UNIVERSITETET I OSLO
Institutt for informatikk

Utvidelse av Creol-språket med synkronisert fletting

Masteroppgave

Are Husby

31. oktober 2005

Sammendrag

Distribuerte systemer består av et antall samarbeidende prosesser som kjører på ulike maskiner og som kommuniserer med meldingsutveklsing. Distribuerte systemer har blitt så vanlige at de utgjør en del av samfunnets infrastruktur. Flere og flere kritiske systemer er distribuerte. Kritiske systemer må være pålitelige og korrekte.

Det er viktig at det er mulig å endre deler av et kritisk, distribuert system på en god måte. Endringer må kunne skje inkrementelt, det vil si i flere separate utviklingstrinn etter hvert som behovene melder seg. I et objektorientert system er det naturlig at endringer implementeres ved hjelp av arv. En typisk situasjon vil være at vi ønsker å arve en klasse i systemet fordi vi vil erstatte den med en subklasse med forbedret implementasjon eller som tilbyr flere metoder.

Objekter i et distribuert system har behov for å synkronisere interaksjonen sin. Arv av synkroniseringskode kan være problematisk. I denne oppgaven brukes problemet med arveanomali som eksempel på hvordan arv av synkroniseringskode fører til problemer som ikke har noen tilfredsstillende løsning når vi bruker de vanligste programmeringskonstruktene.

Creol er et lite språk som brukes til å studere programmeringskonstrukter i en distribuert og objektorientert kontekst. Creol har et forslag til hvordan problemene med arv av synkroniseringskode kan løses på en bedre måte. Forslaget baserer seg på bruken av et konstrukt som heter "synkronisert fletting", som hittil bare har vært beskrevet uformelt. Den operasjonelle semantikken til Creol er definert i det formelle spesifikasjons- og modelleringsspråket Maude.

Hovedmålet for denne oppgaven er å videreutvikle og presisere semantikken til synkronisert fletting med utgangspunkt i den eksisterende, uformelle definisjonen. Jeg diskuterer ulike semantiske tolkninger av både synkronisert fletting og to andre beslektede programmeringskonstrukter. Dette resulterer i en implementasjon av synkronisert fletting i språket Maude, som utvider definisjonen av språket Creol.

Et sekundært mål for oppgaven er å gjøre en tentativ validering og verifikasjon av implementasjonen. Dette gjøres med Maude sine muligheter for maskinell analyse.

Som et eksempel på relatert arbeid i problemområdet, ser vi på hvordan det aspektorienterte programmeringsparadigmet (AOP) forholder seg til problemet med arveaniomali.

Forord

Denne oppgaven er en del av min mastergrad ved institutt for informatikk ved Universitetet i Oslo. Problemstillingen er en del av Creol-prosjektet ved instituttets forskningsgruppe for presis modellering og analyse (PMA).

Studieforløpet mitt har ikke vært helt ortodokst. Denne oppgaven representerer avslutningen på mine åtte år som student. For fem år siden så jeg ikke for meg at jeg skulle holde på med matematisk-naturvitenskapelige fag overhodet. Jeg er glad for at det i skrivende stund ser ut som at jeg kommer i mål med æren i behold og at jeg fremdeles synes informatikk er gøy.

Takk til professor Olaf Owe for veiledning. Han har svart på spørsmålene mine og fungert som et kompass når jeg ikke har visst hvilken retning jeg skulle gå i. Takk til min andre veileder professor Einar Broch Johnsen for faglige innspill og korrektulesing under sterkt tidspress.

Takk til (i alfabetisk rekkefølge) Andreas, Bjørn Arild, Kenneth, Martin, Stian og Tho for helt nødvendig sosialt samvær i alle hverdagene som arbeidet med oppgaven har bestått av.

Takk til foreldrene mine for å ha gitt meg ambisjonen om å studere og for oppmuntring underveis.

Are Husby, Oslo, 30.10.2005.

Innhold

Figurer

Kapittel 1

Innledning

1.1 Problemområdet

1.1.1 Distribuerte, objektorienterte systemer

Et distribuert system er et datasystem som består av et antall samarbeidende programmer som kjører på ulike maskiner og som kommuniserer ved meldingsutveksling. Distribuerte systemer har eksistert side 1970-tallet, men var da begrenset til hver sin bygning. Etter at Internettet har blitt tatt i bruk, har distribuerte systemer blitt verdensomspennende. Eksempler på distribuerte systemer er World Wide Web, online banktjenester og deling av pasientjournaler mellom sykehus, og i det siste har også internett-telefoni begynt å bli utbredt. Distribuerte systemer har blitt så vanlige at de utgjør en del av samfunnets infrastruktur. Uttrykk som "nettverket er maskinen" og "allestedsnærværende datamaskiner" (eng: ubiquitous computing) blir stadig mer relevante.

De fleste distribuerte systemer har høye krav til korrekthet og oppetid. www og internett-telefoni er raskt i ferd med å bli fundamentale deler av verdenssamfunnets kommunikasjons-infrastruktur. Noen systemer er mer kritiske enn andre. I eksemplet med banktjenester, kan pengene til bedrifter og privatpersoner avhenge av systemet. I tilfellet med informasjonsdeling mellom sykehus, kan systemets pålitelighet være et spørsmål om liv og død. For slike systemer er det ekstra viktig å sikre pålitelighet og korrekthet.

Distribuerte systemer må ofte være i virksomhet kontinuerlig. Det er nærmest uunngåelig at et større system får behov for å endres etter at det har blitt satt i drift. Dermed ønsker vi å legge til rette for at programvarekomponenter kan endres inkrementelt og dynamisk. Det vil si at et program kan endres skritt for skritt etter hvert som behovene melder seg og at endringene kan gjøres uten å stoppe aktiviteten i systemet.

Det objektorienterte programmeringsparadigmet er et av de mest brukte i utvikling av distribuerte systemer. Objektorientert programmering kjennetegnes blant annet ved bruk av klasser, som definerer datastrukturer og operasjoner på disse. Disse klassene instansieres, og de resulterende objektene utfører beregninger i samarbeid med hverandre. Objektene samarbeider ved å gjøre kall på hverandres metoder, som utfører operasjonene definert i klassene.

Objektorientering er hensiktsmessig for det første fordi det er det naturlig å modellere komponentene i distribuerte systemer som objekter som kommuniserer med hverandre. For det andre gir objektorientering programmeringstekniske fordeler i utviklingen av hver programvarekomponent. Objektorientering gjør det mulig å gjenbruke kode fra klassedefinisjonene ved arv. Arv lar programmereren definere en ny klasse — en subklasse — som automatisk inneholder en kopi av koden fra en gammel klasse — superklassen. Dette er verdifullt, fordi det tillater utvikling av programmer inkrementelt.

Virtuell binding av metodekall er en fordel med objektorientering som letter gjenbruk av

kode. Virtuell binding gjør at et metodekall som forekommer i en subklasse kan bindes til en metodedefinisjon i en superklasse. Ved å redefinere den samme metoden i subklassen, vil kallet automatisk bindes til subklassens metodedefinisjon.

Denne masteroppgaven er en del av Creol-prosjektet [26]. Prosjektet utvikler et programmeringsspråk som har som mål å være spesielt godt egnet for utvikling av objektorienterte programmer i distribuerte systemer. Prosjektet studerer programmeringskonstrukter og resonneringskontroll. Det vil si at det ser på hvilke instruksjoner og mekanismer et slikt språk bør tilby programmereren og hvordan man kan bevise viktige egenskaper i programmer skrevet i dette språket. Språkets navn er Creol, som er et akronym for "Concurrent Reflective Object-oriented Language".

1.1.2 Arv av synkroniseringskode

Et av målene for Creol-prosjektet er å se på hva som skal til for å kunne endre deler av et distribuert system på en god måte. I et objektorientert system er det naturlig at endringer implementeres ved hjelp av arv. En typisk situasjon vil være at vi ønsker å arve en eksisterende klasse i systemet, fordi vi vil erstatte den med en subklasse med forbedret implementasjon eller som tilbyr flere metoder.

Distribuerte objekter har behov for å synkronisere samhandlingen sin. *Synkroniseringsbetingelsene* til et objekt i et samtidig system definerer hvilke tilstander et objekt kan være i for å kunne utføre operasjonene sine uten å kompromittere dataintegriteten sin. Programkoden som implementerer synkroniseringsbetingelsene kalles *synkroniseringskode*. I motsetning til dette har vi *forretningskoden*, som utfører de oppgavene som utgjør den grunnleggende funksjonaliteten til systemet ("gjør det som egentlig skal gjøres").

Arv av synkroniseringskode kan være problematisk. Vi skal se eksempler på hvordan arv av synkroniseringskode fører til problemer som ikke har noen tilfredsstillende løsning når vi bruker de vanligste programmeringskonstruktene. Et problem som kan oppstå ved arv av synkroniseringskode har blitt kalt *arveanomali* [11, 19, 20]. Problemet består kort sagt i at det er umulig å tilføre ny synkroniseringskode i systemet uten at det nødvendiggjør ikke-trivielle redefinisjoner av metoder. Dette vanskeliggjør inkrementell og sikker endring av systemet.

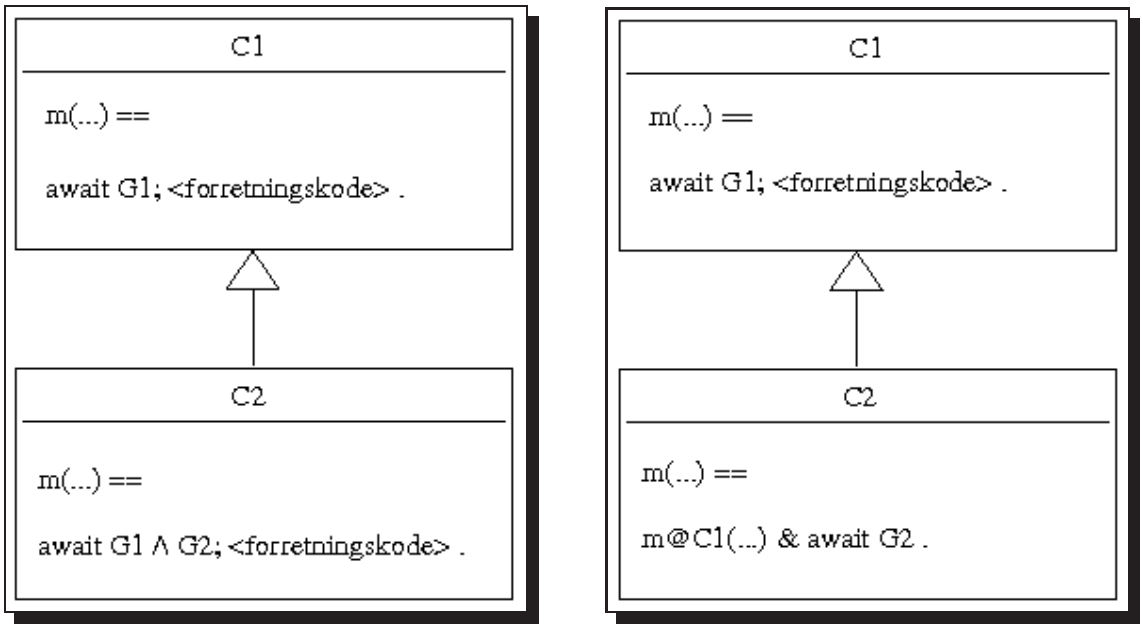
Creol har et forslag til hvordan vi kan arve klasser slik at eksistensen av synkroniseringskode ikke er et like stort problem. Forslaget baserer seg på bruken av en språkkonstruksjon som heter *synkronisert fletting*, representert syntaktisk med symbolet &. Denne språkkonsktruksjonen er skissert i [8, 15].

1.1.3 Creol sin tilnærming til arv av synkroniseringskode

En *vakt* er et boolsk uttrykk som må evaluere til verdien sann før programeksekveringen kan fortsette. Vakter brukes i synkroniseringskoden i en klasse for å implementere synkoniseringsbetingelsene. Når en klasse endres, endres ofte synkroniseringsbetingelsene også. Følgelig må synkroniseringskoden endres, slik at den oppfyller de nye betingelsene.

Det er vanlig at en betingelse styrkes i den arvende klassen, ved at vi legger til en ny betingelse til den gamle. Synkronisert fletting lar oss lage et *aggregat av vakter* fra ulike metoder. Når vi evaluerer konjunksjonen av vaktene i dette aggregatet, finner vi ut om alle synkroniseringsbetingelsene er oppfylte samtidig.

&-operatoren for synkronisert fletting brukes i kombinasjon med en operator @ for statiske kall, som står i motsetning til virtuelle kall. @-operatoren brukes i metodekall til å angi hvilken klasse metodedefinisjonen skal hentes fra når metoden instansieres. Bruken av @ ligner på bruken av Java sitt nøkkelord "super" eller Simula sitt nøkkelord "qua". Bruken av @-operatoren er ikke begrenset til nærmeste superklasse, og den kan brukes ved multippel arv.



(a) Uten synkronisert fletting

(b) Med synkronisert fletting

Figur 1.1: Arv av en klasse med synkroniseringskode i Creol.

Vi ser på et lite eksempel i figur 1.1 for å gi en introduksjon til hvordan synkronisert fletting kan brukes. Vi har en klasse $C1$, hvor en metode m() er definert med Creol-syntaks. m() inneholder forretningskoden, og synkroniseringskravet er ivaretatt av metodevakten *await* $G1$. Vi lar en klasse $C2$ arve $C1$, og i $C2$ er betingelsen for kjøring av forretningskoden strengere. Vi ønsker å forsterke vakten i $C1$ med en ny vakt $G2$. I klassen C2 må både betingelsen angitt med G1 og betingelsen angitt med G2 være oppfylt samtidig for at objektet skal beholde dataintegriteten sin.

I figur 1.1(a) ser vi hvordan m() redefineres uten bruk av &- og @-operatoren. Symbolet $\wedge$ tilsvarer den boolske "and"-operatoren.

I figur 1.1(b) bruker vi de to operatorene & og @. Betydningen av denne programsetningen er at vi kaller metoden m() slik den er definert i klassen $C1$ (@), på den betingelsen at konjunksjonen av $G1$ og $G2$ er sann (&). I motsetning til i figur 1.1(b) trenger ikke programmereren å vite hvordan m() er implementert i $C1$, bortsett fra at m() innledes med en vakt. Vi har full gjenbruk av definisjonen fra $C1$, i motsetning til redefininsjonen i figur 1.1(a). Operatorene & og @ brukes til å lage et aggregat av vakter fra begge klasser.

Hvorfor kan ikke koden i klassen C2 i figur (b) være *await* $G2; m@C1(\ldots)$? Konsekvensen av dette kan bli følgende hendelsesforløp: Ved kjøring passerer vi vakten G2, fordi den evaluerer til sann. Det medfører at m() i C1 kalles. Kjøringen av m() i C1 suspenderes midlertidig fordi G1 er usann. I mellomtiden kan en annen metode i objektet bli kjørt, som resulterer i at objektets tilstand endres slik at G2 blir usann og G1 blir sann. Metoden m() i C1 får nå anledning til å passere G1 og begynne på forretningskoden. Dette står ikke i samsvar med synkroniseringsbetingelsen, som sier at både G1 og G2 skal være sanne samtidig for at forretningskoden skal kunne kjøres. Vi risikerer at objektets dataintegritet krenkes, noe som kan ha alvorlige følger hvis objektet er en del av et kritisk system.

Vi skal se grundig på denne måten å bruke &-operatoren på senere i oppgaven.

1.1.4 Maude og Creol-interpreten

Den operasjonelle semantikken til språket Creol er spesifisert med Maude [4]. Maude er en implementasjon av omskrivingslogikk, og er et språk som er både et programmeringsspråk, en modelleringsnotasjon og et formelt spesifikasjonsspråk. Forenklet framstilt, består en Maude-modell av en formell definisjon av bestanddelene i et system og et antall tilstandsoverganger som er mulige i dette systemet.

Et Maude-program kan eksekveres, og dette tilsvarer å simulere kjøringer av det modellerte systemet. Definisjonen av Creol sin operasjonelle semantikk er dermed i praksis en interpret. Maude har simuleringskommandoer som lar oss analysere modellen ved å se på ulike mulige eksekveringshistorier/kjøringer. Spesielt har Maude en kommando som lar oss søke gjennom alle systemtilstander som er oppnåelige med de spesifiserte overgangene fra en gitt starttilstand.

Maude er et kraftfullt høynivåspråk som tillater en høy grad av abstraksjon. Totalt består definisjonen av hele den operasjonelle semantikken til Creol av 950 linjer. Det er definert i underkant av 30 tilstandsoverganger, samt et antall hjelpefunksjoner. Dette er såpass lite kode at det er godt mulig å endre interpreten for å eksperimentere med ulike semantiske varianter av språkkonstruksjonene.

1.2 Mål for oppgaven

Semantikken til synkronisert fletting har hittil bare blitt skissert og beskrevet uformelt og har ikke vært implementert i interpreten. Hovedmålet for denne oppgaven er å gi en formell definisjon av semantikken til synkronisert fletting, samt å gi en fornuftig implementasjon i Maude.

I denne oppgaven skal jeg videreutvikle og presiserer semantikken til synkronisert fletting med utgangspunkt i den eksisterende, uformelle definisjonen. Vi ser på ulike semantiske tolkninger av synkronisert fletting og andre beslektede konstrukter i Creol. Jeg diskuterer fordeler og ulemper med de ulike tolkningene. Dette gjøres blant annet ved å analysere programeksempler som benytter seg av de berørte operatorene.

Dette er en problemstilling på to nivåer. For det første har vi det abstrakte nivået som handler om hvordan vi vil at semantikken til synkronisert fletting skal være. For det andre er det et programmeringsproblem, fordi ikke bare skal løsningen implementeres i språket Maude, den skal også tilpasses og fungere sammen med den eksisterende Maude-implementasjonen av Creol sin operasjonelle semantikk.

Et sekundært mål med oppgaven er å bruke maskinell analyse til å gjøre en tentativ verifikasjon og validering av implementasjonen av synkronisert fletting. Jeg ønsker å 1) sannsynliggjøre at implementasjonen er feilfri og 2) vise at Creol sin strategi for inkrementell endring av synkroniseringsbetingelser ved arv kan realiseres i programmer som Creol-interpreten kan tolke.

1.3 Metodologi brukt i oppgaven

Oppgavens metode består først og fremst i å bruke Maude. Å definere synkronisert fletting med språket Maude innebærer en utvidelse av Creol-interpreten. Jeg bruker Maude til å analysere implementasjonen av synkronisert fletting ved å studere konsekvensene av valgene jeg har gjort på programmer som tolkes av interpreten. Formelle metoder og Maude beskrives i kapittel 4.

Iterativ systemutvikling er en generell modell for programutvikling som har blitt brukt som beskrevet i 5.1. Når det gjelder metodologi for algoritmedesign, lener jeg meg tungt på splitt-og-hersk-taktikken, beskrevet i 5.4.2. Aspektorientert programmering kan sies å være en metodologi som anvendes på blant annet synkroniseringsproblemer, og som er annerledes enn Creol sin tilnærmingsmåte. AOP beskrives i 2.5.1.

1.4 Oppgavens innhold og struktur

Kapittel 2, 3 og 4 utgjør bakgrunnen for oppgaven. Det er i kapittel 5 og 6 at mitt bidrag presenteres og hvor jeg forsøker å nå målene for oppgaven beskrevet ovenfor.

Kapittel 2 beskriver problemdomenet, som er arv av synkroniseringskode. Mer spesifikt innenfor dette temaet ser vi på problemet med arveanomali. Synkronisert fletting motiveres ved at jeg viser eksempler på hvorfor problemet med arveanomali ikke kan løses på noen god måte med de vanligste objektorienterte språkkonstruktene. Det aspektorienterte programmeringsparadigmet (AOP) foreslår en løsning på problemet med arveanomali. Vi ser på AOP sin tilnærmingsmåte, fordi AOP er et av de viktigste eksemplene på relatert arbeide i problemdomenet.

Kapittel 3 beskriver språket Creol. Vi ser på hvordan distribuerte systemer representeres i Creol og beskriver språkets syntaks forøvrig. Videre ser vi på Creol sin tilnærmingsmåte til problemet med arveanomali. Her ser vi hvordan vi ønsker at synkronisert fletting skal kunne brukes når det er implementert.

Kapittel 4 gir den tekniske bakgrunnen for implementasjonen og mye av den metodologiske bakgrunnen. Jeg beskriver hva Maude er og motiverer bruken av Maude som metodologi i arbeidet med å definere Creol. Det meste av kapitlet beskriver hvordan Maude har blitt brukt til å definere den operasjonelle semantikken til Creol før min utvidelse. Beskrivelsen av Creol-interpreten er samtidig en beskrivelse av språket CMC (Creol Machine Code), som er det språket som interpreten tolker/arbeider med og som Creol-programmer oversettes til. Det er denne interpreten jeg skal utvide med synkronisert fletting.

Kapittel 5 er hvor jeg forsøker å nå hovedmålet for oppgaven. Kapitlet inneholder beskrivelsen av hvordan jeg har brukt Maude til å definere semantikken til synkronisert fletting. Framstillingen er problemorientert og reflekterer således den iterative systemutviklingsmodellen som bruken av Maude legger opp til. Jeg identifiserer de viktigste implementasjonsutfordringene og diskuterer ulike løsninger. Vi ser på konsekvensene av noen uheldige varianter av semantikken til synkronisert fletting.

Kapittel 6 er hvor jeg forsøker å nå det sekundære målet for oppgaven. Her analyserer jeg interpreten slik den har blitt etter at den er utvidet med synkronisert fletting. Dette gjøres ved å bruke interpreten til å kjøre programmer som benytter &-operatoren og å studere resultatet av kjøringene. Jeg simulerer enhver mulig eksekveringshistorie ved hjelp av Maude sin søkekommando.

Kapittel 7 inneholder en oppsummering av utfordringene, oppdagelsene og resultatene i arbeidet med oppgaven.

Kapittel 2

Arv av synkroniseringskode i distribuerte objekter

I dette kapitlet går vi inn i problemområdet, som er arv av synkroniseringskode i distribuerte objekter. Vi starter med å forklare sentrale begreper som distribusjon, synkronisering og arv. Deretter ser vi spesifikt på problemet med arveanomali, som motiverer synkronisert fletting. Vi ser på det aspektorienterte programmeringsparadigmet sin tilnærmingsmåte til arveanomali. Creol sin tilnærmingsmåte beskrives i neste kapittel.

2.1 Åpne distribuerte systemer

Samtidige systemer er en løs kategori av systemer som består av et antall samarbeidende prosesser. Prosessene kan kjøre på samme maskin eller på ulike maskiner i et nettverk. Nettverket kan være alt fra en klynge (eng: cluster), et intranett eller Internettet. Prosessene kan kommunisere med hverandre ved å skrive og lese til delte variabler eller ved meldingsutveksling. Distribuerte systemer er en underkategori av samtidige systemer, hvor prosessene kjører på ulike maskiner og kommuniserer med meldingsutveksling. Hovedhensikten med å lage distribuerte systemer er å dele på ressurser som data, prosessorkraft, minne og lagringsplass, slik at man kan løse oppgaver som blir for store for "stand alone" systemer.

For å forklare hva et distribuert system er, kan det være nyttig å se på de viktigste, generelle utfordringene ved implementasjon av distribuerte systemer [7]:

Heterogenitet Systemene består av ulike typer nettverk, operativsystemer, maskinvare og programmeringsspråk.

Åpenhet Åpenhet er navnet på den egenskapen ved et distribuert system som sier hvorvidt systemet kan utvides og deler av systemet re-implementeres. I et åpent system er det lett å legge til en ressursdelende tjeneste som kan gjøres tilgjengelig for klientprogrammer. Åpne distribuerte systemer kjennetegnes ved at

- grensesnittene til viktige programvarekomponenter blir gjort tilgjengelige for andre programvareutviklere
- grensesnittene til ressursene i systemet publiseres
- kommunikasjonen baserer seg på kjente kommunikasjonsprotokoller og -mekanismer
- systemet kan settes sammen av komponenter fra ulike maskinvare- og programvareprodusenter.

Sikkerhet Kryptering og autentisering må til for å sikre dataene og kommunikasjonen i systemet.

Skalerbarhet Et distribuert system er skalerbart hvis kostnadene med å utvide systemet ikke stiger nevneverdig mye for hver utvidelse.

Feilhåndtering En prosess, maskin eller kommunikasjonskanal kan når som helst feile. Hele subsystemer kan ha nedetid av ulike årsaker. Systemet som helhet skal kunne fungere likevel. Hver komponent må være designet slik den håndterer feil som skjer utenfor seg selv på en god måte.

Samtidighet Flere samtidige prosesser vil føre til flere samtidige henvendelser til en ressurs. Hver ressurs må være implementert slik at dette ikke blir et problem. Synkronisering er et viktig aspekt ved distribuerte systemer.

Transparens Det er ønskelig å gjøre noen aspekter ved distribueringen usynlige. Det vil si at programmereren av en applikasjon ikke trenger å tenke på for eksempel ressursens lokasjon, hvorvidt ressursen speiles/replikeres, implementasjonsdetaljer i andre komponenter og de rent tekniske aspektene ved kommunikasjonen. Man kan også ønske at feil håndteres på en slik måte at de blir transparente for en komponent.

2.2 Synkronisering i distribuerte objekter

Det er naturlig å bruke objektorienterte språk når vi utvikler samtidige systemer. I et system av enheter som eksekverer parallelt, kan vi se på en enhet som et objekt. I løpet av 1990-tallet ble det vanligere å lage samtidige systemer, og dermed har noen samtidige, objektorienterte språk fått stor utbredelse. Eksempler er Java og C#. Terminologien presentert her er hentet hovedsaklig fra [12, 19].

Et objekt tilbyr et antall metoder. I en distribuert sammenheng er det naturlig å implementere et metodekall som en melding til det kallede objektet. Vi sier at et objekt har en mengde av meldinger som det forstår, og denne mengden tilsvarer mengden av objektets metoder. Vi sier at et objekt *aksepterer en melding* når det kjører en metode. Når en melding kan aksepteres, sier vi at den tilsvarende metoden er *beredt* (eng: enabled).

Det er en større utfordring for et distribuert system å bevare dataintegriteten sin enn for et ikke-distribuert system. De distribuerte objektene må synkronisere samhandlingen sin. Dette kan gjøres ved bruk av transaksjonsprotokoller. I tillegg til dette, er det vanlig at det kalte objektet er ansvarlig for å ivareta sin egen dataintegritet. Det betyr at objektet ikke aksepterer en melding hvis det medfører at objektets dataintegritet kompromitteres. Et objekt sine *synkroniseringsbegrensninger* spesifiserer i hvilke tilstander et objekt kan akseptere en melding. De begrenser mengden av meldinger som er akseptable i hver tilstand.

Vi ser på et eksempel med en buffer med begrenset størrelse. Bufferobjektet fungerer som tjener i et samtidig system, hvor klientene putter inn og henter ut elementer ved å kalle metodene put() og get(). Jeg bruker en Java-aktig syntaks.

```
class Buffer { ...
  int bufferSize; // maksimum antall elementer i bufferen
  int numberOfElements; // antall elementer i bufferen på ethvert tidspunkt
  void put(Object el) {...}
  Object get() {...}  }
```

Det er to åpenbare synkroniseringsbegrensninger i et slikt objekt: Vi kan ikke legge til et element til en full buffer og vi kan ikke fjerne et element fra en tom buffer. Sagt på en annen måte: Vi kan ikke

akseptere en put-melding når bufferobjektet er i en full tilstand og vi kan ikke akseptere en get-melding når bufferen er i en tom tilstand.

For mer komplekse bufferobjekter kan det være mange tilstander hvor bufferen er full. Vi er interesserte i å snakke om mengden av tilstander hvor bufferen er for eksempel full i motsetning til mengden av tilstander hvor bufferen ikke er full. Det er mulig å dele objektets tilstandsrom opp i disjunkte delmengder av tilstander. Dette gir oss muligheten til å forenkle språket, ved å referere til disse delmengdene ved å si at objektet er i en full eller ikke-full tilstand.

Generelt kan vi si at en *synkroniseringsbetingelse* er et boolsk uttrykk som er assosiert med én synkroniseringsbegrensning. I en objektorientert sammenheng er det naturlig at en betingelse blir evaluert i kontekst av metodekallets reelle parametre og det kallede objektets tilstand. I tillegg kan objektets meldingshistorie inkluderes i konteksten, slik at aksepten av en ny melding er betinget av de meldingene som har blitt akseptert tidligere.

Objektets adferd må oppfylle objektets synkroniseringsbegrensninger. Den delen av objektets kode som er ment å gjøre dette kalles *synkroniseringskode*. Merk at vi skiller mellom synkroniseringsbegrensningen som spesifikasjon og adferden til synkroniseringskoden. Synkroniseringskoden må være konsistent med synkroniseringsbegrensningene. Hvis ikke, får vi en semantisk kjøretidsfeil, fordi objektet aksepterer en melding som den ikke burde akseptere.

En tradisjonell *vakt* er et boolsk uttrykk i et program som må evaluere til verdien sann for at eksekveringen skal fortsette. Synkroniseringsbetingelser kan implementeres med vakter. En *metodevakt* [11] er en implementasjon av en synkroniseringsbetingelse som er assosiert med en bestemt metode og hvor kjøringen av denne metoden kan påbegynnes hvis og bare hvis vakten dens er sann.

Objektvis synkronisering benytter synkroniseringsprimitiver som for eksempel vakter eller semaforer [1]. Disse primitivene kan brukes på forskjellige måter. En *synkroniseringsstrategi* (eng: synchronization scheme) består av en mengde synkroniseringsprimitiver og en plan for hvordan de bør brukes.

