NTNU
Det skapende universitet

System for nær sanntid ruteovervåkning

Fredrik Larsen

Master i elektronikk
Oppgaven levert: Juni 2006
Hovedveileder: Bjørn B. Larsen, IET

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Oppgavetekst

For kollektivkunder er ruteforsinkelser et irritasjonsmoment som av og til oppstår. For å lette på dette, ønsker Team Trafikk AS å sjekke mulighetene for å realisere en tjeneste som enkelt lar kunder undersøke ruteforsinkelser. Målet med en slik tjeneste er å gi kollektivkundene forutsigbar og pålitelig informasjon om reelle avgangstider pr. stoppested for lokalbussene i Trondheim.

For å kunne lage tjenester for ruteforsinkelser må først et system for nær sanntid ruteovervåkning utvikles. Et slikt system må kunne posisjonere busser innenfor et avgrenset område og kommunisere dette, med tilhørende tidsstempel, til en sentral database. Her er valg av teknologi et viktig punkt, hvor bl.a. pris, kompleksitet, begrensninger og driftstekniske forhold må vurderes. Ingen spesifikke krav er satt til valg av teknologi.

Det totale ruteovervåkningssystemet er todelt og består av et system for ruteovervåkning og et system for å registrere, samt estimere ruteforsinkelser. Systemet for ruteforsinkelser må kunne beregne nåværende ruteforsinkelse og estimere forsinkelsesforløp framover i tid. Dette for at kollektivkunder skal kunne finne aktuell forsinkelse i forhold til rutetid på en bestemt holdeplass, som vil være en framtidig hendelse. For å få til dette må det utvikles gode algoritmer som så nøyaktig som mulig kan forutsi framtidige hendelser. Algoritmen må også samtidig kunne garantere at kunder som benytter seg av slik informasjon ikke risikere å miste den aktuelle bussavgangen.

For å evaluere teknologi og algoritmer skal det utvikles en simulator. Simulatoren skal benyttes til å simulere en virtuell realisering av systemet. Simulatoren samt modellene som benyttes må være nøyaktige nok til å generere relevante data slik at godheten av algoritme og teknologi kan verifiseres. Ved å modellere systemet digitalt kan man raskt og billig vurdere systemegenskaper og identifisere problemområder. Det blir dermed langt enklere å foreta modifikasjoner enn hva tilfellet ville være for en fullskala pilotmodell.

Etter at systemet for ruteforsinkelser er utviklet skal det utvikles en sluttbrukertjeneste. Tjenesten skal formidle forsinkelsesinformasjon til sluttbruker på en brukervennlig måte. En slik tjeneste skal i første omgang utvikles for Internet og baseres på data fra simulator. Andre formidlingsformer, som for eksempel SMS, kan vurderes på et senere tidspunkt hvis tiden strekker til.

For å redusere antall problemer skal tidligere arbeid innenfor relevante fagområder undersøkes. Viktige erfaringer og resultater fra slike ressurser skal diskuteres mot valg tatt i dette prosjektet for på den måten å fremheve hva som er forskjellig og hvorfor dette er riktig. Dette vil forhindre at tid kastes bort på ting som andre har prøvd og forkastet.

Prosjektet skal gjennomføres i nært samarbeid med Team Trafikk AS.

SAMMENDRAG

Tradisjonelt oppleves kollektivtjenester som et dårligere alternativ til å kjøre egen bil. Miljøargumenter hjelper heller ikke spesielt med å få folk til å kjøre mer kollektivt. For å få folk til å sette igjen bilen hjemme må kollektivtransport være like enkelt å bruke som egen bil, helst enklere. Det finnes en rekke tiltak for å oppnå dette, noen av disse er å redusere pris, øke antall ruter, øke antall avganger, redusere forsinkelser ved hjelp av lyskryssprioritering eller dedikerte kollektivspor og lignende.

En annen strategi for å gjøre tjenester mer kundevennlige er å øke kvaliteten på de tjenester som allerede leveres. Den antatt mest virkningsfulle modellen er å redusere virkningen av ruteforsinkelser. Dette kan gjøres ved å gi bedre informasjon til kollektivkunder om slike forhold slik at kunde slipper å vente i spenning på om bussen kommer snart. Med slik informasjon kan en eventuell forsinkelse utnyttes til noe positivt i stedet, som å kjøpe den avisen du vil ha men som du er usikker på om du rekker å kjøpe.

I dette prosjektet har vi evaluert forskjellig løsninger for å bedre informasjonskvalitet til kollektivkunder. Vi har sett på forskjellige måter å spore kollektivvogner ved hjelp av sanntid og nær sanntids systemer slik at forsinkelse kan beregnes. Dette inkluderer både teknologier for å posisjonere vogner samt kommunisere slik informasjon til sentral

Vi er også godt i gang med utvikling et prototypsystem for beregning av ruteforsinkelser samt for presentasjon av slik informasjon til sluttbruker. Systemet tar unøyaktig posisjonsdata fra vogner hvis tilgjengelig, filtrerer dette og estimerer etter beste evne reell posisjon. Deretter benyttes forskjellige løsninger, bl.a. metoder basert historisk data, for å beregne framtidig forløp slik at sluttbruker kan få et godt estimat for ankomsttid for sin holdeplass.

Som et ledd i å evaluere og teste teknologier og system er det brukt mye tid på å utvikle en simulator, med tilhørende modeller, for å simulere typisk kollektivtrafikk. Simulatoren kan simulere både kontinuerlige og diskrete problemer og er utviklet spesielt for å takle problemer som krever mange ”tilpassninger” i form av utstrakt bruk av programkode i modellbeskrivelse som mange andre ferdige løsninger mangler.

I dette prosjektarbeidet har GSM posisjonering markert seg som en veldig aktuell teknologi for bruk i forsinkelsessystem. Teknologien er rimelig og foreløpige resultater fra simulering viser at tekologi har tilstrekkelig nøyaktighet til å gi gode ankomstsestimater. Manuell posisjonering vha. WAP og GPRS er også en veldig spennende løsning. Her posisjonerer bussjåfør selv bussen vha en mobiltelefon når estimert posisjon fra forsinkelsessystem avviker men en viss margin fra virkelig posisjon.

FORORD

Jeg ønsker å takke veileder Bjørn B. Larsen for god veiledning gjennom hele oppgaveperioden.

INNHOLDSFORTEGNELSE

1. INTRODUKSJON

> Et typisk scenario i Trondheim by: *En gruppe mennesker står på en bussholdeplass og venter. Noen er utålmodige og ser ned på klokken om og om igjen. Noen andre virker direkte urolige. Alle på holdeplassen, unge som gamle, stirrer spent ut i samme retning i påvente av at en buss skal komme kjørende. Spenningen forvandles nå og da til glede etterfulgt av skuffelse når en stor hvit bil som ligner litt, eller en buss med feil rutenummer, kommer kjørende.*

Kollektivsystemer som deler vei med andre trafikanter er av natur upålitelig fordi man ikke kan kontrollere trafikken rundt bussene. Denne upåliteligheten kan oppleves som frustrerende for kollektivkunder noe som scenariet over prøver å illustrere. Den eneste måten å gjøre noe med upåliteligheten til systemet er å lukke deler av infrastrukturen for allmennheten og dedikere dette til kollektivtransport. En slik løsning vil som alle skjønner koste enormt mye penger og vil kreve statlige subsidier for å kunne realiseres da få busselskap vil kunne underbygge en slik investering. Det finnes også andre løsninger som ikke løser problemet helt men kan forbedrer påliteligheten, som for eksempel lyskrysspriorитering av kollektivtrafikk.

En annen og mye billigere variant er å prøve å gjøre noe med frustrasjonen i stedet. Dette kan gjøres ved å gi kollektivkunder informasjon slik at de vet hvor forsinket bussen er. På den måten kan man gjøre noe annet med ventetiden enn å stå spent å stirre ut i intet. Det er dette problemet prosjektet prøver å finne en løsning på.

1.1 PROSJEKTETS BIDRAG

Prosjektet tar utgangspunkt i allerede tilgjengelige teknologier og løsninger men kombinerer et uvalg av disse på en unik og spennende måte slik at forsinkelsesinformasjon kan gjøres tilgjengelig til sluttbruker på en rimeligere måte enn tilsvarende andre prosjekter.

Anbefalt løsning baserer seg på å utnytte billig GSM-teknologi til både å posisjonere vogner samt kommunisere slik informasjon til sentral. Første anbefalte løsning forutsetter at nødvendig GSM infrastruktur er utbygd i tilstrekkelig grad i aktuelt område. Løsningen bruker GSM-nettverket til å posisjonere vognenheter vha. av tjenester fra mobilnettoperatør og mobiltelefoner i busser. Den andre løsningen benytter vanlig WAP via GPRS til å la bussjåfører selv registrere posisjon enkelt og raskt via WAP på mobiltelefon etter hvert som buss passerer holdeplasser.

Ved å utnytte veletablert teknologier som GSM og mobiltelefoner kan man begrenses investeringskostnader for et nær -sanntidssystem for kollektivsystem dramatisk. Med siste løsning, dvs. WAP via GPRS, kan man for eksempel komme i gang med et informasjons og forsinkelsessystem ved å installere utstyr for under 1000 kr pr buss. I tillegg er drifts og administrasjonskostnader for en slik løsning veldig lave.

Det er i denne oppgaven også utviklet et prototypsystem som tar unøyaktig posisjonsinformasjon fra vogner og estimerer ankomsttid til ønsket holdeplass basert på historisk data eller rutetabelldata. Et slikt system er nok ikke noe nytt men i dette prosjektarbeidet kunne jeg ikke finne noe system for dette lett tilgjengelig.

Simulatorløsning utviklet for prosjekt er et generelt rammeverk for å simulere både kontinuerlige og diskrete modeller. Det finnes mange diskrete simulatorer fritt tilgjengelig samt simulatorer spesialisert for en undergruppe kontinuerlige modeller, men en simulator for generelle kontinuerlige problemer var vanskelig å finne og en av grunnene til at et simulatorammeverk er utviklet fra bunn av i dette prosjektet. Det nærmeste jeg kunne finne var Simulink fra MathWorks. Så simulatorammeverket er også et lite bidrag.

Modellene som er utviklet i dette prosjektet kan, i kombinasjon med utviklet simulator, brukes til å evaluere andre spennende posisjoneringsteknologier og/eller algoritmer for å beregne forsinkelse og ankomsttider til bussholdeplasser. Spesielt beregning av forsinkelse og ankomsttider kan løses mange forskjellige måter.

1.2 RAPPORTORGANISERING

I kapittel 2 skal vi diskutere forskjellige posisjoneringsteknologier og fordeler og ulemper med disse mot formål i dette prosjektet

I kapittel 3 skal vi se på simulator som er utviklet i dette prosjektet og hvordan den er bygd samt hvordan den kan brukes.

I kapittel 4 beskrives noen verktøy som benyttes i prosjekt. For verktøy som er laget i dette prosjektet beskrives og implementasjonsdetaljer

I kapittel 5 skal vi på modeller som er laget for å emulere kollektivtrafikk. Her diskuteres virkemåte for viktige modeller samt hvordan de er koblet samme.

I kapittel 6 skal vi se på system for beregning av ruteforsinkelser.

I kapittel 7 presenteres noen simulatorresultater.

I kapittel 8, 9 og 10 blir diskusjon, konklusjon og videre arbeid for prosjekt gjennomgått.

1.3 PROSJEKTMÅL

Gjennom forskningsprosjektet Integrerte Betalings og Innformasjonssystem (IBIS) arbeides det med å utvikle et sanntids informasjonssystem for kollektivtrafikk i Trondheim. Prosjektet gjennomføres i et samarbeid mellom transportetatene i Trondheim og SINTEF. Prosjektet er et stort og involverer i tilegg til et omfattende informasjonssystem også kollektivprioritering i signalanleggene (PAK).

Prosjektet er i skrivende stund lagt på is og det eksiterer ingen klare signaler for om og eventuelt når prosjektet gjenopptas.

På bakgrunn av dette er et gjennomgående mål i dette prosjektet å lage en løsning som ”ligger i andre enden” av prosjektet beskrevet over. Dvs. å lage et ”lett” system som er billig å installere, administrere og vedlikeholde. På den måten kan man velge mellom et fullskala informasjonssystem som IBIS eller et kostnadsoptimalisert system som bare estimere og presenterer ankomsttider og ikke noe annet.

Skal man først installere omfattende og dyrt utstyr så kan man heller gjøre det gjennom IBIS og PAK prosjektene slik at man sikrer seg at utstyr er kompatibelt og kan brukes med disse systemene.

2. TEKNOLOGI

For å beregne nøyaktige ankomsttider til bussholdeplasser må kollektivenheter posisjoners og informasjon kommuniseres til sentral for videre behandling. Det finnes mange potensielle teknologier for å posisjonere kollektivvogner og ikke bare GPS. Alle har både sin fordeler og ulemper i forhold til bruk for i et forsinkelsessystem.

Et utvalg teknologier som beskrives i dette kapittelet plukkes ut som gode kandidater for å realisere et forsinkelsessystem. Disse blir i de neste kapitlene modellert og simulert for å vurdere egenskaper.

2.1 POSISJONERINGSTEKNOLOGIER

For å kunne beregne ruteforsinkelser for kollektivtrafikk må vogner først posisjoneres. For å gjøre dette finnes det en rekke interessante løsninger. I dette kapittelet skal vi se på noen av disse.

2.1.1 GPS

GPS (Global Positioning System) er et satellittbasert navigasjonssystem utviklet av U.S. Department of Defense (DOD). GPS systemet brukes til å bestemme posisjon til GPS mottakere i et globalt perspektiv. GPS -systemet består av 24 satellitter som går i 6 forskjellige baner rundt jorda. Satellitten er organisert slik at det til enhver tid er 6 satellitter innenfor rekkevidde til ethvert punkt på jorda.

GPS -satellittene sender kontinuerlig ut radiosignal med eksakt posisjon og tid. Disse signalene mottas av GPS -mottakere som vha. minimum 4 forskjellige satellittsignaler kan posisjonere seg selv ved å måle tidsforskjellen mellom mottatte signal og hvor de kommer fra.

GPS -systemet opererer vha. av en hovedkontrollstasjon i Colorado USA og tre andre kontrollstasjoner rundt om kring i verden. De tre mindre stasjonene videreformidler GPS -signaler som er innenfor rekkevidde til hovedstasjon i Colorado. Her blir ekstremt nøyaktige satellittbaner utregnet for alle de ulike satellittene. Disse beregningene brukes til å oppdatere posisjon til GPS -satellittene individuelt slik at satellittposisjonsdata alltid er korrekt.

GPS systemet består av to informasjonskanaler (L-bånd) hvorav en kanal er tilgjengelig for sivile formål mens en kanal er reservert for militære formål. For sivile formål var det i utgangspunktet lagt inn en usikkerhet på 100 meter. Våren 2001 ble imidlertid feilkilden i den sivile kanalen tatt vekk. Dette har forbedret nøyaktigheten til GPS -systemet fra omtrent 100 meter til under 20 meter.

GPS har vist seg å være interessant for flere sivile formål. Noen av fordelene er listet opp nedenfor:

- God nøyaktighet på posisjoneringen, under 20 meter ved gode forhold
- Systemet dekker hele jordkloden.
- Det koster ingen ting å bruke systemet.
- En GPS mottaker er relativt rimelig.
- Systemet fungerer i all slags vær, 24 timer i døgnet

Noen ulemper med teknologien:

- Posisjoneringen kan forstyrres av høye bygninger, trær og lignende

2.1.2 DGPS

DGPS, eller Differensiel GPS, er en teknikk som brukes for å øke nøyaktigheten til GPS mottakere fra 20 meter til mellom 1 og 3 meter. Dette gjøres ved å plassere ut en GPS mottaker med en kjent fysisk plassering og bruke denne til å beregne feilkorreksjoner på mottatte signaler ved å sammenligne beregnet posisjon mot virkelig posisjon. Disse korreksjonene kan enten lagres og brukes for å etterkorrigere GPS -data eller man kan sende ut korreksjonsdata i sanntid vha. radioutstyr. For å utnytte et slikt DGPS radiosignal må GPS -mottaker tilpasses og enheten må være innenfor dekningsområde til DGPS enheten som er ca 150 km rundt enhet.

Et DGPS system er i forhold til bruk i et forsinkelsessystem noe overdimensjonert. Et vanlig GPS system har mer en god nok nøyaktighet for formålet slik at den økte nøyaktigheten et DGPS system gir ikke gjør mye nytte. Derfor vil et slikt system som er dyrere enn et vanlig GPS -system være uaktuelt.

2.1.3 POSISJONERING VHA LOKALE FASTPUNKTER.

Et slikt posisjoneringssystem benytter en rekke radiosender kalt ”beacons” langs ruter til kjøretøy som skal posisjoneres. Radiosenderne har kjent posisjon, begrenset rekkevidde og sender kontinuerlig ut et laveffekts radiosignal med et unikt identifikasjonsnummer for sender. Kjøretøy som ønskes posisjoneres utstyres med en tilsvarende radiomottaker som kan dekode identifikasjonssignalet til radiosenderne.

Kjøretøyposisjonering vha. lokal fastpunkter kan gjøres på to forskjellige måter. Den enkleste måten er at systemet passivt får beskjed om posisjon idet kjøretøy passerer en slik ”beacon”. Med et slikt system vil man ikke kunne posisjonere kjøretøy når man vil og nøyaktighet er veldig avhengig av distanse mellom sendere. For å posisjonere kjøretøy aktivt med et slikt system kan man mellom radioposter estimere posisjon vha. å ekstrapolere fra siste kjente posisjon og en gjennomsnittsfart beregnet fra historisk data.

Den andre måten er å utstyr kjøretøy med et odometer i tillegg til radiomottaker. Et odometer er et instrument som kan måle tilbakelagt distanse. I en slik løsning blir kjøretøy aktivt posisjonert ved at system kontakter kjøretøy og henter informasjon om hvilken radiosender som siste ble passert og distanse kjørt fra dette punktet. På den måte kan systemet beregne veldig nøyaktig posisjon gitt at kjørerute er kjent.

Fordeler med et slikt system er som sagt god nøyaktighet for versjon med odometer. Noen ulemper med dette systemet er at det krever høye investeringskostnader samt at det koster en del å vedlikeholde radioutstyret som trengs langs ruter og på kjøretøy. Et annet problem er at radiosendere fysisk må plasseres ut langs rutene som kjøretøy, som skal posisjoneres, kjører. Dette innbærer ofte at utstyr er offentlig tilgjengelig med de utfordringer dette medfører. Et annet men mindre problem er at kjøretøy ikke kan posisjoneres utenfor rute. I et ruteforsinkelsessystem er ikke dette noe problem.

2.1.4 LOKALT RADIONETT

Et alternativ til posisjonering via satellitter er å opprette et lokalt referansesystem som kan brukes til å posisjonere kollektivenheter. Via landbaserte sendere oppretter man et system som vha. samme prinsipp som i GPS -posisjonering beregner posisjon til kollektivenhet basert på tidsforskjeller i mottatte radiosignaler.

En viktig forskjell fra GPS baserte systemer er at mottakeren i en slik løsning returnerer mottatt signal slik at basestasjon som sendte signalet i stedet kan beregne tidsforskjell. Mottakeren i kollektivenhet beregner altså ingen ting selv. For å posisjonere en enhet må altså flere basestasjoner ”snakke” med samme kollektivenhet etter hverandre for å beregne posisjon.

Et lokalt radionett koster veldig mye å installere og gir ingen fordeler over hva for eksempel et GPS system gir. Teknologien nevnes her for helhetens skyld

2.1.5 GSM POSISJONERING

Mobiltelefoner kan vha. terminalbaserte eller nettverksbaserte løsninger også posisjoneres. Dette gjør mobiltelefoner og GSM til en spennende teknologi å bruke da en mobiltelefon også kan benyttes til å kommunisere med sentral. Man slipper med andre ord å installere ekstra kommunikasjons utstyr for at posisjoneringssystem skal virke. Prismessig er også GSM teknologi et spennende alternativ da konkurransen er knallhard i dette markedet noe som gir mye for pengene.

For terminalbaserte løsninger, dvs. mobiltelefonbaserte løsninger, må mobiltelefonen aktivt hjelpe til i posisjoneringsprosessen. I slike løsninger må nødvendig funksjonalitet være tilgjengelig i mobiltelefon for å posisjonere enheten. Eksempler på slike løsninger A-GPS (Network-assisted global positioning system), E-OTD (Enhanced observed time difference) og SIM Toolkit.

Til kontrast kan man i nettverksbaserte løsninger posisjonere terminaler uten hjelp fra mobiltelefonen selv. Man er med slike posisjoneringsløsninger ikke begrenset til å posisjonere telefoner tilpasset formålet men kan posisjonere alle typer mobiltelefoner da det er nettverket og teleoperatør som gjør alt arbeidet. Eksempler på nettverksbaserte løsninger er CGI+TA (Cell Global Identity and Timing Advance) og UL-TOA (Uplink Time of Arrival)

Felles for alle posisjoneringsløsninger for mobiltelefoner er at mobilnettoperatøren må ha nødvendig infrastruktur installert. Utstyr som må installeres varierer etter hvilke posisjoneringsteknologi som skal benyttes. Nøyaktigheten til de forskjellige løsningene, spesielt rene nettverksbaserte løsninger, er også avhengig av type utstyr som tilbys av mobilnettoperatør.

TERMINALBASERTE POSISJONERINGSLØSNINGER

A-GPS

I Assisted-GPS løsningen benyttes en innebygd GPS mottaker i mobiltelefonene til å finne posisjon ved forespørsel og returnere denne informasjon tilbake til nettverket. Posisjon kan enten beregnes av mobiltelefonen selv eller ved at GPS signaldata sendes til en sentral datamaskin for videre beregning.

GSM-nettverkt kan i en A-GPS løsning assistere mobiltelefonene med informasjon som gjør at enheten kan posisjoneres mye raskere og bedre enn hva en tilsvarende ren GPS - mottaker vil klare i litt vanskelige situasjoner, som for eksempel i en by med høye bygninger. Dette assisterende signalet virker tilsvarende som feilkorreksjonssignalet til DGPS -løsningen.

E-OTD

E-OTD (Enhanced observer time difference) løsningen krever bare en software oppdatering av mobiltelefonen for å virke, i tilegg til at infrastruktur må være tilgjengelig på mobiloperatørsiden.

Løsningen virker ved at mobiltelefonen måler tidsforskjeller mellom innkommende kontrollsignal fra minst tre BTS'er (Base Transceiver Station), dvs. GSM basestasjoner, rundt enheten. Denne informasjonen sendes så til en sentral tjener som kalles MLC (Mobile Location Centre). Her blir informasjon sammenlignet med nøyaktig informasjon om posisjon til basestasjonene og nøyaktig tid for sending av kontrollsignal. På den måten kan posisjon til mobiltelefon beregnes.

Nøyaktigheten til et slikt system antas å være mellom 60 og 200 meter alt etter forstyrrelser i signal fra bygninger og lignende. E-OTD er en posisjoneringsteknologi med stort potensial for 3G nettverk. Universitet Cambridge i Storbritannia har demonstrert et E-OTD system med en posisjoneringsnøyaktighet på 5 meter i et UMTS nettverk.

NETTVERKSBASERTE POSISJONERINGSLØSNINGER

CGI + TA

CGI + TA (Cell Global Identity & Timing Advance) er en ren nettverksbasert løsning og kan posisjonere terminaler, dvs. mobiltelefoner, vha. parametrene CGI (The Cell Global Identifier) og TA (Timing Advance).

CGI beskriver hvilken fysisk GSM cellesektor, dvs. BTS, mobiltelefonen er tilkoblet og kan brukes til å posisjonere mobiltelefon innenfor et område ved at man vet posisjon til tilkoblet BTS. Nøyaktigheten er veldig avhengig av cellestørrelsen, dvs. hvor stort geografisk område cellen dekker. Noen tall her er 150 meters nøyaktighet for mikroceller og over 30 km i ubebygde strøk. Nøyaktighet er også avhengig av celletype. Celler kan organiseres i sirkulære eller triangulære sektorer, hvor det i siste tilfellet er flere BTS'er på samme fysiske mast. Siste ordning med flere BTS'er på samme mast er veldig vanlig, spesielt i tettbebygde strøk.

TA er en parameter som kan benyttes til å måle avstand fra BTS. TA er en nettverksbestemt verdi som beskriver hvor lang tid det går fra en mobiltelefon tildeles en tidsluke for kommunikasjon og til data fra mobiltelefonen begynner å komme inn. Siden denne tidsforskjellen er proporsjonal med avstand mellom GSM basestasjon og mobiltelefon, kan man estimere mobiltelefonens avstandsradius fra sender. TA brukes av nettverket til å synkronisere kommunikasjon slik at mobiltelefoner langt fra BTS kan kompensere for dette ved å start sending litt før, i henhold til TA -parameter.

TA -parameteren gir vanligvis en nøyaktighet ved bruk i avstansmåling på 550 meter, men utvikling i nyere tid viser en mye bedre nøyaktighet med en TA -oppløsning på rundt 30 meter.

Ved å kombinere CGI og TA i en CGI + TA løsning kan mobiltelefon i de fleste tilfeller posisjoneres til en triangulær sektor med en bestemt avstand fra BTS. Ved å kombinere dette med for eksempel rutedata kan nøyaktighet økes betraktelig.

UL-TOA
UL-TOA (Uplink Time of Arrival) løsningen virker ved å måle eksakt tid for når et mobiltelefonsignal ankommer tre eller flere forskjellige BTS'er. Systemet ligner på E-OTD bare at tidsforskjellen blir beregnet av nettoperatør i stedet for av mobiltelefon. For å beregne dette må en spesiell innretning med navn Location Measurment Units (LMU) installeres ved BTS'ene.

