Universitetet i Oslo
Institutt for informatikk

En abstrakt maskin for Creol i Maude

Marte Arnestad

Hovedfagsoppgave

29. oktober 2003

Forord

Denne hovedoppgaven er en del av en candidates scientiarum-grad. Arbeidet er utført ved forskningsgruppen Presis modellering og analyse (PMA), Institutt for informatikk ved Universitetet i Oslo.

Først og fremst vil jeg takke veilederen min, Olaf Owe. Uten han hadde det ikke blitt noen oppgave. Han fortjener en stor takk for å ha tatt seg tid til å svare på alle mine spørsmål, for å ha gitt meg noen dytt i riktig retning, og for å ha hjulpet gjennom den vanskelige skriveprosessen. Einar Broch Johnsen fortjener også en takk for faglige inspill og korrekturlesing.

Videre vil jeg takke Per Øivind for moralsk støtte og oppmuntringer. Til slutt vil jeg takke de som har lest gjennom oppgaven og kommet med forslag til forbedringer, venner og familie.

Marte Arnestad
29. oktober 2003

Innhold

Kapittel 1

Innledning

Denne oppgaven inngår som et ledd i Creol-prosjektet ved PMA-gruppen ved Institutt for informatikk. Prosjektet går ut på å utvikle et rammeverk egnet for modellering av objektorienterte og distribuerte systemer.

Creol-prosjektet bygger på resultater fra prosjektet Adapt-FT (Adaptation of Formal Techniques to Support the Development of Open Distributed Systems), spesielt språket som ble utviklet kalt **OUN** (Oslo University Notation) [20, 22, 26]. OUN er et høynivå spesifikasjon- og designspråk med en formell semantikk, beregnet på åpne objektorienterte, distribuerte systemer. Det avsluttede prosjektet Adapt-FT hadde blant annet som mål å bygge en plattform, "the Integrator", som støtter formell utvikling av åpne distribuerte systemer. Innebygget i plattformen er (deler av) kodegenerering fra UML (Unified Modelling Language) [11] til OUN, oversettelse fra OUN til Java [14] og fra OUN til resonneringsverktøyet og språket PVS [24].

Creol (Concurrent Reflective Object Oriented Language) [15, 27] er et pågående språk- og forskningsprosjekt. Språket støtter "black-box" grensesnitt-spesifikasjon av parallelle objekter som kommuniserer ved hjelp av metodekall, samt implementasjon av slike objekter i form av imperative klasser der bruk av (nøstede) vakter muliggjør såvel asynkrone som synkrone kommunikasjonsformer, og veksling mellom aktiv og passiv (eller reaktiv) oppførsel. Objektene har hver sin dedikerte prosessor. Fokus er å undersøke programmeringskonstruksjoner som asynkrone kall og (nøstede) vakter, innenfor rammen av et objektorientert paradigme. Man ønsker å bygge en plattform implementert i modelleringsspråket Maude, for å eksperimentere med mekanismene i språket.

Maude [3, 28] er et språk basert på omskrivingslogikk. Det er et forskningsprosjekt startet på SRI (Stanford Research Institute) International, og utviklingen pågår nå flere steder rundt omkring i verden. Maude inne-

holder både lignings- og omskrivingslogikk. Ligningene antas å være konfluente og terminerende, og brukes til algebraiske spesifikasjoner og funksjonell programmering. Omskrivingsreglene representerer endring, man går fra en tilstand til en annen. Flere omskrivingsregler kan anvendes samtidig, noe som gir en naturlig modellering av parallellitet. Maude inneholder også en modell for objektorientering, beregnet på distribuerte systemer.

Det er ønskelig å bygge et verktøy for Creol. Det bør i første omgang bestå av en kompilator som utfører syntaktisk og semantisk sjekk og oversetter til en høynivå "maskinkode" CMC (Creol Machine Code), og en interpret som kan eksekvere CMC.

I denne oppgaven skal vi definere en **abstrakt maskin** for modelleringsspråket Creol med implementasjon i Maude. Den abstrakte maskinen består av

- *definisjon av CMC* - eksekverbar Creol-kode
- *interpreten* - kjernen som eksekverer koden, regler i Maude
- *datastruktur* - datatyper for alle elementene og hjelpefunksjoner

Vi har valgt å fokusere på grunnbyggestener i språket, spesielt vakter og sammensetningen av asynkrone og synkrone metodekall. Derfor har vi valgt å utelate automatisk oversettelse av Creol-kode til CMC, og vi anser dette som en naturlig jobb for den som utfører typesjekking. Av samme grunn fokuserer vi ikke på nedarving i forbindelse med subklasser og virtuell binding av metoder. Dette har ikke påvirkning på selve utførelsen av vakter og metodekall. Det vil ved nedarving også være behov for en oversikt over hvor metoden som er arvet ligger, noe som enklest gjøres ved oppbygging av strukturtre i forbindelse med semantisk sjekk.

1.1 Tema for oppgaven

Den generelle problemstillingen i denne oppgaven kan oppsummeres ved spørsmålet:

> Hvordan kan Maude brukes til å simulere et distribuert system med parallelle objekter, asynkrone kall og nøstede vakter, på en naturlig måte?

Creol er et språk under utvikling. Det er dermed et behov for utprøving av nytenkingen i språket; en enkel måte å undersøke om idéer som ser

gode ut på papiret vil fungere under kjøring. Vi er derfor ute etter et fleksibelt verktøy, som enkelt kan endres med endringene i semantikken.

Ved en interpret får vi et medium for å undersøke språkets attributter ved testing. Men vi får kun resultatet av én mulig kjøring. Creol skal også kunne bevise generelle resultater for alle mulige input. Maude har et innebygget søkeverktøy, search, som undersøker alle tilstander for en gitt starttillstand. Hvordan kan man best dra nytte av denne søkefunksjonen i sammenheng med Creol?

Med dette i tankene kan vi dele opp i to mer konkrete spørsmål:

Hvordan kan vi lage en interpret for Creol-programmer i Maude,
slik at vi fleksibelt kan eksperimentere med hensyn til semantikk?

Vi skal se på en konkret interpret laget i forbindelse med oppgaven, og dens fleksibilitet ved varianter av interpreten.

Hvilken nytte kan vi ha av at Maudes søkeverktøy
brukes på Creol-programmer?

For simulering ut over en enkelt kjøring kan det være ønskelig å bruke Maudes søkeverktøy. Hvor enkelt det er å bruke, og hvilke resultater vi kan oppnå ved bruk av dette verktøyet, skal vi se nærmere på i denne oppgaven.

1.2 Motivasjon

I 2002 ble hovedoppgaven "Kompilator fra OUN til Java" [25] ferdigstilt. Det ble der oppdaget flere svakheter ved å bruke Java som implementasjonspråk og oversatt kode. Et problematisk moment i forbindelse med Java som implementasjonsspråk var mengden av kode som måtte til. For å utvide eller gjøre endringer i kompilatoren må man sette seg inn i store mengder kode. Problematiske momenter i forbindelse med Java som oversatt kode var vanskelighetene for å få til parallellitet mellom objektene, og begrensningen ved et stort antall tråder (threads). Hvis metodene ikke er serialiserte kan de aksessere variable samtidig, men hvis de er serialiserte må man vente aktivt på retur fra kallet. Dette vanskeliggjør OUNs (og Creols) intensjon om å kunne utføre andre oppgaver mens man venter på at en vakt (guard) skal bli sann. Føringene Java, ved å bruke den som oversatt kode, la på OUN var også problematiske. For eksempel ble multippel arv vanskelig, og muligheten for å arve fra mer enn én superklasse ble dermed utelatt.

I Maude trenger vi ikke å tenke på parallellitet mellom objektene siden dette er implisitt gitt. Det er et høynivåspråk så koden ligger nærmere opptil en operasjonell semantikk. Dette gjør koden mer oversiktlig, og vi får sagt mye på få linjer. Koden blir derfor enklere å endre, enten det er for å implementere nye varianter av semantikken eller syntaksen, eller for å utvide den abstrakte maskinen. Siden utførelsen av setninger i Creol gjøres ved regler i Maude, og ikke av Maude-maskinen selv, vil måten Maude utfører sin kode på legge minimale føringer for Creol. Maude er ennå ikke implementert med bruk av ekte parallellitet, så foreløpig begrenses bruken av den abstrakte maskinen til modellering og testing.

1.3 Kort oversikt over oppgaven

I kapittel 2 presenterer vi bakgrunnstoff for oppgaven, deriblant de viktigste begrepene for Creol og Maude.

I kapittel 3 ser vi en semantisk beskrivelse av den abstrakte maskinen. Vi beskriver strukturen og gir en operasjonell semantikk for Creol. På bakgrunn av dette ser vi nærmere på definisjonen av CMC og implementasjonen av den abstrakte maskinen i Maude i kapittel 4.

I kapittel 5 gir vi flere Creol-eksempler, ser på oversettelse til CMC, eksekvering i den abstrakte maskinen og søk ved Maudes søkefunksjon search.

Kapittel 6 vil beskrive varianter av den operasjonelle semantikken, og vi vil se eksekvering av eksempler fra kapittel 5 i disse.

I kapittel 7 oppsummerer vi de viktigste resultatene i oppgaven.

1.4 Konkrete resultater fra arbeidet med oppgaven

Som vi vil se i kapittel 3, 4 og 6 har oppgaven resultert i en operasjonell semantikk for det relevante subsettet av Creol, en definisjon av CMC og en kjørende interpret i forskjellige varianter for CMC. Den fullstendige koden for implementasjonen i kapittel 4 finnes som vedlegg.

Oppgaven bygger opp under en artikkel til Norsk Informatikkonferanse 2003. Artikkelen er lagt ved som vedlegg.

Kapittel 2

Bakgrunn

Vi skal i dette kapittelet se nærmere på språkene Creol og Maude. De viktigste begrepene for begge språkene vil bli gjennomgått. Det er en fordel om leseren har en generell interesse og grunnleggende forståelse av programmeringspråk. Noe kunnskap om logikk vil lette lesingen av definisjonene i delkapittelet om Maude, men er ingen forutsetning.

2.1 Creol

Creol (Concurrent Reflective Object Oriented Language) er et språk under utvikling ved Universitetet i Oslo. Basisidéene bak språket er hentet fra OUN (Oslo University Notation) [20, 22]. Vi skal her gi definisjoner basert på OUNs grammatikk [23].

Creol er et formelt fundert modelleringsspråk laget for å kunne utvikle objektorienterte, åpne distribuerte systemer, der man kan håndtere sikkerhetsaspekter. Det er ment å være en intuitiv notasjon slik at også andre enn "matematikk-eksperter" skal kunne benytte seg av et formelt system. Notasjonen er inspirert av andre objektorienterte språk som Simula [7], Java [14] og Corba [1]. Språket inneholder velkjente objektorienterte begreper som klasser og grensesnitt, begge med separate arvehierarkier. En klasse kan i tillegg implementere et antall grensesnitt.

Et åpent distribuert system kan være resultatet av mange delprogrammer, utviklet på ulike steder og tider. Creol støtter derfor at deler av programmet kan bli kompilert og lagt til et kjørende system [21]. Man kan legge til nye (sub)-klasser, (sub)-grensesnitt, og nye metoder i en klasse. De nye kodedelene kan være avhengig av de gamle ved arv. De kan også implementere gamle og nye grensesnitt. Den gamle koden kan bli gjort avhengig av den nye gjennom nye grensesnitt, og klasseutvidelser.

Vi skal se på Creols grammatikk ved en utvidet BNF hvor

- terminalsymboler er **uthevet**,
- metasymboler står i *kursiv*,
- symboler antatt forstått intuitivt eller beskrevet uformelt står med vanlig tekst,
- valgfrie deler er omsluttet av firkantede klammer [ ],
- repetisjoner ved krøllparenteser {},
- opphøyet i * for null eller flere ganger,
- opphøyet i + for en eller flere ganger.

Da vil [*noe*] være valgfritt å ha med én gang, og {*noe*}* kan gjentas null eller flere ganger. Til informasjon står := for tilordning, = for likhetssammenligning og == mellom metodesignaturen og implementasjonen.

2.1.1 Grensesnitt

Creol-grensesnitt definerer operasjoner som skal støttes i klasser som implementerer grensesnittet. Hvem som kan kalle på metodene defineres ved **with** *grensesnittnavn*. Restriksjonen gis ved at kalleren må ha implementert det angitte grensesnittet. På grunn av denne restriksjonen er det også meningsfylt å ha tomme grensesnitt, for å opprette en kobling mellom to objekter, der kun det ene skal tilby metoder. Hvert grensesnitt kan kun ha én eksplisitt **with**-tilknytting, men på grunn av arv kan den implisitt ha andre **with**-tilknytninger. Restriksjonen er satt for å gjøre grupperingen av metoder mer oversiktlig. Hvis grensesnittet angitt etter **with** er **any**, supergrensesnittet for alle grensesnitt, kan alle objekter kalle metodene.

Definisjon 1 *Grensesnittdeklarasjon*:

interface grensesnittnavn [**[**type-parametre**]**] [**(**objekt-parametre**)**]
[**inherits** grensesnittnavn$^+$]
begin
[**types** type-deklarasjoner]
[**with** grensesnittnavn
{ **op** *metodesignatur* }$^+$]
[semantisk spesifikasjon]
end

Sammen med deklarasjoner av typer kan vi ha tilhørende funksjoner. Det er ennå uavklart om disse skal plasseres her eller i en egen modul, i et underliggende språk.

Den semantiske spesifikasjonen kan inneholde en invariant angitt ved **inv**, en antagelse angitt ved **asm**, og hjelpefunksjoner angitt ved **func** med tilhørende deklarasjoner og ligninger. Invarianten vil være et predikat på kommunikasjonenshistorien, det vil si en sekvens av de observerte hendelser som objektet har tatt del i. Her settes det krav til hvilke oppførsler objektet som implementerer grensesnittet kan ha. Antagelsen er også et predikat på historien, som beskriver oppførselen eksterne objekter må ha for at invarianten skal kunne garanteres.

2.1.2 Klasser

En klasse definerer hvordan objekter skal se ut, og hva de skal inneholde. De inneholder dermed typede variable, implementasjon av metoder og kan også inneholde en konstruktør, *Init*, for å lage en passende initialtilstand, og en startmetode *Run*, for å gjøre objektinstanser av klassen aktive.

Definisjon 2 *Klassedeklarasjon*:

class klassenavn [**[**type-parametre**]**] [**(**parametre**)**]
 [**implements** grensesnittnavn$^+$]
 [**inherits** klassenavn$^+$]
begin
  [**var** variabeldeklarasjoner]
  [**op** init == *imperativ kode* .]
  [**op** run == *imperativ kode* .]
  {**op** *metodedeklarasjon* }*
 {**with** grensesnittnavn
  {**op** *metodedeklarasjon* }$^+$
 }*
  [semantisk spesifikasjon]
end

Metodedeklarasjoner uten **with** foran seg kan kun kalles av objektet selv. Variabeldeklarasjoner gjort over vil senere bli henvist til som objektvariable.

I Creol tillates multippel arv både for klasser og grensesnitt. For grensesnitt er det uproblematisk å ha multippel arv, fordi vi ser på grensesnittene som en union, og det ikke spiller noen rolle hvor signaturen ble opprettet først. For subklasser kan det derimot oppstå navnekonflikter. Ved kall på en metode som ikke unikt kan identifiseres

ved navn og parametertyper finnes det regler for hvilken som skal velges. Vi ser først etter metoden på nærmeste nivå. Hvis det finnes to metoder på samme nivå benyttes den "eldste". For ytterlige detaljer kan leseren se i [21].

2.1.3 Metoder

Metodene i Creol angis ved **op**. I grensesnittene er det kun signaturer som deklareres.

Definisjon 3 *Metodesignatur*:

metodenavn **(** [innparameter: type {, innparameter: type}*]
[**out** utparameter: type { , utparameter: type}*] **)**

I klassene har vi en full deklarasjon, det vil si at metodene inneholder imperativ kode og lokale variable. Metoden kan kalles av objektet selv, eller andre objekter av riktig grensesnitt. Metodene har ingen funksjonsverdi i seg selv, men kan returnere verdier til kallstedet ved hjelp av utparametre.

Definisjon 4 *Metodedeklarasjon*:

metodesignatur == *imperativ kode* **.**

Ved avslutting av metoden vil man tilordne verdier til utparametrene, og returnere disse til kallstedet.

2.1.4 Imperativ kode

Vi skal nå definere imperativ kode:

Definisjon 5 *imperativ kode*
{ [**var** variabelnavn: type {, variabelnavn: type}*]
setning {; setning}* }$^{+}$

Vi kaller en slik setningen eller sammensetningen av dem for program-setning. Ved snakk om et Creol-uttrykk mener vi en sammensetning av typer eller funksjoner med boolske eller aritmetiske operatorer.

Siden notasjonen i Creol er inspirert av andre objektorienterte språk ligner også den imperative koden på språk som Java og Corba. To elementer skiller seg derimot ut; metodekall og vakter. De skal vi se nærmere på her.

Metodekall

I Creol finnes både synkrone og asynkrone kall. Ved synkrone kall venter det kallende objektet aktivt på avslutning av kallet. Med aktiv venting menes at objektet ikke får utføre noen andre oppgaver før returen har ankommet. Dette er ikke gunstig i distribuerte systemer, fordi det fører til dårlig utnyttelse av prosessorkraft. Hvis avslutningen aldri kommer får objektet heller ikke fortsatt eksekveringen, selv om det har andre oppgaver som kunne vært utført uavhengig av metodekallet. Derfor er det også innført asynkrone kall. Objektet kan da fortsette normal eksekvering til returen fra kallet er nødvendig. Dette kan minne om programmering med fremtidsvariable [2, 6, 29, 30]. Man får da utført en programsetning mer, men vil allikevel måtte vente aktivt når returen behøves. Ved bruk av vakt-testing av returen kan man utføre andre oppgaver til returen ankommer. Hvis den aldri kommer vil ikke objektet låses, bare instansen av metoden som gjorde kallet. Bruk av time-out kan hjelpe på dette, men denne mekanismen skal vi ikke se på her. (Den kan oppnås som en mulig forfining av ventevakten, hvor *wait* har med et antall sekunder den skal vente som parameter.)

Siden et objekt har mulighet til å ha flere utestående kall, uten at disse behøver å være nøstet, trenger man en måte å gjenkjenne den spesifikke returen på. Dette gjøres ved etiketter. Når et kall instansieres opprettes en unik etikettverdi for dette kallet. Den lagres i de lokal variablene til metoden som utførte kallet, i en variabel som normalt ikke er aksesserbar for metoden. Dermed kan flere instanser av samme metode ha utestående kall samtidig, uten at disse kan blandes. Etikettverdien tas vare på av metoden som blir kalt, og sendes ved returen.

Definisjon 6 Synkrone og asynkrone kall:

- m(e;x) lokalt synkront kall
- l!m(e) lokalt asynkront kall med etikett
- !m(e) lokalt asynkront kall uten etikett
- l?(x) metoderetur
- o.m(e;x) synkront objektkall
- l!o.m(e) asynkront objektkall med etikett
- !o.m(e) asynkront objektkall uten etikett

hvor m er metodenavnet, e og x henholdsvis inn- og utparametre, l er et etikettnavn og o en objektpeker.

Når et asynkront kall er uten etikett får vi ingen returverdi tilbake. Synkrone kall har ikke etikettnavn, siden vi uansett venter aktivt på retur i dette tilfellet.

Meotderetur kan opptre i to forskjellige sammenhenger, enten som vakt som beskrives nærmere under, eller som selvstendig setning. Hvis vi har setningene $l!o.m(e)\ ;\ l?(x);\ldots$ vil vi vente aktivt på returen, som om det var et synkront objektkall. Derfor vil også synkrone objektkall oversettes til setningene over i CMC, se delkapittel 4.1.4. For lokale kall blir ikke effekten det samme. Hvis vi venter aktivt på en retur fra et asynkront lokalt kall, direkte etter at kallet er gjort, vil objektet aldri få mulighet til å utføre kallet. Dermed vil objektet gå i vranglås. Dette kan sees som en svakhet ved språkets syntaks, men vi ønsker å opprettholde muligheten for å vente aktivt, og vil derfor ikke forby $l?(x);\ldots$. Sammensetning som fører til vranglås burde oppdages av den semantiske analysen, som en feil eller advarsel. Da ligger ansvaret for å bruke konstruksjonen riktig på programmereren.

Vakter

Vakter ble først introdusert av Dijkstra i [8] for å representere ikke-deterministiske valg. I Creol brukes de først og fremst til å styre prosesskontrollen i en parallell setting, noe som igjen bidrar til at rekkefølgen metodene blir eksekvert i kan bli ikke-deterministisk. Vi angir en vakt ved g, kode ved C, og kode med vakt først ved GC.

Definisjon 7 Vi har tre forskjellige vakter, av typen Guard, definert induktivt, og i tillegg en sammensetning av de tre:

- *wait* $\in$ Guard (eksplisitt avløsningspunkt)
- $l\,?(x) \in$ Guard, hvor $l \in$ Label, og $x \in$ Var
- $\phi \in$ Guard, hvor ϕ er et boolsk uttrykk over lokale og objekt-variabler
- $g_1\ \&\ g_2\ \&\cdots\&\ g_n \in$ Guard hvor $g_i \in$ Guard for $1 \leq i \leq n$

For å få en bedre prosess-styring har vi vakten *wait*. Den vil evalueres til false i aktiv kode, og brukes for å eksplisitt styre objektet til å bytte prosess. Når objektet igjen er ledig vil den resterende programsetningen bli hentet tilbake, og *wait* vil være endret til true.

Metodereturvakten evalueres til true dersom avslutningen på metode-kallet har ankommet. Utparametrene x får verdiene fra avslutningen, før koden etter vakten eksekveres.

Bruk av vakter

Vaktene kan være nøstet inni hverandre, enten som $g_1 \longrightarrow C_1 \; ; \; g_2 \longrightarrow C_2$ eller $g_1 \longrightarrow g_2 \longrightarrow C_1 \; ; \; C_2$ der en og en evalueres om gangen. Hvis betingelsen foran første pil er sann fortsetter metoden, hvis ikke settes kodebiten til vent. På den måten kan vakter brukes til eksplisitte prosessor-avløsningspunkter. Man kan også uttrykke ikke-determinisme mellom to kodebiter ved $GC_1 \square \, GC_2$ hvor GC (guarded commands) er en kodebit med vakt. Da vil enten C_1 eller C_2 velges avhengig av hvilken vakt som er sann. Vi kan også kombinere kode på en kvasi-parallell måte angitt ved $GC_1 || \, GC_2$. Dette kan sees som syntaktisk sukker for $GC_1 \; ; \; GC_2 \square \, GC_2 \; ; \; GC_1$.

Sammensetning med & gir en liste av vakter hvor alle skal være sanne samtidig. Man vil i praksis ikke trenge mer enn én boolsk vakt, fordi disse kan slås sammen, og én ventevakt av samme grunn. Metodereturvakter kan ikke slås sammen, men hvis en melding har ankommet kan vi fjerne vakten, og sette tilordningene av returverdiene bak listen av vakter. Fjerning av returvakter kan forsvares ved det faktum at når en melding har ankommet vil returvakten ikke bli usann igjen, siden etiketten er unik. Vi kan sammenligne denne sammensetningen med $\phi \longrightarrow l \; ?(x) \longrightarrow \ldots$. Forskjellen er at ved den første sammensetningen kreves at begge vaktene er sanne samtidig, mens ved det siste alternativet kan ϕ være falsk igjen når $l \; ?(x)$ omsider kan evalueres til sann.

Vi kan også se for oss en sammensetning hvor en av vaktene skal være sann. Det finnes på dette tidspunktet ikke noe slikt symbol i Creol, men vi kan simulere effekten av $g_1 \vee g_2 \longrightarrow C$ ved ikke-determinisme: $g_1 \longrightarrow C \; \square \; g_2 \longrightarrow C$. Hvis C er en lang kodesekvens kan man skrive: $(g_1 \longrightarrow skip \; \square \; g_2 \longrightarrow skip) \; ; \; C$.

2.1.5 Objekter

Objekter opprettes i Creol med **new** klassenavn(parameterliste), hvor parameterlisten svarer til den i klassedeklarasjonen. Parametrene tilordnes direkte til variablene i klassedeklarasjonen, som i Simula. Alle objektene kjører i parallell, og har sin interne aktivitet. Det vil si at objektene prosesserer sine egne metoder, også hvis de er kalt av andre objekter. Ved opprettelse vil et objekt starte med å eksekvere metoden *Init*, som har til hensikt å initialisere variable. Etter det vil *Run*metoden, hvis den finnes, bli eksekvert. *Run*metoden indikerer aktiv oppførsel, mens de andre metodene indikerer passiv, eller reaktiv oppførsel i objektet [15]. Vi kan veksle mellom aktiv og passiv oppførsel ved vakter i *Run*.

Som vi skal se, er kommunikasjonen mellom objektene i den abstrakte maskinen asynkron. Hvert objekt har derfor en kø av kall. Objektet tar seg av ett og ett kall om gangen. For å oppnå veksling mellom aktiv og passiv oppførsel har vi avløsningspunkter. Hvis man benytter seg av *wait* -vakten blir det et obligatorisk avløsningspunkt, koden som eksekveres for øyeblikket suspenderes, og gir rom for eksekvering av en annen metode. Andre vakter kan også opptre som avløsningspunkter dersom de evalueres til `false`. På denne måten kan eksekveringen og fordelingen av prosessorkraft bli mer effektiv.

2.1.6 Eksempel - Read/Write

Her har vi et eksempel på et enkelt Creol-program for å se språket brukt i praksis. Eksempelet vi skal se på er objekter som kontrollerer lese- og skriveaksess til en felles ressurs, hentet fra [22] og [20].

Grensesnitt

Grensesnittet *Read* skal kontrollere leseaksess. Siden samtidig leseaksess er uproblematisk er det ingen restriksjoner.

interface Read [T: Data-Type]
begin
  with any
    op read(**out** d : T)
end

Grensesnittet *Write* skal kontrollere skriveaksess. Vi har her tre metoder, for å åpne for skriving, skrive, og lukke igjen. Her har vi en restriksjon i form av en invariant **inv**, og vi forutsetter at omgivelsene oppfører seg som i antagelsen **asm**.

interface Write [T: Data-Type]
begin
  with any
    op open-write
    op write(d : T)
    op close-write
  asm $\mathcal{H}$ **prs** ($\leftrightarrow$.open-write $\leftrightarrow$.write* $\leftrightarrow$.close-write)*
  inv ($\mathcal{H}$ / $\leftarrow$) **prs**(open-write write* close-write)*
end

Her står $\mathcal{H}$ for kommunikasjonshistorien, hvor symbolet $\rightarrow$ representerer mengder med metodekall, og $\leftarrow$ mengder med metoderetur. Når vi,

som i antagelsen over, ønsker begge i sekvens skriver vi ↔. I invarianten over ser vi det er snakk om $\mathcal{H} / \leftarrow$. Det står for projeksjonen av $\mathcal{H}$ med hensyn på alle metodereturhendelser. Operatoren **prs** står for "prefix of regular expression", og brukes for å beskrive kommunikasjonsmønsteret i historien $\mathcal{H}$. Objektet som implementerer grensesnittet *Write* har kun lov til å svare på en forespørsel av open-write og en forespørsel av close-write om gangen. Antagelsen sier at man utfører "ulovlig" bruk av *Write* hvis man kaller write uten først å ha kalt open-write.

Grensesnittet *Read-Write* skal kontrollere både lese- og skriveaksessen. Hvis lesing er lov uavhengig av skriving kan "skitne les" forekomme. Det vil si at det leses én verdi rett før den skal oppdateres. Hvis leseren planlegger å gjøre beregninger på verdien, for så å oppdatere kan dette få uante følger. Vi må derfor ha metoder for å begrense lesetilgangen.

interface Read-Write [T: Data-Type]
 inherits Write, Read
begin
  with any
    op open-read
    op close-read
  asm $\mathcal{H}$ **prs** (↔.open-write ↔.write* ↔.close-write
          | ↔.open-read ↔.read* ↔.close-read)*
  inv #($\mathcal{H} / \leftarrow$.open-read) - #($\mathcal{H} / \leftarrow$.close-read) = 0
      ∨ #($\mathcal{H} / \leftarrow$.open-write) - # ($\mathcal{H} / \leftarrow$.close-write) = 0
end

Invarianten sikrer her, sammen med antagelsen om rekkefølge på kallene, at ingen setter en lås uten så å låse den opp igjen. Antagelsen her inkluderer antagelsen i *Write* og sikrer, sammen med den arvede invarianten fra Write, at det ikke leses og skrives samtidig og at skrivingen først skjer etter låsing av tilgangen.

Klasser

Vi skal nå se på klasser som implementerer grensesnittene.

Klassen *Read-Control* skal returnere data av type T til objekter av vilkårlig type. Dataene er lagret i variabelen shared-data.

class Read-Control [T: Data-Type]
  implements Read [T]
begin
  var shared-data : T
  with any

```
    op read(out d : T) == true  ⟶ d := shared-data .
end
```

Klassen *Write-Control* skal sørge for at kun en får skrive om gangen. Her har vi en variabel flag som er sann hvis skrivetilgangen er ledig. Variabelen shared-data er den det skrives til. Antagelsen og invarianten er akkurat de samme som i grensesnittet, men gjentas for å kunne bevise at invarianten fra grensesnittet holder. Ved arv av klasser må det sies eksplisitt hvis samme invariant skal holde.

```
class Write-Control [T: Data-Type]
  implements Write [T]
begin
  var shared-data : T,
      flag : bool
  op init == flag := true .
  with any
    op open-write == flag  ⟶ flag := false .
    op write(d : T) == true  ⟶ shared-data := d .
    op close-write == true  ⟶ flag := true .
    asm ℋ prs (open-write write* close-write)*
  inv (ℋ / ←) prs(open-write write* close-write)*
end
```

Klassen *Read-Write-Control* skal sørge for sikker lesing og skriving samtidig. Vi har da behov for å vite hvor mange som leser samtidig, og at ingen gjør det når noen skal skrive. Derfor innføres variabelen nb-readers, hvor nb står for number.

```
class Read-Write-Control [T: Data-Type]
  implements Write [T], Read-Write [T]
  inherits Read-Control, Write-Control
begin
  var nb-readers: int
  op init == nb-readers := 0 .
  with any
    op open-read == (flag ∨ nb-readers ≠ 0) ⟶ flag := false ;
       nb-readers := nb-readers + 1 .
    op close-read == true  ⟶ nb-readers - 1;
       if nb-readers = 0 then flag := true fi .
  asm ℋ prs (open-write write* close-write
          | open-read read* close-read)*
```

inv #($\mathcal{H}$ / ←.open-read) - #($\mathcal{H}$ / ←.close-read) = 0
∨ #($\mathcal{H}$ / ←.open-write) - # ($\mathcal{H}$ / ←.close-write) = 0
∧ #($\mathcal{H}$ / ←.open-write) - # ($\mathcal{H}$ / ←.close-write) = 0
∨ #($\mathcal{H}$ / ←.open-write) - # ($\mathcal{H}$ / ←.close-write) = 1
end

Vi ser at antagelsen er den samme som i grensesnittet. Invarianten er utvidet slik at vi på et gitt tidspunkt har maksimalt en skriver.