2.3 Arv

To fundamentale fordeler med objektorientering er *arv* og *innkapsling* (eng: encapsulation). Arv tillater gjenbruk av kode og en hensiktsmessig og "menneskevennlig" strukturering av koden i et program. En metodedefinisjon fra én klasse (superklassen) kan gjenbrukes i en annen klase (subklassen) med så godt som ingen ekstra anstrengelse i forhold til om bare superklassen hadde metodedefinisjonen. Siden den arvede koden i subklassen er en kopi av koden fra superklassen, vil en endring i superklassen umiddelbart reflekteres i subklassen. Hvis vi har en klasse som vi ønsker å endre, kan det være naturlig å arve den og legge til nye metodedefinisjoner eller redefinere gamle metoder i subklassen.

Innkapslingsprinsippet sier at vi skal kunne forholde oss til et objekt som en "black box" eller en abstrakt datatype. Det vil si at vi ikke trenger å kjenne til hvordan objektets metoder er implementert for å benytte oss av operasjonene som objektet tilbyr. Ofte brukes ordet "informasjonsskjuling" (eng: information hiding) synonymt med innkapsling. Innkapsling letter utviklingen av store programmer. Systemet kan settes sammen av en mengde objekter, hvor hvert objekt ikke trenger å vite noe annet om de andre objektene enn hvilke grensesnitt de tilbyr. Merk at strukturering av koden i metoder også gir innkapsling innenfor ett objekt. Metodesignaturen, som for eksempel i Java består av metodenavn, et antall parametre med angitte typer og typen på metodereturen, kan sees på som metodens grensesnitt. Dermed kan ulike metoder i samme objekt samarbeide utelukkende på grunnlag av hverandres grensesnitt.

Hvis vi ikke kan arve synkroniseringskode, må synkroniseringsbegrensningene reimplementeres i subklassen. Dette medfører mer arbeid og større sannsynlighet for at programmeringsfeil oppstår. Så det er ønskelig å arve synkroniseringsbegrensninger så ofte som mulig. Videre kan det være nødvendig å endre synkroniseringsbegrensningene i subklassen. Følgelig ønsker vi muligheten til å *endre synkroniseringsbegrensningene inkrementelt*. Ikke alle objektorienterte språk har muligheten for dette.

Noen mener at ved arv så bør subklassen være en *subtype* av superklassen. Subtypeforholdet mellom klasser er definert slik: En klasse K' er en subtype av klassen K hvis en instans av klassen K' kan erstatte en instans av klassen K. Dette kalles substitusjonsprinsippet.

Har vi subtyping når synkroniseringsbegrensningene er sterkere i subklassen enn i superklassen? Da er det mulig at en melding som ville blitt akseptert av et objekt av superklassen ikke blir akseptert av et objekt av subklassen, noe som ikke følger substitusjonsprinsippet. For å få subtyping i denne betydningen av ordet, må synkroniseringsbetingelsene tvert imot svekkes i subklassen. Det vanligste er

at synkroniseringsbetingelsene styrkes ved arv. Det er mulig å skille mellom syntaktisk og semantisk subtyping. Syntaktisk subtyping handler om å unngå typefeil ved at et objekt forstår alle meldingene det får. Semantisk subtyping fokuserer på at subklassen skal holde seg innenfor superklassens semantikk, og i denne sammenhengen er det riktig å styrke synkroniseringsbegrensninger.

2.4 Arveanomali

Begrepet *arveanomali* ble lansert i 1993 av Satoshi Matsuoka og Akinori Yonezawa [19]. I denne seksjonen gjør jeg rede for hva arveanomali er, basert på [11, 19, 20].

2.4.1 En utfordring for det objektorienterte paradigmet

Arveanomali er et problem som oppstår i noen tilfeller ved arv av synkroniseringskode. Kort sagt består problemet i at det er umulig å tilføre ny synkroniseringskode i systemet uten at det nødvendiggjør ikke-trivielle redefinisjoner av metoder. Artikkelen [19] gir tre eksempler på hvor alvorlige konsekvensene er:

- Definisjon av en ny subklasse K' av en klasse K gjør det nødvendig å redefinere metoder arvet fra K og fra klasser over K i arvehierarkiet.
- Endring av en metode i en klasse K i arvehierarkiet medfører endring av tilsynelatende urelaterte metoder i klasser både over og under K i arvehierarkiet.
- Definisjon av en metode tvinger andre metoder til å følge en protokoll som de ikke måtte ha fulgt hvis metoden ikke hadde eksistert.

I disse situasjonene er mye av fordelen med arv tapt. Redefineringene krever at vi setter oss inn i implementasjonsdetaljene i andre klasser, hvilket bryter med innkapslingsprinsippet. Det blir vanskelig å gjenbruke kode fra eksisterende klasser når vi ønsker å endre et system inkrementelt. Problemet er så fundamentalt at noen språk for samtidige, objektorienterte systemer har gått bort fra arv i det hele tatt [19].

Forekomstene av arveanomali deles opp i tre klasser, basert på årsaken til at redefinering blir nødvendig:

- Partisjonering av aksepterende tilstander.
- Historiesensitive aksepterende tilstander.
- Endring av aksepterende tilstander.

Matsuoka og Yonezawa peker på at en eventuell forekomst av arveanomali er avhengig av synkroniseringsstrategien til implementasjonsspråket [19]. Det vil si at klasser som ikke kan arves uten redefinisjoner i ett språk, kan arves uten problemer i et annet språk med en annen synkroniseringsstrategi. Problemet med arveanomali er av en slik natur at det generelt ikke er mulig å påstå formelt at et språk unngår eller løser problemet. For det første er problemet alltid knyttet opp til det spesifikke språkets synkroniseringsprimitiver og for det andre kan det vise seg at nye former for anomali oppdages i framtiden.

Framgangsmåten for å vise hvordan et språk håndterer arveanomali er å vise hvordan språkets synkroniseringsstrategi takler konkrete eksempler som er representative for de klassene av anomali som er kjente per i dag. Det blitt utviklet strategier som begrenser problemet, for eksempel AOP og Creol. Det ser ut til å være enighet om at en løsning må innebære å skille synkroniseringskoden fra forretningskoden på en eller annen måte.

2.4.2 Tre klasser av arveanomali

Vi ser nå på de tre klassiske eksemplene på arveanomali. Vi fortsetter på buffer-eksemplet som ble introdusert tidligere i kapitlet. Poenget med disse eksemplene er å vise hvordan arv av synkroniseringskode fører til programmeringsproblemer med utilfredsstillende løsninger.

Partisjonering av aksepterende tilstander

Det er mulig å bruke en synkroniseringsstrategi som baserer seg på å

1. finne relevante tilstandsmengder og kategorisere dem ved å gi dem navn som "full", "empty" og "partial",
2. angi for hvilke tilstandskategorier en melding er akseptabel, eventuelt ikke er akseptabel, og
3. la hver metode angi hvilken tilstandskategori objektet har når metoden terminerer.

Dette kan se slik ut, med en Java-inspirert syntaks:

```
class Buffer {
  put: requires not full
  get: requires not empty
  ...

  void put(Object el) {
    ...
    if (<bufferen er nå full>) become full;
    else become partial;
  }

  Object get() {
    ...
    if (<bufferen er nå tom>) become empty;
    else become partial;
  }
}
```

Strategien leder til arveanomali i følgende tilfelle. Vi definerer en klasse 2Buffer som arver Buffer. Den utvider Buffer ved å tilby en metode get2, som henter to elementer fra bufferen om gangen. Vær oppmerksom på at en klient av bufferen ikke kan oppnå det samme med to suksessive kall på get(). Det er fordi vår kontekst er et samtidig system, hvilket innebærer at meldinger fra andre objekter kan bli akseptert av bufferen mellom klientens første og andre kall.

Hvordan skal vi angi hvorvidt en get2-melding kan aksepteres? Vi har ingen tilstandskategori som forteller oss at vi har minst to elementer i bufferen. Vi er nødt til å angi dette eksplisitt, hvilket innebærer en partisjonering av tilstandskategorien "partial" til to kategorier "one" og "twoOrMore":

```
class 2Buffer extends Buffer {
  get2: requires twoOrMore

  List get2() {...}

  void put(Object el) {
    ...
    if (<bufferen er nå full>) become full;
    else if (<bufferen har nå to eller flere elementer>) become twoOrMore
    else become one;
  }

  Object get() {
    ...
    if (<bufferen er nå tom>) become empty;
    else if (<bufferen har minst to elementer>) become twoOrMore
    else become one;
  }
}
```

Poenget her er at definisjon av en ny metode get2() tvinger fram ikke-trivielle redefinisjoner av de eksisterende metodene put() og get(). Det løser for så vidt problemet, men ikke på en spesielt god måte, siden vi ikke ønsker å måtte gjøre slike redefinisjoner.

Problemet med partisjonering av aksepterende tilstander lar seg løse bedre med en synkroniseringsstrategi basert på vakter. Som et eksempel på en vakt, bruker vi for enkelhets skyld en naiv Java-implementasjon

```
while (<betingelse>) wait();
```

hvor betingelsen er et predikat over objektets tilstand. Hittil har vi sett at et objekt sin tilstandskategori angis av programmereren med et navn som programmereren selv velger. En melding aksepteres hvis tilstanden er definert til å gjøre det. Når vi bruker vakter, aksepteres meldingen hvis vaktens betingelse er oppfylt.

```
class
Buffer { ...
  Object get() {
    while (numberOfElements == 0) wait();
    ...
    numberOfElements = numberOfElements - 1;
  }

  void put(Object el) {
    while (numberOfElements == bufferSize) wait();
    ...
    numberOfElements = numberOfElements + 1;
  }
}

class 2Buffer extends Buffer { ...
   List get2() {
     while (numberOfElements < 2) wait();
     ...
   }
}
```

Bruken av vakter gjør det mulig å angi synkroniseringsbetingelsen for hver enkelt metode på en enkel måte. Vi bruker ikke tilstandskategorier, og unngår dermed problemene med kategorisering av tilstandene.

Historiesensitive aksepterende tilstander

Vakter kan dessverre ikke håndtere alle former for arveanomali på en like pen måte. Anta at vi ønsker å utvide Buffer med en metode gget(), som gjør det samme som get, men hvor en gget-melding ikke kan aksepteres som den første meldingen etter en get-melding. Her er det ikke objektets tilstand som er avgjørende, men objektets meldingshistorie. Følgelig er objektet nødt til å registrere historien sin på en eller annen måte. Vi utvider objektets tilstandsrom med en boolsk variabel afterGet, som sier om den siste meldingen var en get-melding eller ikke. Dermed kan vi bruke vakter til å sjekke den nye variabelen.

```
class GBuffer extends Buffer {
  bool afterGet = false;

  Object gget() {
    while (numberOfElements == 0  or  afterGet) wait();
    ...
  }
```

```
  Object get() {
    ...
    afterGet = false;
  }

  void put(Object el) {
    ...
    afterGet = true;
  }
}
```

Vi ser at vi igjen er tvunget til å redefinere put() og get(), denne gangen for å registrere historien. Vi er nødt til å gjøre detaljerte endringer i arvede metoder som vi ikke hadde trengt å endre hvis vi ikke hadde lagt til den nye metoden gget(). Dette er en utilfredsstillende løsning sett fra et objektorientert standpunkt.

Endring av aksepterende tilstander

Den tredje typen arveanomali oppstår i forbindelse med bruk av såkalte "mix-in"-klasser. En mix-in-klasse er en klasse som ikke instansieres direkte, men arves for at adferden dens skal integreres i adferden til den arvende klassen. Den arvende klassen er ikke en subtype av mix-in-klassen.

Et eksempel på en mix-in-klasse er dette:

```
abstract class Lock {
  bool isLocked = false;

  void lock() {
    while (isLocked) wait();
    isLocked = true;
  }

  void unlock() {
    while (! isLocked) wait();
    isLocked = false;
  }
}
```

I seg selv gjør ikke denne koden noe meningsfullt. Vi bruker den til å utvide funksjonaliteten til vår klasse Buffer, slik at vi kan få et bufferobjekt med en lås som kan brukes til gjensidig utelukkelse av klienter. Vi setter klassen Lock inn i arvehierarkiet ovenfor Buffer:

```
class Buffer extends Lock { ... }
```

Synkroniseringsbegrensningen i dette eksemplet sier at bufferen ikke skal akseptere noen meldinger mens det er låst, med unntak av unlock-meldingen. Alternativt kan vi oppfatte kravet historisk, ved å definere det som at put- og get-meldinger bare kan aksepteres etter en unlock-melding og før en lock-melding. put() og get() er dermed historiesensitive på lignende måte som i gget-eksemplet ovenfor.

Vi er nødt til å endre vaktene til put() og get() slik at de tar låsen med i beregningen:

```
Object get() {
  while (numberOfElements == 0  or  isLocked) wait();
  isLocked = true;
  ...
}

void put(Object el) {
  while (numberOfElements == bufferSize  or  isLocked) wait();
  isLocked = true;
```

```
  ...
}
```

Hva skiller endring av aksepterende tilstander fra partisjonering av aksepterende tilstander og historiesensitive aksepterende tilstander?

I get2-eksemplet måtte vi dele opp objektets tilstandsrom i flere disjunkte deler i subklassen enn vi måtte i superklassen. Likevel vil put() og get() aksepteres av subklassen under essensielt de samme betingelsene som i superklassen, fordi unionen av de nye partisjonene er lik den opprinnelige tilstandsmengden.

I gget-eksemplet måtte vi redefinere put() og get() etter at vi utvidet klassen med gget(). Men de aksepterende tilstandene til put() og get() ble ikke berørt. put- og get-meldinger aksepteres i den nye GBuffer-klassen under de same betingelsene som i klassen Buffer.

I Lock-eksemplet endres betingelsene for å akseptere meldingene. De aksepterende tilstandene til put() og get() er ikke de samme etter at vi har arvet Lock.

2.5 Aspektorientert programmering

I denne seksjonen ser vi på et eksempel på relatert arbeid i problemområdet for denne oppgaven. Det aspektorienterte programmeringsparadigmet (AOP) foreslår en løsning på problemet med arveanomali. Vi skal se på tre varianter av aspektorientert programmering, hentet fra [20]. Vi starter med å forklare hva aspektorientert programmering er.

2.5.1 Hva er aspektorientert programmering?

AOPs historiske kontekst

Modulær programmering [22] er en måte å programmere på hvor programmet er delt opp i komponenter kalt moduler. Hver modul har ideelt sett en lett håndterlig størrelse, et klart definert formål/ansvarsområde og et klart definert grensesnitt mot andre moduler. Det eneste alternativet er monolittiske programmer. Det er for vanskelig å utvikle monolittiske programmer for komplekse problemer, så dermed er ikke spørsmålet om programmet skal være modulært, men hvilke kriterier som skal ligge til grunn for dekomponeringen av programmet til moduler.

I begynnelsen av programmeringens historie ble programmer skrevet direkte i maskinkode, noe som gjorde at programmerere brukte mer tid på å tenke på den spesifikke maskinens instruksjonssett enn på problemet som skulle løses. Så kom språk på høyere nivå, som gjorde det mulig å abstrahere vekk fra den underliggende maskinen. Så kom strukturerte programmer som gjorde det mulig å dekomponere problemene i form av prosedyrer.

David Parnas var i 1972 den første til å foreslå at et av de viktigste kriteriene for dekomponering burde være skjuling av informasjon (eng: information hiding) [21]. Inntil da hadde dekomposisjonen vært gjort ad-hoc, eller på basis av stadier i prosesseringen, og fordelene med modulær programmering var ikke særlig store. Etter dette har det blitt vanlig å dekomponere programmer på basis av abstrakte datatyper og objektorientering. Det objektorienterte paradigmet sin vektlegging på innkapsling og delegasjon er konsistent med prinsippet om skjuling av informasjon. Objektorientering har gjort det mulig å lage programmer som løser mer komplekse problemer. Dette har igjen gjort at systemutviklere har påtatt seg å løse enda større problemer, slik at objektorienterte programmer likevel kan bli uhåndterlige.

Evolusjonen av metodologien fra maskinkode, maskinuavhengig kode, prosedyrer, klasser og så videre har vært en utvikling av måter å dekomponere problemer på. Hver ny metodologi har lettet overføringen fra systemkrav til programmeringskonstrukter. Dermed har det blitt mulig å utvikle systemer som løser mer komplekse problemer. Aspektorientert programmering [9,10,16,18,20] (AOP) kan sees på som et nytt trinn i den historiske utviklingen av den generelle programutviklingsmetodologien. AOP har en tilnærming til dekomponering av programmer som bygger på erfaringer med objektorientert programmering. Konseptet AOP ble først introdusert av Gregor Kiczales med flere [16].

AOPs filosofi

AOP velger å se på et system som en løsning på flere problemer, eller sagt på en annen måte: Systemet må ta seg av et antall hensyn (eng: a concern). Et *kjernehensyn* er et av systemets mest sentrale oppgaver. Summen av alle kjernehensynene utgjør systemets forretningslogikk, og implementeres av forretningskoden.

Det er vanligvis et antall andre hensyn som også må adresseres for at programmet skal kunne kjøre korrekt. Et typisk eksempel er logging. I et objektorientert system vil vi ideelt sett ha høy kohesjon, det vil si at klassene har et klart definert og begrenset ansvarsområde innenfor systemet. Likevel vil klassene i praksis ofte dele sekundære krav med andre klasser. Vi sier at disse kravene går på tvers av de primære kravene, fordi: Objektorienteringen går ut på å skille ut felles adferd og å skyve den høyest mulig opp i arvehierarkiet. Sekundære hensyn blir vanskelig å implementere innenfor den "vertikale" strukturen av klasser, da kodens struktur ikke er designet for å ta seg av disse hensynene. Implementasjonen av disse sekundære hensynene blir da spredt utover et antall klasser, noe som gjør koden uoversiktlig. Vi kaller dem for *tverrgående hensyn* (eng: cross cutting concerns). Et eksempel vil være at hvis vi skriver en applikasjon for å håndtere medisinske pasientjournaler, så vil bokholderiet og indekseringen av disse journalene være et kjernehensyn, mens vi har tverrgående hensyn som å logge historien av endringer i databasen og autentisering av brukere.

Prosedyreorienterte programmeringsspråk som C og Pascal separerer hensyn ved å implementere håndteringen av dem i prosedyrer. Objektorientert språk som Java og C# separerer hensyn i klasser i tillegg. Et av de vanligste AOP-språkene, AspectJ, separerer hensyn i et konstrukt som kalles et *aspekt*. Dette reflekterer hovedideen bak AOP, som er å identifisere ulike aspekter ved et system, og å implementere hvert aspekt i sin egen modul. Dette står ikke i motsetning til objektorientering og kan implementeres som en utvidelse av et objektorientert språk.

Hvis vi begrenser oss til domenet distribuerte systemer, har de viktigste aspektene blitt identifisert til å være forretningslogikk, objektets lokasjon, kommunikasjon og synkronisering [16].

Utvikling av et system ved hjelp av AOP-metodologi har tre faser:

1. Dekomposisjon til aspekter. Systemkravene dekomponeres slik at kjernehensyn og tverrgående hensyn blir identifisert.
2. Implementasjon av hensyn. Hvert hensyn implementeres separat. Dette gjøres ofte i et eget språk. ADLs (Aspect Description Language) er en håndfull språk som brukes sammen med språket AspectJ og er designet for å kunne uttrykke de ulike vanlige aspektene. Hvert aspekt har sitt ADL for at aspektet skal kunne implementeres på en naturlig og kraftfull måte.
3. Rekomposisjon av aspekter. I denne fasen brukes en aspektvever, som er et program som ligner på en kompilator. Aspektene oversettes til det samme språket, og koden for aspektene integreres automatisk til å bli ett program.

Inkrementell endring av modulære programmer bør passe sammen med inkrementell og modulær resonnering, og dette er et tema i Creol-prosjektet, om enn ikke i denne oppgaven. Modulær resonnering omkring AOP-programmer er beskrevet i [17].

2.5.2 AOP sin tilnærming til arveanomali

Vi skal nå se på tre ulike varianter av AOP-språk. Alle variantene er hentet fra [20], som også inneholder korte programeksempler. I denne gjengivelsen begrenser jeg mengden av programkode, da koden ville tatt uforholdsmessig stor plass.

Synchronization patterns

Denne varianten skiller forretningskode fra synkroniseringskode ved å plassere synkroniseringskoden i en egen modul, syntaktisk adskilt fra klassedefinisjonen. Denne modulen kalles et "synkroniseringsmønster" (eng: synchronization pattern). I synkroniseringsmønstret spesifiseres synkroniseringsbegrensningen i et eget metaspråk. Et synkroniseringsmønster har følgende syntaks, bestående av fire blokker:

```
sync_pattern navnPåMønstret
  add_structure ...
```

```
  add_func ...
  mutex ...
  sync ...
```

Blokkene `add_structure`, `add_func` og `mutex` inneholder deklarasjoner av henholdsvis datastrukturer, operasjoner på disse og låser, som alle brukes av synkroniseringsmekanismen. Synkroniseringsstrategien beskrives i `sync`. Her spesifiseres betingelsene for at en melding er akseptabel. Dette gjøres med et nøkkelord "requires", som i essens er en vakt. Vi har sett tidligere at vakter gjør det mulig å håndtere partisjonering av aksepterende tilstander på en ok måte. Men problemet med historiesensitive aksepterende tilstander har fremdeles ingen tilfredsstillende løsning. Synkroniseringsmønster er likevel en forbedring, fordi problemet begrenses ved at redefinisjoner bare finner sted i synkroniseringsmønstret.

Etter at programmereren har implementert synkroniseringsstrategien, oversettes synkroniseringsmønstret automatisk fra metaspråket sitt til et mer lavnivå språk, for eksempel det språket som forretningskoden er skrevet i, og de to aspektene blandes sammen til ett program.

Composition filters

Denne framgangsmåten baserer seg på en idé om filtre i tilknytning til et objekt, som meldinger må passere gjennom når de ankommer eller forlater objektet. Et sammensettingsfilter (eng: composition filter) utvider eller endrer funksjonaliteten til objektets metoder ved å forholde seg til en melding på en spesifisert måte. Dette gir muligheten for modularisering og separasjon av ulike hensyn, i tråd med hovedideen bak aspektorientert programmering. Det kan være flere filtre som en melding må passere gjennom, noe som gir muligheten for inkrementell utvidelse eller endring ved at synkroniseringsbegrensningene til objektet spesifiseres i flere filtre.

Et filter består av 1) en betingelse, som bestemmer når filteret skal brukes, 2) et mønster, som beskriver hvilke metoder filtret gjelder for og 3) en substitusjon, som beskriver hva som skal skje med meldingen. En melding blir enten akseptert og sendt videre til neste filter eller til objektet, eller meldingen lagres i en kø inntil den kan aksepteres. Filteret implementeres i grensesnittet til klassen som inneholder forretningslogikken.

Filtre gjør essensielt det samme som vakter. Arveanomalien viser seg derfor på samme måte, i form av historiesensitive aksepterende tilstander i gget-eksemplet. Synkroniseringsstrategien forholder seg til dette ved å bruke et konstrukt som kalles en ACT (abstract communication type). Dette er en egen klasse som supplerer forretningslogikklassen og dens filtre. En ACT har som ansvar å koordinere objekter på bakgrunn av meldingene de sender seg imellom. I vårt tilfelle vil vi bruke en ACT til å registrere interaksjonshistorien. Når filtret mottar en melding, vil det sende den videre til ACTen, som registrerer om det er en get-melding eller ikke.

Anomalien blir modularisert og begrenset til grensesnittene og ACTene. Nødvendige redefinisjoner foregår der, og klassen som inneholder forretningslogikken forblir urørt. Dette reduserer problemet med arveanomali.

Jeeg

Jeeg er en dialekt av Java som har en synkroniseringsstrategi som kan kalles aspektorientert. Synkroniseringsbegrensningene uttrykkes med metodevakter. Metodevakter er vanligvis implementert som en del av metodedefinisjonen. I Jeeg er metodevakter syntaktisk helt adskilt fra metodekroppen. Et typisk Jeeg-program har følgende form:

```
class Klassenavn {
  sync { metodenavn : <vakt>; ... }

  // standard Java klassedefinisjon
  metodenavn(...) {...}
  ...
}
```

Jeeg er spesielt interessant fordi vaktene uttrykkes i en variant av lineær temporallogikk. Dette gjør at vaktene kan uttrykke egenskaper basert på objektets historie. Vi kan bruke en metavariabel "event", som

er bundet til navnet på den siste metoden som ble kjørt. Dermed kan vi lage klassen GBuffer på denne måten:

```
class GBuffer extends Buffer {
  sync { gget: Previous(event != put)  &&  numberOfElements > 0 ; }

  Object gget() {...}
}
```

Med Jeeg er det mulig å unngå partisjonering av aksepterende tilstander fordi strategien baserer seg på vakter. Det er mulig å unngå de aller fleste historiesensitive aksepterende tilstander fordi vaktene uttrykkes med temporallogikk. Som tidligere nevnt, er det generelt umulig å påstå at et språk er immunt mot arveanomali, men siden temporallogikk er beskrevet formelt, er det mulig å angi nøyaktig hvilke tilfeller som Jeeg kan håndtere.

Figur 2.1: Tverrgående hensyn kan være ubehagelig å måtte forholde seg til.

Kapittel 3

Språket Creol

Språket Creol brukes til å modellere objektorienterte, åpne distribuerte systemer, og å eksperimentere med hvordan språklige konstrukter kan brukes i den sammenhengen. Formålet med dette er å komme fram til en velegnet kombinasjon av språklige konstrukter. Jeg skal nå redegjøre for deler av språket og noen av dets konstrukter. Spesielt skal vi se på den relevante delen av synkroniseringsstrategien til Creol. På websiden til Creol-prosjektet [26] finnes et antall artikler om språket Creol, blant annet [13–15].

3.1 Objektinteraksjon

Kommunikasjon mellom prosesser er fundamentalt for distribuerte systemer og derfor er programmeringskonstrukter for objektinteraksjon ett av flere sentrale temaer for Creol. Distribuerte systemer kan interagere på fire ulike måter. *Synkron meldingssending* bruker blokkerende primitiver for å sende en melding til et annet objekt over kommunikasjonskanalen. Primitivene blokkerer inntil mottageren er klar til å ta imot meldingen. *Asynkron meldingssending* bruker ikke-blokkerende sendeprimitiver. Mottak av meldinger er vanligvis blokkerende for mottakeren. *RPC (Remote Procedure Call)* er blokkerende kommunikasjon over en toveis-kanal. *Rendezvous* er en tilnærmingsmåte som baserer seg på en input- eller accept-setning som fører til at prosessen venter på at en metode skal kalles. Når metodekallets melding ankommer, evalueres metoden og resultatet sendes tilbake.

Objektorientering er den ledende tilnærmingsmåten når man skal implementere distribuerte systemer. Objekter interagerer vanligvis ved hjelp av eksterne metodekall (RPC eller RMI). Metodekall er vanligvis synkrone, fordi mekanismene har blitt utviklet med utgangspunkt i sekvensielle (ikke-samtidige) systemer. Synkrone, eksterne kall i en distribuert kontekst kan lettere gi opphav til ukontrollert venting og vranglås (deadlock) enn i ikke-distribuerte, sekvensielle systemer. For å unngå at et objekt blokkerer, er det vanlig å la objektet starte egne eksekveringsstråder som er dedikerte til å håndtere eksterne metodekall. Dette fører til en mer lavnivå måte å programmere på enn ønskelig. Det er også ønskelig å ha bare én tråd om gangen kjørende i ett objekt. Skillet mellom eksekveringstråder og objekter bryter med modulariteten og innkapslingen som er så sentralt for det objektorienterte paradigmet. Synkrone kall er egnede for små, homogene systemer, men mindre egnet for typiske distribuerte systemer. Asynkron meldingssending er et alternativ til synkrone metodekall som kan gi bedre effektivitet og feiltoleranse, men som ikke tilbyr programmereren den samme nyttige strukturen og disiplinen som metodekall.

Det er et behov for *høynivå programmeringskonstrukter som forener objektorientering og distribusjon på en naturlig måte*. Det synes mest naturlig å modellere distribuerte systemer som objekter som kommuniserer asynkront. Creol-objekter er modellert som om de har sin egen prosessor, og bare én prosess kjører i et objekt om gangen. Kommunikasjonen er eksplisitt asynkron, hvilket betyr at distribusjonen ikke er helt transparent. Creol baserer seg på *asynkrone metodekall* og *eksplisitte prosessorslippunkter*. Prosessorslippunkter er steder i koden som tillater at den inneværende prosessen suspenderes, slik at en annen prosess kan få tilgang til prosessoren. Prosessorslippunkter angis eksplisitt i programkoden. I Creol kan mottak av meldinger gjøres asynkront, mens dette i mange andre språk er en blokkerende operasjon. Dette gjør det mulig å suspendere en prosess som venter på resultatet av et asynkront metodekall.

Creol sine konstrukter for objektinteraksjon forener fordelene med asynkron kommunikasjon på den ene siden og syntaksen til metodekall på den andre.

Syntaktiske kategorier.

$g \in \mathsf{Guard}$
$p \in \mathsf{MtdCall}$
$s \in \mathsf{Stm}$
$t \in \mathsf{Label}$
$v \in \mathsf{Var}$
$e \in \mathsf{Expr}$
$x \in \mathsf{ObjExpr}$
$b \in \mathsf{Bool}$
$m \in \mathsf{Mtd}$

Definisjoner.