Forskjellen i tid for når mobilsignalet blir mottatt av de forskjellige BTS'ene kan brukes til å beregne posisjon vha. enkle trianguleringsteknikker.

Nøyaktighet på denne løsningen er ikke kjent. Denne løsningen er den dyreste løsningen for nettoperatører å installere da man må ha en LMU for hver BTS.

AOA
AOA (Angle of Arrival) er en løsning som benyttes til å finne retning til mobiltelefonsignaler i forhold til BTS og baseres også på bruk av LMU på operatørsiden. LMU'ene måler faseforskjell fra signal vha av antenner organisert i et spesielt system. Når to eller flere slike systemer benyttes kan signal peiles inn og posisjon til avsender finnes, dvs. mobiltelefon, ved å sammenligne peilevinkler til de ulike LMU'ene.

2.1.6. MANUELL POSISJONERING VHA. WAP OG GPRS

Manuell posisjonering er en løsning som ofte blir oversett men som for eksempel i et forsinkelsessystem kan gi gode resultater og krever minimal infrastruktur for å virke. Selv om man ikke realiserer det er det i prinsippet dette man gjør for å posisjonere den man snakker med over telefon. I dette tilfellett utføres manuell posisjonering vha. verbal ”handshaking”, dvs. man forhandler frem en felles forståelse av hvor man er. Manuell posisjonering av kjøretøy blir i dag ofte brukt for å posisjonere drosjer og ved unntakstilfeller i kollektivtrafikk som for eksempel ved rapportering av en ulykke. For at dette skal virke må begge parter ha kunnskap om terreng slik at man kan forstå posisjonsforklaringer som gis til hverandre.

Dette prinsippet kan man bygge videre på i et forsinkelsessystem for bussrutesystem ved å systematisere samt forenkle innrapportering av posisjon. Innrapportering av posisjon kan for eksempel gjøres vha. en mobiltelefon med WAP og GPRS. Hvis systemet vet hvilken rute som kjøres og i tillegg vet avgangstider for holdeplasser kan man vha. brukervennlig WAP -grensesnitt, GPRS og et sjåførnummer for å identifisere sjåfør, oppdatere system når kjøretøy er forsinket samt fortelle hvor mye. Dette kan gjøres ved at bussjåfør hele tiden har en oppdatere liste av holdeplasser, med tilhørende avgangstider, for nærmeste framtid via WAP. Når bussjåfør oppdager at tidspunkt formidlet via WAP er forskjellig fra virkelig tidspunkt kan sjåføren enkelt og greit, før avgang fra bussholdeplassen han er på, velge riktig holdeplass fra listen og på den måte varsle systemet om at buss nå forlater denne holdeplassen. På den måten kan systemet beregne forsinkelse og nye avgangstider for de neste holdeplassene i ruten. Man varsler altså systemet når informasjon i system ikke lenger stemmer med virkeligheten. Dette gjøres enkelt ved at sjåfør trykker på en link vha av mobiltelefonen.

Med GPRS blir en slik løsning veldig billig å installere samt drifte og administrere. Brukergrensesnittet baserer seg på WAP og WML. Dette er utbredte teknologier og er tilgjengelig i de aller fleste mobiltelefonmodeller også de billigste. GPRS er pakkebasert kommunikasjonsløsning som støttes av alle moderne mobiltelefoner og all nettoperatører. Med GPRS kan man være online, dvs. på Internet, kontinuerlige uten at man betaler for annet enn antall megabytes overført informasjon. Med en ren tekstbasert WAP -løsning, dvs. uten overføring av bilder og lignende, kan man overføre mye informasjon før man passerer 1 megabyte. Prisen for overføring varierer men for eksempel med Telenor må man betal 10 kroner pr megabyte. I tillegg kommer abonnementsutgifter.

Denne løsningen er veldig økonomisk attraktiv. Man utnytter allerede tilgjengelig ressurser, her sjåfør, utstyrt med mobiltelefon til å posisjonere kjøretøy. Sjåfører kommuniserer direkte med datasystem slik at man slipper ekstra kostnader med å overføre informasjon til og fra system. Ved å bruke en vanlig billig mobiltelefon får man alt nødvendige tekniske utstyr i en innpakning. Man får WAP, display og tastatur for kommunikasjon mellom system og sjåfør og man for GPRS for trådløs kommunikasjon mellom mobil og system. Og man får alt dette til en god pris da slikt utstyr selges i et veldig konkurranseutsatt marked, nemmelig forbrukermarkedet.

3. SIMULATOR

I dette kapittelet skal vi snakke om hva en simulator er, hva den kan brukes til, hva den brukes til i dette prosjektet samt hvordan simulatoren i dette prosjektet er designet og implementert.

3.1 INTRODUKSJON

For å evaluere teknologier er det i dette prosjektet utviklet en simulator samt modeller for å simulerer typisk kollektivtrafikk for et utvalg bussruter i Trondheim. Formålet med dette er å evaluere potensielle posisjoneringsteknologier med tilhørende filter, algoritmer og sluttbrukertjenester og sjekke egenskaper for disse i forskjellige situasjoner.

Alternativer til å simulere er å fysisk montere det posisjoneringsutstyret som skal testes i busser på utvalgte ruter og evaluere resultater av dette. Problemer med dette er bl.a. at det er mer enn en type teknologi som skal evalueres og at en slik prototyprealisering koster penger og tar mye tid å organisere.

Med en simulatorløsning kan man rimelig enkelt teste ut forskjellige teknologier uten de kostnader en fysisk realisering medfører. Man kan i tillegg eksperimentere med systemet, for eksempel analysere hvilke konsekvenser en plutselig og unormal trafikkoppbygging vil ha for systemet under test.

Problemet med en simulatorløsning er at modeller må utvikles, noe som tar tid, og at godhet til simulatorresultater ikke kan garanteres uten verifisering mot tilsvarende virkelig system. Tidskostnadene for utvikling av modeller er veldig avhengig av hva som skal simuleres og krav til nøyaktighet. Man kan, og må, ofte forenkle en god del slik at kostnad begrenses.

Når det gjelder verifiseringsproblemet kan man enten stole på at modellen er god nok til å gi riktig nok resultater eller realisere de beste forslagene fra simulatorløsning og verifisere disse i stedet for å realisere alle alternativer. Man kan med andre bruke simulator til å finne beste løsning.

3.2 HVA ER EN SIMULATOR

En simulator er et verktøy som kan benyttes til å analysere komplekse systemer. Med en simulator kan systemer som skal undersøkes emuleres under full kontroll av observatør. Systemet emuleres i programvare ved hjelp av en datamaskin.

Med en simulator kan man enkelte analysere og eksperimentere med systemer. En simulator kan benyttes til å analysere systemer som ikke er laget enda, eller analysere systemer med vanskelig tilgjengelighet for eksempel pga. av størrelse eller håndterbarhet, eller systemer som ikke kan forstyrres slik at nødvendige eksperimenter kan gjennomføres o.l.

Med en simulator kan viktige systemkomponenter modelleres og slippes ”fri” i en virtuell verden. Ved observasjon, dvs. innsamling av simulatordata, kan så systemegenskaper observeres og analyseres. En fin egenskap med simulator er at man enkelt kan gjøre ”hva skjer hvis” analyser uten å ty til vanskelige analytiske metoder. Modifiser modeller, start simulator og observer hva som skjer.

En stor fordel med mange simulatorer er at man kan dele opp et stort problem i mindre deler. For eksempel kan en elektrisk krets være veldig kompleks og vanskelig å analysere med tradisjonelle (analytisk) fremgangsmetoder. Med en egnet simulator kan kretsen deles inn i atskilte komponenter som hver for seg er analyserbare. Disse komponentene kan da modelleres og settes sammen vha. regler og en simulator (solver) kan da løse problemet numerisk. Dette deler effektivt problemet opp i små og håndterbare deler.

En simulator vil også i prototypsammenheng enklere belyse potensielle problemer eller effekter som oppstår ved sammensetting av mange mindre deler til et komplekst system. Slike effekter kan være vanskelig å forutsi vha av tradisjonelle metoder. Eksempler her kan være kryssnakk i små elektriske kretser. Ved å simulere kretsen kan man enkelte observere slike effekter for eksempel i form av støy.

Et problem ved simulering er feil. Feil kan oppstå på to forskjellige områder, nemmelig i simulatorkode og i modellbeskrivelser. Feil i simulatorkode er mindre vanlig da kode ofte er godt gjennomtestet. Feil i modellbeskrivelse er derimot mer vanlig, eller retter sagt, nesten uunngåelig. Man kan sjeldent modellere et system, foruten rene teoretiske systemer, nøyaktig nok til at feil ikke oppstår. Dette fordi virkelige systemer har uendelig mange egenskaper og forandrer tilstander kontinuerlig, dvs. at systemet man ønsker å simulere er "analogt". Simulatoren gjør sine beregninger på et digitalt system slik at det analoge må diskretiseres. I denne "konverteringen" vil det oppstå feil. Trikset er å gjøre feilene små nok til at resultatet ikke forstyrres nevneverdig. Måter å gjøre dette på er å modellere flere egenskaper samt øke nøyaktigheten på beregningene. Dette koster selvfølgelig i form av mer tid på utvikling av modeller og simulering av disse.

Resultater av en simulasjon er ikke bedre enn nøyaktighet til de modeller som benyttes.

3.3. SIMULATORTYPER

En simulator klassifiseres ut i fra hvilke type modeller, eller problemer, simulatoren kan håndtere. De mest vanlige modelltyper er stokastiske eller deterministiske, diskrete eller kontinuerlige modeller.

3.3.1 STOKASTISK MODELLER

Stokastiske modeller, typisk også diskrete modeller, benyttes til å simulere statistisk tilfeldighetsfordeling. Slike modeller har ingen gitt oppførsel ved gitte betingelser. Stokastiske modeller vil ved ekte tilfeldighet ikke gi samme verdier ved påfølgende simulasjoner. Slike modeller kan for eksempel benyttes å modellere stråling fra radioaktivt materiale, tilførsel av ny kunder i en butikk kø og lignende.

3.3.2. DETERMINISTISKE MODELLER

Deterministiske modeller modellerer, i motsetning til stokastiske modeller, prosesser hvor utgangsverdier og tilstander er en funksjon av inngangsstimuli. Slike modeller vil ved flere simulasjonsgjennomkjøringer gi samme verdier gitt at stimuli er identiske. Typiske modeller her er for eksempel translasjonsmodeller for fysiske objekter, temperatursensitivt relé. Deterministiske modeller kan være både diskrete og kontinuerlige.

3.3.3 DISKRETE MODELLER

Diskrete modeller er en type modeller som bare skifter tilstand ved diskrete hendelser. Eksempler på slike modeller er typisk digitalelektronikk -systemer, kø -systemer, nettverks -simulator og lignende.

3.3.4 KONTINUERLIGE MODDELLER

Kontinuerlige modeller er en klasse modeller som oppdaterer interne tilstander kontinuerlig. Eksempler på slike modeller er naturlige ting som stråling av radioaktivt materiale. Slike modeller representeres ofte som et sett med differensialligninger.

Et problem med kontinuerlige modeller er selvfølgelig at en datamaskin, og derfor simulator, ikke kan behandle kontinuerlige data. Kontinuerlige modeller kan ”simuleres” ved på en diskret simulator ved å gjøre de diskrete tidsstegene små nok.

3.4 SIMULATOR UTVIKLET FOR PROSJEKT

Som nevnt tidligere er det i dette prosjektet utviklet en simulator samt modeller som kan simulere kollektivtrafikk og tilhørende posisjoneringsteknologier. For på best mulig måte å simulere dette er det utviklet en simulator som kan løse problemer innen alle de fire forskjellige modellkategoriene, kontinuerlige eller diskrete modeller, deterministiske eller stokastiske modeller.

Kontinuerlig modellstøtte, dvs. modeller som kan oppdatere tilstander kontinuerlig, er nødvendig for bl.a. å nøyaktig kunne simulere tidskontinuerlige effekter som forflytning, akselerasjon og lignende. Diskret modellstøtte forenkler implementering av logikk som opptrer maks en gang pr tidssteg gitt en hendelse som for eksempel ”skift til neste rute når buss er på endeholdeplass”. Støtte for deterministiske og stokastiske modeller er nødvendig for å simulere tilfeldige tallfordelinger hvor tilfeldigheten kan kontrolleres. Slike fordelinger benyttes bl.a. for å simulere aggressivitet hos forskjellige bussjåfører.

Støtte for alle modellkategorier finnes i såkalte kryssdomenesimulatorer. Mange simulatorer for kontinuerlige modeller er også såkalte kryssdomenesimulatorer fordi diskrete modeller ofte er et spesialtilfelle av en kontinuerlig modell. Det motsatte er ikke tilfellet.

Det finnes en god del kommersielle programpakker for simulering av kontinuerlige modeller. Problemet med mange av disse er at de er spesialiserte mot en bestemt underkategori med problemer innenfor hovedkategorien kontinuerlige modeller. For eksempel er SPICE en kjent pakke for simulering at kontinuerlige modeller av typen ”analoge elektriske komponenter og kretser”. Andre pakker finnes for simulering av mekanikk, vesker, gasser, kjemi og lignende.

MATLAB Simulink er en kjent kryssdomenepakke for simulering av bl.a. generelle kontinuerlige systemer. Her designes modeller ved å dra komponenter til arbeidsflaten og koble disse sammen som byggeklosser. Det finnes mange ferdige byggeklosser som dekker mange behov, fra de helt grunnleggende komponentene som addere, multiplikatore, integratore o.l. til større ferdige systemer i form av en komponent som for eksempel filter o.l. Man kan nemmelig lage nye komponenter ved å samle komponenter hierarkiisk. På den måten kan komplekse systemer lages relativt brukervennlig.

Problemet med kryssdomenesimulatorer som Simulink er dårlig støtte for bruk av skript i beskrivelsen av diskrete modeller. Verktøyene begrenser seg ofte til tradisjonelle diskrete modeller for eksempel filterkomponenter for digital signalbehandling. Dette passer dårlige med behov for simulering av kollektivtrafikk. I slik simulering må informasjon og oppførsel som for eksempel rutelogikk lett kunne implementeres som diskrete modeller som for eksempel kan brukes til å påvirke kontinuerlige modeller.

Man trenger med andre ord en simulator som lett kan integrere et programutviklingsmiljø slik at vanskelige diskrete modeller lett kan implementeres vha kraftige og generelle kodeverktøy.

Pga av dette er det i dette prosjektet utviklet en egen kryssdomenesimulator tilpasset simulering av kollektivtrafikk.

Det er som sagt tidligere utviklet en simulator for dette prosjektet. Simulatoren som er utviklet er inspirert av Simulink -systemet og er bygd opp rundt konsepter som komponenter og kommunikasjonskanaler. Komponenter definerer simulatormodeller og kanaler definerer kommunikasjonsrør som benyttes til å binde komponentene sammen.

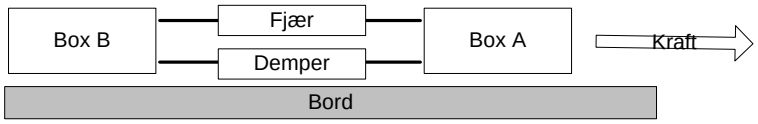
I dette kapittelet vil jeg beskrive design og arkitektur av simulatorløsning ved hjelp av eksempler på hvordan løsning kan brukes. Dette er ikke fordi simulatoren veldig avansert eller vanskelig å forstå men fordi simulatorløsningen har mange arkitekturelementer som fort kan bli for mye å holde orden på. Bruk av konkrete eksempler gjør det enklere å danne et bilde av de ulike delene og viktigheten av disse.

3.5 ARKITEKTUR

Simulatoren er arkitekturmessig designet rundt tre hovedklasser med navn *Sim*, *Entity* og *Channel*. Klassen *Sim* er hovedklassen som styrer simuleringen. Klassen *Entity* er en baseklasse som beskriver grensesnitt mellom modell og simulator. Klassen *Channel* beskriver grensesnitt for hvordan kontinuerlige komponenter kommuniserer seg imellom.

For å simulere modeller i simulatoren må klassen *Entity* benyttes. Denne klassen beskriver grensesnittmetoder for hvordan modeller kan integreres og kjøres i simulator. Metodene fylles, ved klassearv, med modellspesifikk oppførsel og simulatoren vil så ta seg av kjøring automatisk. Klassen *Entity* benyttes både for diskrete og kontinuerlige modeller.

Klassen *Channel*, eller kanal, benyttes i simulator for å modellere signalflyt mellom kontinuerlige modeller. Klassen kan skrives til og leses fra og har i tillegg funksjonalitet for å knytte sammen kanaler. Kanaler kan ses på som io-porter som komponenter, her entiteter, benytter til å kommunisere med omverdenen.



Figur 3.1 – Boks-fjær-demper system

For å forenkle videre systembeskrivelser kan det være greit med et konkret eksempel å forholde seg til. Som eksempel har jeg valgt å bruke et kloss-fjær-demper-system. To klosser er koblet sammen vha. ei fjær og en demper på et friksjonsløst bord. Vi skal vha. simulering finne ut hva som skjer hvis vi tilfører den ene klossen en konstant kraft. Dette er også et fint eksempel på hva simulatorer er flinke til, nemmelig til å finne ”gode nok” svar på analytisk vanskelig problemer.

For å forenkle videre benevnes klasse med stor bokstav, for eksempel *Kloss*, og instanser med liten, for eksempel *kloss*. Flere instanser benevnes med tall etter instansnavn, for eksempel *kloss1*, *kloss2* osv.

For å simulere noe må man først lage simulatormodeller som simulatoren forstår. Dette gjøres ved å lage klasser som arver egenskaper fra baseklassen *Entity*. Slike klasser kalles modellklasser og kan modellerer funksjonalitet ved å fylle inn tomme grensesnittmetoder fra baseklassen *Entity* med modellspesifikk kode. For enkelhets skyld kan vi kalle en slik modellklasse *Kloss*. I den nye klassen *Kloss*, som arver fra *Entity* og inneholder modellspesifikk kode, instansierer man så nødvendige io-kanaler og registrerer disse med klassen *Kloss* slik at klassen kan kommunisere med omverdenen. Når dette er utført registreres så en objektinstansen av klassen *Kloss* med simulator.

Det er viktig å huske at alle modellklasser, som *Kloss* i dette eksemplet, arver fra grunntypen *Entity*. På den måten kan simulatoren behandle alle de ulike modellene likt, vha grensesnittmetodene i klassen *Entity*. Så når ordet entiteter nevnes, mens ikke mange instanser av grunnklassen *Entity* men mange instanser av klasser som arver fra *Entity*, dvs. mange modellklasser.

Når modeller er instansiert som entitetsinstanser og registrert med simulatorinstans og io-kanaler registrert med tilhørende entitetsinstanser, må io-kanaler kobles sammen. Dette gjøres enkelt ved å adressere kanaler i de ulike entitetene som skal kommunisere og eksekvere kanalmetoden *connect()*. Som i eksemplet over må for eksempel fjær kommuniser kraft til kloss slik at kloss kan flytte på seg. Vi kan da si at ”*out_force*” er en registrert kanal i modellklassen *Fjær* og at ”*in_force*” er en registrert kanal for modellklassen *Kloss.* Vi kobler da enkelt signalportene sammen slik:

”*kloss.in_force.connect(fjær.out_force)*”

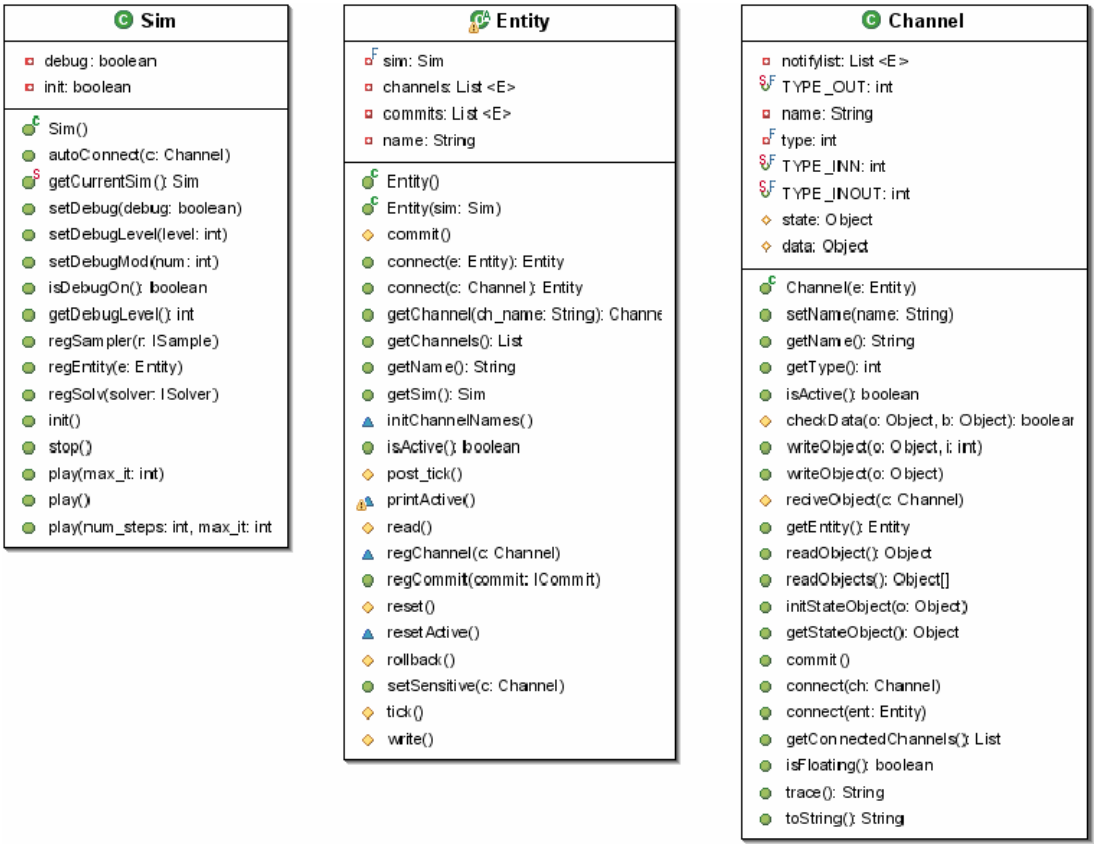
I en slik sammenkobling vil instansen *kloss*, ved simulering, bli varslet ved forandringer i noen av de tilkoblede kanalene. På den måten kan tilstand oppdateres og modell holdes synkronisert i forhold til inngangssignaler. Når fjær i dette eksemplet setter et signal på ”*out_force*” vil kloss varsles og oppdateres fordi signalene er koblet sammen.

Når alle modellklasser er instansiert, koblet sammen og registrert, kjøres simulatoren ved å kalle metoden ”*play()*” i simulatorinstansen *sim*. Denne metoden vil da gå gjennom en liste av registrerte entiteter og kjøre gjennom modellogikk som er spesifisert i den arvede modellklassen. Etter et gitt antall iterasjoner, eller til et kriterium er oppfylt, avsluttes simulasjonen.

For at simulering skal være nyttig må informasjon kunne hentes ut. Dette gjøres ved å bruke en rekke forskjellige ”prober” som kan sample for eksempel attributtverdier og funksjonsverdier til modellklasser og signalverdier som går gjennom kanaler i simulator. Verdiene som samples kan så eksporteres til for eksempel Excel for videre behandling eller tegenes direkte i diagramvinduer for direkte analyse. Dette kan gjøres mens simulator kjører for lange simuleringer, eller prober kan settes opp til å vise resultater etter simulering.

3.6 VIRKEMÅTE

Under er et klassediagram for klassene *Sim*, *Entity* og *Channel*. Diagrammet viser de viktigste metodene og attributtene. Fargene og symbolene foran metodene og attributtene angir synlighet til metoden, eller attributtet, for andre klasser. Grønn sirkel betyr at metoden eller attributtet er tilgjengelig fra andre klasser. Andre farger betyr begrenset tilgang.



Figur 3.2 – UML diagram for klassene Sim, Entity og Channel

For å lage simulerbare modeller må man som sagt tidligere implementere modellogikk ved å lage en ny klasse som arver fra klassen *Entity* og i den nye klassen overstyre et utvalg metoder fra *Entity*. Metodene som kan overstyres for å implementere modellogikk er *read()*, *write()* og *tick()*. Dissen metoden kalles fra simulator når modellkode skal kjøres. Metodene *read()* og *write()* overstyres hvis man ønsker å lage diskrete modeller og *tick()* overstyres for kontinuerlige modeller.

3.6.1 DISKRETE MODELLER

For diskrete modeller, dvs. modeller som ikke skifter tilstander kontinuerlige men ved diskrete hendelser, må metodene *read*() og *write*() benyttes. Diskrete modeller blir i simulatoren simulert ved hjelp av en tofase prosess for kommunikasjon og oppdatering av tilstander. I første fase kalles metoden *read*(). I denne fasen kan modeller bare lese og ikke utføre noen operasjoner som kan oppdatere tilstand. Modeller kan lese tilstand fra andre modeller for på den måten å gjøre seg klar til neste fase som er en skrive og oppdateringsfase. I skrive- og oppdateringsfasen kalles metoden *write*() og modeller kan fritt oppdatere interne tilstander basert på informasjon lest i forrige fase. Ingen modeller har lov til å lese informasjon i denne fasen.