2.2 Maude

Maude [3] er et deklarativt, høynivåspråk basert på omskrivingslogikk. Det støtter både likhets- og omskrivingslogiske beregninger. Den funksjonelle delen av Maude er utviklet fra språket OBJ [12]. Omskrivingslogikken gjør det mulig å representere ikke-deterministiske, parallelle beregninger. Maude har også en måte å representere objekter og meldinger på. Vi skal nå se nærmere på funksjonell Maude, omskrivingslogikk og objektorientert Maude. Definisjonene som gis er basert på [3, 17, 31].

2.2.1 Funksjonell Maude

En funksjonell modul (**fmod**) spesifiserer en fler-sortet[1] likhetsspesifikasjon. Den består av en signatur og en mengde ligninger.

Definisjon 8 Signatur
En fler-sortet signatur (S, Σ) består av et sett S av sorter og en familie $\{\Sigma_{w,s} | w \in S^*, s \in S\}$ av funksjonsymboler. Det vil si at $\Sigma_{w,s}$ er et sett av funksjonsymboler med aritet w og verdi av sort s. Vi skriver gjerne f : w → s ∈ Σ for f ∈ $\Sigma_{w,s}$. Hvis w er tom blir f ofte kalt en konstant av sort s.

Sorter opprettes ved nøkkelordet **sort** i Maude. Deklarasjoner av funksjonsymbol skrives **op** f : w -> s .

Ligninger reduserer termer, så før vi kan definere ligninger skal vi definere grunntermer og generelle termer med variable.

Definisjon 9 Grunntermer
Gitt en fler-sortet signatur (S, Σ) kan vi definere et sett av grunntermer med sort S, $\mathcal{T}_\Sigma = \{\mathcal{T}_{\Sigma,s} | s \in S\}$ induktivt definert ved betingelsene:

[1] Fler-sortet betyr i denne sammenhengen at man kan ha flere sorter, altså typer, og må ikke forveksles med sortering.

1. $\Sigma_{\epsilon,s} \subseteq \mathcal{T}_{\Sigma,s}$, det vil si at hver konstant av sort s er en grunnterm av sort s.

2. Hvis $f \in \Sigma s_1 \ldots s_n, s$ og $t_1 \in \mathcal{T}_{\Sigma,s_1}, \ldots, t_n \in \mathcal{T}_{\Sigma,s_n}$, og $n \geq 1$, så er $f(t_1, \ldots, t_n) \in \mathcal{T}_{\Sigma,s}$. Det vil si at et funksjonsymbol anvendt på termer av riktig sort gir en term av sort s.

3. Hvert sett $\mathcal{T}_{\Sigma,s}$, er det minste settet som tilfredstiller betingelsene over. Med det mener vi at kun termer som kan bli bygget opp av konstanter og funksjonsymboler anvendt på grunntermer av riktig sort, er grunntermer.

En grunnterm er altså uten variable, noe som gjør beregninger mindre interessante. Derfor må vi også definere termer med variable.

Definisjon 10 Termer
Gitt en fler-sortet signatur (S, Σ) og et variabelsett X = $\{X_s | s \in S\}$ vil S-sortet sett av termer $\mathcal{T}_\Sigma(X) = \{\mathcal{T}_{\Sigma,s}(X) | s \in S\}$ være definert induktivt ved betingelsene:

1. $X_s \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$. Det vil si, en variabel av sort s er også en term av sort s.

2. $\Sigma_{\epsilon,s} \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$. En konstant av sort s er også en term av sort s.

3. $f(t_1, \ldots, t_n) \in \mathcal{T}_{\Sigma,s}(X)$ hvis $f \in \Sigma_{s_1 \ldots s_n, s}$ og $t_i \in \mathcal{T}_{\Sigma,s_i}(X)$ for hver i som er $1 \leq i \leq n$.

4. $\mathcal{T}_\Sigma(X)$ er det minste S-sortete settet som tilfredstiller betingelsene over.

Da kan vi definere ligninger:

Definisjon 11 Ligning
Gitt en fler-sortet signatur (S, Σ), er en ligning et tripel (X, t, t') skrevet $(\forall X)t = t'$, hvor X er et S-sortet variabelsett uten navneoverlapp med Σ, og t og t' er termer av samme sort, det vil si $t, t' \in \mathcal{T}_{\Sigma,s}(X)$ for en $s \in S$.

En betinget ligning er ett 2(n + 1)+1-tuppel $(X, u_1, \nu_1, \ldots, u_n, \nu_n, t, t')$ for $n \geq 1$, skrevet

$$(\forall X)u_1 = \nu_1 \wedge \ldots \wedge u_n = \nu_n \Longrightarrow t = t',$$

så sant det finnes sorter $s_1, \ldots, s_n, s \in S$ med $t, t' \in \mathcal{T}_{\Sigma,s}(X)$ og $u_i, \nu_i \in \mathcal{T}_{\Sigma,s_i}(X)$ for hver $i \in \{1, \ldots, n\}$

En ligning angis i Maude ved nøkkelordet **eq**, eller **ceq** hvis det er en betinget ligning. Ligningene er symmetriske i den bakenforliggende teorien, men ved reduksjon er de asymmetriske av tekniske grunner. De anvendes venstre mot høyre og forventes å være terminerende og konfluente. Med konfluens menes at vi til slutt skal ende opp med samme term, selv om vi underveis i reduseringen har mulighet til å følge flere forskjellige veier. I terminerende og konfluente systemer finnes derfor unike normalformer. Den funksjonelle delen av Maude eksekveres ved reduksjoner **red**.

2.2.2 Omskrivingslogikk

Hovedidéen bak omskrivingslogikk er at vi kan ha utvikling og endringer, og ikke bare deterministisk likhet. Med implisitt gitt parallellitet kan dynamisk oppførsel i et distribuert system representeres av omskrivingsregler.

Omskrivingslogikk [17] er en sunn og komplett logikk. Før vi definerer en omskrivingsteori skal vi se på et par enklere begreper.

Et sett Σ er et alfabet av funksjonsymboler $\Sigma = \{\Sigma_n | n \in \mathbb{N}\}$ der alle symbolene har definerte sorter. En Σ-algebra er da et sett A, sammen med en tilordning av en funksjon $f_A : A^n \rightarrow A$ for hver $f \in \Sigma_n$ med $n \in \mathbb{N}$.
T_Σ står for grunntermer, og $T_\Sigma(X)$ for termer med variabler inneholdt i settet X.
Gitt et sett E av Σ-ligninger, vil $T_{\Sigma,E}$ representere ekvivalensklassen av termer $T_{\Sigma,E} = \{[t]_E | t \in T_\Sigma\}$.
Da vil $t =_E t'$ angi kongruens modulo E for to termer t og t'. Vi skriver $[t]_E$ eller bare $[t]$ for E-ekvivalensklassen av t.

Vi kan da definere en omskrivingsteori

Definisjon 12 En omskrivingsteori $\mathcal{R}$ er et fire-tuppel $\mathcal{R} = (\Sigma, E, L, R)$. Σ er et alfabet av funksjonsymboler $\Sigma = \{\Sigma_n | n \in \mathbb{N}\}$. E er et sett av Σ-ligninger. L er et sett med navn. R er et sett av par $R \subseteq L \times (T_{\Sigma,E}(X)^2)^+$, der første komponent er et navn og andre komponent er en ikketom sekvens med par av termer, av E-ekvivalensklassen. X er et tellbart uendelig sett med variabler $X = \{x_1, \ldots, x_n, \ldots\}$. Elementene i R kalles omskrivingsregler. En omskrivingsregel $(l, ([t], [t']))$ er da en *ubetinget* regel skrevet:

$$l : [t] \longrightarrow [t']$$

og $(l, ([t], [t'])([u_1], [v_1]) \ldots ([u_k], [v_k]))$ en *betinget* regel skrevet:

$$l : [t] \longrightarrow [t'] \; if \; [u_1] \longrightarrow [v_1] \wedge \cdots \wedge [u_k] \longrightarrow [v_k]$$

Gitt en omskrivingsteori R sier vi at R avleder sekventen $[t] \longrightarrow [t']$ og skriver

$$R \vdash [t] \longrightarrow [t']$$

hvis og bare hvis $[t] \longrightarrow [t']$ kan oppnås ved applikasjon av reglene:

1. **Refleksivitet.** For hver $[t] \epsilon\ T_{\Sigma,E}(X)$,

$$\frac{}{[t] \longrightarrow [t]}$$

2. **Kongruens.** For hver $f \epsilon\ \Sigma_n$, $n \epsilon\ \mathbb{N}$

$$\frac{[t_1] \longrightarrow [t_1'] \cdots [t_n] \longrightarrow [t_n']}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t_1', \ldots, t_n')]}$$

3. **Utbyttbarhet.** For hver omskrivingsregel $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ i R,

$$\frac{[w_1] \longrightarrow [w_1'] \cdots [w_n] \longrightarrow [w_n']}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x'})]}$$

   der $\overline{x}$ står for $x_1, \ldots, x_n$ og $\overline{w}$ står for $w_1, \ldots, w_n$.

4. **Utbyttbar med betingelse** For hver omskrivingsregel $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ hvis $[u_1(\overline{x})] \longrightarrow [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \longrightarrow [v_k(\overline{x})]$ i R,

$$\frac{\begin{array}{c}[w_1] \longrightarrow [w_1'] \cdots [w_n] \longrightarrow [w_n'] \\ [u_1(\overline{w}/\overline{x})] \longrightarrow [v_1(\overline{w}/\overline{x})] \cdots [u_k(\overline{w}/\overline{x})] \longrightarrow [v_k(\overline{w}/\overline{x})]\end{array}}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x'})]}$$

   der $\overline{x}$ står for $x_1, \ldots, x_n$ og $\overline{w}$ står for $w_1, \ldots, w_n$.

5. **Transitivitet.**

$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

Det må påpekes at symmetri ikke er blant reglene over. Nettopp dette gjør at omskrivingslogikk kan uttrykke irreversible endringer over tid.

I Maude ser en betinget regel ut som følger:

crl [navn] c => c' **if** betingelse .

hvor c og c' er *konfigurasjoner* eller subkonfigurasjoner. I vårt tilfelle er en konfigurasjon et multisett av termer av gitte sorter.

Eksekvering av omskrivingslogikk skjer ved at en gitt startkonfigurasjon (starttilstand) omskrives i et visst antall omskrivingssteg, eller til vi oppnår en sluttkonfigurasjon (sluttilstand) ved terminering. Eksekveringen utføres ved å anvende **rew** (rewrite) eller **frew** (fair rewrite). Begge benytter en *Round Robin* strategi med hensyn på hvilken regel som velges. Forskjellen er at **rew** fortsetter med samme del av konfigurasjonen så lenge det er mulig, mens **frew** skal være mer rettferdig ved at den omskriver den delen som har vært klar lengst. Disse to eksekveringsmulighetene tar som sagt utgangspunkt i en starttilstand, c, og resulterer i ett mulig ekseveringsløp. For å utføre en omskriving av c i 100 omskrivingssteg skriver vi:

$$\textbf{rew}\ [100]\ c\ .$$

For simulering av flere mulige eksekveringsløp har vi **search**, som er et søk gjennom alle tilstander. Det er fortsatt én mulig starttilstand, men her kan vi se på alle mulige tilstander vi kan oppnå fra denne, hvis mengden er endelig.

Søk utføres ved å oppgi en starttilstand, hvordan tilstanden vi ønsker å finne skal se ut, hvor mange tilstander vi ønsker å finne, og hvor mange steg vi ønsker å utføre for å finne tilstanden. I tillegg kan vi oppgi en betingelse for tilstanden. Tilstanden vi ønsker å finne er beskrevet som et mønster, og vi leter etter en tilstand som kan passe inn i dette mønsteret ved mønstergjenkjenning (pattern matching).

$$\textbf{search}\ [\mathsf{antall}]\ t_0\ \rightarrow\ t_p\ \textbf{such that}\ \mathsf{betingelse}\ .$$

Termen t_0 er starttilstanden, t_p er en term som kan inneholde variable, og betingelse er en betingelse på samme form som en betinget regel. Et søk kan også utføres uten betingelse. En term t tilfredstiller søkbetingelsene dersom t_p er en instans av t og betingelse holder for substitusjonen. Hvor mange tilstander vi skal finne bestemmes av antall, som også kan utelates fullstendig. Da vil søket finne alle tilstander som passer betingelsene. Det finnes forskjellige typer piler som skal erstatte $\rightarrow$, nemlig =>, =>*, =>+, eller =>! som står for:

=> tilstander som kan oppnås i *nøyaktig ett* steg fra starttilstanden t_0.

=>* tilstander som kan oppnås i *null eller flere* steg.

=>+ tilstander som kan oppnås i *ett eller flere* steg.

=>! tilstander som ikke kan omskrives videre, *sluttilstander*.

2.2.3 Objektorientert Maude

Ved en objektorientert spesifikasjon vil konfigurasjonen inneholde et multisett av klasser, objekter og meldinger. Det finnes en variant av Maude, "Full Maude" [9, 10], som har klasser, objekter, meldinger og konfigurasjoner av disse, innebygget. Der kan man operere med multippel arv, og objekter blir opprettet fra klassedefinisjoner. En klassedefinisjon representeres ved

$$\textbf{class}\ C \mid a_1 : s_1, \ldots, a_n : s_n\ .$$

hvor C er klassenavnet, a_i er navnene på attributtene, og s_i er de tilhørende sortene, for $1 \leq i \leq n$.

Objekter i Maude er termer av typen

$$< O : C \mid a_1 : \nu_1, \ldots, a_n : \nu_n >$$

hvor O er objektidentifikatoren, C er klassen, a_i er navnene på objektattributtene og ν_i er de tilhørende verdiene, for $1 \leq i \leq n$.

I "Full Maude" vil objekter automatisk se ut som beskrevet over, ut i fra klassedefinisjonen. Dermed trenger man ikke selv å definere objekter som egne typer, men opererer med objekter av klasse C, med attributter som i klassen. Hvis en klasse C_i er subtype av klassen C_k vil C_i inneholde attributtene til C_k i tillegg til sine egne. Objekter av klassen C_i vil, i tillegg til sine egne regler, passe inn i regler for objekter av klassen C_k.

For en konfigurasjon bestående av objekter, for eksempel
$c \equiv A_1\ A_2\ A_3\ \ldots$ har vi

$$\frac{A_i \longrightarrow A_i'}{\ldots A_i \ldots \longrightarrow \ldots A_i' \ldots}$$

og

$$\frac{\begin{array}{c} A_i \longrightarrow A_i' \\ A_j \longrightarrow A_j' \end{array}}{A_i\ A_j \longrightarrow A_i'\ A_j'}$$

i ett steg. Merk at transitivitet ikke er brukt. Objektene A_i og A_j, A_i' og A_j' må være uavhengige av hverandre.

"Full Maude" er en prototype implementert i Maude. Den oversetter en objektorientert modul til en vanlig Maude-modul, som kan eksekveres av Maude-maskinen. "Full Maude" er ikke helt stabilt ennå, så vi nøyer oss med å adoptere noen elementer derfra, og implementerer det i "Core Maude". I en kommende versjon av Maude skal direkte behandling av objekter og arv bli inkludert. Av andre utvidelsesplaner kan innebygget "socket support" for nettverksprogrammering mot internett nevnes.

Kapittel 3

Abstrakt Maskin

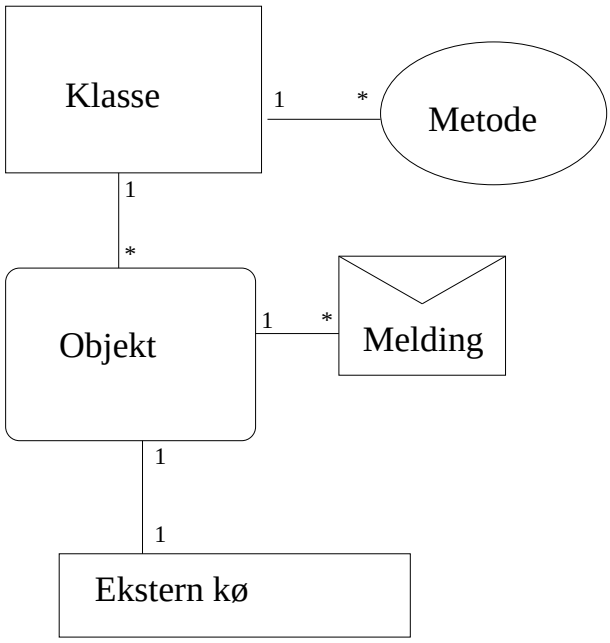
Vi skal i dette kapittelet beskrive en abstrakte maskinen for Creol med fokus på grunnbyggestener i språket. Vi skal derfor se på hvordan Creol-koden blir strukturert, hvilke hjelpefunksjoner vi trenger for å eksekvere språket, og til slutt definere en operasjonell semantikk for Creol. Vår målsetting er en komposisjonal operasjonell semantikk for å tillate eksekvering i parallell. Selve implementasjonen blir beskrevet i kapittel 4.

3.1 Datastruktur

Strukturinformasjonen vi har oppgitt i et Creol-program er grensesnitt og klasser. I den kjørbare koden kan vi anta at uttrykk og setninger er type-riktige, fordi typeanalyse allerede er utført. Siden arv er utenfor fokus i denne oppgaven ser vi helt bort fra grensesnittene, da de ikke inneholder informasjon vi har bruk for i denne omgang. Informasjonen gitt i klassene må representeres på en måte som gjør den tilgjengelig for oss under eksekvering. Vi skal se på elementene i figur 3.1: klasser, metoder, objekter og eksterne køer vil bli representert som tupler omsluttet av spisse klammer, < ... >, mens meldinger blir representert som termer. Dette følger "Full Maude"-konvensjonene ved at de varige elementene er representert som Maude-objekter, mens flyktige meldinger er termer.

3.1.1 Klasser

En Creol-klasse inneholder klassenavn, variable og metoder. Vi skal representere denne informasjonen i et tuppel. Klassenavnet identifiserer tuppelet og lagres i attributtet Cl. Hvis klassen har en eller flere parametre listes disse opp i attributtet Att, deretter resten av variablene. *Init* skal kun eksekveres ved oppretting av nye objekter og blir derfor



Figur 3.1: Den abstrakte maskinen, representert ved forskjellige figurer for de ulike elementene, men med UML-lignende relasjoner.

holdt adskilt fra de andre, invokerbare metodene. I vår representasjon av klasser vil dermed *Init* være et eget attributt $Init$ som inneholder et par av imperativ kode uten vakter, og lokale variable. *Run* inngår sammen med de andre metodene. Disse presenteres som egne tupler i et multisett $Mtds$. Vi får da et klassetuppel:

$$< Cl, Att, Init, Mtds, \ldots >$$

hvor ... skal fylles ut med det vi senere måtte trenge av tilleggs-attributter.

3.1.2 Metoder

Metodene inneholder lokale variable og imperativ kode. Tuppelet vil da bli representert ved:

$$< Mtdname, Latt, Code >$$

hvor metodenavnet i $Mtdname$ identifiserer tuppelet. Attributtet $Latt$ skal inneholde lokale programvariable, det vil si en liste av par med variablenavn og verdi. $Code$ inneholder metodens imperative kode.

Metodene må kunne aksesseres av objektene instansiert fra klassen. Metodene kan enten befinne seg kun i klassen eller i hvert objekt i tillegg. Hvis metodene befinner seg i hvert objekt får vi mange kopier, som tar opp unødvendig plass. Siden Creol også skal kunne støtte dynamiske oppdateringer bestemte vi oss for at metodene kun skulle ligge i klassen, som er det vanlige for objektorienterte språk. Det gir færre kopier av metodene og færre steder å oppdatere kode. Vi har også bedre kontroll over når en metode er oppdatert, siden den bare befinner seg ett sted.

3.1.3 Objekter

Informasjonen som overføres til ett nytt objekt fra klassen ved opprettelse er objektvariable og *Init*. I tillegg vil det opprettes en unik identifikator. Objektet trenger dermed et attributt for identifikatoren kalt Id og ett for objektvariable kalt Att. Vi må også vite hvilken klasse objektet er av med tanke på metodeoppslag, og har derfor et eget attributt for klassenavnet, Cl.

Siden objektet skal kunne variere mellom å være aktivt og passivt trenger vi et attributt for aktiv og et for ventende kode. Attributtet for aktiv kode kaller vi Pr (program). Det vil inneholde imperativ kode. De tilhørende lokale variable legges i attributtet $Lvar$, som en liste av par med variabelnavn og verdier. Den ventende koden legges i attributtet PrQ (program queue). Her står koden på vent sammen med sine lokale variable, representert som par. Objektet blir da representert som et tuppel:

$$< Id, Cl, Pr, PrQ, Lvar, Att, \ldots >$$

hvor ... skal fylles ut med det vi senere måtte trenge av tilleggsattributter.

Siden Creol-objekter skal ha en kø av innkommende kall og returer, oppretter vi en kø assosiert med objektet. Denne kan være et attributt i objektet, som PrQ, eller en egen ekstern kø. Hvis vi bruker PrQ til å ta imot metodekall og returer må objektet selv sjekke om det har mottatt noen meldinger. Vi ønsker best mulig effektivitet med tanke på parallelle transisjoner (se regler i delkapittel 3.2.10) og velger derfor en ekstern kø, representert som et eget sidestilt objekt. Den kan da motta metodekall og returer uavhengig av objektet selv, slik at objektet kan være opptatt andre gjøremål samtidig som meldinger mottas. Køen representeres som et tuppel:

$$< QId, Ev >$$

hvor QId er en identifikator der køen knyttes til sitt objekt, og Ev er et multisett av metodekall og metodereturer.

Når vi skal gjøre et objektkall eller et asynkront lokalt kall som vi ønsker retur på, må vi ha en måte å finne tilbake til kallet på. Vi oppretter derfor et attributt for å telle kall som gjøres, og som skal tildele et unikt nummer til hvert kall, kalt $Lcnt$ (Labelcount). Vi vil videre kalle denne verdien for *etikettverdi*, og navnet verdien tilknyttes for *etikettnavn*.

Metodekall blir sendt som meldinger til køen til det kalte objektet, og hentes ut fra meldingskøen av dette objektet. Meldingen må være merket med identifikator for mottaker og avsender, og inneholde informasjon om metodenavn og aktuelle parametre, foruten etikettverdien. Meldingene er enten start eller avslutting på et metodekall.

$$invoc(L, O, O', M, I)$$

og

$$comp(L, O, J)$$

hvor L er etiketten, O er objektet metoden er kalt fra, O' er objektet som skal motta kallet, M er metodenavnet, I er en liste med aktuelle parametre og J er en liste med returverdier. Vi får da en entydig identifikator for hvert kall ved (L, O), det vil si etikettverdi og objektidentitet. Asynkron behandling av meldingene kan oppnås ved at meldingene blir plassert i konfigurasjonen som egne elementer.

Ved lokale synkrone kall er vi nødt til å utføre kallet direkte for å unngå vranglås. Hvis vi utsetter å utføre kallet vil vi vente aktivt på returen, og dermed aldri komme videre i eksekveringen. Siden kallet skal utføres direkte må vi plassere koden i metoden som kalles rett inn i Pr. En mulighet er å sette koden som skal eksekveres inn, foran koden det skal returneres til. Ulempen med denne løsningen er at vi enten må overskrive de lokale variablene eller slå listene sammen. Vi må finne en løsning hvor vi slipper dette. Hvis vi plasserer koden det skal returneres til, altså fortsettelsen av Pr, på PrQ vil de lokale variablene bevares. Men hvordan skal vi da finne tilbake til riktig returkode?

Den enkleste måten å løse dette problemet på kunne være å opprette ett eget sted å legge denne koden og lokale variable. Vi kunne kalle dette attributtet $Stack$, og det ville være arrangert med en "last in first out" (LIFO) ordning. Dette svarer til en vanlig run-time stack for prosedyrekall i et én-prosessor system [13]. Siden hvert objekt utfører sine oppgaver individuelt vil det også fungere i denne sammenhengen, fordi alle synkrone kall på stacken vil være nøstet inne i hverandre. Ulempen med denne løsningen er at vi får et ekstra attributt å forholde oss til i objektet. Hva med å bruke PrQ i stedet for $Stack$?

Vi har allerede en måte å identifisere metodekall på, nemlig ved etikettverdien i $Lcnt$. Men hvordan skal vi bruke denne ved synkrone kall? Det er ingen retursetning i koden opprinnelig. Hvis vi forsøker å oversette synkrone kall til $l\,!\,m(e)\,;\,l?(x)\,;\ldots$ slik vi gjør med synkrone objektkall vil metoden lastes inn i PrQ, og dermed føre til vranglås. Vi må derfor beholde det synkrone kallet slik som det er. For å håndtere synkrone kall innføres et primitiv i den abstrakte maskinen som ikke finnes i Creol. Vi kaller det continue og lar det ha en etikettverdi som parameter. Primitivet legges til etter den innlastete metodekoden for å si fra hvilken kode vi skal fortsette med. For at vi skal hente ut riktig kode fra PrQ legger vi til en returvakt foran programsetningen. I dette tilfellet er det snakk om en returvakt som skal gjøres sann direkte etter at metodekallet er ferdig eksekvert. Vi har derfor ikke behov for et etikettnavn, men kan uttrykke $n\,?(x)$ direkte, hvor n er etikettverdien vi får fra $Lcnt$. Når eksekveringen av metoden er ferdig kommer vi til continue(n), og kan hente ut nettopp denne programsetningen fra PrQ. Dermed har vi en løsning som bruker PrQ til å lagre koden og de tilhørende lokale variable. Vi velger denne løsningen, og trenger dermed ikke å innføre attributtet $Stack$.

Det endelige objekt-tuppelet vil da se ut som følger:

$$< Id, Cl, Pr, PrQ, Lvar, Att, Lcnt >$$

3.1.4 Nye Objekter

Det finnes flere forskjellige måter å opprette et nytt objekt på. Hvert objekt kan være ansvarlig for de objektene det oppretter, hver klasse kan ta ansvar for sine objekter, eller man kan ha ett "superobjekt" som tar seg av all objektoppretting.

Superobjektet, eller proto-objekt som det opereres med i [18], forkastes fordi variable og annen informasjon må sendes til proto-objektet, noe som kan forårsake inkonsistens i forhold til klassens definisjon. Ved å sende informasjonen ved meldinger åpner vi for å knytte foreldede versjoner av objektvariabler og *Init* til det nyopprettede objektet, fordi vi kan ha dynamiske oppdateringer. Et annet moment er effektiviteten når kun ett objekt, uansett klasse, kan opprettes per omskrivingssteg. I et distribuert system kan ulike deler av systemet befinne seg på forskjellige maskiner. Dette vil kunne bidra til en lavere effektivitet dersom en maskin går ned, eller kommunikasjonen mellom maskinene er treg.

Vi har valgt å la hver klasse opprette objekter. Det må da finnes en objektteller som økes for hvert nytt objekt som instansieres. Hvert objekt vil da få en unik identifikator ved at klassenavnet og objekt-telleren blir konkatenert. Det må da kreves at tall er ulovlig i slutten av

klassenavn. Vi kunne tenke oss en annen variant hvor tall og klassenavn settes sammen som et par. Da slipper vi restriksjoner for klassenavn, og kan dessuten fjerne attributtet hvor klassenavnet står oppført i objektet.

En ulempe ved å bruke klassen til å opprette objekter er effektiviteten ved opprettelse av mange objekter av samme klasse. Kun ett objekt av én klasse kan bli opprettet i ett omskrivingssteg, siden klassen må opptre i omskrivingsregelen. For å bedre effektiviteten kan vi opprette kopier av klassen. Vi må da sørge for at inkonsistens ikke forekommer, ved å ha en instans med skriveaksess, og flere med leseaksess. Siden vi trenger skriveaksess til objekttelleren er det en idé å flytte denne til objektene, og kun finne informasjonen som trengs i klassen. Identifikatoren må da inneholde objektnavnet til den som oppretter objektet konkatenert med telleren. En ulempe ved denne løsningen er at vi etter hvert vil få en sekvens av tall i navnet, som blir vanskeligere å lese. Vi må også sørge for at sekvensen av tall får en skilleoperator mellom seg, for å sikre at identifikatoren er unik. Hvis objekttelleren flyttes til objektene må man ha en ny ordning som sikrer unik identifikator ved opprettelse av det første objektet.

Hvis objekter skal kunne opprette nye objekter uavhengig av klassen må de inneholde metoden *Init*, samt objektvariabler med nullverdier. Ved oppdatering av *Init* eller objektvariablene må da alle objekter også oppdateres. Hvis et objekt av en annen klasse skal opprettes må man uansett ha innsyn enten til klassen eller et objekt av samme klasse. Vi forkaster derfor løsningen hvor objekter skal kunne opprettes uten innsyn i klassen.

Kopiering av et ubestemt antall ved likhetsligninger er hittil ikke mulig i Maude. En strategi for å generere et bestemt antall kopier er en mulighet. Vi har valgt å la en slik effektivisering ligge, og lar derfor objekttelleren ligge i klassen. Ved en effektivisering kan dette enkelt endres. Klassetuppelet vil dermed se ut som følger:

$$< Cl, Att, Init, Mtds, Ocnt >$$

hvor $Ocnt$ er objekttelleren.

Når et objekt opprettes skal vi starte med å eksekvere *Init* for så å fortsette med *Run* før noe annet kan skje. For å sørge for at det skjer i akkurat den rekkefølgen finnes flere løsninger. Hvis høyst en av *Init* eller *Run* kan ha lokale variable kan begge metodene lastes rett inn i Pr, med semikolon mellom. Siden vi ønsker muligheten til å ha lokale variable for begge er ikke dette en løsning, da vi i så fall må slå de lokale variablene sammen i én liste og navnekonflikter kan oppstå.

En annen løsning er å starte med *Init* i Pr og plassere *Run* i PrQ. Vi kan da ha lokale variable i begge metodene, og slipper uheldige feil. Ulempen

med denne løsningen er at vi ikke er garantert at *Run* eksekveres direkte etter *Init*. Et scenario hvor et metodekall til objektet gjøres rett etter at det er opprettet kan føre til at kallet betjenes før *Run*. Vi velger derfor å automatisk kalle *Run* i slutten av *Init*. Vi kan da enten ha et asynkront kall uten etikett, et med etikett, men uten returvakt, eller et synkront kall. Avslutningen på det synkrone kallet vil da ligge i PrQ under hele eksekveringen hvis *Run* er ikke-terminerende. *Init* og *Run* vil hverken ha inn- eller utvariable, derfor vil det være mest naturlig å velge et asynkront kall uten etikett, slik at vi slipper unødvendige programsetninger på PrQ. Men siden de asynkrone kallene lastes opp på PrQ får vi samme problem som hvis *Run* skal settes direkte på PrQ ved opprettelse av objektet. Derfor er vi nødt til å gjøre et synkront kall. *Init* gjør avslutningsvis et synkront kall på *Run* .