$g ::= \mathit{wait} \mid b \mid t? \mid g_1 \wedge g_2 \mid g_1 \vee g_2$
$p ::= x.m \mid m@\mathit{classname} \mid m$
$\mathrm{S} ::= s \mid s; \mathrm{S}$
$s ::= \mathbf{skip} \mid (\mathrm{S}) \mid \mathrm{S}_1 \square \mathrm{S}_2 \mid \mathrm{S}_1 \| \mathrm{S}_2 \mid \mathrm{S}_1 \& \mathrm{S}_2$
$\mid \mathrm{V} := \mathrm{E} \mid v := \mathbf{new}\ \mathit{classname}(\mathrm{E})$
$\mid \mathbf{if}\ b\ \mathbf{then}\ \mathrm{S}_1\ \mathbf{else}\ \mathrm{S}_2\ \mathbf{fi}$
$\mid \mathbf{while}\ b\ \mathbf{do}\ \mathrm{S}\ \mathbf{od}$
$\mid t!p(\mathrm{E}) \mid !p(\mathrm{E}) \mid p(\mathrm{E}; \mathrm{V}) \mid t?(\mathrm{V})$
$\mid \mathbf{await}\ \ g \mid \mathbf{await}\ \ g \wedge t?(\mathrm{V}) \mid \mathbf{await}\ \ g \wedge p(\mathrm{E}; \mathrm{V})$

Figur 3.1: Utdrag av Creol sin syntaks for metodedefinisjoner, med typiske termer fra hver kategori. Termer med store bokstaver slik som S, V, og E representerer lister, mengder eller multimengder av den gitte syntaktiske kategorien, tilsvarende den lille bokstaven.

3.2 Syntaks og semantikk

3.2.1 Syntaks

Vi ser på Creol sin syntaks. Grensesnitt er viktige i Creol, men er ikke relevante for vår problemstilling. Jeg kommer derfor ikke til å beskrive grensesnitt og bruken av dem. Det kan nevnes at objektvariabler (-pekere) types med grensesnitt og at Creol tillater multippel arv av grensesnitt så vel som klasser.

En Creol-klasse defineres slik:

class <klassenavn> (<klasseparametre>)
  inherits <klassenavn> <klassenavn>...
  implements <grensesnittnavn> <grensesnittnavn>...
begin
  var <variabeldeklarasjoner>
  op <metodenavn>(**in** <innparam> **out** <utparam>) == <metodekropp>
  op <metodenavn>(**in** <innparam> **out** <utparam>) == <metodekropp>
  ...
with <grensesnittnavn> <grensesnittnavn>...
  op <metodenavn>(**in** <innparam> **out** <utparam) == <metodekropp>
  op <metodenavn>(**in** <innparam> **out** <utparam>) == <metodekropp>
  ...
...
end

Etter nøkkelordet **inherits** følger en liste av parametriserte navn på superklasser. Etter **implements** følger en oppramsing av de grensesnittene som klassen støtter. Variabeldeklarasjonene etter nøkkelordet **var** angir de globale variablene/attributtene til klasseinstansene.

Etter **with** følger et antall grensesnittnavn som fungerer som kogrensesnitt. Kogrensesnitt angir hvilke typer av objekter som objekter av denne klassen er villig til å ta imot meldinger fra. Hvis A kaller en metode i B, vil B kunne gjøre typeriktige kall på A. Dette åpner for protokollaktig interaksjon. Termen "Any" istedenfor en liste med grensesnittnavn angir at vi aksepterer meldinger fra alle objekter. Vi vet da ikke noe om kaller, og kan dermed ikke gjøre kall tilbake. Hvis vi ønsker at en metode skal være skjult/privat, utelukkes metoden fra klassens grensesnitt. Det er derfor ikke nødvendig å deklarere metoder som private. De private metodene plasseres etter **begin** og før **with** i klassedefinisjonen.

En metode deklareres med en liste av innparametre og en liste av utparametre. En metodekropp kan innledes med deklarasjoner av lokale variabler. Et utsnitt av syntaksen for metodedefinisjoner vises i figur 3.1.

3.2.2 Prosessorslippunkter og vakter

Creol-objekter modellerer distribuerte objekter som har sin egen dedikerte prosessor og som eksekverer et antall prosesser. I Creol er alle prosesser i objektet kjørende metoder. Sagt på en annen måte: En prosess er en metodeinstans eller en rest av en metodeinstans. Et objektet har attributter som er tilgjengelige for alle prosesser. I tillegg har hver prosess sine egne lokale variabler. Objekter kan ha både aktive og reaktive prosesser. En reaktiv prosess er en metode som instansieres og kjøres utelukkende på grunn av meldinger fra andre objekter. En aktiv prosess kjører vanligvis fra det øyeblikket objektet opprettes, og gjør kall på andre objekter. Kjøringen av prosesser flettes som følge av prosessorslippunkter.

Potensielle prosessorslippunkter angis eksplisitt i programkoden ved bruk av await-setninger. Setningen *await* G vil føre til at setningens prosess suspenderes hvis vakten G evaluerer til usann. Det er dette som menes med at prosessoren "slippes". Etter at prosessoren har blitt frigjort, skeduleres en vilkårlig, ventende prosess. Hvis en vakt er sann, sier vi at vakten er *beredt*. En setningssekvens eller en prosess er beredt hvis dens første setning er beredt.

Typen Guard er definert induktivt slik:

- *wait* $\in$ Guard (ubetinget slippunkt),
- $t? \in$ Guard, hvor $t \in$ Label,
- $b \in$ Guard, hvor b er et boolsk uttrykk over objektets tilstand,
- $g_1 \wedge g_2$, hvor $g_1, g_2 \in$ Guard.

Setningen **await** *wait* vil alltid føre til at prosessoren slippes. Svarvakten $t?$ er beredt hvis det har ankommet et svar på kallet merket t. Evalueringen av vakter er atomisk og uten sideeffekter. **await** $g \wedge t?(\text{V})$ er en forkortelse for **await** $g \wedge t?$; $t?(\text{V})$, og vi lar **await** $g \wedge p(\text{E};\text{V})$, hvor p er et metodekall (eksternt eller internt), være en forkortelse for $t!p(\text{E})$; **await** $g \wedge t?(\text{V})$ for en ubrukt merkelappverdi t.

Slippunkter gjør at objektet ikke trenger å blokkere mens det venter på metoderetur. Andre prosesser som venter på å få kjøre kan skeduleres og nye metoder kan kalles mens vi venter. Hvis det kallede objektet aldri svarer, unngår vi derfor at hele objektet går i vranglås, ettersom annen aktivitet er mulig. Når svaret ankommer, må prosessen konkurrere med andre beredte prosesser om å få fortsette å kjøre.

3.2.3 Metodekall og objektinteraksjon

Vi ser på de ulike typene metodekall i Creol. En oversikt over de ulike kallene vises i figur 3.2 på neste side som et supplement til BNF-beskrivelsen i figur 3.1 på forrige side og den følgende teksten. Metoder kan kalles synkront eller asynkront og interne metoder kan i tillegg kalles virtuelt eller statisk. Vi skal se på forskjellen på disse fire. Metoderetur kan mottas blokkerende eller ikke-blokkerende.

Det er vanlig i objektorienterte språk at metoder kan redefineres i subklasser, og i et arvehierarki kan vi da ha et antall implementasjoner av den samme metoden. Et *virtuelt kall* bindes til en metodedefinisjon automatisk og usynlig for programmereren. Hvilken metodedefinisjon som faktisk velges er betinget av det kalte objektets arvehierarki og kjøretidssystemet. I *statiske kall* i Creol, derimot, angir programmereren eksplisitt hvilken klasse som metodedefinisjonen skal hentes fra.

Synkrone metodekall blokkerer inntil metodereturen foreligger, på samme måte som RPC (Remote Procedure kall). Synkrone kall har formen $p(\text{E};\text{V})$, som er en forkortelse for $t!p(\text{E});t?(\text{V})$, hvor t er en variabel som inneholder en hittil ubrukt merkelappverdi.

Returverdier fra kallet hentes eksplisitt til variabellisten V med setningen $t?(\text{V})$. Hvis et svar har ankommet, så blir returverdiene tilordnet V og kjøringen fortsetter. Hvis svaret ikke har ankommet, blokkerer prosessen. Ved lokale, synkrone kall slippes prosessoren midlertidig, ved at den kallende prosessen suspenderes ubetinget og instansen til den kalte metoden skeduleres. Ved terminering av den kalte prosessen, reskeduleres den kallende prosessen.

Merk at gjensidige, eksterne, synkrone kall vil føre til vranglås for de kallende prosessene, om enn ikke for objektene deres.

Asynkrone kall er derimot ikke blokkerende. Et asynkront metodekall gjøres med setningen $t!x.m(\text{E})$, hvor $t \in$ Label gir en lokalt unik referanse til kallet, x er en objektpeker, m et metodenavn og E en liste reelle parametre. Merkelapper kan unnlates hvis man ikke ønsker et svar.

$!m(\text{E})$	Asynkront, lokalt, virtuelt kall
$!x.m(\text{E})$	Asynkront, fjernt kall
$!m@c(\text{E})$	Asynkront, lokalt, statisk kall
$t!m(\text{E})$	Med merkelapp, for å kunne identifisere svarmeldinger.
$t!x.m(\text{E})$	
$t!m@c(\text{E})$	
$t?(\text{V})$	Synkront (blokkerende) mottak av metoderetur.
	Returverdiene fra kallet merket t tilordnes variablene i V.
await $t?(\text{V})$	Asynkront (ikke-blokkerende) mottak av metoderetur.
$t!m(\text{E});\ t?(\text{V})$	Synkront, lokalt, virtuelt kall.
$t!x.m(\text{E});\ t?(\text{V})$	Synkront, fjernt kall.
$t!m@c(\text{E});\ t?(\text{V})$	Synkront, lokalt, statisk kall.
$m(\text{E};\text{V})$	$= t!m(\text{E});\ t?(\text{V})$
$x.m(\text{E};\text{V})$	$= t!x.m(\text{E});\ t?(\text{V})$
$m@c(\text{E};\text{V})$	$= t!m@c(\text{E});\ t?(\text{V})$
await $g \wedge m(\text{E};\text{V})$	$= t!m(\text{E});$ **await** $g \wedge t?(\text{V})$
await $g \wedge x.m(\text{E};\text{V})$	$= t!x.m(\text{E});$ **await** $g \wedge t?(\text{V})$
await $g \wedge m@c(\text{E};\text{V})$	$= t!m@c(\text{E});$ **await** $g \wedge t?(\text{V})$

Figur 3.2: Creols ulike metodekall. Variablene er de samme som i figur 3.1 på side 20. Lokale og statiske kall er alltid interne. Fjerne kall er eksterne med unntak av når x evaluerer til det kallende objektets navn.

Den kallende prosessen kan fortsette kjøringen etter kallet. Eventuell metoderetur mottas eksplisitt på et senere tidspunkt. Metoderetur kan taes imot asynkront ved å kombinere svarsetningen med en await-setning i *await* $t?(\text{V})$. Betydningen av dette er at hvis svaret ikke har ankommet, så suspenderes prosessen som venter på svaret, istedenfor å blokkere. Hvis metodereturen fra det asynkrone kallet ikke foreligger når den etterspørres, vil prosessen blokkere eller suspendere seg selv, avhengig av om programmereren angir synkront eller asynkront mottak.

Metoder kan alltid kalles asynkront, fordi det kallede objektet ikke har anledning til å blokkere kommunikasjon. Forbikjøring av metoder (eng: method overtaking) er tillatt, det vil si at hvis metoder kalles hos et objekt i én rekkefølge, så kan objektet starte kjøringen av metodeinstansene i en annen rekkefølge.

De engelske ordene "local " og "remote methodcall" kan kanskje oversettes med *lokale* og *fjerne metodekall*. Dette må vi skille fra det vi kan kalle *interne* og *eksterne metodekall*. Det første ordparet henviser til syntaksen på kallene, mens det andre ordparet forteller hvorvidt det kalte og det kallende objektet er det samme. Et lokalt kall er uten objektidentifikator, mens et fjernt kall er *med* objektidentifikator. Et internt kall refererer til en metode som er definert i det kallende objektet sin klasse, mens et eksternt kall refererer til en metode i et annet objekt. Alle statiske kall er lokale, og alle lokale kall er interne. Men ikke alle fjerne kall er eksterne. Hvis x i $x.m(\text{E})$ refererer til det samme objektet som kallet gjøres fra, så er kallet internt. Legg merke til at lokale kall kan være asynkrone.

Siden lokale kall ikke trenger noen objektidentifikator som prefix, kan de identifiseres syntaktisk som interne. Nøkkelordet *this* brukes for at et objekt skal kunne referere til seg selv, og i spesifikasjonen av den operasjonelle semantikken er $m(\text{E})$ definert som $this.m(\text{E})$. Dermed har vi at alle ikke-statiske kall behandles uniformt av interpreten som om de er eksterne.

3.2.4 Operatorer for intern kontrollflyt

Intern kontrollflyt styres av prosessorslippunkter sammen med andre konstrukter. I tillegg til if- og while-setninger, har Creol et par andre konstrukter for kontrollflyt som ikke er like vanlige, nemlig operatorene $\|$, $\square$ og & . $\|$ og $\square$ har fått sin operasjonelle semantikk formelt spesifisert i Creol-interpreten. Dette

kommer vi tilbake til i 4.2.15 på side 46. Inntil da nøyer vi oss med de uformelle og intuitive definisjonene nedenfor.

I setningene $S1 \| S2$, $S1 \square S2$ og $S1 \& S2$, hvor $S1$ og $S2$ er vilkårlige setningssekvenser, sier vi at $S1$ og $S2$ er "ledd" i henholdsvis |||-, □- eller & -setningen. Det kan være vilkårlig mange ledd i alle de tre setningstypene.

Setningen $S1 \| S2$ innebærer en fletting av de to kodesekvensene. Det betyr at hvis vi under kjøring kommer til en setning i $S1$ som ikke er beredt, så kjører vi $S2$ istedenfor, og omvendt. Det betyr at vi kan få en veksling fram og tilbake mellom å kjøre (delsekvenser av) $S1$ og $S2$, inntil en av sekvensene er tomme. Hvis ingen av leddene i |||-setningen er beredte, er ikke |||-setningen beredt, og prosessen suspenderes.

Setningen $S1 \square S2$ innebærer et ikke-deterministisk valg av et av leddene. Et vilkårlig, beredt ledd velges ut for kjøring, mens resten av leddene ignoreres. Alternativt kan vi forstå betydningen av setningen som at den erstattes av et av sine beredte ledd. Hvis ingen ledd er beredte, er □-setningen ikke beredt, og prosessen suspenderes.

Setningen $S1 \& S2$ innebærer synkronisert fletting av leddene. Med dette menes at leddene skal flettes hvis alle leddene er beredte samtidig. Hvis ikke alle leddene er beredte samtidig, er ikke &-setningen beredt, og prosessen suspenderes. Synkronisert fletting er foreslått og uformelt definert i [8, 15] på en måte som bevisst gir få føringer for Maude-implementasjon av den operasjonelle semantikken:

> La $S1$ og $S2$ være vilkårlige setningslister og $G1$ og $G2$ vakter. Synkronisert fletting — (**await** $G1; S1$) & (**await** $G2; S2$) — er definert som **await** $G1 \wedge G2; S1; S2$. Ikke-bevoktede argumenter til & behandles som om de er bevoktet av en sann vakt. Metodekall ekspanderes.

Hva vil det si at metodekall ekspanderes? I en setning (**await** $G1; S1$) & $m(\text{E}; \text{V})$ hvor metoden m() sin metodekropp er **await** $G2; S2$, må konjunksjonen av $G1$ og $G2$ være sant i kontekst av objektattributtene, listen av reelle parametre E og objektets meldingskø før kjøringen kan begynne. Merk at når m() instansieres, så inngår verdiene i E i m() sine lokale variabler. Det betyr at $G2$ kan være betinget av parametrene i kallet.

Vi ser at i definisjonen ovenfor settes $S1$ og $S2$ sammen med ;-operatoren for sekvensiell sammensetting. Som et alternativ til dette har det blitt foreslått å bruke operatoren $\|$ for ikke-deterministisk fletting. Det vil kunne gi en liten effektivitetsfordel. I tillegg kunne vi trenge å presisere at ekspansjonen bare gjelder kall som er synkrone og interne. Det er ikke aktuelt å ekspandere eksterne kall, fordi da skal den kalte metoden kjøre i et annet objekt. Hvert objekt har eksklusivt ansvar for sine egne prosesser, og objektene samhandler utelukkende på bakgrunn av grensesnitt og meldingsutveksling. Det er mulig å se for seg ekspandering av asynkrone, interne kall, men det er kunstig. Det ligger i naturen til asynkrone kall at eksekveringstidspunktet ikke er kritisk, så vi trenger neppe å synkronisere på dem.

Vi kommer foreløpig til å bruke denne definisjonen av synkronisert fletting:

> Synkronisert fletting — (**await** $G1; S1$) &(**await** $G2; S2$) — er definert som **await** $G1 \wedge G2; (S1 \| S2)$. Ikke-bevoktede argumenter til & behandles som om de er bevoktet av en sann vakt. Metodekall som er synkrone og interne ekspanderes.

3.3 Creol sin tilnærming til arveanomali

Vi skal nå se på Creol sin synkroniseringsstrategi. For å demonstrere den, bruker vi de klassiske arveanomali-eksemplene som vi så i seksjon 2.4.2 på side 10. De samme eksemplene (med små endringer i koden) vil bli analysert i kapittel 6. Bruk av &-operatoren har tidligere blitt vist i [8, 15].

Vi starter med å definere bufferklassen i Creol, som inneholder metodene put() og get():

```
class Buffer(bufferSize: Nat)
begin
  var buf: List, noElmts: Nat .
with Any .
  op put(in el: Data) == await noElmts < bufferSize;
       buf := buf + el;
       noElmts := noElmts + 1 .
  op get(out el: Data) == await noElmts > 0;
```

```
        el := head(buf);
        buf := remove(buf, head(buf));
        noElmts := noElmts - 1 .
end
```

Vi bruker await-setninger for å *utsette* kjøringen av put() hvis bufferen er full og for å utsette get() hvis bufferen er tom.

3.3.1 Partisjonering av aksepterende tilstander

Vi lar en ny klasse 2Buffer arve klassen Buffer, og legger til metoden get2():

```
class 2Buffer(2bufferSize: Nat) inherits Buffer(2bufferSize)
begin with Any .
  op get2(out el1: Data, el2: Data) === await noElmts >= 2;
       get@Buffer(;el1);
       get@Buffer(;el2) .
end
```

Vi bruker en await-setning til å utsette kjøringen av get2() til det finnes minst to elementer i bufferen. Så bruker vi @-konstruktet for å kalle get() metoden slik den er definert i superklassen.

Bruken av statiske kall

Det er ikke nødvendig å bruke @ her, da klassen 2Buffer tilbyr get() via arv. Det vil si at virtuell binding er garantert å velge denne definisjonen av metoden. Jeg bruker den likevel, fordi & og @ er en god kombinasjon blant annet med tanke på resonnering: Det er lettere å resonnere omkring statiske kall enn virtuelle, da vi ser umiddelbart hvilken metodedefinisjon som blir instansiert.

I programeksemplet ovenfor, gjør bruken av @ at det blir mulig å redefinere get() i klassen 2Buffer eller en subklasse av 2Buffer på et senere tidspunkt, uten at implementasjonen av get2() blir berørt. Det kan diskuteres hvorvidt dette er ønskelig, men muligheten er der.

Creol støtter multippel arv av klasser. I arbeidet med utviklingen av definisjonen av synkronisert fletting har jeg simulert kjøring av Creol-programmer som er oversatt til CMC-kode (mer om det senere). Ved ett tilfelle hadde programmet uventet adferd på grunn av et uventet resultat av den dynamiske bindingen av et metodekall. Det fantes to metoder med samme navn i ulike grener i arvehierarkiet, og det var en annen metodedefinisjon som ble instansiert enn den jeg forventet. Løsningen på problemet var å gjøre kallet statisk istedenfor dynamisk.

Det ser ut som om Creol legger opp til en programmeringsstil hvor statiske kall brukes så ofte som mulig.

3.3.2 Historiesensitive aksepterende tilstander

Vi lar en ny klasse GBuffer arve 2Buffer, og definerer metoden gget() slik at denne ikke kan kjøres umiddelbart etter en get-operasjon. Vi er nødt til å registrere denne delen av meldingshistorien, og bruker variabelen afterGet til dette. I dette eksemplet ser vi hvordan &-operatoren brukes:

```
class GBuffer(gbufferSize: Nat) inherits 2Buffer(gbufferSize)
begin
  var afterGet: Bool=false .
with Any .
  op put(in el: Data) == put@Buffer(el;);
       afterGet := false .
  op get(out el: Data) == get@Buffer(;el);
       afterGet := true .
  op get2(out el1, el2: Data) == get2@2Buffer(;el1,el2);
       afterGet := true .
  op gget(out el: Data) == (await not afterGet) & get@Buffer(;el) .
```

```
        afterGet := true .
end
```

Merk at implementeringen av synkroniseringsbetingelsen i gget() tvinger fram en redefinisjon av *alle* de andre metodene i klassen. Skaden er likevel liten, fordi vi kan bruke @ til å kalle de respektive metodene slik de er definert i superklassen og enkelt utvide metodene med en tilordningssetning. Vi har innkapsling av superklassens forretningskode så vel som synkroniseringskode.

Det mest interessante for oss er å se på den første setningen i gget(). Setningen sier at vi skal forsterke get() sin skjulte synkroniseringsbetingelse med gget() sin eksplisitt angitte betingelse. Alternativt kan gget() defineres til å kalle get() i sin egen klasse, noe som kanskje gir enklere kode: (**await** not afterGet) & get@GBuffer(;el) .

3.3.3 Endring av aksepterende tilstander

Vi skal framprovosere anomalien "endring av aksepterende tilstander". Vi definerer en mix-in-klasse som kan brukes til å gi gjensidig utelukkelse av metoder i bufferklassen. Jeg velger å konsekvent bruke @ også når det ikke er nødvendig, av grunnene nevnt ovenfor.

```
class Lock
begin
  var lock: Oid=nullptr, noHasLock: Nat=0 .
  op waitSync(in origin: Oid) == await lock = origin .
  op waitFree() == await lock = nullptr .
with Any .
  op lock(in origin: Oid) == waitFree@Lock();
       lock := origin;
       noHasLock := noHasLock + 1 .
  op unlock(in origin: Oid) == waitSync@Lock(origin;);
       lock := nullptr;
       noHasLock := noHasLock - 1 .
end
```

Vi lar en ny klasse MutexBuffer arve GBuffer og Lock, for å gjøre bufferen låsbar. Her får vi se et godt eksempel på bruk av &-operatoren. Variabelen "caller" er en såkalt "read-only"-variabel som er lokal for metoden og som inneholder objektidentifikatoren til det objektet som gjorde metodekallet.

```
class MutexBuffer(mutexBufferSize:Nat) inherits GBuffer(mutexBufferSize), Lock
begin with Any
   op put(in el: Data) == lock@Lock(caller;) & put@GBuffer(el;);
        unlock@Lock(caller;) .
   op get(out el: Data) == lock@Lock(caller;) & get@GBuffer(;el);
        unlock@Lock(caller;) .
   op get2(out el1: Data, el2: Data) == lock@Lock(caller;) & get2@GBuffer(;el1,el2);
        unlock@Lock(caller;) .
   op gget(out el: Data) == lock@Lock(caller;) & gget@GBuffer(;el);
        unlock@Lock(caller;) .
end
```

Synkroniseringsbetingelsene i metodene fra GBuffer forsterkes av synkroniseringsbetingelsene i metodene fra Lock. Vi ser igjen at alle metodene blir berørt av anomalien og må redefineres. Men som tidligere, består redefineringen syntaktisk av en ren utvidelse. Programmereren trenger ikke vite noe om implementasjonen av metodene i GBuffer eller i Lock. Det eneste vi trenger å forholde oss til, er at det lages et aggregat av synkroniseringsbetingelsen fra hver av de to metodene i hver &-setning. Vi trenger ikke engang å vite om metodene er bevoktet, fordi ubevoktede metoder betraktes som bevoktet av en sann vakt og har derfor ingenting å si for sannhetsverdien til konjunksjonen av vakter.

3.4 Sammenligning av Creol og AOP

Vi har sett at Creol tilbyr inkrementell styrking av synkroniseringsbegrensninger. Hvis vi sammenligner Creol med AOP, ser vi at:

- Selv om en metode redefineres, har vi innkapsling av metodedefinisjonen i superklassen. I likhet med AOP, blir forretningskoden ikke berørt.
- I motsetning til noen AOP-teknikker, så blir ikke synkroniseringskoden i superklassen berørt heller.
- I motsetning til noen AOP-teknikker, så blir synkroniseringsbetingelsene implementert i det samme språket som brukes til forretningskoden. Det er derfor ingen kostnader med å oversette og veve synkroniseringskoden sammen med forretningskoden.

Hittil har vi bare sett at synkroniseringsbegrensinger styrkes i Creol. Dette er det vanlige, fordi man vanligvis må ta hensyn til invarianter i programmet. En fordel med å isolere synkroniseringskoden syntaktisk i en egen modul slik som i AOP, er at den kan endres vilkårlig og for eksempel svekkes istedenfor å styrkes. Hvordan kan vi få til inkrementell *vilkårlig* endring av synkroniseringsbegrensninger i Creol?

Vi oppnår dette ved å skille synkroniseringskode og forretningskode i ulike metoder. Se på følgende eksempel:

```
class C
begin
  op sync() == await someGuard .
  op business() == <forretningskode> .
with Any .
  op m() == sync@C(); business() .
end
```

Her er det i utgangspunktet ingen grunn til å ikke definere m() til å være "await someGuard; <forretningskode>". Men vi får muligheten til å gjøre en av disse to redefinisjonene:

```
class C' inherits C
begin with Any .
  op m() == business@C() .
end

class C' inherits C
begin
  op sync() == await someOtherGuard .
with Any .
  op m() == sync@C'() & business@C() .
end
```

Her ofrer vi noe innkapsling for å oppnå vilkårlig inkrementell endring. Programmereren av klassen C' er nå avhengig av å vite hvordan m() er implementert i C, nærmere bestemt at m() er implementert med to metoder og signaturen til metoden som implementerer forretningskoden. Arbeidet med å dele én metode opp i to metoder som inneholder synkroniseringskode og forretningskode er det samme som AOP gjør i fasen med å dekomponere et program til aspekter ved å identifisere kjernehensyn og tverrgående hensyn.

Forøvrig kan det nevnes at i likhet med Jeeg, så har Creol-prosjektet studert bruken av en lokal variabel som registrerer objektets kommunikasjonshistorie [3, 14]. Vi går ikke inn på denne mekanismen i denne oppgaven.

Kapittel 4

Maude og Creol-interpreten

Språket Creol er definert i språket Maude. I dette kapitlet presenterer jeg Maude, og gjør rede for Maude-implementasjonen av Creol-interpreten slik den var da arbeidet med denne oppgaven startet.

4.1 Maude

Maude [4, 5, 27] er et høynivå spesifikasjons- og modelleringsspråk. Språket er en implementasjon av omskrivingslogikk, og er derfor formelt fundert. I dette kapitlet presenteres Maude ved først å gi en kort beskrivelse av omskrivingslogikk. Deretter motiverer jeg bruken av formelle metoder til å modellere og spesifisere systemer med. Deretter ser jeg på hvilken klasse av programmeringsspråk som Maude hører hjemme i, i forhold til andre språkklasser som er mer kjente. Til slutt beskrives Maude sin syntaks, det vil si hvordan Maude implementerer omskrivingslogikken.

Core-Maude er navnet på den versjonen av Maude som brukes i denne oppgaven. Det finnes flere utvidelser av språket, for eksempel Full Maude, som tilbyr objektorientert modellering, og Real-Time Maude, som er designet for å lage modeller hvor tid spiller en viktig rolle.

4.1.1 Omskrivingslogikk som spesifikasjonsformalisme

Maude er en av flere eksisterende implementasjoner av omskrivingslogikk. En omskrivingsteori er et 4-tuppel $\mathcal{R} = (\Sigma, E, L, R)$ hvor signaturen Σ definerer en mengde av funksjonssymboler, E definerer ligninger mellom termer, L er en mengde merkelapper som brukes som regelnavn og R er en mengde av merkede omskrivingsregler.

Omskrivingsregler brukes på termer av gitte sorter. Sorter spesifiseres i likhetslogikk (Σ, E). Likhetslogikken utgjør et funksjonelt subspråk i omskrivingslogikk, og støtter algebraisk spesifikasjon. Når vi modellerer systemer i omskrivingslogikk, blir systemkomponentene typisk representert av termer av ulike sorter, definert i likhetslogikk. Den globale tilstandskonfigurasjonen er da definert som en multimengde av disse termene.

Omskrivingslogikk utvider algebraiske spesifikasjonsteknikker med omskrivingsregler. Omskrivingsregler supplerer ligningene som definerer termspråket, og brukes til å beskrive den dynamiske adferden til et system. En omskrivingsregel kan sees på som en lokal overgangsregel som tillater en instans av mønsteret t å forandre seg til å bli en instans av et mønster t'. Hvis det er nødvendig med hjelpefunksjoner for å uttrykke semantikken, defineres disse typisk i likhetslogikk. Hjelpefunksjoner blir evaluert mellom hver tilstandsovergang, det vil si at ligninger appliseres på termene før reglene. Hvis omskrivingsregler kan brukes på disjunkte deler av konfigurasjonen, kan tilstandsovergangene skje parallelt. Følgelig er samtidighet implisitt i omskrivingslogiske spesifikasjoner. Omskrivingsregler kan være betingede, hvor det er mulig å formulere betingelsen som en konjunksjon av omskrivinger og ligninger som må holde for at hovedregelen skal kunne brukes:

$$\textit{subkonfigurasjon} \longrightarrow \textit{subkonfigurasjon}\ \textbf{\textit{if}}\ \textit{betingelse}$$

4.1.2 Formelle metoder

Hvorfor brukes formelle metoder til å spesifisere og modellere systemer? I denne seksjonen motiveres bruken av Maude som en formell metode, basert på [24, 25].