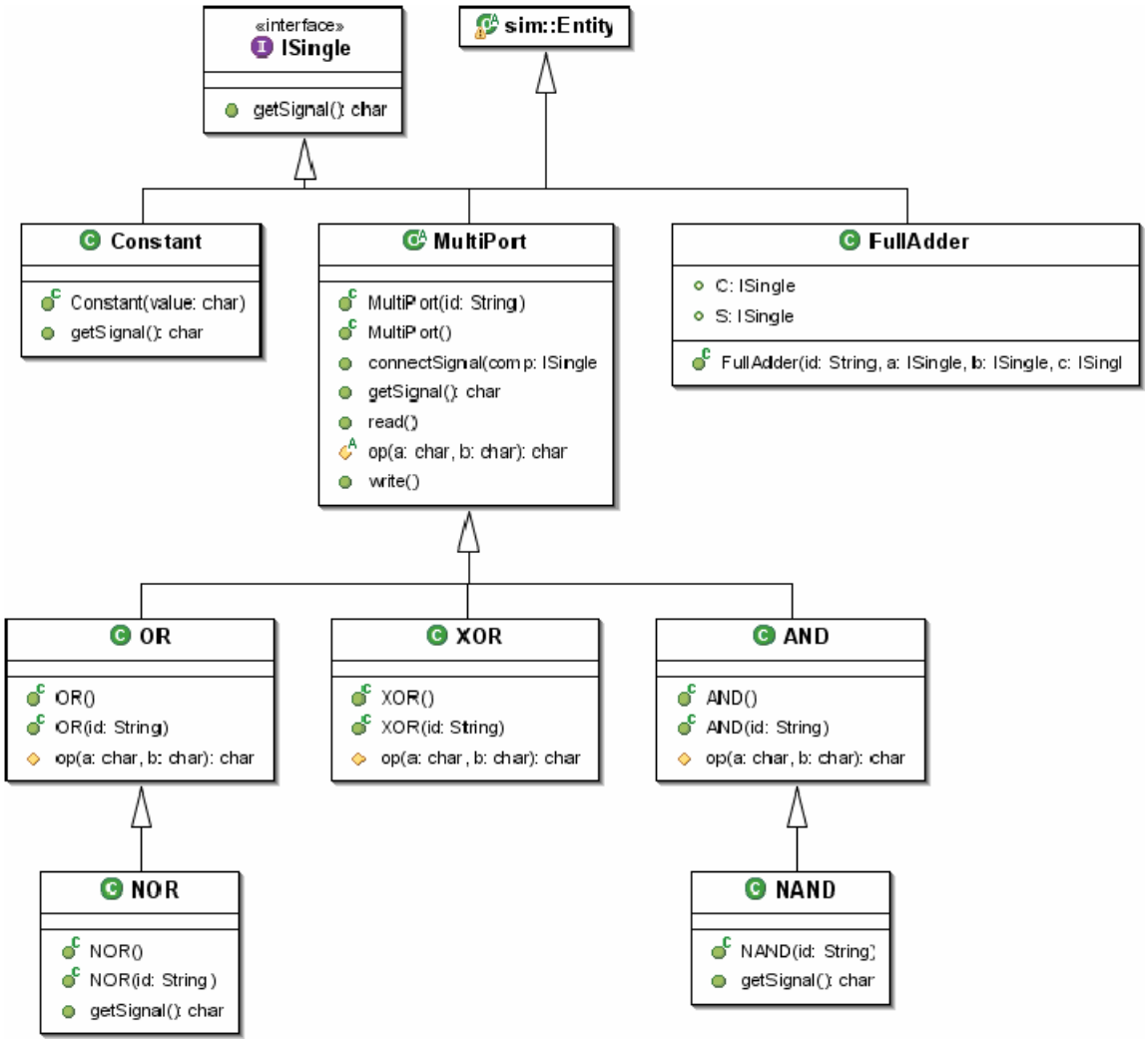
Diskrete modeller benytter ikke klassen *Channel* ved lesing av tilstandsdata fra andre modeller. Modellobjektene kan hente informasjon direkte fra andre modellobjekter pga. tofaseprosessen. Tofaseprosessen synkroniserer lese- og skriveprosessene slik at alle modeller leser de samme tilstandsdata i lesefasen. I neste fase, skrive og oppdateringsfasen, har ikke modellene leserett lenger og kan derfor oppdatere tilstand uten fare for at andre modeller leser før og etter en tilstandsoppdatering og på den blir usynkrone.

Simulatoren forholder seg til en liste av modeller og kaller modellmetoder i sekvensielle orden. En tofaseordning er viktig for å synkronisere informasjon mellom alle registrerte diskrete modeller i simulatoren. Hvis modeller fikk lov til å lese og oppdatere samtidig ville for eksempel modeller øverst på listen lese forskjellig data enn modellene nederst på listen. Med andre ord unngår vi at registreringsrekkefølge for modellene har noe å si ved å benytte en tofaseprosess.

Kommunikasjon mellom diskrete modellobjekter gjøres direkte, uten bruk av kanaler, noe som forenkler utvikling av modeller betraktelig. I stedet for å sette opp et simuleringssystem av komponenter, porter og linker kan man benytte standard objektorienterte prinsipper. Kommunikasjon kan for eksempel gjøres gjennom vanlige metodekall og et grensesnitt kan derfor benyttes til å definere hvordan en samling modeller kan snakke sammen. Men det er viktig å overholde avtalen om å ikke oppdatere tilstandsvariabler i leseprosessen.

Eksempel
Under er lite eksempel på en diskret type simulering. Eksemplet viser raskt hvordan modeller kan lages og hvordan simuleringen oppdaterer tilstandsvariabler. Eksemplet modellerer en Ripple Carry Adder, en enkel digital addisjonskrets. Kretsen benytter logiske porter av typen AND, OR og XOR.

Kretsen kan fordelaktig simuleres vha. diskrete modeller fordi logiske porter i virkeligheten har en treghet i seg, dvs. at portene bruker litt tid på å forandre tilstand. Dette faller naturlig sammen med diskrete modeller som bare kan oppdaterer tilstand ETTER at de modeller den er avhengig av har oppdater til ny tilstand. Kontinuerlig modeller derimot kan skifte tilstand samtidig med at modellene de er avhengig av skifter til ny tilstand.



Figur 3.3 – UML diagram Ripple Carry Adder

Over er et klassediagram av eksemplet vist. Som diagrammet viser kan man ved diskrete modeller benytte seg av vanlige objektorientert prinsipper som arv, bruk av grensesnitt osv. fordi objektene ikke trenger egne kommunikasjonskanaler men kan kommunisere vha vanlige metodekall.

De logiske portene OR, XOR, AND, NOR og NAND er modellert ved hjelp av arv fra klassen *MultiPort*. *MultiPort* er designet som en generell hjelpeklasse for komponenter som kan ha 1 eller flere innganger og kun 1 utgang, noe som passer for de logiske portene som skal modelleres.

Multiport leser utganger fra andre komponenter ved hjelp av grensesnittet *ISingle*. *ISingle* definerer en grensesnittmetode med navn *getSignal*() som brukes til å lese signaler fra forskjellige komponenter. Metoden *read*() i klassen *Multiport* vil i lesefasen av kalles av simulator. I denne metoden finnes logikk som går igjennom alle registrerte *ISingle*-instanser, dvs. komponenter som er tilkoblet, og kaller metoden *op(char a, char b)* med verdier fra *getSignal*() fra *ISingle*-instansene.

Metoden *op(char a, char b)* er abstrakt i *MultiPort* -klassen og må derfor defineres av klasser som arver fra *MultiPort*. Det er i denne metoden logikk defineres. Resultat fra metodekall til *op(char a, char b)* lagres i en midlertidig variable, dvs. en variabel som ikke kan leses utenfra klassen. Dette for å unngå å forandre på tilstand i lesefasen. I skrive og oppdateringsfasen kalles metoden *write*(). I denne metoden oppdateres objekt til ny tilstand ved å kopiere verdi fra midlertidig variabel til tilstandsvariabel for slik å skifte utgangsverdi til kalkulert verdi.

Kodeeksemplet under viser hvordan MultiPort er implementert. Merk bruk av midlertidig variabel i metoden *read*() og hvordan verdi til den midlertidige variabelen kalkuleres.

```
public abstract class MultiPort extends Entity implements ISingle
{
    private List comps = new ArrayList();

    private char sig = 'Z';
    private char next_sig = 'Z';

    public void connectSignal(ISingle comp)
    {
        comps.add(comp);
    }

    public char getSignal()
    {
        return sig;
    }

    protected abstract char op(char a, char b);

    public void read()
    {
        char prev = ((ISingle) comps.get(0)).getSignal();
        for(int i = 1; i < comps.size(); i++)
        {
            ISingle comp = (ISingle) comps.get(i);
            char s = comp.getSignal();
            prev = op(s,prev);
        }

        if(comps.size() == 1)
            prev = op(prev,prev);

        next_sig = prev;
    }

    public void write()
    {
        sig = next_sig;
    }
}
```

Figur 3.4 - Kildekode for klassen Multiport

Kodeeksemplet under viser en enkelt AND-modell. Klassen arver fra *MultiPort* og definerer oppførsel ved å implementere den abstrakte metoden *op* fra baseklassen *MultiPort*. Som vi ser, kan vi med arv slippe å implementere logikk som er lik for mange porter. Her ser vi at metodene *read*() og *write*() som alle diskrete modeller må definere blir tatt hånd om i baseklassen *MultiPort*.

```
public class AND extends MultiPort
{

    protected char op(char a, char b)
    {
        if(a == '0' || b == '0')
            return '0';
        else if(a == 'Z' || b == 'Z')
            return 'Z';
        return '1';
    }
}
```

Figur 3.5 - Kildekode for klassen AND

Etter at grunnleggende logiske porter er modellert kan en fulladder implementeres ved å sette samme disse portene som vist i kodeeksemplet under. Utgangsverdi til *FullAdder* finner vi i attributtene S, for sum, og C, for carry. Som vi ser er S koblet til nS og C koblet til nC som definert i konstruktør for klasse.

```
public class FullAdder extends Entity
{
    public ISingle S;
    public ISingle C;

    public FullAdder(String id, ISingle a, ISingle b, ISingle c)
    {
        XOR p = new XOR();
        p.connectSignal(a);
        p.connectSignal(b);

        XOR nS = new XOR(id + " sum");
        nS.connectSignal(p);
        nS.connectSignal(c);

        AND g = new AND();
        g.connectSignal(a);
        g.connectSignal(b);

        AND tmp = new AND();
        tmp.connectSignal(p);
        tmp.connectSignal(c);

        OR nC = new OR(id + " carry");
        nC.connectSignal(g);
        nC.connectSignal(tmp);

        S = nS;
        C = nC;
    }
}
```

Figur 3.6 - Kildekode for klassen FullAdder

For å kjøre simulator kan følgene kodeeksempel benyttes.

```
public class Test
{
    public static void main(String[] args) throws Exception
    {
        Sim sim = new Sim();

        Constant a1 = new Constant('1');
        Constant a2 = new Constant('1');

        Constant b1 = new Constant('1');
        Constant b2 = new Constant('0');

        FullAdder s1 = new FullAdder("a1",a1,b1,new Constant('0'));
        FullAdder s2 = new FullAdder("a2",a2,b2,s1.C);

        sim.play(5, 2);
    }
}
```

Figur 3.7 - Initialiseringskode

Kodesnutten *sim.play(5, 2)* i eksemplet over starter simulator og kjører 5 ”hendelser” fremover, dvs. at modeller oppdateres 5 ganger. Argumentet 2 har ingen påvirkning for diskrete modeller og kan her ses bort fra.

Resultater fra simulering er her vist som en utskriftslogg. Som vi ser starter komponentene i ukjent tilstand, dvs. Z. Etter hvert som signaler utbrer seg skifter utgangene til fulladderne.

```
a1 sum sig: Z
a1 carry sig: Z
a2 sum sig: Z
a2 carry sig: Z
a1 sum sig: 0
a1 carry sig: Z
a2 sum sig: Z
a2 carry sig: Z
a1 sum sig: 0
a1 carry sig: 1
a2 sum sig: 0
a2 carry sig: Z
a1 sum sig: 0
a1 carry sig: 1
a2 sum sig: 0
a2 carry sig: Z
a1 sum sig: 0
a1 carry sig: 1
a2 sum sig: 0
a2 carry sig: 1
```

Figur 3.8 – Utskriftslogg

3.6.2 KONTINUERLIG MODELL

Kontinuerlige modeller må som diskrete modeller også implementeres ved å lage en klasse som arver fra klassen *Entity*. I den nye klassen implementeres modellogikk ved å omdefinere den tomme metoden *tick*() fra baseklassen *Entity*. Dette til forskjell fra diskrete modeller som implementerer modellogikk ved å omdefinere metodene *read*() og *write*() fra samme baseklassen *Entity*. Kontinuerlige modeller trenger bare å omdefinere en metode da slike modeller ikke trenger en tofaseprosess som er tilfellet for diskrete modeller.

Kontinuerlige modeller er egentlig ikke kontinuerlige i ordets rette forstand, da dette er umulig å implementere korrekt på digital maskinvare. Simulatoren kan ved hjelp av små nok tidssteg finne løsninger på kontinuerlige problemer som er god nok, dvs. at diskretiseringsfeil er så små at resultatet ikke blir nevneverdig forstyrret av dette.

Kontinuerlige modeller må bruke kanaler for å kommunisere og kan ikke som diskrete modeller lese direkte fra andre modellinstanser. Dette fordi man i kontinuerlige simuleringer ikke har noen stabile og atskilt faser hvor modeller er synkrone. Modeller påvirker hverandre hele tiden slik at kanaler må benyttes for å holde orden på hvilke modeller som har forandret tilstand og hvilke modeller som er avhengig av forandringer i disse modellene slik at disse kan varsles.

Kanaler opprettes ved å instansiere klassen *Channel* og koble den til en *Entity*-instans, dvs. til et modellobjekt. Kanalene kan så kobles sammen med andre kanaler tilhørende andre modellinstanser for å danne en kommunikasjonslink mellom modeller.

Ved simulering vil simulatorinstansen *sim* gå gjennom alle registrerte entiteter, dvs. alle modellinstanser både diskrete og kontinuerlige, og først kalle metodene *read*() og *write*() og så til slutt *tick*(). I *tick*() fasen vil man i kontinuerlige modeller lese data fra innkanaler og sette nye verdier på utkanaler. Hva som er inn og utkanaler bestemmes bare av hvordan kanalene brukes, dvs. om du leser eller skriver til en kanal. Det er ingen tekniske forskjell, begge ”typene” er av samme klasse *Channel* og kan være både inn og utkanal.

Etter at simulatorinstansen har kjørt igjennom *tick*() metoden til alle registrert entitetsinstanser en gang, starter en ny iterasjon. I den nye iterasjonen sjekker simulatoren først om entiteten er aktiv. En entitet får status aktiv hvis en annen entitet fra forrige iterasjon skriver til en kanal som er koblet til den gjeldende entiteten, dvs. at det ligger data i en eller flere kanaler som venter på å bli prosessert. Hvis entiteten er aktiv kalles metoden *tick*() slik at data prosesseres. Etter at simulatoren har gjort dette for alle aktive entiteter starter en ny iterasjon. Dette fortsetter til det ikke er flere aktive entiteter igjen, dvs. at det ikke finnes noen ny data som skal prosesseres, eller at simulator når maks antall spesifiserte iterasjoner.

En slik ordning sikrer at alle entiteter får muligheten til å behandle data uansett i hvilken rekkefølge de er registrert i. Entiteter tidlig på simulatorlisten som er avhengig av entiteter lengre ned på listen vil få informasjon fra disse, men en iterasjon etter.

Som forklart tidligere er ikke kontinuerlige modeller egentlig kontinuerlige men emulerer dette ved hjelp av små nok tidssteg. Små nok er relativt og må ses i sammen hva slags problem som skal simuleres, dvs. hvor mye feil som tolereres. Mindre tidssteg gir mindre feil, men tar lengre tid å simulere.

Når vi snakker om tidssteg her er dette bare for å gjøre forklaringer enklere. Simulatoren kan simulere kontinuerlige modeller uten noe form for tid da tid er en modell som alle andre modeller. Det som er kontinuerlig er kommunikasjonen mellom modellene, hvor en type kommunikasjon kan være for eksempel tidsverdi. Tidssteg er egentlig et tilstandssteg, dvs. at simulatoren har beregnet ferdig modelltilstand ved hjelp av iterasjonene som forklart tidligere og forsetter med beregninger av ny tilstand.

Kanaler benyttes, i tillegg til å overføre informasjon, også til å holde ordne på tilstandsinformasjon til modeller. Modeller er tilstandsavhengig hvis man i beregning av utsignaler bygger videre på tidligere beregnede verdier, dvs. tilstand til modell. Et eksempel på en slik modell er en integrator som for hvert tidssteg bygger videre på tidligere beregninger. Eksempel på modeller uten tilstand er modeller som beregninger utgangsverdier på bakgrunn av inngangsverdier alene, som for eksempel en addisjons-modell.

Tilstandsinformasjon hentes fra kanaler ved metoden *getState*(). Ved kall til denne metoden vil kanalen returnere beregnet verdi fra forrige serie iterasjoner. Denne verdien er den samme helt til simulator har beregnet ferdig ny tilstand, dvs. at iterasjonene er ferdig, hvor da den nye verdien tar plassen til den gamle tilstandsverdien.

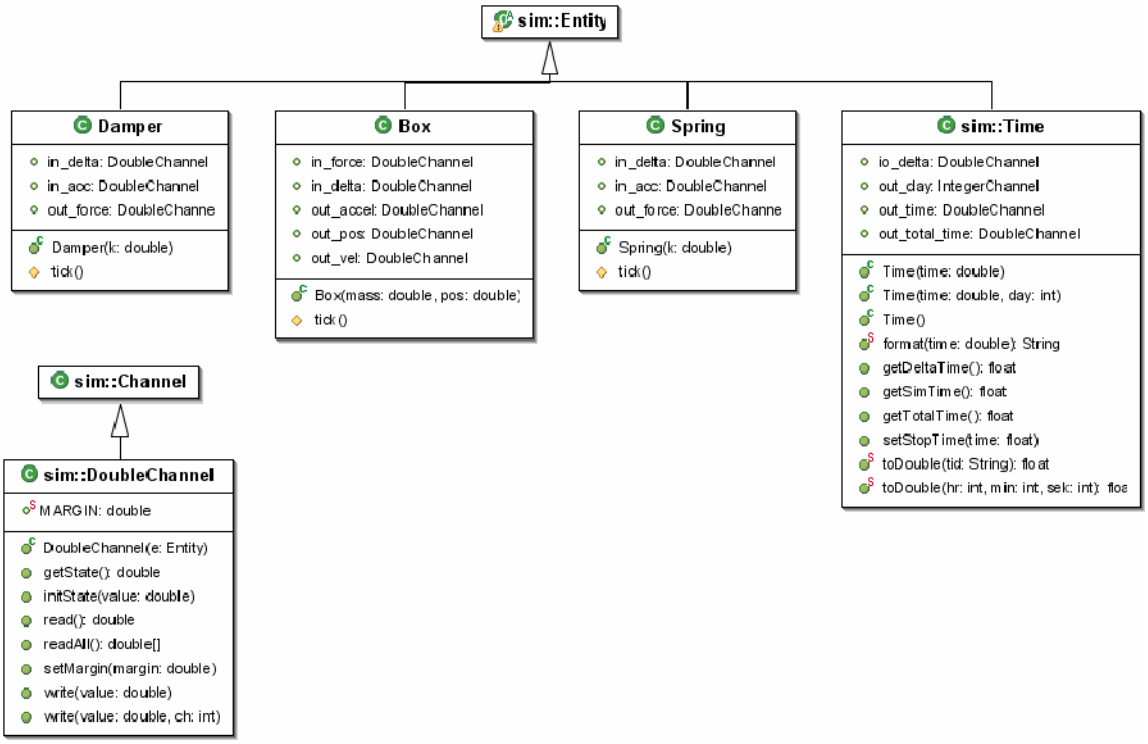
Vi kan dele modelldesign inn i to hovedstrukturer, tilbakekoblede og ikke tilbakekoblede strukturer. I ikke tilbakekoblede strukturer går signalvei fra entitet til entitet og ender til slutt i en utkanal som ikke er tilkoblet noen annen entitet. Tilbakekoblede strukturer danner signalringer, dvs. to eller flere entiter som er koblet til hverandre direkte eller indirekte slik at en eller flere signalveier danner en ring.

I ikke tilbakekoblede strukturer vil man med et begrenset antall iterasjon kunne prosessere ferdig listen med entiteter. I tilbakekoblede strukturer kan ikke det samme garanteres. Her kan signalet ende opp med å gå i ring uendelig lenge, gitt uendelig mange iterasjoner, hvis ikke noe aktivt settes inn for å unngå dette.

Det finnes to forskjellige måter å unngå at tilbakekoblede strukturer ”låser seg”, den ene metoden er å begrense antall iterasjoner og den andre er designe modellene slik at signalverdier konvergerer mot en verdi og at ingen nye verdier skrives til kanaler så lenge forskjellene mellom verdi som skal skrives og forrige verdi er mindre enn en angitt margin. Siste metode er å foretrekke da dette gir bedre kontroll på simulatorfeil. Marginer kan enkelt settes for spesialiserte kanaler, dvs. kanaler som bare kan lese og skrive en bestemt type informasjon, som for eksempel tall. Konvergerende signaler kan bare oppnås ved å designe modeller for dette. Uten konvergerende modeller hjelper det ikke å sette signalmarginer, og simulatoren vil etter hvert nå iterasjonsgrensen og avbryte beregningene brått for så å fortsette til neste simulatortilstand og begynne på nye iterasjoner.

Eksempel
Under er følger et eksempel på en kontinuerlig simulering. I eksemplet skal vi simulere et kloss-fjær-demper –system som blir utsatt for en ekstern kraft i gitt tid for så å fortsette på egen hånd. Klossen tar sum av krefter som innsignal og gir ut et akselerasjonssignal. Demper og fjær tar inn et differensielt akselerasjonssignal som angir forskjell i akselerasjon for endepunktene til *fjær* og *demper*.

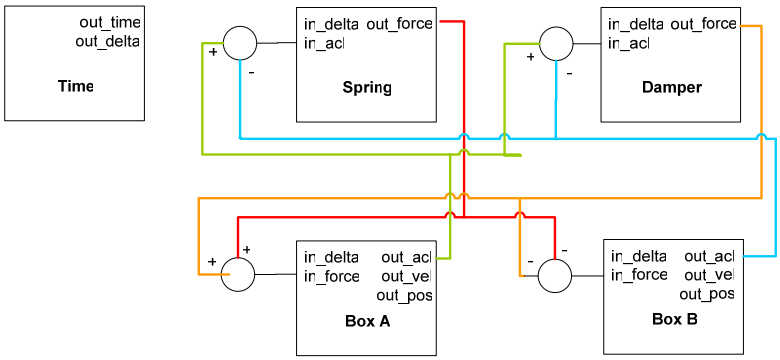
Under følger et klassediagram for systemet hvor de viktigste klassene er med. Som vi ser består systemet av fire hovedmodeller nemmelig *kloss*, *demper*, *fjær* og *tid*. Hver av klassene definerer en rekke kommunikasjonskanaler som offentlige objektattributter, dvs. attributter som er tilgjengelig for andre klasser. På den måten kan kanaler enkelt kobles sammen.



Figur 3.9 – UML kloss-fjær-demper system

Alle modellklasser arver fra klassen Entity og omdefinerer metoden *tick*(). I denne metoden beskrives modellogikken for hvordan utsignaler tilhørende klassen varierer med innsignaler og tilstand. Det benyttes i dette eksemplet klassen *DoubleChannel* for å overføre signaler. *DoubleChannel* er en såkalt spesialisert kanal, dvs. en kanal som arver generell oppførsel fra baseklassen *Channel* men som i tillegg definerer noen nye metoder for å forenkle noen typer operasjoner. I dette tilfellet forenkler *DoubleChannel* bruk av talltype double som signalverdi.

Under er et komponentdiagram for et *kloss-fjær-demper* system. Som vi ser instansieres det to boksklasser med navn henholdsvis Box A og Box B og en instans av modellene for fjær, demper og tid.



Figur 3.10 – Komponentdiagram for kloss-fjær-demper system

Sirklene i diagrammet over er addisjonsinstanser som summerer innsignaler og setter sum på ut-kanal som er koblet til diverse andre instanser. Som vi ser har alle modeller bortsett fra tid en kanal med navn *in_delta*. Denne kanalen er for alle instanser i dette eksemplet koblet til tilsvarende kanal *out_delta* hos instansen *tid*. Kanalen *out_delta* benyttes i eksempel til å signalisere tidssteg, dvs. hvor mye tid tidsinstansen øker klokken med for hver tilstandssteg til simulator.

Diagrammet viser også hvordan kraft som *fjær* og *demper* utover, er positiv for kloss A og negativ for kloss B. Det samme gjelder for akselerasjonssignalet inn til *fjær* og *demper* fra klossene. Akselerasjon fra kloss A er positivt for fjær og demper mens akselerasjon fra kloss B er trekkes fra. Dette fordi differanseakselerasjon benyttes til å modellere endepunktakselerasjon.