Vi har nå beskrevet i detalj en abstrakt maskin som består av klasser, objekter med tilhørende eksterne køer, og meldinger. I figur 3.2 kan vi se en klasse med to instansierte objekter. Attributtene i de forskjellige elementene er kort forklart. Objektene har gjort metodekall til hverandre, derav meldingene.

I tuppelform ville konfigurasjonen i figuren sett ut som følger:

$$
\begin{aligned}
&< O, Att, Init, Mtds, Ocnt > \\
&< O1, Id, PrQ, Lvar, Att, Lcnt > \\
&< Q_{O1}, Ev > \\
&< O2, Id, PrQ, Lvar, Att, Lcnt > \\
&< Q_{O2}, Ev > \\
&invoc(2, O2, O1, M, I) \\
&comp(1, O2, I)
\end{aligned}
$$

Ut ifra informasjonen på figuren kan vi ikke si noe om inn- og utparametrene i meldingene. Disse er derfor kalt I i tuplene.

3.2 Operasjonell Semantikk

Vi tar utgangspunkt i et utkast til en operasjonell semantikk for Creol fra [16]. Denne har vi så tilpasset endringer i Creol og arbeid med interpreten, og dette resulterer i den operasjonelle semantikken for Creol vi nå skal se på. Hver semantisk regel vil få et beskrivende navn skrevet i *kursiv*, som vi skal henvise til i kapittel 4.3.

I den operasjonelle semantikken bruker vi g for Guard, x for (liste av) variable, e for (liste av) uttrykk, C for programsetning(er), GC for

Figur 3.2: Den abstrakte maskinen, med detaljer

setning(er) med vakt først, m for metodenavn og l for etikettverdi. Vi bruker $\uplus$ som notasjon for multisett-union, $\vdash$ for konkatenering av lister, og ε for tom sekvens. Før vi går inn på de semantiske reglene skal vi se på noen hjelpefunksjoner brukt i de semantiske reglene.

3.2.1 Hjelpefunksjoner i den Abstrakte Maskinen

Når flere oppgaver må utføres i samme steg lar vi noen bli håndtert av funksjoner. Et eksempel på en slik funksjon er oppslag av en variabelverdi. I Creols operasjonelle semantikk vil vi anvende følgende hjelpefunksjoner: Φ, val, eval, evalG, update og $\mapsto$. I tillegg har vi Var og $Class$ som hjelper oss å undersøke om noe er i et gitt domene.

Funksjonen Φ henter en metodeinstans fra objektets klasse. Denne består av et par av kode og tilhørende variable som representerer en metode fra objektets klasse. Hvis vi kaller metoden Φ med parametrene metodenavn, et multisett med metoder, og variabelliste får vi paret (*programsetning,variable*) i retur. Vi vil bruke notasjonen Φ_{Pr} for kun å hente koden, og tilsvarende Φ_{Lvar} for å hente lokale variable. Ved Φ_{Pr} vil variabellisten utelates fra parametrene. For detaljer om implementasjonen av Φ henvises til delkapittel 4.2.

Funksjonen val skal slå opp verdien til en variabel. val vil, i tillegg til variabelen, få med $Lvar \vdash Att$ som parameter. Ved overlastede programvariable beholdes verdien fra $Lvar$. Funksjonene eval og evalG vil henholdsvis beregne (lister av) uttrykk og sannhetsverdien til en vakt. eval vil ha uttrykket og $Lvar \vdash Att$ som parametre, og returnere verdien til det ferdig beregnede uttrykket. evalG vil ha vakten og $Lvar \vdash Att$ som parametre, i tillegg får den med Ev for å kunne se etter returmeldinger i tilfelle metodereturvakter.

Funksjonen $\mapsto$ skal skrive ny verdi til angitt variabelnavn enten i variabellisten $Lvar$ eller Att. Eksempelvis vil Att[flag $\mapsto$ true] gjøre at variabelen flag i Att blir satt til true. Vi har også en variant av $\mapsto$, betegnet ved $\mapsto_l$, som tilordner en liste av verdier til en liste av variable. Eksempelvis vil [a b $\mapsto_l$ 3 4] resultere i at a tilordnes 3 og b tilordnes 4. Funksjonen update får med en liste med par av variabelnavn og verdier, og en liste kun med verdier, som parametre. Den skal oppdatere det samme antall variable som listen med verdier er lang, og starte forfra. Eksempelvis vil update((x : 5, y : 6, z : 7), 2 4) resultere i (x : 2, y : 4, z : 7).

3.2.2 Transisjonsregler

Vi ser nå på transisjonsreglene i den abstrakte maskinen. Som nevnt tidligere blir meldingene plassert i konfigurasjonen, for deretter å bli hentet inn i den eksterne køen ved følgende regler.

- Hente inn kall

$$\frac{}{< Q_{Id'},\ Ev >\ invoc(l, Id, Id', m, e) \longrightarrow \\ < Q_{Id'},\ Ev \uplus invoc(l, Id, Id', m, e) >}$$

- Hente inn retur

$$\frac{}{< Q_{Id},\ Ev >\ comp(l, Id, e) \longrightarrow \\ < Q_{Id},\ Ev \uplus comp(l, Id, e) >}$$

Q er indeksert med Id for å vise hvilket objekt køen tilhører. Id og Q_{Id} er begge unike identifikatorer.

3.2.3 Tilordninger

- *Tilordning av lokal variabel:*

$$\frac{x \in Var(Lvar)}{< Id,\ Cl,\ x := e\ ;\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \longrightarrow \\ < Id,\ Cl,\ Pr,\ PrQ,\ Lvar[x \mapsto e'],\ Att,\ Lcnt >}$$

  hvor e' er $eval(e, Lvar \mapsto Att)$

Hvis x befinner seg blant de lokale variablene beregnes uttrykket e, og lagres i x.

- *Tilordning av objektvariabel:*

$$\frac{x \in Var(Att)}{< Id,\ Cl,\ x := e\ ;\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \longrightarrow \\ < Id,\ Cl,\ Pr,\ PrQ,\ Lvar,\ Att[x \mapsto e'],\ Lcnt >}$$

  hvor e' er $eval(e, Lvar \mapsto Att)$

Hvis x befinner seg blant objektvariablene beregnes uttrykket e, og lagres i x.

3.2.4 If- og while-setninger

Hvis if-setningens betingelse evalueres til true utføres koden, hvis ikke forkastes setningen.

- *If-setning, sann betingelse:*

$$\frac{eval(g,\ Lvar \mapsto Att)\ = \text{true}}{< Id,\ Cl,\ (\textbf{if}\ g\ \textbf{then}\ C\ \textbf{fi})\ ;\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \longrightarrow \\ < Id,\ Cl,\ C\ ;\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt >}$$

- *If-setning, usann betingelse:*

$$\frac{eval(g, Lvar \mapsto Att) = \text{false}}{< Id, Cl, (\textbf{if } g \textbf{ then } C \textbf{ fi}) ; Pr, PrQ, Lvar, Att, Lcnt > \longrightarrow < Id, Cl, Pr, PrQ, Lvar, Att, Lcnt >}$$

En mulig utvidelse av språket kan være å kunne teste metoderetur i if-setninger.

- *While-løkke:*

$$\frac{}{< Id, Cl, (\textbf{while } g \textbf{ do } GC \textbf{ od}) ; Pr, PrQ, Lvar, Att, Lcnt > \longrightarrow < Id, Cl, (\textbf{if } g \textbf{ then } GC ; (\textbf{while } g \textbf{ do } GC \textbf{ od}) \textbf{ fi}) ; Pr, PrQ, Lvar, Att, Lcnt >}$$

While-løkken blir behandlet som en rekursiv if-setning, og testen av betingelsen overlates til if-reglene.

3.2.5 Vakter

Hvis vaktene evalueres til sann utføres programsetningen. Hvis ikke flyttes programsetningen med vakt og lokale variable over på køen PrQ. Vi ser her på de tre formene for basale vakter.

- *wait -vakt*

$$\frac{}{< Id, Cl, wait \longrightarrow C, PrQ, Lvar, Att, Lcnt > \longrightarrow < Id, Cl, \varepsilon, PrQ \uplus (\text{true} \longrightarrow C, Lvar), \varepsilon, Att, Lcnt >}$$

En *wait*-vakt gjør at aktiv blir suspendert med vakten true.

- *Boolsk vakt, sann betingelse*

$$\frac{eval(g, Lvar \mapsto Att) = \text{true}}{< Id, Cl, g \longrightarrow C, PrQ, Lvar, Att, Lcnt > \longrightarrow < Id, Cl, C, PrQ, Lvar, Att, Lcnt >}$$

- *Boolsk vakt, usann betingelse*

$$\frac{eval(g, Lvar \mapsto Att) = \text{false}}{< Id, Cl, g \longrightarrow C, PrQ, Lvar, Att, Lcnt > \longrightarrow < Id, Cl, \varepsilon, PrQ \uplus (g \longrightarrow C, Lvar), \varepsilon, Att, Lcnt >}$$

- *Metodereturvakt, sann betingelse*

$$\frac{}{\begin{array}{l}< Q_{Id},\ Ev \uplus comp(l', Id, e) > \\ < Id,\ Cl,\ l?(x) \longrightarrow C,\ PrQ,\ Lvar,\ Att,\ Lcnt > \longrightarrow \\ < Q_{Id},\ Ev > \\ < Id,\ Cl,\ [x \mapsto_l e]\,;\ C,\ PrQ,\ Lvar,\ Att,\ Lcnt >\end{array}}$$

hvor $val(l, Lvar) = l'$

Her finner vi returmeldingen ved å sjekke om l' og verdien til l har samme verdi. I stedet for å bruke en funksjon som leter gjennom den eksterne køen utnytter vi at Ev er et multisett, og matcher meldingen ved etiketten, for så å fjerne den fra køen. I neste steg vil listen av returparametre tilordnes de korrekte variablene i $Lvar$ og Att.

- *Metodereturvakt, usann betingelse*

$$\frac{evalG(l?(x),\ Lvar \mapsto\!\dashv Att,\ Ev)\ = \mathsf{false}}{\begin{array}{l}< Q_{Id},\ Ev > \\ < Id,\ Cl,\ l?(x) \longrightarrow C,\ PrQ,\ Lvar,\ Att,\ Lcnt > \longrightarrow \\ < Q_{Id},\ Ev > \\ < Id,\ Cl,\ \varepsilon,\ (l?(x) \longrightarrow C,\ Lvar) \uplus PrQ,\ \varepsilon,\ Att,\ Lcnt >\end{array}}$$

Hvis metodereturvakten er usann flytter vi den til PrQ.

3.2.6 Aktivering av programsetning på køen

Når Pr er tom kan vi hente en ny programsetning fra køen PrQ. For å bli eksekvert må vakten evalueres til sann. Vi deler evalueringen opp i regler for boolske vakter og metodereturvakter, for å få eksplisitt kontroll med meldinger i den eksterne køen.

- *Aktiver programsetning med boolsk vakt:*

$$\frac{eval(g,\ L \mapsto\!\dashv Att)\ = \mathsf{true}}{\begin{array}{l}< Id,\ Cl,\ \varepsilon,\ (g \longrightarrow C,\ L) \uplus PrQ,\ Lvar,\ Att,\ Lcnt > \longrightarrow \\ < Id,\ Cl,\ C,\ PrQ,\ L,\ Att,\ Lcnt >\end{array}}$$

- *Aktiver programsetning med metodereturvakt:*

$$\frac{}{\begin{array}{l}< Id,\ Cl,\ \varepsilon,\ (l?(x) \longrightarrow C,\ L) \uplus PrQ,\ Lvar,\ Att,\ Lcnt > \\ < Q_{Id},\ Ev \uplus comp(l', Id, e) > \longrightarrow \\ < Id,\ Cl,\ [x \mapsto_l e]\,;\ C,\ PrQ,\ L,\ Att,\ Lcnt > \\ < Q_{Id},\ Ev >\end{array}}$$

hvor $val(l, Lvar) = l'$

3.2.7 Metodekall

Vi skal her se på både lokale kall og objektkall. For de lokale kallene ser vi på både asynkrone og synkrone kall, og for objektkall ser vi på asynkrone kall med og uten etikett. Det er ikke nødvendig å behandle synkrone objektkall separat, siden det oversettes fra $o.m(e;x);\ldots$ til $l!o.m(e;x);l?(x);\ldots$.

- *Objektkall med etikett:*

$$\frac{}{\begin{array}{l}< Id,\ Cl,\ l\,!\,o.m(e)\,;\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \longrightarrow \\ < Id,\ Cl,\ Pr,\ PrQ,\ Lvar[l \mapsto Lcnt],\ Att,\ Lcnt + 1 > \\ invoc(Lcnt, Id, o, m, e')\end{array}}$$

hvor e' er $eval(e, Lvar \mapsto Att)$

Det kallende objektet genererer en melding til objektet som kalles. Det er så opp til objektets kø å behandle forespørselen.

- *Objektkall uten etikett:*

$$\frac{}{\begin{array}{l}< Id,\ Cl,\ !\,o.m(e)\,;\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \longrightarrow \\ < Id,\ Cl,\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt + 1 > \\ invoc(Lcnt, null, o, m, e')\end{array}}$$

hvor e' er $eval(e, Lvar \mapsto Att)$

Vi ser at objektkall med og uten etikett er veldig like. Den største forskjellen er hva man sender som returadresse. Hvis vi ikke ønsker noen retur, noe vi uansett ikke gjør uten etikett, sender vi med **null**. I tillegg lagres ikke $Lcnt$ ved et asynkront kall uten etikett, men den økes allikevel for historiens og symmetriens del.

- *Starte metodekall:*

$$\frac{m \in Calls(Id)}{\begin{array}{l}< Q_{Id},\ invoc(l, Id', Id, m, e) \uplus Ev > \\ < Id,\ Cl,\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \\ < Cl,\ Att,\ (Init,\ Var),\ Mtds,\ Ocnt > \longrightarrow \\ < Q_{Id},\ Ev > \\ < Id,\ Cl,\ Pr,\ (\Phi(m,\ Mtds,\ (l, Id', e))) \uplus PrQ, \\ \ Lvar,\ Att,\ Lcnt > \\ < Cl,\ Att,\ (Init,\ Var),\ Mtds,\ Ocnt >\end{array}}$$

Her har vi tatt med kravet om at m skal være en eksisterende metode. Selve testingen av dette er overlatt til typesjekker, men vi ønsker å ha en robust semantikk og har derfor testen med.

Et asynkront lokalt kall kan enten gjøres på samme måte som et asynkront objektkall, eller som den semantiske regelen *Starte metodekall* over, uten å involvere den eksterne køen. Vi velger siste alternativ, siden man da får lastet koden inn i PrQ i ett steg, mot minimum to steg ved det første alternativet.

- *Asynkront lokalt kall med etikett:*

$$
\begin{array}{l}
< Id,\ Cl,\ l\,!\,m(e)\,;\,Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \\
< Cl,\ Att,\ (Init,\ Var),\ Mtds,\ Ocnt > \longrightarrow \\
< Id,\ Cl,\ Pr,\ PrQ \uplus (\Phi(m,\ Mtds,\ (Lcnt, Id, e'))), \\
\quad Lvar[l \mapsto Lcnt],\ Att,\ Lcnt + 1 > \\
< Cl,\ Att,\ (Init,\ Var),\ Mtds,\ Ocnt >
\end{array}
$$

hvor e' er $eval(e, Lvar \mapsto Att)$

Det asynkrone kallet uten etikett blir veldig likt. Forskjellen er at $Lcnt$ ikke lagres på kallstedet, og at Id er uinteressant for metoden som kalles.

- *Synkront lokalt kall:*

$$
\begin{array}{l}
< Id,\ Cl,\ m(e\,;\,x)\,;\,Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \\
< Cl,\ Att,\ (Init,\ Var),\ Mtds,\ Ocnt > \longrightarrow \\
< Id,\ Cl,\ (\Phi_{Pr}(m,\ Mtds))\,;\,continue(Lcnt), \\
\quad (Lcnt\,?(x)\,;\,Pr,\ Lvar) \uplus PrQ, \\
\quad (\Phi_{Lvar}(m,\ Mtds,\ e')),\ Att,\ Lcnt + 1 > \\
< Cl,\ Att,\ (Init,\ Var\},\ Mtds,\ Ocnt >
\end{array}
$$

hvor e' er $eval(e, Lvar \mapsto Att)$

Ved et synkront lokalt kall lastes koden direkte inn i Pr, og de tilhørende lokale variablene plasseres i $Lvar$. Resten av koden etter kallet plasseres på PrQ, sammen med sine lokale variable. Vi legger til merket continue for påminnelse om hvor vi skal fortsette etter endt eksekvering, og $Lcnt\,?(x)$ foran koden vi skal fortsette med.

- *Avslutning av synkront lokalt kall:*

$$\frac{}{\begin{array}{l}< Id,\ Cl,\ continue(n),\ (n?(x)\ ;\ C, L) \uplus PrQ,\ Lvar,\ Att,\ Lcnt > \\ < Q_{Id},\ Ev \uplus comp(n, Id, e) > \longrightarrow \\ < Id,\ Cl,\ [x \mapsto_l e]\ ;\ C,\ PrQ,\ L,\ Att,\ Lcnt > \\ < Q_{Id},\ Ev >\end{array}}$$

Vi ser at programsetningen det skal fortsettes med kan hentes direkte ut av PrQ og at variablene tilordnes.

3.2.8 Nytt objekt

- *New:*

$$\frac{}{\begin{array}{l}< Cl,\ Att,\ (Init,\ Var),\ Mtds,\ Ocnt > \\ < Id,\ Cl',\ x := \textbf{new}\ Cl(e)\ ;\ Pr,\ PrQ,\ Lvar,\ Att',\ Lcnt > \longrightarrow \\ < Q^{NewId},\ \varepsilon > \\ < NewId,\ Cl,\ Init,\ \varepsilon,\ Var,\ update(Att, e'), 1 > \\ < Cl,\ Att,\ (Init,\ Var),\ Mtds,\ Ocnt + 1 > \\ < Id,\ Cl',\ x := NewId\ ;\ Pr,\ PrQ,\ Lvar,\ Att',\ Lcnt >\end{array}}$$

hvor e' er $eval(e, Lvar \mapsto Att')$
og $NewId$ er $Cl \mapsto Ocnt$

Når et objekt blir opprettet lages også en tilhørende kø. Objektet får en unik identifikator ved klassenavnet konkatenert med objekttelleren. Paret $(Init,\ Var)$ er den imperative koden i metoden *Init*, og lokale variable. $Init$ plasseres i Pr, de lokale variablene i $Lvar$. Objektparametrene e evalueres til e' og plasseres først i objektvariablene Att. Vi minner om at funksjonen update oppdaterer de første parametrene i en variabelverdi-liste, så lenge det er flere verdier igjen i uttrykkslisten.

3.2.9 Ikke-determinisme

Ved å teste vakten før vi velger programsetning styres det ikke-deterministiske valget slik at vi alltid får utført en kodebit. Hvis ingen av vaktene er sanne legges kodebiten på køen PrQ. Ikke-determinisme kan behandles tilsvarende i PrQ, når vaktene skal hentes ut.

- □, *første vakt er sann*

$$\frac{evalG(g_1,\ Lvar \mapsto Att,\ Ev)\ = \mathsf{true}}{\begin{array}{l} < Q_{Id},\ Ev > \\ < Id,\ Cl,\ (g_1 \longrightarrow C_1 \Box GC_2)\ ;\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \longrightarrow \\ < Q_{Id},\ Ev > \\ < Id,\ Cl,\ g_1 \longrightarrow C_1\ ;\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \end{array}}$$

- □, *andre vakt er sann*

$$\frac{evalG(g_2,\ Lvar \mapsto Att,\ Ev)\ = \mathsf{true}}{\begin{array}{l} < Q_{Id},\ Ev > \\ < Id,\ Cl,\ (GC_1 \Box g_2 \longrightarrow C_2)\ ;\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \longrightarrow \\ < Q_{Id},\ Ev > \\ < Id,\ Cl,\ g_2 \longrightarrow C_2\ ;\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \end{array}}$$

Vakten behandles da i neste steg av den passende semantiske regelen.

- □, *begge vakter er usanne*

$$\frac{\begin{array}{c}(evalG(g_1,\ Lvar \mapsto Att,\ Ev) \vee \\ evalG(g_2,\ Lvar \mapsto Att,\ Ev))\ = \mathsf{false}\end{array}}{\begin{array}{l} < Id,\ Cl,\ (g_1 \longrightarrow C_1 \Box\ g_2 \longrightarrow C_2)\ ;\ Pr,\ PrQ,\ Lvar,\ Att,\ Lcnt > \\ < Q_{Id},\ Ev > \longrightarrow \\ < Id,\ Cl,\ \varepsilon,\ PrQ \uplus ((g_1 \longrightarrow C_1 \Box\ g_2 \longrightarrow C_2)\ ;\ Pr, Lvar), \\ \quad Lvar,\ Att,\ Lcnt > \\ < Q_{Id},\ Ev > \end{array}}$$

3.2.10 Parallellitet mellom objekter

Objektene i Creol skal være parallelle på den måten at de kan utføre oppgaver samtidig, og uavhengig av hverandre. Hvis A_i et tupler for $i \geq 1$ så

$$\frac{A_i \longrightarrow A_i'}{\ldots A_i \ldots \longrightarrow \ldots A_i' \ldots}$$

og hvis A_j også er tupler for $j \geq 1$ så

$$\frac{\begin{array}{c}A_i \longrightarrow A_i' \\ A_j \longrightarrow A_j'\end{array}}{A_i\ A_j \longrightarrow A_i'\ A_j'}$$

Objektene A_i og A_j, A_i' og A_j' må være uavhengige av hverandre.

3.2.11 Oppgavens bidrag til Creols operasjonelle semantikk

Den operasjonelle semantikken for Creol definert over, er basert på et skissepreget notat "An Interleaving Evaluation Semantics for Object Oriented Method Calls" [16] og representerer en vesentlig utvidelse av denne skissen. Disse endringene skyldes at videre arbeid med språket såvel som implementasjonen av reglene i Maude har avduket mangler i den opprinnelige skissen. For å klargjøre dette kapittelets bidrag til Creols operasjonelle semantikk, vil vi derfor nevne de største endringene og utvidelsene av semantikken i forhold til den opprinnelige skissen. Alle punktene viser til hvordan det ble gjort i [16]. Vi starter med de viktigste forskjellene:

- Etiketter er ikke innført. Derfor er (reglene for) kall og vakter noe annerledes.
- Synkrone lokale kall finnes ikke.
- Semantikken i [16] er ikke formulert i omskrivingslogikk, men beskrives ved en tilstandsovergang-relasjon, uten bruk av mønstergjenkjenning.
- Det finnes to eksterne køer for hvert objekt, en inn- og en utkø. Derfor er alle semantiske regler som involverer meldinger og køer annerledes.
- Ved opprettelse av nye objekter slås *Init* og *Run* sammen til (*Init* ; *Run*). Attributtet for lokale variable i det nye objektet er tomt. Det er heller ikke mulig å ha parametre til objekter.
- Boolske vakter og metodereturvakter behandles i én regel, uten å definere hva som skjer ved en metodereturvakt og hvordan det skal gjøres. Dette kan føre til en bieffekt på den eksterne innkøen når en annen vakt enn metodereturvakt behandles.
- Innlasting fra det som tilsvarer PrQ gjøres i én regel. Her kan vi, som i punktet over, få en bieffekt på den eksterne innkøen.
- Innhenting av metodekode gjøres ved en funksjon som bare er delvis definert, uten å ha med klassen i regelen. I denne oppgaven gjøres det ved mønstergjenkjenning.
- Det finnes en funksjon **get** som kan hente utparametre fra en metoderetur. Denne kommer i tillegg til en metodereturvakt, så man spør først om den har kommet, for så å hente returverdiene med **get**.

- Kall oppføres som en funksjon **call** med parametrene metodenavn og innparametre, og i tillegg objektidentitet dersom det er snakk om et eksternt kall.
- Retur oppføres som en funksjon **reply** med parametrene objektidentitet, metodenavn, inn- og utparametre.

Kapittel 4

Implementasjon av den abstrakte maskinen i Maude

I dette kapittelet vil implementasjonen av en abstrakt maskinen for Creol i Maude bli beskrevet i detalj. Implementasjonen består av tre forskjellige deler: definisjonen av CMC og oversettelse fra Creol, omskrivingsregler basert på den operasjonelle semantikken, og hjelpefunksjoner og typer.

Siden vi allerede har definert datastrukturen og den operasjonelle semantikken i kapittel 3 vil mye se likt ut, og noen av forklaringene gjentas. Det er et poeng at dette kapittelet skal kunne leses uavhengig av resten av oppgaven, for eksempel for å få nok informasjon til videre utvikling.

4.1 Creol Machine Code

For at Creol skal være enklere å interpretere i Maude er det hensiktsmessig å definere input til interpreten på en form som gjør det enkelt å eksekvere i Maude. Språket som definerer dette formatet kaller vi Creol Machine Code (CMC). Creol-objekter, klasser, metoder og uttrykk defineres som termer i Maude. Vi skal her se på oversettelsen fra Creol til CMC ved en funksjon $[\![\ ]\!]$ som tar Creol-kode og gir CMC. I definisjonene for oversettelsesfunksjonen er typene i CMC indirekte definert.

Koden er forsøkt oversatt slik at CMC blir så likt Creol-kode som mulig. Det er allikevel noen nødvendige forskjeller som enten skyldes Maude-spesifikke momenter, eller behov for informasjon som i Creol er semantisk bundet, men som trengs eksplisitt i CMC. Oversettingsfunksjonen er ikke implementert, så foreløpig gjøres oversettelsen for hånd. Selve oversettelsen faller utenfor fokus for oppgaven, og dette arbeidet er såpass omfattende at implementasjonen av $[\![\ ]\!]$ er utelatt. Det er selvsagt

at Creol trenger en typesjekker for å kunne brukes i større skala. Vi ser for oss at denne også inkluderer oversettelsesfunksjonen. Allikevel trenger vi definisjonen på hvordan det skal gjøres for å tilfredstille interpretens krav. Dette skal vi se nærmere på her.

4.1.1 Eksekvering

En ferdig oversatt Creol-kode resulterer i en konfigurasjon av klasser. For å starte eksekveringen har vi en spesialterm for oppretting av det første objektet (**new** ⟦klassenavn(parametre)⟧). Termen opptrer sidestilt med klassene i startkonfigurasjonen. Det opprettes et objekt av denne klassen, og deretter går eksekveringen som normalt, ved utføring av kode i slike objekter. Ved at nye objekter kan opprettes, vil konfigurasjonen etter hvert inneholde objekter, klasser, eksterne køer og meldinger. Denne konfigurasjonen er et multisett. Vi slipper å inkludere regler for parallellitet mellom objekter, siden disse er inkludert i Maude-maskinen. Hvis en subkonfigurasjon er uavhengig av en annen, kan omskrivingsregler anvendes på begge subkonfigurasjoner samtidig.

4.1.2 Typer og verdier

Creols predefinerte variabeltyper må representeres i Maude, inklusive Creols universaltype *Data* som kan være enten boolsk, string, heltall, liste eller objektidentitet. Den naive måten å angripe problemet på er å la *Data* være supertype for de tilsvarende innebygde Maude-typene. Dette viste seg å være problematisk siden Maude har predefinerte funksjoner som for eksempel '+' både for tekststrenger og heltall. Man får da beskjed om at operatoren _+_ har blitt importert både fra pakkene NAT og STRING, fra Maudes bibliotek `"prelude.maude"` uten felles arvbakgrunn, og at sortene Nat og String derfor ikke kan være subtyper for samme supertype. Med *Data* som supertype for begge disse sortene, vet ikke lenger Maude-maskinen hvilken av funksjonene som skal benyttes, og vil derfor ikke importere modulen som definerer *Data*.

En løsning er å omgå problemet, ved å redefinere Maudes predefinerte funksjoner. Det kan gjøres ved å endre en kopi av filen `"prelude.maude"` som automatisk importeres ved oppstart. En ulempe ved denne løsningen er at biblioteket oppdateres når Maude oppdateres. Det kan da komme enda flere konflikter enn ved tidligere versjoner. I perioden arbeidet med interpreten foregikk ble Maude oppgradert fra Maude 1 til Maude 2.0, og mistanken ble bekreftet. Omfanget av antall funksjoner berørt økte drastisk. Vi kunne selvfølgelig beholdt den gamle versjonen av `"prelude.maude"`, men siden Maude er i sterk utvikling

ønsker vi ikke å begrense oss til én versjon. Denne måten å løse problemet på ble derfor forkastet.

En annen løsning er å innkapsle hver av variabeltypene, ved funksjoner som tar de forskjellige typene som parameter og returnerer noe av typen $Data$. Man trenger da i tillegg likhetslogiske funksjoner som pakker verdiene ut igjen ved behov for beregning. Denne løsningen fungerte tilfredstillende, og er derfor valgt.

Definisjon 13 Oversetting fra Creol -verdier til $Data$. Indirekte gir dette en definisjon av hvordan variabeltypen $Data$ er bygget opp.

- $[\![\text{null}]\!] \rightarrowtail$ **null**
- $[\![\textit{heltall}]\!] \rightarrowtail$ **int**(*heltall*)
- $[\![\textit{tekststreng}]\!] \rightarrowtail$ **str**(*tekststreng*)
- $[\![\textit{boolsk verdi}]\!] \rightarrowtail$ **bool**(*boolsk verdi*)
- $[\![elem_1, \cdots, elem_k]\!] \rightarrowtail$ **list**($[\![elem_1]\!] \ \cdots \ [\![elem_k]\!]$)

Bortsett fra **null**, som er en konstantfunksjon, er hver av de ovenstående en funksjon med parameter som vist i parentes. Tilsvarende kan andre typer legges til, for eksempel produkt-typer og union-typer.

Elementene i listen kan være av en eller flere forskjellige $Data$-typer, med blank som skilleoperator.

En objektidentitet eksisterer først under eksekvering. Denne er dermed ikke synlig for programmereren, annet enn typen, og trenger ikke en egen funksjon selv om den er av typen $Data$. Vi bruker Maude-typen "quoted identifier" (Qid) til å identifisere objekter, og lar $Data$ være supertype av Qid.