Formelle metoder letter programverifikasjonen. Med "formelle metoder" menes språk, teknikker og redskaper som er basert på matematisk logikk. Når vi har en veldefinert, matematisk modell, kan vi bruke matematiske teknikker for å resonnere stringent omkring modellen. Vi skal se nærmere på hvorfor det er slik.

Det er to hovedgrunner til å utvikle en formell modell av et system: 1) Vi kan gi en presis, entydig og veldefinert beskrivelse av systemet. 2) Det kan være lettere å analysere en formell modell enn en uformell, spesielt med maskinell analyse. Noen fordeler med maskinell analyse framfor manuell analyse er:

- Det er vanskelig å bevise noe om en uformell modell. I teorien er det umulig å bevise noe om noe som ikke er formelt definert.
- Håndbeviser har ofte feil. Det er lett å overse uventede tilfeller i beviser og resonnementer. Når man beviser egenskaper ved en modell, er det lett å overse de samme tilfellene i beviset som de som er oversett i designet/spesifikasjonen av modellen.
- Maskinell analyse er ikke forutinntatt, og overser derfor ingenting. For eksempel vil Maude sin søkefunksjon analysere samtlige oppnåelige systemtilstander fra en gitt starttilstand med "brute force".

Veldefinert semantikk

I et språk som brukes som spesifikasjonsformalisme, må språkkonstruktene ha en veldefinert semantikk. Semantikken til imperative programmer er gitt av hvordan det imperative programmeringsspråket manipulerer datamaskinens minne. I motsetning til dette, vil semantikken til formelle, deklarative programmer være gitt av velkjente, matematiske konstrukter, som for eksempel algebra, kategori eller funksjon. Dette tillater bruk av kjente, matematiske teknikker for å analysere og verifisere modellen. Det åpner også for bruk av forskjellige redskaper for automatisk og maskinell analyse. Uformelle og semiformelle spesifikasjonsformalismer som for eksempel UML er ikke veldefinerte i betydningen at de har en matematisk semantikk.

Den klare semantikken til en formell spesifikasjon har flere fordeler:

- En formell spesifikasjon er nødvendigvis entydig, i betydningen at alle symbolene i spesifikasjonen har en veldefinert semantikk. Utviklingen av et system starter alltid med en uformell spesifikasjon, enten den er formulert i et naturlig språk, med hjelp av diagrammer eller med andre notasjoner. Formalisering gjør at alle uklarheter og tvetydigheter lukes ut fra den uformelle spesifikasjonen, slik at antallet misforståelser reduseres.
- Alle antagelser og forutsetninger blir gjort eksplisitte. Hvis den uformelle spesifikasjonen mangler nødvendige antagelser, blir det tydeliggjort. Hvis den inneholder implisitte antagelser, unngår man problemer som følge av at de blir oversett.
- Det kan være mulig å avgjøre nøyaktig hvilke tilstander og oppførsler som er lovlige og ulovlige ifølge spesifikasjonen, uten at det nødvendigvis kan gjøres maskinelt. Det kan være mulig å avgjøre uten tvil hvorvidt systemspesifikasjonen er oppfylt av implementasjonen av systemet. Det er også mulig å lage en formell kravspesifikasjon og bevise at systemspesifikasjonen oppfyller kravspesifikasjonen. Dette kan for eksempel gjøres i Maude med temporallogiske formler og den innebygde modellsjekkeren.

Abstraksjon

Formelle metoder gjør det lett å lage abstrakte modeller. Generelt kan vi definere en modell som en representasjon av virkeligheten hvor det irrelevante er fjernet. En god modell er så abstrakt som mulig, det vil si inneholder så lite informasjon som mulig, uten å miste relevant informasjon. Fordelene med abstraksjon er:

- Abstraksjon hjelper oss med å beholde sakligheten i vårt eget arbeid. Vi slipper å bruke ressurser på unødvendig arbeid. For eksempel kan det hende at vi ønsker for eksempel å modellere samtidige systemer, men ønsker å slippe å forholde oss eksplisitt til hvordan mekanismene for samtidighet virker.
- Det er lettere å lese og forstå programmene/spesifikasjonene for mennesker. Et høynivåspråk er potensielt mer naturlig og intuitivt enn et lavnivåspråk.
- Det blir færre misforståelser. En formalisme som både har en klar semantikk og er intuitiv og lett forståelig, letter samarbeidet mellom mennesker fra forskjellige faglige og språklige/nasjonale miljøer.

Prototyping, maskinell analyse og beviser

I en systemutviklingsprosess er det om å gjøre å oppdage feil så tidlig som mulig. Jo tidligere en feil oppdages, jo mindre blir kostnadene med å rette den. En iterativ systemutviklingsprosess er ofte bra for å finne feil i systemet. Prototyping hører naturlig hjemme i en iterativ prosess: En prototyp brukes til å prøvekjøre eller simulere systemet, slik at feil kan oppdages på et tidlig stadium. Deretter lokaliseres årsaken til feilene i spesifikasjonen og spesifikasjonen endres. Så utvikler man en ny prototyp i samsvar med den nye spesifikasjonen.

Maude legger til rette for iterativ utvikling av spesifikasjoner ved prototyping, og tilbyr flere måter å analysere prototypen på maskinelt. Maude-programmer har både en veldefinert semantikk og kan gjøres svært abstrakte. I tillegg er Maude-programmer eksekverbare. Vi har altså eksekverbare spesifikasjoner. I det øyeblikk du har laget en systemspesifikasjon har du også laget en modell og en prototyp, som kan analyseres maskinelt. En forandring i spesifikasjonen reflekteres umiddelbart i prototypen og omvendt. I Maude er programmering, systemspesifisering og modellering være det samme. Kjøring av Maude-programmet er det samme som å simulere systemet. Dette er bra av følgende grunner:

- Når du først har spesifikasjonen, er det er ingen ekstra kostnad med å implementere prototypen, eller å oversette spesifikasjonen til en modell i en annen notasjon.
- Mennesker er en uuttømmelig kilde til feil i utvikling av programvare. Hvis mennesker skal være involvert i prosessen med å lage en modell på bakgrunn av en spesifikasjon, risikerer vi å modellere noe annet enn det vi har spesifisert.
- En Maude-spesifikasjon kan beskrive en programstruktur. Når en Maude-modell skal implementeres, kan modellen sees på som en beskrivelse av strukturen på implementasjonen. Vi kan se på en del av Maude-spesifikasjonen og implementere den, før vi ser på en annen del av spesifikasjonen og implementerer den. vi kan også velge å se på Maude-spesifikasjonen som en høynivåimplementasjon, som forfines og konkretiseres til å bli den endelige implementasjonen.

Simulering er ikke nok til å garantere feilfrihet. En vellykket simulering fra en starttilstand gir bare en garanti for den gitte starttilstanden og én eksekveringshistorie. En garanti for at et program er feilfritt krever en garanti for alle starttilstander og alle oppførsler. Det er dette som er et bevis for at programmet er korrekt.

Når det gjelder metodologi for spesifikasjon og verifikasjon av systemer er det stor forskjell i kostbarhet og styrke mellom uformell spesifikasjon og bevis. Her er noen analysemetoder som kan rangeres etter økende styrke og kostbarhet:

1. Formell spesifikasjon.
2. Prototyping og simulering.
3. Analyse av alle mulige systemtilstander fra én starttilstand. I Maude kan det gjøres med den innebygde søkekommandoen eller med den innebygde modellsjekkeren for temporallogiske formler.
4. Analyse fra mange initialtilstander med narrowing.
5. Bevis: Analyse av alle mulig systemtilstander fra alle mulige initialtilstander med teorembeviser og induktive teknikker.

Nok en analysemetode er statistisk sampling av probabilistiske systemer, hvor et spesifisert krav kan være et gitt resultat i 80% av simuleringene. Forøvrig gir Maude muligheten til å definere egne analysestrategier ved bruk av Maude sine metaprogrammeringsfasiliteter.

Det er veldig kostbart å bevise at et større program er korrekt. Derfor gjelder det å styrke spesifikasjonen med analyse før man begynner på arbeidet med beviser. Maude fyller en del av dette metodologiske intervallet.

4.1.3 Språkklasser

Språket Creol er imperativt. Maude tilhører en språkklasse som er en mellomting mellom imperative og deklarative språk, kalt single-assignment-språk. Maude har et funksjonelt subspråk. Hva betyr dette?

Vi har flere klasser av programmeringsspråk, og tre av dem er

1. deklarative språk, som for eksempel funksjonelle eller logiske språk,
2. imperative, som for eksempel prosedyreorienterte og objektorienterte språk og
3. single-assignment-språk.

Et program som er skrevet i et deklarativt språk beskriver eksplisitt hvilke egenskaper det ønskede resultatet skal ha, men sier ingenting om hvordan resultatet skal oppnås. Enhver implementasjon av måten å komme fram til resultatet på er akseptabel, så lenge resultatet har de spesifiserte egenskapene.

I motsetning til dette, vil et program som er skrevet i et imperativt språk beskrive eksplisitt hvordan resultatet skal oppnås, og det sier bare implisitt hvilke egenskaper resultatet skal ha. Disse egenskapene er gitt av egenskapene til de resultatene som følger av å bruke den beskrevne framgangsmåten. Prosedyren for å produsere det ønskede resultatet tar form av en sekvens av operasjoner.

Imperative språk kjennetegnes ved at de har tilordningssetninger. Tilordninger er destruktive, på den måten at en tilordnet verdi erstatter den verdien som variabelen har fra før. Dette er uproblematisk fordi imperative språk utfører setninger i ordnet rekkefølge. Dette gir opphav til konseptet flytkontroll. Imperative språk er nært knyttet til Von Neuman-modellen for datamaskinarkitektur.

Deklarative språk beskjeftiger seg med statiske konsepter istedenfor dynamiske (hva istedenfor hvordan). Derfor avhenger de ikke av konsepter som rekkefølge, flytkontroll eller tilordninger. Et program som er skrevet i et deklarativt språk vil bestå av de ligningene som er tilstrekkelige for å karakterisere de ønskede resultatene. Deklarative språk er ikke konseptuelt knyttet til Von Neuman-modellen. De åpner derfor for muligheten for å bruke nye maskinarkitekturer som støtter parallellitet i større grad, for eksempel dataflytmaskiner (eng: dataflow machine).

Språkklassene funksjonelle og logiske språk er subtyper av klassen deklarative språk. Funksjonelle språk beregner verdiene av funksjoner, og er baserte på lambdakalkyle og rekursive ligninger. Et program som er skrevet i et funksjonelt språk vil typisk bestå av en uordnet mengde ligninger som definerer funksjonene og verdiene.

Single-assignment-språk ligner på imperative språk ved at de har tilordningssetningen og typiske kontrollflytkonstrukter som for eksempel if-setninger og løkker. De skiller seg imidlertid fra imperative språk ved at en variabel bare kan tilordnes én gang per gjennomløp av en ytterste løkke av programeksekveringen. Det gjøres unntak for tilordningssetninger i while-løkker. Denne begrensningen endrer tilordningssetningens natur, slik at den blir en statisk assosiasjon mellom et variabelnavn og en verdi, istedenfor en dynamisk, destruktiv operasjon. Denne statiske egenskapen gjør at vi ikke trenger den ordnede eksekveringsrekkefølgen som imperative språk har, det vil si at de ikke bruker semikolon mellom setningene som de fleste imperative språk gjør. Tilordningssetninger i single-assignment-språk kan eksekveres så snart uttrykket i setningens høyre side kan evalueres. Single-assignment-språk er også egnet for dataflytmaskiner.

Maude passer inn i kategorien single-assignment-språk. Maude er en implementasjon av omskrivingslogikk. I omskrivingslogikk har vi overgangsregler, som beskriver endringer i en systemtilstand. Reglene anvendes ikke i en ordnet rekkefølge. Følgelig bruker ikke Maude tegnet ";" for å ordne rekkefølgen på tilstandsovergangene, i motsetning til imperative programmeringsspråk. To helt sentrale egenskaper ved omskrivingslogikk som spesifiseringsnotasjon er at 1) regler kan anvendes i vilkårlig rekkefølge og 2) så lenge de brukes på disjunkte deler av systemkonfigurasjonen, kan de brukes samtidig med hverandre. Dermed har vi implisitt modellert ikke-determinisme og samtidighet så lenge vi ikke eksplisitt angir noe annet.

4.1.4 Maude sin Syntaks

Moduler

Et Maude-program/spesifikasjon er satt sammen av moduler, ved at en modul importer andre moduler, som igjen kan importere andre moduler. I Core Maude har vi to typer moduler, funksjonelle og systemmoduler. Disse deklareres slik:

fmod $MODULNAVN1$ **is**	**mod** $MODULNAVN2$ **is**
$\langle modulkropp \rangle$	$\langle modulkropp \rangle$
endm	**endm**

En modulkropp i en funksjonell modul består av setninger for import av moduler, deklarasjoner av sorter, deklarasjoner av funksjonssymboler, deklarasjoner av variabler og definisjoner av ligninger. En modulkropp i en systemmodul kan inneholde det samme som en funksjonell modul, og i tillegg omskrivingsregler.

Maude har et standard bibliotek i filen "prelude.maude", som består av en mengde moduler som definerer grunnleggende datatyper, som for eksempel heltall, flyttall tekststrenger og boolske verdier. Denne filen lastes inn ved hver oppstart av maude-interpreten, slik at programmereren kan få tilgang til datatypene ved å importere de relevante modulene fra "prelude.maude" til sine nye moduler.

Moduler som skal importeres fra andre filer, må lastes eksplisitt med en setning som plasseres utenfor moduldeklarasjonen. Hvis modulen MODULNAVN1 ligger i filen "fil1.maude" kan den importeres med nøkkelordet "including" slik:

in $fil1.maude$
mod $MODULNAVN2$ **is**
including $MODULNAVN1$.
$\langle resten\ av\ modulkroppen \rangle$
endm

Kommentarer i koden angis med $***$ for én linje eller $***(\langle kommentarer \rangle)$ for flere linjer.

Sort- og variabeldeklarasjoner

Sorter brukes til å kategorisere grunntermer, som representerer verdier. Vi deklarerer også subsortrelasjoner, som ligner på subtyping. Et eksempel:

sort $Element.$ **sorts** $List\ Set.$
subsorts $Nat < Element < List\ Set.$

Vi ønsker å definere datatypene liste og mengde. I de to første setningene deklarerer vi tre sorter som vi skal bruke til dette. I andre linje deklareres *Element* til å være subsort av *List* og *Set*. Betydningen av dette er at et element er en liste og en mengde. Med andre ord, en liste/mengde kan bestå av bare ett element. Sorten Nat er den predefinerte sorten for naturlige tall. Vi har nå lagt opp til å definere lister og mengder av naturlige tall.

Variabler deklareres ved å angi variabelnavn og sort:

var $L : List.$ **var** $S : Set.$ **vars** $E\ E' : Element.$

Operatordeklarasjoner

Operatorer kan sees på som navn på funksjoner. De kan defineres til å brukes infix eller prefix. Vi ønsker å definere en operasjon som legger et element til en liste. Her er tre ulike måter å deklarere dette på, hvor tegnet _ angir hvor argumentene skal plasseres.

op $append : List\ Element -> List.$
op $add\_to\_ : Element\ List -> List.$
op $\_;\_ : List\ List -> List.$

Operatorene sammen med variablene danner termer som $append(L, E)$, *add E to L* og $L\ ;\ E$.

Operatoredeklarasjoner kan ha attributter, gitt ved følgende nøkkelord. **ctor** angir konstruktørtermer. Slike termer har ikke variabler og er byggesteiner for eller verdier i mer komplekse termer. Hvis vi antar at ligningsmengden i en funksjonell modul (se nedenfor) er terminerende og konfluent, vil hver grunnterm (term uten variabler) bli forenklet til en konstruktørterm. **assoc** angir at operatoren er assosiativ. **comm** angir at operatoren er kommutativ. **id** : $\langle term \rangle$ angir et identitetselement. **format**($\langle formateringsvalg \rangle$) brukes for å formatere utskrift til skjerm. **prec** angir operatorens presedens, som avgjør hvordan termer skal parseres når flere operatorer brukes. For eksempel har $*$ sterkere presedens enn $+$ i uttrykket $x+y*z$, som gjør at uttrykket tolkes som $x + (y * z)$ og ikke som $(x + y) * z$. **ditto** brukes ved overlasting av operatorer som en forkortelse for de samme attributtene som ble brukt ved forrige deklarasjon av en operator med samme navn.

Et eksempel på bruk av operatorattributter: Vi ønsker å definere sammensettingsoperatorer for lister og mengder. Først definerer vi to konstanter som representerer tomme lister og mengder. Konstanter er funksjoner uten argumenter. En liste er per definisjon assosiativ, og en mengde er per definisjon både assosiativ og kommutativ. Dermed kan sammensettingsoperatorene deklareres slik:

op *none* : $->$ *List* [*ctor*].
op *empty* : $->$ *Set* [*ctor*].
op _ ; _ : *List List* $->$ *List* [**ctor assoc id** : *none*].
op _ _ : *Set Set* $->$ *Set* [**ctor assoc comm id** : *empty*].

Funksjonelle moduler

Det er vanlig at funksjonelle moduler spesifiserer datatyper. En datatype er en mengde verdier og operasjoner på disse. Operatordeklarasjoner spesifiserer verdier og syntaksen til operatorene. Ligninger spesifiserer semantikken til operatorene.

En ligning angir to grunntermer som kan inneholde variabler. Hvis ligningens venstreside matcher en term i konfigurasjonen, reduseres termen til å matche mønstret i ligningens høyreside. Både ligninger og regler kan være betingede, i form av en if-setning på slutten av regelen/ligningen. I tillegg kan en regel/ligning sin høyreside inneholde en tradisjonell if-then-else-setning, slik at vi kan få to nivåer av betingelser i en regel/ligning:

eq $\langle mønster \rangle$ $= \langle mønster' \rangle$.
eq $\langle mønster \rangle$ $=$ **if** $\langle betingelse \rangle$ **then** $\langle mønster' \rangle$ **else** $\langle mønster'' \rangle$ **fi**.
ceq $\langle mønster \rangle$ $= \langle mønster' \rangle$ **if** $\langle betingelse \rangle$.
ceq $\langle mønster \rangle$ $=$ **if** $\langle betingelse \rangle$ **then** $\langle mønster' \rangle$ **else** $\langle mønster'' \rangle$ **fi**
if $\langle betingelse' \rangle$.

Ligninger brukes for å forenkle uttrykk og brukes inntil ingen ligning matcher noen del av konfigurasjonen. Da er termene på normalform. Kommandoen **red** tar en term og reduserer den til sin normalform. Det antas at likhetsspesifikasjonen er terminerende og konfluent (Church-Rosser). Det vil si at det er programmererens ansvar å sørge for at ligningene reduserer termer til stadig enklere termer inntil de ikke kan forenkles mer og slik at reduksjon av et uttrykk resulterer i det samme resultatet uansett hvilken rekkefølge ligningene anvendes i.

Et par eksempler: Vi definerer semantikken til append-operatoren, som virker på lister. Slik vi har definert sorten Set hittil, er den en multimengde. For å definere den som en mengde, må vi fjerne like forekomster med en ligning. Merk at rekkefølgen på variablene i termen ikke spiller noen rolle på grunn av kommutativitetsegenskapen.

vars $E\ E'$: *Element*. **var** S : *Set*.
eq $append(L, E)$ $= L; E$.
ceq $E\ E'\ S$ $= E\ S$ **if** $E\ ==\ E'$.

Den siste ligningen kan også se slik ut: **eq** $E\ E\ S = E\ S$. Dette er et eksempel på at vi ofte kan velge mellom å bruke betingede ligninger og mønstergjenkjenning for å avgjøre om en ligning skal appliseres på en term.

Maude har en innebygget operator :: for sammenligning av sorter. Den brukes til å lage et utsagn om at en term av en angitt sort. For eksempel, med de variablene vi har deklarert, vil uttrykket `L :: List` evaluere til den boolske verdien sann. Dette kan brukes som betingelse i en ligning på følgende måte:

sorts *Good Bad.* **subsorts** *Good Bad* < *Element.*
ceq *append*(L, E) = $L; E$ **if** $E ::$ *Good.*

Betingelsen sikrer at ligningen ikke brukes til å utvide listen L med en term som av en eller annen grunn er av en uønsket sort. Det er variabelen E sin *minste sort* som blir sjekket. Anta at E er bundet til en term av sort Good. Da er E av både sorten Element og av sorten Good siden en Good er en Element. Siden Good er "nederst" eller "lengst til venstre" i subsortdeklarasjonen så er termens minste sort Good.

Systemmoduler

Mens funksjonelle moduler vanligvis beskriver statiske fenomener, beskriver systemmoduler dynamisk oppførsel i det modellerte systemet. En regel beskriver en mapping fra en tilstand til en annen, og er ikke reversibel, det vil si at relasjonen ikke er symmetrisk. Et eksempel:

rl [*birthday*] : *age*(N) => *age*$(N + 1)$.

hvor *age* er en egendefinert operator og N en variabel av den innebygde sorten Nat for naturlige tall. Regler kan være betingede og bruke if-tester på samme måte som ligninger.

Hvorfor har vi regler i tillegg til ligninger? Det er ikke noe krav om at regelmengden skal gi et terminerende eller konfluent system. Det er godt mulig å modellere flere tilstandsoverganger fra én systemtilstand. I tillegg kan flere regler brukes samtidig på ulike, disjunkte delkonfigurasjoner. Dette gjør det lett å modellere samtidighet og ikke-determinisme i systemet, noe som i høyeste grad er relevant for distribuerte systemer.

4.1.5 Simulering og maskinell analyse med Maude

Kjøring av et Maude-program er simulering av det modellerte systemet. Simuleringen starter fra en konkret starttilstand. Alle termer i konfigurasjonen reduseres til sin normalform mellom hver regelapplikasjon. Hvis flere regler kan anvendes på ikke-overlappende deler av konfigurasjonen, vil de bli brukt samtidig. Vi sier at spesifikasjonen *omskrives* hvis regler anvendes på den. Vi sier at spesifikasjonen *reduseres* hvis bare ligninger anvendes på den. Den fasen hvor en regel brukes eller hvor flere regler brukes samtidig, kalles et *omskrivingssteg*. Hvis konfigurasjonen verken kan reduseres eller omskrives, har vi nådd en *sluttilstand*.

Det er flere måter å eksekvere en Maude-spesifikasjon på. Nå presenteres de kommandoene jeg har brukt i arbeidet med denne masteroppgaven.

- **red** ⟨*term*⟩ . reduserer en term så langt det er mulig ved å bruke ligninger og uten å bruke regler.
- **rew** ⟨*term*⟩ . gir en sluttilstand som er oppnåelig fra initialtilstanden ⟨*term*⟩, forutsatt at spesifikasjonen er terminerende. I en modell av et system tilsvarer dette simulering av en mulig kjøring/eksekveringshistorie. I teorien brukes omskrivingsreglene i vilkårlig rekkefølge, i praksis har Maude-interpreten en deterministisk og usofistikert omskrivingsstrategi.
- **rew** [n] ⟨*term*⟩ . vil gi den tilstanden som er resultatet etter maksimalt n omskrivingssteg. En måte å observere systemets adferd på er å gi denne kommandoen flere ganger og inkrementere antall omskrivingssteg for hver gang. Dette kan være nyttig ved ikke-terminerende spesifikasjoner. Et eksempel er feilsøking for å finne ut hvorfor en spesifikasjon som burde være terminerende ikke er det.
- **frew** ⟨*term*⟩ . er lik **rew**, med unntak av at eksekveringsstrategien er slik at alle reglene forsøkes brukt like mange ganger. Dette vil vanligvis gi en annen omskrivingssekvens enn ved bruk av **rew**. **frew**-kommandoen kan i likhet med **rew**ta et tall som angir antall omskrivingssteg.
- **search** ⟨*term*⟩ ⟨*pil*⟩ ⟨*mønster*⟩ . søker med en bredde-først-strategi etter termer som er oppnåelige fra initialtilstanden ⟨*term*⟩ og som matcher mønstret ⟨*mønster*⟩. Mønstret er en grunnterm som kan inneholde variabler. ⟨*pil*⟩ er enten

  => 1 søk etter alle termer som er oppnåelige med ett omskrivingssteg fra initialtilstanden,

=> * søk etter alle termer som er oppnåelige etter null eller flere omskrivingssteg,

=> + søk etter alle termer som er oppnåelige i ett eller flere omskrivingssteg eller

=>! søk etter alle oppnåelige termer som ikke kan omskrives ytterligere. I en modell av et system vil dette være et søk etter alle terminerende tilstander.

- **search** ⟨*term*⟩ ⟨*pil*⟩ ⟨*mønster*⟩ **such that** ⟨*betingelse*⟩ . er som ovenfor, men hvor vi søker etter alle termer som både matcher et mønster og oppfyller en betingelse. Et eksempel:
  search $age(0)$ => $*$ $age(N)$ **such that** $N > 67$.
- **search** [*n*] ⟨*term*⟩ ⟨*pil*⟩ ⟨*mønster*⟩ . søker etter maksimalt n løsninger. Kan brukes med such that-betingelse.
- **show path** n . brukes umiddelbart etter en søkekommando og viser omskrivingssekvensen som ledet fram til resultat nummer n (en term) i søket. Denne kommandoen er nyttig for å finne ut hvordan en uønsket tilstand kan oppstå.

4.2 Interpreten

Interpreten som jeg skal beskrive i denne seksjonen er et av de foreløpige resultatene av Creol-prosjektet [13–15, 26]. Mye av den første versjonen av interpreten ble utviklet av Marte Arnestad [2].

Creol-interpreten er en formell spesifikasjon av Creol sin operasjonelle semantikk. Den er spesifisert i Maude, hvilket gjør spesifikasjonen eksekverbar. Vi har altså en modell av Creol som tillater oss å simulere prosessering av Creol-programmer.

Interpreten inkluderer en spesifikasjon av språket CMC (Creol machine code), som er det språket som interpreten faktisk tolker. Det har en syntaks som ligner på Creol sin. CMC består i likhet med språket Creol av et antall klassedefinisjoner. Skillet mellom disse språkene gjør at vi er frie til å utvide CMC etter hvert som vi møter tekniske utfordringer med spesifikasjonen av Creol sitt kjøretidsmiljø. Vi kan lage operatorer og setningstyper i CMC som lar oss løse de tekniske/praktiske problemene som måtte oppstå.

Vi antar at Creol-kode kan oversettes til CMC maskinelt. Det har blitt arbeidet med en kompilator for Creol, men i denne oppgaven har all CMC-kode blitt skrevet for hånd.

I denne seksjonen beskriver jeg de delene av interpreten som er relevante for implementasjonen av synkronisert fletting. Nærmere bestemt er det modellen av konfigurasjonen, vakter og metodekall, samt operatorene for fletting og ikke-deterministisk valg av setningslister. Den komplette spesifikasjonen finnes i vedlegg A.

Reglene og ligningene kan inneholde lange termer med mange variabler. Ofte er det bare en mindre subterm av en større term som endres. For å lette lesbarheten, utelukker jeg irrelevante attributter i objektene når jeg beskriver ligninger og regler.

For å unngå forvirring må jeg avklare hva jeg mener med følgende begreper:

metodedefinisjon En metodedefinisjon er inneholdt i en klassedefinisjon, og består av en liste av lokale variabler som inkluderer metodens formelle parametre og en setningssekvens.

metodeinstans Når en metode kalles, lages det en kopi av metodedefinisjonens setningsliste, og metodens variabler instansieres med reelle parametre og eventuelle forhåndsbestemte verdier. Det er denne setningssekvensen med variabler som manipuleres av interpreten når metoden kjøres.

prosess En intuitiv, generell forklaring på hva en prosess vil være at en prosess er et program som kjører, i motsetning til en programtekst, som ikke er noe annet enn nettopp en tekst. Mer presist kan vi si at en prosess består av programkode og en tilstand. I CMC er enhver prosess en instansiert metodedefinisjon. Slik som vi snart skal definere metodeinstans, er globale variabler ekskludert. Dette skiller seg fra den vanlige forståelsen av ordet prosess. Jeg vil forsøke å bruke ordet metodeinstans som definert ovenfor når globale variabler er ekskludert.

sorten Process Interpreten spesifiserer en sort *Process*:

op $\_,\_ : ProgList\ Subst -> Process$ [**ctor**].

En Process er et tuppel $\langle setningssekvens, variabelbindinger \rangle$, representert henholdsvis av en term av sort *ProgList* og en term av sort *Subst*. En Process brukes til å oppbevare en metodeinstans i arbeidet med binding og skedulering.

metode Med ordet metode mener jeg enten metodedefinisjon eller prosess. Jeg håper det kommer klart fram av sammenhengen hva som menes.

4.2.1 Modellen av systemkonfigurasjoner

Et CMC-program representerer et distribuert system. Når interpreten tolker et CMC-program, har vi selvfølgelig en systemtilstand. Denne systemtilstanden modelleres som en term av sort *Configuration*. Den er en multimengde av fire ulike typer objekter: Klassedefinisjoner, klasseinstanser, meldinger og meldingskøer, representert med termer av hver sin sort. Lokasjoner og kommunikasjonskanaler i det distribuerte systemet modelleres ikke. Vi skal se på hvordan hver av de fire systemkomponentene er bygd opp.

sort $Configuration$.
subsorts $Object\ MMsg\ Queue\ Class < Configuration$.
op $none : -> Configuration[ctor]$.
op $\_\_ : Configuration\ Configuration -> Configuration$ [**ctor assoc comm id** : $none$].

Sorten MMsg betegner en multimengde av meldinger.