Under vises et eksempel på hvordan en kontinuerlig modell kan implementeres. Her ser vi at ved hjelp av innsignalene *in_force* og *in_delta* kan kalkulere ny akselerasjon, fart og posisjon. Legg merke til bruk av tilstandsinformasjon i form av *getState*() metodekall til kanal. For hver iterasjon er for eksempel fart lik fart fra forrige tilstandsberegning pluss fartsøkning representert av innsignalene akselerasjon og tid.

```
public class Box extends Entity
{
    public DoubleChannel in_force = new DoubleChannel(this);
    public DoubleChannel in_delta = new DoubleChannel(this);

    public DoubleChannel out_accel = new DoubleChannel(this);
    public DoubleChannel out_pos = new DoubleChannel(this);
    public DoubleChannel out_vel = new DoubleChannel(this);

    private double m;

    public Box(double mass, double pos)
    {
        this.m = mass;
        this.out_pos.initState(pos);
        this.out_vel.initState(0);
    }

    protected void tick()
    {
        out_accel.write(in_force.read() / m);

        out_vel.write(out_vel.getState() +
                out_accel.read() * in_delta.read());

        out_pos.write(out_pos.getState() +
                out_vel.read() * in_delta.read());
    }
}
```

3.11 – Kildekode for klassen Box

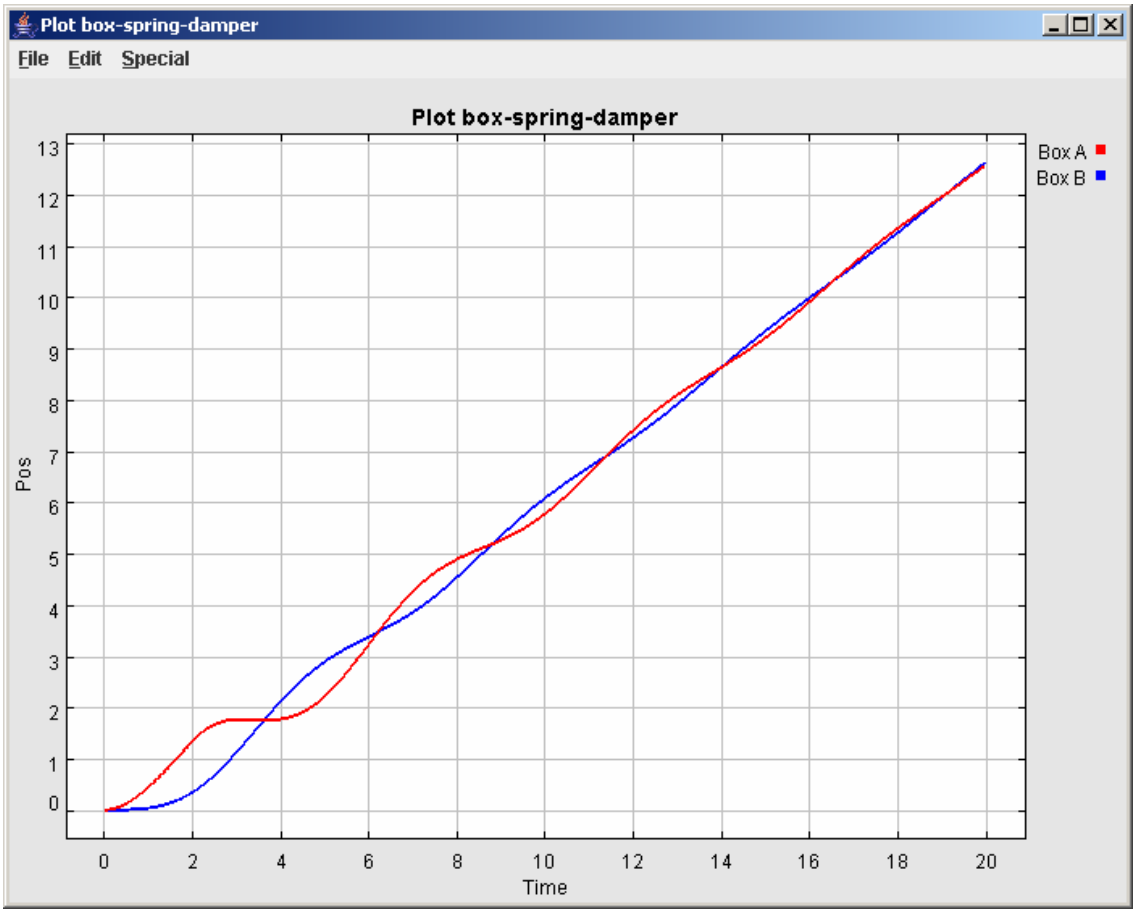
Det er viktig å huske at et tilstandssteg for simulator består av mange iterasjon og at tilstander ikke forandres under iterasjonene. Et tilstandssteg kan for enkelhets skyld sammenlignes med et tidssteg. Iterasjonene benyttes til å beregne nye tilstander, slik som eksemplet over.

Vi kan anta at fartstilstand er 1. Så får vi inn signaler for akselerasjon og tid på henholdsvis 1 og 1. Da blir nye fart 1 + 1*1 som er 2. Så kommer en ny iterasjon med ny akselerasjon på 1.5. Da blir ny fart 1 + 1.5 * 1 som er 2.5. Slik fortsetter det til simulatormodeller konvergerer mot en fast verdi, for eksempel akselerasjon på 1.8. Nå verdier konvergerer fortsetter simulator med en ny gjennomkjøring, nå med fartstilstand 2.8.

Under vises et eksempel på hvordan en simulering av forklart eksempel kan settes opp. Her instansieres modeller og kanaler kobles sammen. Så startes simulering med *sim.play(2000,100)*. 2000 angir antall tilstandssteg simulatoren skal simulere. Med tidsdelta på 0.01 blir dette en simuleringstid på 20 sek. 100 angir maks antall iterasjoner.

```
public class Test
{
    public static void main(String[] args) throws Exception
    {
        Sim sim = new Sim();

        Time time = new Time();
        sim.autoConnect(time.io_delta);
        time.io_delta.initState(0.01);

        Box2 boxA = new Box2(1, 0);
        Box2 boxB = new Box2(2, 0);
        Spring spring = new Spring(1);
        Damper damper = new Damper(0.2);
        Comperator comp = new Comperator();

        spring.in_acc.connect(new Adder().connect(boxA.out_accel)
                .connect(new Negate().connect(boxB.out_accel)));

        damper.in_acc.connect(new Adder().connect(boxA.out_accel)
                .connect(new Negate().connect(boxB.out_accel)));

        comp.in_a.connect(new Constant(2));
        comp.in_b.connect(time.out_time);
        comp.out_if_ina_bigger.connect(new Constant(1));
        comp.out_if_inb_bigger.connect(new Constant(0));

        boxA.in_force.connect(new Adder()
                .connect(comp)
                .connect(new Negate().connect(spring.out_force))
                .connect(new Negate().connect(damper.out_force)));

        boxB.in_force.connect(new Adder()
                .connect(spring.out_force)
                .connect(damper.out_force));

        Logger log = new Logger("Plot box-spring-damper");
        log.connect(time.out_time);
        log.connect(boxA.out_pos, "Box A");
        log.connect(boxB.out_pos, "Box B");

        sim.setDebug(true);
        sim.setDebugMod(1000);
        sim.setDebugLevel(5);
        sim.play(2000, 100);
        log.toPlot();
    }
}
```

Figur 3.12 - *Instansieringskode*



Figur 3.13 – Diagramutskrift kloss-fjær-demper system

Diagrammet over viser resultater av en kloss-fjær-demper simulering. I denne simuleringen blir Kloss A utsatt for en ekstern kraft i 2 sek. Etter dette fortsetter systemet av seg selv uten at noen eksterne krefter påvirker..

4. VERKTØY

I dette kapitelet skal vi snakke prosjektarbeidet som verken hører til under kategoriene simulator eller modell. Jeg har valgt å kalle dette arbeidet for verktøy da arbeidet omfatter små hjelpesystem som brukes i simuleringen. Vi nevner også noen ferdige verktøy som benyttes i prosjekt for ordens skyld.

4.1. UTVIKLINGSVERKTØY

For å realisere simulator, verktøy, modeller, system osv. benyttes programutviklingsverktøyet Java fra Sun Microsystems. Java er en komplett plattform for uvikling og kjøring av Java programvare. Viktige egenskaper for plattformen er rask utviklingstid, god ytelse og multiplattformstøtte.

Produktivitet

Utviklingstid er en viktig faktor ved valg av verktøy i et programutviklingsprosjekt. I dette prosjektet settes utviklingstid høyest, dvs. over faktorer som for eksempel ytelse. Java tilbyr veldig gode vilkår for å øke produktivitet og dermed minske utviklingstid. Et viktig bidrag til god produktivitet er at Java kommer med et omfattende bibliotek av ferdige funksjoner i form av klasser og rammeverk som kan brukes fritt i eget prosjektet. I dette biblioteket finnes i stikkordsform for eksempel rammeverk for filbehandling, kommunikasjon over nettverk, lage brukergrensesnitt, multimedia, objektlister, komprimering, kryptering, databasetilkobling, XML, osv. Listen er lang.

Community

I tillegg til biblioteket som følger med plattformen finnes det mye programvare, verktøy osv. fritt tilgjengelig på Internet i form av ”Open source” programvare. Java er et veldig populært verktøy og det finnes derfor mye som er laget og ligger fritt tilgjengelig for andre interesserte på Internet. For eksempel kan man laste ned en Jakarta Tomcat fra Apache Community som er en gratis webserver for å lage dynamiske websider i Java. Den store tilgjengeligheten av fri programvare laget av ”the community” bidrar også til god produktivitet.

Hotspot
Java er et ”interpreted” språk, dvs. at kildekode ikke oversettes direkte til maskinkode som en datamaskin forstår men til ”bytecode” som er Java -plattformen sin utgave av maskinkode. Ved kjøring av Java programvare må man benytte et annet program til å oversette ”bytecode” til maskinkode slik at datamaskinen kan kjøre programmet. Dette har historisk vært et problem i form av trege Java -applikasjoner pga. oversettelsestrinnet, som ikke andre verktøy som for eksempel c og c++ var plaget med. I dag benyttes en rekke avanserte teknikker til å oversette ”bytecode” til maskinkode på en måte som gjør at man til og med kan oppnå raskere kjøring av Java -kode enn tilsvarende c++ kode. For eksempel benyttes HotSpot kompilering og optimalisering. I tillegg til å oversette bytecode til maskinkode ligger HotSpot i bakgrunnen og profilerer kodegjennomkjøring. Når HotSpot identifiserer noen deler av koden som kjører ofte og som ikke er optimalisert, vil den bruke en god del energi på å optimalisere disse delene. I denne optimaliseringsprosessen vil java benytte seg av alle tilgjengelig hjelpemidler som for eksempel SSE3 instruksjonsett som nyere prosessore er utstyrt med. Dette kalle adaptiv optimalisering. Dette kan for eksempel ikke et C++ program gjøre da slike programmer er statisk kompilert, dvs. at man på forhånd må optimalisere for minste felles multiplum når det gjelder instruksjonssett som alle potensielle kunder kan ha. Man kan ikke garantere at alle har for eksempel SSE3 og kan derfor ikke kompilere for dette selv om for eksempel 80% har dette instruksjonssettet tilgjengelig.

Multiplatform
En viktig grunn til at man i Java ikke oversetter kildekode direkte til maskinkode er for å kunne kjøre Java-applikasjoner på flere prosessorarkitekturer samt flere plattformarkitekturer. For å få til dette benytter Java seg av et prinsipp som kalles virtuell maskin, eller VM. Java -applikasjoner kjørers ved å starte en slik virtuell maskin som igjen laster og kjører Java -applikasjonen. Den virtuelle maskinen oversetter da bytecode, som er maskinkode for den virtuelle maskinen, til plattformspesifikke instruksjoner. Hovedpoenget med en slik løsning er å kunne kjøre Java -programmer på mange forskjellige plattformer uten tilpasninger i kildekode, eller kompilert kode, ved hjelp av slike virtuelle maskiner. Man lager en virtuell maskin for hver plattform som ønskes støttet. I dag støttes Solaris, Windows og forskjellige varianter av Linux.

4.2 KARTSYSTEM

Et grafisk kartsystem er utviklet for dette prosjektet. Kartsystemet er nyttig fordi kollektivsimulering i prosjektet er geografisk orientert. Kartsystemet benyttes til bl.a. å visualisere geografisk data som kart, symboler, punkter, strekninger og lignende. Kartsystemet benyttes også til å forenkle inntasting av geografisk data til simulator og til å visualisere simulatorresultater.

Implementasjon

Kartsystemet er utviklet for å være modulært, dvs. at det lett kan utvides til å vise forskjellige typer geografisk informasjon. Til grunn i kartsystemet ligger et matrisesystem. Matrisesystemet benyttes til å holde orden på forskjellige lineære transformer samt koordinatdata. Lineære transformer benyttes for å tegne geografisk data fra forskjellige koordinatsystemer i samme koordinatsystem. Transformer benyttes også for å holde orden på hvilket geografisk utsnitt som skal vises, dvs. kamera.

Kartsystemet opererer med et selvutviklet koordinatsystem som referansesystem som andre koordinatsystemer transformeres til når data tegnes slik at data blir riktig presentert i forhold til hverandre. Grunnen til at det brukes et selvlaget referansesystem i stedet for ett av mange utbredte koordinatsystemer er først og fremst for å normalisere tallstørrelser og for den andre å kunne bruke vanlige flyttall til å representere en koordinat. Hva slags referansesystem som benyttes er ikke så viktig da man enkelt kan transformere til og fra andre systemer. Det som er viktig er å ha et godt definert referansesystem som andre data med andre referansesystemer kan relateres til.

Referansekoordinatsystemet som benyttes i kartsystemet bygger på det kartesiske koordinatsystemet, dvs. det systemet vi er kjent med fra matematikken hvor koordinater beskrives med en x og en y koordinat for to dimensjoner og z for den tredje hvis den eksisterer. Koordinatsystemet har origo som vist ved kryss i figur under på en molokant på Ila nordvest for Trondheim sentrum. En meter øst er definert som en positiv enhet x og en enhet sør er definert som en positiv enhet y. Himmelretninger er her i forhold til virkelig nord og ikke magnetisk som hele tiden varierer.

Figur 4.1 Kart med referansepunkt

Det kan virke litt risikabelt å definere origo for et referansesystem til en kant på en fysisk molo som faktisk kan forandre seg i løpet av tiden. Dette er ikke noe problem da jeg har transformmatrise fra referansesystem til andre systemer som for eksempel det kjente koordinatsystemet UTM med datum WGS 84. Jeg kan da bruke inverstransformen for disse til å finne tilbake til origo og referansesystem hvis molo for eksempel blir borte. Det er det som i praksis blir gjort da jeg taster inn data ut ifra andre kart som så blir transformert til referansesystem. Kant på molo blir aldri brukt i noen annen sammenheng enn definisjonsbeskrivelse av referansesystem.

Koordinatsystemer blir i verktøyet definert som lineære transformasjoner av referansesystemet beskrevet i forrige avsnitt. Dette er ikke helt riktig da forskjellige koordinatsystemer ikke alltid kan beskrives som en lineær transformasjon av en annen. Dette fordi jorden ikke er flat og forskjellige systemer kan bruke forskjellige mål for bl.a. krumhet av jordoverflate. Med flat mener vi ikke overflate uten bakker og fjell, men at avstanden fra havoverflaten til sentrum av jorden er lik over alt. Dette betegnes som datum for et koordinatsystem og tar hensyn til at jorden ikke er helt sirkelrund. Denne effekten gir ikke så store ulineære utslag for avgrenses geografiske områder. Så i dette verktøyet antar vi at jorden er flat i og rundt Trondheim.

For å transformere mellom koordinatsystemer benyttes transformasjonsmatriser. For hvert koordinatsystem lages en transformasjonsmatrise som mapper koordinater fra det gjeldende koordinatsystem til referansesystemkoordinater. Transformasjonen er av type ”affine transform”, som vil si at i tillegg til at transformasjonene er lineære blir også parallellitet mellom rette linjer tatt vare på slik at parallell linjer i det ene koordinatsystemet er parallell også i det andre.

Ved å bruke slike transformasjonsmatriser kan man enkelt sette sammen koordinatsystemer i et hierarki ved å multiplisere sammen transformasjonsmatriser. For eksempel kan man enkelt finne UTM koordinater fra pikselkoordinater til et kartbilde hvis man vet transformen fra bilde til referansesystem og fra UTM til referansesystem. Ved å multiplisere bildetransformasjon med den inverse av UTM transformasjonen får man en ny matrise som beskriver transformasjonen direkte fra piksel til UTM - koordinater. For å lage en transformasjon fra UTM til bildekoordinater, altså motsatt, er det bare å invertere matrisen. Et slikt system gjør det enkelt for kartsystemet å behandle ulike koordinatsystemer.

En transformasjon beskrives som sagt ved hjelp av matriser. For en to-dimmensjonal ”Affine” transformasjon benyttes en 3 x 3 transforamsjonsmatrise. Matrisen beskriver hvordan man ved hjelp av en translasjon etterfulgt av en variende grad av skalering, rotering og vridning kan komme fra den en systemet til det andre.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & Kx & Tx \\ Ky & Sy & Ty \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$x' = Sx \cdot x + Kx \cdot y + Tx$$
$$y' = Sy \cdot y + Ky \cdot x + Ty$$

Over ser vi hvordan en vanlig lineær 2D transformeringsmatrise er utvidet til også å inneholde en translasjonskomponent. Med dette kan vi nå transformere rundt et vilkårlig punkt og ikke bare rundt origo. En slik transform kalles som nevnt tidligere ”Affine transform”. *Tx* og *Ty* er translasjonskoordinater. Gitt at *Kx* og *Ky* er 0 da er *Sx* og *Sy* skaleringsfaktorer og gitt det motsatte da er *Kx* og *Ky* forvridningsfaktorer.

For å lage slik transformer uten å prøve seg frem med ulike verdier for translasjon, rotasjon, skalering og vridning kan man regne seg fram til en transform ved hjelp av 3 koordinater fra hvert system og litt algebra. Hver av de tre koordinatparene må representere samme fysiske lokasjon for at en transform skal kunne beregnes og må ikke ligge på en linje. Et eksempel kan klare opp litt.

Hvis vi skal mappe et bildekart av piksler til UTM koordinatsystem må vi finne tre koordinater fra hvert system som representere samme posisjon. Dette kan for eksempel gjøres ved å finne tre posisjoner på bildekartet og finne tilsvarende UTM -koordinater for disse med GPS eller lignende verktøy. Da har vi tre pikselkoordinater fra bilde og tilsvarende tre UTM koordinater fra GPS. Med slik informasjon kan vi da lage en transformasjonsmatrise. Dette gjøres enkelt ved å løse følgende ligningssett for de tre koordinatparene, hvor X og Y er koordinater vi skal transformere fra og X' og Y' er koordinater fra systemet vil skal transformerer til. A, B, C , D, E og F er her transformasjonsverdier. Merk at bare to av i alt seks ligningssett er vist her. De fire andre er helt lik men med de to andre koordinatparene. Dette belyses bedre i neste avsnitt.

$$AX + BY + C = X'$$
$$DX + EY + F = Y'$$

I matriseform kan problemet settes opp som vist under. Den første matrisen inneholder som over koordinater fra system vi skal transformere fra. Fra eksemplet blir da første matrise pikselkoordinater fra bildekart. Denne matrisen skal så multipliseres med en transformasjonsmatrise, som blir den midterste matrisen, for å få koordinater i UTM koordinatsystem, som blir matrisen til høyre. Vi løser så ligningen med hensyn på matrisen i midten, som altså blir transformasjonsmatrisen.

$$\begin{bmatrix} X_1 & Y_1 & 1 \\ X_2 & Y_1 & 1 \\ X_3 & Y_3 & 1 \end{bmatrix} \bullet \begin{bmatrix} A & D \\ B & E \\ C & F \end{bmatrix} = \begin{bmatrix} X'_1 & Y'_1 \\ X'_2 & Y'_2 \\ X'_3 & Y'_3 \end{bmatrix}$$

Ved å løse ligningssettet over får vi ut transformasjonsmatrisen som svar.

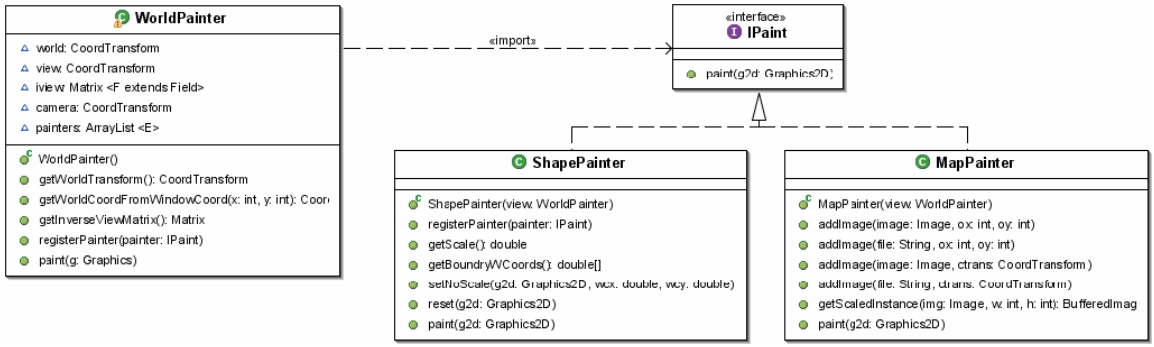
Kamera

Med et slikt matrise-transform-system på plass for å holde orden på koordinatsystemer er det rimelig enkelt å videreutvikle systemet med et virtuelt kamera, dvs. en transform som representerer hva slags utsnitt som skal vises, dvs. origo og zoom -nivå i forhold til referansesystem. På den måten kan man enkelt bevege seg rundt i kartsystemet ved å manipulere kameramatrisen alene.

Det virtuelle kameraet er implementert som en vanlig transformasjonsmatrise. Koordinater til kameraet er translasjonskoordinater i matrisen og zoom er skaleringsfaktorer. Når kartsystemet skal tegne geografisk data blir først kameramatrisen invertert og så lagt til grunn for alle andre transformasjoner i systemet, dvs. at den inverse kameramatrisen multipliseres med andre transformasjonsmatriser som igjen multipliseres med andre matriser i hierarkiet og i den rekkefølgen. På den måten får vi transformert for eksempel bildekoordinater ned til referansesystem og så videre opp til kamerasystem. I kamerasystemet som er implementert i dette prosjektet styres kamera enkelt vha. mus. Man klikker og drar for å panorere og bruker musehjulet for å zoome.

Arkitektur
Kartsystemet er delt opp i de tre hovedklassene *WorldPainter*, *MapPainter* og *ShapePainter*. *WorldPainter* er hovedklassen og har ansvaret for å sette opp grafisk brukegrensesnitt for system slik at systemet har noen plass å tegne samt sette opp transformasjonsmatriser for referansesystem og kamera. *WorldPainter* kobler også sammen kameramatrisen med musebevegelser i det grafiske brukergrensesnittet slik at brukeren kan manipulere det virtuelle kameraet og bevege seg rundt i kartsystemet.

WorldPainter tegner ikke noe selv men tilrettelegger slik at tegning av geografisk data blir enklere for andre klasser. Klasser som ønsker å tegne geografisk informasjon registrerer bare en klasseinstans hos *WorldPainter* ved å kalle metoden *registerPainter(IPaint painter)*. Slike klasser må implementere grensesnittet *IPaint* som bare definerer en enkel tegnemetode. Når dette er gjort vil *WorldPainter*, når den skal tegne, gå gjennom alle registrerte *IPaint* -komponenter og la disse tegne seg selv vha av metoden *paint(Graphics2D g)* som definert i grensesnittet *IPaint*. Metoden *getInverseViewMatrix()* hos *WorldPainter* benyttes da som grunnmatrise for alle *IPaint* -komponenter.



Figur 4.2 – UML diagram for kartsystem

MapPainter er en enkel klasse som tar seg av tegning av bildekart for kartsystem. Bilder registreres i *MapPainter* med en tilhørende transformasjonsmatrise. *MapPainter* må som andre tegneobjekter registrere seg med *WorldPainter*. Når *WorldPainter* tegner opp brukergrensesnitt vil da *MapPainter* etter hvert få kontroll og tegne de registrerte bildekartene i registrert rekkefølge og i henhold til invers kameramatrise fra *WorldPainter* multiplisert med transformasjonsmatrise for kart.

ShapPainter er en klasse for tegning i kartsystem med Graphics2D rammeverkt fra Java. Med dette rammeverket kan man tegne vektorbasert grafikk, som for eksempel linjer, sirkler, tekst osv.

Med *ShapePainter* tilpasse operasjoner i Graphics2D koordinatsystemer i stedet for piksler i et vindu. Man kan for eksempel tegne en sirkel med UTM senterkoordinater og gitt antall meter radius i stedet for x,y pikselkoordinater. Man kan også med *ShapPainer* velge om det som tegnes skal tegnes uavhengig av kamera-zoom, dvs. om symbolet skal være like stort uansett hvor mye man zoomer, eller om symbolet skal følge referansesystemet, dvs. at symbolet har en fast radius, for eksempel på 5 meter.

For å tegne med *ShapePainter* må man registrere en tegneklasse hos *ShapePainter*. En slik klasse er en klasse som implementerer grensesnittet *IPain* som nevnt tidligere. Når *ShapPainter* skal tegne seg selv, fra forespørsel fra *WorldPainter*, vil den så gå gjennom alle registrerte objekter og la disse tegne seg selv vha. tjenester i Graphics2D og *ShapPainter*.

4.3 KONFIGURASJONSSYSTEM

For å lese og skrive konfigurasjonsdata er det utviklet et generelt rammeverk for behandling av trestrukturinformasjon. Rammeverkt støtter lesing og skriving av strukturert informasjon, til og fra ren tekst som for eksempel tekstfiler. Rammeverket støtter også traversering og filtrering av informasjon slik at man enkelt kan lage datautsnitt av de data som er interessante.

Mye av funksjonaliteten til konfigurasjonsrammeverket er implementer av bare en stor klasse med navn Node. En node, dvs. en objektinstans av klassen Node, består av et navn og en verdi av typen tekst. En node kan også inneholde andre noder, dvs. noder som da blir barn til den noden som inneholder disse. På den måten kan man organisere navn-verdi-par-informasjon i en trestuktur.

