Typeomgivelserne kan udvides med flere simple typer (f.eks. decimaltal) og konstruktorer for komplekse typer, som grafer, træer og mængder. Det kræver at der ændres i metoden så de nye typeerklæringer kan behandles, ækvivalenter i databasen skal findes og retningslinier for konverteringen i navigationsgrænsefladen gives.

De simple typer kunne også begrænses, med f.eks. regulære udtryk. Der skulle angives konstruktioner for formuleringen af de begrænsede typer til kravspecifikationen. Regulære udtryk kan genereres for informationen i databasen ved begrænsnings check (med operatoren `check`). Klient-side og server-side valideringen skal ligeledes udvides med check på regulære udtryk (ML og Java Script indeholder metoder til check af regulære udtryk). De begrænsede typer vil gøre det webbaserede systems typesikkerhed endnu stærkere, og kunne bidrage til udformning af testsuiter.

Funktionssignaturerne

En udvidelse der tillader sammensatte typer som argumenter i funktionssignaturerne kunne foretages. Måden argumenterne sendes som CGI parametre skulle serialiseres for at muligøre værdier af sammensatte typer. Server-side og klient-side valideringen skulle også udvides for at være i stand til at behandle de sammensatte typer. Valideringen af de sammensatte værdier klient-side skulle tilføjes til *formscripts.js* , på samme måde som valideringen af de simple typer. Server-side valideringen af de simple typer foretages ved at anvende konverteringsfunktionerne allerede i basisbiblioteket til ML. Hvis valideringen udvides kan der tilføjes et bibliotek indeholdende valideringsfunktioner for værdier af de tiladte typer.

Versionskontrol

Ved udviklingen af et større webbaseret system ville det være rart med versionskontrol på konfigurationsfilerne. Hvis der blev ændret i funktionssignaturen eller navigationsdiagrammet så skulle det kun være ændringerne der skulle opdateres i konfigurationsfilerne.

11.2 Ideer til værktøjer

Omfanget af den del af det webbaserede system der kan genereres automatisk kan udbygges. Følgende er ideer vi har overvejet.

Automatisk normalisering af typer til databasegeneration

Typeerklæringerne kan normaliseres efter en simpel opskrift, hvor værktøjet når det identificerer en typeerklæringer der bryder den normaliserede form deles op, ved at introducerer typenavne for del udtrykkene.

Automatisk beregning af "Closure" for databaseskemaer

Når en database er designet ønskes det ofte at verificerer at den er på en af de fem normalformer. Der kan forekomme afhængigheder mellem attributter der ikke kan udledes af funktionssignaturerne. F.eks. afhængigheden mellem postnummer og by, der normalt ikke betyder at der er en funktion med signaturen:

$f : postnummer \rightarrow by$

Det er muligt for designeren af det webbaserede system at gennemgå typerne og markerer afhængighederne. Det er derfor muligt at lave et værktøj der følger metoder for verifikation af normalformaer og derved kan foretage opsplitning af databaseskemaerne, f.eks. efter metoden i [19]. Det ville være interessant hvis værktøjet yderligere satte referencielle begrænsninger på de opsplittede tabeller og oprettede *views* der samlede tabellerne, for at simplificere forespørgslerne fra funktionsbiblioteket.

Databasebibliotek

Ved at analyserer typeomgivelserne for et webbaseret system kan der genereres et databasebibliotek der kan danne en grænseflade mellem funktionsbiblioteket og databasen, her kunne typekonverteringer foretages. De næste to punkter kunne også indkluderes i et databasebibliotek.

SQL-udtryksgenerator

Ud fra viden om arkitekturen af webbaserede systemer kunne man udvide fasen i metoden for kodegenerering med et værktøj. Der kunne generer et implementeringsskelet for funktionsbiblioteket. Dette bibliotek skulle indeholde funktionserklæringerne som i prototype biblioteket. Yderligere kan vi ud fra funktionssignaturerne genererer kode der opretter en forbindelse til databaseserveren, som den rigtige bruger, og lukker forbindelsen efter funktionen afsluttes.

Hvis der foretages en kategorisering af funktionssignaturer der skriver i databasen (funktioner med en sidevirkning, `unit`) og de annoteres med deres brug af databasen, dvs. om de indsætter, opdaterer eller sletter information i tabellerne, kan der genereres skeletter for SQL-forespørgelserne. Den komplette forespørgsel kan genereres, hvis funktionen er en front-end funktion. Følgende er et eksemple der kan give en ide om hvorledes man kunne generere SQL forespørgsler:

$f : CourseId \rightarrow Course$ (front-end funktion)

Feltet for $CourseId$ er valgt som nøgle for tabellen for $Course$, følgende forespørgsel kan genereres:

```
SELECT *
FROM Course
WHERE CourseId = 'argument til f'
```

`SELECT` og `FROM` linien genereres ud fra resultatet for funktionen, `WHERE` genereres ud fra argumentet.

Automatisk låsning af tabeller i databasen

Hvilke tabeller en funktion anvender i databasen kan man analysere sig frem til og disse tabeller kan herefter låses under funktionens afvikling. Funktionen ovenfor (f) kan låse tabellen $Course$ umiddelbart efter at der er etableret en forbindelse til databaseserveren. Transaktionslåsning kan foregår som følger:

```
CONNECT TO databasenavn AS forbindelse USER brugernavn;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
...
COMMIT;
```

Der vil startes en transaktion der sikrer at der ikke er andre der ændre i en tabel, efter at funktionen har benytte den, inden der sendes en afslutning på transaktionen.

Hvis der opstår en fejl fra databasen efter at transaktionen er påbegyndt vil ML biblioteket der styrer kommunikationen med databaseserveren returnerer en exception, der kan behandles ved at sende en `ROLLBACK` der annullerer hele transaktionen.