4.2.2 Klasser

Termer av sort *Class* konstrueres med følgende operator:

sort $Class$.
op $< \_ : Cl \mid Vs : \_,\ Inh : \_,\ Att : \_, Mtds : \_, Ocnt : \_ > :$
$Cid\ Nat\ InhList\ InitSubst\ MMtd\ Nat -> Class$ [**ctor**].

Her er en kort beskrivelse av parametrene:

Cl Klasseidentifikatoren.

Vs Versjonsnummer som viser hvor mange ganger klassen har blitt endret etter at den ble en del av konfigurasjonen, med andre ord hvor mange ganger klassen har blitt dynamisk oppdatert.

Inh Liste over klasser som denne klassen arver fra, inkludert disse klassenes versjonsnummer og klasseparametre.

Att Objektattributter. Liste av variabler som vil være tilgjengelige for alle prosessene til et objekt av denne klassen.

Mtds Multimengde av metodedefinisjoner.

Ocnt Teller som viser hvor mange ganger klassen har blitt instansiert.

4.2.3 Objekter

Termer av sort *Object* konstrueres med følgende operator:

sort $Object$.
op $< \_ : Ob \mid Cl : \_,\ Pr : \_,\ PrQ : \_,\ Lvar : \_,\ Att : \_,\ Lcnt : \_ > :$
$Oid\ ClVs\ ProgList\ MProg\ Subst\ Subst\ Nat -> Object$ [**ctor**].

Ob Objektnavn.

Cl Klasseidentifikator som beskriver hvilken klasse objektet er en instans av, inkludert klassens versjonsnummer

Pr Når en prosess skeduleres, legges programkoden i Pr. Denne listen av setninger er en delmengde av en metodekropp.

PrQ Kø av ventende prosesser. PrQ er en multimengde og ikke en liste, hvilket innebærer at vi ikke har noen spesifisert køordning.

Lvar Liste av lokale variabler. Dette er variablene til den prosessen som kjører for øyeblikket. De er ikke tilgjengelige for andre prosesser.

Att Liste av globale variabler. Dette er objektattributtene til klasseinstansen. Disse variablene er tilgjengelige for alle objektets prosesser.

Lcnt Teller som brukes til å gi prosesser en identifikator. Tallet i Lcnt kaller jeg en merkelappverdi, og spiller en viktig rolle i håndteringen av metodekall. Hver gang en metode kalles, kopieres verdiene fra Ob og Lcnt til metodens obligatoriske lokale variabler caller og label. Etterpå inkrementeres Lcnt. Slik får hvert kall en globalt unik identifikator. Merkelappverdien brukes til å koordinere en svarmelding med prosessen som venter på denne meldingen/metodereturen.

Når vi generelt snakker om skeduleringsmekanismen i et kjøretidssystem, mener vi ofte alle operasjoner som er involvert i å starte og stoppe kjøringen av prosesser, inkludert håndtering av køer. Vi kan skille mellom skedulering og dispatching. Skedulering handler da om å bestemme rekefølgen på prosessene som skal kjøres, ved å røkte en kø av prosesser i henhold til en viss køordning. Dispatching handler om den mer begrensede, tekniske oppgaven å faktisk starte kjøringen av den prosessen som ligger først i skeduleringskøen. Når jeg snakker om å skedulere en prosess i CMC, mener jeg å finne metodeinstansen i PrQ, plassere instansens setningsliste i Pr og variablene dens i Lvar.

4.2.4 Metoder

Sorten *Mtd* betegner metodedefinisjoner. En metodedefinisjon er inneholdt i en klassedefinisjon, og har formen

sort *Mtd.*
op $< \_ : Mtdname \mid Latt : \_,\ Code : \_ > :\ Qid\ Subst\ ProgList -> Mtd$ [**ctor**].

Mtdname Metodenavn.

Latt Liste av variabler som er lokale for metoden. Listen inkluderer metodens formelle parametre. Alle metoder har to "read-only" variabler som heter "caller" og "label", som initieres med henholdsvis objektnavnet til det kallende objektet og en merkelappverdi. Til sammen identifiserer caller og label metodens prosess unikt i konfigurasjonen.

Code Liste av programsetninger.

4.2.5 Setninger og setningslister

De ulike setningene i CMC modelleres ved å deklarere operatorer som gir konstruktørtermer av sorten *Stm* eller *Prog*. Sorten *Stm* betegner atomiske setninger og *Prog* sammensatte setninger. Her følger kommenterte deklarasjoner av operatorene.

(1) **op** _ := _ : *AidList List* −> *Stm* [**ctor**].
(2) **op** _ := *new*_(_) : *Aid Cid List* −> *Stm* [**ctor**].
(3) **op** *if*_*th*_*el*_*fi* : *Expr ProgList ProgList* −> *Stm* [**ctor**].
(4) **op** *if*_*th*_*fi* : *Expr ProgList* −> *Stm*.
(5) **op** *while*_*do*_*od* : *Expr ProgList* −> *Stm* [**ctor**].
(6) **op** _(_; _) : *Mid List List* −> *Stm* [**ctor**].
(7) **op** !_(_) : *Mid List* −> *Stm* [**ctor**].
(8) **op** _!_(_) : *Qid Mid List* −> *Stm* [**ctor**].
(9) **op** _?(_) : *Qid List* −> *Stm* [**ctor**].
(10) **op** *end* : *List* −> *Stm* [**ctor**].
(11) **op** *continue* : *Nat* −> *Stm* [**ctor**].
(12) **op** *await*_ : *Guard* −> *Prog* [**ctor**].
(13) **op** *await*_ : *ExtGuard* −> *Prog*.
(14) **op** _[]_ : *ProgList ProgList* −> *Prog* [**ctor assoc**].
(15) **op** _|||_ : *ProgList ProgList* −> *Prog* [**ctor assoc**].

(1) Tilordning. En liste av variabler (første parameter) tilordnes en liste av verdier (andre parameter) i én tilordningssetning.

(2) Opprettelse av et nytt objekt. Parametre er en objektpeker, klasseidentifikator og reelle klasseparametre.

(3) If-then-else. Parametre: Et (boolsk) uttrykk og to setningssekvenser.

(4) If-then. Parametre: Et uttrykk og en setningssekvens.

(5) While. Parametre: Et uttrykk og en setningssekvens.

(6) Synkront kall. Parametre: En metodeidentifikator, en liste av innparametre og en liste av utparametre.

(7) Asynkront kall uten merkelapp, det vil si at svarmelding ikke forventes. Parametre: En metodeidentifikator og en liste av innparametre.

(8) Asynkront kall med merkelapp. Svarmelding mottas på et senere tidspunkt i koden. Parametre: Metodeidentifikator, innparametre, utparametre.

(9) Svarsetning. Asynkront, blokkerende mottak av en svarmelding. Parametre: Merkelappverdien til kallet som vi venter på svar fra og en liste av utparametre. Utparametrene tilordnes hvis svarmeldingen foreligger.

(10) Metoderetur. Brukes av den kalte metoden for å sende svarmeldingen sin. Parameteren er en liste av uttrykk som evalueres til returverdier. Slik interpreten er spesifisert, trenger ikke end-setningen å være den siste setningen i metoden. Det vil si at en metode kan avgi returverdiene uten å terminere, men bare én svarmelding vil bli akseptert av objektet som kalte denne metoden.

(11) Siste setning i en metodeinstans som har blitt kalt internt og synkront. Setningen tvinger fram skedulering av den kallende prosessen. Parameteren er en merkelappverdi som tilsvarer verdien i den kallende prosessens svarsetning.

(12) Prosessorslippunkt. Parameteren er en mengde av vakter, som alle må være sanne for at synkroniseringsbetingelsen skal være oppfylt.

(13) Prosessorslippunkt. Parameteren er en utvidelse av mengden av vakter med en type vakt som er syntaktisk sukker for å både sjekke om en metoderetur foreligger og å tilordne utparametrene i kallet. Evaluering av et uttrykk har *sideeffekt* hvis evalueringen av uttrykket resulterer i en endring av prosessens tilstand. I vårt tilfelle består sideeffekten av at utparametrene i en slik metodereturvakt blir tilordnet når vakten evaluerer til sann.

(14) Ikke-deterministisk valg. Parametrene er to setningslister.

(15) Fletting. Parametrene er to setningslister,

Setninger settes sammen til setningslister med symbolet semikolon:

op *empty* : $->$ *ProgList* [**ctor**].
op _ ; _ : *ProgList ProgList* $->$ *ProgList* [**ctor assoc id** : *empty*].

CMC har et par setninger som Creol ikke har. end-setningen ligner return-setningen i Java, og inneholder en liste av formelle parametre som utgjør utvariablene i kallet. Når end-setningen kjøres, evalueres parameterlisten og verdiene legges i en comp-melding. Comp-meldingen sendes fra det kalte objektet til det kallende objektet og inneholder metodereturen. I en kompilering fra Creol til CMC vil end-setningen bli satt inn automatisk.

Continue-setningen brukes i CMC til å tvinge fram skeduleringen av en bestemt prosess. Dette brukes spesielt i forbindelse med interne, synkrone metodekall. For at den kalte metoden skal få tilgang til prosessoren, må den kallende prosessen suspenderes. Den kalte prosessen blir utvidet med en continue-setning slik at den kallende prosessen blir skedulert umiddelbart etter at den kalte prosessen terminerer.

4.2.6 Variabler og datatyper

En variabel(binding) er et tuppel $\langle variabelnavn, verdi \rangle$, representert henholdsvis av termer av sort *Aid* og *Data*. Sorten *Subst* betegner en liste av variabelbindinger/-substitusjoner.

sorts *BndVar Subst InitSubst.* **subsort** *Qid* $<$ *Aid.*
subsorts *BndVar* $<$ *Subst* $<$ *InitSubst.*

op _ : _ : *Aid Data* $->$ *BndVar* [**ctor format**(! *o o o*)].
op *no* : $->$ *Subst* [**ctor**].
op _ , _ : *Subst Subst* $->$ *Subst* [**ctor assoc id** : *no*].

Alle navn i Creol blir til termer av sort Qid i CMC. Sorten Qid (Quoted identifier) er en innebygget sort for identifikatorer i Maude. Ved å bruke Qid istedenfor Maude-variabler får vi full kontroll over evalueringen av variablene, ved at evaluering og tilordning beskrives eksplisitt i interpreten.

Datatyper types ved en funksjonsnotasjon hvor typen på verdien angis eksplisitt. For eksempel vil heltallet 5 og den boolske verdien sann uttrykkes med termene $int(5)$ og $bool(true)$. Dette er eksempler på verdier, og alle verdier er av sort Data. En liste av uttrykk og/eller verdier er en term på formen $list(int(5)\ bool(true)\ldots)$. Uinitialiserte variabler får automatisk verdien *null*. Her er noen av deklarasjonene av datatypene fra interpreten.

subsorts *Oid* $<$ *Data* $<$ *Expr DataList* $<$ *List.*
subsorts *Qid* $<$ *Aid* $<$ *AidList* $<$ *List.*
subsorts *Nil* $<$ *AidList DataList.*
subsort *Aid* $<$ *Expr.*

op *null* : $->$ *Data* [*ctor*].
op *int* : *Int* $->$ *Data* [*ctor*].
op *str* : *String* $->$ *Data* [*ctor*].
op *bool* : *Bool* $->$ *Data* [*ctor*].
op *char* : *Char* $->$ *Data* [*ctor*].
op *float* : *Float* $->$ *Data* [*ctor*].
op *pair* : *Data Data* $->$ *Data* [*ctor*].
op *pair* : *Expr Expr* $->$ *Expr* [*ctor*].
op *list* : *DataList* $->$ *Data* [*ctor*].
op *list* : *List* $->$ *Expr* [*ctor*].
op *set* : *DataList* $->$ *Data* [*ctor*].
op *set* : *List* $->$ *Expr* [*ctor*].
op *ob* : *Qid* $->$ *Oid* [*ctor*]. $***$ *objektidentifikator*
op *nullptr* : $->$ *Oid* [*ctor*].

Verdien *nullptr* har blitt lagt til av meg, da det ikke eksisterte noen verdi for nullpekere for objektvariabler. Objektvariabler i Creol types med grensesnitt. Forøvrig antas det at typeanalyse har blitt utført i oversettelsen fra Creol til CMC.

4.2.7 Vakter og prosessorslippunkter

Termene som representerer de ulike vaktene er deklarert slik:

sorts *Guard Wait Return ExtGuard.*
subsorts *Return Expr Wait* < *Guard* < *ExtGuard.*

op *wait* : −> *Wait*[**ctor**].
op _? : *Qid* −> *Return*[**ctor**].
op _? : *Nat* −> *Return*[**ctor**].
op _?G(_) : *QidList* −> *ExtGuard.*
op *nothing* : −> *Guard*[**ctor**].
op _&_ : *GuardGuard* −> *Guard* [**ctor id** : *nothing* **assoc comm prec** 55].
op _&_ : *GuardExtGuard* −> *ExtGuard* [**ctor ditto**].

Vaktene brukes sammen med await-setningen slik:

await wait betyr at prosessen skal suspenderes betingelsesløst én gang.

await *myCall* **?** betyr at prosessen suspenderes inntil det foreligger retur fra den metoden som har merkelappverdi lik verdien til den brukerdefinerte CMC-variabelen *myCall*.

await *N* **?** betyr at prosessen suspenderes inntil det foreligger retur fra den metoden som har merkelappverdi lik *N*.

await *myCall* **?G(***outVar*1 *outVar*2 . . .**)** er en vakt med sideeffekt. Den innebærer suspendering inntil metoderetur foreligger, hvorpå CMC-variablene i listen *outVar*1 *outVar*2 . . . tilordnes returverdiene.

await nothing betyr at prosessen ikke lenger skal suspenderes. Som vi ser av deklarasjonen av &-operatoren for vakter, er *nothing* &-operatorens identitetselement.

Merk at *Expr* er subsort av *Guard*, slik at et uttrykk som $N < N'$ også er en vakt. For eksempel betyr setningen *await* $(N < N')$ & N''? at denne prosessen ikke skal fortsette kjøringen sin før N er mindre enn N' og vi har mottatt metoderetur fra det kallet/den metoden som har en merkelappverdi lik N''.

4.2.8 Meldinger

Meldinger konstrueres med følgende operatorer:

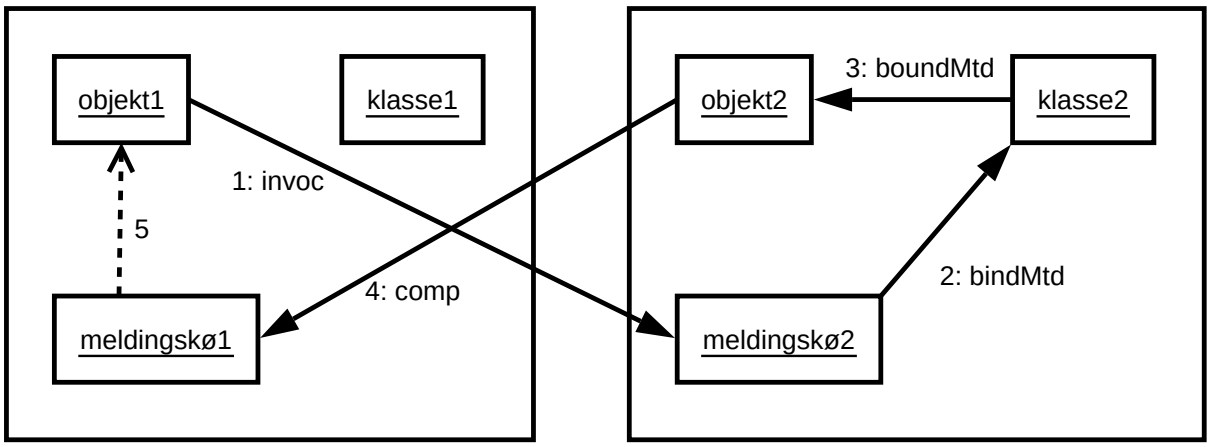
op *invoc*(_, _, _) : *Oid Mid DataList* −> *Msg* [**ctor**].
op *bindMtd* : *Oid Qid List InhList* −> *Msg* [**ctor**].
op *boundMtd* : *Oid Process* −> *Msg* [**ctor**].
op *comp*(_) : *DataList* −> *Msg* [**ctor**].

Det kan være nyttig å ha et øye på figur 4.1 på neste side mens du leser dette:

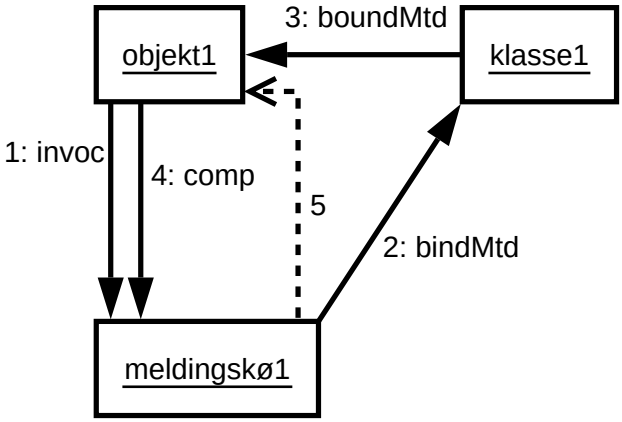
invoc Sendes fra det kallende objektet til det kalte objektets meldingskø. Gir signal om å sende tilbake en instans av en bestemt metode. Parametre: Objektnavnet til det kalte objektet, metodeidentifikatoren og en liste av verdier som inkluderer avsender sitt objektnavn, kallets merkelappverdi og reelle parametre.

bindMtd Intern kommunikasjon fra det kalte objektets meldingskø til det kalte objektets klasser. Brukes av mottakeren av en invoc-melding til å finne den spesifiserte metodedefinisjonen i en av sine klassedefinisjoner i arvehierarkiet. Parametre: Det kalte objektets navn, metodenavnet, listen av verdier og en liste av klassene i arvehierarkiet til den kalte metoden.

boundMtd Intern kommunikasjon fra det kalte objektets klasse til det kalte objektet. Inneholder den etterspurte metodeinstansen. Parametre: Det kalte objektet sitt navn og en metodeinstans.



(a) Eksternt metodekall



(b) Internt metodekall

Figur 4.1: Figur (a) viser informasjonsutvekslingen ved et eksternt kall. I figur (b) er det kallende og det kalte objektet det samme. I trinn 5 (stiplet pil) overføres returverdiene fra kallet. Dette er skjer direkte ved hjelp av en regel, uten bruk av en melding.

comp Svarmelding. Sendes fra det kalte objektet til det kallende objektets meldingskø. Inneholder en liste av verdier som inkluderer det kallende objektets (mottagerens) objektnavn, kallets merkelappverdi og returverdier fra den kalte metoden. Returverdiene stammer fra evalueringen av endsetningen, som har blitt satt inn i metodekroppen av interpreten.

I tillegg har vi følgende melding:

op $new\_(\_) : Cid\ List- > Msg.$

new-meldingen brukes til å instansiere klassedefinisjonene i CMC-programmet. Når vi definerer en initialtilstand som skal prosesseres av Maude, inkluderer vi en new-setning for hvert objekt som skal eksistere i systemet fra starten av kjøringen. Et CMC-program består altså av et antall klassedefinisjoner og et antall new-meldinger. Parametre i meldingen er navnet på den klassen som skal instansieres og en liste av verdier som objektattributtene skal initialiseres med.

4.2.9 Meldingskøer

Termer av sort $Queue$ konstrueres med følgende operator[1]:

subsort $Msg < MMsg$.
op $none : -> MMsg$ [**ctor**].
op $\_ + \_ : MMsg\ MMsg -> MMsg$ [**ctor assoc comm id** : $none$].

op $< \_ : Qu \mid Ev : \_,\ Keep : \_ > : Oid\ MMsg\ NatS -> Queue$ [**ctor**].

Qu Objektnavn. Dette er navnet på det objektet som køen tilhører.

Ev Multimengde av invoc- og comp-meldinger adressert til objektet som denne køen er assosiert med. Ved første øyekast synes det kanskje mer naturlig å bruke en liste av meldinger, side det tross alt er en kø. Distribuerte systemer er notorisk ikke-deterministiske, og vi må anta at kommunikasjonen i systemet er uforutsigbar. Ved å modellere meldingskøen uten noen spesifisert køordning, modellerer vi implisitt vilkårlig variasjon i transmisjonstiden til meldinger som sendes mellom objektene.

Keep Liste av merkelappverdier som brukes til å holde en oversikt over hvilke comp-meldinger køen aksepterer. comp-meldinger fra asynkrone kall uten svarsetning kastes.

4.2.10 Variabler brukt i interpreten

I figur 4.2 på neste side vises en alfabetisk oversikt over Maude-variabler som brukes i de reglene og ligningene vi skal se på. Når det er behov for flere variabler av samme sort, brukes ofte tegnet " ′ " som i $N\ N'\ N''$.

4.2.11 Meldingsutveksling og binding av metodekall

Vi antar at interpreten skal tolke et lokalt, synkront metodekall, og ser på modellen for dette skritt for skritt. Grunnen til at jeg velger å se på denne kallformen er at den er meste relevant for implementasjonen av synkronisert fletting. Det kallende og det kalte objektet er det samme, men jeg skiller mellom de to, for å vise hvordan de samme reglene brukes ved eksterne kall.

Reduksjon av synkront kall til asynkront Creol og CMC tilstreber en uniform behandling av alle typer metodekall. Som vi ser her, er et synkront kall definert som et asynkront kall umiddelbart etterfulgt av et asynkront mottak av metoderetur.

eq $< O : Ob \mid Pr : (MM(I;J)); P, Lcnt : N > \ < O : Qu \mid Keep : H >$
$=$
$< O : Ob \mid Pr : (!MM(I); (N?(J)); P), Lcnt : N > \ < O : Qu \mid Keep : H; N >$.

Videre defineres lokale kall som fjerne, med følgende enkle ligning:

eq $!Q(I) = !'this.Q(I)$.

Sending av invoc-melding Deretter blir fjerne kall omskrevet til meldinger med følgende regel:

rl $[remote - async - reply]$:
$< O : Ob \mid Pr : (!OE.Q(I)); P, Lvar : L, Att : A, Lcnt : N >$
$=>$
$< O : Ob \mid Pr : P, Lvar : L, Att : A, Lcnt : N + 1 >$
$invoc(eval(OE, (A, L)), Q, (O\ int(N)\ evalList(I, (A, L))))$.

Her ser vi at vi legger en invoc-melding i konfigurasjonen. Meldingen er da å betrakte som sendt. Termen $eval(OE, (A, L))$ innebærer en evaluering av objektpekeren OE i kontekst av objektets tilstand, bestående av de globale variablene A og de lokale variablene L. $evalList(I, (A, L)$ innebærer en evaluering av alle uttrykkene i listen I i kontekst av den samme tilstanden. Vi ser at Lcnt inkrementeres, slik at neste metodekall vil få en ubrukt merkelappverdi.

[1] Operatorattributtet **ctor** er lagt til av meg. Dette synes å være en triviell forglemmelse av den som i sin tid skrev denne Maude-koden.

var	A	: $Subst$.	Liste av globale variabelsubstitusjoner.
var	C	: Cid.	Klassenavn.
var	CV	: $ClVs$.	Liste av navn på superklasser med versjonsnummer.
var	DL	: $DataList$.	Liste av verdier.
var	E	: $Expr$.	Uttrykk.
var	G	: $Guard$.	Vakt.
var	H	: $NatS$.	Liste av heltall, merkelappverdier i meldingskø.
vars	I J	: $List$.	Liste av uttrykk og/eller verdier.
var	IN	: $List$.	Liste av innparametre i metodekall.
var	L	: $Subst$.	Liste av lokale variabelsubstitusjoner.
var	M	: $MMsg$.	Multimengde av meldinger.
var	MM	: Mid.	Metodeidentifikator i fjernt/lokalt/statisk kall.
var	Mt	: $MMtd$.	Multimengde av metodedefinisjoner.
var	N	: Nat.	Heltall. Brukes oftest til merkelappverdier.
var	O	: Oid.	Objektnavn.
var	OE	: $Expr$.	Objektpeker. Evalueres til metodenavn.
var	P	: $ProgList$.	Liste av programsetninger.
var	Q	: Qid.	Identifikator. Variabel-/metode-/klassenavn.
var	R	: $ProgList$.	Liste av programsetninger. “Resten” av setningene.
var	S	: $InhList$.	Liste av klasser som vi arver fra
var	SP	: $Prog$.	Sammensatt programsetning ([], |||, &, *await*).
var	St	: Stm.	Atomisk programsetning.
var	V	: Nat.	Versjonsnummer for klasser.
var	W	: $MProg$.	Multimengde av metodeinstanser (Process-termer).
var	X	: $Expr$.	Uttrykk, bl.a. brukt om boolsk vakt.
var	$PROC$	: $Process$.	Metodeinstans.
var	OUT	: $List$.	Liste av utparametre i metodekall.

Figur 4.2: Maude-variabler brukt i interpreten.

Mottak av invoc-melding Neste regel viser hvordan invoc-meldingen blir mottatt av det kalte objektets meldingskø:

$$\textbf{rl}\ [invoc-msg] : < O : Qu|Ev : M > \quad invoc(O, MM, DL)$$
$$=> < O : Qu|Ev : M + invoc(O, MM, DL) > .$$

Binding av kallet Neste skritt er å oversette invoc-meldingen til en bindMtd-melding.

$$\textbf{rl}\ [receive-call-req] :$$
$$< O : Ob|Cl : C\#V > \quad < O : Qu|Ev : M + invoc(O, Q, DL) >$$
$$=>$$
$$< O : Ob|Cl : C\#V > \quad < O : Qu|Ev : M > \quad bindMtd(O, Q, DL, C\#V[nil]).$$

BindMtd-meldingens siste parameter angir metodens klasse. Vi kommer til å finne metodedefinisjonen i denne klassen eller en av dens superklasser.

Følgende ligning tar seg av virtuell binding med multippel arv. BindMtd-meldingen traverserer alle de ulike arvegrafene som den opprinnelige klassen måtte ha, inntil metoden finnes. BindMtd-meldingens siste parameter er en liste av klasseidentifikatorer. Denne listen utvides med arvelisten til hver klasse i arvehierarkiet som ikke tilbyr den ønskede metoden. Når den riktige klassen er funnet, legges metodeinstansen i en boundMtd-melding. (Dette skjer med hjelpefunksjonen get, som jeg ikke går nærmere inn på.)

$$\textbf{eq}\ \ bindMtd(O, Q, I, (C\#V'[J])\ S') \quad < C : Cl|Inh : S, Mtds : Mt >$$
$$=$$
$$\textbf{if}\ Q\ in\ Mt\ \textbf{then}\ boundMtd(O, get(Q, Mt, I))\ \textbf{else}\ bindMtd(O, Q, I, S\ S')\ \textbf{fi}$$
$$< C : Cl|Inh : S, Mtds : Mt > .$$

Subtermen $(C\#V'[J])\ S'$ sier at vi tidligere ikke har lett etter metodedefinisjonen i klassen C eller klassene i listen S'. Variablene V' (klasserversjonsnummer) og J (klasseparametre) er irrelevante.

Når metodedefinisjonen er funnet, instansieres den, og metodeinstansens Process blir andre parameter boundMtd-meldingen i følgende ligning. boundMtd-meldingen finner objektet som skal kjøre metoden. Metodeinstansen kopieres over til objektets liste av ventende prosesser i PrQ.

$$\textbf{eq}\ \ boundMtd(O, (P', L')) \quad < O : Ob|PrQ : W >$$
$$= < O : Ob|PrQ : (clear(P'), L') : W > .$$

Hjelpefunksjonen clear() fjerner eventuelle innledende, sanne vakter fra koden.

Vi har nå kommet så langt at kallet er bundet og ligger i prosesskøen og venter på å bli skedulert. Det kan være verd å merke seg til senere at invoc-meldinger håndteres med regler, mens resten håndteres med ligninger.

4.2.12 Skedulering og kontekstbytte ved interne kall

Interne kall medfører per definisjon at den kallende og den kalte prosessen kjører i samme objekt. Ved et synkront kall, har vi at den kallende prosessen er skedulert, mens den kalte prosessen ligger i prosesskøen PrQ og venter på å bli skedulert. Vi må suspendere den kallende prosessen, skedulere den kalte prosessen og reskedulere den kallende når den kalte terminerer. For å bevare synkroniteten i kallet, kan vi ikke tillate at noen andre prosesser kjører mellom disse tre hendelsene.

I følgende regel ser vi at merkelappverdien N i den kallende prosessen i Pr matcher merkelappverdien i label i en av prosessene i PrQ. Prosessene "bytter plass", ved at den kallende prosessen legges i PrQ og den kalte prosessen legges i Pr.

$$\textbf{rl}\ [local-reentrace] :$$
$$< O : Ob|Pr : (N?(J)); P, PrQ : (P', ('caller : O), ('label : int(N)), L') : W,$$
$$\quad Lvar : L >$$
$$=>$$
$$< O : Ob|Pr : P'; continue(N), PrQ : (awaitN?; (N?(J)); P, L) : W,$$
$$\quad Lvar : ('caller : O), ('label : int(N)), L' > .$$

Merk hvordan de lokale variablene håndteres. Den kallende prosessen sine lokale variabler L flyttes fra Lvar til Process-termen i PrQ. Den kalte prosessens lokale variabler L' flyttes til Lvar samtidig med at prosessens programsetninger plasseres i Pr. I tillegge ser vi at begge de involverte setningslistene har blitt utvidet: Den nyskedulerte prosessen har fått en continue-setning sist i setningslisten sin. Dette er for at vi garantert skal få kjørt den kallende prosessen når den kalte metoden terminerer. Den kallende, suspenderte prosessen har fått en await-setning først i listen sin. Vi kommer tilbake til await-setninger og vakter, men foreløpig kan vi nøye oss med å si at vitsen med dette er å hindre at prosessen blir skedulert før metodereturen foreligger.