Klasse Node er stor fordi den inneholder mange hjelpefunksjoner for å gjøre lesing og skriving av informasjon enklest mulig. Mange metoder som er implementert i klassen kan med enkelhet utføres av en kombinasjon av andre metoder. Dette er også gjerne hvordan mange av disse hjelpemetodene er implementert.

Et eksempel på navn-verdipar informasjon organisert i trestuktur:

```
familie{
  hund{
    navn = Voffsen
  }
  person{
    fornavn = Ola
    etternavn = Norman
  }
  person{
    fornavn = Kari
    etternavn = Norman
  }
}
```

Figur 4.3 - Konfigurasjonseksempel

Her ser vi at rotnoden med navn ”familie” inneholder tre barnenoder, en node med navn hund og to noder med navn ”person”. I hver av person -nodene har vi informasjon om fornavn og etternavn i form av noder med verdi i tilegg til navn.

Rammeverket kan enkelt skrive og lese slik tekst fra for eksempel tekstfiler. Når rammeverket skal skrive slik informasjon begynner man med rotnodeobjektet og går så systematisk igjennom alle noder og skriver ut informasjon om navn og verdi for disse i tekstformat som igjen lagres i fil. For å lese slik informasjon fra en tekstfil kreves det litt mer. Til dette er det utviklet en parser, dvs. et program som kjenner igjen tegn og tegnsammensetninger, som kalles ord, og basert på rekkefølgen av slike klassifiserte ord bygger opp et nodetree.

Rammeverket definerer en rekke funksjoner for å finne informasjon, dvs. en bestemt node. Dette kan gjøre ved å gå gjennom alle noder systematisk fra rotnode til man finner den aktuelle noden. Dette er lite effektivt da man i slike søk må gå gjennom alle noder og ikke utnytter det man vet om hvor i trestruktuen aktuell informasjons finnes. For eksempel hvis man skal lese etternavn og fornavn til en person kan man utnytte at slike noder finnes under rotnode familie og undernode person og ingen andre plasser. På den måten slipper vi å søke gjennom node hund.

For å forenkle denne prosessen kan man definere spørringer vha. av noe som ligner XPath adresser kjent fra XML -verdenen. For eksempel hvis vi vil finne alle fornavn-noder til bare personer kan vi utføre spørringen vha. metoden *selectAll(”familie/person/etternavn”)* på rotnoden. Denne metoden gir tilbake en liste av noder som oppfyller kriteriene definert som tekstargument til metoden. Her spør vi etter en etternavn-node som har foreldrenode med navn person og igjen har foreldrenode med navn familie. Man kan også snevre inn søket ytterligere til å gjelde for eksempel bare personer med fornavn ”Kari” ved spørringen *selectAll(”familie/person[fornavn=’Kari’]/etternavn”)*. Her gjelder det samme som for forrige eksempel med et ytterligere krav om at person-noden må ha en fornavn-node med verdi ”Kari”.

Konfigurasjonsrammeverket har mye til felles med diverse XML teknologier, som også er en måte å organisere informasjon i en trestuktur. Med til forskjell fra XML som er bundet av tunge standarder er konfigurasjonsrammeverket i dette prosjektet utviklet for å være lett i bruke. Man unngår for eksempel krav som støtte for forskjellige tekstkodinger, at noder kan organiseres som både attributter og elementer og at tekst kan eksistere rundt elementer. Alt dette pluss mye mer gjør XML –teknologi ”tung” og uegnet til bruk som konfigureringsverktøy for dette prosjektet, da informasjon må leses ofte. Da rammeverket utviklet i dette prosjektet innholder en undergruppe av de egenskaper XML -standarden spesifiserer kan man konvertere fra noder til XML. Det motsatte er ikke tilfellet. Noe interne tester viser at rammeverket implementert i dette prosjektet er rundt regnet 1000 ganger raskere en hva en DOM4J er på XPath søkeoperasjoner for tilsvarende informasjon. DOM4J er et populært XML -rammeverk. Dette er ikke fordi DOM4J implementasjonen er dårlig og min er bra men fordi min implementasjon er veldig enkel.

5. MODELLER

I dette kapittelet skal vi snakke om simulatormodellene som brukes i prosjekt for å simulere kollektivtrafikk.

5.1 INTRODUKSJON

For å simulere kollektivtrafikk, slik at posisjoneringsteknologier, algoritmer og system for beregning av ruteforsinkelser kan evalueres, er det utviklet en rekke simulatormodeller som etter beste evne til sammen prøver å emulere oppførsel og egenskaper til et virkelig kollektivsystem.

I dette prosjektet er det under modelleringsfasen tatt utgangspunkt i rute 52 hos Team Trafikk i Trondheim. Dette er bare et utgangspunkt og det er på ingen måte et prosjektmål å simulere rute 52 nøyaktig.

Derimot er et viktig prosjektmål å simulere en typisk kollektivrute nøyaktig nok slik at resultater fra simulering kan brukes til å evaluere det som skal evalueres. Dette vil si at modeller vektlegger egenskaper og oppførsel som antas å ha betydning forsinkelsessystem.

Simulatormodellene som er utviklet prøver etter beste evne å simulere hvordan busser i kollektivtrafikk oppfører seg når de kjører kollektivruter. Eksempler på kollektivtrafikkfenomener som modelleres og simuleres er:

- Busser kjører en planlagt serie turer pr dag.
- Ved endeholdeplasser venter buss til neste planlagt avgang
- Busser akselererer og bremser.
- Busser tilpasser fart etter kjøreforhold som svinger, trafikk og fartsgrense.
- Forskjellige busskjåfører har forskjellig aggressivitet.
- Samme bussjåfør har på samme dag samme aggressivitet på ruteturene.
- Når buss ligger etter skjema vil sjåfører i varierende grad prøve å ta inn dette.
- Buss stopper ved ruteholdeplasser og stoppetid varierer.
- Rushtrafikk om morgen og ettermiddag.
- Noen områder har forskjellig trafikkmønster enn andre.
- Lyskryss og andre store veikryss har et periodisk stoppemønster.
- Trafikkforhold er forskjellig fra dag til dag, men trendene er lik.

Med en tur menes her en rutegjennomkjøring. En buss kan kjøre samme rute mange ganger i løpet av samme dag. En slik gang benevnes med ordet tur i dette prosjektet, eller engelsk ”trip”.

Det er viktig å huske at modellene er utviklet etter beste skjønn og baseres på mange antagelser slik at modellene på ingen måte representerer den virkelige rute 52. For å gjøre det enklere kan man tenke seg at simulatoren gir ut data fra en fantasirute i en fantasiverden, som ligner på rute 52. Spørsmålet som skal besvares er hvilke resultater systemet på denne fantasi ruten kan vise til og om resultater kan videreformidles til virkelige ruter i den virkelige verden, som for eksempel Team Trafikk sin rute 52 i Trondheim.

5.2 RUTE 52

Som nevnt tidligere tar prosjektet utgangspunkt i rute 52 hos Team Trafikk i Trondheim. Denne ruten strekker seg fra Pirsentret i nord til Vestlia i sør. Figuren under viser noen veistrekninger med forskjellige farger. Disse strekningene inngår i rute 52 og benyttes av simulator til å finne ut hvor bussen kan kjøre.

Figur 5.1 – Kart over rute 52

Rute 52 er delt inn i fire forskjellige underruter med navn 52.P, 52.P.A, 52.S og 52.S.A Underrutene 52.P og 52.S er den samme ruten bare med motsatt kjøreretning, hvor underrute 52.P går fra nord til sør, dvs. fra Pirsenter til Vestlia endeholdeplass, og 52.S går motsatt. Underrutene 52.P.A og 52.S.A er ruter som ikke går via Othilienborg på tur fra og til sentrum, dvs. blå strekning på kart.

Holdeplasser er markert med rød prikk med tilhørende holdeplassnavn.

5.3. OMRÅDER

Simulatoren kan simulere fenomener som er geografisk begrenset til et område. For å få til dette benyttes en enkel polygonbeskrivelse av gjeldende område. Slike områder benyttes av simulator til for eksempel å modellere lokale trafikkmønster, lyskryss osv. I figurene under kan vi se at Trondheim sentrum er markert vha. av et slikt polygon, samt noen lyskryss.

Med dette systemet kan man lage modeller som bare påvirker oppførsel til andre modeller som er innenfor markert område.

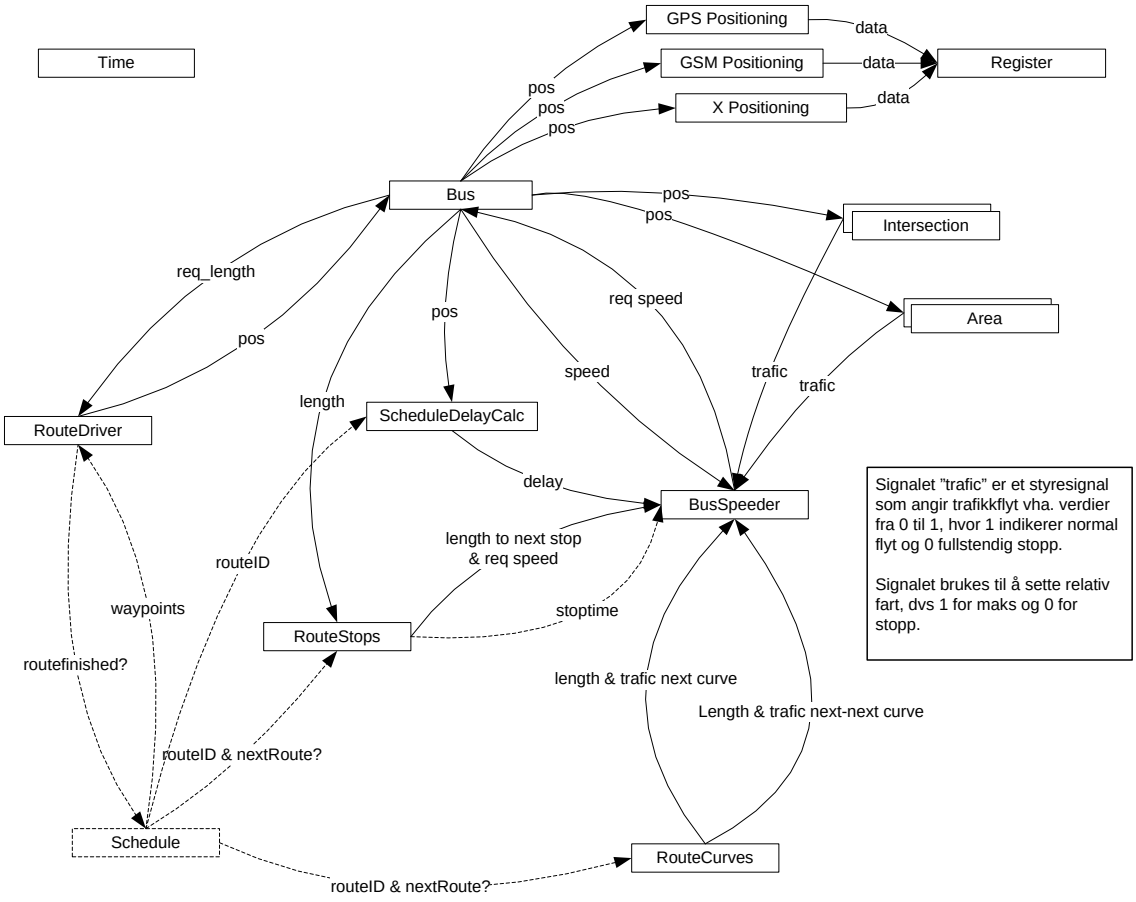
5.4 MODELLKONFIGURASJOJN

For å konfigurere simulatormodeller brukes konfigurasjonsrammeverket beskrevet i kapittel 6 Verktøy. Konfigurasjonsdata benyttes bl.a. til å beskrive rutedata som avgangstider, hvilken buss som skal kjøre hvilken rute, hvilke veiestrekninger som skal benyttes samt informasjon om kryss og andre områder.

5.5 OVERSIKT MODELLER

Det er utiklet en rekke simulatormodeller som i felleskap prøver å imitere et kollektivsystem. Systemet består bl.a. av modeller for buss, rutesystem, posisjonering og lignende. De fleste modellene er kontinuerlige og er koblet sammen ved hjelp av kanaler, eller signallinjer om du vil. Diskrete modeller benytter vanlige metodekall ved kommunikasjon.

Under er det vist en figur av viktige modeller for systemet og hva slags signaler som kommuniseres dem imellom. Heltrukket linje indikerer kontinuerlige operasjoner, mens stiplede linje indikerer diskrete operasjoner.

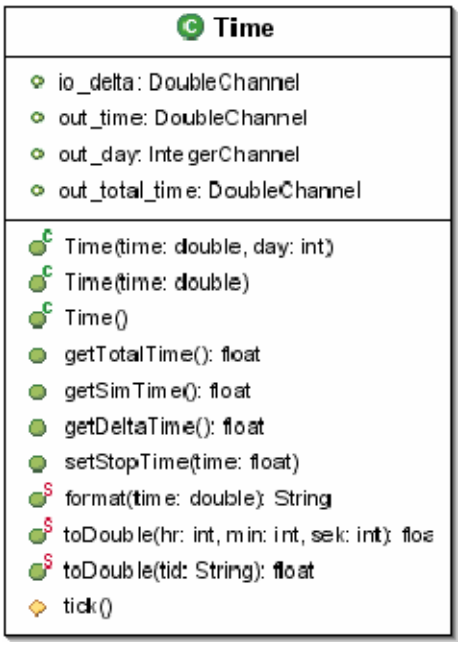


Figur 5.2 Oversikt modeller

I figur 5.2 over kan vi se hvordan modeller som styrer oppførsel til rutebusser er organisert. Diagrammet viser sammenkoblinger for en buss. En simulering kan bestå av mange samtidige rutebusser. Disse er for normalt uavhengig av hverandre bortsett fra modellene Intersection og Area hvor samme modellinstans er koblet til alle de forskjellige bussinstansene.

Vil i følgende av snitt forklare de viktigste modellene som inngår i simulator. Konkrete modellverdier, fordelinger og lignende som for eksempel busshastighet nevnes ikke da dette er en konfigurasjonssak som forandres fra simulering til simulering. Verdier av betydning for simulasjonsresultater vil beskrives i sammenheng med resultater.

5.5.1 TIME



Time – modellen er en modell som simulerer tid. For å gjøre dette må modellen initialiseres med en start tid og en tidsdelta og så vil modellen øke tiden med en tidsdelta pr tilstandssteg simulatoren gjør.

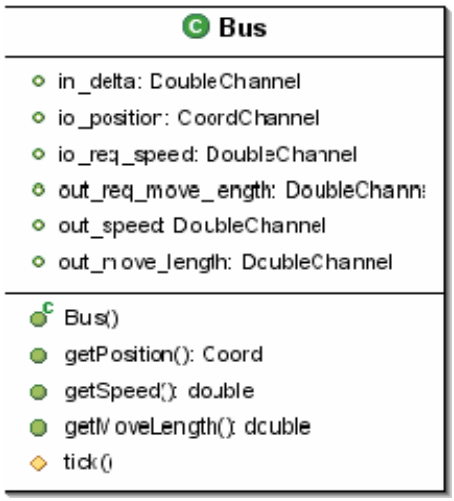
Simulatortid kan leses fra andre modeller i simulator via kanalen til modellen. Her er alle kanaler utkanaler bortsett fra io_delta som er både inn og ut-kanal. Denne kanalen styrer tidsdelta, dvs. hvor mye tid modellen skal øke simulatortiden med for hvert tilstandssteg. Hvis ingen modeller skriver til denne kanalen forblir tidsstegene i simulator faste med initialisert verdi som tidsdelta.

Ved å koble seg til *io_delta* – kanalen kan man altså modellere systemer med dynamisk tidssteg og da modellen er kontinuerlig trenger man ikke å sette delta-verdi før andre modeller rekker å lese tidsverdi fra modell. Man kan sette ny delta-verdi vilkårlig og alle modeller som er avhengig av denne verdien vil beregne ny tilstand ved å starte en ny iterasjon, inkludert tidsmodellen selv.

Man kan med andre ord sette tidsdelta basert på simulatordata. For eksempel hvis en modell oppdager at en buss fra et tilstandssteg til neste beveger seg inn på et område så kan modellen via binærsøkemetoden modifisere tidsdelta slik at simulator ender opp i den tilstand hvor buss passerer områdegrense, med konfigurerbare nøyaktighet. På den måten kan nøyaktighet økes betraktelig uten at alle tidssteg må være små.

Metoden tick() arves fra baseklassen Entity som beskrevet i kapittel 5. Simulator. Her implementeres modellogikk som i dette tilfellet bare inkrementerer tiden med tidsdelta. Metodene med bokstaven C i navnet er forskjellige konstruktrørmetoder til klassen. Her legges initialiseringkode. Grønn farge både for metoder og attributter indikerer at ressursen er tilgjengelig for alle andre klasser, gul og rød indikerer begrenset tilgang.

5.5.2 BUS



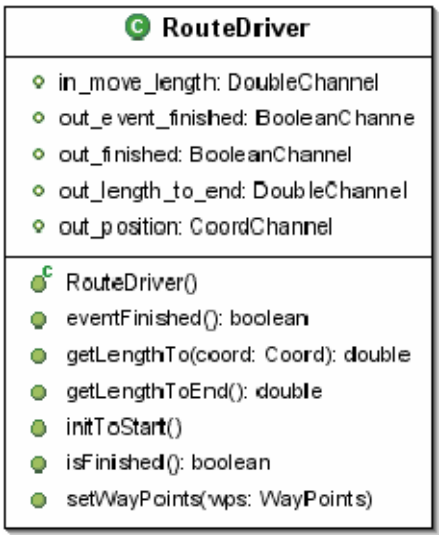
Bus er en enkel simulatormodell som representerer en virtuell buss. Modellene er ”dum” i og med at den alene ikke gjor noe som helst. Tilstand for modelle er helt avhengig av input fra andre modeller i simulator.

Som vi ser av klassediagrammet til høyre består modellen, som de fleste andre modeller i systemet, av en rekke ”get” metoder for avlesing av tilstander i tillegg til kanaler selv om modellen er en ren kontinuerlig modell. Disse metodene er hjelpemetoder for eventuelle diskrete modeller slik at informasjon kan benyttes av disse også. Dette er rene lesemetoder, dvs. at man ikke kan endre tilstand ved å kalle disse metoden. Tilstand kan kun endre ved. å skrive signaler til kanaler.

Rollen til modellen i simulering er å representere en samlemodell som andre modeller kan bruke til å ”forhandle” frem simulatortilstander, som for eksempel modellen BusSpeeder og BusDriver gjør. Disse modellene snakker sammen ved hjelp av signalene i denne modellen.

BusSpeeder begynner i hver simulatoriterasjon med å sette en bussfartverdi basert på verdier fra andre modeller via kanalen *io_req_speed*, hvor *io* står for in/out dvs at kanalen kan brukes som en inn og ut kanal og *req* står for request. Dette blir av bussmodellen omgjort til lengdeverdi vha. av tidsverdi fra *in_delta*-kanalen som angir tidsstegverdi fra tidsmodell i simulering. Lengdeverdi blir så skrevet ut via kanalen *out_req_move_length*. Som vi ser av oversiktsfigur blir dette signalet brukt av *RouteDriver* modellen. Denne modellen kalkulerer så ny posisjon basert på gammel posisjon, ønsket kjørelengde som spesifisert fra signal og ruteinformasjon som hvilke veistrekninger som gjelder osv. Denne nye posisjonen blir så sendt tilbakt til bussmodellen og videreformidlet til andre modeller som lytter på posisjonssignal fra buss. Den nye posisjonen kan i noen modeller resultere i andre vilkår for busshastighet slik at BusSpeeder igjen skriver ny fart på kanalen *io_req_speed*, og på den måten vil simuleringen iterere til fartsverdi for buss konvergere mot en verdi som alle modeller enes om.

5.5.3 ROUTEDRIVER



RouteDriver -modellen benyttes til å styre posisjon til bussmodell ved å spesifisere en flyttelengde og hvilken rute som skal kjøres.

Hvor langt bussen skal flyttes fra en tilstand til neste får modellen vite via *in_move_length* kanalen som i simulator er koblet direkte til bussmodell. Hvor bussen skal flyttes kalkuleres basert på tilstandsposisjon og rute som kjøres. Når ny posisjon er klar oppdateres *out_position* –kanalen som i simulator er koblet tilbake til buss.

RouteDriver er koblet opp mot Schedule klassen som administrerer alt som har med ruter og ruteturer for hver buss å gjøre. Fra denne klassen får RouteDriver informasjon om rute som skal kjøres i form av en rekke rutepunkter, waypoints, som skal navigeres.

Når modellen har kjørt ferdig ruten signaliseres dette ved hjelp av kanalene *out_finished* og *out_event_finished*. Begge kanalene indikerer at ruten er ferdigkjørt, men *out_event_finished* indikerer dette i det tilstandssteget det skjer og går tilbake til vanlig tilstand etterpå selv om ruten fortsatt er ferdigkjørt, derav event som betyr hendelse. Slike signaler benyttes til å trigge noe.

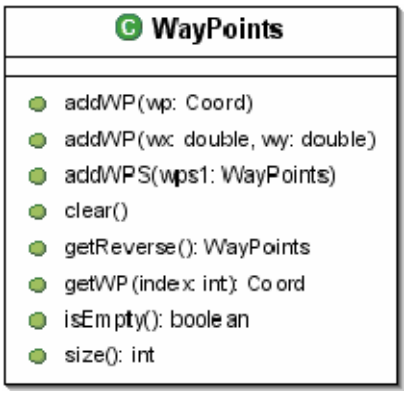
Finished-signalet benyttes bl.a. til å kommuniserer til Schedule av rute er ferdigkjørt. Ved en slik hendelse vil Schedule finne ut hvilken rute som er planlgt for bussen i neste runde samt starttid for gjennomkjøring og kommunisere dette til RoutDriver via metoden setWayPoints(wps). Starttid kommuniseres til BusSpeeder som tar seg av alt som har med fart å gjøre. Schedule er som modellfiguren viser en diskrete modell og bruker derfor metoder for å kommunisere ruteinformasjon. På den måten er RouteDriver både kontinuerlig og diskrete, dvs den benytter data fra begge domene til å beregne ny tilstand.

Metoden getLengthTo(coord) benyttes av andre modeller for å beregne veiavstand i forhold til oppsatt rute og en koordinat. Dette tjenesten benyttes bl.a. modellen RouteStops for å beregne avstand til neste holdeplass som igjen benyttes av andre modeller for å andre ting.

Modellen er implementert rundt hjelpeklassene WayPoints, WPUtils og WPDriver. WPUtils brukes til å sette opp ruten, WayPoints holder på rutekoordinatene og WPDriver benyttes til å beregne nye posisjoner basert på hvor langt man beveger seg langs ruten.

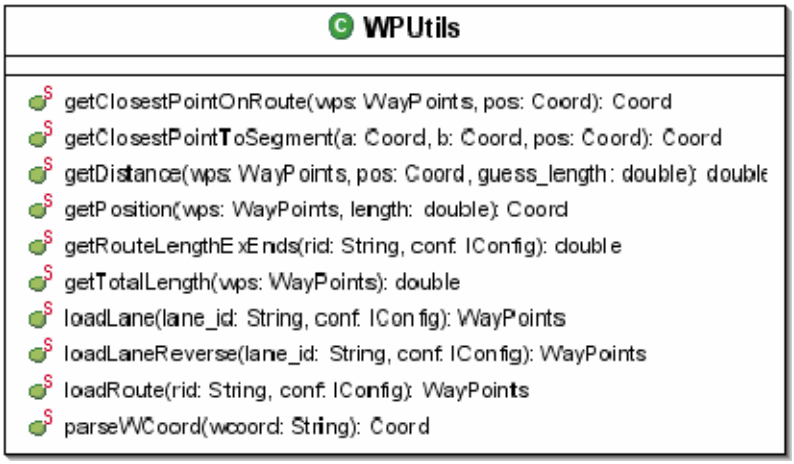
5.5.4 WAYPOINTS

Veibeskrivelser for ruter er organisert som en rekke veikoordinater, waypoints, som beskriver endepunkter for små veistrekningene som hver rute er bygd opp av. Klassen WayPoints er en klasse som brukes til å holde en liste av slike veikoordinater som til sammen danner en rute. Klassen har metoder for å legge til koordinatoer, hente ut og slette koordinater. Klassen kan også returnere en ny instans av seg selv med koordinater i motsatt rekkefølge. Dette er praktiske hvor for eksempel bussen skal kjøre samme rute tilbake.



5.5.5 WPUTILS

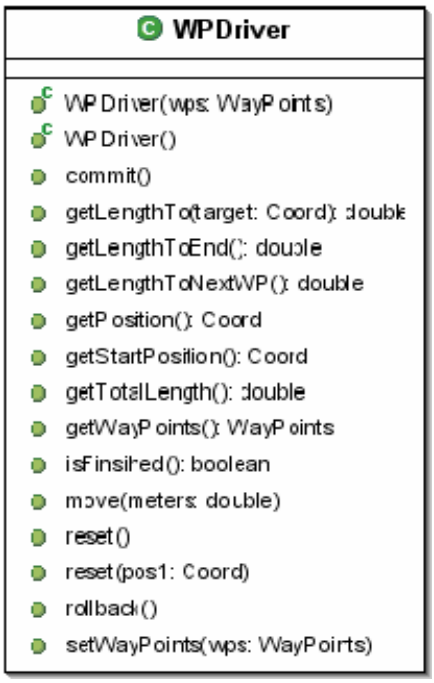
Klassen WPUtils er en hjelpeklasse med en rekke statiske metoder som forenkler oppgaver som å last inn lagrede veisegmenter og eventuelt bygge opp en rute av slike segmenter. Klassen inneholder også funksjonalitet for divsere ruteberegninger.