4.1.3 Uttrykk

Når uttrykk skal defineres i CMC ønsker vi i størst mulig grad å bruke de samme symbolene som er benyttet i Creol. Ved å benytte den naive løsningen for universaltypen $Data$ kom også disse i konflikt med Maudes predefinerte funksjoner. I første forsøk på å løse problemet la vi til apostrofer (') foran hvert av symbolene. Symbolene blir da sammenblandet med den innebygde typen Qid, og et uttrykk e_1 '+ e_2 kan dermed også være en liste. For å bli kvitt dette problemet ble det definert egne tekstlige symboler som pluss, minus og lignende. Ved å bruke innkapslede typer løste problemet seg, siden Maudes predefinerte typer ikke lenger er subtyper av $Data$. Nå kan man benytte Creols egne symboler, bortsett fra når det er overlasting i Creol. Dette inkluderer

også unær og binær minus. Siden Maude ikke har mulighet til å legge forskjellig prioritet på de to får vi flertydig parsering av for eksempel $-int(5) - int(4)$ uansett hvordan parentesene ordnes.

I definisjonene nedenfor bruker vi notasjonen u for generelle uttrykk, s for tekstuttrykk, og i for talluttrykk.

Definisjon 14 Oversettelse fra Creol-uttrykk til $Expr$:

Uttrykk med annet symbol enn i Creol :

- $Data$ er en subsort av $Expr$
- $[\![$- $u]\!] \rightarrowtail$ **neg** $[\![u]\!]$
- $[\![s_1 + s_2]\!] \rightarrowtail [\![s_1]\!]$ **cat** $[\![s_2]\!]$

Uttrykk med samme symbol som i Creol :

- $[\![i_1 + i_2]\!] \rightarrowtail [\![i_1]\!] + [\![i_2]\!]$
- $[\![u_1$ - $u_2]\!] \rightarrowtail [\![u_1]\!]$ - $[\![u_2]\!]$
- $[\![u_1 * u_2]\!] \rightarrowtail [\![u_1]\!] * [\![u_2]\!]$
- $[\![u_1 / u_2]\!] \rightarrowtail [\![u_1]\!] / [\![u_2]\!]$
- $[\![u_1 < u_2]\!] \rightarrowtail [\![u_1]\!] < [\![u_2]\!]$
- $[\![u_1$ <= $u_2]\!] \rightarrowtail [\![u_1]\!]$ <= $[\![u_2]\!]$
- $[\![u_1 > u_2]\!] \rightarrowtail [\![u_1]\!] > [\![u_2]\!]$
- $[\![u_1$ >= $u_2]\!] \rightarrowtail [\![u_1]\!]$ >= $[\![u_2]\!]$
- $[\![u_1 = u_2]\!] \rightarrowtail [\![u_1]\!] = [\![u_2]\!]$
- $[\![u_1$ /= $u_2]\!] \rightarrowtail [\![u_1]\!]$ /= $[\![u_2]\!]$
- $[\![$**not** $u]\!] \rightarrowtail$ **not** $[\![u]\!]$
- $[\![u_1$ **and** $u_2]\!] \rightarrowtail [\![u_1]\!]$ **and** $[\![u_2]\!]$
- $[\![u_1$ **or** $u_2]\!] \rightarrowtail [\![u_1]\!]$ **or** $[\![u_2]\!]$

4.1.4 Imperativ kode

Vi fortsetter å bruke u for et enkelt uttrykk, i tillegg kommer e for en liste av uttrykk, og x for en liste av variable.

Definisjon 15 Navn blir oversatt til typen Qid:

- ⟦variabelnavn⟧ $\rightarrowtail$ 'variabelnavn
- ⟦etikettnavn⟧ $\rightarrowtail$ 'etikettnavn
- ⟦klassenavn⟧ $\rightarrowtail$ 'klassenavn
- ⟦metodenavn⟧ $\rightarrowtail$ 'metodenavn

Vi legger til en apostrof (') foran navn for at de skal kunne være med i en Maude-type, og dermed være inneholdt i en tellbar uendelig mengde navn. Hvis vi skulle brukt en egendefinert type måtte hvert eneste navn være definert som funksjon uten parametre i Maude, noe som intuitivt er lite praktisk!

Definisjon 16 Variabeldeklarasjoner, også kalt vardekl, blir oversatt til typen $ListQidVal$ på følgende måte:

- ⟦variabel: Type⟧ $\rightarrowtail$ (⟦variabel⟧ : **null**)

Variabeldeklarasjoner blir oversatt til et par av navn og verdi for at vi skal kunne ta vare på verdien i Maude. Alle variable blir satt til **null** uansett type, og må derfor bli initialisert på et senere tidspunkt. Når verdien til en variabel endres må dette paret oppdateres.

Definisjon 17 Tilordning, if-setninger og while-løkker blir oversatt som følger:

- ⟦variabel := uttrykk⟧ $\rightarrowtail$ ⟦variabel⟧ := ⟦uttrykk⟧
- ⟦variabel := **new** klassenavn $(u_1, \ldots, u_n)$⟧ $\rightarrowtail$
  ⟦variabel⟧ := **new** ⟦klassenavn⟧ (⟦u_1⟧ ... ⟦u_n⟧)
- ⟦**if** uttrykk **then** kode **fi**⟧ $\rightarrowtail$ **if** ⟦uttrykk⟧ **th** ⟦kode⟧ **fi**
- ⟦**if** uttrykk **then** kode1 **else** kode2 **fi**⟧ =
  if ⟦uttrykk⟧ **th** ⟦kode1⟧ **el** ⟦kode2⟧ **fi**
- ⟦**while** uttrykk **do** kode **od**⟧ $\rightarrowtail$
  while ⟦uttrykk⟧ **do** ⟦kode⟧ **od**

Av samme grunn som datatyper kolliderer if-setningen med Maudes innebygde funksjon. Vi må derfor endre noe på nøkkelordene, og velger å endre **then** til **th** og **else** til **el**. Det kunne vært gjort andre endringer, og det hadde holdt å endre ett nøkkelord, men ble gjort som over fordi vi da fikk en symmetri med alle nøkkelordene på to bokstaver.

Definisjon 18 Kall blir oversatt som følger:

- ⟦m (e ; x)⟧ ⤖ ⟦m⟧ (⟦e⟧ ; ⟦x⟧)
- ⟦l ! m(e)⟧ ⤖ ⟦l⟧ ! ⟦m⟧ (⟦e⟧)
- ⟦! m(e) ⟧ ⤖ ! ⟦m⟧ (⟦e⟧)
- ⟦l ? (x)⟧ ⤖ ⟦l⟧ ? (⟦x⟧)
- ⟦o.m(e ; x)⟧ ⤖ 'la ! ⟦o⟧ . ⟦m⟧ (⟦e⟧) ; 'la ? (⟦x⟧)
- ⟦l ! o.m(e) ⟧ ⤖ ⟦l⟧ ! ⟦o⟧ . ⟦m⟧ (⟦e⟧)
- ⟦! o.m(e)⟧ ⤖ ! ⟦o⟧ . ⟦m⟧ (⟦e⟧)

hvor m er et metodenavn, e og x henholdsvis inn- og utparametre, l er et etikettnavn og o en objektpeker. Metoder uten parametre får en tom liste "nil". Vi minner om at et synkront objektkall $o.m(e;x)$ blir oversatt til et asynkront kall og en retursetning med aktiv venting. Det er derfor viktig at metoder med synkrone objektkall får lagt til $'la$ som lokal variabel, og at den omdøpes dersom det oppstår navnekonflikter med andre etikettnavn i metoden.

Definisjon 19 Vakter blir oversatt som følger:

- ⟦wait ⟶ kode⟧ ⤖ wait —> ⟦kode⟧
- ⟦l ? (x) ⟶ kode ⟧ ⤖ ⟦l⟧ ? (⟦x⟧) —> ⟦kode⟧
- ⟦uttrykk ⟶ kode ⟧ ⤖ ⟦uttrykk⟧ —> ⟦kode⟧

Vaktene som er sammensatt med & oversettes hver for seg, som i definisjon 19. Som nevnt tidligere (se delkapittel 2.1.4) vil alle de boolske vaktene slås sammen til en vakt med **and**, og vi vil kun ha én *wait*-vakt. Vi vil også benytte oss av en miste mulige vakt, den tomme vakten nothing.

4.1.5 Klasse

Creol-klasser er representert ved Maude-objekter på formen

$$< Cl \mid Att, Init, Mtds, Ocnt >,$$

hvor Cl er klassenavnet, Att er en liste av variable, $Ocnt$ er antall objekter instansiert fra klassen som brukes ved **new**, og $Mtds$ er et multisett av metoder. Når et objekt trenger en metode lastes denne fra $Mtds$ i objektets tilhørende klasse. Attributtet $Init$ inneholder denne metodene. Att inneholder både klasseparametre og andre deklarerte variable. Dette attributtet kunne derfor vært delt i to, men siden begge deler behandles likt er det naturlig og plassbesparende å kun ha ett. Variablene listes opp med **null**verdier slik at vi får en liste med par. Denne listen kan da kopieres direkte inn i et nyopprettet objekt.

Oversettelsen blir som følger, og vi ser kun på de delene av klassen som det fokuseres på i denne oppgaven. Vi ser derfor bort fra arv i sin helhet, inkludert hvem som har tilgang til forskjellige metoder. I tillegg har vi utelatt den semantiske spesifikasjonen, med invarianter og antagelser, siden disse ikke behandles her.

Definisjon 20 Klasseoversettelse

⟦ **class** *klassenavn* (*vardekl0*)
  begin
    [**var** *vardekl*]
    [**op** init == *vardekl1*
                kode1 .]
    [**op** run == *vardekl2*
                kode2 .]
    { **op** *metodedeklarasjoner* }*
   {**with** grensesnittnavn
    {**op** *metodedeklarasjoner* }*
   }*
  end ⟧ ⟶
< ⟦*klassenavn*⟧ : Cl |
   Att: (⟦ *vardekl0* ⟧, ⟦ *vardekl* ⟧),
   Init: (⟦ kode1 ⟧ ; ⟦run(nil ; nil)⟧, ⟦ *vardekl1* ⟧),
   Mtds: ⟦ *metodedeklarasjoner* ⟧ *
         ⟦ run == *vardekl2* kode2 . ⟧,
   Ocnt: 0 >

Vi ser at det legges til et synkront kall på *Run* i *Init*. *Run* oversettes på samme måte som de andre metodene. Det påpekes at * er skillesymbolet mellom metodetuplene.

4.1.6 Metode

En metode er representert ved et objekt på formen

$$< Mtdname \mid Latt, Code >$$

hvor $Latt$ er metodens lokale attributter. Denne inkluderer en variabel som kan lagre etikettverdien til kallet og en variabel som kan lagre identifikatoren til kalleren. $Code$ er metodens imperative kode.

Definisjon 21 Metodeoversettelse

⟦ *metodenavn* (x_1: type, ... , x_n: type **out** y_1: type, ... ,y_n: type) ==
vardekl
kode **.** ⟧ ⟼
< ⟦*metodenavn*⟧ : Mtdname |
Latt: (('label : **null**), ('caller : **null**),
(⟦x_1⟧ : **null**), ⟦...⟧, (⟦x_n⟧ : **null**), ⟦*vardekl*⟧,
(⟦y_1⟧ : **null**), ⟦...⟧, (⟦y_n⟧ : **null**)),
Code: ⟦kode⟧ ; end(⟦y_1 ... y_n⟧) >

De lokale variablene vil starte med plass til etiketten, deretter følger plass til kallers navn og innvariable, og til slutt andre lokale variable og utvariable. Avslutningen med punktum erstattes med en egen setning `end` som har en liste av navnene på utparametrene. For en parser som har bygget opp en struktur av Creol-koden er det enkelt å huske utparametrene, og tilrettelegge for eksekvering av interpreten ved å sette inn slike eksplisitte end-setninger.

4.1.7 Objekt

Siden objekter ikke blir opprettet før ved run-time oversettes ikke disse av kompilatoren. Allikevel må vi definere en standard for hvordan objektene skal se ut i Maude. Et Creol-objekt blir representert ved et Maude-objekt

$$< Id \mid Cl, Pr, PrQ, Lvar, Att, Lcnt >$$

hvor Id er objektets identifikator, Cl er klassenavnet, Pr er den aktive koden, PrQ er et multisett av ventende kodebiter, $Lvar$ og Att er lokale- og objektvariable. Til slutt har vi $Lcnt$ som er en teller for etikettverdier ved metodekall.

Hvert Creol-objekt har en tilhørende kø. Den er definert som følger i Maude:

$$< QId \mid Ev >$$

hvor QId er køens identifikator, konstruert ved en funksjon q som tar det tilhørende objektets identifikator. Ev er et multisett av uleste eksterne meldinger. Her kunne vi også ha valgt en annen struktur enn multisett (se kapittel 6). Meldingene er enten start eller avslutning på et metodekall:

$$invoc(L, O, O', M, I)$$

og

$$comp(L, O, J)$$

hvor L er etiketten, O er objektet som kaller metoden, O' er objektet kallet skal til, M er metodenavnet, I er en liste over aktuelle innparametre og J er en liste over returverdier.

4.2 Hjelpefunksjoner

For å få Creol til å eksekvere som ønsket i interpreten behøver man mer enn bare struktur og omskrivingsregler. Å slå opp verdien til en variabel er et eksempel på noe man ønsker skal skje umiddelbart. Derfor bruker vi likhetslogiske funksjoner til slike formål. De likhetslogiske funksjonene blir utført mellom omskrivingsstegene, og er en viktig del av den underliggende abstrakte maskinen.

Definisjon 22 Funksjoner:

val finner verdien til en variabel.

eval beregner et uttrykk, returnerer typen $Data$.

evalB beregner et uttrykk, returnerer typen $bool$.

evalI beregner et uttrykk, returnerer typen int.

evalS beregner et uttrykk, returnerer typen $string$.

evalG beregner sannhetsverdien til en vakt, returnerer typen $bool$.

evalList beregner verdien til hvert uttrykk i en liste, returnerer en liste med elementer av typen $Data$.

inqueue ser om en melding befinner seg i køen.

occursIn sjekker om en variabel befinner seg i en liste.

makeAssignment lager tilordning mellom variabel og verdi.

alter endrer verdien til en variabel.

getNames henter alle navn på variable fra en liste.

getVar henter en metodes lokale variable.

getCode henter en metodes kode.

get henter en metodes kode og lokale variabler.

Vi skal nå se implementasjonen av enkelte av de overstående funksjonene, der vi bruker variabelnavnene:

- Q og R for Qid,
- E for $Expr$,
- L for $ListQidVal$,
- B for boolsk verdi,
- C for heltall,
- S for tekststreng,
- I og J for lister,
- N for positivt heltall,
- O for objektidentitet,
- MM for meldingskø,
- P for programkode,
- MT for multisett av metoder

Funksjonen `val` skal slå opp verdien til variabelnavnet Q i listen av variable som er sendt med. Den leter rekursivt gjennom listen, og returnerer verdien om den finnes. Hvis ikke returneres variabelnavnet selv.

```
eq val(Q, no) = Q .
eq val(Q, ((R : E), L)) =
  if R == Q then E else val(Q, L) fi .
```

Funksjonen `eval` skal beregne verdien til et uttrykk i tillegg til å slå opp verdier der det trengs. For de minste mulige uttrykkene, trengs ingen beregning. Variabelnavn vil matches mot verdi med `val`.

```
eq eval(null, L) = null .
eq eval(bool(B), L) = bool(B) .
eq eval(int(C), L) = int(C) .
eq eval(str(S), L) = str(S) .
eq eval(Q, L) = val(Q, L) .
```

Uttrykkene som skal resultere i en boolsk verdi omsluttes av bool(u). Funksjonene går rekursivt innover i uttrykket, fjerner innpakningen, endrer funksjonssymbol der det trengs, og lar Maude håndtere selve beregningen.

```
eq eval(not E, L) = bool(not evalB(E,L)) .
eq eval(E and E', L) = bool(evalB(E,L) and evalB(E',L)) .
eq eval(E or E', L) = bool(evalB(E,L) or evalB(E',L)) .

eq eval((E > E'), L) = bool(evalI(E,L) > evalI(E',L)) .
eq eval((E >= E'), L) = bool(evalI(E,L) >= evalI(E',L)) .
eq eval((E < E'), L) = bool(evalI(E,L) < evalI(E',L)) .
eq eval((E <= E'), L) = bool(evalI(E,L) <= evalI(E',L)) .

eq eval(E = E', L) = bool((eval(E,L) == eval(E',L))) .
eq eval(E /= E', L) = bool((eval(E,L) =/= eval(E',L))) .
```

Over ser vi at funksjonssymbolene **not**, **and** og **or** knyttes til boolske uttrykk. Derfor kalles det på `evalB` som returnerer en boolsk verdi. Sammenligningsoperatorene $<$, $\leq$, $\geq$ og $>$ forventer tall, derfor kalles `evalI` som beregner talluttrykk og returnerer heltall. Disse operatorene kan utvides til å behandle andre typer, for eksempel tekstrenger. Sammenligningsoperatorene $=$ og $\neq$ kan derimot ta alle typer, derfor sammenlignes de som den generelle typen $Data$.

For tekststrenger har vi kun en operator, konkatenering. Funksjonen `evalS`, som returnerer en tekststreng, kalles.

```
eq eval(E cat E', L) = str(evalS(E, L) + evalS(E', L) ) .
```

For heltall har vi de fire hovedoperatorene, og unær minus, som skal resultere i en heltallsverdi. Her omsluttes uttrykket av int(u), og det kalles rekursivt innover.

```
eq eval((neg E), L) = int(- evalI(E, L)) .
eq eval((E + E'), L) = int(evalI((E + E'), L)) .
eq eval((E - E'), L) = int(evalI((E - E'), L)) .
eq eval((E * E'), L) = int(evalI((E * E'), L)) .
eq eval((E / E'), L) = int(evalI((E / E'), L)) .
```

Funksjonen evalList evaluerer hvert av uttrykkene i listen ved å bruke eval, til listen er tom.

```
eq evalList(nil, L) = nil .
eq evalList(E I, L) = eval(E, L) evalList(I, L) .
```

Funksjonen evalG brukes spesielt i forbindelse med ikke-determinisme, men kunne også vært benyttet for å undersøke sammensatte vakter.

```
eq evalG(nothing --> P, L, MM) = true .
eq evalG(wait & G --> P, L, MM) = evalG(G --> P, L, MM) .
eq evalG(E --> P, L, MM) = evalB(E, L) .
eq evalG((Q ? ( J )) & G --> P, L, MM) =
 inqueue(evalI(Q, L), MM) and evalG(G --> P, L, MM) .

eq evalG(SP ; P, L, MM) = true .
eq evalG(P [] P', L, MM) =
 evalG(P, L, MM) or evalG(P', L, MM) .
```

Vi ser at det finnes en ligning for hver vakttype. Tilfellet med wait resulterer i sann fordi den garantert blir sann om noen steg. Vi har også ligninger for tilfellet uten vakt, og for nøstet ikke-determinisme. Ligningen uten vakt trengs for at vi skal kunne tillate ikke-deterministiske valg uten vakter. Ligningen for nøstet ikke-determinisme trengs hvis vi skal fange opp at det finnes en sann vakt, uavhengig av rekkefølgen.

Funksjonene occursIn og alter brukes gjerne i sammenheng med hverandre. Ved å undersøke om variabelen er i en gitt liste kan vi deretter endre variabelverdien til variabelen i denne listen.

```
eq Q occursIn no = false .
eq Q occursIn ((R : D), L) =
  if Q == R then true else (Q occursIn L) fi .

eq alter(Q, E, no) = no .
eq alter(Q, E, ((R : D), L)) =
  if R == Q then (R : E), L
  else (R : D), alter(Q, E, L) fi .
```

Funksjonen occursIn leter etter et variabelnavn i en liste, og returnerer true hvis den finner navnet. Hvis listen har blitt tom uten at en retur er gjort vet vi at variabelnavnet ikke finnes, og false returneres. Funksjonen alter endrer verdien til en variabel i en variabelliste, og returnerer den oppdaterte listen. Hvis vi finner navnet byttes verdien med den nye, og dette paret returneres sammen med resten av listen. Hvis vi ikke

finner navnet kalles det videre rekursivt, og paret som ble sjekket legges til foran kallet. Hvis vi ikke har funnet variabelen før listen er tom returneres den dermed uendret.

For å finne en returmelding i køen må vi finne den med samme etikettverdi som vi leter etter. Køattributtet er et multisett med sambindingsoperator +.

```
eq inqueue(N, none) = false .
eq inqueue(N, comp(N, O, J) + MM) = true .
eq inqueue(N, comp(N', O, J) + MM) =
  if N == N' then true else inqueue(N, MM) fi .
```

Hvis køen er tom kan vi med sikkerhet si at meldingen ikke finnes der. Siden køen er et multisett kan Maude-maskinen sortere meldingene på en vilkårlig måte. Vi har derfor tatt med ligningen i midten, hvor vi ser at tallene er like, ved at de har samme variabelnavn. Den siste ligningen går gjennom meldingene en etter en, og tar ut meldinger etterhvert som de er undersøkt. Finner vi meldingen returneres `true`.

Funksjonene getCode og getVar omtales henholdsvis som Φ_{Pr} og Φ_{Lvar} i delkapittel 3.2. Hver av dem leter gjennom et multisett med metoder, med sambindingsoperator *.

```
eq getCode(Q, < R : Mtdname | Latt: L,Code: P > * MT, I) =
  if Q == R then makeAssignment(getNames(L), I) ; P
  else getCode(Q, Mt, I) fi .

eq getVar(Q, < R : Mtdname | Latt: L, Code: P > * MT) =
  if Q == R then L else getVar(Q, Mt) fi .
```

Funksjonen getCode skal som navnet tilsier hente ut kode. I tillegg lager den tilordninger for parametre og nødvendig kallinformasjon, som navn på kalleren og etikettverdi. Tilordningene lages av makeAssignment (se under). Når metoden er funnet returneres koden, med tilordningene foran. Hvis ikke kalles getCode rekursivt med resten av multisettet med metoder. Funksjonen getVar skal hente ut lokale variable fra en metode. Hvis riktig metode er funnet returneres variabellisten, hvis ikke kalles metoden rekursivt med resten av metodemultisettet.

```
eq get(Q, < R : Mtdname | Latt: L, Code: P > * MT, I) =
  if Q == R then
    nothing --> makeAssignment(getNames(L), I) ; P, L
  else get(Q, MT, I) fi .
```

Funksjonen get er en kombinasjon av getCode og getVar. Den skal returnere et par av kode og variable. Siden alle par som settes på den

indre køen PrQ må starte med en vakt legges det til en tom vakt nothing. Denne vakten evalueres til sann ved mønstergjenkjenning. Vi unngår dermed en betinget test.

```
eq makeAssignment(J, nil) = empty .
eq makeAssignment(Q I, E J) =
   (Q := E) ; makeAssignment(I, J) .
```

Funksjonen makeAssignment konstruerer tilordninger. Den tar to lister som parametre, en med navn og en med verdier. Så lenge det er flere verdier igjen kalles funksjonen rekursivt, og den returnerer en programsetning.

```
eq getNames(no) = nil .
eq getNames((Q : D), L) = Q getNames(L) .
```

Siden funksjonen makeAssignment krever en ren navneliste har vi funksjonen getNames, som henter ut navnene fra en variabelliste.

4.3 Regler

Interpreten er implementert i standard Maude. Vi vil her presentere reglene i "Full Maude" stil [3] for å gjøre de mer leselige. I "Full Maude" stil omtales de attributtene i en regel som det *leses fra*, på venstre side, og de attributtene som *endres*, på høyre side av overgangspilen.

Noen av de semantiske reglene blir slått sammen ved å anvende en if-else-setning i høyresiden av regelen, i stedet for en enkel test på hver. Da får vi færre regler, og får effektivisert de tilfellene hvor det bare er ett riktig valg. I tilfellene hvor ikke-determinisme skal være med på å bestemme hva som utføres er det fortsatt nødvendig med to regler. Enkle regler som tilordning kan overføres ganske direkte, så de skal vi ikke se på. De fleste reglene bør være gjenkjennelige fra den operasjonelle semantikken, men vi skal her gå i mer detalj på hva som faktisk skjer.

4.3.1 Vakter

Vi har essensielt tre basale vakter i Creol . Den velkjente boolske vakten, en metodereturvakt og ventevakten. Vi har også konjunksjoner av de forskjellige typene. Her ser vi på regler for de tre basale vaktene.

Boolsk vakt
Den boolske vakten fungerer ved at vakten kan passeres dersom det boolske uttrykket beregnes til true, hvis ikke plasseres hele

programsetningen i køen. Vi har slått sammen de to semantiske reglene *Boolsk vakt, sann betingelse* og *Boolsk vakt, usann betingelse*, ved å anvende en if-else-setning i høyre side av regelen.

```
rl [boolguard] :
< O : Id | Pr: E --> P, PrQ: W, Lvar: L, Att: A >
=>
if evalB(E, (L, A)) then
< O : Id | Pr: P >
else
< O : Id | Pr: empty, PrQ: W : (E --> P, L), Lvar: no >
fi .
```

Beregnes vakten til true utføres P. Hvis vakten beregnes til false plasseres hele programsetningen i PrQ sammen med de lokale variablene, i påvente av at vakten skal bli true .

Metodereturvakt
Her kan kjøringen kun fortsette hvis avslutning på kallet har kommet i den eksterne køen.

```
crl [returnguard] :
< O : Id | Pr: X ? ( J ) --> P, Lvar: L >
< q( O ) : QId | Ev: M + comp(N, O, K) >
=>
< O : Id | Pr: makeAssignment(J, K) ; P >
< q( O ) : QId | Ev: M >
if (int(N) == val(X, L)) .
```

Funksjonen makeAssignment lager tilordningssetninger som utføres i senere steg. Så hvis J er navnet på en variabel og K en verdi vil returen fra makeAssignment bli $J := K$, som tilordningsregelen så behandler. De lokale variablene overskrives med programsetningens tilhørende variable L, og avslutningen fjernes fra den eksterne køen.

Hvis metodereturen ikke finnes i den eksterne køen ønsker vi at koden settes på vent i PrQ.

```
crl [returnguard_notinqueue] :
< O : Id | Pr: (X ? ( J )) --> P, PrQ: W, Lvar: L >
< q( O ) : QId | Ev: M >
=>
< O : Id | Pr: empty, PrQ: W : ((X ? ( J )) --> P, L),
  Lvar: no >
< q( O ) : QId | Ev: M >
if not inqueue(evalI(X, L), M) .
```

Grunnen til at disse reglene er beholdt todelt, på samme måte som de semantiske reglene er at vi måtte sett gjennom køen tre ganger ved funksjon hvis meldingen var der. En for å se om den var der, en for å få returverdiene, og en for å ta den ut av den eksterne køen. Nå gjøres alle tre oppgavene ved mønstergjenkjenning, det vil si at hvis mønsteret vi leter etter finnes i køen kan regelen brukes, noe som er mindre komplekst enn søking ved funksjon. Hvis meldingen ikke er der finner man heller ikke mønsteret. For å være sikker på at den ikke er der må det sjekkes ved en funksjon.

Ventevakt
Poenget med denne vakten er å legge inn et obligatorisk avløsningspunkt for at andre metoder skal få slippe til.

```
rl [waitguard] :
< O : Id | Pr: wait --> P, PrQ: W, Lvar: L >
=>
< O : Id | Pr: empty, PrQ: W : (bool(true) --> P, L),
  Lvar: no > .
```

Det gjøres ved at programsetningen flyttes fra å være aktiv til å bli passiv i PrQ. Da kan objektet enten hente programsetningen direkte tilbake, eller ta en annen med sann vakt. Ved sammenligning med den semantiske regelen ser vi kun Maude-spesifikke forskjeller.

Konjunksjoner
Vi kan i tillegg til enkeltstående vakter ha en sammensetning av disse med operatoren &. Vaktene vil da sees på som en mengde hvor alle må være sanne samtidig. Mengden er bygget opp med nothing som minste identitet, og det er kommutativt og assosiativt. Evalueringen gjøres ved at metodereturvakter og ventevakten evalueres før en eventuell boolsk vakt. Dette kan forsvares ved at en metodereturvakt ikke blir usann igjen når metodereturen først er kommet, og at *wait* omgjøres til true ved evaluering. Det kan ikke forsvares å gjøre det omvendt da en boolsk vakt kan bli usann igjen mens vi venter på metodereturer eller utfører en ventevakt.

Vi har valgt å gjøre det på en slik måte for å få enkle, generelle regler og, ikke minst, så få som mulig. Ved å gjøre det på denne måten kan vi ende opp med en tom vakt. Vi trenger derfor regler for å behandle den tomme, alltid sanne, vakten.

```
rl [emptyguard] :
< O : Id | Pr: nothing --> P >
=>
< O : Id | Pr: P > .
```

```
rl [emptyguard_st] :
< O : Id | Pr: empty, PrQ: (( nothing --> P), L') : W >
=>
< O : Id | Pr: P, PrQ: W,  Lvar: L' > .
```

Vi trenger en regel for en tom vakt i det aktive attributtet Pr, og en for køen PrQ. Den første regelen kunne like godt vært en ligning, siden det ikke skjer andre endringer enn å fjerne vakten. Når vi skal hente en programsetning fra den indre køen trenger vi derimot en regel siden flere attributter er involvert.

Se vedlegget med Maude-kode for et fullstendig sett med regler i forbindelse med operatoren &.

4.3.2 Aktivering av programsetning på køen

Når objektet ikke har mer å jobbe med kan det hente en programsetning fra køen. Den vil enten ha en metodereturvakt eller en boolsk vakt, eller en sammensetning av disse.

Ved en boolsk vakt beregnes uttrykket, og regelen utføres kun hvis det evalueres til `true` .

```
crl [guard_st] :
< O : Id | Pr: empty, PrQ: (( E --> P ), L) : W, Att: A >
=>
< O : Id | Pr: P, PrQ: W, Lvar: L >
if eval(E, (L , A)) .
```

Er uttrykket E sant utføres programsetningen P. De lokale variablene overskrives med programsetningens tilhørende variable L. Sammenlignet med den operasjonelle semantikken er det kun Maude-spesifikke forskjeller.

```
crl [return_guard_st] :
< O : Id | Pr: empty, PrQ: (( X ? ( J )) --> P, L) : W >
< q( O ) : QId | Ev: M + comp(N, O, K) >
=>
< O : Id | Pr: makeAssignment(J, K) ; P, PrQ: W, Lvar: L >
< q( O ) : QId | Ev: M >
if (int(N) == (val(X, L))) .
```

Hvis avslutningen med samme merke som den vi venter på finnes i den eksterne køen, settes returvariablene, som over, og avslutningen fjernes fra køen. Sammenlignet med den semantiske regelen *Aktiver programsetning med metodereturvakt* i 3.2.6 ser vi at regelen over har en mer konkret test av etikettverdien.

4.3.3 Metodekall

Den største forskjellen mellom lokale kall og objektkall er at objektet må utføre koden selv ved et lokalt kall. Vi skal derfor forklare reglene hver for seg, selv om det selvfølgelig er mange likheter.