Generering af .htaccess filer

Som tidligere beskrevet placeres de genererede filer i en biblioteksstruktur der svarer til brugergrupperne. Oversagen til dette er at det muligør at benytte webserverens restriktionsfacilitet til adgangsbegrænsning. Restriktionerne defineres i *.htaccess* -filer.

Et værktøj for generering af *.htaccess* -filer kunne tilføjes i fasen for kodegenerering. Vores metode benytter en bibliotekets struktur der er klar til *.htaccess* , denne skulle genereres i *.htaccess* -filen, og der skulle genereres brugere og kodeord.

CSS generering

Vi har i metoden ikke beskæftiget os med layout af det udviklede webbaserede system, men *.MSP* filerne er inddelt med `<div>` tags og ved indsættelse af dynamiske genererede elementer benyttes attributten `class` til at inddele værdierne i klasser. Der er muligt ud fra dette at genererer CSS filer der indeholder erklæringer for et skelet i layoutet af dokumentet.

På den måde er der en grænseflade mellem layout og navigation der angiver hvilke klasser der er til stede i navigationsfilerne.

Automatisk layout validering

Som beskrevet i kapitel 9 kan layoutet valideres efter nogle retningslinier givet i [5]. Der er mange af disse retninglinier der er på en form, der er mulig at validerer med et værktøj. Det kunne derfor lette testen af layoutet at udvikle et værktøj der kunne give fejl beskeder og advarsler baseret på en analyse af *.msp* og *.css* filer.

11.3 Andre fremgangsmåder

Vi vil nu kigge på nogle fremgangsmåder der kunne erstate dele af vores metode eller bidrage med noget vi ikke har arbejdet med.

Normaliser til CNF => database på BCNF

Frem for at overlade verificeringen af om databasen overholder en normalform, kunne typeomgivelserne betragtes som en contekst-fri grammatik, og føres på Chomskys Normal Form (CNF), vi opfatter simpel typer som alfabetet og de komplekse typer som produktionssymbolerne. Typeomgivelser angivet som en grammatik på CNF indeholder produktioner på to former:

- $typeNavn = simpelType$
- $typeNavn = typeNavn * typeNavn$

Det betyder at når typeomgivelserne benyttes til at generere en database vil alle tabeller være forbundet til netop to komponenter, felter eller tabeller. I [19] vises at når en tabel indeholder to komponenter vil den være på Boyce-Codd Normal Form.

Caching af dokumenter

Mange webbaserede system formidler informationer der bliver vist for brugeren oftere end den ændres. Vi har overvejet en metode hvor designeren markerer sider der ofte skal hente de samme oplysninger fra databasen. Ud fra denne information kan der laves et system der holder øje med ændringer i databasen. Hvis der sker en ændring i den del af databasen der anvendes i et cachet dokument, genereres dokumentet på ny. Når en bruger forespøger dokumentet, behøver webserveren ikke bruge tid på behandlingen af dynamiske elementer, men kan istedet sende det cachede dokument til brugeren.

Generer semi-struktureret database

Alle informationer præsenteret gennem de webbaserede systemer der udvikles med vores metode er gemt i en relations database. Det kunne forsøges at anvende en semistruktureret database istedet. Type omgivelserne er på en form der kan opfattes som en contekst-fri grammatisk, ud fra dette kan der laves et værktøj der kan oversætte typeomgivelserne til en DTD eller et XML Schema for indholdet i den semistrukturerede database.

Der er dog stadig en del ting omkring semistrukturerede databaser der ikke er afklaret, som hvorledes der kan sikres brugeradgangsbegrænsninger og referenciel integritet. Der findes heller ikke nogen standard (endnu) for forespørgselssprog for semistrukturerede databaser.

Anvendelse af cookies/sessions

Et alternativ til at gemme oplysninger på en databaseserver er at gemme oplysningerne på brugerens computer. Vi har valgt relations databasen da den kan indeholde alle oplysninger, både brugeroplysninger og dem der præsenteres igennem det webbaserede system. Bruger relaterede oplysninger kunne anbriges i cookies eller sessions. Der er grunde for og imod denne fremgangsmåde, og dem må man veje mod de oplysninger der bliver gemt.

Fordelene ved at gemme oplysningerne på brugersiden er at man sparer plads og belastning på databaseserveren. Ulemperne er at oplysningerne ikke sikres, da de forlader det aflukkede område bag firewallen, og oplysningerne er ikke til rådighed til datamining.

For eksemple kan vi betragte en internet forretning der ønsker at opbevarer oplysninger om varerkatalog, kundernes ordre og indkøbsvogne. Varerkataloget er nødt til at befinde sig på en databaseserver. Kundernes ordre og indkøbsvogne kan opbevares på databaseserveren eller hos brugeren. Kundernes ordre bør nok opbevares hos internet forretningen da de kan få brug for oplysningerne senere, til regnskaber osv. Kundernes indkøbsvogn kan opbevares hos kunden da databasen ellers kan komme til at indeholde mange indkøbsvogne der aldrig fører til en bestilling fordi kunden har ombestemt sig før ordren er givet.

Man kunne udvide metoden med en måde at annoterer funktionssignaturerne med oplysninger om hvorvidt der skal anvendes cookies eller session istedet for databasen. Der kunne laves et værktøj der kunne genererer MSP kode til at foretage behandlingen af informationerne af cookies og sessions.

Kapitel 12

Konklusion

Denne rapport dokumenterer udviklingen af en metode, med tilhørende værktøjer, til teoribaseret udvikling af pålidelige webbaserede systemer. Vores mål var at komme med en værktøjsunderstøttet *software engineering* metode til konstruktion af pålidelige webbaserede systemer.

Fordelene ved at anvende vores metode til udvikling af webbaserede systemer er at man opnår:

- Pålidelighed
- Hurtig prototypeudvikling
- Modulopdeling

12.1 Pålidelighed

I afsnit 1.3 definerede vi begrebet pålidelige webbaserede systemer, ved at give en række egenskaber systemerne skulle opfylde. Vi vil nu konkludere på hvorvidt disse egenskaber er opnået med vores metode.