Kallet vårt har nå blitt bundet og prosessen skedulert. Hvis denne prosessen før eller siden terminerer korrekt, vil følgende regel bli brukt:

rl $[reply] : < O : Ob | Pr : (end(J)); P, Lvar : L, Att : A >$
$=>$
$< O : Ob | Pr : P, Lvar : L, Att : A > \quad comp(evalList('caller\ 'label\ J, (A, L)))$.

End-setningen fører til at en comp-melding blir lagt i konfigurasjonen. Den inneholder en liste av returverdier. Hjelpefunksjonen evalList() evaluerer alle variablene i comp-meldingen i kontekst av objektets tilstand (A, L).

Her ser vi at comp-meldingen blir mottatt av det kallende objektet sin meldingskø:

rl $[reply - msg]$:
$< O : Qu | Ev : M, Keep : H; N; H' > \quad comp(O\ int(N)\ DL)$
$=>$
$< O : Qu | Ev : M + comp(O\ int(N)\ DL), Keep : H; H' >$.

Vi husker at den kallende prosessen fikk en await-setning lagt til i begynnelsen av setningslisten sin for at prosessen ikke skulle bli skedulert før metodereturen forelå. Nå foreligger metodereturen, i form av en comp-melding i meldingskøen. Denne ligningen fjerner await-setningen, slik at prosessen kan skeduleres etterpå.

eq $< O : Ob | PrQ : ((await(N? \ \&\ G); P), L') : W >$
$< O : Qu | Ev : M + comp(O\ int(N)\ DL) >$
$=$
$< O : Ob | PrQ : ((await\ G; P), L') : W >$
$< O : Qu | Ev : M + comp(O\ int(N)\ DL) >$.

Følgende regel brukes når den kalte metoden terminerer. Den tvinger fram reskedulering av den kallende prosessen. Match på merkelappverdien N gjør at vi finner den riktig prosessen i PrQ.

rl $[continue]$:
$< O : Ob | Pr : continue(N), PrQ : (((N?(J)); P), L') : W, Lvar : L >$
$=>$
$< O : Ob | Pr : ((N?(J)); P), PrQ : W, Lvar : L' >$.

Vi ser at vi får et nytt kontekstskifte ved at Lvar overskrives med den skedulerte prosessens variabler L'. Det er ingen vits i å ta vare på den kjørende prosessen sine lokale variabler, fordi den terminerer i dette omskrivingssteget og har allerede sendt fra seg returverdiene.

Til slutt ser vi at den reskedulerte, kallende prosessen henter returverdiene fra meldingskøen. Hver av utvariablene i listen J tilordnes hver sin verdi fra listen DL.

eq $< O : Ob | Pr : (N?(J)); P > \quad < O : Qu | Ev : M + comp(O\ int(N)\ DL) >$
$=$
$< O : Ob | Pr : (J := DL); P > \quad < O : Qu | Ev : M >$.

4.2.13 Evaluering av vakter

Vi bruker hjelpefunksjonen enabled() til å evaluere vakter.

op $enabled : ProgListSubstMMsg- > Bool.$

Parametrene er en setningsliste, prosessens variabler (både lokale og globale) og køen av meldinger som ligger i meldingskøen. Det siste brukes til å evaluere de vaktene som sjekker om det foreligger metoderetur. En setningsliste er *beredt* hvis enabled() returnerer sann/true når setningen med tilhørende tilstand og meldingskø blir gitt som parameter.

eq $enabled(SP; SP'; P, L, MM) = enabled(SP, L, MM).$
eq $enabled(P[]P', L, MM) = enabled(P, L, MM)$ **or** $enabled(P', L, MM).$
eq $enabled(P|||P', L, MM) = enabled(P, L, MM)$ **or** $enabled(P', L, MM).$
eq $enabled(await(wait\ \&\ G), L, MM) = false.$
eq $enabled(await(X\ \&\ G), L, MM) = evalB(X, L)$ **and** $enabled(await\ G, L, MM).$
eq $enabled(await((Q?)\&\ G), L, MM) = inqueue(evalI(Q, L), MM)$ **and** $enabled(await\ G, L, MM).$
eq $enabled(await((N?)\&\ G), L, MM) = inqueue(N, MM)$ **and** $enabled(await\ G, L, MM).$
eq $enabled(empty, L, MM) = true.$
eq $enabled(St, L, MM) = true.$

Hjelpefunksjonene evalB og evalI tar et CMC-uttrykk og evaluerer et til henholdsvis en boolsk verdi og en heltallsverdi. Hjelpefunksjonen inqueue() gir en boolsk verdi som forteller om svarmeldingen tilhørende den gitte merkelappverdien har ankommet.

Det er den første setningen i en setningsliste som avgjør om setningslisten er beredt. Funksjonen kalles rekursivt for å finne den første setningen i en setningsliste. []- og |||-setninger er sammensatt av flere setningslister, så enabled() brukes på hver av disse listene. Wait-vakten er alltid usann. Innledende wait-vakter fjernes fra alle prosesser som ligger i prosesskøen PrQ, slik at prosessen kan reskeduleres. Metodereturvakter evalueres i kontekst av meldingskøen; vakten er sann hvis den etterspurte meldingen ligger i køen. Setninger uten vakter (og som ikke er []- eller |||-setninger) er definert til å være beredt, da det er meningen at de skal behandles som om de har sanne vakter.

4.2.14 Prosessorslippunkter

Suspendering Dette er regelen for suspendering:

crl $[suspend]$:
$< O : Ob | Pr : P, PrQ : W, Lvar : L, Att : A >$
$< O : Qu | Ev : M >$
$=>$
$< O : Ob | Pr : empty, PrQ : W : (clear(P), L), Lvar : no, Att : A >$
$< O : Qu | Ev : M >$
if not $enabled(P, (L, A), M).$

Den sier at hvis den neste programsetningen i en prosess ikke er enabled, suspenderes prosessen. Vi ser at prosessens lokale variabler blir tatt vare på ved å lagre dem sammen med setningslisten i PrQ.

Boolske vakter La oss anta at første setning er en await-setning som er beredt. Da brukes ikke den ovenforstående regelen. Avhengig om await-setningen inneholder en metodereturvakt eller ikke, brukes en av de følgende reglene replyguard-inQ eller boolguard.

rl $[boolguard]$:
$< O : Ob | Pr : await\ E; P, PrQ : W, Lvar : L, Att : A >$
$=>$
if $evalB(E, (A, L))$ **then**
$< O : Ob | Pr : P, PrQ : W, Lvar : L, Att : A >$ **else**
$< O : Ob | Pr : empty, PrQ : W : (await\ E; P, L), Lvar : no, Att : A >$ **fi**.

Her ser vi at hvis en boolsk vakt er sann, så slettes hele await-setningen.[2]

[2]Hvordan får vi match på denne regelen når den boolske vakten er sammensatt? For at en vakt skal kunne være inneholdt i variabelen E (sort Expr), skulle man tro at sorten Guard var subsort av Expr, men det er faktisk motsatt. Svaret er at det finnes en ligning **eq** $E\ \&\ E' = E$ **and** E'. som reduserer en sammensatt boolsk vakt til en term av sort Expr.

Metodereturvakter En await-setning kan inneholde en metodereturvakt. Som vi husker fra 4.2.7 på side 39, er det to typer metodereturvakter: Enten er merkelappverdien angitt med en Maude-variabel N av sort Nat eller så er den angitt av en term Q av sort Qid, som representer en CMC-variabel. Fra CMC sin synsvinkel er en Maude-variabel en konkret verdi. Vi ønsker uniform behandling av de to typene metodereturvakter. Det får vi ved å evaluere CMC-variabelen med hjelpefunksjonen evalI(). Dermed har vi at alle metodereturvakter angir merkelappverdien i form av Maude-variabelen N:

eq $< O : Ob | Pr : await(Q? \ \& \ G); P, Lvar : L >$
$= < O : Ob | Pr : await(evalI(Q, L)? \ \& \ G); P, Lvar : L > .$

Nå kan vi bruke følgende regel:

rl $[replyguard - inQ]$:
$< O : Ob | Cl : Pr : await(N? \ \& \ G); P >$
$< O : Qu | Ev : M + comp(O \ int(N) \ DL) >$
$=>$
$< O : Ob | Pr : await \ G; P >$
$< O : Qu | Ev : M + comp(O \ int(N) \ DL) > .$

Den sier at hvis en etterspurt metodereturmelding ligger i meldingskøen, så fjernes den tilhørende vakten fra await-setningen. Det kan godt hende at await-setningen inneholder flere metodereturvakter, og i så fall brukes regelen på nytt. Await-setningen kan inneholde en blanding av metodereturvakter og boolske vakter som kan være vilkårlig stor, men siden operatoren & er kommutativ, vil regelen alltid finne en metodereturvakt hvis den er der. Etter hvert som vakter fjernes fra await-setningen, vil vi sitte igjen med setningen $await \ nothing$, som slettes med ligningen **eq** $(await \ nothing); empty = empty$.

Metoderetur til suspenderte prosesser Anta at en prosess har blitt suspendert og ligger i PrQ. La oss si at det var en metodereturvakt som førte til suspenderingen. Så kommer den forventede metodereturen. Følgende ligning forholder seg til dette ved å fjerne vakten mens prosessen ligger i PrQ. Dermed er prosessen beredt og skedulerbar igjen.

eq $< O : Ob | PrQ : ((await \ N? \ \& \ G; P), L') : W >$
$< O : Qu | Ev : M + comp(O \ int(N) \ DL) >$
$=$
$< O : Ob | PrQ : ((await \ G; P), L') : W >$
$< O : Qu | Ev : M + comp(O \ int(N) \ DL) > .$

Skedulering Neste regel skedulerer en prosess når prosessoren er ledig. Det kan være første gang prosessen kjører, eller prosessen kan ha blitt suspendert på et tidligere tidspunkt.

crl $[PrQ - enabled]$:
$< O : Ob | Pr : empty, PrQ : (P', L') : W, Lvar : L, Att : A >$
$< O : Qu | Ev : M >$
$=>$
$< O : Ob | Pr : P', PrQ : W, Lvar : L', Att : A >$
$< O : Qu | Ev : M >$
if $enabled(P', (A, L'), M)$.

4.2.15 Operatorene for ikke-deterministisk valg og fletting

Operatorene som danner setningene for henholdsvis ikke-deterministisk valg og fletting av setningslister er deklarert slik:

op $\_[]\_ \ : \ ProgList \ ProgList \ -> \ Prog$ [**ctor assoc**].
op $\_|||\_ \ : \ ProgList \ ProgList \ -> \ Prog$ [**ctor assoc**].

Reglene for disse setningene bruker hjelpefunksjon ready(), som er en utvidelse av enabled(). Vi bruker enabled() til å avgjøre om en prosess er beredt, og hvis den ikke er beredt skal den suspenderes. Vi kaller det *passiv venting* når en setning suspenderes. Prosessen ligger da i PrQ så lenge den ikke er beredt. Vi kaller det *aktiv venting* hvis en prosess er beredt, men likevel ikke har noen progresjon. Dette er det samme som blokkering. Et typisk eksempel på dette er blokkering ved et synkront kall. Hjelpefunksjonen ready() brukes til å avgjøre om en setning fører til aktiv venting. En setningsliste som ikke fører til aktiv venting kaller vi "klar". En beredt setning er ikke nødvendigvis klar, men en klar setning er alltid beredt.

Ikke-deterministisk valg Vi ser på reglene for ikke-deterministisk valg:

crl $[nondet - p1]$:
$< O : Ob|Pr : (P[]P'); R, Lvar : L, Att : A > \quad < O : Qu|Ev : M >$
$=>$
$< O : Ob|Pr : P; R, Lvar : L, Att : A > \quad < O : Qu|Ev : M >$
if $ready(P, (L, A), M)$.

crl $[nondet - p2]$:
$< O : Ob|Pr : (P[]P'); R, Lvar : L, Att : A > \quad < O : Qu|Ev : M >$
$=>$
$< O : Ob|Pr : P'; R, Lvar : L, Att : A > \quad < O : Qu|Ev : M >$
if $ready(P', (L, A), M)$.

Her har vi to setningslister P og P', og det spiller ingen rolle hvilken av dem som får kjøre. Vi velger en av dem, og fjerner resten av []-setningen. Det er mest rasjonelt å kjøre en setningsliste som ikke medfører venting, derav betingelsene. P og P' kan inneholde []-setninger med vilkårlig mange elementer, og da brukes bare disse reglene om igjen, inntil vi sitter igjen med en setning som ikke har [] som toppsymbol.

Fletting Slik er den operasjonelle semantikken til fletting definert:

crl $[merge - first]$:
$< O : Ob|Pr : ((SP; P)|||P'); R, Lvar : L, Att : A > \quad < O : Qu|Ev : M >$
$=>$
$< O : Ob|Pr : SP; (P|||P'); R, Lvar : L, Att : A > \quad < O : Qu|Ev : M >$
if $ready(SP, (L, A), M)$.

crl $[merge - second]$:
$< O : Ob|Pr : (P|||P'); R, Lvar : L, Att : A > \quad < O : Qu|Ev : M >$
$=>$
$< O : Ob|Pr : (P'|||P); R, Lvar : L, Att : A > \quad < O : Qu|Ev : M >$
if $ready(P', (L, A), M)$ **and not** $ready(P, (L, A), M)$.

eq $(empty|||P) = P$.

Den første regelen er ment å velge det første elementet i |||-setningen og kjøre det, på den betingelse at elementet er ready. Den andre regelen er ment å endre rekkefølgen på elementene i |||-setningen, slik at hvis det finnes et ready element, så plasseres det først i setningen. Ligningen fjerner en |||-operator etter at merge-first har ekstrahert den siste setningen i et element. Det er en liten feil i dette, som beskrives i 5.5.1 på side 59.

Kapittel 5

Implementasjon av synkronisert fletting

I dette kapitlet definerer jeg en operasjonell semantikk for synkronisert fletting i Creol. Framstillingen er problemorientert og gjenspeiler utviklingen jeg har vært gjennom, med en gradvis økende forståelse for problemene. Kapitlet inneholder både en beskrivelse av Maude-implementasjonen av synkronisert fletting og diskusjoner av valg som har blitt gjort underveis. Implementasjonen er ikke en ren utvidelse av interpreten, fordi jeg har endret definisjonen av operatorene [] og |||.

Kapitlet starter med en metodologisk betraktning om utviklingsprosessen.

Det kan nevnes allerede nå at i slutten av kapitlet kritiserer jeg min egen løsning og foreslår forbedringer. Grunnen til at implementasjonen ikke er korrigert i henhold til denne kritikken, er at tidsrammen for denne oppgaven ikke tillater det.

Jeg samler alle sort- og operatordeklarasjoner jeg har gjort i forbindelse med utvidelse og endring av interpreten i figur 5.1 på neste side, så det skal bli lett å finne tilbake til dem ved behov. De forklares etter hvert.

Jeg kommer ofte til å skissere Maude-regler, -ligninger og omskrivingssekvenser med pseudokode. Dette er nyttig for å visualisere og drøfte ideer på et tidlig stadium. Det er ofte lettest å tenke løst omkring ulike implementasjonsalternativer når vi abstrahere vekk fra korrekt Maude-syntaks og korrekt bruk av variabler. I slike skisser bruker jeg pilen $\rightsquigarrow$ istedenfor Maude sine symboler `=` og `=>` for henholdsvis ligningner og regler . Det er meningen å ikke skille mellom ligninger og regler, da jeg bare ønsker å uttrykke noe så upresist som at vi kommer fra det ene til det andre. Jeg lar $\rightsquigarrow$ også representere en omskrivingssekvens som kanskje må implementeres med bruk av flere regler og/eller ligninger. I skissene blander jeg mellom å bruke metavariabler som for eksempel S1, S2,... for vilkårlige programsetninger og variabler som er deklarert i figur 4.2 på side 42.

Både i skisser og korrekt gjengivelse av Maude-kode utelater jeg irrelevante deler av termer som representerer et objekt. Et objekt i en Maude-regel kan representeres slik: $< O : Ob : |Cl : CV, Pr : P, PrQ : W, Lvar : L, Att : A, Lcnt : N >$. En regel/ligning referer vanligvis ikke til alle attributtene i objektet. La oss si at vi bare er interessert i Pr-attributtet i objektet. For å øke lesbarheten, lager jeg en forkortelse av termen, slik at objektet representeres på denne måten: $< O : Ob : |Pr : P >$.

I en setning S1 & S2,hvor S1 og S2 er vilkårlige programsetninger, kaller jeg S1 og S2 for *setningsledd*. Et ledd kan selv være sammensatt. For eksempel er S1 & S2 et ledd i setningen (S1 & S2) [] S3. Vi kan supplere denne definisjonen ved å bruke subsort-deklarasjonene i figur 5.1 på neste side: En setningssekvens S1 er et ledd i setningen S1 & S1' hvis S1 ikke er av sort SyncMerge. Tilsvarende er S2 et ledd i S2 [] S2' hvis S2 ikke er av sort Nondet og S3 er et ledd i S3 ||| S3' hvis S3 ikke er av sort Merge.

5.1 Implementasjonsprosessen

Prosessen fram til den endelige løsningen har ikke foregått ved at jeg først har satt meg ned og tenkt ut alt og deretter tastet inn min "perfekte" idé. Jeg startet med en kort og uformell definisjon av synkronisert fletting. Da jeg begynte å skrive Maude-kode, ble det raskt klart at valgene jeg gjorde hadde betydning for semantikken til konstruktet. Jo mer jeg kodet, jo flere valg ble jeg nødt til å gjøre som var en konkretisering av semantikken. Ved å velge én løsning, ble jeg nødt til å velge bort de andre alternativene. Dette er i tråd med at Maude-programmer er semantiske spesifikasjoner.

$sorts\ SyncMerge\ Merge\ Nondet\ Run\ RunMtd\ ProgElement.$
$subsorts\ SyncMerge\ Merge\ Nondet\ Run\ RunMtd\ ProgElement\ <\ Prog.$

$op\ \_\&\_\ :\ ProgList\ ProgList\ ->\ SyncMerge\ [ctor\ assoc\ comm].$
$op\ \_|||\_\ :\ ProgList\ ProgList\ ->\ Merge\ [ctor\ assoc\ comm].$
$op\ \_[]\_\ :\ ProgList\ ProgList\ ->\ Nondet\ [ctor\ assoc\ comm].$
$op\ \_///\_\ :\ ProgList\ ProgList\ ->\ Prog\ [ctor\ assoc].$
$op\ insert\ :\ Nat\ ->\ Stm\ [ctor].$
$op\ element\ :\ Process\ ->\ ProgElement\ [ctor].$
$op\ awaitingElement\ :\ Process\ ->\ ProgElement\ [ctor].$
$op\ runMtd\ :\ Process\ ->\ RunMtd\ [ctor].$
$op\ run\ :\ ProgList\ ->\ Run\ [ctor].$
$op\ toggleMergeOrder\ :\ ProgList\ ->\ ProgList.$

Intuisjon:

/// brukes sammen med |||-operatoren for å implementere den varianten av flettemekanismen som gir delvis ordnet fletting. Forskjellen mellom ||| og /// er at /// ikke er kommutativ.

toggleMergeOrder brukes i flettemekanismen for å bytte mellom |||- og ///-operatoren.

element brukes i mekanismen for ekspansjon av metodekall til å oppbevare en del av en &-, |||- eller []-setning eller en metodeinstans.

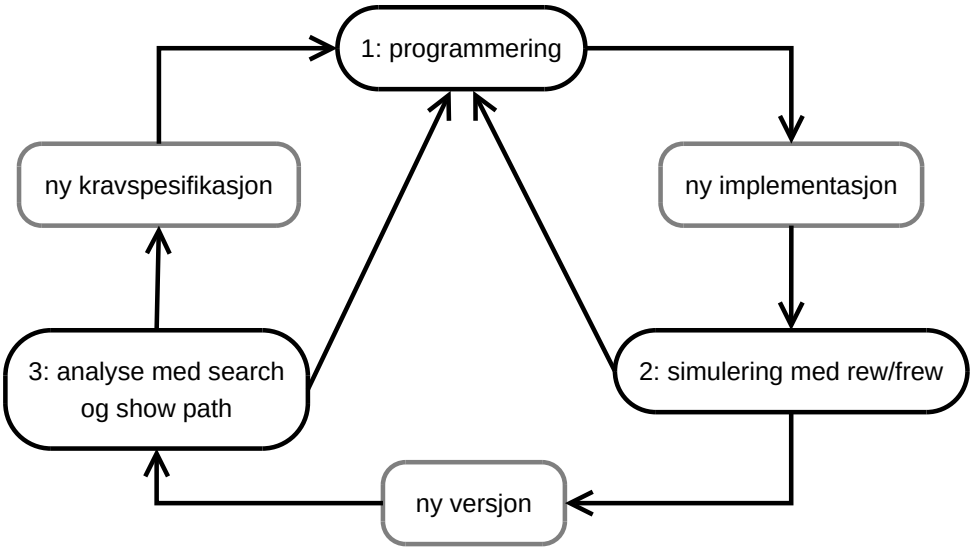
awaitingElement brukes i mekanismen for ekspansjon av metodekall til å oppbevare en term hvor en subterm har blitt hentet ut og plassert i en element-setning. En awaitingElement-setning inneholder alltid en insert-setning som markerer hvor den ekstraherte subtermen stod.

insert brukes i mekanismen for ekspansjon av metodekall til å angi et sted i en awaitingElement-setning hvor en annen term skal plasseres på et senere tidspunkt.

run brukes i mekanismen for ekspansjon av metodekall til 1) å isolere setningsledd i en sammensatt setning, slik at det blir mulig å operere på ett ledd om gangen og 2) å markere at leddet er behandlet slik at vi unngår ikke-terminering.

runMtd representerer et ekspandert metodekall. Inneholder metodens kode og lokale tilstand, slik at 1) kallparametre evalueres i riktig kontekst ved videre ekspansjon og 2) det blir lett å implementere kontekstskifter ved kjøring av ekspanderte kall.

Figur 5.1: Deklarasjoner av sorter, subsorter og operatorer. Variabler er deklarert i figur 4.2 på side 42.



Figur 5.2: Iterativ utviklingsprosess.

5.1.1 Iterativ prosess

Maude legger til rette for iterativ utvikling av systemspesifikasjoner ved blant annet å tilby kommandoene rew, frew, search og show path. Figur 5.2 viser hvordan utviklingsprosessen har vært. I figuren representerer grå bokser tilstander og sorte bokser aktiviteter. Hver iterasjon har startet med en mer eller mindre uformell og uklar kravspesifikasjon og forståelse av semantikken til synkronisert fletting. Så har jeg tenkt ut en idé for hvordan jeg kan implementere synkronisert fletting slik at kravspesifikasjonen blir oppfylt. Jeg har deretter forsøkt å implementere ideen. Ved simpelthen å begynne å kode, har det hendt at jeg har fått idéer som var bedre, eller skjønt raskt at jeg har vært på fundamentalt feil spor.

Etter å ha implementert en løsning eller deler av en løsning, laget jeg noen CMC-programmer og kjørte dem med Maude sin rew- og frew-kommando. Dette var den primære metoden for debugging. På denne måten oppdaget jeg også feil som var fatale, og som jeg da så at var konsekvenser av en dårlig designidé i utgangspunktet. Når simulering med rew/frew ga ønsket oppførsel, kalte jeg løsningen for en ny versjon.

Når jeg hadde en tilsynelatende velfungerende versjon, testet jeg den ved å analysere de samme CMC-eksemplene med kommandoen search. Vanligvis søkte jeg etter alle mulige sluttilstander. Disse tilstandene studerte jeg manuelt. På denne måten fant jeg ut om det var mulig for programmet å terminere i en uønsket tilstand. I noen tilfeller var det ikke mulig å se hvordan en uønsket sluttilstand hadde oppstått. Da brukte jeg "show path"-kommandoen. Dette ga meg muligheten til å oppdage mer subtile problemer med programmet. Maude sin søkekommando har hjulpet meg med å finne feil som har vært vanskelig å se manuelt og som enten ville ha blitt oppdaget ved simulering på et senere tidspunkt, eller ikke i det hele tatt.

Etter å ha vært gjennom disse fasene, pleide jeg å ende opp med en dypere forståelse, slik at jeg kunne begynne på nytt, med nye ideer til forbedringer og varianter av semantikken.

5.1.2 Testing og debugging

Testing av implementasjonen av synkronisert fletting har bestått av CMC-eksempler som har blitt analysert med bruk av rew/frew og search. I ettertid ser jeg at testingen kunne vært mer systematisk. Ny Maude-kode har blitt laget med fokus på å løse nye, spesifikke problemer eller fylle spesifikke mangler. Følgelig har jeg designet nye CMC-eksempler og rew/search-kommandoer med disse problemene for øyet. Faren med denne framgangsmåten er at man kan komme i skade for å kaste barnet ut med badevannet: Når en versjon har bestått den nye testen, kan versjonen ha blitt endret slik at den ville ha feilet den gamle testen. Siden jeg tross alt har hatt noen grad av bevissthet på dette, har flere av versjonene vært gjennom de samme testene.

Testdrevet utvikling [6] er et konsept som har som hovedidé å lage testene så tidlig som mulig, gjerne før man begynner å implementere systemet overhodet. Dette kunne jeg med fordel ha benyttet meg av. Jeg kunne ha startet utviklingsprosessen med å lage en test bestående av noen CMC-programmer med tilhørende rew/search-kommandoer. Disse burde være designet for å teste de sentrale eller nødvendige egenskapene ved synkronisert fletting, og blitt brukt på hver versjon av implementasjonen. Metoden tvinger fram et fokus på systemkravene som påvirker implementasjonsvalgene. Jeg tror dette ville vært sunt for tankegangen og idéutviklingen min.

Framgangsmåten ville også gjort det lett å forklare hvorfor jeg har forlatt en versjon, ved at jeg kunne ha henvist til versjonens reaksjon på testen. Siden testingen min ikke har vært 100% systematisk er det vanskelig å dokumentere hvorfor jeg har forkastet spesifikke ideer.

Men kanskje er det naivt å tenke at man kan utvikle en ferdig testsuite før man begynner på implementasjonen. Det er vanligvis slik at kravspesifikasjonen utvikles, modnes og forandrer seg under prosessen med å implementerer systemet. Slik har det også vært for meg. Dermed er det uunngåelig at man ikke vet hva man skal teste før et godt stykke ut i implementasjonsprosessen.

5.2 Initiell forståelse av semantikken til synkronisert fletting

Utgangspunktet for arbeidet med synkronisert fletting er den uformelle definisjonen som vi presenterte i 3.2.4 på side 23, og som jeg repeterer her:

> Synkronisert fletting — (**await** $G1; S1$) & (**await** $G2; S2$) — er definert som **await** $G1 \wedge G2; (S1 \| S2)$. Ikke-bevoktede argumenter til & behandles som om de er bevoktet av en sann vakt. Metodekall som er synkrone og lokale ekspanderes.

Det er kravet om ekspansjon av metodekall som gjør implementasjonen av synkronisert fletting vanskelig. Hva menes med at metodekall ekspanderes? Hvis vi har en metode m() med metodekropp $await\ G1; S1$, er $m()$ & $(await\ G2; S2)$ definert som $await\ G1$ & $G2; S1 \,|||\, S2$. Når vi prøver å konkretisere denne definisjonen og se for oss en Maude-implementasjon, viser det seg å være langt fra enkelt. Vakten og resten av metodekroppen fra m() kan ikke uten videre settes inn i annen kode. Kjøringen av m() innebærer å kjøre en annen prosess. En metode har et tilstandsrom med variabler som er lokale for metoden. I tillegg kan vakten G2 i m() bestå av et uttrykk som avhenger av parametrene gitt i kallet på m(). Det kreves et *kontekstskifte* for å kjøre m().

Anta at vi har en setning $S1$ & $m()$. For at vi skal kunne bruke enabled() til å avgjøre om m() er beredt, trenger vi 1) m() sin kode, 2) m() sine instansierte, lokale variabler, 3) objektattributtene til objektet som &-setningen kjører i og 4) meldingene i objektets meldingskø. På grunn av 1) og 2) er vi nødt til å binde kallet på m() før vi kan avgjøre om &-setningen er beredt. Vi ser at den overordnede hendelsesrekkefølgen i kjøringen av en &-setning må være:

1. Binding av eventuelle innledende metodekall i hvert ledd.
2. Testing av om alle leddene er beredte samtidig, ved bruk av enabled() som er beskrevet i avsnitt 4.2.13 på side 44.
3. Hvis &-setningen er beredt: Kjøring av alle leddene. Hvis &-setningen ikke er beredt: Suspendering av prosessen.

Jeg har endret definisjonen av operatoren [] for ikke-deterministisk valg til også å benytte seg av ekspanderte metodekall. Grunnen til dette var at jeg så et Creol-kodeeksempel i et utkast til en artikkel, hvor en slik variant av [] ble benyttet. Mengden merarbeid på grunn av dette var minimal, siden mekanismen er lik for [] og &. Jeg har ikke vurdert hvorvidt det er ønskelig med ekspandering av kall i kontekst av []. For å beholde fokuset på & skriver jeg ikke noe mer om [] enn det jeg gjør i avsnitt 5.8.

Beredthetssjekk

Definisjonen legger opp til at interpreten manipulerer CMC-setningen
$(await\ G1; S1)$ & $(await\ G2; S2)$ ved å legge til en ny await-setning med vakter som er funnet i leddene i &-setningen. Interpreten virker slik at gitt
$await\ G1$ & $G2; S1 \,|||\, S2$ så vil $S1 \,|||\, S2$ bli kjørt hvis $await\ G1$ & $G2$ er beredt. Hvorvidt await-setningen er beredt avgjøres som tidligere nevnt med hjelpefunksjonen enabled().