Lokale kall

Vi har tre forskjellige lokale kall: asynkront med retur, asynkront uten retur, og synkront. Vi skal se på reglene for alle tre og se på forskjeller og likheter.

Asynkront lokalt kall uten retur
Vi kan ha et asynkront kall uten retur hvis avslutning av metodekallet ikke interesserer oss og vi ikke trenger utparametrens verdi.

```
rl [localcall-asynchronic-noreturn] :
< O : Id | Cl: C, Pr: ! X ( I ) ; P, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
< C : Cl | Mtds: MT >
=>
< O : Id | Pr: P,
PrQ: get(X, MT, int(N) null evalList(I,(A,L))) : W,
Lcnt: N + 1 >
< C : Cl | > .
```

Funksjonen `get` henter metodens kode og lokale variabler. De aktuelle parametrene I legges i de lokale variablene, sammen med navnet på hvem returen skal til, og etiketten på kallet. Siden vi ikke er interessert i en retur blir $'caller$ satt til **null**, som er vår konvensjon.

Asynkront lokalt kall med retur
Det asynkrone kallet med retur ligner mye på kallet over. Forskjellen ligger i at vi lagrer etikettverdien på kallstedet, og sender med kallers objektidentifikator. Når det eventuelt kommer en returvakt eller setning et sted senere i koden må vi se etter nettopp denne etikettverdien i den eksterne køen.

```
rl [localcall-asynchronic-return] :
< O : Id | Cl: C, Pr: (X ! Y ( I )) ; P, PrQ: W, Lvar: L,
Att: A, Lcnt: N >
< C : Cl | Mtds: MT > =>
< O : Id | Pr: (X := int(N)) ; P,
PrQ: get(Y, MT, int(N) O evalList(I, (A,L)) ) : W,
Lcnt: N + 1 >
< C : Cl | > .
```

Vi ser her at *O* sendes med som parameter for å lagres i metodens lokale variabler som returadresse. Klassen er eksplisitt med i regelen, fordi vi her må ha innsyn i metodene. Ved sammenligning med den semantiske regelen *Asynkront lokalt kall med etikett* i 3.2.7 ser vi at tilordningen av etikettverdien først skjer i neste steg i denne regelen. I tillegg ser vi i detalj hva funksjonen Φ, her kalt get, får med av parametre.

Synkront lokalt kall

Et synkront kall skal utføres før vi kan fortsette eksekvering av påfølgende kode. Det vil si at vi venter aktivt på returen. Hvis vi venter aktivt i *Pr* får vi aldri utført metodekallet, noe som fører til vranglås.

```
rl [localcall-synchronic] :
< O : Id | Cl: C, Pr: ( X ( I ; J )) ; P, PrQ: W,
  Lvar: L, Att: A, Lcnt: N >
< C : Cl | Mtds: MT >
=>
< O : Id | Pr: getCode(X, MT, int(N) O evalList(I,(L, A)))
  ; continue(N), PrQ: ((N ? (J)) ; P, L) : W,
  Lvar: getVar(X, MT), Att: A, Lcnt: N + 1 >
< C : Cl | > .
```

Vi setter den ventende koden på *PrQ* og setter inn et merke med etikettverdien etter metodens kode. Merket continue sier fra at vi har gjort et lokalt synkront kall, og at vi har ventende kode som skal tas av *PrQ* med samme etikettverdi. Ved sammenligning med den semantiske regelen *Synkront lokalt kall* ser vi ingen påfallende forskjeller annet enn det Maude-syntaktiske.

Når metoden er utført kommer vi til setningen continue. Vi kan da forvente at det ligger en returmelding på den eksterne køen, med samme etikettverdi.

```
rl [continue] :
< O : Id | Pr: continue(N), PrQ: ((N ? (J)) ; P, L) : W >
< q( O ) : QId | Ev: M + comp(N, O, K) >
=>
< O : Id | Pr: makeAssignment(J, K) ; P, PrQ: W, Lvar: L >
< q( O ) : QId | Ev: M > .
```

Tilordningen av returvariablene gjøres ved funksjonen makeAssignment og meldingen fjernes fra køen.

Objektkall

Ved objektkall skiller ikke interpreten mellom synkrone og asynkrone kall. Grunnen til det er at kallerobjektet ikke har noen innvirkning i hvordan objektet utfører sin metode. Skillet mellom asynkront og synkront kall ligger her i hvordan kallobjektet venter, aktivt eller passivt. Det vil si at det kun er to regler for å starte kallene: en med og en uten etikett.

Kall med etikett
Ved et kall med etikett ønsker kalleren et svar, og må derfor ta vare på etikettverdien.

```
rl [objectcall-label] :
< O : Id | Pr: Q ! X . Y (I) ; P, Lvar: L, Att: A, Lcnt: N >
=>
< O : Id | Pr: (Q := int(N)) ; P, Lcnt: (N + 1) >
invoc(N, O, val(X, (L, A)), Y, evalList(I, (L, A))) .
```

Objektets id sendes med i meldingen, for at mottakeren skal ha en returadresse.

Kall uten etikett
Vi benytter objektkall uten etikett hvis vi ikke ønsker noen returverdi, akkurat som ved lokale asynkrone kall uten etikett.

```
rl [objectcall-nolabel] :
< O : Id | Pr: ! X . Y ( I ) ; P, Lvar: L, Att: A, Lcnt: N >
=>
< O : Id | Pr: P, Lcnt: (N + 1) >
invoc(N, null, val(X, (L, A)), Y, evalList(I, (L, A))) .
```

Siden kalleren ikke behøver returen i dette tilfellet sendes ingen returadresse med, det vil si avsender står som **null**. Vi sender allikevel med etikettnummer, og øker $Lcnt$ med en for oversiktens del. Det er altså intet behov for en unik etikett i dette tilfellet, men vi ønsker å opprettholde at hvert kall har en unik identifikator for historiens og bevisenes del.

Å motta et objektkall
Et objekt kan når som helst hente ut meldinger fra sin ytre kø. Det vil da legge kallet på vent, klart til eksekvering.

```
rl [receivecall] :
< O : Id | Cl: C, PrQ: W >
< q( O ) : QId | Ev: M + invoc(N, O', O, Y, I) >
< C : Cl | Mtds: MT > =>
```

```
< O : Id | PrQ: W : get(Y, MT, (int(N) O' I)) >
< q( O ) : QId | Ev: M >
< C : Cl | >
```

Kode og lokale variabler legges på PrQ, og aktuelle parametre, kaller og etikett blir lagret i de tilhørende lokale variablene. Her finner vi ingen forskjeller ved sammenligning med den semantiske regelen *Starte metodekall*.

Aktiv venting på retur fra objektkall
Hvis det gjøres et synkront objektkall venter kalleren aktivt på returen. Det vil si at hvis objektet som utfører metoden kaller på en metode i kalleren vil dette resultere i vranglås. Her kunne det være ønskelig med en time-out mulighet, men det er ennå ikke innarbeidet i Creol eller CMC.

```
crl [getreply] :
< O : Id | Pr:  (X ? ( J )) ; P, Lvar: L >
< q( O ) : QId | Ev: M + comp(N, O, K) >
=>
< O : Id | Pr: makeAssignment(J, K) ; P >
< q( O ) : QId | Ev: M >
if (int(N) == val(X, L)) .
```

Hvis riktig returmelding har kommet tilordnes returvariablene.

Avslutning av metode

På slutten av en metode skal det stå end(utparameternavn) uansett om metoden har utparametre eller ikke. Finnes ingen utparametre skal det stå end(nil). Kallerens konvensjon er at hvis han ønsker beskjed om avslutning av kallet sender han med objektpeker til seg selv, hvis ikke sendes tom peker, **null**.

```
rl [reply] :
< O : Id | Pr: (end( J )) ; P, Lvar: L, Att: A >
=>
< O : Id | Pr: P >
if val('caller, L) == null then none
else comp( evalNum('label, L),
      val('caller, L),
      evalList(J, (L, A) )) fi .
```

Her fortsetter objektet med P, hvis den i det hele tatt inneholder noe. Hvis $'caller$ ønsker melding tilbake, det vil si har en annen verdi enn **null**, sendes en avslutningsmelding med etikettverdi, identifikatoren til objektet og utparametre.

4.3.4 Nye objekter

For opprettelse av nye objekter brukes **new** ⟦Klassenavn(parametre)⟧. I interpreten vil det, i tillegg til det nye objektet, opprettes den tilhørende eksterne køen.

```
rl [new] :
< C : Cl | Att: A, Init: (SP, L), Ocnt: N >
< O : Id | Pr: (X := new C(I)) ; P, Latt: L', Att: A' >
=>
< O : Id | Pr: (X := qid(string(C) + string(N))) ; P >
< C : Cl | Ocnt: (N + 1.0) >
< qid(string(C) + string(N)) : Id | Cl: C,
Pr: makeAssignment(getNames(A), evalList(I, (L',A'))) ; SP,
PrQ: none, Lvar: L, Att: A, Lcnt: 1 >
< q(qid(string(C) + string(N))) : QId | Ev: none > .
```

Den unike identifikatoren til det nye objektet blir navnet på klassen konkatenert med $Ocnt$. En innebygget funksjon i Maude, string sørger for at tallet og teksten kan konkateneres, qid formaterer teksten tilbake til typen Qid. $Ocnt$ er definert til å være et reelt tall fordi Maude kun har en innebygget konverteringsfunksjon for reelle tall. På opprettelsesstedet vil identifikatoren bli lagret, og brukt som objektpeker. Siden vi har en annen regel som tar seg av tilordning utfører vi ikke det i denne omgang. Sammenlignet med den semantiske regelen *New* ser vi ingen forskjeller annet enn det Maude-spesifikke.

Hjelpefunksjonen makeAssignment lager tilordninger mellom de første objektvariablene og de aktuelle parametrene. Oversettelsen til CMC skal ha satt formelle parametre som de første variablene i listen. Siden objektvariablene står med **null**-verdier hentes bare navnene ut ved funksjonen getNames. Funksjonen evalList henter og beregner verdiene til de aktuelle parametrene i listen I.

4.3.5 Ikke-determinisme

Vi kan se for oss at Maude utfører ikke-determinismen, uten å se på vaktene først. Reglene blir da som følger:

```
rl [ikke-determ] :
< O : Id | Pr: (P [] P') ; R >  =>
< O : Id | Pr: P ; R > .

rl [ikke-determ2] :
< O : Id | Pr: (P [] P') ; R >  =>
< O : Id | Pr: P' ; R > .
```

Denne løsningen har den ulempen at vi ikke er sikre på å få utført noen av setningene. Vi ønsker derfor å skrive reglene som i den operasjonelle semantikken. Der utføres reglene på følgende måte:

```
crl [ikke-determ-p1] :
< O : Id | Pr: (P [] P') ; R, Lvar: L, Att: A >
< q( O ) : QId | Ev: M >
=>
< O : Id | Pr: P ; R >
< q( O ) : QId | >
if evalG(P, (L , A), M) .

crl [ikke-determ-p2] :
< O : Id | Pr: (P [] P') ; R, Lvar: L, Att: A >
< q( O ) : QId | Ev: M >
=>
< O : Id | Pr: P' ; R >
< q( O ) : QId | >
if evalG(P', (L , A), M) .

crl [ikke-determ-ingen] :
< O : Id | Pr: (P [] P') ; R, PrQ: W, Lvar: L, Att: A >
< q( O ) : QId | Ev: M >
=>
< O : Id | Pr: empty, PrQ: W : ((P [] P') ; R, L), Lvar: no >
< q( O ) : QId | >
if not evalG(P [] P', (L , A), M) .
```

Reglene undersøker om vakten er sann ved funksjonen evalG, og vakten utføres først i neste steg av vakt-reglene. Det nevnes ingen vakt eksplisitt i regelen, fordi vi ønsker å tillate ikke-determinisme uten vakter.

Det er mulig å slå sammen de tre reglene over med if-else-setninger, men da vil ikke-determinismen bli mer deterministisk ved at g1 alltid velges over g2. Med Maudes nåværende strategi, Round Robin, vil g1 velges foran g2 uansett, så man kan spørre seg om fordelen med tre regler i stedet for én. Poenget er at vi med reglene over kan innføre en ny strategi som gjør valget mer ikke-deterministisk, noe som ikke er mulig med if-else-setningen. Ved bruk av Maudes søkefunksjon vil det samme være tilfelle.

4.3.6 Kvasi-parallellitet

I delkapittel 2.1.4 har vi sett at kvasi-parallellitet $P||P'$ defineres som syntaktisk sukker for $P;P'[]\,P';P$. Vi kan ha en regel som gjør denne

oversettelsen. Her må det presiseres at det er snakk om parallellitet *innad i* objektet, ikke *mellom* objektene.

```
rl [kvasi-parallell] :
< O : Id | Pr: (P || P') ; R >
=>
< O : Id | Pr: (P ; P' [] P' ; P) ; R >
```

Videre behandling vil reglene for ikke-determinisme ta seg av.

For å kunne interpretere at to programsetninger eksekveres i vilkårlig rekkefølge vil oversettelsen ikke alltid virke tilfredsstillende. Ved tilfellet med ikke-terminerende kode vil programsetning nummer to aldri få slippe til. Dette kan løses ved at den ene programsetningen plasseres på vent i PrQ. Vi overlater til Maude å velge hvilken metode som skal utføres først, reglene blir som følger:

```
rl [parallell1] :
< O : Id | Pr: (P || P') ; R, PrQ: W, Lvar: L >
=>
< O : Id | Pr: P, PrQ: (P' ; R, L) : W > .

rl [parallell2] :
< O : Id | Pr: (P || P') ; R, PrQ: W, Lvar: L >
=>
< O : Id | Pr: P', PrQ: (P ; R, L) : W > .
```

Ved å la en av metodene vente i PrQ vil den ha mulighet for å slippe til selv om den andre metoden er ikke-terminerende, vel å merke hvis den første metoden inneholder et eller flere avløsningspunkter. Her har vi utsatt sjekk av vakter til neste steg siden begge uansett skal bli utført.

Dessverre holder ikke denne metoden mål. Siden vi må ha med de lokale variablene i paret som settes på køen får vi to kopier av de lokale variablene. Hvis ingen av variablene er relatert til det som gjøres i koden vil dette fungere fint. Hvis det derimot gjøres noe med variablene i koden vil vi da kunne få to forskjellige verdier på samme variable, noe som ikke holder i forhold til definisjonen av Creol. Vi må derfor bruke første definerte regel, og være oppmerksom på hvordan parallelloperatoren kan oppføre seg, for å unngå problemet med ikke-terminerende kode.

Kapittel 5

Eksempler

Vi vil i dette kapittelet se på flere velkjente eksempler. For hvert eksempel skal vi se nærmere på oversettelse til CMC, eksekvering i Maude og diverse søk.

5.1 Fakultet

Vi starter med et enkelt program for å teste rekursjon og lokale kall. Programmet skal beregne fakultet av et tall som gis med ved oppstart.

5.1.1 Klasse

```
class Fakultet(beregn: int)
begin
  var fakultet: int

  op init == fakultet := 1 .
  op run == fac(beregn ; fakultet) .
  op fac(n: int out f: int) ==
          if (n > 2) then
             fac(n-1 ; f); f := n * f
          else f := n fi .
end
```

Metoden fac beregner n * (n-1)! og kalles rekursivt til n er 2. Vi har ikke tatt med grensesnitt. Programmet er så enkelt og lite at grensesnittet ville vært tomt.

Hvis klassen over faktisk skulle vært i bruk ville det være naturlig å se det i en større samenheng. Metoden fac burde kunne kalles utenfra, med flere forskjellige verdier. I denne sammenheng er det naturlig å gjøre

eksempelet så enkelt som mulig og å se svaret ut i fra resultatet Maude kommer med.

Vi kan også ha en variant av programmet med asynkrone kall. Siden vi ikke kan fortsette metoden fac uten returverdien får vi ikke utført noe imellom kallet og returen. Vi endrer kallet $fac(n-1;f);\ldots$ til $la\,!fac(n-1)\,;\,la\,?(f) \longrightarrow \ldots$. Vi er nødt til å ha en metodereturvakt, ikke metoderetursetning, etter kallet. Hvis vi venter aktivt på returen får ikke objektet mulighet til å utføre metodekallet siden dette lastes inn i køen PrQ (se delkapittel 2.1.4).

5.1.2 Oversettelse til CMC

Ved å følge oversettelsesreglene i delkapittel 4.1 kan vi håndoversette koden til CMC. Den blir da:

```
< 'Fakultet : Cl |
  Att: ('beregn : null), ('fakultet : null),
  Init: ('fakultet := int(1)) ; 'run(nil ; nil), no,
  Mtds:
  < 'fac : Mtdname |
    Latt: ('label : null), ('caller : null),
          ('n : null), 'f : null,
    Code: if 'n > int(2) th
            ('fac('n - int(1) ; 'f)) ;
            'f := ('n * 'f)
          el 'f := 'n fi ;
          end ( 'f)
  > *
  < 'run :  Mtdname |
    Latt: ('label : null), ('caller : null),
    Code: 'fac('beregn ; 'fakultet) >,
  Ocnt: 0.0
>
```

Vi minner om at *no* står for tom variabelliste, * er operatoren for sammensetting av metoder, og at $'label$ og $'caller$ alltid står først i variabellisten til andre metoder enn *Init*. Merk at gruppering ved parenteser er nødvendig for at Maude skal kunne parsere koden. I variabellister og setningssekvenser trenger vi parentes rundt hvert element bortsett fra det siste. Vi trenger ikke parentes hvis elementet står alene.

5.1.3 Eksekvering

Ved kjøring av programmet ser vi at fakultet blir satt til korrekt sum. Man må selvfølgelig opprette et objekt av klassen, ved

new 'Fakultet(int(*tall*)), for å få vite *tall* !. Hvis vi gjør en omskriving ved **rew**, der konfig er oversettelsen over, får vi resultatet:

```
rew konfig (new 'Fakultet(int(5))) .

rewrites: 978 in 10ms cpu (0ms real)
(97800 rewrites/second)
result Configuration:
< q('Fakultet0.0) : QId | Ev: none >
< 'Fakultet : Cl |
   Att: ('beregn : null),'fakultet : null,
   Init: ('fakultet := int(1)) ; 'run(nil ; nil), no,
   Mtds:
   < 'fac : Mtdname |
      Latt: ('label : null),('caller : null),
            ('n : null),'f : null,
      Code: if 'n > int(2) th
              ('fac('n - int(1) ; 'f)) ;
               'f := ('n * 'f)
            el 'f := 'n fi ; end('f) > *
   < 'run : Mtdname |
      Latt: ('label : null),'caller : null,
      Code: ('fac('beregn ; 'fakultet)) ; end(nil) >,
   Ocnt: 1.0 >
< 'Fakultet0.0 : Id |
   Cl: 'Fakultet,Pr: empty,
   PrQ: none, Lvar: no,
   Att: ('beregn : int(5)),'fakultet : int(120),
   Lcnt: 6 >
```

Den interessante delen av resultatet ser vi i objektet, nærmere bestemt i objektvariablene. Ved å følge omskrivingen trinn for trinn kan vi se at de lokale kallene oppfører seg som forventet og at avslutningen av synkrone kall går uproblematisk.

5.1.4 Søk

Et naturlig søk for et program som er ment å være deterministisk og terminerende, er antall sluttilstander det er mulig å oppnå. Da får vi vite om programmet faktisk er deterministisk for gitt input. Søket blir dermed

$$\textbf{search}\ c_s\ \text{new}\ 'Fakultet(int(5))\ =>!\ C : Configuration\ .$$

hvor c_s er konfigurasjonen med CMC-kode for programmet med synkront kall, og C er en variabel som kan gjenkjenne alle mulige sluttkonfigurasjoner. Resultatet ble at Maude-maskinen kun fant en sluttilstand:

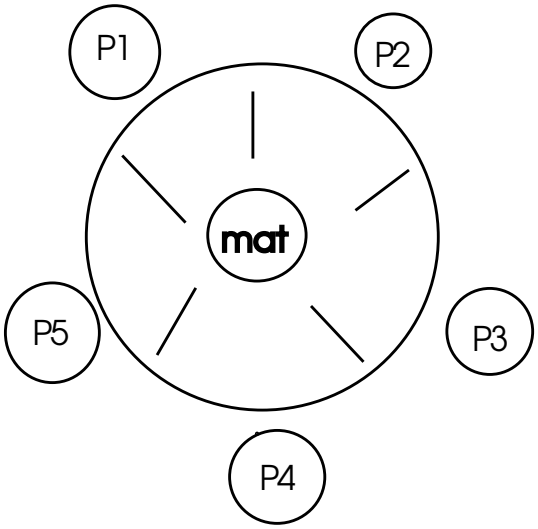
```
Solution 1 (state 43)
states: 44  rewrites: 996 in 10ms cpu (0ms real)
(99600 rewrites/second)
C:Configuration -->
< q('Fakultet0.0) : QId | Ev: none >
< 'Fakultet : Cl |
   Att: ('beregn : null),'fakultet : null,
   Init: ('fakultet := int(1)) ; 'run(nil ; nil),no,
   Mtds:
   < 'run : Mtdname |
      Latt: ('label : null),'caller : null,
      Code: ('fac('beregn ; 'fakultet)) ; end(nil) > *
   < 'fac : Mtdname |
      Latt: ('label : null),('caller : null),
            ('n : null),'f : null,
      Code: if 'n > int(2) th
              ('fac('n - int(1) ; 'f)) ;
               'f := ('n * 'f)
            el 'f := 'n fi ; end('f) >,
   Ocnt: 1.0 >
< 'Fakultet0.0 : Id |
   Cl: 'Fakultet, Pr: empty,
   PrQ: none,Lvar: no,
   Att: ('beregn : int(5)),'fakultet : int(120),
   Lcnt: 6 >

No more solutions.
```

Vi kan dermed si at programmet er korrekt for denne starttilstanden, siden vi kun fant én korrekt sluttilstand. Ved å erstatte de synkrone kallene med asynkrone kall får man det samme resultatet, men med litt flere omskrivinger.

5.2 De Spisende Filosofer

I 1965 formulerte og løste Dijkstra et synkroniseringsproblem kalt "De Spisende Filosofer". Dette problemet er mye brukt til å vise hvordan man kan oppnå eller unngå vranglås eller utsulting.

Rundt et bord sitter fem filosofer. De veksler mellom å tenke og spise. Midt på bordet står en bolle mat, og mellom hver filosof er det en spisepinne. For å kunne spise trenger hver filosof både venstre og høyre spisepinne. I vår beskrivelse ligger spisepinnen til høyre for hver filosof i filosofobjektet. For å kunne få i seg mat er de dermed avhengige av å spørre sidemannen til venstre om lån av spisepinne. Her er et eksempel på en enkel, ikke-deterministisk algoritme, som dessverre ikke forhindrer filosofene fra å sulte i hjel.



Figur 5.1: Filosofene rundt bordet, med hver sin spisepinne

5.2.1 Grensesnitt

En filosof skal, når han blir sulten, prøve å få en spisepinne fra venstre sidemann i tillegg til sin egen. Ved hjelp av metoden borrowStick sjekker han om naboens spisepinnen er ledig. Han legger spisepinnene tilbake ved metoden returnStick når han er ferdig å spise.

Butleren plasserer hver filosof ved bordet og gir ham navnet på naboen til venstre ved metoden getNeighbor.

```
interface Phil
begin
  with Phil
    op borrowStick
    op returnStick
end
```

```
interface Butler
begin
  with Phil
    op getNeighbor(out n:Phil)
end
```

Metodene definert i grensesnittene er alle tilgjengelig for objekter som implementerer grensesnittet *Phil*. Grunnen til at grensesnittet *Phil* gjør metoder tilgjengelige for "seg selv" er at disse to metodene skal være tilgjengelig for andre objekter med samme grensesnitt. Metodene borrowStick og returnStick skal jo være tilgjengelig for nabofilosofen.

5.2.2 Klasser

Klassen *Philosopher* inneholder metodene grensesnittet *Phil* definerte, i tillegg til metoder kun tilgjengelige for seg selv: think, eat og digest.

```
class Philosopher(butler: Butler)
  implements Phil
begin
  var hungry: bool, chopstick: bool,
      neighbor: Phil, history: string

  op init == chopstick := true; hungry := false;
             history := "";
             butler.getNeighbor(;neighbor) .
  op run == true  ⟶ !think || true  ⟶ !eat || true  ⟶ !digest .
  op think == not hungry ⟶ <thinking...>; wait  ⟶ !think .
  op eat == var l : label;
             hungry ⟶ l!neighbor.borrowStick;
             (chopstick ∧ l?()) ⟶ <eating...>; hungry := false;
             !neighbor.returnStick; wait  ⟶ !eat .
  op digest == wait  ⟶ (hungry := true; wait  ⟶ !digest) .

 with Phil
  op borrowStick == chopstick ⟶ chopstick := false .
  op returnStick == chopstick := true .
end
```

Vi benytter oss av kvasi-parallellitet i metoden run. Siden de tre metodene som kalles er rekursive og ikke-terminerende er vi nødt til å gjøre asynkrone kall for at alle tre skal slippe til. Da lastes hver av metodene inn i PrQ, og det er opp til Maude-maskinen å velge hvilken som skal utføres.

For å lettere se hva som skjer ved eksekvering legger vi til en tekststreng history som bygges opp tegnvis ved en "t" i metoden think, en "e" i eat og en "d" i digest. I en tilstand hvor history har verdien "tde" betyr dette at filosofen har tenkt, blitt sulten og spist en gang hver.

Butler-klassen oppretter fem filosofer, for så å vente på forespørsler om nabonavn fra filosofene.

```
class Butler
  implements Butler
begin
  var p1: Phil, p2: Phil, p3: Phil,
          p4: Phil, p5:Phil

  op init ==
    p1 := new Phil(this); p2 := new Phil(this);
```

```
    p3 := new Phil(this); p4 := new Phil(this);
    p5 := new Phil(this) .
  with Phil
    op getNeighbor(out n) ==
       if caller = p1 then n := p2
       else if caller = p2 then n := p3
       else if caller = p3 then n := p4
       else if caller = p4 then n := p5
       else n := p1 fi fi fi fi .
end
```

Vi ser at *Butler* er en passiv klasse, siden den ikke inneholder en run-metode.

5.2.3 Oversettelse til CMC

Vi kan igjen oversette ved å bruke reglene for oversettelse, og får da:

```
< 'Philosopher : Cl |
   Att: ('butler : null), ('hungry : null),
        ('chopstick : null), ('neighbor : null),
        ('history : null),
   Init: ('chopstick := bool(true)) ;
         ('hungry := bool(false)) ;
         ('history := str(""))
         ('label ! 'butler . 'getNeighbor(nil)) ;
         ('label ? ('neighbor)) ; 'run(nil ; nil),
         ('label : null),
   Mtds: < 'think : Mtdname |
            Latt: ('caller : null), ('label : null),
            Code: not 'hungry -->
                  ('history := ('history cat str("t"))) ;
                  wait --> ! 'think(nil) ; end(nil) > *
         < 'eat : Mtdname |
            Latt: ('caller : null), ('label : null),
                  ('l : null),
            Code: 'hungry -->
                  ('l ! 'neighbor . 'borrowStick(nil)) ;
                  ('chopstick & ('l ? (nil))) -->
                  ('history := ('history cat str("e"))) ;
                  ('hungry := bool(false)) ;
                  (! 'neighbor . 'returnStick(nil) ) ;
                  wait --> ! 'eat(nil) ; end(nil) > *
         < 'digest : Mtdname |
            Latt: ('caller : null), ('label : null),
            Code: wait --> ('hungry := bool(true)) ;
```

```
                ('history := ('history cat str("c"))) ;
                wait --> (! 'digest(nil)) ; end(nil) > *
        < 'borrowStick : Mtdname |
           Latt: ('caller : null), ('label : null),
           Code: 'chopstick --> ('chopstick := bool(false)) ;
                  end(nil) > *
        < 'returnStick : Mtdname |
           Latt: ('caller : null), ('label : null),
           Code: ('chopstick := bool(true)) ; end(nil) > *
        < 'run : Mtdname |
           Latt: ('caller : null), ('label : null),
           Code: (bool(true) --> ! 'think(nil)) ||
                 (bool(true) --> ! 'eat(nil)) ||
                 (bool(true) --> ! 'digest(nil))
                 ; end(nil) >,
   Ocnt: 0.0
>
```

Siden vi ikke kommer til å se på *Butler*-klassen videre i eksekveringen utelater vi oversettelsen av denne her.

Vi vil i alle kjøringer og søk under ha en starttilstand bestående av oversettelsen til CMC for klassene *Philosopher* og *Butler*, og en melding om opprettelse av et Butlerobjekt.

5.2.4 Eksekvering

Ved forsøk på eksekvering med **rew** vil bare den første filosofen slippe til. Maude-maskinen vil forsøke å utføre mest mulig på ett objekt før den slipper til andre. Fordi metoden digest alltid kan utføres vil ingen andre slippe til. Den første filosofen vil tenke én gang og deretter bare bli mer og mer sulten. Ved omskriving i 500 steg vil filosofobjektet se slik ut:

```
< 'Philosopher0.0 : Id |
    Cl: 'Philosopher, Pr: empty,
    PrQ: ((nothing --> ! 'digest(nil) ; end(nil)),
         ('label : int(46)),'caller : null) :
         ((not 'hungry -->
          ('history := ('history cat str("t"))) ;
          wait --> ! 'think(nil) ; end(nil)),
         ('label : int(7)),'caller : null) :
         (('chopstick & ('l ?(nil))) -->
         ('history := ('history cat str("e"))) ;
         ('hungry := bool(false)) ;
          ! 'neighbor . 'returnStick(nil) ;
          wait --> ! 'eat(nil) ; end(nil)),
        ('label : int(4)),('caller : null),'l : int(6),
    Lvar: no,
    Att: ('butler : 'Butler0.0),('hungry : bool(true)),
```

```
        ('chopstick : bool(true)),
        ('neighbor : 'Philosopher1.0),
         'history : str("tddddddddddddddddddddd
                        ddddddddddddddddddddd"),
    Lcnt: 47 >
```

Vi ser av historien at filosofen kun kan utføre metoden digest. Fordi denne metoden er klar konstant gir ikke Maude-maskinen slipp på det første filosofobjektet og naboen får aldri slippe til. Dermed får han heller ikke svart på om spisepinnen er ledig.

Med **frew** er fordelingen mellom filosofene jevnere. Historien til den første filosofen blir da "tddetttt". Hver av filosofene vil tenke, bli sulten og spise et par ganger, for så bare å tenke. Grunnen til at en metode velges mange ganger fremfor de andre kommer av Maudes interne vekting ved multisett, bygget opp med en konstruktør som er assosiativ, kommutativ og med en minste identitet. Uansett om metodene settes inn foran, bak eller i midten resulterer Maudes sortering i at samme metode velges om og om igjen. Vi skal derfor se på andre måter å definere køen på i kapittel 6.