Statisk validering Vi har udviklet en lang række værktøjer der statisk checker en række egenskaber. Alle steder hvor der under udvikling af systemet foretages manuelle justeringer, sikres det at disse er valide. Dette betyder f.eks. at faste skjulte værdier der er angivet i konfigurationsfiler valideres for korrekt type.

Integritet sikres i database Databasen genereres ud fra funktionssignaturerne, metoden sikrer integritetsbegrænsninger og adgangsbegrænsninger. Værktøjerne *ergen* og *sqlgen* sikrer integritet i databasen, se kapitel 6.

Typesikre omgivelser for implementering Værktøjet *ProjGen* genererer en ML signaturfil. ML oversætteren sikrer på baggrund af denne at implementeringen overholder typeangivelserne. Se kapitel 8.

Navigationssikkerhed Navigationsdiagrammer valideres som beskrevet i kapitel 7. Implementeringen af navigationen er værktøjsbaseret og værktøjet sikrer korrekt generering af brugerinteraktionen angivet i navigationsdiagrammerne.

Browsersikre sider Alle konfigurerbare filer testes som beskrevet i afsnit 8.4.2. Da kun XHTML 1.0 Strict tillades vil hypertekstgeneratorerne altid returnere browsersikre sider.

Brugerrettigheder sikres Når navigationsdiagrammet valideres testes det samtidigt at alle brugere har adgang til de funktioner og sider der er specificeret og ikke andre. Se kapitel 7.

Typecheck af parametre Genereringen af kode sikrer at der foretages typecheck af alle parametre, på både klientside og serverside. Se kapitel 8.

Metoden sikrer i denne forstand pålidelige webbaserede systemer.

12.2 Hurtig prototypeudvikling

Vores metode kræver at funktionssignaturerne og navigationsdiagrammet specificeres. På baggrund af disse kan en prototype af det webbaserede system genereres. Vi har vist at det er muligt at assistere designeren i mange af de beslutninger der skal tages ved udviklingen.

Dette kan gøres på baggrund af den viden vi har samlet omkring webbaserede systemer og kravene til det specifike system.

Sammenligner vi vores metode med andre metoder indenfor området har vi ikke set nogen der kan fremstille en prototype lige så hurtigt. For alle implementeringsplatformene gælder det at der skal væsentligt mere omfattende kode til, end de funktionssignaturer og det navigationsdiagram som vi genererer prototypen ud fra.

De objektorienterede metoder beskrevet i [15] starter på et lavere abstraktionsniveau end vores metode. De giver dokumentation og specifikation af systemet og letter derved implementationsprocessen, men der er ingen understøttelse af værktøjer der kan hjælpe med at generere kode.

12.3 Modulopdeling

Vores arbejde er centreret omkring de fire hovedfaser i metoden. For hver fase har vi udviklet begrebsdannelser, teorier og værktøjer. Begrebsdannelserne har været fundamentet for vores metode og har resulteret i en bredere forståelse af webbaserede systemer. På fundamentet er der udviklet teorier for hvorledes man når fra en kravspecifikation til et færdigt webbaseret system. Til at assistere ved udviklingen er der implementeret en række værktøjer.

Det er vist at et webbaseret system kan deles op i moduler for funktionalitet, navigation, indhold og layout. Dette er en fordel når systemet skal implementeres, testes og vedligeholdes. Den klare opdeling der sikres af grænseflader mellem modulerne, gør det muligt at implementere hvert modul for sig.

Ligeledes muliggøres separat test af hvert modul. Prototypeimplementationer gør test af navigation og layout mulig på et tidligt stadie.

Dokumentationen kan samles fra de informationer der har repræsenteret det webbaserede system under udviklingen. Med kendskab til metoden er det overskueligt at vedligeholde systemet. Det er en fordel at ændringer til funktionaliteten blot skal rettes i funktionssignaturerne og i funktionsbiblioteket, mens resten af systemet automatisk vedligeholdes af værktøjerne.

De metoder vi har set der bygger på objektorienteret udvikling begynder med at specificere en konceptmodel over objekterne i systemet, for herefter at specificere funktionalitet og navigation. Der er ofte problemer i denne fremgangsmåde da det er svært at foretage en opdeling der er hensigtsmæssig i forbindelse med implementering og test.

12.4 Afrunding

Arbejdet med udviklingen af værktøjerne har givet os mulighed for at stikke en smule dybere, end det normalt er muligt ved en ren udvikling af teorier. Vi har fået en indsigt i nogle af de konkrete problemer der er forbundet med udviklingen af webbaserede systemer.

Værktøjerne er testet på eksempelsystemer som det der er beskrevet i kapitel 4, systemet fra [2] og udvalgte specifikke tests af værktøjerne. Vi har dog ikke haft tid til at teste værktøjerne som beskrevet i kapitel 9, men vi er klar over at dette burde gøres.

Vi mener hermed at vi har banet vejen for en værktøjsunderstøttet *software engineering* metode til konstruktion af pålidelige webbaserede systemer. Som beskrevet i kapitel 11 er der mange potientielle udvidelser til metoden, som det kunne være interesseant at arbejde videre med.

Vi mener at den metode vi har beskrevet i denne rapport er tæt på at være fuldt anvendelig til praktisk udvikling af pålidelige webbaserede systemer.

Litteratur

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the web.* Morgan Kaufmann Publishers, 2000.

[2] Niels Bach. En metode til udvikling af web-baserede systemer. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2001.

[3] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The <bigwig> project.

[4] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated html, 2001.

[5] Wendy Chisholm, Gregg Vanderheiden, and Ian Jacobs. Web content accessibility guidelines 1.0, 1999.