Vi kan oppnå den samme semantikken uten å generere den ekstra await-setningen. Det gjør vi ved å utvide definisjonen av enabled() med en spesifikasjon av hvordan den skal forholde seg til en &-setning: (P og P' er vilkårlige programsetninger, L en samling variabelbindinger og M en samling meldinger fra objektets meldingskø. L og M er konteksten som P & P' sjekkes i.)

$$eq\ enabled(P\ \&\ P', L, M) \quad = enabled(P, L, M)\ and\ enabled(P', L, M).$$

Nå kan vi bruke enabled() til å finne ut om alle vakter som forekommer fremst i alle leddene i en &-setning er sanne samtidig. Vi kan gi hele setningen $(await\ G1; S1)\ \&\ (await\ G2; S2)$ som parameter til enabled() i en betinget omskrivingsregel. Hvis setningen er beredt, fjerner vi await-setningene fra leddene, og forekomstene av & erstattes med |||. Vi har da at den syntaktiske omskrivingen

$$(await\ G1; S1)\ \&\ (await\ G2; S2)\ \rightsquigarrow\ await\ G1\ \&\ G2; S1\ |||\ S2$$

er semantisk nøyaktig den samme som

$$(await\ G1; S1)\ \&\ (await\ G2; S2)\ \rightsquigarrow\ S1\ |||\ S2$$
hvis &-setningen er beredt, som avgjøres med $enabled((G1; S1)\ \&\ (G2; S2),\ L, M)$.

Jeg valgte den siste varianten, til tross for at den første er mest intuitiv. Grunnen var i utgangspunktet at jeg så at |||- og []-setninger ble sjekket på denne måten. Valget ble gjort før jeg hadde full oversikt over alle aspektene ved implementasjonen av synkronisert fletting, men i ettertid viser det seg å være et heldig valg. Som vi skal se snart, kan leddene ha ulike tilstandsrom. Hvis vi skal generere en ekstra vakt på grunnlag av vaktene fra leddene, må denne nye vakten i så fall evalueres i kontekst av disse ulike tilstandsrommene. Det virker ikke som en praktisk løsning.

5.3 Håndtering av metodeinstanser i kontekst av synkronisert fletting

Anta at vi har en setning $S1\ \&\ m(IN; OUT)$. La oss si at vi binder kallet på m(), og har mottatt metodeinstansen i form av en term av sort Process (MC, ML), hvor MC er m() sin programkode og ML er m() sine instansierte, lokale variabler. (MC, ML) må gis som parameter til enabled(). Hvordan skal interpreten håndtere termen (MC, ML)? Nærmere bestemt:

- På hvilken måte skal (MC, ML) eller informasjonen i (MC, ML) oppbevares i objektet slik at den er tilgjengelig for enabled()?
- Hva er den mest hensiktsmessige måten å oppbevare informasjonen i (MC,ML) på for den kommende kjøringen av prosessen (MC, ML)?

De følgende skissene av regler er ment å visualisere drøftingen av de ulike semantiske alternativene. Jeg utelater objektattributter som ikke er relevante for spørsmålet om håndtering av metodeinstansen.

Det er mulig å se for seg fire ulike steder å oppbevare metodeinstansen på: 1) I objektets meldingskø, 2) i objektets programkø, 3) vi kan utvide objektet med et nytt attributt og legge den der eller 4) i objektets Pr-attributt. Jeg har valgt alternativ 4).

5.3.1 Alternativ 1: Oppbevaring i meldingskøen

Meldingskøen er definert til å bare ta invoc- og comp-meldinger, men det er sannsynligvis enkelt å gjøre endringer slik at instansen av m() kan bli oppbevart der. Vi kan skissere dette alternativet slik:

$$\begin{gathered}
< O : Ob | Pr : SP \& m(IN; OUT), PrQ : W, Lvar : L, Att : A > \\
< O : Qu | Ev : M + (MC, ML) > \\
\rightsquigarrow \\
< O : Ob | Pr : SP ||| m(IN; OUT), PrQ : W, Lvar : L, Att : A > \\
< O : Qu | Ev : M + (MC, ML) > \\
if\ enabled(SP \& MC, ((L, A), ML), M)
\end{gathered}$$

Termen ((L,A),ML) representerer sammenslåingen av tilstandsrommene L, A og ML. A slås sammen med L, og fordi A står sist vil A ha presedens i tilfelle navnekonflikt mellom variabler i de to tilstandsrommene. På samme måte får variabelbindingene i ML presedens over (L,A). Dette er problematisk, men vi bryr oss ikke om det i denne sammenhengen.

5.3.2 Alternativ 2: Oppbevaring i PrQ

Det eksisterer allerede en bindingsmekanisme, og det er ønskelig å gjenbruke den i størst mulig grad. Den eksisterende mekanismen legger alle metodeinstanser i objektets prosesskø (attributtet PrQ). Hvis vi ønsker å hindre at dette skjer i kontekst av synkronisert fletting, må vi lage spesiell kode for det. Derfor er det kanskje mer naturlig å la m() sin instans ligge i PrQ. Vi kan da prøve å få til noe slikt:

$$
\begin{gathered}
< O : Ob | Pr : SP \& m(IN;OUT), PrQ : W : (MC, ML), Lvar : L, Att : A > \\
< O : Qu | Ev : M > \\
\rightsquigarrow \\
< O : Ob | Pr : SP ||| m(IN;OUT), PrQ : W : (MC, ML), Lvar : L, Att : A > \\
< O : Qu | Ev : M > \\
if\ enabled(SP \& MC, ((L, A), ML), M)
\end{gathered}
$$

I PrQ (variabelen W) ligger det vilkårlig mange andre prosesser. Hvordan skal vi kunne finne den riktige prosessen, den som tilhører m() og som skal gis som parameter til enabled()? En løsning er å bytte ut kallet på m() i Pr med en term som kan brukes til å identifisere m() sin instans i PrQ. Det burde være enkelt, siden det allerede eksisterer en lignende mekanisme for å hente prosesser fra PrQ til Pr ved metodekall og kontekstskifter forøvrig som ikke forekommer i kontekst av synkronisert fletting.

Men det er et større problem med dette alternativet. Anta at m() bindes og m() sin instans ligger i PrQ. Anta at &-setningen som m() tilhører viser seg å ikke være beredt. Da suspenderes prosessen. Interpreten har ikke fått spesifisert noen skeduleringspolicy, så den velger å skedulere en vilkårlig, beredt prosess blant de som ligger i PrQ. Det betyr at m() sin instans kan bli skedulert uavhengig av &-setningen. Dette bør ikke skje. Siden kallet på m() forekommer i &-setningen, kan vi ikke tillate at m() sin instans kjøres før &-setningen er beredt.

5.3.3 Alternativ 3: Oppbevaring i en egen prosesskø

Alternativ 3 er et forsøk på å svare på det ovenfornevnte problemet. Hva om vi utvider objektdefinisjonen med et nytt attributt PPrQ (priority program queue), som er dedikert til å oppbevare prosesser som krever spesiell behandling? Vi kan da gjøre om alle leddene i &-setningen til termer av sort Process og plassere dem i det nye attributtet PPrQ.

Vi kan se for oss at i Pr ligger nå en ny setningstype som ikke inneholder noe mer informasjon enn at det skal kjøres en &-setning. Algoritmen for kjøring av &-setningen vil da være å evaluere de vilkårlig mange prosessene i PrQ med enabled(). Hvis alle er beredte, skeduleres prosesser fra PPrQ inntil PPrQ er tom. Deretter kan vi fortsette å skedulere prosesser fra PrQ som vanlig. En mulig fordel er at PPrQ kan brukes til prioritetsskedulering i andre sammenhenger også.

Denne løsningen har flere store problemer. For det først vil det være et stort inngrep i den eksisterende interpreten å legge til et nytt attributt i objektdefinisjonen. For det andre må vi avklare hva som skal skje med prosessene i PrQ hvis &-setningen ikke er beredt og dens prosess derfor suspenderes. For det tredje får vi problemer med alle kontekstskiftene: Gitt en setning $S1\&S2\&S3$, hvor ingen av leddene innledes med et metodekall. Leddene er setningssekvenser i samme prosess og har følgelig alle variabler felles. Dette alternativet medfører at prosessen deles opp i flere separate prosesser. Hvis koden i ett ledd endrer verdien til en variabel, vil ikke endringen reflekteres i tilstanden til de andre leddene, hvilket den burde, siden leddene i utgangspunktet tilhørte samme prosess.

5.3.4 Alternativ 4: Oppbevaring i Pr

Dette er det alternativet jeg har valgt, fordi det virker enklest å realisere.

Her oppbevarer vi metodeinstansen i objektets Pr-attributt. Vi benytter oss av den eksisterende bindingsmekanismen, slik at metodeinstansen legges i PrQ. Det burde være mulig, ved en ren utvidelse

av interpreten, å legge til en regel eller ligning for å flytte instansen fra PrQ til Pr. Samtidig pakker vi instansen inn i en ny setningstype, som for eksempel kan se slik ut: runMtd((MC,ML)). Den er deklarert i figur 5.1 på side 50. Vi kaller dette for en runMtd-setning, og lar den erstatte kallet på m() der hvor det er plassert i &-setningen:

$$
\begin{gathered}
< O : Ob | Pr : SP \& m(IN; OUT), Lvar : L, Att : A > \\
< O : Qu | Ev : M > \\
\rightsquigarrow \\
< O : Ob | Pr : SP \& runMtd(MC, ML), Lvar : L, Att : A > \\
< O : Qu | Ev : M > \\
\rightsquigarrow \\
< O : Ob | Pr : SP ||| runMtd(MC, ML), Lvar : L, Att : A > \\
< O : Qu | Ev : M > \\
if\ enabled(SP\ \&\ runMtd(MC, ML), (L, A), M)
\end{gathered}
$$

Vi kan enkelt gjøre en beredthetssjekk av en slik runMtd-setning. Vi skal straks se at det gjøres ved å utvide enabled() med en ligning.

$$eq\ enabled(runMtd(P, L'), L, M) \quad = enabled(P, (L, L'), M).$$

I ligningen er P koden til en metode, L' metodens lokale variabler, L prosessens tilstand (lokale variabler og objektattributter) og M objektets meldingskø. Merk at i ligningens høyreside slår vi sammen prosessens tilstand med metodens, uten at vi får problemer med navnekonflikter, fordi L' har presedens foran L.

Det er tre klare fordeler med dette alternativet. For det første trenger vi ikke å tenke på hvordan vi skal finne metodeinstansen i PrQ når vi skal gi instansen som parameter til enabled(). Etter at alle relevante kall er bundet kan vi gi hele den ekspanderte &-setningen direkte til enabled().

For det andre, hvis &-setningen ikke er beredt og prosessen suspenderes, slipper vi problemet med hva vi skal gjøre med metodeinstansen. Instansen er en integrert del av koden som suspenderes, i og med at prosessen (MC,ML) nestes inn i &-setningens ledd. De to prosessene suspenderes og skeduleres sammen, uten at vi trenger å endre de delene av interpreten som spesifiserer hvordan dette skjer.

For det tredje unngår vi navnekonflikt mellom variabler. Merk at i alternativ 1 og 2 er den andre parameteren til enabled-funksjonen ((L,A),ML), mens i alternativ 4 er parameteren (L,A). I termen ((L,A),ML) slår vi sammen tilstanden til &-setningens prosess med de lokale variablene til metodeinstansen (ML). Det kan hende at variabler i ML har samme navn som variabler i (A,L). Slik interpreten er spesifisert, tillates bare unike variabelnavn i en tilstand, så i alternativ 1 og 2 risikerer vi å miste/overskrive variabler. Man kan kanskje tillate seg å forutsette at ingen lokale variabler har samme navn som en global variable, men det er helt klart urimelig å forutsette at alle navn på lokale variabler i alle metoder er unike. Vi må derfor skille de ulike prosessenes lokale tilstander, og ikke prøve å slå de sammen, som i alternativ 1 og 2. Alternativ 4 løser dette problemet ved 1) å oppbevare metodeinstansens lokale tilstand inne i &-setningen og 2) ved å utvide enabled() til å vurdere koden i en runMtd-setning i den riktige konteksten.

Hvordan en runMtd-setning kjøres beskrives i avsnitt 5.7.2 på side 77.

5.4 Modning av forståelsen av semantikken til synkronisert fletting

Min første versjon av implementasjonen var mangelfull. Det var forventet. Gjennom analysen av den oppdaget jeg flere egenskaper som & må ha. Vi skal nå se nærmere på dette.

5.4.1 Hovedtrekkene i den første versjonen

Følgende regel brukes i første fase i prosesseringen av &-setningen. LMID er en variabel for metodeidentifikatorer brukt i lokale, synkrone kall.

$rl[\& - bind]$:
$< O : Ob|Pr : (((LMID(IN;OUT)); P)\ \&\ P'); R, Lcnt : N >$
$< O : Qu|Keep : H >$
$=>$
$< O : Ob|Pr : (!LMID(IN));\ (((N?(OUT)); P)\&P'); R, Lcnt : N >$
$< O : Qu|Keep : H; N > .$

Regelen starter bindingen av kall som forekommer fremst i et ledd i &-setningen. Vi ønsker å bruke mest mulig av den eksisterende bindingsmekanismen. Kallsetningen deles opp i en !-setning og en ?-setning. !-Setningen plasseres fremst i Pr. Dermed vil den eksisterende bindingsmekanismen ta seg av alt arbeidet, inntil prosessen til det bundne kallet ligger i PrQ. &-operatoren er deklarert til å være kommutativ, så denne regelen vil bli brukt på alle lokale, synkrone kall som forekommer fremst i et ledd i &-setningen.

Videre lager vi en regel som finner metodeinstansen i PrQ, pakker den inn i en runMtd-setning og plasser denne i &-setningen.

Neste fase består i å vurdere om &-setningen er beredt. Men det kan vi ikke gjøre før vi vet at alle kall som skal bindes har blitt bundet. Hvis ikke annet er spesifisert, antar vi at Maude-interpreten velger å bruke regler i vilkårlig rekkefølge. Vi må derfor sørge for at prosesseringen ikke går til neste fase for tidlig. Derfor gjøres regelen med beredt-testen betinget. Vi definerer en hjelpefunksjon allCallsBound() som tar en &-setning og returnerer true hvis alle kall som skal bindes i kontekst av &-setningen er bundet.

Vi trenger også en hjelpefunksjon mergeify(), som tar en &-setning og returnerer den med alle forekomster av & erstattet med ||| og med innledende await-setninger fjernet fra leddene. Grunnen til at vi ikke kan gjøre dette direkte i en regel, men trenger en egen hjelpefunksjon, er at vi må ta høyde for at det kan være vilkårlig mange ledd i &-setningen.

Vi kan nå lage følgende regel:

$crl[\& - evaluate]$:
$< O : Ob|Pr : (P\&P'); R, Lvar : L, Att : A >$
$< O : Qu|Ev : M >$
$=>$
$< O : Ob|Pr : mergeify(P\&P'); R, Lvar : L, Att : A >$
$< O : Qu|Ev : M >$
$if\ allCallsBound(P\&P')\ and\ enabled((P\&P'), (A, L), M).$

Vi har omformet &-setningen til en |||-setning, som kan kjøres med den eksisterende mekanismen for |||, utvidet med den nye regelen for runMtd-setninger beskrevet tidligere. Vi har altså en implementasjon som tilsynelatende oppfyller definisjonen i 5.2 på side 52.

5.4.2 Lærdommer av analysen av den første versjonen

Ekspansjon av kall i ekspanderte kall

Testing av foregikk ved å kjøre ulike CMC-kodeeksempler. Jeg ble raskt klar over at implementasjonen av ekspansjonen av metodekall ikke tok høyde for at metoder kan kalle metoder. Jeg skal prøve å forklare hva jeg mener med dette. Se på denne setningen: $SP\ \&\ mtd1(IN;OUT)$ Anta at mtd1() sin metodekropp begynner med et kall på en metode mtd2(). mtd2() sin metodekropp kan igjen begynne med et kall på mtd3(), osv.

Som vi husker fra 3.3, er operatoren & ment å brukes til å styrke synkroniseringsbegrensninger ved å danne et aggregat av vakter fra ulike metoder og await-setninger. Synkroniseringsbetingelsene kan være satt sammen av gradvise kall på supermetoder i arvehierarkiet, andre metoder i samme klasse og await-setninger i samme metode. Vi må derfor gå ut ifra at innenfor råderommet (eng: scope) av &-operatoren kan antallet metoder som innledes med kall på andre metoder være vilkårlig stort.

Vi er nødt til å binde samtlige av disse kallene før vi prøver å vurdere om &-setningen er beredt, siden det er konjunksjonen av alle metodenes vakter som avgjør om &-setningen er beredt eller ikke. Vi kan kalle dette for *dyp binding*, ettersom bindingen av ett kall kan medføre en serie av bindinger.[1] Den siste metoden i en slik serie av bindinger vil være en metode som ikke begynner med et internt, synkront

[1]Det samme begrepet brukes visstnok i forbindelse med statisk/leksikalsk scoping i forbindelse med binding av variabler. Det er ingen sammenheng mellom denne betydningen og min ad hoc-definisjon.

metodekall, eller som begynner med en &-setning hvor ingen av leddene begynner med et (ditto) kall. Et unntak fra dette vil være hvis vi får en løkke av bindinger. For eksempel kan en metode mtd1() starte med et kall på mtd2(), som starter med et kall på mtd1().

Når et kall har blitt bundet slik at den resulterende metodeinstansen har blitt pakket inn i en runMtd-setning, må interpreten på en eller annen måte inspisere koden til metodeinstansen for å se om den begynner med et kall som skal bindes.

&-setninger kan være nestede

Jeg oppdaget at versjonen ikke tok høyde for &-setninger som var sammensetninger av &-, |||- og []-setninger. Se på følgende setninger:

$GS1 \vert\vert\vert (GS2 \vert\vert\vert GS3)$	$GS1 \& (GS2 \& GS3)$	$GS1 [] (GS2 [] GS3)$
$GS1 \vert\vert\vert (GS2 \& GS3)$	$GS1 \& (GS2 \vert\vert\vert GS3)$	$GS1 [] (GS2 \vert\vert\vert GS3)$
$GS1 \vert\vert\vert (GS2 [] GS3)$	$GS1 \& (GS2 [] GS3)$	$GS1 [] (GS2 \& GS3)$

Interpreten må kunne håndtere alle disse, vilkårlige kombinasjoner av dem og med vilkårlig mange ledd.

Hvordan skal vi finne et kall som forekommer et sted inne i en vilkårlig dypt nestet term av &-, [], og |||-setninger? Vi kan ikke bruke mønstergjenkjenning direkte på nestede setninger. For eksempel vil vi trenge en regel for hver av de ni setningene ovenfor for å klare å finne et kall som forekommer innenfor parentesene (høyre side).

Splitt og hersk

Innen informatikken er "splitt og hersk" et viktig paradigme innen algoritmedesign. Paradigmet kjennetegnes ved at et komplekst problem deles opp i flere mindre problemer av samme eller lignende type, som igjen deles opp i enda mindre problemer. Dette skjer gjerne rekursivt. Oppdelingen fortsetter inntil problemene blir enkle nok til å kunne løses direkte. Løsningene på subproblemene blir deretter kombinert slik at de til sammen gir en løsning på det opprinnelige, komplekse problemet. Denne teknikken er basisen for effektive algoritmer for mange ulike problemer.

Splitt-og-hersk-teknikken egner seg godt til å løse problemet med komplekst sammensatte &-setninger. Vi kjenner allerede til teknikken fra Maude sitt funksjonelle subspråk. For eksempel har vi sett hvordan enabled() deler opp setningssekvenser i subsekvenser. Er det mulig for oss å bruke en hjelpefunksjon rekursivt på samme måte som enabled() for å lokalisere og binde kall i kontekst av &-operatoren?

Anta at vi har en setning $P \& P'$ hvor P og P' er vilkårlig komplekst sammensatte setninger:

$$< O : Ob | Pr : P \& P', Lvar : L, Att : A >$$

La oss si at vi gir $P \& P'$ sammen med objektets tilstand (L, A) som parametre til en innbilt hjelpefunksjon bind(). Denne vil enkelt kunne dele opp &-setningen i sine ledd, som igjen kan deles opp på samme måte hvis det skulle være nødvendig. La oss si at bunnen av rekursjonen (base case) er en setningssekvens som innledes med et metodekall som vi trenger å binde. Hvordan kan vi binde dette kallet? Vi ønsker å benytte oss av den eksisterende bindingsmekanismen, og er derfor avhengige av å få lagt en invoc- eller bindMtd-melding i konfigurasjonen:

$$bind(LMID(IN; OUT); P, L) \rightsquigarrow invoc(\ldots)$$

Kan vi lage en regel som plasserer en bind-term i konfigurasjonen utenfor objektet?:

$$< O : Ob | Pr : P \& P', Lvar : L.Att : A > \\ bind(P \& P', (L, A))$$

Vi forestiller oss at bind-termen reduseres til en invoc-melding. Invoc-meldinger som ligger i konfigurasjonen, blir plukket opp av de eksisterende reglene for håndtering av kallmeldinger. Problemet med denne løsningen er mottaket av den resulterende metodeinstansen. Hvordan skal vi få plassert en metodeinstans på riktig sted i den opprinnelige &-setningen? I motsetning til kall som ikke forekommer i kontekst av &, har vi her ikke muligheten til å bruke merkelappverdien i objektets Lcnt-attributt.

I neste seksjon løser jeg dette problemet ved å bruke splitt og hersk-taktikken på en annen måte. Denne måten løser også problemet med metodekall som forekommer i koden til ekspanderte metodekall.

5.5 Ekspansjon av kall i kontekst av synkronisert fletting

I denne seksjonen gjør jeg rede for hvordan jeg har valgt å binde metodekall i kontekst av &. Vi ser detaljert på Maude-koden som lar oss bygge opp en &-setning med ekspanderte kall.

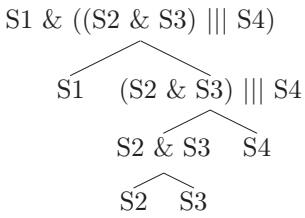
Jeg har tidligere gjort det klart at vi må ta høyde for at en &-setning kan være vilkårlig komplekst sammensatt av &-, []- og |||-setninger. Dermed er den første utfordringen å finne kallene. Videre har jeg gjort rede for valget om at kallenes metodeinstanser skal plasseres i runMtd-setninger i leddene i &-setningen. Når dette er gjort, er kallene i &-setningen ekspanderte.

Jeg minner om at figuren med variabeldeklarasjoner finnes på side 42 og at figuren med operatordeklarasjoner finnes på side 50.

5.5.1 Hvordan finne kall som skal ekspanderes?

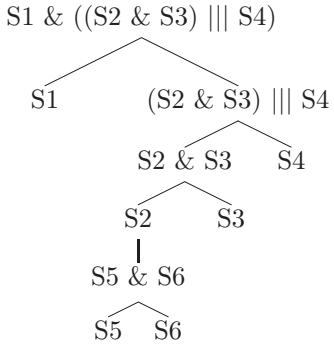
La oss si at vi har en setning S1 & ((S2 & S3) ||| S4), hvor leddene S1, S2,... er setningssekvenser som ikke innledes med en &-, []- eller |||-setning. Vi er interesserte i å binde eventuelle innledende, synkrone og interne metodekall i alle disse leddene unntatt S4. Grunnen til at vi ikke ønsker å binde i S4 er at |||-operatoren i motsetning til & og (min versjon av) [] ikke ekspanderer metodekall. Vi bruker splitt og hersk-teknikken for å isolere hvert ledd. Det vil gi oss muligheten til å se på den første setningen i hvert ledd.

Vi kan visualisere ideen bak mekanismen ved å tegne et trediagram. Vi starter med en &-setning. Vi lager et tre av delsekvenser. Grenene vokser dybde først, inntil vi har isolert et ledd som ikke begynner med en &-, []- eller |||-setning.



Maude-interpreten forsøker å bruke alle ligninger mellom hvert forsøk på å bruke en regel. Dette har jeg benyttet meg av til å basere mekanismen på en spesiell arbeidsfordeling mellom ligninger og regler. Grunnen til at treet ekspanderer i en dybde-først-rekkefølge er at jeg bruker ligninger til å ekspandere med og regler til å folde med. Med å "folde" mener jeg å neste koden i en node inn i koden i foreldrenoden. Ligninger ekspanderer en gren inntil vi når løvnoden dens. En folderegel reduserer dybden/lengden på en gren med ett nivå. Hvis det er mulig, vil en ny gren vokse etter hver folding. Dette forklares nærmere i 5.5.3.

Poenget med å isolere leddene er å finne kall som skal bindes. La oss si at S2 innledes med et metodekall. Vi må da binde denne metoden. Metodeinstansen plasseres i treet som en forlengelse av grenen. Anta at koden i denne metodeinstansen innledes med en &-setning. Da fortsetter ekspansjonen:



Vi ser at en node i treet kan være to ulike typer termer: Enten har vi et ledd fra en &-, []- eller |||-setning eller så har vi en metodeinstans. I min utvidelse av interpreten har jeg innført to operatorer som begge brukes til å oppbevare begge disse termtypene:

$sort\ ProgElement. \quad subsort\ ProgElement\ <\ Prog.$
$op\ element\ :\ Process\ -\!>\ ProgElement\ [ctor].$
$op\ awaitingElement\ :\ Process\ -\!>\ ProgElement\ [ctor].$
$op\ insert\ :\ Nat\ -\!>\ Stm\ [ctor].$

En element-term tilsvarer det som på ethvert tidspunkt er løvnoden i treet. En foreldrenode representeres med en awaitingElement-term. Navnet gjenspeiler at dette representerer en kodesekvens som vi på et senere tidspunkt skal neste en kodesekvens inn i. Vi skal straks se hvordan disse operatorene brukes. Mye av dette kapitlet handler nettopp om hvordan bruken av dem løser problemene med ekspansjon av metodekall. Insert-setninger brukes til å angi hvor i en awaitingElement-setning et ledd eller en metodeinstans skal settes inn. Tallet som insert-setningen tar som parameter er merkelappverdien til den tilhørende metodeinstansen. Hvis tallet er 0, skal insert-setningen erstattes av et setningsledd og ikke en metodeinstans.

Hvordan isolere et ledd i en sammensatt setning?

Før vi går videre, ser vi på en liten feil i den varianten av interpreten som ble gitt meg. Måten jeg fant feilen på er et eksempel på en fordel med den typen maskinell analyse som Maude tilbyr. Dette er definisjonen av ||| som den var før jeg endret dem:

$op\ \_|||\_\ :\ ProgList\ ProgList\ -\!>\ Prog\ [ctor\ assoc].$

$crl[merge-first]:$
$<O:Ob|Pr:((SP;P)|||P');R,Lvar:L,Att:A>$
$<O:Qu|Ev:M>$
$=>$
$<O:Qu|Ev:M>$
$<O:Ob|Pr:SP;(P|||P');R,Lvar:L,Att:A>$
$if\ ready(SP,(L,A),M).$

$crl[merge-second]:$
$<O:Ob|Pr:(P|||P');R,Lvar:L,Att:A>$
$<O:Qu|Ev:M>$
$=>$
$<O:Qu|Ev:M>$
$<O:Ob|Pr:(P'|||P);R,Lvar:L,Att:A>$
$if\ ready(P',(L,A),M)\ and\ not\ ready(P,(L,A),M).$

$eq\ (empty|||P)\ =\ P.$

Denne varianten av |||-operatoren er ikke kommutativ, så leddene i en |||-setning er ordnet i en listerekkefølge. Ideen bak dette er at hvis den fremste setningen i det fremste leddet er klar, så kjøres den. Hvis setningen ikke er klar, plasseres setningens ledd bakerst i listen. Dette gjentas for de andre leddene.

Feilen gir seg utslag i følgende uventede omskriving av en |||-setning med tre ledd:

$$S1\ |||\ S2\ |||\ S3 \rightsquigarrow S1\ |||\ S2\ ;\ S3$$

Det er regelen "merge-first" i kombinasjon med ligningen ovenfor som forårsaker dette. Vi forventer at variabelen SP skal bindes til det fremste leddet i |||-setningen, men siden SP er av sort Prog, kan den likegodt bindes til en term som inneholder flere ledd. Maude-interpreten velger å binde SP til (S1 ||| S2), P til empty og P' til S3. Det som da skjer er

$$S1\ |||\ S2\ |||\ S3$$
$$\rightsquigarrow_{merge-first}$$
$$(S1\ |||\ S2);(empty|||\ S3)$$
$$\rightsquigarrow_{ligning}$$
$$(S1\ |||\ S2);S3$$

Denne feilen ble oppdaget ved en tilfeldighet da jeg analyserte et CMC-program ved å kjøre programmet ett omskrivingssteg om gangen og å studere objektets tilstand manuelt etter hvert omskrivingssteg. Dette gjøres ved å gi Maude kommandoene "rew [10] init .", "rew [11] init .", "rew [12] init .", og så videre, hvor init er en term som reduseres til en konfigurasjon. Simulering har den fordelen framfor manuell analyse at maskinen ikke er forutinntatt. Grunnen til at jeg ikke oppdaget denne feilen ved lesing av koden, var at jeg leste et innhold inn i regelen "merge-first" som passet med forventningene mine. Denne feilen var i tillegg vanskelig å oppdage fordi den har små konsekvenser for adferden til programmet — den kan i mange tilfeller ikke oppdages ved å søke etter alle sluttilstander med search-kommandoen.

For å unngå slike feil, trenger vi en måte å isolere ledd i &-, |||- og [] setninger på. Jeg har vurdert tre alternativer, og som vi skal se etter hvert, har jeg brukt både alternativ 2 og 3 ved forskjellige anledninger.