5.2.5 Søk

Med et ikke-terminerende program nytter det ikke å søke etter slut-tilstander. Her kan det være interessant å se om det finnes kjøringer der alle filosofene får spist, hvor bare en får spist, eller der vi får en vranglås. Med vranglås mener vi at filosofene hverken får spist eller tenkt, men bare blir mer og mer sultne, siden metoden digest alltid kan brukes. Vi forsøker å gjøre et søk for å se om vi finner en slik tilstand. Alle filosofene vil være sultne, og ingen vil ha tilgang til spisepinnen sin. Søket blir dermed:

```
search [1] konfig (new 'Butler(nil)) =>*
< P:Oid : Id | Cl: 'Philosopher,
Pr: P:GuardProg, PrQ: W:MProg,
Lvar: L:ListQidVal,
Att: ('butler : S:Oid), ('hungry : bool(true)),
('history : D:Data), ('chopstick : bool(false)),
('neighbor : O:Oid) > ...
C:Configuration  .
```

der ... står for de resterende filosofobjektene, og de oversatte klassene befinner seg i konfig. Dette resulterer i alt for mange tilstander som må sjekkes, og Maude-maskinen går rett og slett tom for minne. Maudes søkefunksjon gjør et bredde-først søk i tilstandstreet, og må dermed ta vare på alle tilstander den har vært innom så langt. Svaret blir derfor

"Aborted". Vi kan med dette svaret *tro*, med en viss rimelighet, at en vranglås ikke finnes, men vi kan ikke *vite* det med sikkerhet. Det er derfor ønskelig å forsøke andre tilnærminger.

For å unngå at Maude går tom for minne kan vi dele opp søket i flere deler. Vi vil da lete etter en tilstand som kan føre til vranglås, men som er enklere å oppnå, for så å søke videre mot vranglås fra den tilstanden. Vi kan for eksempel forsøke å finne tilstanden over for en og en filosof. Det fungerer dessverre like dårlig som å finne tilstanden for alle filosofene. Vi kan oppnå færre tilstander ved å fjerne historien, og nøye oss med å undersøke for tre filosofer, men heller ikke dette hjelper. Vi skal derfor se på en litt mer finkornet oppdeling.

For å komme i vranglås må alle filosofene sette sin spisepinne til opptatt, samtidig som de venter på å få låne naboens spisepinne. Metoden eat er den eneste som har en sammensatt vakt i seg. Vi kan dermed identifisere metoden kun ved å se etter denne vakten. Det samme gjelder metoden borrowStick, som er den eneste med 'chopstick som vakt. Vi starter med å søke etter en filosof som skal til å utføre borrowStick:

```
search [1] konfig =>*
< O:Oid : Id | Cl: 'Philosopher,
Pr: 'chopstick --> P:GuardProg
PrQ: (('chopstick & ('l ? (nil))) -->
P1:GuardProg, L1:ListQidVal) : W0:MProg,
Lvar: L0:ListQidVal,
Att: ('butler : S0:Oid), ('hungry : D:Data),
('chopstick :  D':Data), ('neighbor : O0:Oid) >
C:Configuration .
```

Vi får da en tilstand som passer med mønsteret over, og i tillegg resten av konfigurasjonen, la oss kalle hele $c1$. Da er det bare å lete videre etter en tilstand med to slike filosofobjekter $c2$, og så tre filosofobjekter fra denne igjen, $c3$. Neste steg blir da å se etter en tilstand hvor alle objektene har satt 'chopstick til false :

```
search [1] c3 =>*
< O:Oid : Id |
Cl: 'Philosopher, Pr: empty,
PrQ: (('chopstick & ('l ? (nil))) -->
P1:GuardProg, L1:ListQidVal) : W0:MProg,
Lvar: L0:ListQidVal, Att: ('butler : S0:Oid),
('hungry : D:Data), ('chopstick :  D':Data),
('neighbor : O0:Oid) > ...
C:Configuration .
```

hvor ... erstattes av de to andre filosofobjektene. Vi får da en tilstand hvor 'chopstick er satt til false og 'hungry til true hos alle filosofene, og

har dermed funnet én mulig vranglås. [1] Det ble, på lignende vis, utført et søk for å finne vranglås blant fem filosofer. Søket måtte da deles opp i åtte deler mot fire med tre filosofer.

5.3 Alternating Bit Protocol

Anta at sender og mottaker er i et nettverk der meldinger kan gå tapt. Vi ønsker likevel å sikre at alle meldinger kommer frem, og blir mottatt i riktig rekkefølge. Løsningen på problemet blir å si fra om mottatte meldinger, og å sende meldingen, samt beskjed om mottatt melding, flere ganger. Sender fortsetter å sende samme melding til beskjed om at den er mottatt har kommet, og mottaker fortsetter å sende beskjed om mottatt melding til han får en ny. Vi antar at et nettverk vil slippe meldingen gjennom hvis den sendes mange nok ganger, og at linken bevarer rekkefølge.

5.3.1 Grensesnitt

Sender og mottaker skal kommunisere gjennom en link. Vi ser på linken som en liste som bevarer rekkefølgen, der man setter inn bakerst og tar ut først.

```
interface Link [E: Element]
begin
  with any
    op get(out elem: Element)
    op put(elem: Element)
end
```

Grensesnittet *Link* tilbyr metodene put og get til omverdenen.

Grensesnittene for *Sender* og *Receiver* tilbyr ingen metoder til omverdenen og er derfor tomme. Ideelt sett burde de tilby metoder for å gi *Sender* meldinger og hente meldinger ut av *Receiver*.

```
interface Sender        interface Receiver
begin                   begin
end                     end
```

[1] Søkene ble gjort før etiketter ble innført. Det er en mulighet for at tilstedeværelsen av etiketter vanskeliggjør et slikt søk fordi det gir opphav til én ny tilstand per etikettnummer for tilstander som tidligere var identiske. Det blir da veldig mange flere tilstander å søke gjennom, og en vranglås kan bli vanskeligere å finne, selv om den fortsatt er der. Mistanken er ikke undersøkt.

Vi velger allikevel å ta med grensesnittene siden de brukes til typingsformål. All typing er gjort ved grensesnitt, men siden vi ikke har noen restriksjoner på disse grensesnittene kunne vi også brukt supergrensesnittet **any**.

5.3.2 Klasser

For å sette i gang meldingsutvekslingen har vi en klasse *Start* som oppretter linker, sender og mottaker.

```
class Start (sendList: list)
begin
  type Pair == Data, bool
  var ls, lr: Link,
        r: Receiver, s: Sender
  op init ==
    ls := new Link [Pair];
    lr := new Link [bool];
    r := new Receiver (ls, lr);
    s := new Sender (ls, lr, sendList) .
end
```

Linkene ls og lr skal henholdsvis være linker fra sender og fra mottaker.

Klassen *Link* skal implementere funksjonene deklarert i grensesnittet. Vi ser for oss at en liste i Creol har samme type funksjoner som Standard ML [19], det vil si head, tail og concat, som henholdsvis tar ut det første elementet i listen, tar ut alle elementene i listen bortsett fra det første, og setter sammen to lister. Vi implementerer funksjonene i Maude, som følger.

```
eq head(list(D I)) = D .

eq tail(list(D I)) = list(I) .

eq concat(list(I), list(J)) = list(I J) .
```

hvor D er av typen $Data$, og I og J er av typen $List$, det vil si lister med mellomrom som skilletegn.

Klassen *Link* kan da bruke funksjonene som om de var dens egne, men de må bevares tekstlig slik de står nå, i oversettelse til CMC.

For at Link skal modellere upålitelig overføring av meldinger implementerer vi to metoder, lose og duplicate, som henholdsvis mister og dupliserer en melding. Vi ønsker at disse skal kalles ikke-deterministisk, og implementerer run slik at enten ingen eller en av metodene kalles.

```
class Link [E: Element]
  implements Link [E]
begin
  var msgList: list

  op init == msgList := nil .
  op run ==
    while true do
      wait  ⟶ skip  □ (wait  ⟶ ! duplicate  □ wait  ⟶ ! lose)
    od .
  op duplicate ==
    if msgList /= nil then
      msgList := concat(head(msgList), msgList)
    fi .
  op lose ==
    if msgList /= nil then
      msgList := tail(msgList)
    fi .
  with any
    op get(out elem: Element) ==
      msgList /= nil  ⟶
      elem := head(msgList); msgList := tail(msgList) .
    op put(elem: Element) ==
        msgList := concat(msgList, elem) .
end
```

Her vil metoden duplicate duplisere første melding i linken og lose vil miste den første. Det kunne ha vært mer naturlig å miste eller duplisere en vilkårlig melding, men det blir også mer komplisert siden vi ikke har mønstergjenkjenning i Creol . Siden vi kan miste en melding må metoden get vente til meldinglisten inneholder noe før den kan hente ut en melding.

Klassen *Sender* trenger metoder for å sende en melding gjentatte ganger, motta bekreftelser og produsere ny melding.

```
class Sender (linkSend: Link [Data, bool], linkRec: Link [bool], sendq: list)
  implements Sender
begin
  var cnt: bool, sendbuff: Data
  op init == cnt := false .
  op run == produce
    while true do
      send □ rec-ack
    od
```

```
  op produce ==
    if sendq /= nil and sendbuff = null then
      sendbuff := head(sendq);
      sendq := tail(sendq);
      cnt := not cnt;
    fi .
  op send ==
    if sendbuff /= null then
      ! linkSend.put((sendbuff, cnt))
    fi .
  op rec-ack ==
    var ackcnt: bool
    linkRec.get(;ackcnt)
    if ackcnt = cnt then
      sendbuff := null; produce
    fi .
end
```

Klassen *Receiver* trenger metoder for å motta meldinger fra *Sender*, og for å sende bekreftelse på mottatt melding. Vi kombinerer begge disse oppgavene i en metode.

```
class Receiver (linkSend: Link [Data, bool], linkRec: Link [bool])
  implements Receiver
begin
  var recq: list, reccnt: bool
  op init == reccnt := false .
  op run == while true do receive od .
  op receive ==
    var elem: Data, x: bool
    linkSend.get(;(elem, x));
    if x = not reccnt then
      recq := concat(recq, elem);
      reccnt := x
    fi
    ! linkRec.put(reccnt) .
end
```

5.3.3 Oversettelse til CMC

Vi bruker oversettelsesreglene og får da:

```
*** LINK ****
< 'Link : Cl |
   Att: 'msgList : null,
   Init: ('msgList := list(nil)) ; 'run(nil ; nil), no,
   Mtds: < 'get : Mtdname |
            Latt: ('label : null), ('caller : null),
                  'elem : null,
            Code: 'msgList /= list(nil) -->
                  ('elem := head('msgList)) ;
                  ('msgList := tail('msgList)) ;
                  end('elem) > *
         < 'put : Mtdname |
            Latt: ('label : null), ('caller : null),
                  ('elem : null),
            Code: ('msgList := concat('msgList, list('elem)))
                   ; end(nil) > *
         < 'lose : Mtdname |
            Latt: ('label : null), ('caller : null),
            Code:  if 'msgList /= list(nil) th
                   'msgList := tail('msgList) fi
                   ; end(nil) > *
         < 'duplicate : Mtdname |
            Latt: ('label : null), ('caller : null),
            Code: if 'msgList /= list(nil) th
                   'msgList :=
                    concat(head('msgList), 'msgList) fi
                    ; end(nil) > *
         < 'run : Mtdname |
            Latt: ('label : null), ('caller : null),
            Code: while bool(true) do
                  (wait --> empty) []
                  ((wait --> ! 'duplicate(nil)) []
                   (wait --> ! 'lose(nil))) od  ; end(nil) >,
   Ocnt: 0.0
>

*** SENDER ***
< 'Sender : Cl |
   Att: ('linkSend : null), ('linkRec : null),
        ('sendq : null), ('cnt : null), ('sendbuff : null),
   Init: ('cnt := bool(false)) ; 'run(nil ; nil) , no,
   Mtds: < 'run : Mtdname |
            Latt: ('label : null), ('caller : null),
            Code: ('produce(nil ; nil)) ;  while bool(true) do
                  ('send(nil ; nil)) [] 'rec-ack(nil ; nil) od
                   ; end(nil) > *
         < 'produce : Mtdname |
            Latt: ('label : null), ('caller : null),
            Code: if ('sendq /= list(nil)) and
```

```
                    ('sendbuff = null) th
                  ('sendbuff := head('sendq)) ;
                  ('sendq := tail('sendq)) ;
                   'cnt := not 'cnt fi ; end(nil)  > *
         < 'send : Mtdname |
            Latt: ('label : null), ('caller : null),
            Code: if 'sendbuff /= null th
                  (! 'linkSend . 'put (('sendbuff , 'cnt))) fi
                  ; end(nil) > *
         < 'rec-ack : Mtdname |
            Latt: ('label : null), ('caller : null),
                  ('ackcnt : null), ('la : null),
            Code: ('la ! 'linkRec . 'get (nil)) ;
                  ('la ? ('ackcnt)) ;
                   if 'ackcnt = 'cnt th ('sendbuff := null) ;
                      'produce (nil ; nil) fi ; end(nil) >,
   Ocnt: 0.0
>

*** START ***

< 'Start : Cl |
   Att: ('sendList : null), ('ls : null), ('lr : null),
('r : null), ('s : null),
   Init: (('ls := new 'Link(nil)) ;
         ('lr := new 'Link(nil)) ;
         ('s := new 'Sender('ls 'lr 'sendList)) ;
         ('r := new 'Receiver('ls 'lr)), no),
   Mtds: none,
   Ocnt: 0.0
>
```

Receiver blir utelatt herfra da oversettelsen gjøres rett frem. Siden hverken oppretting av typer eller mulighet for type-parametre er implementert i den abstrakte maskinen inkluderer vi en setning for å definere Pair direkte i Maude-koden, ved

```
op _,_ : Data Data -> Data .
```

Det benyttes Data for å gjøre konstruksjonen generell, selv om vi vet at andre parameter i dette tilfelle skal være boolsk.

5.3.4 Eksekvering

Vi ønsker med eksekvering av dette eksempelet å se om alle meldingene kommer frem og i riktig rekkefølge. Siden vanlig omskriving med **rew** favoriserer ett objekt vil den første linken som blir opprettet ta over all prosessorkraft og ingenting skjer. Vi skal derfor kun se på eksekvering med **frew** her. Vi forsøker å gjøre en omskriving med **frew** i 150 steg

hvor vi gir med den oversatte CMC-koden (inkludert *Receiver*) og en opprettelse av *Start* med en liste meldinger:
new 'Start(list(str("a") str("b") str("c"))).

Det resulterer i at linken fra sender til mottaker er slik:

```
< 'Link0.0 : Id | Cl: 'Link,
   Pr: (wait --> empty) ; while bool(true) do
       (wait --> empty)[](wait --> ! 'duplicate(nil))
        []wait --> ! 'lose(nil) od ; end(nil) ; continue(1),
   PrQ: (1 ?(nil),no) :
        ((nothing --> ('label := int(2)) ; ('caller := null) ;
         if 'msgList /= list(nil) th
         'msgList := concat(list(head('msgList)), 'msgList)
         fi ; end(nil),
         ('label : null),'caller : null) :
          ((nothing --> ('label := int(3)) ;
          ('caller := 'Receiver0.0) ;
           'msgList /= list(nil) -->
           'elem := head('msgList) ;
           'msgList := tail('msgList) ; end('elem)),
          ('label : null),('caller : null),'elem : null) :
          ((nothing --> ('label := int(4)) ;
          ('caller := null) ;
          ('elem := (str("a") , bool(true))) ;
          ('msgList := concat('msgList, list('elem))) ;
            end(nil)),('label : null),
          ('caller : null),'elem : null) :
          (nothing --> ('label := int(6)) ;
          ('caller := null) ;
          ('elem := (str("a") , bool(true))) ;
          ('msgList := concat('msgList, list('elem))) ;
            end(nil)),('label : null),
          ('caller : null),'elem : null,
   Lvar: ('label : int(1)),'caller : 'Link0.0,
   Att: 'msgList : list(nil),
   Lcnt: 3
>
```

Vi ser at linken har mottatt flere meldinger fra sender med det første elementet i senderlista. En instans av metoden duplicate Vi ser også at mottakeren har gjort et kall på metoden get.

Første steg på veien til hele senderlista hos mottaker er jo at linken mottar en melding. Vi forsøker å eksekvere i 10 000 steg, men da ligger bare enda flere forsøk på å legge til første element på PrQ.

Fordi det virker som om linken bare utfører while-løkken i metoden run modererer vi linken til å bli sikker, ved å fjerne run. Etter 100 omskrivinger ser sender-linken da slik ut:

```
< 'Link0.0 : Id | Cl: 'Link,
   Pr: empty,PrQ: none,
   Lvar: ('label : int(4)),('caller : null),
         'elem : (str("a") , bool(true)),
   Att: 'msgList : list(str("a") , bool(true)),
   Lcnt: 1
>
```

Her er første melding blitt lagret, og vi ser at et kall på get fra mottakeren ligger på den eksterne køen. Etter 968 omskrivingssteg har mottaker mottatt alle meldingene.

En helt sikker protokoll hvor man ikke kan miste meldinger har ikke behov for å sende meldinger mange ganger, og poenget med skiftende bit går bort. Vi forsøker derfor å implementere en enklere run-metode for linken, hvor det er mulig å miste meldinger:

op run ==
  wait ⟶ ! lose; ! run .

Etter 1000 omskrivingssteg vil mottaker ha mottatt to meldiger, og etter 2040 omskrivingssteg har alle meldingene kommet frem. Vi ser dermed at det fungerer bedre med et rekursivt kall på run, enn med en while-løkke.

5.3.5 Søk

Vi antar ut ifra eksekveringene med den sikre protokollen at det skal være mulig å få frem alle meldingene i riktig rekkefølge også med den usikre protokollen. Ved å bruke søk kan vi belyse dette ytterligere. Vi kan for eksempel undersøke om det finnes en tilstand der alle meldingene er oppført hos mottaker.

```
search [1] konfig
(new 'Start(list(str("a") str("b") str("c")))) =>*
< 'Receiver0.0 : Id |
   Cl: 'Receiver,Pr: P:GuardProg,
   PrQ: MP:MProg,
   Lvar: L:ListQidVal,
   Att: ('linkSend : 'Link0.0),('linkRec : 'Link1.0),
        ('recq : list(str("a") str("b") str("c"))),
         'reccnt : B:Data ,
   Lcnt: N:Nat >
C:Configuration .
```

hvor konfig er klassene oversatt til CMC. Søket genererer alt for mange tilstander for Maude-maskinen og den går tom for minne. Programmet

er høyst ikke-deterministisk, noe som fører til mange tilstander. Vil det hjelpe dersom vi fjerner ikke-determinismen i senderen? Vi forsøker det, og gjør igjen et søk som over. Dette hjelper dessverre lite.

Vi kan i stedet forsøke å dele opp søket. Problemet i eksekveringen over var at sender-linken ikke behandlet metodekallene fra sender. Vi kan søke etter et tilfelle hvor linken er i ferd med å behandle et slikt metodekall for å se om det i det hele tatt er mulig.

```
search [1] konfig
(new 'Start(list(str("a") str("b") str("c")))) =>*
C:Configuration
< 'Link0.0 : Id | Cl: 'Link,
   Pr: ('label := int(N:Nat)) ; ('caller := O:Oid) ;
       ('elem := (S:Data , B:Data)) ;
       ('msgList := concat('msgList, list('elem))) ; end(nil),
PrQ: W:MProg, Lvar: L:ListQidVal,
Att: A:ListQidVal,
Lcnt: N:Nat >
```

Dette søket blir også avsluttet uten svar. Vi har sett at metodeinstansen vi søker etter over har stått på køen PrQ ved eksekvering av programmet og forsøker derfor et søk på dette:

```
search [1] konfig
(new 'Start(list(str("a") str("b") str("c")))) =>*
< 'Link0.0 : Id | Cl: 'Link,
   Pr: empty,
   PrQ: ((nothing --> ('label := int(4)) ; ('caller := null) ;
        ('elem := ('a , bool(true))) ;
        ('msgList := concat('msgList, list('elem))) ; end(nil)),
        ('label : null),('caller : null),'elem : null) :
         W:MProg, Lvar: L:ListQidVal, Att: A:ListQidVal,
   Lcnt: N:Nat >
C:Configuration
```

men heller ikke her får vi resultater. For å undersøke om programmet rett og slett blir for komplekst for søk i Maude prøver vi å gjøre tilsvarende søk i den protokollen med forenklet run-metode. Vi har sett ved vanlig eksekvering at mottaker kan motta alle meldinger, og kjører derfor det første søket om igjen. Dessverre gir heller ikke dette søket svar. Heller ikke med den sikre protokollen terminerer søket normalt. Vi forsøker å lete etter den tilstanden vi så ved normal eksekvering i den sikre protokollen og kjører søket:

```
search [1]  konfig
(new 'Start(list(str("a") str("b") str("c"))))
=>*
```

```
< 'Receiver0.0 : Id | Cl: 'Receiver,
   Pr: if bool(true) th ('receive(nil ; nil)) ;
       while bool(true) do 'receive(nil ; nil) od fi ;
       end(nil) ; continue(1),
   PrQ: 1 ?(nil),no,
   Lvar: ('label : int(1)),'caller : 'Receiver0.0,
   Att: ('linkSend : 'Link0.0),('linkRec : 'Link1.0),
        ('recq : list(str("a") str("b") str("c"))),
         'reccnt : bool(true),
   Lcnt: 46 >
C:Configuration
```

Også dette søket avsluttes uten resultater. Vi må ned til én melding hos mottakeren før søket gir et resultat. Når vi må dele opp i såpass små deler for den sikre protokollen kan vi anta at Maudes søkefunksjon ikke er spesielt godt egnet for komplekse programmer. Hvis vi sammenligner dette programmet med eksempelet for de spisende filosofer ser vi at det her er veldig mye mer ikke-determinisme. Da sier det seg selv at det blir mange tilstander med flere valg for hver while-iterasjon i både linkene og senderen. Men selv når vi fjerner all ikke-determinisme er programmet for komplekst for at søk skal være nyttig.

Kapittel 6

Varianter av den operasjonelle semantikken

I seksjon 5.2 så vi at Maudes multisett ble skjevere fordelt enn vi skulle ønske. Vi skal i dette kapittelet se hvordan vi kan få frem ulike eksekveringer ved å bruke forskjellige kø-ordninger for den interne køen.

6.1 Intern kø med Round Robin-ordning

Vi oppdaget at multisettene ble skjevt fordelt ved å bruke Maudes innebygde assosiativitet og kommutativitet. For å ha mer kontroll over skjevhetene kan vi innføre lister i stedet i den interne køen PrQ. Vi lar listen ha en Round Robin kø-ordning, det vil si at den tar kode ut først, hvis vakten er sann, og setter kode inn bakerst hvis den er falsk.

For å bytte fra multisett til liste er det nok å fjerne det at typen er kommutativ. Vi vil fortsatt ha "none" som identifikator-element, og kan dermed titte et vilkårlig sted i køen ved mønstergjenkjenning, hvis nødvendig. Ved synkrone kall ønsker vi en spesifikk programsetning. Da kan vi, i stedet for å sortere listen, bare hente ut den riktige setningen ved regelen:

```
rl [continue] :
< O : Id | Pr: continue(N),
  PrQ: W : (N ? (J) ; P, L) : W' >
< q( O ) : QId | Ev: M + comp(N, O, K) >
=>
< O : Id | Pr: makeAssignment(J, K) ; P,
  PrQ: W : W', Lvar: L >
< q( O ) : QId | Ev: M > .
```

hvor W og W' enten kan inneholde kodepar, eller være tomme. Endringen består i å legge til W foran setningen, og er helt triviell. Hvis avslutningen har kommet forrest i køen uten å ha blitt hentet ut av en continue-setning må vi flytte den bakerst. Vi behøver da ikke undersøke sannhet, men vet automatisk at avslutningen på det synkrone kallet ikke har funnet sted siden Pr er tom. Regelen blir som følger.

```
rl [synchron_ending_st] :
< O : Id | Pr: empty, PrQ: ((N ? ( J )) ; P, L) : W >
=>
< O : Id | Pr: empty, PrQ: W : ((N ? ( J )) ; P, L) > .
```

Reglene som angår innsetting og uttagning av den interne køen må endres ved bytte til liste. Når noe skal tas ut må det komme fra begynnelsen av listen, når det settes inn noe skal det inn bakerst. Endringen av regler som setter noe inn i PrQ er også triviell. Innsettingen må skje bakerst i køen, der det ikke allerede gjøres. Dersom vakten ikke er sann må programsetningen flyttes bakerst i køen. Hver av uttagningsreglene får dermed en if-else setning på høyre side, som enten flytter koden over i aktiv modus, eller setter den bakerst i listen. Her ser vi et eksempel på en slik regel:

```
rl [boolguard_st] :
< O : Id | Pr: empty, PrQ: (( E --> P ), L) : W, Att: A >
=>
if evalB(E, (A , L)) then
< O : Id | Pr: P, PrQ: W, Lvar: L >
else
< O : Id | PrQ: W : (( E --> P ), L) >
fi .
```

En ulempe med å bruke Round Robin-ordning er at en vakt kan bli sann mens den står midt i listen, men er usann igjen når den kommer først. Da vil den bli flyttet helt bakerst igjen. Dette kan skje om og om igjen, og føre til at programsetningen aldri får slippe til. Vi skal derfor se på en liste med en annen type ordning i 6.2.

6.2 Intern kø med FIFO-ordning

Ved å ha en FIFO-ordning (First In First Out) av listen får vi en determinisme hvor alle programsetningene i køen har like sjanser for å slippe til etter hvert. Jo lenger de har stått i køen, jo lenger frem kommer de.

Vi vil ha de samme endringene som ved Round Robin-ordning med hensyn på typen liste, innsetting i *PrQ* og regelen `continue`. Der forskjellene ligger er ved uttagningsreglene. For å få til en FIFO-ordning må den som kom først inn i lista komme først ut, såfremt vakten er sann. Vi må derfor innføre en ny funksjon, som kan plukke ut vakter fra programsetninger og sjekke om de beregnes til sann eller falsk. Vi kaller funksjonen `trueGuard`. Den har (en del av) den interne køen, variabellister, og den eksterne køen som parametere, og returnerer en boolsk variabel. Finner vi én sann vakt returneres `true`. Hvis vi kommer oss gjennom hele listen uten å finne noen returneres `false`. Implementasjoen blir da som følger:

```
eq trueGuard(none, L, MM) = false .
```

Er køen med programsetninger tom finnes ingen sanne vakter, vi returnerer dermed `false` .

```
eq trueGuard((nothing --> P, L) : MP, L', MM) = true .
```

Hvis vi finner en tom vakt som står alene er den garantert sann, og vi returnerer `true` .

```
eq trueGuard((((E --> P) , L) : MP), L', MM) =
evalB(E, (L, L')) or trueGuard(MP, L', MM) .
```

Ved en boolsk vakt, hvor *E* er et uttrykk, er vakten sann dersom *E* evalueres til sann. Det er også en mulighet for å finne sanne vakter i resten av køen.

```
eq trueGuard(((Q ? (J)) & G --> P, L) : MP, L', MM) =
( inqueue(evalI(Q, L), MM)
  and trueGuard((G --> P, L) : MP, L', MM) )
or trueGuard(MP, L', MM) .
```

En returvakt er sann dersom returen ligger i den eksterne køen. Vi må derfor sjekke det med funksjonen `inqueue`. Siden det kan være en sammensatt vakt, og alle deler av den må være sanne, undersøker vi resten av vakten med `trueGuard`. Som i ligningen over har vi en sjanse for at det finnes en sann vakt i resten av køen.

```
eq trueGuard((wait & G --> P, L) : MP, L', MM) =
trueGuard((G --> P, L) : MP, L', MM) .
```

Siden funksjonen evaluerer vakter i *PrQ* evalueres *wait* til `true` . Vi må derfor evaluere resten av vakten, hvis det er noen. Dersom *G* er falsk vil resten av køen gjennomsøkes.

I tillegg til vaktene over må vi ha med avslutningen for lokale synkrone kall. Den vil alltid være usann siden vi bruker funksjonen trueGuard i tilfeller hvor Pr er tom.

```
eq trueGuard(((N ? ( J )) ; P, L) : MP, L', MM) =
trueGuard(MP, L', MM) .
```

Vi tar dermed bort avslutningen og undersøker resten av køen.

Regelen for uttagning av en boolsk vakt blir da seende ut som følger.

```
crl [bool_guard_st] :
< O : Id | Pr: empty, PrQ: W' : (( E --> P), L') : W,
  Lvar: L, Att: A >
< q( O ) : QId | Ev: M >
=>
< O : Id | Pr: P, PrQ: W' : W,  Lvar: L' >
< q( O ) : QId | >
if evalB(E, (A , L')) and not trueGuard(W', (A, L), M) .
```

En ulempe med denne løsningen er at vi binder opp den eksterne køen i flere regler enn før, siden vi trenger innsyn i denne for å utføre funksjonen trueGuard. For å minimere denne ulempen kunne vi tenke oss å ikke utføre testen på vakter som alltid er sanne i PrQ, som *wait* og den tomme vakten, på en slik måte:

```
rl [emptyguard_st] :
< O : Id | Pr: empty, PrQ: (( nothing --> P), L') : W >
=>
< O : Id | Pr: P, PrQ: W,  Lvar: L' > .
```

Dette er en dårlig idé, fordi vi da kan ende opp med programsetninger med nothing eller wait som vakt midt i listen, men en usann vakt først. Vi må derfor utføre alle uttagningsreglene på den samme måten som vist for boolsk vakt.

For å slippe å binde opp den eksterne køen i regler kan vi se for oss at all sjekking av metodereturvakter skjer ved likhetslogiske ligninger. Siden ligningene utføres mellom hvert omskrivingssteg, og tar null tid, vil dette være mer effektivt, i hvert fall teoretisk sett. Funksjonen kan endre metodereturvakten til en tom vakt, og legge tilordningen av returverdier etter vakten. Vi kan da kun se på vaktene i PrQ og slipper å ta hensyn til den eksterne køen. Dette vil kreve en omlegging av alle regler som omhandler metodereturvakter, og funksjoner som evalG og trueGuard. Vi skal ikke gå nærmere inn på denne løsningen her.

6.3 Flere interne køer

Siden vi har forskjellige typer vakter kan det være naturlig å prioritere forskjellig. Vi kan for eksempel differensiere prioriteten mellom ventevakter og de andre vaktene. En enkel måte å få til en slik prioritering på er å ha to køer, førsteprioriterte kø og andreprioriterte kø. Et objekt får da formen:

$$< Id \mid Cl, Pr, PrQ1, PrQ2, Lvar, Att, Lcnt >$$

De to køene kan være multisett eller lister, og vi følger da tilnærmingen fra henholdsvis kapittel 4.3, seksjon 6.1 eller seksjon 6.2. Vi vil her se på køene som multisett. I tillegg til å legges inn i objektdefinisjonen må kø nummer to legges til i alle de eksisterende reglene.