En viktig beregning denne klassen kan gjøre er å beregne hvor det nærmeste punktet på en rute er i forhold til en vilkårlig koordinat. Denne funksjonaliteten benyttes bla. av RouteStops-modellen til å finne ut hvor bussen må stopp for å betjene holdeplasser. Verdien som returneres er rutelengde, dvs hvor langt man må kjøre ruten for å komme til punktet. På den måten begrenser man posisjonsproblemet til et endimmensjonalt problem

Metoden som beregner dette tar i tillegg til en rutebeskrivelse, i form av WayPoints, også et lengdeargument som brukes som utgangspunkt for å finne nermeste punkt. Dette fordi noen strekninger kan kjøres flere ganger på samme rute.

5.5.6 WPDRIVER

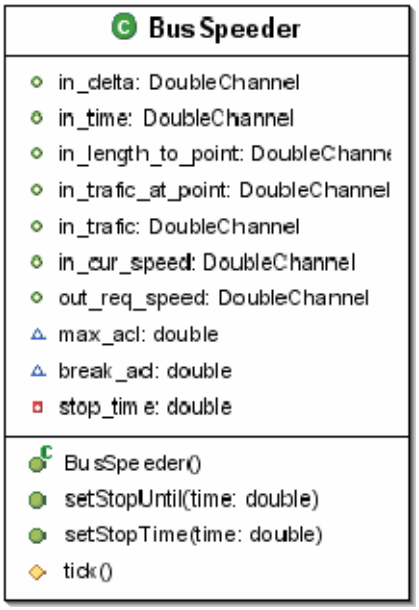


WPDriver er en klasse som brukes til å følge en rute i form av en serie veikoordinater. Klassen instansieres med en WayPoint-instans og så er det bare å bruke metoden move(meter) for å flytt seg langs ruten. For å finne ut hvor man er kan man bruke getPosition() metoden.

Modellen RouteDriver benyttes denne klassen for å finne ut hvor bussen skal kjøre. Når en rute er ferdig er det bare å laste inn en ny med metoden setWayPoints og fortsette.

Metodene commit() og rollback() benyttes for å gjøre klassen kompatibel som tilstandsdata for bruk av kontinuerlige modeller. Metodene er definert i grensesnittet ICommit som WPDriver implementerer og gjøre det mulig for klassen å følge iterasjonsprosessen i simulator. Ved nye iterasjoner kalles metoden rollback() som ruller tilbake forandringer i objektet siden siste tilstandssteg i simulator. Når simulering konvergere kalles commit() og utregnet tilstand blir da utgangspunkt for neste runde med iterasjoner.

5.5.7 BUSSPEEDER



BusSpeeder –modellen styrer fart til bussmodellen. Fart til buss er en sentral komponent i simulering av kollektivtrafikk da fart er det enste parameteret som i simulering og det virkelige liv som ikke er fast bestemt. Fart varierer i det virkelige liv med sjåfør, trafikkforhold, veiforhold, værforhold, tidspress og lignende. I denne simuleringen prøver vi etter beste evene å modellere dette.

Modellen bruker akselerasjon for å øke eller minke fart slik at lengde til for eksempel neste sving er nødvendig informajon for å finne ut når bussen må begynne å bremse for å holde seg innenfor gitte akselerasjonsgrenser.

Akselerasjon blir av modellen baregnet på bagrunn av signaler fra alle modeller i figur som som har piler inn til BusSpeeder, nemmelig nåværende fart fra busmodell, lengde til neste holdeplass fra

RoutStop-modellen, lengde til neste sving og anbefalt fart i sving samt lengde til neste-neste sving og anbefalt fart i denne fra RouteCurves-modellen, forsinkelse fra ScheduleDelayCalc -modellen og til slutt trafikkforhol fra både Intersection og Area – modellene.

Som vi ser benytter modellen lengdeinformasjon for beregne akselerasjon og dermed fart. For hver gang modellen kalkulerer en ny fart som skiller seg fra forrige bergning i forrige iterasjon med en viss margin må alle andre modellen oppdatere lengdeinformasjon da denne informasjon bygger direkte på hvilken fart bussen har. Når nye lengder er kalkulert må også BusSpeeder kalkulerer ny akselerasjon da akselerasjon er direkte avhenig av lengdeinformasjonen fra de andre modellen. Som vi ser har vi en sirkel av reaksjoner som til slutt konvergerer mot en fart slik at alle involverte modeller ligger i likevekt. Når denne likevekten er oppnådd fortsetter simulator med å bergne tilstand for neste tilstandssteg.

Akselerasjon, både gass og bremse-akselerasjon, og toppfart er i modellen normalfordelt rundt en middelverdi. Akkurat hvilke middel og varians -verdier som benyttes variere fra hvilket scenario som skal simmuleres og det gir derfor ingen mening å oppgi modellparametre i en generell modellgjennomgang. Dette må eventutel oppgis sammens med simulatorresultater. Man kan også lett bruke andre fordelinger. Tilfeldigheten i forskjellige fordelinger kan enkelt kontrolleres i sentral konfigurasjon. Modellen kan altså lett skifte mellom å være deteministisk og stokastisk, dvs. gi samme oppførsel i hver simmulering eller tilfeldig.

Hensikten med å gjøre sentrale verdier som toppfart og akselerasjon tilfeldig er å emulere forskjellige sjåførkaraktiristikker. Noen sjåfører er mer aggresive enn andre i trafikken og dette gjenspeiles i modellen med høyere akselerasjonsgrense evt høyere toppfart.

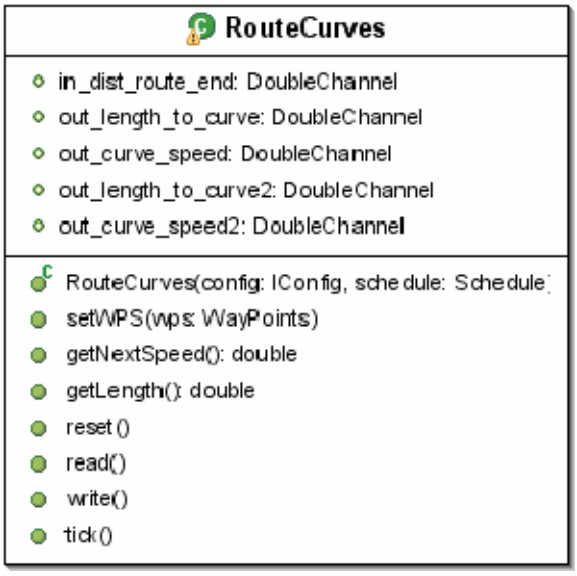
Det er viktig og huske busser i simulator ikke kjører samme rute om og om igjen. Hvis for eksempel.en sjåfør er blir simulert med ekstrem aggresivitet vil dette gi seg uslag på alle ruter den akutelle bussen vil kjøre i løpet av dagen. Man vil derfor se dette gi utslag på forskjellig ruter etter hvert som bussen, med den aktuelle bussjåføren, kjøre disse.

Lenge til punkt og anbefalt fart i punkt, hvor punkt er for eksempel en holdeplass, fra andre modeller kobles til kanalene *in_length_to_point* og *in_trafic_at_point*. Som vi ser benyttes begrepet trafic for å beskrive fart. Med trafic menes her trafikkflyt og betegnes med verdier fra 1 for normal flyt til 0 for full stopp. Grunnen til å bruke en slik ordning er for å holde konkrete verdier for fart i BusSpeeder og i andre modeller bare spesifisere hvordan modellen påvirker fart.

I tillegg til fartpunkter som beskrevet over benytter modellen fartsområder *vha. in_trafic* kanalen. Her sette relativ fart vha. trafic fra modeller som Intersection og Area. Disse modellene sjekker posisjon til buss og hvis buss er innenfor et spesifisert område vil modellen sette relativ vha. denne kanalen. Hvis flere modeller speifiserer fart vha av trafic i et bestmt område vil modellen multiplisere sammen verdiene og bruke resultat for å beregne akselerasjon og fart.

For å beregne akselerasjon sjekker modellen alle slik lengde til punkt og fart i punkt for å finne den kobinasjonen av lengde og fart som gir størst bremseakselerasjon. Hvis en kombinasjon overgår en fastsatt akselerasjonsgrenses vil modellen begynne å bremes slik at fart er som spesifisert innen buss har kjørt angitt lengde. Hvis ingen kombinasjoner av lengde fart trigger bremse vil buss akselerer for å nå toppfart multiplisert med trafic i området bussen er. Når målfart nesten er nådd vil bussen regulere seg inn på målfart vha. små akselerasjonskorrigeringer

5.5.8. ROUTECURVES



Denne modellen har til hensikt å modellere påvirkningen veiforhold har for fart til buss. Modellen påvirker BusSpeeder-modellen for å styre fart til buss. Modellen analyserer ruten som skal kjøres for svinger og hvor knappe disse er, hvor knappe svinger kjøres saktere enn slake svinger.

Modellen gir BusSpeeder informasjon om lengde og knappet til neste sving og neste neste sving vha. av kanalene out_length_to_curve, out_curve_speed, out_length_to_curve2 og out_curve_speed2. Grunnen til å fortelle BusSpeeder om to svinger etter hverandre er at først sving kan være slag etterfultg av en knapp kort lengde etter. For at BusSpeeder skal få muligheten til å bremse ned til den andre må modellen kunne se svingen.

-Noen andre modeller
RouteStops har til hensikt å modellere rutestop på bussholdeplasser langs ruten. Modellen bruker WPUtils som forklart tidligere samt koordinater for holdplasser til å bestemme hvor på ruten bussen skal stoppe. Som andre modeller som styrer fart bruker også denne modellen BusSpeeder for å oppnå dette hvor lengde til neste holdeplass kommuniseres.

Schedule er som nevnt tidligere en modell som holder orden på hvilke ruter bussen skal kjøre. Schedule er en diskrete modell, dvs. at den kun gjør noe idet simulator skifter tilstand. Modellen benytter seg av informasjon fra RouteDriver til å finne ut om nåværende rute er ferdigkjørt. Hvis så henter modellen informasjon fra konfigurasjonsrammeverket om neste rute som skal kjøres og når. Dette videreformidles bl.a til RouteStops og tilbake til RouteDriver.

Modellene Intersection og Area modellere trafikkforhold begrenset til et angitt geografiske områder. Are modellen brukes bl.a. til å fortelle simulator at trafikforhol i Trondheim sentrum er forskjellige fra trafikkforhold for eksempel opp ved Nardo. Modellene benytter polygodata for å beskrive områder og påvirkere da alle busser innenfor angitt område vha. BusSpeeder modellen og *in_trafic* kanalen.

Ara –modellen benyttes til å modellere trafikkforhold og trafikkutvikling for at avgrenset område i løpet av en dag. Dette benyttes bla.a til å modellere rushtrafikk om morgen og ettermiddag, og generelt at trafikk varierer i løpet av dagen. Trafikkforhold og utvikling for de forskjellige områdene er konfigurerbart vha konfigurasjonsrammeverk.

Intersection –modellen benyttes til å modellere lyskryss, eller andre store kryss, som påvirker trafikk periodisk ved at trafikk stopper. Slike kryss modelleres som områder rundt aktuelt kryss og påvirker bare busser som er innenfor angitt område. Kryssene konfigureres med en stoppe og kjøreperiode. Det tas ikke hensyn til kjøreretning eller at lyskryss ofte er oppsatt i bølger, dvs at grønt lys følger bilen.

PositioningTech er en modell for å emulere forskjellige posisjoneringsteknologier. Modellen tar korrekt posisjon fra buss og legger til en konfigurerbar størrelse normalfordelt støy. Denne støydata blir så overført til en register-modell hvor den lagres eller formidles direkte til system for beregning av ruteforsinkelser.

Tre forskjellige posisjoneringsteknologier modelleres, nemmelig GPS, GSM og manuell. GSP og GSM modelleres med standard avvik på heholdsvis 25 meter og 250 meter og oppdateringsfrekvens på 4 sek og 20 sek. Manuell posisjonering modelleres med 0 meter standard avik men oppdateres kun når buss forlater holdeplass.

6. SYSTEM

I dette prosjektet er det utviklet et system som med posisjon og rute -informasjon fra busser kan beregne forsinkelse og estimere ankomsttider for bussholdeplasser fram i tid. Systemet er i dette prosjektet koblet mot simulator bare via posisjonsdata fra bussene samt ruteinformasjon i konfigurasjon. Det er med andre ord ingen annen viktig informasjon som overføres som for eksempel at buss er ferdig med en rute. Dette må systemet finne ut av selv vha av posisjonsdata og rutedata alene. På den måten kan simulator på et senere tidspunkt byttes ut med virkelig posisjonsdata uten at systemet for beregning av ruteforsinkelser må oppdateres på noen måte.

6.1 FORSINKELSE OG ANKOMSTESTIMERING

For å beregne ankomsttider til bussholdeplasser fram i tid må først posisjon til buss som kjører den aktuelle ruten estimeres. Posisjon må estimeres, ikke beregnes, fordi teknologiene som brukes for å posisjonere har varierende nøyaktighet og varierende oppdateringsfrekvens.

Når posisjoner er estimert kan systemet på bakgrunn av en ruteplan beregne nåværende ruteforsinkelse og med bakgrunn i denne verdien beregne ankomsttider for holdeplasser frem i tid. Dette kan gjøres på forskjellige måter. Den enkleste måten er å ta utgangspunkt i ruteplan og legge til den estimerte forsinkelsen for alle fremtidige holdeplasser. Denne metoden kan gi tidspunkter som gjør at kollektivkunde i betydelig grad misser bussen da en forsinket buss antagelig vil prøve å kjøre inn en eventuell forsinkelse.

En lignende men bedre metode er å beregne med bakgrunn i historisk data for den aktuelle ruten, hvor mye forsinkelse det er mulig eller sannsynlig å kjøre inn fra der bussen er nå og til holdeplassene fremover. Med mulig menes her at man bruker den verste oppføring til å kalkulere innkjøringspotensial, dvs. den bussen i historien som har kjørt strekningen raskest. Med sannsynlig menes at man bruker data som i konfigurerbar grad flytter utgangspunkt for beregningen fra verste oppføring til oppføringer som bedre representerer gjennomsnittet. På den måten kan man med konfigurerbar grad av sikkerhet sikre seg mot situasjoner hvor kunder misser bussen pga. feil i estimater. Dette forutsetter at datagrunnlaget er stort nok. Økt nøyaktighet går på bekostning av økt antall kunder som misser bussen. Man kan for eksempel si at i 98% av tilfellene skal esitmatoren gi riktig eller tidligere estimat for ankomsttider til holdeplasser og på den måte ta bort de verste tilfellene fra beregningen.

I tilegg til metoden over kan en ankomstestimator utnytte at trafikken som skaper forsinkelser, ofte følger et fast mønster, for eksempel at trafikken er treger kl 0700 enn kl 0900 og at dette med stor sannsynlighet gjentar seg for hver ukedag. Man kan sikkert se andre trender som at forskjellige ukedager har forskjellige trafikkarakteristikker. Det samme gjelder sikkert for hva slag vær det er og hvilken sesong man er i.

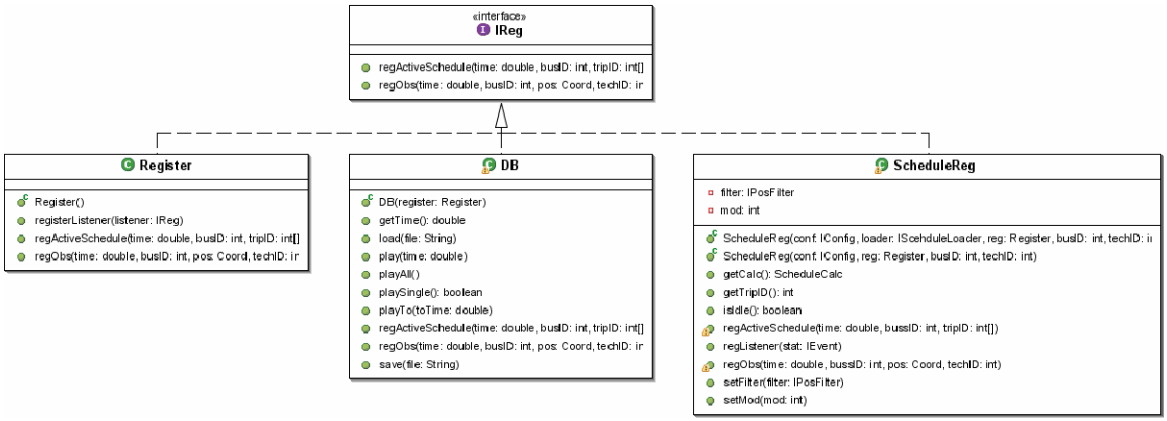
Ved å utnytte slike mønster kan man øke nøyaktigheten på estimatet ytterligere. I stedet for å finne nåværende forsinkelse og legge til maks innkjøringspotensial basert på rutedata fra tidligere gjennomkjøringer, kan man basere seg på rutedata som i tilegg er kjørt under tilsvarende forhold, som for eksempel tid på dag, hvilken ukedag det er og lignende.

Et siste punkt for å gjøre estimatoren enda bedre er å estimere nåværende forsinkelse til buss i forhold til historisk data i stedet for rutetabell for på den måten å finne forsinkelse ut ifra hvor bussen normalt er på det aktuelle tidspunktet og ikke hva som står i rutetabell. På den måten kan man få en bedre beskrivelse av virkelig forsinkelse. Denne forsinkelsen kan så konverteres tilbake til rutetabellverdi slik at kollektivkunde kan finne ut nå bussen kommer til bussholdeplass.

6.2 ARKITKETUR

Systemet for beregning av ruteforsinkelser er koblet mot kollektivsystem vha. grensesnittet *IReg*. I dette prosjektet er systemet koblet mot simulator, som er et virtuelt kollektivsystem. *IReg* definerer to metoder. Den første metoden *regActiveSchedule(time, busID, tripID)* benyttes til å registrere ruteplan for busser. Systemet må vite dette for å kunne beregne ankomsttider til holdeplasser hvor buss enda ikke har påbegynt rute. Parameteren busID representerer her et identifikasjonsnummer for hvilken buss det gjelder og tripID er her en referanse til en konfigurasjonsoppføring for hva slags rute det gjelder og planlagt starttidspunkt for denne. Slike oppføringer, dvs. ruter med tilhørende avgangstidspunkt, kalles en trip, eller tur om du vil. På den måten har systemet oversikt over busser og hvilke ruter disse skal kjøre i nærmeste framtid.

Etter at turene en buss skal kjøre er registrert må systemet oppdateres med posisjonsinformasjon om bussene slik at forsinkelser kan estimeres. Dette gjøres vha. metoden *regObs(time, busID, pos, techID)* som er definert i systemgrensesnittet *IReg*. Her er *time* tidspunkt for observasjon, *busID* hvilken buss det gjelder, *pos* er posisjonskoordinater og *techID* angir hvilken teknologi som er brukt for å posisjonere buss. Ved hjelp av slike observasjoner kan da systemet estimere med varierende grad av nøyaktighet hvor buss er, hvilken rute den kjører og hvor forsinket den der. Husk at hvilken rute en buss kjører ikke er direkte gitt, selv om systemet vet hvilke rute den skal kjøre i nærmeste framtid. For å vite hvilken rute som kjøres må systemet finne ut når en buss er ferdig men en rute og begynner med en annen uten annet å støtte seg på enn unøyaktig posisjonsdata.



Figur 6.1 – UML diagram systemarkitektur

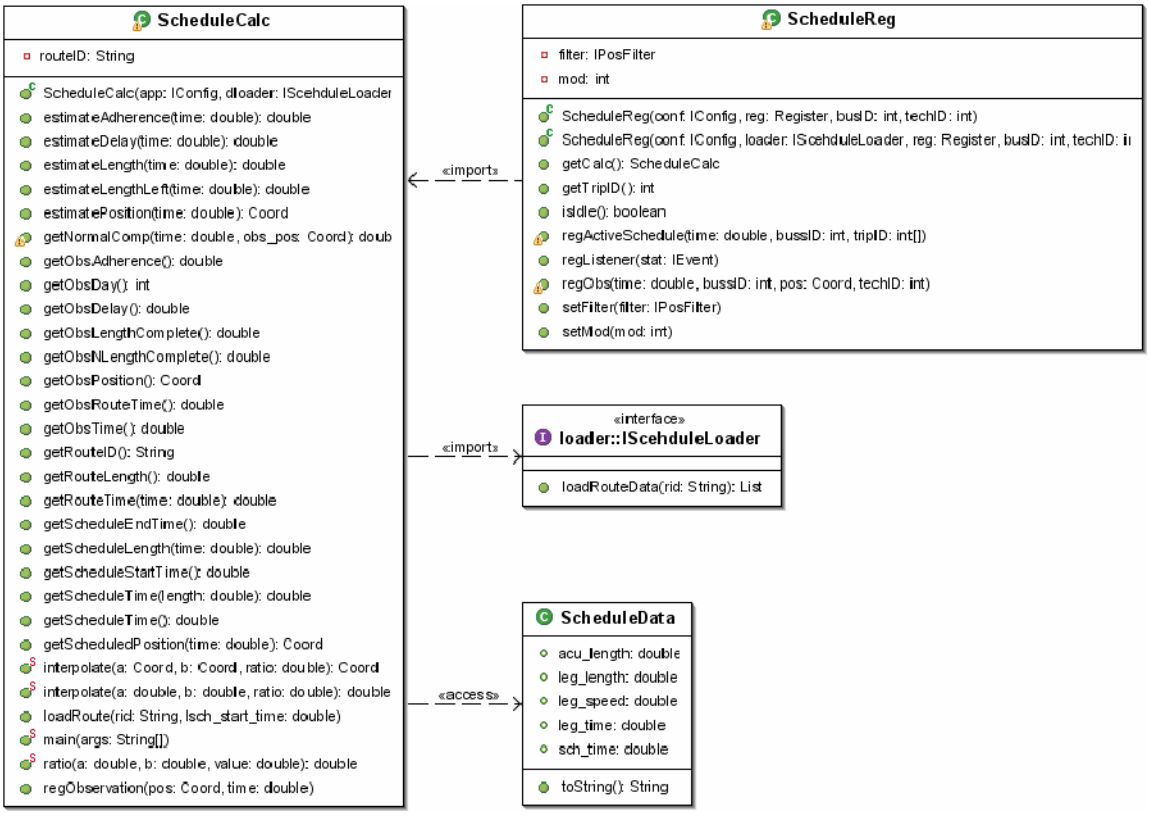
Klassen Register i figuren over er en videreformidlende klasse og er den klassen som instansieres og brukes til å koble sammen forsinkelsessystem med data fra kollektivsystem. I denne klassen kan andre klasser som implementerer grensesnittet *IReg* registrere seg via metoden *registerListener(listener)* og få tilgang til informasjons som sendes til system. Dette gjøres enkelt ved at informasjon som sendes til register ”kopieres” og videreformidles til alle registrerte lyttere.

En slik lytter er klassen *DB*. Denne klassen en hjelpeklasse som lagrer informasjon som sendes til system slik at informasjon kan spilles av igjen uten å simulere på nytt. Dette gjøres ved at men i slike tilfeller instansierer systemet uten å koble systemet til simulator og laster inn tidligere lagret informasjon til *DB* -klassen og kaller en av mange *play-metoder*. Da vil tidligere innspilte hendelser sendes til register-instansen som så videreformidler disse til andre lyttere. For de andre lytterne er det uvesentlig om data kommet fra simulator, virkelig system eller fra en tidligere innspilt logg.

6.2.1 POSISJONSREGISTRERING

Det opprettes og registreres i register en instans av *ScheduleReg* for hver aktive buss i systemet. Disse instansene har ansvaret for å overvåke vær sin buss og bruker klassen *ScheduleCalc* for å gjøre dette. *ScheduleCalc* kan vha. en ruteplan estimere posisjon til kollektivvogner og vha. observasjoner fortløpende korrigere posisjonsestimater. Klassen har også en god del andre metoder direkte knyttet til det å holde orden på hvor vogner er og hvor forsinket de er i forhold til ruteplan.

ScheduleReg har ansvaret for å videreformilde posisjonsdata fra buss til *SchedulCalc* samt holde orden på hvilke ruter som skal kjøres når. Når *ScheduleCalc* indikerer at rute er ferdigkjørt vil neste rute registrert i system initialiseres av *ScheduleReg*.



Figur 6.2 – UML diagram av ScheduleCalc og ScheduleReg

Når *ScheduleReg* får posisjonsinformasjon fra kollektivsystem blir dette videreformidlet til *ScheduleCalc* med metoden *regObservation(pos,time)*. Denne metoden vil da først estimere, basert på tidligere utregnet forsinkelse, hvor bussen er i rutekoordinater og bruke dette som utgangspunkt til å finne ny ruteposisjon. Rutekoordinater er kjørt lengde av total rutelengde. Grunnen til å benytte et utgangspunkt i beregning av nye rutekoordinater er at i noen ruter kan deler av rute kjøres to eller flere ganger, som for eksempel i en avstikker. I slike tilfeller er det umulig å bergene ruteposisjon ut ifra observert geografisk posisjon alene. Med et bra estimat som utgangspunkt blir dette enklere. Når nye rutekoordinater er funnet blir ny forsinkelse beregnet ut ifra ruteplan. Denne nye forsinkelsen blir så brukt som utgangspunkt for å estimere posisjon i neste registrering. Slik fortsetter systemet om og om igjen.