[6] Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the world-wide web: A survey. *ACM SIGMOD Record*, 27(3):59–74, 1998.

[7] Athula Ginige and San Murugesan. Guest Editors' introduction: Web engineering: An introduction. *IEEE MultiMedia*, 8(1):14–18, January 2001.

[8] M. R. Hansen and H. Rischel. *Introduction to Programming using SML.* Addison Wesley Longman, 1999.

[9] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 2001.

[10] William Kent. A simple guide to five normal forms in relational

database theory. *Communications of the ACM*, 26(2):120–125, February 1983.

[11] S. Larsen and M. R. Hansen. Systematic design and implementation of web-based systems. In *International Symposium on Telecommunications - IST2001*, pages 38–41. ITRC, 2001.

[12] Ted Nelson. A file structure for the complex, the changing, and the indeterminate. *In Proceedings of the 20-th National ACM Conference*, pages 84–100, August 1965.

[13] Steven Pemberton and et al. Xhtml 1.0 the extensible hypertext markup language (second edition), August 2002.

[14] Filippo Ricca and Paolo Tonella. Understanding and restructuring web sites with reweb. *IEEE MultiMedia*, 8(2):40–51, April 2001.

[15] Gustavo Rossi, Daniel Schwabe, and Fernando Lyardet. Web application models are more than conceptual models. In *ER Workshops*, pages 239–253, 1999.

[16] Daniel Schwabe, Luiselena Esmeraldo, Gustavo Rossi, and Fernando Lyardet. Engineering Web applications for reuse. *IEEE MultiMedia*, 8(1):20–31, January 2001.

[17] Peter Sestoft. Systematic software test, July 1998.

[18] Ian Sommerville. *Software Engineering*. Addison-Wesley, sixth edition, 2001.

[19] J. D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, 2002.

Bilag A

Eksempelsystem

Genererede filer for systemet der er udviklet som eksempel i kapitel 4.

A.1 Signaturfil

```
ShowCurr : Name       -> Curr [student]
ShowCourse : CourseId -> Course [student]
AddCourse : Name * CourseId -> unit [teacher]
#
Name     = string
CourseId = int
CourseDesc = string
Course   = CourseId * CourseDesc
Courses = Course list
Curr     = Name * Courses
```

A.2 E/R-diagram

```
(Name, Attribut(SQLTEXT, KEY))
(CourseId, Attribut(SQLINT, KEY))
(CourseDesc, Attribut(SQLTEXT, NONKEY))
(Course, Entity([AttribRel(CourseDesc), AttribRel(CourseId)]))
(CourseIdx, Attribut(SQLINT, KEY))
```

```
(Courses, WeakEntity([Relation(Mark(ManyToOne(Course))), AttribRel(
    CourseIdx)]))
(Curr, Entity([WeakRelation(OneToMany(Courses)), AttribRel(Name)]))
```

A.3 SQL-sætninger

```sql
CREATE TABLE Curr(
      Name TEXT PRIMARY KEY
);

CREATE TABLE Courses(
      CourseIdref INT REFERENCES Course(CourseId),
      CourseIdx INT PRIMARY KEY,
      Nameref TEXT PRIMARY KEY REFERENCES Curr(Name)
);

CREATE TABLE Course(
      CourseId INT PRIMARY KEY,
      CourseDesc TEXT
);


CREATE USER student;
CREATE USER teacher;

GRANT SELECT ON Curr, Courses TO student;
GRANT SELECT ON Courses, Course TO student;
GRANT INSERT, DELETE, UPDATE ON Courses, Curr, Course TO teacher;
```

A.4 Navigationsdiagram

```xml
<?xml version="1.0"?>
<navigationgraph xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns="" xsi:noNamespaceSchemaLocation="../es/
    model_es.xsd">
      <webpage title="StartPage">
            <elem id="s1" type="Static">
                  <character type="Name"/>
            </elem>
            <linkelem destination="CurrInfoPage" id="link1">
                  <funapps>
                        <funapp funname="ShowCurr" dyndest="d3">
                              <arglist>
                                    <argument refelem="s1" ctype="
                                        Name"/>
                              </arglist>
```

```
                </funapp>
            </funapps>
        </linkelem>
    </webpage>
    <webpage title="CurrInfoPage">
        <elem id="s2" type="Static">
            <character type="CourseId"/>
        </elem>
        <elem id="d3" type="Dynamic">
            <character type="Curr"/>
        </elem>
        <linkelem destination="CurrChanPage" id="link2">
            <funapps>
                <funapp funname="AddCourse" dyndest="d5">
                    <arglist>
                        <argument refelem="d3" ctype="
                            Name"/>
                        <argument refelem="s2" ctype="
                            CourseId"/>
                    </arglist>
                </funapp>
            </funapps>
        </linkelem>
        <linkelem id="link3" destination="CourseInfoPage">
            <funapps>
                <funapp funname="ShowCourse" dyndest="d4">
                    <arglist>
                        <argument refelem="d3" ctype="
                            CourseId"/>
                    </arglist>
                </funapp>
            </funapps>
        </linkelem>
    </webpage>
    <webpage title="CourseInfoPage">
        <elem id="d4" type="Dynamic">
            <character type="Course"/>
        </elem>
    </webpage>
    <webpage title="CurrChanPage">
        <elem id="d5" type="Dynamic">
            <character type="unit"/>
        </elem>
    </webpage>
    <alphabet>
        <ctype name="unit">
            <unit/>
        </ctype>
        <!--
```

```
        Name = string
        CourseId = int
        Curr = Name * Course list
        CourseDesc = string
        Course = CourseId * CourseDesc
        -->
        <ctype name="Name">
            <stype>
                <string/>
            </stype>
        </ctype>
        <ctype name="CourseId">
            <stype>
                <int/>
            </stype>
        </ctype>
        <ctype name="Curr">
            <carprod>
                <refctype>
                    <ref reference="Name"/>
                </refctype>
                <refctype>
                    <list>
                        <ref reference="Course"/>
                    </list>
                </refctype>
            </carprod>
        </ctype>
        <ctype name="CourseDesc">
            <stype>
                <string/>
            </stype>
        </ctype>
        <ctype name="Course">
            <carprod>
                <refctype>
                    <ref reference="CourseId"/>
                </refctype>
                <refctype>
                    <ref reference="CourseDesc"/>
                </refctype>
            </carprod>
        </ctype>
    </alphabet>
    <funsigs>
        <funsig funname="ShowCurr" usergroup="Student">
            <args>
                <arg type="Name"/>
            </args>
```

```
                <res type="Curr"/>
            </funsig>
            <funsig funname="ShowCourse" usergroup="Student">
                <args>
                    <arg type="CourseId"/>
                </args>
                <res type="Course"/>
            </funsig>
            <funsig funname="AddCourse" usergroup="Teacher">
                <args>
                    <arg type="Name"/>
                    <arg type="CourseId"/>
                </args>
            </funsig>
        </funsigs>
        <startpage side="StartPage"/>
        <accessspec>
            <usergroup groupname="Student" subdir="/">
                <accessright side="StartPage"/>
                <accessright side="CurrInfoPage"/>
                <accessright side="CourseInfoPage"/>
            </usergroup>
            <usergroup groupname="Teacher" subdir="/participant/">
                <include group="Student"/>
                <accessright side="CurrChanPage"/>
            </usergroup>
        </accessspec>
</navigationgraph>
```