Man vil ikke kunne hente ut kode fra køen med andreprioritet med mindre den førsteprioriterte køen enten er tom, eller kun inneholder kode med falske vakter. Vi bruker derfor funksjonen definert i seksjon 6.2 trueGuard. Regelen for å hente ut en vakt fra den andreprioriterte listen blir som følger.

```
crl [2pri_st] :
< O : Id | Pr: empty, PrQ1: W, PrQ2: (P, L) : W', Att: A >
< q( O ) : QId | Ev: M >
=>
< O : Id | Pr: P, PrQ2: W', Lvar: L >
< q( O ) : QId | Ev: M >
if not trueGuard(W, A, M) .
```

Siden alle vaktene på køen med andreprioritet er ventevakter som har blitt flyttet over fra aktive Pr vil alle vaktene være sanne. Hvis vi også tar hensyn til at vaktene kan være sammensatte, må vi enten undersøke for alle typer vakter på den andreprioriterte køen, eller utsette testingen av vakten til den ligger på Pr.

Regelen for en ventevakt på Pr blir som følger.

```
rl [waitguard] :
< O : Id | Pr: wait --> P, PrQ2: W, Lvar: L >
=>
< O : Id | Pr: empty, PrQ2: W : (P, L), Lvar: no > .
```

Her fjerner vi rett og slett vakten helt, siden alle programsetningene på denne køen tidligere startet med ventevakter.

Vi minner om at vi fortsatt kan ha ventevakter på køen med førsteprioritet. Hvis en metode starter med en ventevakt, og lastes inn, vil den havne på den første køen.

6.4 Eksekveringer

I denne seksjonen vil vi se på eksekvering av eksempler fra kapittel 5 i de forskjellige variantene over sammenlignet med den originale varianten fra kapittel 4.3.

6.4.1 De spisende filosofer

Vi skal se spesielt på to aspekter ved resultatene vi får etter 500 omskrivingsteg, nemlig historien og antall kall gjort av filosofene i $Lcnt$. Historien representerer det hver enkelt filosof har fått gjennomført. Med antall kall kan vi lettere sammenligne fordelingen mellom filosofene. Det er selvfølgelig en sammenheng mellom disse to. Vi skal først se på eksekveringer med bruk av **rew** (rewrite) og deretter med bruk av **frew** (fair rewrite).

Rewrite

Hvis vi tar et tilbakeblikk på seksjon 5.2.4 og bruker multisett-varianten av PrQ viser kjøringer at den første filosofen har gjort 47 kall. De andre filosofene får ikke slippe til, og den første filosofen får dermed aldri spist.

Ved eksekvering av varianten med Round Robin kø-ordning blir resultatet for den første filosofen som følger.

```
< 'Philosopher0.0 : Id |
    Cl: 'Philosopher,Pr: empty,
    PrQ: ((('chopstick & ('l ?(nil))) -->
         ('history := ('history cat str("e"))) ;
         ('hungry := bool(false)) ;
          ! 'neighbor . 'returnStick(nil) ;
          wait --> ! 'eat(nil) ; end(nil)),
         ('label : int(4)),('caller : null),'l : int(7)) :
         ((nothing --> ('label := int(29)) ;
         ('caller := null) ;
          wait --> ('hungry := bool(true)) ;
         ('history := ('history cat str("d"))) ;
          wait --> ! 'digest(nil) ; end(nil)),
         ('label : null),'caller : null) :
         (not 'hungry -->
         ('history := ('history cat str("t"))) ;
          wait --> ! 'think(nil) ; end(nil)),
         ('label : int(6)), 'caller : null,
    Lvar: ('label : int(28)),'caller : null,
    Att: ('butler : 'Butler0.0),('hungry : bool(true)),
         ('chopstick : bool(true)),
```

```
        ('neighbor : 'Philosopher1.0),
         'history : str("tddddddddddddddddddddddd"),
    Lcnt: 30 >
```

Her er historien vesentlig kortere og kallene færre enn varianten med indre kø som multisett. Grunnen er blant annet at Round Robin-varianten bruker flere omskrivingssteg på å flytte elementer frem og tilbake i den interne køen PrQ. Heller ikke her får de andre filosofene slippe til, så den første filosofen får aldri spist.

Ved eksekvering av varianten med FIFO-ordnet liste er resultatet nøyaktig det samme som en multisett-kø, bortsett fra ordningen i den indre køen.

```
< 'Philosopher0.0 : Id |
    Cl: 'Philosopher,Pr: empty,
    PrQ: ((('chopstick & ('l ?(nil))) -->
         ('history := ('history cat str("e"))) ;
         ('hungry := bool(false)) ;
          ! 'neighbor . 'returnStick(nil) ;
          wait --> ! 'eat(nil) ; end(nil)),
         ('label : int(4)),('caller : null),'l : int(7)) :
         ((not 'hungry -->
         ('history := ('history cat str("t"))) ;
          wait --> ! 'think(nil) ; end(nil)),
         ('label : int(6)),'caller : null) :
         (nothing --> ! 'digest(nil) ; end(nil)),
         ('label : int(46)),'caller : null,
    Lvar: no,
    Att: ('butler : 'Butler0.0),('hungry : bool(true)),
         ('chopstick : bool(true)),
         ('neighbor : 'Philosopher1.0),
          'history : str("tddddddddddddddddddddd
                          ddddddddddddddddddddddd"),
    Lcnt: 47 >
```

Her ser vi at de vaktene som har vært forsøkt gjennomført lengst står først. Det hjelper allikevel lite når de andre filosofene ikke slipper til...

Resultatet for eksekveringen av varianten med to køer skiller seg heller ikke ut i noen stor grad.

```
< 'Philosopher0.0 :
    Id | Cl: 'Philosopher,
    Pr: ('history := ('history cat str("d"))) ;
        wait --> ! 'digest(nil) ; end(nil),
    PrQ1: ((not 'hungry -->
          ('history := ('history cat str("t"))) ;
           wait --> ! 'think(nil) ; end(nil)),
```

```
       ('label : int(7)),'caller : null) :
       (('chopstick & ('l ?(nil))) -->
       ('history := ('history cat str("e"))) ;
       ('hungry := bool(false)) ;
        ! 'neighbor . 'returnStick(nil) ; wait -->
        ! 'eat(nil) ; end(nil)),
       ('label : int(4)),('caller : null),'l : int(6),
PrQ2: none,
Lvar: ('label : int(40)), 'caller : null,
Att: ('butler : 'Butler0.0),('hungry : bool(true)),
     ('chopstick : bool(true)),
     ('neighbor : 'Philosopher1.0),
      'history : str("tddddddddddddddddd
                      ddddddddddddddddd"),
Lcnt: 41 >
```

Vi har her fått utført noen færre kall enn med én multisett-kø, men flere enn ved Round Robin-ordnet liste. Grunnen til at vi får utført færre kall enn ved én multisett kø kan være at vi bruker flere steg ved å ha to køer å velge mellom. Heller ikke her får filosofen spist, da ingen av de andre slipper til. Dette er, som vi har sett, det generelle resultatet for alle variantene ved bruk av **rew**.

Vi endrer koden i eksempelet, slik at filosofen bare får lov til å bli sulten når han ikke allerede er det. Det vil si at vakten i metoden digest endres fra wait til not hungry. Da blir resultatene for alle varianter annerledes. Siden det ikke alltid er en sann vakt for den første filosofen å utføre slipper hans nabo til nok til å låne bort spisepinnen sin. Hvis vi bare ser på historien og antall kall blir resultatene som følger, hvor 1 står for den første filosofens attributter og 2 står for den andre filosofens attributter:

- Multisett
  - historie 1: str("tdedededededededededed")
  - antall kall 1: 44
  - historie 2: str("td")
  - antall kall 2: 7
- Round Robin
  - historie 1: str("td")
  - antall kall 1: 9
  - historie 2: null
  - antall kall 2: 1

- FIFO
  - historie 1: str("tdetdetdetdetdetdetdetd")
  - antall kall 1: 43
  - historie 2: str("td")
  - antall kall 2: 9
- To køer
  - historie 1: str("tdedededededededed")
  - antall kall 1: 41
  - historie 2: str("td")
  - antall kall 2: 9

Vi ser at resultatene her er ganske like for varianten med multisett og to køer. Round Robin og FIFO-variantene skiller seg ut, ved henholdsvis kort historie og én "t" i hver sykel.

I varianten med Round Robin er historien spesielt kort. Siden vi har if-else setninger i reglene som omhandler uthenting av vakter fra PrQ ved Round Robin-ordningen, vil det alltid være noe å gjøre for den første filosofen. Programsetningene flyttes dermed rundt og rundt i den indre køen, uten at vaktene endrer sannhetsverdi av den grunn. Ved bruk av Round Robin kan det derfor være lurt å legge til en strategi for ikke å benytte seg av disse reglene dersom alle vaktene er usanne.

FIFO er den eneste varianten hvor filosofen får tenkt mer enn én gang. Dette skjer fordi listen er ordnet, og at man alltid velger den første med sann vakt.

Grunnen til at de to andre variantene ikke tenker mer enn første gangen er at Maudes multisett vekter `digest`-koden tyngre enn `think`-koden, og den ene blir alltid valgt fremfor den andre.

Fair rewrite

Vi starter med å se på eksekveringen av varianten fra kapittel 4.3. Igjen ser vi på originalversjonen av koden, hvor digest-metoden starter med vakten `wait`. Ved en eksekvering med **frew** i 500 omskrivingssteg får vi følgende resultater:

- Multisett
  - historie 1: str("tdetd")
  - antall kall 1: 13

  - historie 2: str("tde")
  - antall kall 2: 9
  - historie 3: str("td")
  - antall kall 3: 9
  - historie 4: str("td")
  - antall kall 4: 9
  - historie 5: str("t")
  - antall kall 5: 7

Her er antall kall jevnere fordelt mellom filosofene, selv om det fortsatt er første filosof som dominerer, og antallet går jevnt nedover for de andre. Alle filosofene blir opprettet rett etter hverandre, så det er ikke annen grunn til det enn at Maude-maskinen velger skjevt.

Ved eksekvering av varianten med Round Robin-ordningen er antall kall enda jevnere fordelt, og ikke jevnt synkende. Vi ser for eksempel at den femte filosofen har flere kall enn den fjerde.

- Round Robin
  - historie 1: str("tded")
  - antall kall 1: 13
  - historie 2: str("td")
  - antall kall 2: 7
  - historie 3: str("tded")
  - antall kall 3: 13
  - historie 4: str("td")
  - antall kall 4: 9
  - historie 5: str("tded")
  - antall kall 5: 10

Ved eksekvering med FIFO-ordnet liste er fordelingen mellom filosofene mer ujevn. Vi ser her at den første filosofen har 21 kall, mens den femte bare har 6. Antall kall er jevnt synkende.

- FIFO
  - historie 1: str("tdddettddettd")
  - antall kall 1: 21
  - historie 2: str("tddet")
  - antall kall 2: 13

  - historie 3: str("tddet")
  - antall kall 3: 12
  - historie 4: str("t")
  - antall kall 4: 6
  - historie 5: str("")
  - antall kall 5: 6

Ut ifra det vi kan se av denne fordelingen kan vi ønske oss en strategi ved bruk av FIFO-ordning som velger filosof-objektene tilfeldig.

Ved eksekvering av varianten med to køer ser vi en jevnt synkende fordeling, hvor alle filosofene får tenkt og fire av de spist.

- To køer
  - historie 1: str("tdetddet")
  - antall kall 1: 15
  - historie 2: str("tdettt")
  - antall kall 2: 11
  - historie 3: str("tdet")
  - antall kall 3: 10
  - historie 4: str("tde")
  - antall kall 4: 8
  - historie 5: str("td")
  - antall kall 5: 7

Av alle de forskjellige variantene er denne varianten en som både er ganske jevn og hvor filosofene har fått gjort mange kall til sammen. FIFO-varianten har noen flere kall samlet, men historiene er kortere.

Fordi antall kall er jevnere fordelt ved en rettferdig omskriving er det vanskeligere å se en trend ved bare 500 omskrivinger. Vi skal derfor se på kjøringer med 10 000 omskrivingssteg for å se om en generell trend viser seg.

Ved eksekvering av multisettvarianten vil alle filosofene etter en kort stund bare tenke. Dette grunnet skjevt utvalg fra Maude-maskinens side.

- Multisett
  - historie 1: str("tddettttttttttttttttttttttttt...")
  - antall kall 1: 224

  - historie 2: str("tdetddetddetdddettttttttttttt...")
  - antall kall 2: 226
  - historie 3: str("tddettttttttttttttttttttttt...")
  - antall kall 3: 218
  - historie 4: str("tddetdetddettttttttttttttttt...")
  - antall kall 4: 221
  - historie 5: str("tdetddettttttttttttttttttt...")
  - antall kall 5: 214

Det samme skjer ved eksekvering av varianten med to køer, nettopp fordi disse også er multisett.

- To køer
  - historie 1: str("tdetddettttttttttttttt...")
  - antall kall 1: 252
  - historie 2: str("tdettttttttttttttttttt...")
  - antall kall 2: 248
  - historie 3: str("tdettttttttttttttttttt...")
  - antall kall 3: 247
  - historie 4: str("tdettttttttttttttttttt...")
  - antall kall 4: 244
  - historie 5: str("tdett")
  - antall kall 5: 10

Ved eksekvering av Round Robin-varianten ser vi at digest-metoden utføres mange flere ganger enn think og eat til sammen. Vi ser også at antall kall ligger rundt 150, med en differanse på rundt 10 mellom filosofene. Her har filosofene gått i vranglås!

- Round Robin
  - historie 1: str("tdededdetdeddetdetdddetdetdetdeddddetddeddddetddetdeddetdeddetdeddetdededddddddd...")
  - antall kall 1: 157
  - historie 2: str("tdededdetdetdetdddetdetdetdeddddetddeddedddetdeddetdeddetddetddetddddddddd...")
  - antall kall 2: 141

  - historie 3: str("tdeddetdetdeddetdetddetdetddetdddet
    dddetddetddetdetdetdedddetddetdetdetddddddddd...")
  - antall kall 3: 146
  - historie 4: str("tdeddetdedededdetdededdddetdddetddde
    tddeddetddetdeddetdededdetdeddddddddddd...")
  - antall kall 4: 140
  - historie 5: str("tddetddetdetdetddedededededddetddeddd
    edddetdetddetddetddetdeddetddddddddddd...")
  - antall kall 5: 134

Ved eksekvering av FIFO-varianten er think mye oftere utført enn ved Round Robin. De tre første filosofene utfører bare digest mot slutten, men konfigurasjonen er ikke i vranglås for det. Den fjerde filosofen har en ledig spisepinne, men har ikke blitt valgt så han kan låne den bort. Vi ser at fordelingen mellom filosofene er veldig ujevn.

- FIFO
  - historie 1: str("tddetddettdetddettdetdetddettdetddet
    tdetddettddettdetddettdetddettdetdde
    tddettdetddettdetddettdetddettddddddd...")
  - antall kall 1: 451
  - historie 2: str("tddetdetddettdetdetdetdetdetddettde
    tdetddettddddddddddddddddddddddddd...")
  - antall kall 2: 269
  - historie 3: str("tdetdetdetdetdetdetdetdetddddddddddd...")
  - antall kall 3: 180
  - historie 4: str("tddettddettd")
  - antall kall 4: 21
  - historie 5: str("td")
  - antall kall 5: 8

Sammenligning av variantene

Vi ønsker å se på hvor mye filosofene har fått gjort ved vanlig omskriving, **rew**, og i tillegg se på fordelingen mellom filosofene ved omskriving med **frew**.

For den originale versjonen av eksempelet, eksekvert med **rew**, har alle filosofene fått utført like lite fornuftig, så dette er ikke interessant

å sammenligne. Vi ser derfor på fordelingen ved versjonen med den modererte digest-metoden.

Her kommer Round Robin-ordningen klart dårligst ut. FIFO-ordningen er den mest effektive med tanke på fordeling mellom spising og tenking, og sammenlagt antall kall filosofene har gjort.

Ved omskriving i 500 steg med **frew** er FIFO-ordningen og ordningen med to køer de som er mest effektive. FIFO-ordningen har gjort noen flere kall enn to køer, men favoriserer de første filosofene mer. I ordningen med to køer har filosofene tenkt og spist litt mer enn i FIFO-ordningen. Det er derfor vanskelig å si at den ene er klart bedre enn den andre.

Vi tar kjøringene etter 10 000 omskrivingssteg med i betraktningen. Da vil vi, ut ifra fordelingen mellom filosofene og antall ganger de har spist og tenkt, si at Round Robin-ordningen er den beste selv om den har gått i vranglås. Hvis vi kunne fått til en bedre fordeling mellom filosofene i FIFO-ordningen vil den klart komme best ut, både i effektivitet med hensyn på antall kall gjort, og antall ganger spist og tenkt.

6.4.2 Alternating Bit Protocol

Ved dette eksempelet er det interessant å se om protokollen i det hele tatt kommer noen vei mot å få levert meldinger, og hvor mange omskrivingssteg det eventuelt tar å få levert alle. Vi skal derfor se på både den sikre og usikre versjonen av protokollen, og kun gjøre omskrivinger med **frew**.

Hvis vi tar et tilbakeblikk på 5.3.4 og bruker multisett versjonen av PrQ ser vi at sender-linken velger instansen av metoden run om og om igjen. Vi forsøker allikevel å eksekvere den originale protokollen med de forskjellige variantene, i tillegg til den sikre protokollen.

FIFO

Ved eksekvering med FIFO-ordnet liste er det etter 1000 omskrivingssteg mange kopier av den første meldingen i listen til sender-linken. Meldingen har også kommet frem til mottakeren, og er lagret i mottakslisten. Selv om det ligger flere bekreftelser fra mottaker i listen til mottaker-linken har ikke senderen byttet melding. Etter 10 000 omskrivingssteg sendes fortsatt den første meldingen. Dette fører til mistanke om at senderen velger metoden send mye oftere enn han mottar bekreftelser ved metoden rec-ack. Vi forsøker oss derfor med en versjon uten ikke-determinisme i senderen.

Med den nye versjonen ser eksekveringen mer lovende ut. Etter 1000 omskrivingsteg har de to første meldingene kommet frem til mottaker. Alle meldingene kommer frem etter 1454 omskrivingsteg.

```
frewrite [1454] in AB-PROT2 : konfig
   (new 'Start(list(str("a") str("b") str("c"))))  .
rewrites: 18096 in 30ms cpu (30ms real) (603200 rewrites/second)
```

Med den sikre varianten av protokollen, uten ikke-determinisme hverken i linker eller sender er alle meldingene sendt på 758 omskrivingsteg.

```
frewrite [758] in AB-SAFE : konfig
   (new 'Start(list(str("a") str("b") str("c"))))  .
rewrites: 13602 in 20ms cpu (30ms real) (680100 rewrites/second)
```

Round Robin

Ved eksekvering av varianten med Round Robin-ordnet liste er det etter 1000 omskrivingssteg kommet to meldinger til mottaker. Da er det litt overraskende at den andre meldingen fortsatt står i sender-bufferet etter 2000 omskrivingssteg. Det tar 400 omskrivingssteg til før siste melding ligger klar til sending. Da ligger det 21 kopier av melding nummer to i sender-linkens liste, så det tar totalt 6751 omskrivingssteg før alle meldingene har kommet frem til sender.

```
frewrite [6751] in AB-PROT : konfig
   (new 'Start(list(str("a") str("b") str("c")))) .
rewrites: 81597 in 140ms cpu (140ms real) (582835 rewrites/second)
```

Selv om vi her har kommet frem til et svar med den originale protokollen forsøker vi å eksekvere versjonen uten ikke-determinisme i senderen også. Der ser vi, etter 1000 omskrivingssteg to meldinger hos mottaker, og den tredje står klar til sending hos senderen. Etter 1136 steg er alle meldingene ankommet.

```
frewrite [1136] in AB-PROT2 : konfig
   (new 'Start(list(str("a") str("b") str("c")))) .
rewrites: 14503 in 30ms cpu (30ms real) (483433 rewrites/second)
```

Med den sikre versjonen går det, som antatt enda raskere. Kun 713 omskrivingssteg må til før alle meldingene har kommet frem.

```
frewrite [713] in AB-SAFE : konfig
   (new 'Start(list(str("a") str("b") str("c")))) .
rewrites: 12792 in 20ms cpu (20ms real) (639600 rewrites/second)
```

To køer

For dette eksempelet forventes varianten med to køer å være meget vellykket. Store deler av problemene vi hadde i varianten med multisett er at instansen av run ble valgt alt for ofte, men siden det er ventevakt foran alle de tre ikke-deterministiske valgene kommer instansen av run til å havne på køen med andreprioritet. Da får ikke linken duplisert eller mistet meldinger, med mindre det ikke er noe annet å gjøre.

De positive forventningene viser seg å være berettiget. Etter bare 1490 omskrivinger er alle meldingene ankommet mottaker:

```
frewrite [1490] in AB-PROT : konfig
   (new 'Start(list(str("a") str("b") str("c")))) .
rewrites: 17024 in 40ms cpu (30ms real) (425600 rewrites/second)
```

Eksekvering av versjonen uten ikke-determinisme i sender er derimot et overraskende negativt resultat. Det tar hele 1539 omskrivingssteg å få alle meldingene til mottaker.

```
frewrite [1539] in AB-PROT2 : konfig
   (new 'Start(list(str("a") str("b") str("c")))) .
rewrites: 18070 in 30ms cpu (40ms real) (602333 rewrites/second)
```

Den sikre versjonen av protokollen oppfører seg mer som forventet. Der tar det 713 omskrivingssteg før alle meldingene er fremme hos mottaker.

```
frewrite [713] in AB-SAFE : konfig
   (new 'Start(list(str("a") str("b") str("c")))) .
rewrites: 12792 in 20ms cpu (20ms real) (639600 rewrites/second)
```

Sammenligning av variantene

Det er for dette eksempelet klarere tendenser for å sammenligne de forskjellige variantene. Varianten med to køer kom helt klart best ut hvis vi ser på de tre versjonene samlet. Fra å ikke komme frem til noe svar for den originale protokollen i FIFO-ordningen, og ordningen med multisett kommer vi frem til et svar på 40 millisekunder! Vi kan da forvente oss enda bedre resultater ved å bruke to køer med en annen ordning enn multisett.

Hvis vi ser på versjonene av protokollen separat ser vi at Round Robin-ordningen kom best ut ved versjonen uten ikke-determinisme i sender. Varianten med to køer og Round Robin-ordningen har akkurat samme resultat for den sikre protokollen.

Vi kan dermed konkludere med at det, for dette eksempelet, er varianten med to køer som er den mest egnede.

Kapittel 7

Konklusjon

Vi har i denne oppgaven sett på en abstrakt maskin for Creol, og hvordan denne kan bli implementert i Maude. Hovedpunktene i oppgaven er

- en semantisk struktur for den abstrakte maskinen

  Vi gir en generell overordnet struktur for klasser, objekter, metoder og meldinger i kapittel 3.1. Gjennom diskusjoner gjør vi oss opp en formening om hvor de forskjellige attributtene skal befinne seg, og på hvilken form de skal være.

- operasjonell semantikk for Creol

  Vi definerer en operasjonell semantikk for Creol i kapittel 3.2. Disse semantiske reglene skal gi en formell beskrivelse av språkets oppførsel, i tillegg til å gi grunnlag for de omskrivingslogiske reglene som kan implementeres i Maude.

- definisjon av CMC og oversettelse fra Creol til CMC

  Vi definerer CMC, og hvordan en oversettelse fra Creol skal gjøres i kapittel 4.1. Her ser vi også på diskusjoner rundt hvilke valg som er gjort. Definisjonen kan senere brukes for å implementere en oversetter.

- implementasjon ved omskrivingsregler

  I kapittel 4.3 ser vi den operasjonelle semantikken implementert i Maude-syntaktiske omskrivingsregler. Her har vi gjort forsøk med ulike varianter av reglene, diskutert disse, og forkastet uegnede regler. I kapittel 4.2 defineres funksjonene som blir brukt i forbindelse med reglene.

- eksempler i Creol

  Vi har sett flere eksempler på Creol-programmer oversatt til CMC, eksekvert i Maude, og gjort til gjenstand for diverse søk i kapittel

5. Fra resultatene eksemplene gir kan vi se et behov for andre varianter av den operasjonelle semantikken.

- varianter av den operasjonelle semantikken

  I kapittel 6 ser vi på varianter av den operasjonelle semantikken. I fokus for endringene er forskjellige ordninger av den interne køen. De utførte endringene er indre kø med FIFO-ordning og Round Robin-ordning, og to indre køer med multisett-ordning og forskjellig vekting. Vi har sett eksekvering av de større eksemplene fra kapittel 5 i variantene, og sammenlignet variantene for hvert av eksemplene.

For å oppsummere hovedpunktene ytterligere skal vi forsøke å svare på delspørsmålene fra kapittel 1.1.

7.1 Delspørsmål 1

Det første delspørsmålet lød som følger:

> Hvordan kan vi lage en interpret for Creol-programmer i Maude, slik at vi fleksibelt kan eksperimentere med hensyn til semantikk?

Vi har sett en implementasjon av en abstrakt maskin, med struktur og regler for eksekvering av CMC i kapittel 4. Hele den abstrakte maskinen er på 750 linjer kode deriblant 33 omskrivingsregler. Bruken av omskrivingsregler for eksekvering gjør det lett å få oversikt over hva som skjer hvor.

I kapittel 6 vises at vi ved enkle grep kan forandre den operasjonelle semantikken. Derfor mener vi at denne måten å konstruere en interpret på er fleksibel med hensyn til eksperimentering på semantikken.

7.2 Delspørsmål 2

Det andre delspørsmålet lød som følger:

> Hvilken nytte kan vi ha av at Maudes søkeverktøy brukes på Creol-programmer?

Vi ser at søk kan brukes til å bekrefte at et Creol-program fungerer korrekt for én gitt startkonfigurasjon i delkapittel 5.1.4. Dette kan gi oss en pekepinn om at det er mulig å bevise generelle resultater for programmet, med andre verktøy utenfor Maude.

Vi ser hvordan søk kan brukes til å bekrefte mistanker om mulig vranglås fra en gitt startkonfigurasjon i delkapittel 5.2.5. Dette er et generelt motbevis på fravær av vranglås i programmet. Søket blir mer komplisert enn ønskelig siden det er nødvendig å dele det opp. For å få mindre kompliserte søk er det to momenter som kan studeres nærmere: Maude-maskinen og Creol-koden.

Maudes søkefunksjon fungerer nå slik at den gjør et bredde-først søk. Da må den ta vare på alle tilstander den hittil har vært innom, og kan fort gå tom for minne. Det hadde vært en fordel om man i Maudes søkefunksjon også hadde hatt en mulighet til å velge dybde-først søk, med en begrensning på hvor langt ned søket kunne gå i hver gren, slik at det ikke fortsetter i det uendelige for ikke-terminerende programmer. Da hadde det vært nok å huske tilstandene i den grenen det jobbes med i øyeblikket, i tillegg til utgangstilstandene.

Det som genererer aller flest tilstander fra Creol-kode er ikke-determinisme. Ved å fjerne ikke-determinisme vil vi gjøre søk lettere tilgjengelig, men vi vil også miste en måte å simulere ønsket ikke-determinisme som kan inntreffe i et distribuert program styrt av mange forskjellige faktorer.

Vi ser av søkene forsøkt gjort i delkapittel 5.3.5 at Maudes søkefunksjon, slik den er i dag, ikke egner seg så godt for komplekse programmer i CMC. Selv om vi fjerner all ikke-determinisme i AB-protokollen finner vi fortsatt ingen interessante resultater, fordi søkene ikke terminerer normalt.

Vi kan fra søkene i de tre eksemplene trekke den konklusjonen at Maudes søkefunksjon fungerer bra for enkle terminerende Creol-programmer, men at den ikke er til stor hjelp for større programmer, eller programmer med mye ikke-determinisme. Søkefunksjonen kan brukes for å teste ut om en mistanke stemmer for mer komplekse programmer, men ofte vil det koste mer enn det smaker å finne det svaret man er ute etter.

7.3 Videre arbeid

Det er flere retninger det videre arbeidet med den abstrakte maskinen kan gå, og flere av forslagene under er det allerede igangsatt arbeider for å utføre.

Den viktigste oppgaven for at den abstrakte maskinen skal kunne brukes i praksis er å få i stand en kompilator med automatisk oversettelse til CMC. Hvis kompilatoren også inkluderer subklasser og virtuelle bindinger bør CMC og den abstrakte maskinen utvides til å støtte

dette. Den abstrakte maskinen kan også utvides til å støtte dynamiske oppdateringer.

Vi har i denne oppgaven sett at en indre kø med multisettordning ikke gir de beste resultatene for de eksemplene vi har gitt. Det er vanskelig å si noe konkret om hvilken variant som generelt sett egner seg best, så det kan være en idé å undersøke dette nærmere for større og flere eksempler, og med mer fokus på effektivitet.

Maude har et metanivå [4, 5] som gjør at Maude kan brukes som et metaspråk. Med et metaspråk kan man lage programmer som manipulerer andre programmer. Man kan blant annet angi strategier for eksekveringer. Det kunne være interessant å lage forskjellige strategier for bruk av omskrivingsreglene, slik at reglene for eksempel kunne bli valgt tilfeldig, eller få forskjellig vekting. Da vil kanskje problemene rundt valg av kø-ordning bli mindre viktig.

I de aller fleste tilfeller omhandler spesifikasjonen av invarianter og antakelser i Creol kommunikasjonshistorien. Derfor kan logging av denne historien og automatisk testing mot den semantiske spesifikasjonen være særdeles nyttig. Man kan da sikre at gitte invarianter holder ved å ikke tillate ulovlige kjøringer. CMC bør da inkludere den semantiske spesifikasjonen.

Bibliografi

[1] Ron Ben-Natan. *CORBA : a guide to the common object request broker architecture*. McGraw-Hill, New York, 1995.

[2] Denis Caromel and Yves Roudier. Reactive programming in Eiffel//. In J.-P. Briot, J. M. Geib, and A. Yonezawa, editors, *Proceedings of the Conference on Object-Based Parallel and Distributed Computation*, volume 1107 of *Lecture Notes in Computer Science*, pages 125–147. Springer-Verlag, Berlin, 1996.

[3] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, August 2002.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marí-Oliet, and J. Meseguer. Metalevel computation in Maude. In *In 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[5] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In *World Congress on Formal Methods (FM'99)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer-Verlag, 1999.

[6] Gianpaolo Cugola and Carlo Ghezzi. CJava: Introducing concurrent objects in Java. In M. E. Orlowska and R. Zicari, editors, *4th International Conference on Object Oriented Information Systems (OOIS'97)*, pages 504–514, London, 1997. Springer-Verlag.