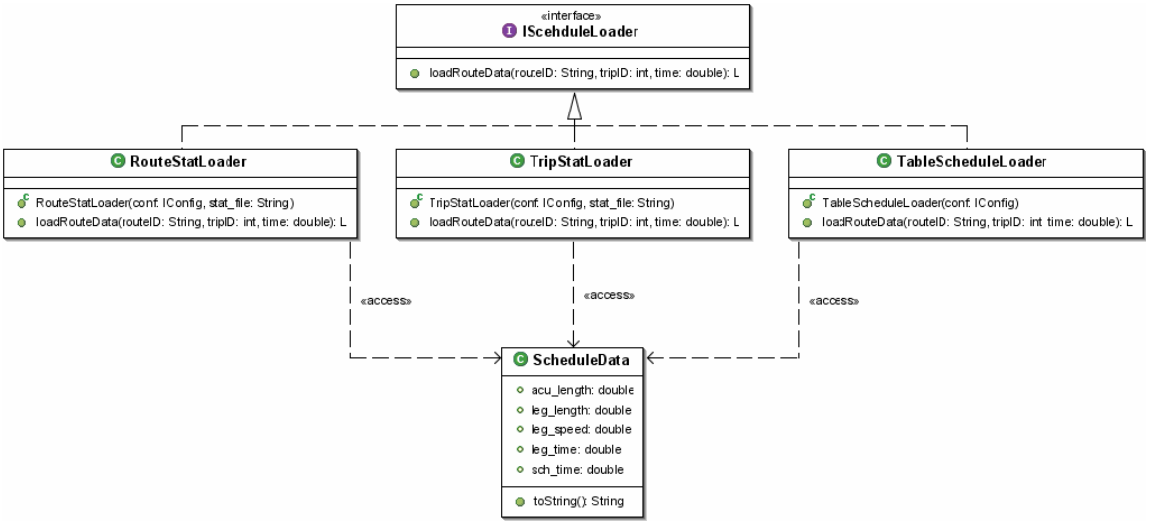
6.2.2 RUTEPLAN

ScheduleCalc benytter en ruteplan for å estimere posisjon til buss samt at forsinkelse til buss beregnes ut i fra denne. En ruteplan kan for eksempel være en vanlig rutetabell, eller man kan ta utgangspunkt i historisk data. Det er viktig og huske at selv om vi bruker et grunnlag for ruteplan enn den ordinære rutetabellen kollektivkunder kjenner kan forsinkelsesverdier lett konverteres tilbake til verdier for ordinær rutetabell.

Dette er et viktig konsept fordi systemet kan bruke historisk data og beregne forsinkelse i forhold til dette. Fordelen med det er at man bedre kan match oppførsel for på den måten å finne ut hvor mye bussen er forsinket i forhold til hva den normalt er gitt tilsvarende situasjon. Ved å beregne forsinkelses i forhold til en slik plan kan vi kompensere for normale forstyrrelser og holde forsinkelsesverdier noen lunde konstant.

Et eksempel kan gjøre det enklere å skjønne. Gitt at vi posisjonerer en buss i rushtrafikk. Et stykke ut i rute finner vi ut fra rutetabell at bussen, gitt observert posisjon, er 5 min forsinket. En tid senere registrerer vi 3 min forsinkelse. For bussen som registreres er trafikken normal for denne tiden på dagen. Til kunde vil man alt etter når man spør få to forskjellige forsinkelsesverdier selv om bussen holder normal hastighet i forhold til tidspunktet på dagen. Ved å lage ruteplaner basert på nettopp tid på dagen, kan man bedre kompensere for slikt. Da vil ruteplanen gi 0 min forsinkelse for begge observasjoner. Ved å relatere den nye ruteplanen som baseres på historiske data til ordinær ruteplan kan man få oppgitt verdier i forhold til rutetabell som kunde kjenner. Med en slik løsning vil kunde uansett når han undersøker forsinkelse få samme forsinkelsesverdi.

I dette prosjektet er det utviklet tre forskjellige ruteplaner som kan plugges inn i *ScheduleCalc* vha. grensesnittet *IScheduleLoader*. Disse er *TableScheduleLoader* som bruker vanlig tabelldata, *RouteStatLoader* som bygger ruteplan på rutehistorikk og til slutt *TripStatLoader* som bygger ruteplan på turhistorikk. Med turhistorikk menes historikk basert på samme rute og avgangstidpunkt som gjeldene tur.

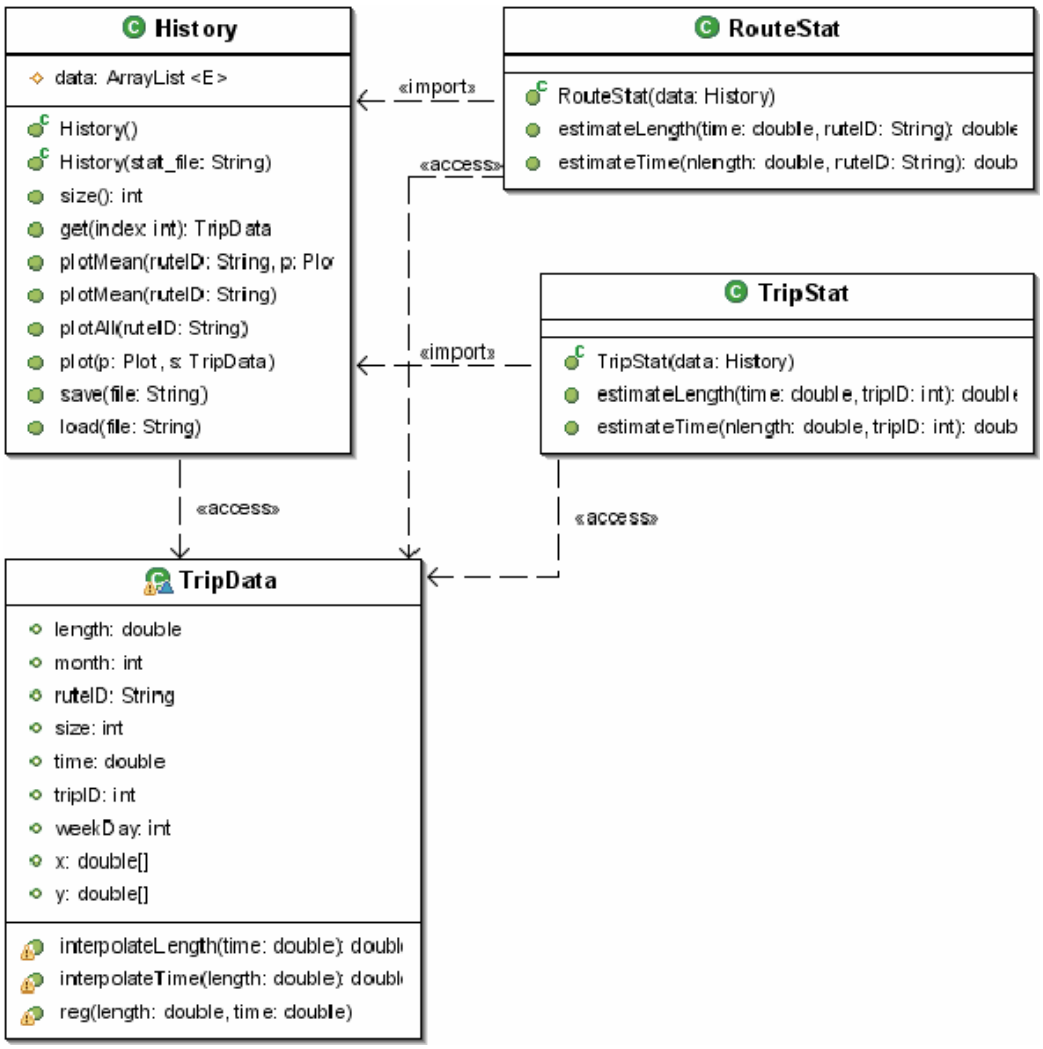


Figur 6.3 – UML diagram for forskjellige ruteplan generatorer

6.2.3 HISTORIKK

Rutehistorikk og turhistorikk, som brukes av *RouteStatLoader* og *TripStatLoader* genereres av klassene *RouteStat* og *TripStat*. Disse klassene går igjennom alle lagrede rutegjennomkjøringer vha. klassen *History* og sorterer ut de som passer kriterier som rutenummer for *RouteStat* og turnummer for *TripStat*. Når datagrunnlaget er filtrert blir et gjennomsnitt beregnet.

Metodene *estimateTime(length, ruteID/tripID)* gir en basert på filtrert historisk grunnlag en snittverdi for hvilken tid busser har gitt rutelengde. Det er denne metoden som benyttes for å bygge en ruteplan. Man starter med lengde 0 og noterer tid, øker så lengde men en gitt verdi og noterer tid på nytt. Man har da en ruteplan basert på filtrert historikk, hvor filter er samme rute, eller samme tur.



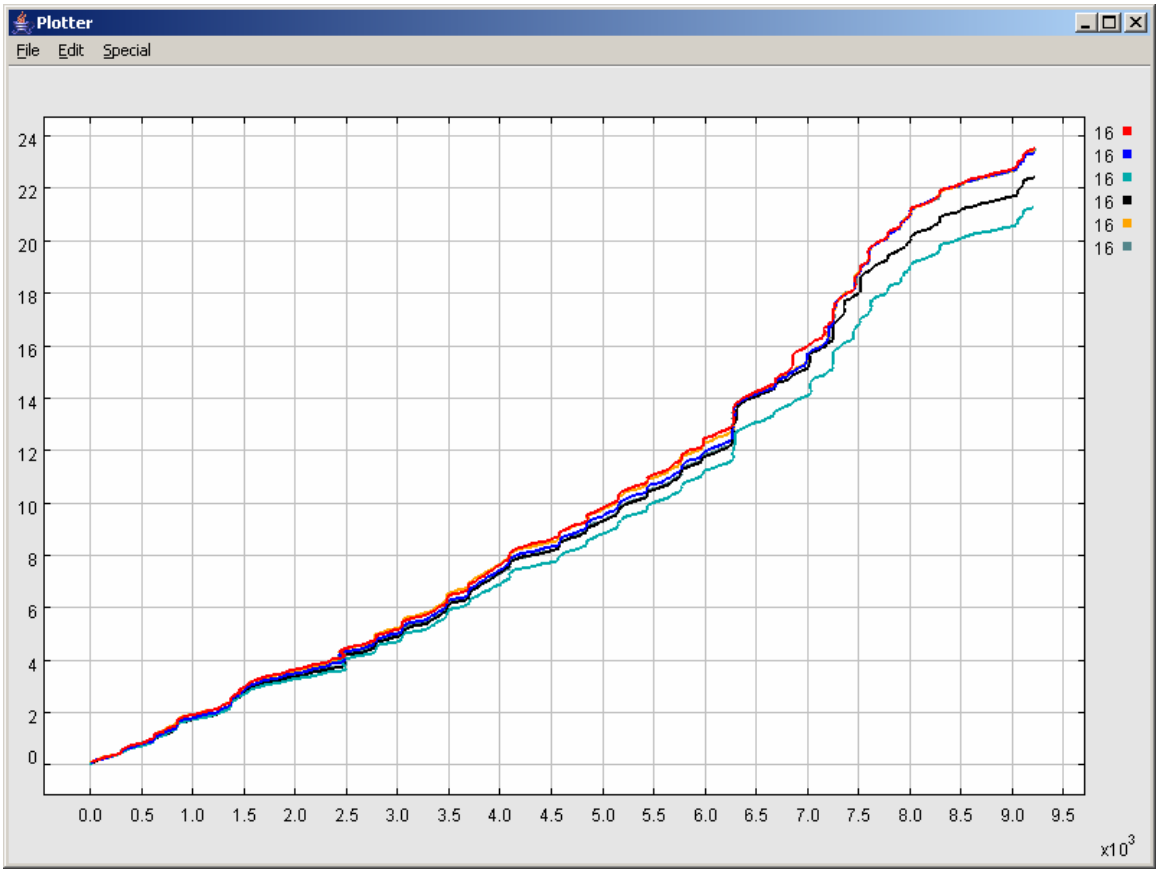
Figur 6.4 – UML diagram for selektiv historikk generatorer

6.3 POSISJONERING VED MYE STØY

Når posisjoneringsteknologier med mye støy benyttes blir observerte posisjoner filtrert før de registreres i *ScheduleCalc*. I dette prosjektet kan man velge mellom to forskjellige måter. Man kan konvertere observert posisjon til en ruteposisjon, dvs. rutedistanse, og filtrere på denne verdien eller man kan konvertere observert posisjon til en forsinkelsesverdi som den observerte posisjonen representere og filtrere på forsinkelse.

Fordelen med å filtrere på forsinkelse er at man utnytter kunnskap om rute fra tidligere gjennomkjøringer. Hvis man for eksempel benytter *TripStatLoader* vil man hvis bussen kjører som normalt ha 0 i forsinkelse hele ruten. Aggressive sjåfører vil gi seg utslag i en forsinkelse med konstant vekst, dvs. en forsinkelse som øker lineært med rutedistansen. Å filtrere bort forstyrrelser i slike enkle signaler er rimelig enkelt og man kan benytte ganske sterke filtre, dvs. filter som tar bort relativt mye støy.

Under er et tid-distanse – diagram for seks forskjellige rutegjennomkjøringer hvor rutene er kjørt til samme tid på døgnet men på forskjellige dager. Tid oppgis i minutter på y – akse og distanse i meter på x –akse. Forsinkelse beregnes som tid minus snitt-tid gitt observert rutedistanse. Som vi ser under vil forsinkelsen til de forskjellige turene være relativt lineære da man i subtraksjonen får fjernet alle disse riplene. Riplene kommer av fysiske ting som holdeplasser, kryss og lignende.

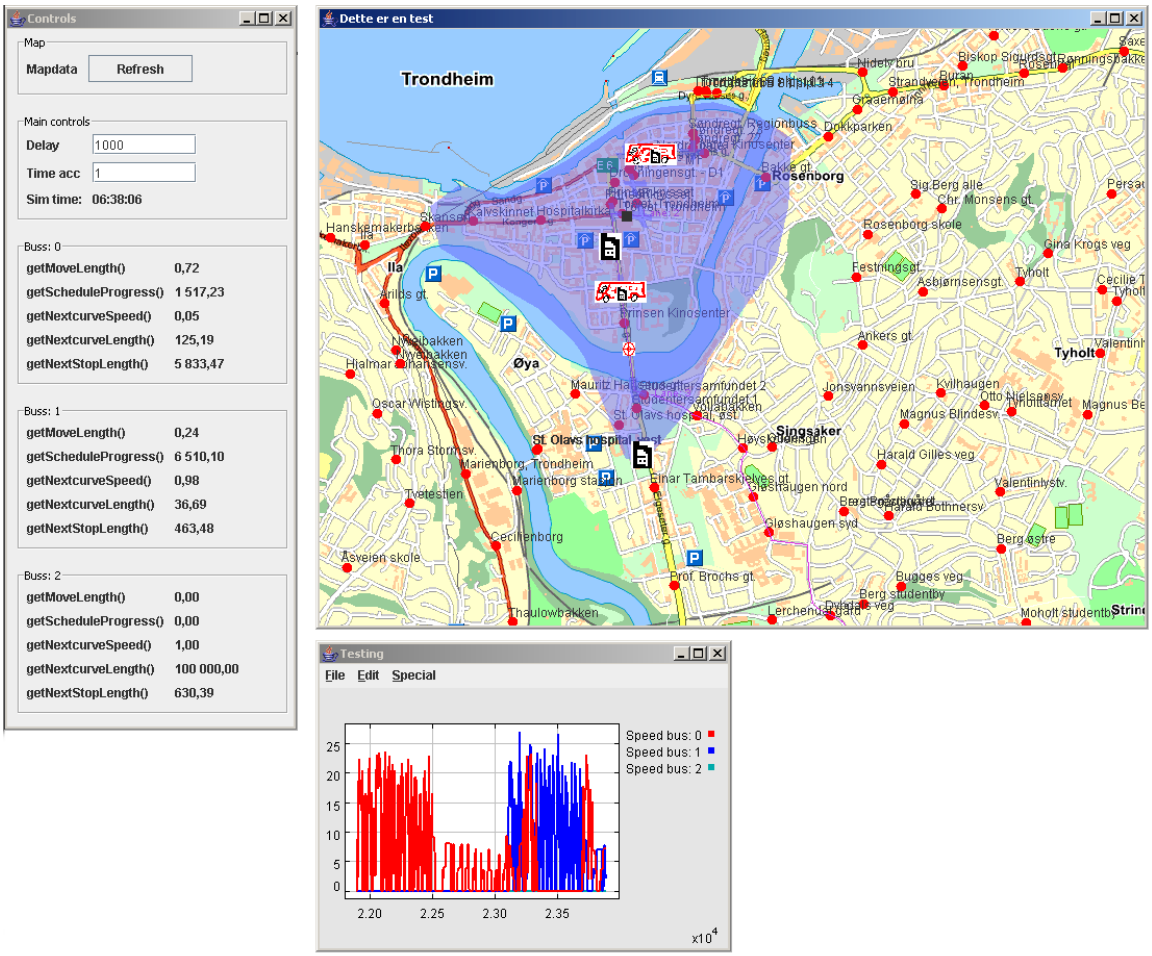


7. RESULTATER

I dette kapittelet presenteres et utvalg resultater av arbeid som er gjort i dette prosjektet. Dessverre er ikke alt ferdig utviklet slik at resultatdelen er noe amputert. Det gjelder spesielt resultater for beregning av ankomsttider.

7.1 SIMULATOR OG VERKTØY

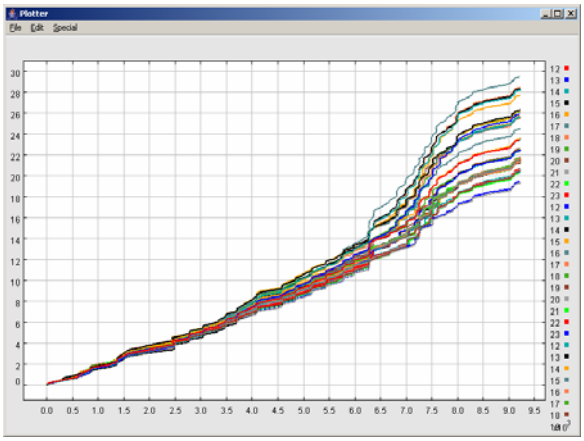
Simulatoren som er utviklet i dette prosjektet vises i aksjon i bildet under. Her simuleres et virutelt kollektivsystem. Til venstre i bildet vises et verktøy som brukes til å sample simulatorverdier. Brukergrensesnittet bygges opp automatisk basert på parametere som skal smaples. Diagrammet nederst viser far-tid-diagram for bussene i simulator. Midt i figuren vises kartsystemet med live data fra simulering. Her er bussene som kjører polottet med buss-symboler. GPS og GSM posisjonernigskoordinater er også tegnet inn med henholdvis liten og stor mobil-lignede symbol. Svart firkant markere neste holdeplass, rødt kryss markre hvor bussene skal være etter ruteplan og rødt spørsmålstegn indikere hvor systemet tror bussene er basert på GSM posisjoneringsteknologi.



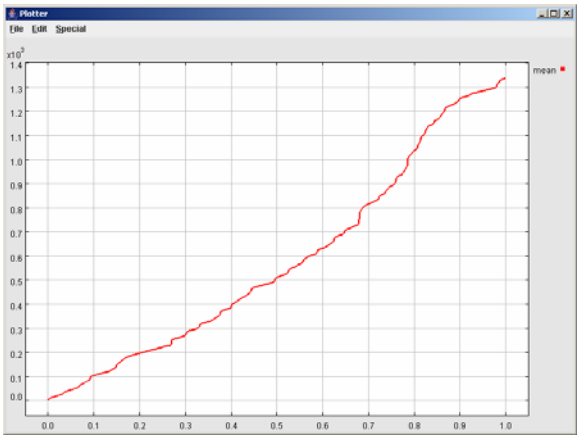
Figur 7.1 – Bilde av kartsystem, probesystem og fartsdiagram fra simulatorkjøring

7.2 MODELLER

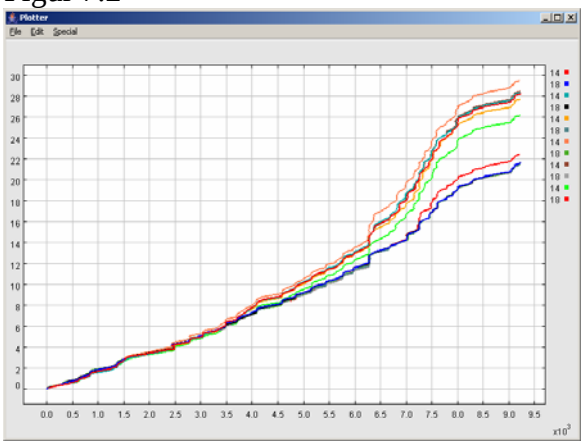
Under følger noe tid-distanse -diagrammer fra simulering. Figur 7.2 viser utskrift for rute 52 med retning mot sentrum seks dager etter hverandre. Som vi ser, påvirkes de forskjellige rutegjennomkjøringene forskjellig men hvor de påvirkes er mye likt.



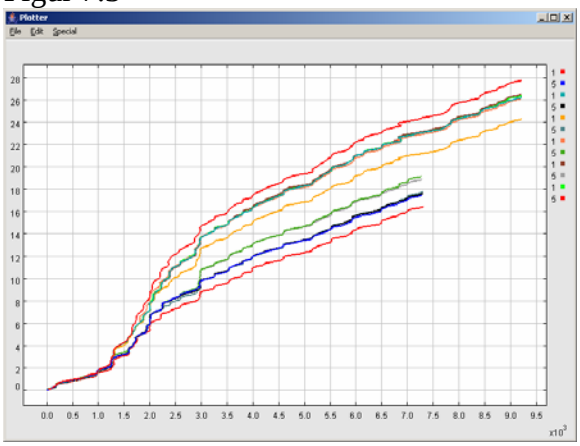
Figur 7.2



Figur 7.3



Figur 7.4



Figur 7.5

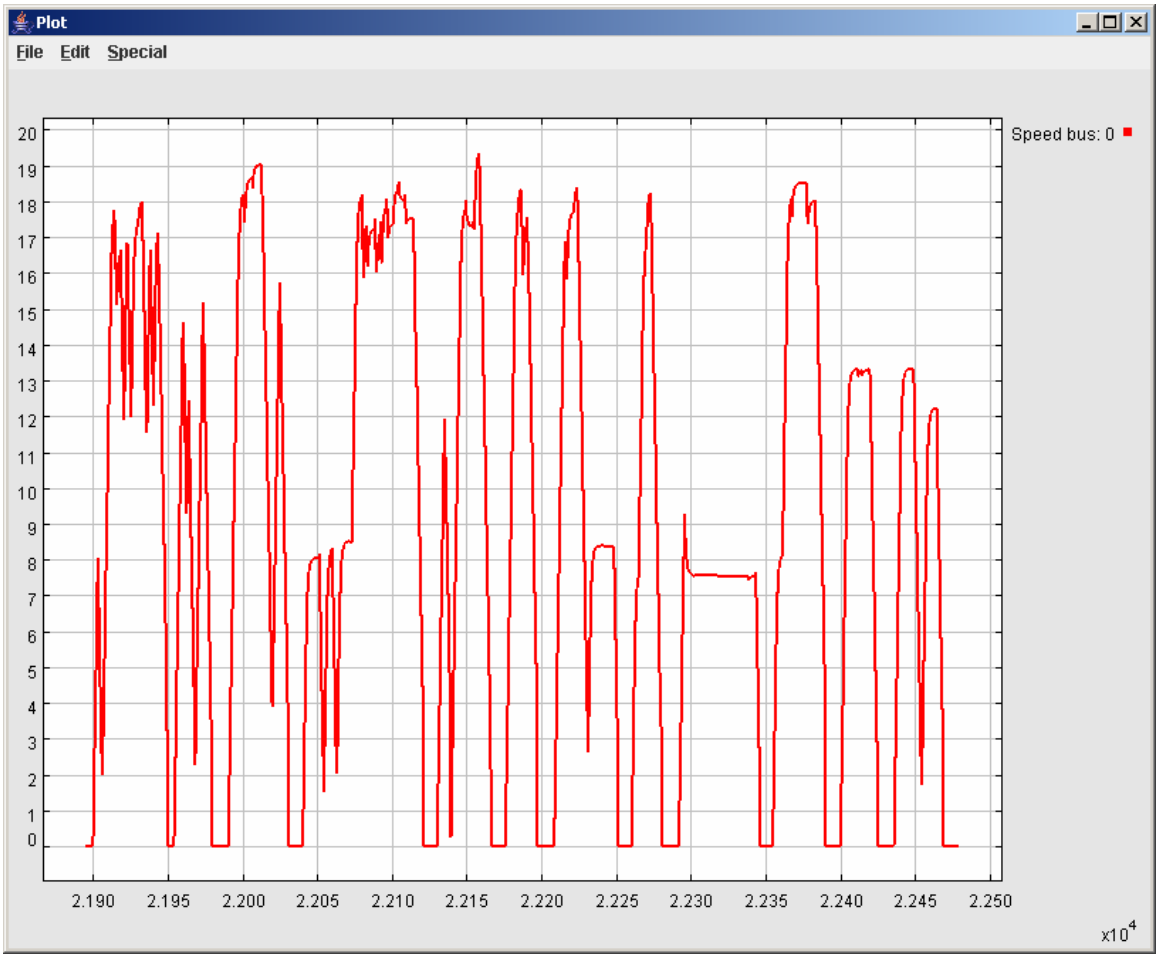
Figur 7.3 viser utskrift fra datagrunnlag for en *RoutStatLoader* for samme rute som beskrevet over. Data fra denne klassen representerer et snitt av data fra figur 7.2.