A.5 Proj.sig

```

signature Proj = sig

type Name     = string
type CourseId = int
type CourseDesc = string
type Course  = ( CourseId * CourseDesc )
type Curr    = ( Name * Course list )

val ShowCurr : Name -> Curr
val ShowCourse : CourseId -> Course
val AddCourse : Name * CourseId -> unit

end
```

A.6 Proj.sml

```
(*automatically generated test implemetation*)
type Name    = string
type CourseId = int
type CourseDesc = string
type Course  = ( CourseId * CourseDesc )
type Curr    = ( Name * Course list )


fun ShowCurr (NameArg) = (NameArg, [(1, "string:CourseDesc" )] )
fun ShowCourse (CourseIdArg) = (CourseIdArg, "string:CourseDesc" )
fun AddCourse (NameArg, CourseIdArg) = ()
```

A.7 CourseInfoPage.sta

```
<sta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns=""
     xsi:noNamespaceSchemaLocation="sta.xsd">
<div id="dummy"/>
</sta>
```

A.8 CurrChanPage.sta

```
<sta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns=""
     xsi:noNamespaceSchemaLocation="sta.xsd">
<div id="dummy"/>
</sta>
```

A.9 CurrInfoPage.demand.sta

```
<demandsta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="" xsi:noNamespaceSchemaLocation="demand.sta.xsd">
<elem id="s2">
  <character type="CourseId">
    <input type="text" name="s2" />
  </character>
</elem>
</demandsta>
```

A.10 CurrInfoPage.sta

```
<sta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns=""
     xsi:noNamespaceSchemaLocation="sta.xsd">
<submit id="link2" destination="CurrChanPage">
 <input type="submit" value="CurrChanPage" />
</submit>
<submit id="link3" destination="CourseInfoPage">
 <input type="submit" value="CourseInfoPage" />
</submit>
</sta>
```

A.11 StartPage.demand.sta

```
<demandsta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="" xsi:noNamespaceSchemaLocation="demand.sta.xsd">
<elem id="s1">
 <character type="Name">
  <input type="text" name="s1" />
 </character>
</elem>
</demandsta>
```

A.12 StartPage.sta

```
<sta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns=""
     xsi:noNamespaceSchemaLocation="sta.xsd">
<submit id="link1" destination="CurrInfoPage">
 <input type="submit" value="CurrInfoPage" />
</submit>
</sta>
```

A.13 CourseInfoPage.msp

```

<?MSP
open Msp
open codegenlib
open sig2xhtml

 val formCounter = ref 0
 fun getFormCount() = Int.toString(!formCounter)
 fun getFormCountInc() = (formCounter := (!formCounter + 1) ;
      getFormCount() )
?>

```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD␣XHTML␣1.0␣Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
  <head>
    <meta http-equiv="Content-Type" content="text/html;␣charset=iso-8859-1" />
    <title>CourseInfoPage</title>
    <script type="text/javascript" src="formscripts.js"></script>
  </head>
  <body>
<?MSP
open Mosmlcgi
exception badtypedCGIParameter
exception illegalFunction of string
open Proj (* Contains all functions *)

val d4 = let val funName = % "d4fun"
   in case funName of
        "ShowCourse" =>
             let
                val arg0 = case ( Int.fromString(%(funName^"arg0")) )of
                                  SOME(i) => i
                                | NONE => raise badtypedCGIParameter
             in
                ShowCourse(arg0)
             end
      | _ => raise illegalFunction("d4:"^funName)
   end
?>
    <div id="dummy" />
  </body>
</html>
```

A.14 CurrChanPage.msp

```

<?MSP
open Msp
open codegenlib
open sig2xhtml

  val formCounter = ref 0
  fun getFormCount() = Int.toString(!formCounter)
  fun getFormCountInc() = (formCounter := (!formCounter + 1) ; getFormCount() )
?>

```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD␣XHTML␣1.0␣Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
  <head>
    <meta http-equiv="Content-Type" content="text/html;␣charset=iso-8859-1" />
    <title>CurrChanPage</title>
    <script type="text/javascript" src="formscripts.js"></script>
  </head>
  <body>
<?MSP
open Mosmlcgi
exception badtypedCGIParameter
exception illegalFunction of string
open Proj (* Contains all functions *)

val d5 = let val funName = % "d5fun"
   in case funName of
        "AddCourse" =>
             let
                val arg0 = % (funName^"arg0")
                val arg1 = case ( Int.fromString(%(funName^"arg1")) )of
                                  SOME(i) => i
                                | NONE => raise badtypedCGIParameter
             in
                AddCourse(arg0, arg1)
             end
      | _ => raise illegalFunction("d5:"^funName)
   end
?>
    <div id="dummy" />
  </body>
</html>
```