[7] Ole-Johan Dahl, Kristen Nygaard, and Bjørn Myrhaug. Simula 67 Common base language. Technical report, report no. S-2, Norwegian Computing Center, Oslo, 1968.

[8] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[9] Francisco Durán. *Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, 1999.

[10] F. Durán and J. Meseguer. The Maude specification of Full Maude. Technical report, SRI International, May 1999. `http://maude.cs.uiuc.edu/papers/`.

[11] Martin Fowler. *UML distilled : a brief guide to the standard object modeling language*. The Addison-Wesley object technology series. Addison-Wesley, Boston, 3rd edition, 2004. covers Version 2.0 OMG UML Standard.

[12] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Conference record of the 14th ACM Symposium on Principles of Programming Languages (POPL)*, pages 52–66. ACM Press, 1987.

[13] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 3rd edition, 1998.

[14] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, Mass., 2nd edition, 2000.

[15] Einar Broch Johnsen, Olaf Owe, and Marte Arnestad. Combining active and reactive behavior in concurrent objects. In *Proc. of the Norwegian Informatics Conference (NIK'03)*. Tapir, November 2003. To appear.

[16] Einar Broch Johnsen. An interleaving evaluation semantics for object oriented method calls, October 2002. Internal Report, Department of informatics, University of Oslo.

[17] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[18] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML (Revised)*. MIT Press, Cambridge, Mass., May 1997.

[20] Olaf Owe and Isabelle Ryl. A notation for combining formal reasoning, object orientation and openness. *Research report 278, University of Oslo*, 1999.

[21] Olaf Owe and Isabelle Ryl. On combining object orientation, openness and reliability. In *Proc. of the Norwegian Informatics Conference (NIK'99)*. Tapir, November 1999.

[22] Olaf Owe and Isabelle Ryl. Oun a formalism for open, object oriented, distributed systems. *Research report 270, University of Oslo*, 1999.

[23] Olaf Owe and Isabelle Ryl. Oun grammar. Technical report, Department of informatics, University of Oslo, 1999.

[24] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[25] Marcus Persson. Kompilator fra OUN til Java. Master's thesis, Department of informatics, University of Oslo, February 2002.

[26] ADAPT-FT website. `http://www.ifi.uio.no/~adapt/`.

[27] CREOL website. `http://www.ifi.uio.no/~creol/`.

[28] Maude website. `http://maude.cs.uiuc.edu/`.

[29] Y. Yokote and M. Tokoro. Concurrent programming in Concurrent-Smalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. The MIT Press, Cambridge, Mass., 1987.

[30] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Sigplan Notices*, 21(11):258–268, November 1986.

[31] Peter Ølveczky. Formal modeling and analysis of distributed systems in Maude, 2003. Compendium inf220, Department of informatics, University of Oslo.

Vedlegg

Arbeidet med denne oppgaven har resultert i en eksekverbar abstrakt maskin for Creol . Koden for den orginale varianten av den operasjonelle semantikken er det første vedlegget.

Det andre vedlegget er en artikkel som er akseptert til Norsk Informatikkonferanse 2003, som resultatene fra oppgaven har bidratt til. Artikkelen ble skrevet våren 2003, og endelig ferdigstilt i oktober 2003. Artikkelen presenteres 26. november under NIK'2003 i Oslo.

Maude-kode

```
************** STRUKTUR **************

*** variabeltype ***
fmod DATA is
pr QID .
pr STRING .
pr INT .
pr FLOAT .

sorts Data List Oid Expr .

subsort Qid < Oid .
subsort Oid < Data .

op null : -> Data [ctor] .
op int : Int -> Data [ctor] .
op str : String -> Data [ctor] .
op bool : Bool -> Data [ctor] .
op list : List -> Data [ctor] .

subsort Data < Expr .
subsort Expr < List .

op nil : -> List [ctor] .
op __ :  List List -> List [ctor assoc id: nil] .

endfm

**** Lister av variabler og verdier (Lvar og Att) ****

fmod LIST-QID-VAL is
protecting DATA .

sorts ListQidVal Pair .

op _:_ : Qid Data -> Pair [ctor] .

subsorts Pair < ListQidVal .
```

```
op no : -> ListQidVal [ctor] .
op _,_ : ListQidVal ListQidVal -> ListQidVal [ctor assoc id: no] .

endfm

*** Definisjon av (CREOL) programkode ***

fmod PROG is
protecting LIST-QID-VAL .

sorts Prog GuardProg PairProg MProg Guard Wait Return .

subsort Prog < GuardProg .

op empty : -> GuardProg [ctor] .
op _;_ : GuardProg GuardProg -> GuardProg [id: empty ctor assoc] .

*** expressions
ops not_ neg_ : Expr -> Expr .
ops _+_ _-_ _*_ _/_ _cat_ : Expr Expr -> Expr .
ops _<_ _<=_ _>_ _>=_ : Expr Expr -> Expr .
ops _and_ _or_ _/=_ _=_ : Expr Expr -> Expr .

*** assign, if og while
op _:=_ : Qid Expr -> Prog .
op if_th_fi : Expr GuardProg -> Prog .
op if_th_el_fi : Expr GuardProg GuardProg -> Prog .
op while_do_od : Expr GuardProg -> Prog .

*** new
op _:= new_(_) : Qid Qid List -> Prog .

*** local calls
op _(_;_) : Qid List List -> Prog .      *** synkront kall (med retur)
op !_(_) : Qid List -> Prog .            *** asynkront kall uten retur
op _!_(_) : Qid Qid List -> Prog .       *** asynkront kall med label

*** object calls
op !_._(_) : Qid Qid List -> Prog .       *** uten label
op _!_._(_) : Qid Qid Qid List -> Prog .  *** med label

op _?(_) : Qid List -> Prog .             *** retursetning
op _?(_) : Qid List -> Return .           *** returvakt
op _?(_) : Nat List -> Prog .             *** synkron avslutningsvakt
op end : List -> Prog .                   *** end
op continue : Nat -> Prog .               *** synkron avslutning
```

```
*** guards
op wait : -> Wait .

subsorts Return Expr Wait < Guard .

op nothing : -> Guard [ctor] .
op _&_ : Guard Guard -> Guard [id: nothing assoc comm] .
op _-->_ : Guard GuardProg -> GuardProg .

*** ikke-detmenisme og parallell
op _[]_ : GuardProg GuardProg -> GuardProg .
op _||_ : GuardProg GuardProg -> GuardProg [assoc] .

*********************************************************

*** PairProg er par av Prog og variabelliste

op none : -> PairProg [ctor] .
op _,_ : Prog ListQidVal -> PairProg [ctor] .
op _,_ : GuardProg ListQidVal -> PairProg [ctor] .

*********************************************************

*** Multisett av Prog og PairProg

subsort PairProg < MProg .

op _:_ : MProg MProg -> MProg [ctor assoc comm id: none] .

endfm

*** Definisjon av klasse ***

fmod CLASS is
protecting PROG .

sorts Class Mtd MMtd Cid .

subsort Qid < Cid . *** Cid er klasseid.

op < _: Mtdname | Latt:_, Code:_> : Qid ListQidVal GuardProg ->
Mtd [ctor] .

subsort Mtd < MMtd .

op none : -> MMtd [ctor] .
op _*_ : MMtd MMtd -> MMtd [ctor assoc comm id: none] .

op <_: Cl | Att:_, Init:_, Mtds:_, Ocnt:_> :
```

```
Cid ListQidVal PairProg MMtd Float -> Class .

endfm

*** Definisjon av Objekter ***

fmod OBJECT is
protecting CLASS .

sort Object .

subsort Qid < Oid .
op null : -> Oid [ctor] .

op <_: Id | Cl:_, Pr:_, PrQ:_, Lvar:_, Att:_, Lcnt:_> :
Oid Cid GuardProg MProg ListQidVal ListQidVal Nat -> Object [ctor] .

endfm

fmod QUEUE is
protecting DATA .

sort Msg MMsg Kid Queue .

subsort Msg < MMsg .

op none : -> MMsg [ctor] .
op _+_ : MMsg MMsg -> MMsg [ctor assoc comm id: none] .

*** kall og retur
op invoc(_,_,_,_,_) : Nat Oid Oid Qid List -> Msg [ctor] .
op comp(_,_,_) : Nat Oid List -> Msg [ctor] .

*** start new-melding
op new_(_) : Qid List -> Msg .

*** feilmeldinger
op error : String -> Msg [ctor] .
op warning : String -> Msg [ctor] .

*** kø, med funksjon q for kønavn.
op q : Oid -> Kid .

op < _: QId | Ev:_> : Kid MMsg -> Queue .

endfm

*** Konfigurasjon ***
fmod CONFIG is
```

```
protecting OBJECT .
protecting QUEUE .

sort Configuration .

subsorts Object MMsg Queue Class < Configuration .

op none : -> Configuration [ctor] .
op __ : Configuration Configuration ->
        Configuration [ctor assoc comm id: none] .

endfm

*************** HJELPEFUNKSJONER **************
fmod FUNKSJONER is
pr QUEUE .
pr OBJECT .

*** Kø-funksjon ***
op inqueue : Nat MMsg -> Bool . *** sjekker om Msg er i køen

*** Klasse/metode-funksjoner ***
op getCode : Qid MMtd List -> GuardProg . *** for lokale <-> kall, kun kode
op getVar : Qid MMtd -> ListQidVal .      *** henter lokale var.
op get : Qid MMtd List -> PairProg .      *** henter par (kode, var)

op getNames : ListQidVal -> List .        *** henter navn

op trueGuard : MProg ListQidVal MMsg -> Bool . *** finnes sann guard?

*** variabel-liste-funksjoner ***
op val : Qid ListQidVal -> Data . *** skal gi verdi mot variabelnavn
op _occursIn_ : Qid ListQidVal -> Bool . *** sjekker om var finnes i listen
op alter : Qid Data ListQidVal -> ListQidVal . *** endrer var qid til data
op evalList : List ListQidVal -> List . *** mapper en liste til verdier
op eval : Expr ListQidVal -> Data .     *** beregner uttrykk
op evalB : Expr ListQidVal -> Bool .
op evalI : Expr ListQidVal -> Int .
op evalS : Expr ListQidVal -> String .
op evalG : GuardProg ListQidVal MMsg -> Bool .

op makeAssignment : List List -> Prog . *** Første liste inneholder navn,
                                        *** andre verdier,
                                        *** funksjonen lager navn := verdi

*** variable
vars Q Q' R : Qid .
vars O O' O'' : Oid .
vars L L' : ListQidVal .
```

```
vars D E : Data .
var S : String .
var Mt : MMtd .
var A : Mtd .
var G : Guard .
var SP : Prog .
vars P P' P'' : GuardProg .
var MP : MProg .
vars I J : List .
vars M M' : Msg .
var MM : MMsg .
vars X X' : Expr .
vars B B' : Bool .
vars N N' : Nat .
vars C C' : Int .

eq val(Q, no) = Q .
eq val(Q, ((R : X), L)) = if R == Q then X else val(Q, L) fi .

eq evalList(nil, L) = nil .
eq evalList(X I, L) = eval(X, L) evalList(I, L) .

eq Q occursIn no = false .
eq Q occursIn ((R : D), L) =
  if Q == R then true else (Q occursIn L) fi .

eq alter(Q, E, no) = no .
eq alter(Q, E, ((R : D), L)) =
  if R == Q then (R : E), L
  else (R : D), alter(Q, E, L) fi .

*** skal bli Dataverdier ***
*** data
eq eval(null, L) = null .
eq eval(bool(B), L) = bool(B) .
eq eval(int(C), L) = int(C) .
eq eval(str(S), L) = str(S) .
eq eval(Q, L) = val(Q, L) .

*** data-bool
eq eval(not X, L) = bool(not evalB(X, L)) .
eq eval(X and X', L) = bool(evalB(X, L) and evalB(X', L)) .
eq eval(X or X', L) = bool(evalB(X, L) or evalB(X', L)) .

eq eval((X > X'), L) = bool(evalI(X, L) > evalI(X', L)) .
eq eval((X >= X'), L) = bool(evalI(X, L) >= evalI(X', L)) .
eq eval((X < X'), L) = bool(evalI(X, L) < evalI(X', L)) .
eq eval((X <= X'), L) = bool(evalI(X, L) <= evalI(X', L)) .
```

```
eq eval(X = X', L) = bool((eval(X, L) == eval(X', L))) .
eq eval(X /= X', L) = bool((eval(X, L) =/= eval(X', L))) .

*** data-string
eq eval(X cat X', L) = str(evalS(X, L) + evalS(X', L) ) .

*** data-int
eq eval((neg X), L) = int(- evalI(X, L)) .
eq eval((X + X'), L) = int(evalI((X + X'), L)) .
eq eval((X - X'), L) = int(evalI((X - X'), L)) .
eq eval((X * X'), L) = int(evalI((X * X'), L)) .
eq eval((X / X'), L) = int(evalI((X / X'), L)) .

*** skal bli boolske verdier
eq evalB(bool(B), L) = B .
eq evalB(Q, L) = evalB(val(Q, L), L) .
eq evalB(not X, L) = not evalB(X, L) .

eq evalB(X and X', L) =  evalB(X, L) and evalB(X', L) .
eq evalB(X or X', L) = evalB(X, L) or evalB(X', L) .

eq evalB((X > X'), L) = evalI(X, L) > evalI(X', L) .
eq evalB((X >= X'), L) = evalI(X, L) >= evalI(X', L) .
eq evalB((X < X'), L) = evalI(X, L) < evalI(X', L) .
eq evalB((X <= X'), L) = evalI(X, L) <= evalI(X', L) .

eq evalB(X = X', L) = (eval(X, L) == eval(X', L)) .
eq evalB(X /= X', L) = (eval(X, L) =/= eval(X', L)) .

*** skal bli string
eq evalS(str(S), L) = S .
eq evalS(Q, L) = evalS(val(Q, L), L) .
eq evalS(X cat X', L) = evalS(X, L) + evalS(X', L) .

*** skal bli tall
eq evalI(int(C), L) = C .
eq evalI(Q, L) = evalI(val(Q, L), L) .
eq evalI((neg X), L) = (- evalI(X, L)) .
eq evalI((X + X'), L) = evalI(X, L) + evalI(X', L) .
eq evalI((X - X'), L) = evalI(X, L) - evalI(X', L) .
eq evalI((X * X'), L) = evalI(X, L) * evalI(X', L) .
eq evalI((X / X'), L) = evalI(X, L) quo evalI(X', L) .

*** sjekker en vakt
eq evalG(SP ; P, L, MM) = true .
eq evalG(nothing --> P, L, MM) = true .
eq evalG(wait & G --> P, L, MM) = evalG(G --> P, L, MM) .
eq evalG(E --> P, L, MM) = evalB(E, L) .
eq evalG((Q ? ( J )) & G --> P, L, MM) =
```

```
 inqueue(evalI(Q, L), MM) and evalG(G --> P, L, MM) .
eq evalG(P [] P', L, MM) =
   evalG(P, L, MM) or evalG(P', L, MM) .

*** ser om noe er i køen
eq inqueue(N, none) = false .
eq inqueue(N, comp(N, O, J) + MM) = true .
eq inqueue(N, comp(N', O, J) + MM) =
if N == N' then true else inqueue(N, MM) fi .

*** henter kode
eq getCode(Q, < R : Mtdname | Latt: L, Code: P > * Mt, I) =
if Q == R then makeAssignment(getNames(L), I) ; P
else getCode(Q, Mt, I) fi .

*** henter kode og variabler
eq get(Q, < R : Mtdname | Latt: L, Code: P > * Mt, I) =
if Q == R then
  nothing --> (makeAssignment(getNames(L), I) ; P), L
else get(Q, Mt, I) fi .

*** lager tilordninger
eq makeAssignment(nil, J) = empty .
eq makeAssignment(J, nil) = empty .
eq makeAssignment(Q I, X J) = (Q := X) ; makeAssignment(I, J) .

*** henter navn
eq getNames(no) = nil .
eq getNames((Q : D), L ) = Q getNames(L) .

*** henter variable
eq getVar(Q, < R : Mtdname | Latt: L, Code: P > * Mt) =
if Q == R then L else getVar(Q, Mt) fi .

*** ser etter sann vakt i PrQ
eq trueGuard(none, L, MM) = false .
eq trueGuard((nothing --> P, L) : MP, L', MM) = true .
eq trueGuard((((X --> P) , L) : MP), L', MM) =
  evalB(X, (L, L')) or trueGuard(MP, L', MM) .
eq trueGuard((( (Q ? ( J )) & G) --> P, L) : MP, L', MM) =
  (inqueue(evalI(Q, L), MM) and
     trueGuard((G --> P, L) : MP, L', MM))
  or trueGuard(MP, L', MM) .
eq trueGuard(((N ? ( J )) ; P, L) : MP, L', MM) =
   trueGuard(MP, L', MM) .
eq trueGuard((wait & G --> P, L) : MP, L', MM) =
   trueGuard((G --> P, L) : MP, L', MM) .

endfm
```

```
mod INTERPRET is
pr CONFIG .
pr FUNKSJONER .
pr CONVERSION .

vars O O' : Oid .
vars L L' A A' : ListQidVal .
vars P P' R R' : GuardProg .
var SP : Prog .
var B B' : PairProg .
vars C C' : Cid .
vars Q X Y : Qid .
var D : Data .
vars W W' : MProg .
vars I J K : List .
var M : MMsg .
var Ms : Msg .
var Mt : MMtd .
var F : Float .
vars E E' : Expr .
vars N N' : Nat .
vars G G' : Guard .

*** assign ***
rl [assign] :
< O : Id | Cl: C, Pr: (X := E) ; P, PrQ: W,  Lvar: L, Att: A, Lcnt: N >
=>
if X occursIn L then
< O : Id | Cl: C, Pr: P , PrQ: W,
Lvar: alter(X, eval(E, (L, A)), L), Att: A, Lcnt: N >
else if X occursIn A then
< O : Id | Cl: C, Pr: P , PrQ: W,  Lvar: L,
Att: alter(X, eval(E, (L, A)), A), Lcnt: N >
else
error( "variable_does_not_exist: " + string(X) +
       " in " + string(O) )
fi fi .

*** if ***
rl [if] :
< O : Id | Cl: C, Pr: (if E th R fi) ; P,PrQ: W,  Lvar: L,
Att: A, Lcnt: N >
=>
if evalB(E, (A , L)) then
< O : Id | Cl: C, Pr: R ; P, PrQ: W,  Lvar: L, Att: A, Lcnt: N >
else
< O : Id | Cl: C, Pr: P, PrQ: W,  Lvar: L, Att: A, Lcnt: N > fi .
```

```
rl [if-el] :
< O : Id | Cl: C, Pr: if E th P el P' fi ; R, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
=>
if evalB(E, (A , L)) then
< O : Id | Cl: C, Pr: P ; R, PrQ: W,  Lvar: L, Att: A, Lcnt: N >
else
< O : Id | Cl: C, Pr: P' ; R, PrQ: W,  Lvar: L, Att: A, Lcnt: N >
fi .

*** while ***
rl [while] :
< O : Id | Cl: C, Pr: while E do R od ; P, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
=>
< O : Id | Cl: C, Pr: (if E th (R ; (while E do R od)) fi) ; P,
  PrQ: W, Lvar: L, Att: A, Lcnt: N > .

*** new ***
rl [new-start] :
(new C(I))
< C : Cl | Att: A, Init: (P, L), Mtds: Mt, Ocnt: F >
=>
< C : Cl | Att: A, Init: (P, L), Mtds: Mt, Ocnt: (F + 1.0) >
< qid(string(C) + string(F)) : Id | Cl: C,
Pr: makeAssignment(getNames(A), I) ; P, PrQ: none,
Lvar: L, Att: A, Lcnt: 1 >
< q(qid(string(C) + string(F))) : QId | Ev: none > .

rl [new] :
< O : Id | Cl: C, Pr: (X := new C'(I)) ; P, PrQ: W,  Lvar: L,
 Att: A, Lcnt: N >
< C' : Cl | Att: A', Init: (P', L'), Mtds: Mt, Ocnt: F >
=>
< O : Id | Cl: C, Pr: (X := qid(string(C') + string(F))) ; P,
  PrQ: W, Lvar: L, Att: A, Lcnt: N >
< C' : Cl | Att: A', Init: (P', L'), Mtds: Mt, Ocnt: (F + 1.0) >
< qid(string(C') + string(F)) : Id | Cl: C',
  Pr: makeAssignment(getNames(A'), evalList(I, (L, A))) ; P',
  PrQ: none, Lvar: L', Att: A', Lcnt: 1 >
< q(qid(string(C') + string(F))) : QId | Ev: none > .


*** Ikke-determenistisk valg ***
crl [ikke-determ-p1] :
< O : Id | Cl: C, Pr: (P [] P') ; R, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
=>
```

```
< O : Id | Cl: C, Pr: P ; R, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
if evalG(P, (L , A), M) .

crl [ikke-determ-p2] :
< O : Id | Cl: C, Pr: (P [] P') ; R, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
=>
< O : Id | Cl: C, Pr: P' ; R, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
if evalG(P', (L , A), M) .

crl [ikke-determ-ingen] :
< O : Id | Cl: C, Pr: (P [] P') ; R, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
=>
< O : Id | Cl: C, Pr: empty, PrQ: W : ((P [] P') ; R, L),
 Lvar: no, Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
if not evalG(P [] P', (L , A), M) .

*** Parallell ***
rl [parallell] :
< O : Id | Cl: C, Pr: (P || P') ; R, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
=>
< O : Id | Cl: C, Pr: ((P ; P') [] (P' ; P)) ; R , PrQ: W,
Lvar: L, Att: A, Lcnt: N > .

*** Guard ***
rl [boolguard] :
< O : Id | Cl: C, Pr: (E --> P) ; P', PrQ: W,
Lvar: L, Att: A, Lcnt: N >
=>
if evalB(E, (A , L)) then
< O : Id | Cl: C, Pr: P ; P', PrQ: W,  Lvar: L, Att: A, Lcnt: N >
else
< O : Id | Cl: C, Pr: empty, PrQ: W : (E --> P ; P', L),
 Lvar: no, Att: A, Lcnt: N >
fi .

crl [returnguard] :
< O : Id | Cl: C, Pr: ((X ? ( J )) & G --> P) ; P', PrQ: W,
  Lvar: L, Att: A, Lcnt: N' >
< q( O ) : QId | Ev: M + comp(N, O, K) >
```

```
=>
< O : Id | Cl: C, Pr: G --> makeAssignment(J, K) ; P ; P',
  PrQ: W, Lvar: L, Att: A, Lcnt: N' >
< q( O ) : QId | Ev: M >
if (int(N) == (val(X, L))) .

crl [returnguard_notinqueue] :
< O : Id | Cl: C, Pr: (( X ? ( J )) & G --> P) ; P', PrQ: W,
  Lvar: L, Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
=>
< O : Id | Cl: C, Pr: empty, PrQ: W : ((X ? ( J )) & G --> P ; P', L),
  Lvar: no, Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
if not inqueue(evalI(X, L), M) .

rl [waitguard] :
< O : Id | Cl: C, Pr: (wait & G --> P) ; R, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
=>
< O : Id | Cl: C, Pr: empty, PrQ: W : (G --> P ; R, L),
Lvar: no, Att: A, Lcnt: N > .

rl [emptyguard] :
< O : Id | Cl: C, Pr: nothing --> P, PrQ: W,  Lvar: L, Att: A, Lcnt: N >
=>
< O : Id | Cl: C, Pr: P, PrQ: W,  Lvar: L, Att: A, Lcnt: N > .

*** hente ut "statements" av PrQ ***
crl [return_guard_st] :
< O : Id | Cl: C, Pr: empty, PrQ: (( X ? ( J )) & G --> P, L') : W,
  Lvar: L, Att: A, Lcnt: N' >
< q( O ) : QId | Ev: M + comp(N, O, K) >
=>
< O : Id | Cl: C, Pr: G --> makeAssignment(J, K) ; P, PrQ: W,
  Lvar: L', Att: A, Lcnt: N' >
< q( O ) : QId | Ev: M >
if (N == (evalI(X, L'))) .

crl [bool_guard_st] :
< O : Id | Cl: C, Pr: empty, PrQ: (( E --> P), L') : W,
 Lvar: L, Att: A, Lcnt: N > =>
< O : Id | Cl: C, Pr: P, PrQ: W,  Lvar: L', Att: A, Lcnt: N >
if evalB(E, (A , L')) .

rl [wait_guard_st] :
< O : Id | Cl: C, Pr: empty, PrQ: (( wait --> P), L') : W,
  Lvar: L, Att: A, Lcnt: N > =>
< O : Id | Cl: C, Pr: P, PrQ: W, Lvar: L', Att: A, Lcnt: N > .
```

```
rl [emptyguard_st] :
< O : Id | Cl: C, Pr: empty, PrQ: (( nothing --> P), L') : W,
 Lvar: L, Att: A, Lcnt: N > =>
< O : Id | Cl: C, Pr: P, PrQ: W,  Lvar: L', Att: A, Lcnt: N > .

rl [parallell_st] :
< O : Id | Cl: C, Pr: empty, PrQ: ((P || P') ; R, L') : W,
 Lvar: L, Att: A, Lcnt: N > =>
< O : Id | Cl: C, Pr: (P || P') ; R , PrQ: W,
 Lvar: L', Att: A, Lcnt: N > .

crl [ikke-determ_p1_st] :
< O : Id | Cl: C, Pr: empty, PrQ: ((P [] P') ; R, L') : W,
 Lvar: L, Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
=>
< O : Id | Cl: C, Pr: P ; R , PrQ: W,
 Lvar: L', Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
if evalG(P, (L , A), M) .

crl [ikke-determ_p2_st] :
< O : Id | Cl: C, Pr: empty, PrQ: ((P [] P') ; R, L') : W,
 Lvar: L, Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
=>
< O : Id | Cl: C, Pr: P' ; R , PrQ: W,
 Lvar: L', Att: A, Lcnt: N >
< q( O ) : QId | Ev: M >
if evalG(P', (L , A), M) .

*** kall ***
*** lokale kall ***
rl [localcall-asynchronic-noreturn] :
< O : Id | Cl: C, Pr: (! X ( I )) ; P, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
< C : Cl | Att: A', Init: B', Mtds: Mt, Ocnt: F >
=>
< O : Id | Cl: C, Pr: P,
PrQ: get(X, Mt, int(N) null evalList(I, (L, A)) ) : W,
 Lvar: L, Att: A, Lcnt: N + 1 >
< C : Cl | Att: A', Init: B', Mtds: Mt, Ocnt: F > .

rl [localcall-asynkron-label] :
< O : Id | Cl: C, Pr: (X ! Y ( I )) ; P, PrQ: W,  Lvar: L,
Att: A, Lcnt: N >
< C : Cl | Att: A', Init: B', Mtds: Mt, Ocnt: F >
=>
```

```
< O : Id | Cl: C, Pr: (X := int(N)) ; P,
PrQ: get(Y, Mt, int(N) O evalList(I, (A,L)) ) : W,
 Lvar: L, Att: A, Lcnt: N + 1 >
< C : Cl | Att: A', Init: B', Mtds: Mt, Ocnt: F > .

rl [localcall-synchronic] :
< O : Id | Cl: C, Pr: ( X ( I ; J )) ; P, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
< C : Cl | Att: A', Init: B', Mtds: Mt, Ocnt: F >
=>
< O : Id | Cl: C, Pr: getCode(X, Mt, int(N) O evalList(I, (L, A)))
; continue(N), PrQ: ((N ? (J)) ; P, L) : W, Lvar: getVar(X, Mt),
Att: A, Lcnt: N + 1 >
< C : Cl | Att: A', Init: B', Mtds: Mt, Ocnt: F > .

rl [continue] :
< O : Id | Cl: C, Pr: continue(N), PrQ: ((N ? (J)) ; P, L') : W,
Lvar: L, Att: A, Lcnt: N' >
< q( O ) : QId | Ev: M + comp(N, O, K) >
=>
< O : Id | Cl: C, Pr: makeAssignment(J, K) ; P, PrQ: W,
Lvar: L', Att: A, Lcnt: N' >
< q( O ) : QId | Ev: M > .

*** objektkall ***
rl [objectcall-label] :
< O : Id | Cl: C, Pr:  (Q ! X . Y ( I )) ; P, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
=>
< O : Id | Cl: C, Pr: (Q := int(N)) ; P, PrQ: W,
Lvar: L, Att: A, Lcnt: (N + 1) >
invoc(N, O, val(X, (L, A)), Y, (evalList(I, (L, A)))) .

rl [objectcall-nolabel] :
< O : Id | Cl: C, Pr:  (! X . Y ( I )) ; P, PrQ: W,
Lvar: L, Att: A, Lcnt: N >
=>
< O : Id | Cl: C, Pr:  P, PrQ: W,  Lvar: L, Att: A, Lcnt: N + 1 >
invoc(N, null, val(X, (L, A)), Y, (evalList(I, (L, A)))) .

*** ta imot kall ***
rl [receivecall] :
< O : Id | Cl: C, Pr: P, PrQ: W,  Lvar: L, Att: A, Lcnt: N' >
< q( O ) : QId | Ev: M + invoc(N, O', O, Y, I) >
< C : Cl | Att: A', Init: B', Mtds: Mt, Ocnt: F >
=>
< O : Id | Cl: C, Pr: P, PrQ: ( get(Y, Mt, (int(N) O' I))) : W,
 Lvar: L, Att: A, Lcnt: N' >
< q( O ) : QId | Ev: M >
```

```
< C : Cl | Att: A', Init: B', Mtds: Mt, Ocnt: F > .

*** svare på kall ***
rl [reply] :
< O : Id | Cl: C, Pr:  (end ( J )) ; P, PrQ: W,  Lvar: L,
Att: A, Lcnt: N >
=>
< O : Id | Cl: C, Pr: P, PrQ: W,  Lvar: L, Att: A, Lcnt: N >
if val('caller, L) == null then
none
else
comp( evalI('label, L),
      val('caller, L),
      evalList(J, (L, A) ))
fi .

*** vente aktivt på svar ***
crl [getreply] :
< O : Id | Cl: C, Pr:  (X ? ( J )) ; P, PrQ: W,  Lvar: L,
Att: A, Lcnt: N' >
< q( O ) : QId | Ev: M + comp(N, O, K) >
=>
< O : Id | Cl: C, Pr: makeAssignment(J, K) ; P, PrQ: W,  Lvar: L,
Att: A, Lcnt: N' >
< q( O ) : QId | Ev: M >
if (int(N) == val(X, L)) .

*** ta inn meldinger i køen ***
rl [invocmsg] :
< q( O' ) : QId | Ev: M > invoc(N, O, O', X, I)
=>
< q( O' ) : QId | Ev: M + invoc(N, O, O', X, I) > .

rl [compmsg] :
< q( O ) : QId | Ev: M > comp(N, O, K)
=>
< q( O ) : QId | Ev: M + comp(N, O, K) > .

endm
```