Figur 7.4 viser 12 forskjellige rutegjennomkjøringer hvor de seks øverste kjøres mellom kl 0630 og 0730 og de seks nederst kjøres 0930 til 1030. Diagrammet viser altså påvirkningen rushtrafikk har i simuleringen.

Figur 7.5 viser det samme som figur 7.4 men nå med motsatt retning, dvs. fra sentrum. Grunnen til at de 6 nederst turene er kortere er fordi de egentlig er en annen rute, som er kortere, men fra start til rundt 6 km er de helt like.

Figur 7.4 og 7.5 er ment å vise påvirkningen trafikk i Trondheim sentrum har på busser i simulator. Man kan se at forsinkelser her spesielt har stor påvirkning for ruter som start nært sentrum. Man kan også klart fra figur 7.4 og 7.5 se at rutetabeller passer dårlige for å fange opp effekter som rush-trafikk. Forsinkelsessystemet i dette prosjektet kan benytte rutedata fra tidligere gjennomkjøringer til å beregne ankomsttid. I figur 7.5 kan man se at man allerede 8 minutter før ankomst til holdeplasser på slutten av rute kan få gode estimater ved å utnytte kunnskap om tidligere gjennomkjøringer.

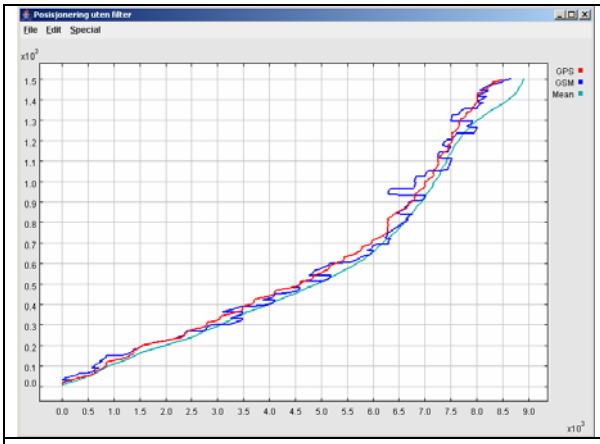
I figuren under vises et fart-tid-diagram for begynnelsen av rute 52 mot sentrum. Her kan vi se hvordan *BusSpeeder* –modellen bruker både bremse og gass –akselerasjon for å forandre fart på buss. Vi kan også se hvordan simulator simulerer typisk rutekjøring, med holdeplasser, fartstilpasning til svinger samt områder med trafikk.



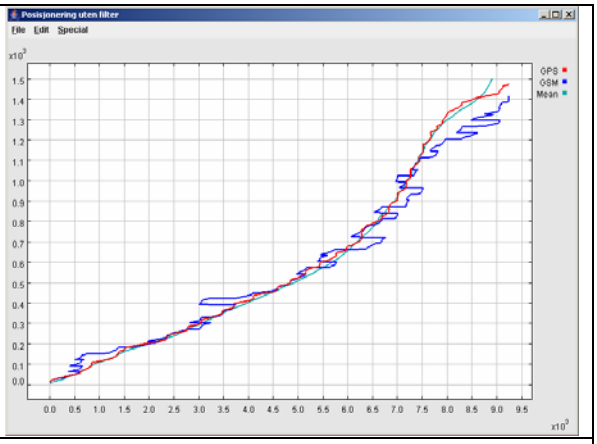
Figur 7.6 – Fart-tid diagram

7.3 POSISJONERING MED GPS OG GSM

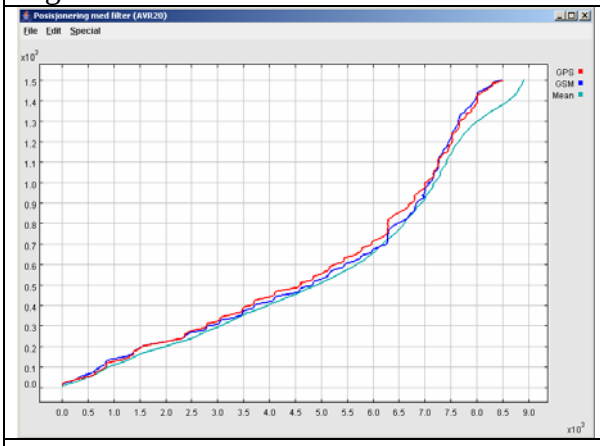
I figurene nedenfor vises noen eksempler på hva slags nøyaktighet man oppnår i simulator ved bruk av GPS og GSM posisjonering. I disse eksemplene konverteres observert posisjon til en forsinkelsesverdi som så filtreres og konverteres tilbake til ruteposisjon og registreres i *ScheduleCalc* klassen.



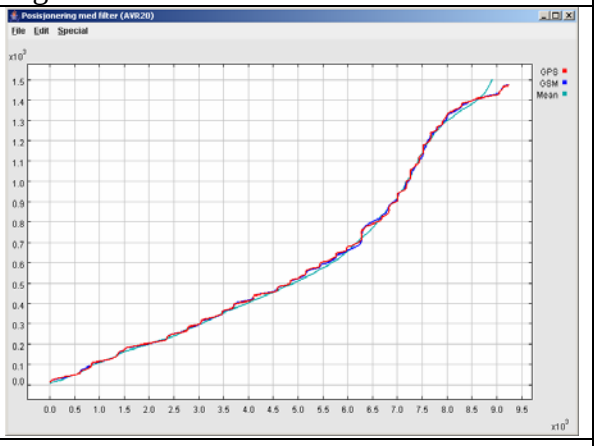
Figur 7.8



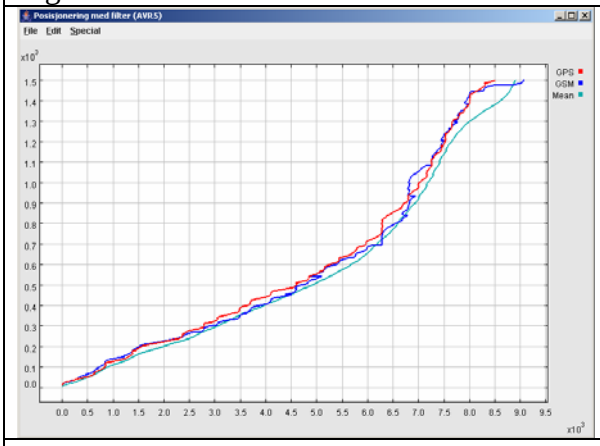
Figur 7.9



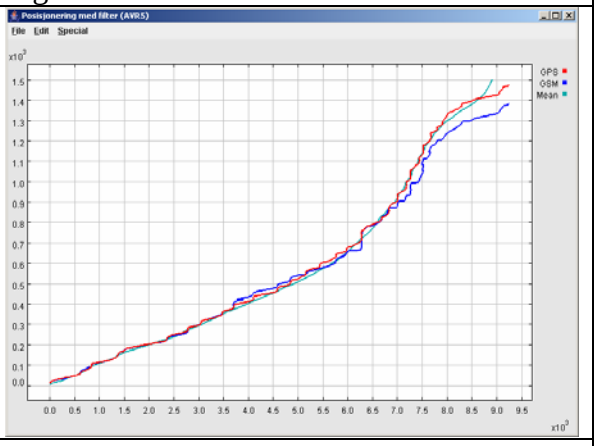
Figur 7.10



Figur 7.11



Figur 7.12



Figur 7.13

Kolonnene viser samme filter og samme bussrute, men to forskjellige turer. Radene viser posisjonsestimat med gitt filter.

GPS modellen i simulator er normalfordelt rundt virkelig bussposisjon med et standard avvik på 25 meter. GSM modellen er også normalfordelt rundt virkelig bussposisjon men med et standard avvik på hele 250 meter. Posisjonsdata oppdateres for hvert 4. sekund og hvert 20. sekund for henholdsvis GPS og GSM.

Øverste diagramrad viser posisjonsdata ufiltrert. Her ser vi hvordan unøyaktigheten til GSM posisjonering påvirkere forsinkelsesberegningen, gitt de forutsetninger som nevnes over. Vi ser også at unøyaktigheten kommer ekstra godt til syne i siste del av ruten, dvs. den del som går gjennom Trondheim sentrum. Dette fordi det er hastigheten her er lav, slik at unøyaktigheter utgjør ekstra mye i forhold til rutetid

I den midterste diagramraden er det brukt et midlerfilter på 20 punkter for å midle ut unøyaktigheter i forsinkelsesverdier for så å estimere ruteposisjon. Som vi ser ligger GSM verdier tett inntil GPS verdiene for begge dagene.

I nederst rad er et midlerfilter på 5 punkt brukt. Her ser vi at unøyaktigheter på slutten av ruten slipper litt igjennom.

8. DISKUSJON

Prosjektet, sett i ettertid, ble nok litt for ambisiøst og kunne med fordel vært delt opp i flere mindre samarbeidende prosjekt. For eksempel et prosjekt for å evaluere forskjellige posisjoneringsteknologier samt forskjellige måter å estimere forsinkelse og ankomsttider. Et annet prosjekt for å utvike simulator og modeller for virtuell kollektivtrafikk. Og et siste prosjekt for å utvike et komplett system, dvs. programvare og nødvendig utstyr, for beregning og presentasjon av ankomstinformasjon. Det at alt dette ble presset inn i et prosjekt har godt på bekostning av spesielt evalueringsdelen til å begynne med samt resultatdelen. For systemdelen av prosjektet er det hovedsakelig sluttbrukertjenester som mangler. Dette er ikke noen stor oppgave.

Utfordring ved utvikling av simulator, med tilhørende virtuell kollektivtrafikk, må ta mye av skylden for at for at deler av prosjektarbeid er noe begrenset. Den største utfordringen var fullstendig mangel på egnet litteratur for hvordan en simulator, for å løse generelle kontinuerlige problemer, bør designes. Mye tid ble pga. dette viet en utgave som senere måtte skrapes pga. arkitekturmessige begrensninger. I dette dragsuget ble også en god del modellarbeid berørt.

Med denne erfaringen kan man lure på om det å utvikle en egen simulator var en god løsning. Til dette mener jeg fortsatt ja. Som jeg skriver i rapporten kan man med en simulator og virtuell kollektivtrafikk eksperimenter mye enklere med for eksempel forskjellige eksotiske posisjoneringsteknologier, filtertyper, algoritmer for beregning av forsinkelse og ankomsttider, sluttbrukertjenester, operatørtjenester osv. Alt dette uten å bruke tid eller penger på å fysiske realisere de ulike teknologialternativene. Andre ting kan igjen være umulig, eller veldig vanskelig, å teste i et virkelig system da man ikke kan kontrollere miljøet i like stor grad som i en simulert utgave.

En stor fordel med å ha laget simulator og modeller for virtuell kollektivtrafikk er at det nå er veldig enkelt for andre interesserte å teste ulike løsninger som har med spesielt forskning på ny posisjoneringsteknologier og ankomstberegninger for kollektivindustrien. Modeller og system kan tilpasse nye krav etter brukes behov da det i dette prosjektet lagt mye energi i å gjøre deler av system lett utskiftbare vha. klart definerte grensesnitt og lignende. Man kan også bruke simulatorløsningen alene til å simulere andre typer kontinuerlige problemer da simulatoren ikke er knyttet til en spesiell type modeller. Med en ferdig simulator tilgjengelig er det i hvert fall ingen tvil om at det å modellere et kollektivsystem er fordelaktig og mer lønnsomt framfor å realisere, med hensikt å evaluere.

Et annet spørsmål er om det kanskje var unødvendig å lage en helt ny simulator og ikke heller bruke en som var ferdig utviklet. I jakten på en slik løsning fant jeg bare et reelt alternativ som ikke var knyttet til et bestemt domene av kontinuerlige problemer, som for eksempel analoge kretser og det var Simulink fra MathWorks. Problemet med Simulink er at løsningen er veldig komponentorientert og støtter dårlig bruk av programkodelogikk i modellbeskrivelse eller som integrerte støttetjenester. Dette er nødvendig for å effektivt modellere kollektivtrafikk da mange ting krever logikk som vanskelig kan bygges ved å sette samme basiskomponenter. Dette gjelder for eksempel bruk av konfigurasjonsdata til å sette opp busser med tilhørende ruter samt hvordan rutelogikken modelleres. Mange ting kan gjøres mye enklere med tilgang til et standard programutviklingsverktøy som Java og det er det simulator utviklet i dette prosjekt er tilrettelagt for å støtte.

Et problem med å evaluere ting med en simulator er at modeller i simulator også må verifiseres før man kan stole på resultater fra en simulering. Resultater blir ikke bedre enn kvaliteten på modellen som benyttes. Verifisering av modeller kan gjøres ved å sammenligne simulert respons med tilsvarende respons fra et virkelig system. Eller man kan som i dette prosjektet snu på flisa og konkludere med bakgrunn i oppførsel til simulert system, dvs. at konklusjon bare gjelder for virkelige system dersom virkelig system innehar de samme karakteristikker.

Modellene som er laget for å simulere kollektivsystem er utiklet for å emulere hva som antas for å være typisk oppførsel for kollektivtrafikk, dvs. det er ikke på noen måte prøvd å replikere et virkelig system, bare typisk oppførsel. I dette prosjektet har vi etter beste evne prøvd å modellere ting som vi mener i stor grad påvirker et system for beregning av ruteforsinkelser og ankomsttider for på den måten å best kunne verifisere at systemet virkelig virker. Noen egenskaper som vi mener er viktig og som modelleres er for eksempel at busser kjøres av forskjellige sjåfører med forskjellig aggressivitet, at buss akselererer for å skifte fart, at buss stopper på holdeplasser og er der i varierende tid, at trafikkforhold og lyskryss påvirker fart, at trafikkforhold generelt følger et mønster som for eksempel rush, at forsinkede busser prøver å kjøre inn forsinkelse og lignende. For komplett gjennomgang av hva som modelleres, se modellkapittelet.

En ting som antagelig ikke modelleres korrekt er GSM posisjoneringsteknologi. I den foreløpige modellen blir denne teknologien modellert vha posisjon til buss pluss en normalfordelt usikkerhet med standard avvik på 250 meter. Denne usikkerheten er garantert ikke helt normalfordelt men følger antagelig et mønster basert på hvor TBS'ene som måler posisjon er plassert, dvs. basestasjonen. Et annet punkt her er at GSM nevnes uten noen annen forklaring på hvilken type GSM posisjonering det er snakk om. Dette fordi det her menes generell posisjonering vha av GSM og da er tjenestetilbud fra nettoperatør i aktuelt område avgjørende. Man kan med stor sikkerhet anta at dette tilbudet minimum dekker CGI + TA type posisjonering som i de fleste tilfeller oppgis med en nøyaktighet på 500 meter eller mindre for 90 % av posisjoneringene i byer. Nyere utstyr av for samme teknologi klarer mye bedre nøyaktighet. Det finnes også andre posisjoneringsteknologier med bedre nøyaktighet. Vi mener at konklusjon ikke påvirkes av denne usikkerheten da vi i modell antar en ganske "unøyaktig" presisjon og at virkelig system vil være mer nøyaktig. Men det kan hende at et eventuelt mønster kan påvirke systemet. Dette må undersøkes videre i et eventuelt videre arbeid.

For å estimere posisjon, forsinkelse og ankomsttider benytter utviklet prototypsystem historisk data i beregningsarbeidet, dvs. logger fra tidligere rutegjennomkjøringer eller tidligere rutegjennomkjøringer på samme dagstidspunkt. Av simulatorresultater kan man se at sjåførkarakteristikker i nevneverdig grad påvirker ankomsttidspunkt og at dette kanskje kan utnyttes for å øke estimatkvaliteten. Aggressive sjåfører er, som simulert i simulator, ofte generelt aggressive dvs. at man ofte kan stole på at en sjåfør som er rask på en rutegjennomkjøring også er rask på neste. Dette kan undersøkes i eventuelt videre arbeid.

9. OPPSUMMERING OG KONKLUSJON

I dette prosjektet har vi arbeidet med å finne mulig løsninger for å beregne ruteforsinkelser til kollektivtrafikk og estimere fremtidige ankomsttider til bussholdeplasser. I dette arbeidet har vi bl.a. sett på forskjellige posisjoneringsteknologier for å posisjoner kollektivvogner. Vi har laget en simulator for å simulere generelle kontinuerlige modeller. Vi har utviklet modeller for å simulere et virtuelt kollektivsystem. Vi har utviklet en prototyp forsinkelsessystem. Vi har så testet forsinkelsessystem vha. simulert kollektivtrafikk og vurdert egenskaper til teknologier, algoritmer og filter.

9.1 GSM POSISJONERING

Posisjonering av kollektivvogner er nødvendig for å kunne esitmere ankomsttid med god nøyaktighet. Til dette formålet finnes det rikt utvalg teknologier med sine fordeler og ulempler. GPS er en selvskreven kandidat som er både relativt rimelig og nøyaktig for bruk i et forsinkelsessystem og brukes bl.a. til nettopp dette i dag i noen systemer.

GSM er en annen veldig spennede teknologi for posisjonering og slår GPS når det gjelder innvisteringskostnader og i det manuelle alternativet matcher driftskostnadene. Her benyttes en vanlig mobiltelfon for å posisjonere kollektivvog. Dette kan gjøres automatisk gjennom nettoperatør vha. teknologi som CGI + TA eller manuelt ved at bussjåføren selv oppdatere system vha. WAP og GPRS når estimert posisjon i system er forskjellig fra virkelig posisjon.

For at et posisjoneringssystem skal fungere må informasjon kommuniseres mellom system og vogner. Dette er ikke bare posisjonsdata men også administrasjonsdata som for eksempel det å spesifisere hvilke ruter som kjøres. For en GPS løsning må man installere radiomodem eller GPRS modem slik at posisjonsdata kan kommuniseres. I tilegg må man finne en løsning som muligjør kommunikasjon av ruteinformasjon til system. Dette kan gjøres vha. at sjåfør registrerer informasjon på en terminal i buss som er koblet til system eller for eksempel verbalt vha eksisterende radiosamband ved at noen sitter å oppdaterer system basert på verbal kommunikasjon med bussjåfører. Det siste alternativet kan være vanskelig å gjennomføre når mange busser skal kommunisere ruteinformasjon.

Med en GSM –løsningen får man alt dette i en rimelig enhet. Enheten kan kommunisere med system vha GPRS og sjåfør kan kommunisere direkte med system via grensenittene WAP, LCD og tastatur på mobiltelefon. Da dette er forbrukerteknologi er også prisen veldig rimelig. Man får for eksempel en Nokia 3120 som har all nødvendig teknologi til under 1000 kroner pr enhet.

I forhold til mål for prosjekt beskrevet innledningsvis er manuell posisjonering vha GPRS og WAP den løsningen som anbefales mest. Innvisterings og driftskostnader er ligger langt under noen andre teknologier og system kan oppgraderes til automatisk posisjonering hvis det senere oppdages at manuell posisjonering blir for tungvindt, og dette uten noen nye inviseringer da dette er en mulighet som eksistere for alle GSM mobiltelefoner. Ved at sjåfør kommuniserer direkte med system slipper man også ekstra administrasjonskostnader tilknyttet det få informasjon inn i system.

Foreløpige resultater fra simulasjon indikerer at manuell posisjonering vha WAP og GPRS gir veldig god nøyaktighet og at automatisk posisjonering vha. CGI + TA i de fleste tilfeller kan gjøres gode nok med riktig filter og hvis kunnskap om rute som kjøres utnyttes. Dette er resultater som bygger på ”worst-case” scenario. Det er grunn til å tro at nøyaktighet er bedre, eller i hvertfall vil med tid bli bedre, enn modelert i simulator. Så her har man alt å vinne.

Dessverre er prosjekt ikke helt ferdig og dette gjelder spesielt siste del som involverer forsinkelsessystem slik at resultater for dette mangler. Vi kan derfor ikke konkret fortelle hva slags nøyaktighet som kan oppnås med GSM og forsinkelsessystem sammenlignet med rutetabeller som benyttes i dag. Men det er ingen tvil ut fra de resultater som eksisterer at GSM posisjonering, både automatisk og manuell, i kombinasjon med et historikkbasert forsinkelsessytemet kan øke nøyaktigheten for avgangstider for bussholdeplasser betydelig i fohold til en vanlig rutetabell. Denne konklusjonen gjelder for simulert system.

Om konklusjonen over kan realateres til virkelig systemer er avhengig av om virkelig system ligner på simulert system. Simulert system er modellert ut ifra egenskaper virkelig kollektivtrafikk har som vi mener er viktige i forhold til posisjonering og estimering av ankomsttider. Selv om ikke verdier er er direkte sammenlignbare mener vi at oppførsel som er viktig for at et forsinkelsessystem skal virke kan sammenlignes. Dette er subjektiv mening og modeller må verifiseres for å få bedre innsikt i dette spørsmålet. Dette kan gjøres ved å teste ut deler av system i virkeligheten og sammenligne oppførsel.

9.2 GENERELT

Det er utviklet en simulator i dette prosjektet for å simulere kollektivtrafikk og på den måten teste ulike løsninger for posisjonering, ankomstberegninger, filtrering og lignende. Simulatoren kan simulere kontinuerlige eller diskrete og stokastiske eller determiniske modeller.

I tilegg til simulator er det utviklet en rekke verktøy for å støtte under modellering og simulering. Noen slike verktøy er et kartsystem for å visualisere geografisk data, konfigurasjonsrammeverk for modeller, probesystem for å visualisere simulatorparametre osv.

Det er utviklet en rekke modeller som etter beste evne prøver å emulere kollektivtrafikk. Modellene er utviklet med utgangspunkt i rute 52 hos Team Trafikk i Trondheim. Modellene emulerer egenskaper som akselerasjon, holdeplasser, trafikk og lignende.

Til slutt er det er utvikling av et komplett forsinkelsessystem påbegynt. Har utnyttes kunskap om rute i form av historisk data for på best mulig måte å estimere posisjon vha av unøyaktig posisjonsdata og finne forsinkelse. Systemet benytter også historisk data for å estimere hvordan forsinkelsen vil utvikle seg fremover for på den måte å kunn beregne gode estimater for ankomsttider til bussholdeplasser.

9.3 AVSLUTTENDE ORD

Prosjektet har vært interessant og utfordrende. En del problemer med utvikling av simulator har fått følger iform av et noe begrenset resultat-kapittel. Men måten å arbeide på, dvs det å bruke simuleringsverktøy for å evaluere løsninger, er absolutt veldig interessant. Dette er egentlig ikke noe nytt da simuleringsverktøy er i flittig bruk for eksempel innenfor elektronikkindustrien. Men det å bruke slike verktøy til å evaluere løsninger for kollektivsystemer er jeg fra før ikke kjent med.

10. VIDERE ARBEID

Under følger forslag til ideer man kan arbeide videre med. Dette er ideer jeg har plukket opp under arbeid men ikke har hatt tid til å forfølge eller rett og slett arbeid som ikke er ferdig.

Forsinkelsessystem må implementeres ferdig. Det som mangler er å få laget sluttbrukertjenesten som kalkulerer forsinkelsesverdier basert på historisk data tilbake til rutetabell-format og så presentere dette vha. en Web-løsning og en Wap-løsning. Når forsinkelsessystem er ferdig utviklet må konkrete resultater for nøyaktighet tjeneste i forhold til rutetabell kalkuleres.

En ny og bedre modell for GSM posisjonering kan også med fordel utvikles og testes. I dette prosjektet benyttes en bussposisjon pluss en normalfordel unøyaktighet til å simulere denne teknologien. Dette stemmer trolig dårlig med virkelig oppførsel.

Modelldata må verifiseres for å kunne si noe helt sikkert om ytelse til system. Dette kan gjøres ved å for eksempel realisere et forsinkelsessytem for et utvalg ruter med GPS posisjonering for maks nøyaktighet og se om det er noen effekter av stor viktighet for ytelse som ikke blir modellert.

CGI + TA posisjoneringsteknologi benytter bl.a. avstand mellom mobil og BTS for å bestemme posisjon. Det kan da være interessant bruke denne avstandsinformasjonen i kombinasjon med BTS-posisjon til manuelt å beregne vogn-posisjon ut ifra rute som kjøres. På den måten kan vi øke nøyaktigheten da vi effektivt begrenser mulige områder mobiltelefone er på. Det som må undersøkes er muligheter for å få tilgang til avstandsinformasjon og posisjonsinformasjons til BTS'en(e) som er brukt i posisjoneringen.

11. LITTERATURLISTE

Liste av artikler/bøker det henvises til i rapporten.

Tveit, Ørjan (2001) Posisjoneringssystem for kollektivtrafikk
Forprosjekt kollektivprioritering i Trondheim, SINTEF

Ramakrishna, (2006) Bus Travel Time Prediction Using GPS data
http://www.gisdevelopment.net/proceedings/mapindia/2006/student%20oral/mi06stu_84.htm

Lin, Hong-En, (2005) A Review of Travel-Time Prediction in Transport
http://www.easts.info/on-line/proceedings_05/1433.pdf

Patnaik, Jayakrishna (?) Estimation of Bus Arrival Times Using APC Data
http://www.nctr.usf.edu/jpt/pdf/JPT%207-1%20Chien.pdf

Chien, Steven (2003) Use of Neural Network to predict bus travel times
http://transportation.njit.edu/nctip/final_report/UseDynamicToPredictBusTimes.pdf

Andersson, Christorffer (?) Mobile Positioning – Where You Want to Be
http://www.wirelessdevnet.com/channels/lbs/features/mobilepositioning.html

Java J2SE – Utviklingsplattform, benyttes for utvikling og kjøring av system
http://java.sun.com/j2se/index.jsp