A.15 CurrInfoPage.msp

```

<?MSP
open Msp
open codegenlib
open sig2xhtml

  val formCounter = ref 0
  fun getFormCount() = Int.toString(!formCounter)
  fun getFormCountInc() = (formCounter := (!formCounter + 1) ; getFormCount() )
?>

```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
    <title>CurrInfoPage</title>
    <script type="text/javascript" src="formscripts.js"></script>
  </head>
  <body>
<?MSP
open Mosmlcgi
exception badtypedCGIParameter
exception illegalFunction of string
open Proj (* Contains all functions *)

val d3 = let val funName = % "d3fun"
   in case funName of
        "ShowCurr" =>
             let
                val arg0 = % (funName^"arg0")
             in
                ShowCurr(arg0)
             end
      | _ => raise illegalFunction("d3:"^funName)
   end
?>
<?MSP$
let
 fun link2FormCont (AddCoursearg0) = $ (flattenFrags `
    <div id="CurrChanPageLink^(getFormCountInc())">
      <form id="CurrChanPageLinkForm^(getFormCount())" method="get" action="/participant/CurrChanPage.msp" onsubmit="return submitCurrChanPageLinkForm^(getFormCount())();">
        <div>
          <input name="d5fun" type="hidden" value="AddCourse" />
          <input name="AddCoursearg0" type="hidden" value="^AddCoursearg0" />
          <input name="AddCoursearg1" type="text" />
          <a href="javascript:void(submitCurrChanPageLinkForm^(getFormCount())());">
            <input type="submit" value="CurrChanPage" />
          </a>
        </div>
      </form>
<script type="text/javascript">
 function submitCurrChanPageLinkForm^(getFormCount())() {
   return submitForm('CurrChanPageLinkForm^(getFormCount())', [['AddCoursearg1','int']]);
```

```
 }
</script>
   </div>
')
val valList =
 let
  fun NameFromCurr0 (x,_) = [x]
 in (NameFromCurr0 d3) end
in
 prsep Empty link2FormCont valList
end
?>
<?MSP$
let
 fun link3FormCont (ShowCoursearg0) = $ (flattenFrags `
   <div id="CourseInfoPageLink^(getFormCountInc())">
     <form id="CourseInfoPageLinkForm^(getFormCount())" method="get"
          action="/CourseInfoPage.msp" onsubmit="return 
         submitCourseInfoPageLinkForm^(getFormCount())();">
       <div>
         <input name="d4fun" type="hidden" value="ShowCourse" />
         <input name="ShowCoursearg0" type="hidden" value="^
              ShowCoursearg0" />
         <a href="javascript:void(submitCourseInfoPageLinkForm^(
              getFormCount())());">
           <input type="submit" value="CourseInfoPage" />
         </a>
       </div>
     </form>
<script type="text/javascript">
 function submitCourseInfoPageLinkForm^(getFormCount())() {
   return submitForm('CourseInfoPageLinkForm^(getFormCount())', []);
 }
</script>
   </div>
')
val valList =
 let
  fun CourseIdFromCurr1 (x,_) = [Int.toString x]
  fun CourseIdFromCurr0 (_,L) = appfun(List.map CourseIdFromCurr1 L)
 in (CourseIdFromCurr0 d3) end
in
 prsep Empty link3FormCont valList
end
?>
  </body>
</html>
```

A.16 StartPage.msp

```
<?MSP
open Msp
open codegenlib
open sig2xhtml

 val formCounter = ref 0
 fun getFormCount() = Int.toString(!formCounter)
 fun getFormCountInc() = (formCounter := (!formCounter + 1) ;
     getFormCount() )
?>

<!DOCTYPE html PUBLIC "-//W3C//DTD␣XHTML␣1.0␣Strict//EN" "http://
    www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
 <head>
  <meta http-equiv="Content-Type" content="text/html;␣charset=iso
      -8859-1" />
  <title>StartPage</title>
  <script type="text/javascript" src="formscripts.js"></script>
 </head>
 <body>
<?MSP$
let
 fun link1FormCont () = $ (flattenFrags `
  <div id="CurrInfoPageLink^(getFormCountInc())">
   <form id="CurrInfoPageLinkForm^(getFormCount())" method="get"
       action="/CurrInfoPage.msp" onsubmit="return␣
       submitCurrInfoPageLinkForm^(getFormCount())();">
    <div>
     <input name="d3fun" type="hidden" value="ShowCurr" />
     <input name="ShowCurrarg0" type="text" />
     <a href="javascript:void(submitCurrInfoPageLinkForm^(
         getFormCount())());">
      <input type="submit" value="CurrInfoPage" />
     </a>
    </div>
   </form>
<script type="text/javascript">
 function submitCurrInfoPageLinkForm^(getFormCount())() {
  return submitForm('CurrInfoPageLinkForm^(getFormCount())', [['
      ShowCurrarg0','string']]);
 }
</script>
  </div>
`)
```

```
val valList = [()]
in
 prsep Empty link1FormCont valList
end
?>
  </body>
</html>
```

Bilag B

Verificering af navigationsdiagrammer

I dette appendiks beskrives de tests værktøjet *CheckNavDiag*, udfører for at validere et navigationsdiagram.

Værktøjet *CheckNavDiag* udfører følgende tests på en givet XML-fil med et navigationsdiagram.

Først valideres navigationsdiagrammet med Schema-filen *navdiag.xsd* . Herved sikres, at følgende egenskaber er overholdt:

- At der kun er korrekt afsluttede tags og at der ikke er nogle tekstsegmenter.
- Strukturen overholdes, det vil sige, at der kun er de forventede elementer i XML-filen. F.eks. vil den sikre at, hvis et *funapp*-element[1] indeholder et *arglist*-element[2], så skal der være mindst 1 *argument*-element indeholdt i *arglist*-elementet.
- At attributter skal være af korrekt type. F.eks. vil den sikre at *type*-attributten for et *elem*-element skal være enten “Static” eller “Dynamic”.

[1]Element der definerer en funktionsapplikation.
[2]Element der indeholder en liste af argumentelementer.

- At identifikatorer rent faktisk er unikke. F.eks. vil den sikre, at der ikke vil være flere funktioner med samme navn.
- At referencer altid peger på et lovligt element. F.eks. indeholder et *linkelem*-element en reference til linkets destination, og det sikres, at det, der refereres, er et *webpage*-element.

Det har i nogle tilfælde været nødvendigt at anvende flere begrænsninger på samme element. Et eksempel er: Når et *argument*-element refererer til det *elem*-element, værdien skal komme fra, så skal det sikres, at *elem*-elementet er indeholdt i samme *webpage*-element, som det *linkelem*-element, argumentet er indeholdt i. Samtidigt skal et *funapp*-element kunne referere til det *elem*-element hvori funktionens resultat skal placeres. Her skal der altså også refereres til et *elem*-element, men denne gang skal det være muligt at referere til *elem*-elementer indeholdt i et vilkårligt *webpage*-element. Det betyder, at der i XML-Schemaet skal være to begrænsninger på identifikatoren i et *webpage*-element. En der gælder inden for hvert *webpage*-element, og en der gælder for alle *webpage*-elementerne.

Ingen cykliske typer.

Når navigationsdiagrammet er blevet valideret korrekt mod XML-Schemaet, så skal det sikres, at det ikke indeholder cykliske typedefinitioner. Dette gøres ved rekursivt at traversere typedefinitionerne og samtidig markere de besøgte definitioner. Hvis en i forvejen markeret typedefinition mødes, så meldes fejl.

Typer i funktionsapplikationer og funktionssignaturer passer sammen.

Når vi er sikre på, at der ikke eksisterer cykliske typedefinationer, så skal det sikres, at typerne for funktionsapplikationer stemmer med typerne i de tilsvarende funktionssignaturer. Dette gøres ved gennemløb af alle *funapp*-elementer i hele systemet, hvor det testes, at der er en tilsvarende funktionssignatur med samme antal argumenter, og at typerne stemmer overens for alle argumenterne.

Argumenttype passer med refereret element.

Herefter sikres det, at typen på et argument stemmer overens med det element argumentet refererer til. Dette gøres ved gennemløb af alle *funapp*-elementer i hele systemet. For de *funapp*-elementer, der indeholder argumenter, sikres for hvert argument, at dets type er den samme, eller kan udtrækkes fra typen af det *elem*-element, argumentet refererer til.

Links har passende funktionsapplikationer.

Nu sikres det, at alle dynamiske elementer altid vil kunne genereres. Dette gøres ved gennemløb af alle *webpage*-elementer i navigationsdiagrammet. For hver af disse sider sikres, at samtlige dynamiske elementer på siden vil blive genereret af alle *linkelem*-elementer i systemet, der har siden som destination.

Ingen døde sider

Dernæst sikres, at der ikke forekommer døde sider, det vil sige sider, der ikke kan nås fra startsiden ved at følge links. Dette gøres ved at:

1. Find og marker det *webpage*-element, der refereres til fra *startpage*-elementet.
2. Marker alle de *webpage*-elementer der kan nås ved rekursivt at følge referencer i de *linkelem*-elementer, der er indeholdt i de enkelte *webpage*-elementer.
3. Undersøge at navigationsdiagrammet ikke indeholder *webpage*-elementer, der ikke er markeret.

Adgangsbegrænsning er defineret for sider

Nu sikres, at der er defineret adgangsbegrænsning for alle sider. Dette gøres ved at sikre, at der er lige mange *webpage*-elementer og *accessright*-elementer i systemet. Dette er tilstrækkeligt da XML-Schemaet allerede har sikret, at alle *accessright*-elementer refererer til et *webpage*-element, og at ingen *accessright*-elementer refererer til de samme *webpage*-elementer.

Ingen cykliske brugergrupper.

Som det næste sikres, at der ikke eksisterer cykliske brugergruppe-inkluderinger. Dette gøres ved rekursivt at traversere brugergruppe-inkluderingerne og samtidig markere de besøgte brugergrupper. Hvis en i forvejen markeret brugergruppe mødes, så meldes fejl.

Rettigheder sikres og overholdes

Herefter sikres at alle brugergrupper faktisk kan komme hen til de sider, de ifølge brugergruppernes accessspecifikationer skal have lov til, og ikke andre. Samt at alle brugergrupper faktisk kan få udført de funktioner de ifølge funktionssignaturen skal have lov til, og ikke andre. Dette gøres ved, i et gennemløb af alle *usergroup* elementer, at:

1. Finde alle de sider en bruger i gruppen vil kunne nå fra startsiden ved en dybde-først-søgning, der følger links.
2. Finde alle de sider der er refereret til af *accessright*-elementer indeholdt i brugergruppen, samt inkluderede brugergrupper.
3. Undersøge at de sider, som blev refereret til af *accessright*-elementer, er præcist de samme sider, som brugeren kunne nå fra startsiden.
4. Finde alle de funktioner en bruger i gruppen vil kunne få udført ved at finde funktionsreferencer i alle funktionsapplikationer

indeholdt i links, der har både anker og destination blandt de sider, som brugeren kunne nå fra startsiden.

5. Finde alle de funktionssignaturer der refererer til brugergruppen eller inkluderede brugergrupper.
6. Undersøge at de funktioner, der blev fundet via funktionsapplikationer, er præcist de samme funktioner, som blev fundet via funktionssignaturerne.

Ingen snyd med funktionsrettigheder.

Endeligt sikres det, at brugere ikke, ved at ændre på skjulte genererede CGI-parametre, kan snyde sig til at køre funktioner, de ikke ifølge funktionsspecifikationen har ret til. Dette gøres ved i et gennemløb af alle dynamiske elementer at:

1. Finde alle funktionsapplikationer i hele systemet der refererer til dette dynamiske element som destination.
2. Undersøge at alle funktioner refereret til af funktionsapplikationerne, refererer brugergrupper, der er de samme som eller er inkluderet i den brugergruppe, der indeholder et *accessright*-element, der igen refererer til den side, der indeholder det aktuelle dynamiske element.

Bilag C

Verificering af konfigurationsfiler

I dette appendiks beskrives hvorledes værktøjet *CheckSta* validerer konfigurerbare *.sta* og *.demand.sta* filer mod en given XML-repræsentation af et navigationsdiagram.

De to typer konfigurerbare filer valideres hver for sig, og har hver deres XML Schema fil, der inkluderer begrænsede versioner af det fulde XML Schema for XHTML 1.0 Strict. Navigationsdiagrammet benyttes til at fastlægge hvilke filer der skal være i systemet, og hver af disse valideres som følger.

C.1 .sta-filer.

Først valideres *.sta* -filen med XML Schema filen *sta.xsd* . Schema filen sikrer at følgende egenskaber er overholdt:

- Alle tags er korrekt afsluttede.
- Strukturen overholdes, det vil sige at der kun er de forventede elementer i *.sta* -filen.

- Der forekommer kun lovlige attributter.
- Alle påkrævede attributter er sat, og har korrekte typer. F.eks. kan man ikke angive at der skal være et linkanker uden at angive hvilket link der er tale om.

Når *.sta* -filen er blevet valideret korrekt med XML Schemaet, sikres det, at alle aktiveringer af visualisering, referer dynamiske elementer der faktisk er på siden i navigationsdiagrammet.

Herefter sikres det, at der i *.sta* -filen er definition af mindst et linkanker, for hvert linkelement på siden i navigationsdiagrammet.

Nå det er gjort vendes testen og det sikres at alle definitioner af linkanker i *.sta* -filen refererer et linkelement i navigationsdiagrammet. Samtidigt sikres at alle linkanker definitionerne i *.sta* -filen indeholder konstruktioner der er "trykbare". Med "trykbare" konstruktioner menes noget der hos klienten vil blive vist, og som kan benyttes til at følge linket.
Vi accepterer følgende som "trykbare" konstruktioner:

- Tekst der ikke kun indeholder blanktegn.
- Billeder.
- Knapper (der via XML Schemaet er begrænset til at være af typen `submit`, da linket ellers ikke virkede under IE).

C.2 .demand.sta-filer.

Først valideres *.demand.sta* -filen med XML Schema filen *demand.sta.xsd* . Schema filen sikrer at følgende egenskaber er overholdt:

- Alle tags er korrekt afsluttede.
- Strukturen overholdes, det vil sige at der kun er de forventede elementer i *.demand.sta* -filen. F.eks. vil en specifikation af et *statisk* element kun kunne indeholde enten `input`, `select`, `textarea` eller `label` elementer.

- Alle påkrævede attributter er sat, og har korrekte typer.
- Der er kun lovlige attributter. F.eks. må et `input` element ikke have type attributten `button`, da det så vil blive til en knap, og knapper kan ikke returnere værdier.
- Det samme *statiske* element ikke er specificeret flere steder.
- Alle `input`, `select` og `textarea` elementer, indeholdt i en specifikation af et *statisk* element, har `name` attributen sat til at matche det *statiske* elements identifikator. Dette er nødvendigt for at opnå korrekt navngivet CGI parametre.

Når *.demand.sta* -filen er blevet valideret korrekt med XML Schemaet, sikres følgende for hver af de *statiske* elementer, på den tilsvarende side i navigationsdiagrammet:

- *.demand.sta* -filen indeholder en specifikation af det *statiske* element.
- Specifikationen indeholder `input`, `select` eller `textarea` tags. Dette er nødvendigt da kun disse 3 typer tags kan returnere den ønskede CGI parameter.
- Kun `input` tags der definerer radioknapper må forekomme flere gange i en specifikation.
- For specifikationer der indeholder `input` tags, der definerer radioknapper eller skjulte værdier, skal der være angivet en fast værdi for hver af disse.
- Alle dele af en specifikation der angiver CGI værdier som ikke kan ændres hos klienten, har korrekt type. Dette inkluderer faste værdier for radioknapper, de forskellige valgmuligheder under en `select`, skjulte værdier samt de forskellige varianter af indtastningsfelter hvor der kun er givet læserettigheder.

For at muliggøre brugen af scripts til avanceret agrument indsamling, og samtidigt tillade delvis validering af de konfigurerbare filer, udskrives der en advarsel når systemet f.eks. kræver ændring af rettigheder for at kunne producere gyldige værdier. Vi skal her gøre opmærksom på, at hvis der udskrives advarsler kan vores metode ikke længere garantere et fuldt navigerbart system.

Advarsler udskrives for:

- `input`, `select` eller `textarea` tags der er deaktiveret.
- `input` tags der definerer diverse indtastningsfelter hvor der kun er givet læserettigheder, og ikke angivet nogen værdi.
- `select` tags indeholdende deaktiverede valgmuligheder, eller valgmuligheder der ikke indeholder tekst.
- `textarea` tags hvor der kun er givet læserettigheder, og ikke angivet nogen værdi.