1) Isolering med hjelpefunksjoner I tråd med typisk funksjonell programmering, kan vi lage to hjelpefunksjoner som vi kan kalle for eksempel getFirst() og removeFirst(). De tar begge en |||-setning som parameter og returnerer henholdsvis den fremste setningen i det fremste leddet og |||-setningen hvor den fremste setningen i det fremste leddet er fjernet. Med en ny regel r1 som appliserer disse funksjonene på |||-setningen, kan vi få følgende omskrivingssekvens:

$$\begin{gathered}(SP;P)|||P'|||P'' \\ \rightsquigarrow_{r1} \\ getFirst((SP;P)|||P'|||P'');removeFirst((SP;P)|||P'|||P'') \\ if\ ready(getFirst((SP;P)|||P'|||P''),\ldots) \\ \rightsquigarrow_{ligninger} \\ SP;(P|||P'|||P'')\end{gathered}$$

2) Isolering med sortsammenligning Vi kan bruke Maude sin operator :: for sammenligning av sorter. Interpreten som ble gitt til meg hadde en variant av |||-operatoren som var deklarert til å gi opphav til termer av sort Prog. I min deklarasjon (figur 5.1 på side 50) gir den opphav til termer av en ny sort Merge. Dermed vet vi at en term av sort Merge er en |||-setning med minst to ledd. Sorten Merge er subsort av Prog. Vi har nå muligheten til å bruke en tenkt regel regel r2 på en del av |||-setningen, og sette som betingelse at delen ikke skal inneholde mer enn ett ledd:

$$(SP;P)|||P' \rightsquigarrow_{r2} SP;(P|||P')\ if\ not\ SP :: Merge$$

3) Isolering med run-setningen Av grunner som forklares senere, har jeg endret interpreten slik at det første som skjer når en |||-setning kjøres, er at alle leddene pakkes inn i en run-setning. run-setningen er beskrevet i avsnitt 5.5.1 på neste side og 5.5.5 på side 68. Denne fasen av kjøringen representeres nedenfor av den tenkte regelen r3. Vi kan da enkelt isolere og gripe det fremste leddet med mønstergjenkjenning i en tenkt regel r4:

$$(SP;P)|||P'|||P'' \rightsquigarrow_{r3} run(SP;P)|||run(P')|||run(P'')$$

$$\begin{gathered}run(SP;P)|||P' \rightsquigarrow_{r4} SP;(run(P)|||P') \\ if\ ready(SP,\ldots)\end{gathered}$$

Ekstraksjon av ledd fra roten i treet

Vi fortsetter med beskrivelsen av min utvidelse av interpreten. Det første som skjer i interpreten når Pr møter en &- eller []-setning er at den relevante av de følgende to ligningene blir brukt. Ligningene er like med unntak av operatoren, så jeg beskriver bare den ene.

$ceq\ < O : Ob|Pr : (P\&P'); R, Lvar : L >$
$=$
$< O : Ob|Pr : element((P, L)); (insert(0)\&P'); R, Lvar : L >$
$if\ not\ (P :: Run\ or\ P :: SyncMerge).$

$ceq\ < O : Ob|Pr : (P[]P'); R, Lvar : L >$
$=$
$< O : Ob|Cl : CV, Pr : element((P, L)); (insert(0)[]P'); R, Lvar : L >$
$if\ not\ (P :: Run\ or\ P :: Nondet).$

Vi ser at leddet P blir tatt ut av &-setningen og erstattet med en term insert(0). P blir pakket inn i en element-setning sammen med en kopi av de lokale variablene. Element-setningen plasseres fremst i Pr for at den skal kunne bli behandlet i neste skritt. Etter at element-setningen en gang i framtiden er ferdigbehandlet, skal koden som den inneholder plasseres tilbake der hvor den stod. Insert-setningen forteller oss hvor det er. Tallet 0 i insert-setningen brukes som standardverdi når insert-setningen markerer innsettingsstedet til et setningsleddledd, i motsetning til når den markerer en metodeinstans.

Vi ser på betingelsen i ligningen, hvor vi bruker Maude sin operator :: for sortsammenligning. Betingelsen gjør at vi vet at når ligningen blir brukt, så er P et atomisk ledd, det vil si at P ikke er en &-setning bestående av flere ledd. Hvis P hadde vært av sorten SyncMerge, ville P vært en &-setning med minst to ledd (se subsortdeklarasjonen i figur 5.1 på side 50). Da ville betingelsen hindret at ligningen ble brukt.

Betingelsen hindrer også at ligningen blir brukt på P hvis P er av sort Run. Sorten Run brukes for å unngå uønsket ikke-terminering. Et ledd som har blitt ekstrahert, behandlet av andre regler/ligninger og så satt på plass igjen, settes på plass innpakket i en run-setning. Dette gjøres for å hindre at ligningen vist ovenfor brukes om og om igjen på det samme leddet. Hvis et ledd er av sort Run, vil ligningen bli forsøkt applisert på et annet ledd i setningen. Hvis alle leddene er av sort Run, vet vi at alle kall som skal bindes har blitt bundet. Ligningen vil bli brukt på alle leddene i &-setningen fordi &-operatoren er kommutativ.

Ligningen for |||-setninger er lik de to ovenforstående, med unntak av betingelsen, som har to nivåer:

$ceq\ < O : Ob|Pr : ((SP; P)|||P'); R, Lvar : L >$
$=$
$if\ SP :: SyncMerge\ or\ SP :: Nondet$
$then\ < O : Ob|Pr : element(((SP; P), L)); (insert(0)|||P'); R, Lvar : L >$
$else\ < O : Ob|Pr : (run(SP; P)|||P'); R, Lvar : L >$
fi
$if\ not\ ((SP; P) :: Run\ or\ (SP; P) :: Merge).$

Som i de to andre ligningene, sikrer betingelsen at ligningen blir brukt på ett ledd, og at dette leddet ikke allerede har blitt behandlet. Videre sier if-testen i ligningen at hvis leddets første setning er en &- eller []-setning, så ekstraherer vi leddet og pakker det inn i en element-setning. Hvis ikke, lar vi leddet forbli i |||-setningen, pakket inn i en run-setning. Det markerer at leddet er ferdig behandlet. Vi behandler ledd fra |||-setninger på denne måten fordi vi ikke skal ekspandere kall i kontekst av |||.

Bruken av den første ligningen på vår setning S1 & ((S2 & S3) ||| S4) gir i første omgang en av følgende to mulig omskrivinger.[2] L er prosessens lokale variabler.

$$S1\&((S2\&S3)|||S4) \rightsquigarrow$$
$$element((S1, L)); (insert(0)\&((S2\&S3)|||S4))$$
eller
$$element(((S2\&S3)|||S4, L)); (S1\&insert(0))$$

[2]Valget mellom flere mulige omskrivinger er avhengig av omskrivingsstrategien. Maude-interpreten har en standard, deterministisk omskrivingsstrategi, men vi forholder oss til valget som om det var ikke-deterministisk. Det er mulig for brukeren å designe sin egen omskrivingsstrategi, ved å benytte Maude sin mulighet for metaprogrammering.

Ulike tilstandsrom i element-setninger

Hvorfor trenger vi tilstand i element-setninger? Element-operatoren tar en term av sort Process som parameter, og en Process-term består av programkode og variabelbindinger. Som vi ser av ligningene ovenfor, er det den lokale tilstanden som inkluderes i element-setningen. Det er unødvendig å ta vare på den lokale tilstanden i element-setninger som inneholder kode fra den opprinnelige &-setningen. Vi kunne istedenfor ha klart oss med en element-setning som bare tok kode, og brukt variablene i objektattributtet Lvar ved binding.

Vi må huske at det ikke er noen forskjell på kjørende metoder og prosesser i Creol. Prosessene i et objekt har ulike men overlappende tilstandsrom: Alle prosesser har tilgang til objektattributtene, og alle prosesser har en mengde lokale variabler, som bare prosessen har tilgang til. En metode sine formelle parametre er en delmengde av metodens/prosessens lokale variabler.

Vi trenger lokal tilstand når vi skal binde kall som forekommer i koden til ekspanderte kall (se avsnitt 5.4.2 på side 56). Når vi binder kall i en &-setning, blir alle de ulike metodeinstansene liggende i Pr i element- eller awaitingElement-setninger. De lokale variablene i Lvar gjelder ikke for metodeinstansene, bare for &-setningens prosess. Siden hvert kall har sine reelle parametre, og parametrene evalueres i kontekst av hver metode sin tilstand, må hver metode sin tilstand taes vare på, separat fra Lvar.

Ekstraksjon av ledd fra grener i treet

Vi trenger egne ligninger for å ekstrahere ledd fra setninger som er pakket inn i element-setninger. Konseptet er det samme, og betingelsene fungerer på samme måte. Siden den lokale tilstanden er inneholdt i element-setningen, trenger vi ikke tilgang til objektets Lvar-attributt og kan derfor la være å se på objektet i disse ligningene, i motsetning til ovenfor.

$$
\begin{array}{l}
ceq\ element(((P \& P'); R, L)) \\
= \\
element((P, L)); awaitingElement(((insert(0) \& P'); R, L)) \\
if\ not\ (P :: SyncMerge\ or\ P :: Run).
\end{array}
$$

$$
\begin{array}{l}
ceq\ element(((P [] P'); R, L)) \\
= \\
element((P, L)); awaitingElement(((insert(0) [] P'); R, L)) \\
if\ not\ (P :: SyncMerge\ or\ P :: Run).
\end{array}
$$

$$
\begin{array}{l}
ceq\ element((((SP; P) ||| P'); R, L)) \\
= \\
if\ SP :: SyncMerge\ or\ SP :: Nondet \\
then\ element(((SP; P), L)); awaitingElement(((insert(0) ||| P'); R, L)) \\
else\ element(((run(SP; P) ||| P'); R, L)) \\
fi \\
if\ not\ ((SP; P) :: Merge\ or\ (SP; P) :: Run).
\end{array}
$$

Vi ser her hvordan awaitingElement-setningen brukes. Den inneholder en setningssekvens og en tilstand på samme måte som element-setningen. Ledd fra koden i awitingElement-setningen ekstraheres og erstattes av en insert-setning på samme måte som før.

Ved bruk av de ligningene vi har til nå, vil vi kunne få følgende omskrivingssekvens:

$$
\begin{array}{c}
S1 \& ((S2 \& S3) ||| S4) \\
\rightsquigarrow \\
element(((S2 \& S3) ||| S4, \ldots)); (S1 \& insert(0)) \\
\rightsquigarrow \\
element(((S2 \& S3,) ||| run(S4), \ldots)); (S1 \& insert(0)) \\
\rightsquigarrow \\
element((S2 \& S3, \ldots); awaitingElement((insert(0) ||| run(S4), \ldots)); \\
(S1 \& insert(0)) \\
\rightsquigarrow \\
element((S2, \ldots)); awaitingElement((insert(0) \& S3, \ldots); \\
awaitingElement((insert(0) ||| run(S4), \ldots)); (S1 \& insert(0))
\end{array}
$$

Element-setningen lengst til venstre i den nederste termen representerer løvnoden S2 i trediagrammet på side 5.5.1 på side 58.

Hvorfor er det nødvendig med awaitingElement-setningen? Hvorfor kan vi ikke bruke to element-setninger istedenfor? Vi trenger å bevare en fast rekkefølge på de ekstraherte leddene. Uten å skille de to setningene fra hverandre, kan vi få følgende omskrivingssekvens:

$$element((S2\&S3,\ldots)$$
$$\rightsquigarrow$$
$$element((S2,\ldots)); element((insert(0)\&S3,\ldots)$$
$$\rightsquigarrow$$
$$element((S2,\ldots)); element((S3,\ldots)); element((insert(0)\&insert(0),\ldots)$$

Dette blir fort kaotisk: Anta at S3 innledes med en &-setning, og at leddene i denne setningen ekstraheres og plasseres til venstre for S3 sin element-setning og til høyre for S2 sin element-setning. Da blir det vanskelig å vite hvilken element-setning S2 skal plasseres i etter at S2 er ferdigbehandlet. Vi får også problemer med bindingen av kall. Kan vi binde et kall i S3 ovenfor når S2 er plassert foran (det vil si til venstre for) S3 i Pr? Hva gjør vi hvis S2 ikke har et kall som skal bindes, men S3 har det? Dette problemet lar seg løse ved at vi inkluderer objektet i ligningene:

$$< O : Ob | Pr : element((P\&P',\ldots); R >$$
$$\rightsquigarrow$$
$$< O : Ob | Pr : element((P,\ldots)); element((insert(0)\&P',\ldots); R >$$

En slik ligning vil alltid bare ekstrahere ledd fra den fremste element-setningen i Pr. Dermed får vi en fast rekkefølge og en dybde-først-ekspandering av trediagrammet.

Vi har altså to alternativer 1) Ligninger hvor objektet inngår og hvor vi bruker én setningstype og 2) ligninger uten objektet, men med to setningstyper. Jeg implementerte først alternativ 1), men det endelige valget mitt falt på 2). På den ene siden ønsker vi færrest mulig operatorer og setningstyper, men på den andre siden ønsker vi å begrense mengden kode ved å ikke inkludere objektet i ligningene/reglene, slik at interpreten skal være så lett som mulig å lese. Jeg har lagt mest vekt på det siste.

5.5.2 Hvordan binde et kall når det er funnet?

Ligningene for ekspansjon som er presentert tidligere vil brukes inntil vi når et ledd som ikke innledes med en &-, []- eller |||-setning. Hvis dette leddet innledes med et internt, synkront metodekall, må kallet bindes. Vi må se nærmere på hvordan vi avgjør om kallet skal bindes og hvordan bindingen skal foregå når avgjørelsen først er tatt.

Hvordan vite om kallet skal bindes?

Som sagt er det bare interne, synkrone kall som skal bindes. Vi husker fra figur 3.2 på side 22 at interne, synkrone kall har tre mulige former: Virtuelt kall uten objektidentifikator[3], kall med objektidentifikator hvor objektidentifikatoren evaluerer til det samme som *this* og statiske kall. Vi identifiserer lett synkrone kall ved at både inn- og utparametre forekommer i kallet. Men vi trenger en måte å avgjøre om et synkront kall er internt på. Jeg har vurdert tre alternativer.

Alternativ 1: Hjelpefunksjon Vi kan spesifisere en boolsk hjelpefunksjon isInternalCall() som svarer på spørsmålet for oss:

$$\begin{array}{ll} op\ isInternalCall & :\ Mid\ Cid\ Oid\ Subst- > Bool. \\ eq\ isInternalCall(OE.Q, C, O, L) & =\ if\ eval(OE, L) == O\ then\ true \\ & \quad else\ false\ fi. \\ eq\ isInternalCall(Q@C, C, O, L) & = true. \\ eq\ isInternalCall(Q, C, O, L) & = true. \\ eq\ isInternalCall(MM, C, O, L) & = false\ [owise]. \end{array}$$

[3]Ved finlesing av interpreten kan det se ut som om dette tilfellet ikke forekommer i CMC, ettersom det eksisterer en ligning $eq\ !Q(I) = !'this.Q(I)$. Men denne ligningen har ikke rukket å bli brukt på det tidspunktet hvor vi har behov for å binde.

Den første ligningen evaluerer objektidentifikatoren i kallet for å se om det kallede objektet er lik det kalte objektet. Den andre ligningen sier at kallet er internt fordi det er statisk. Den tredje sier at kallet er internt fordi det er lokalt, det vil si at det ikke forekommer noen objektidentifikator. Den fjerde ligningen sier at hvis vi ikke har noen av de ovenfornevnte tilfellene, så er ikke kallet internt.

Vi kan dermed enkelt lage en ligning/regel som binder kallet på den betingelsen at kallet er synkront (mønstergjenkjenning) og internt (betingelsen på slutten av regelen):

$$\begin{gathered} ceq \ < O : ob | Pr : element((MM(IN;OUT);P,L));R > \\ \rightsquigarrow \\ < O : ob | Pr : element(\ldots);R > \quad bindMTd(O,MM,\ldots) \\ if\ isInternalCall(MM,\ldots). \end{gathered}$$

Alternativ 2: Sjekke termens sort med :: Jeg vurderte om det var mulig å bruke Maude sin operator for sortsammenligning :: til å avgjøre om et kall er internt. Termer som representerer statiske kall kan for eksempel få sorten StaticCall, og termer som representerer virtuelle kall uten metodeidentifikator kan for eksempel få sorten VirtualCall. Vi kan deklarerer følgende sorthierarki:

$sorts\ StaticCall\ VirtualCall\ InternalCall\ Call.$
$subsorts\ StaticCall\ VirtualCall < InternalCall < Call < Stm.$

Gitt et metodekall $MM(IN;OUT)$ av ukjent type, kan vi lage en betinget ligning/regel

$$\begin{gathered} < O : Ob | Pr : element((MM(IN;OUT);P,\ldots));R > \\ \rightsquigarrow \\ < O : Ob | Pr : element((\ldots;P));R > \quad bindMTd(O,MM,\ldots) \\ if\ MM(IN;OUT) :: InternalCall. \end{gathered}$$

Problemet med denne måten å løse problemet på er virtuelle kall med metodeidentifikator. Her er det ikke typen eller sorten til kallet som avgjør om det er internt, men forholdet mellom kalleren og den kallede: De må være like for at kallet skal være internt. Dette kan kanskje løses ved å lage en ligning som fjerner en objektpeker fra metodekallet hvis objektpekeren evaluerer til det samme som $this$.

Alternativ 3: Mønstergjenkjenning I avsnittet om hjelpefunksjonen isInternalCall() identifiserte vi mønstrene og betingelsene som avgjør om et kall er internt. Vi kan bruke den samme innsikten til å lage en regel for hvert av de tre tilfellene, uten å trenge å lage noen ny hjelpefunksjon eller å deklarere nye sorter. Vi kommer tilbake til detaljene i denne løsningen.[4]

$***$ **kall med Oid**
$ceq < O : Ob | Cl : C\#V, Pr : element((((OE.Q(IN;OUT));P),L'));R,$
$\quad Lvar : L, Att : A, Lcnt : N >$
$=$
$< O : Ob | Cl : C\#V, Pr : awaitingElement(((insert(N);(N?(OUT));P),L'));R,$
$\quad Lvar : L, Att : A, Lcnt : N+1 >$
$bindMtd(O,Q,(O\ int(N)\ evalList(IN,(A,L'))),C\#V[nil])$
$if\ eval(OE,(A,L)) == O.$

$***$ **lokalt virtuelt kall uten Oid**
$eq < O : Ob | Cl : C\#V, Pr : element((((Q(IN;OUT));P),L'));R,$
$\quad Att : A, Lcnt : N >$
$=$
$< O : Ob | Cl : C\#V, Pr : awaitingElement(((insert(N);(N?(OUT));P),L'));R,$
$\quad Att : A, Lcnt : N+1 >$
$bindMtd(O,Q,(O\ int(N)\ evalList(IN,(A,L'))),C\#V[nil]).$

[4]Bruken av bindMtd-meldinger forklares i avsnitt 5.5.3 på neste side og ?-setningen forklares i avsnitt 5.7.2 på side 77.

$***$ **statisk kall**
$eq < O : Ob|Pr : element((((Q@C(IN; OUT)); P), L')); R,$
$\quad Att : A, Lcnt : N >$
$=$
$< O : Ob|Pr : awaitingElement(((insert(N); (N?(OUT)); P), L')); R,$
$\quad Att : A, Lcnt : N + 1 >$
$bindMtd(O, Q, (O\ int(N)\ evalList(IN, (A, L'))), C\#0[nil]).$

Konklusjon Vi har tre alternativer som alle er tilsynelatende gjennomførbare. Ulempen med alternativ 1 er at vi må utvide interpreten med en ny operator. I alternativ 2 må vi utvide med flere nye sorter. I alternativ 3 trenger vi ingen nye operatorer eller sorter, men utvider interpreten med tre nye regler. Jeg har implementert alternativ 3, fordi jeg tenkte at jo færre operatorer og sorter interpreten har, jo lettere blir den å lese og forstå. I ettertid har jeg tenkt at det er riktigere å legge mest vekt på at beskrivelsen av semantikken til Creol består av færrest mulig regler. Dermed framstår alternativ 1 som det beste, siden det er det letteste å lese.

5.5.3 Binding uten bruk av Maude-regler

Som nevnt i avsnitt 5.5.1 på side 58, har jeg basert bindingsmekanismen i kontekst av synkronisert fletting på at ekspansjonen av metodekall i &-setningen skjer utelukkende med ligninger. Dette gir en elegant arbeidsfordeling mellom ligninger og regler i mekanismen. Ideen trengte seg på da det viste seg å være vanskelig å definere tilstander som indikerer at mekanismen er i ekspansjonsfasen og tilstander som indikerer at vi er i foldingsfasen.

Vi ønsker å ekspandere i to tilfeller:

- Vi har en setningssekvens eller et ledd hvor den første setningen er en &-, []- eller |||-setning og hvor det i denne setningen finnes et ledd som ikke er av sort Run (se side 61).
- Vi har et ledd hvor den første setningen er et internt, synkront metodekall.

I alle andre tilfeller ønsker vi å folde. Siden vi må anta at Maude-interpreten anvender regler i tilfeldig rekkefølge, er det vanskelig å lage en mengde regler hvor vi unngår å folde før vi har ekspandert ferdig. Vi vet derimot at Maude-interpreten alltid bruker ligninger før regler. Dette bruker vi til vår fordel, ved å implementere ekspansjon utelukkende med ligninger og folding utelukkende med regler. Selv om en ligning matcher samme tilstand som en regel, er vi aldri i tvil om hvilken av dem som blir brukt.

Det er mulig å se på applikasjon av en regel som en modellering av at det tar litt tid å prosessere en programsetning. Fra denne synsvinkelen er det rimelig å bruke ligninger istedenfor regler, ettersom interne kall vil kreve kortere tid og ingen kommunikasjon i et virkelig system.

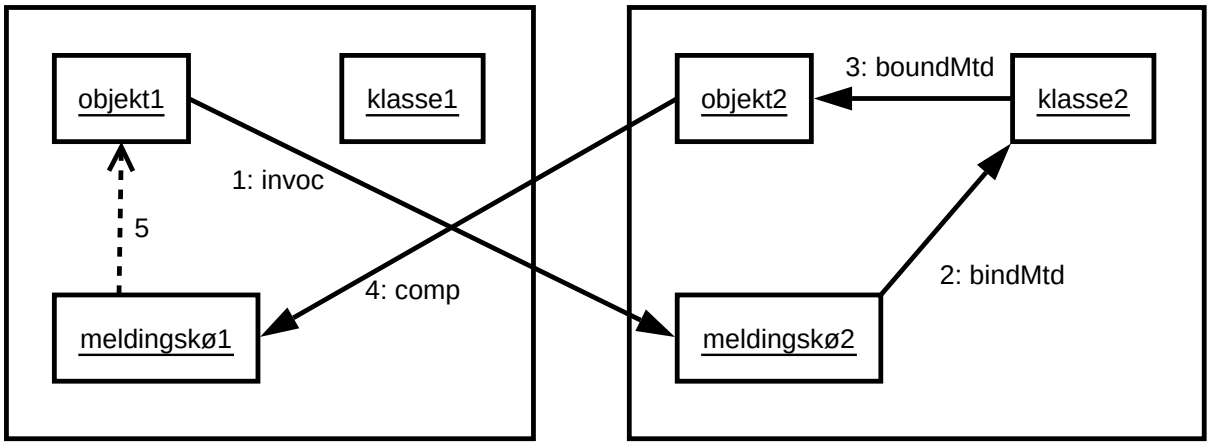
Dette krever at vi snor oss rundt det faktum at den eksisterende bindingsmekanismen benytter seg av et antall regler. Vi kan lage en egen mekanisme til vårt formål, men det er ønskelig å gjenbruke mest mulig kode. Vi ser på i hvilken grad det er mulig å gjenbruke den eksisterende løsningen.

Betrakt figur 5.3 på side 67, subfigur (a) og (b). I den eksisterende bindingsmekanismen gir !-setningen i et kall opphav til en invoc-melding. En invoc-melding er en forespørsel om en spesifikk metodeinstans. Transporten av denne meldingen fram til meldingskøen til det objektet som tilbyr metoden, håndteres av regelen "invoc-msg" (linje 893 i A). Videre omformes en ankommet invoc-melding til en bindMtd-melding med en av de to reglene som begge heter "recieve-call-req" (linje 814 og 821). Resten av bindingsprosessen, inntil metodeinstansen ligger i det kallende objektet sitt attributt PrQ, skjer med ligninger.

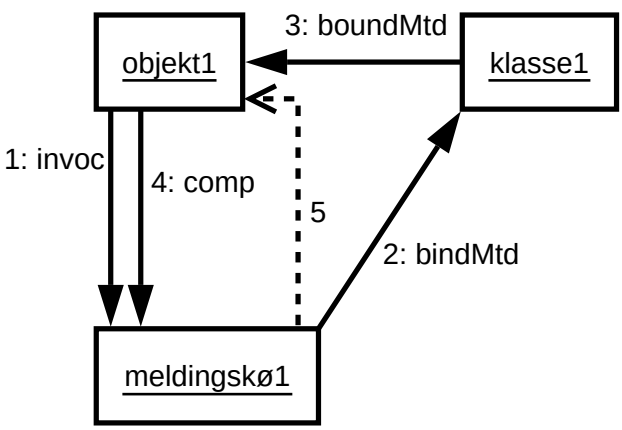
Vi skal nå se på de to alternativene jeg har vurdert for å unngå bruk av regler. Jeg har implementert begge, og invoc-løsningen er inkludert i vedlegg B som kommentarer i koden.

bindMtd-løsningen

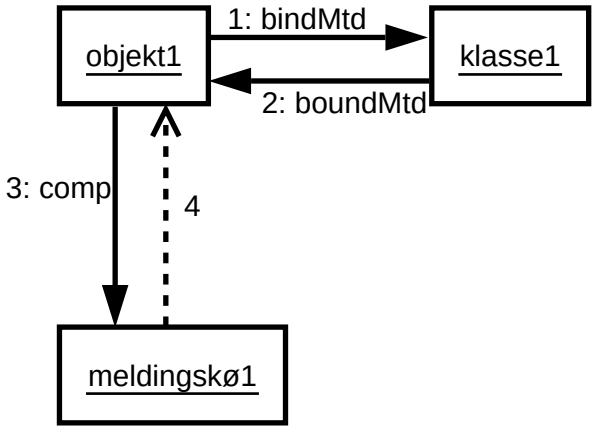
Maude-koden for dette alternativet har vi sett i avsnitt 5.5.2 på forrige side. Tidlig i designprosessen begrenset vi oss til å ekspandere bare synkrone metodekall som er interne. Det betyr at til vårt formål har vi strengt tatt ikke behov for en mekanisme for transport av meldinger mellom objekter. Det er ikke urimelig å spørre seg om vi ikke kan omgå hele meldingskøen. Når vi skal binde kall i ekspansjonsfasen av en &-setning, kan vi hoppe over genereringen av invoc-meldingen og gå rett fra en !-setning til en bindMtd-melding. Dette er illustrert i figur 5.3 på neste side, subfigur (c).



(a) Eksternt metodekall



(b) Internt metodekall



(c) Internt metodekall i &-setninger.

Figur 5.3: Figur (a) viser informasjonsutvekslingen ved et virtuelt metodekall som ikke forekommer i kontekst av &. Alle kall håndteres likt med hensyn på meldingsutveksling, så figur (b) er lik figur (a) hvor det kalte objektet er lik det kallende. invoc-meldingen transporteres med bruk av en regel, og bindMtd-meldingen genereres med bruk av en regel. I figur (c) ser vi at vi unngår bruk av disse reglene når vi hopper over invoc-meldingen. Hvis kallet er statisk, må vi hente informasjon fra objektet for å finne ut hvilken klasse metodedefinisjonen finnes i. Dette gjøres i så fall i trinn 2 (a) og (b) og trinn 1 i (c). I trinn 5 i (a) og (b) og trinn 4 i (c) (stiplet pil) overføres returverdiene fra kallet direkte ved bruk av en regel i motsetning til en melding.

invoc-løsningen

Det er mulig å beholde invoc-meldingen med tilhørende hendelsesforløp og meldinger og samtidig unngå bruk av regler. Vi kan definere ligninger som håndterer invoc-meldinger når kallet er internt.

Følgende tre ligninger behøves for å få lagt invoc-meldingen i konfigurasjonen. De brukes istedenfor de tre ligningene på side 64. Merk at istedenfor å legge en bindMtd-melding i konfigurasjonen, så legges en !-setning fremst i Pr. Vi overlater til den eksisterende bindingsmekanismen å generere en invoc-melding på grunnlag av !-setningen.

$***$ **kall med Oid**
$ceq\ < O : Ob|Pr : element((((OE.Q(IN;OUT));P),L'));R, Lvar :$
$\quad L, Att : A, Lcnt : N >$
$=$
$< O : Ob|Pr : (!OE.Q(IN)); awaitingElement(((insert(N);(N?(OUT));P),L'));R,$
$\quad Lvar : L, Att : A, Lcnt : N >$
$if\ eval(OE,(A,L)) == O.$

$***$ **lokalt virtuelt kall uten Oid**
$eq\ < O : Ob|Pr : element((((Q(IN;OUT));P),L'));R, Lcnt : N >$
$=$
$< O : Ob|Pr : (!Q(IN)); awaitingElement(((insert(N);(N?(OUT));P),L'));R,$
$\quad Lcnt : N > .$

$***$ **statisk kall**
$eq\ < O : Ob|Pr : element((((Q@C(IN;OUT));P),L'));R, Lcnt : N >$
$=$
$< O : Ob|Pr : (!(Q@C)(IN)); awaitingElement(((insert(N);(N?(OUT));P),L'));R,$
$\quad Lcnt : N > .$

Denne løsningen krever i tillegg tre nye ligninger. Den eksisterende bindingsmekanismen bruker regelen "invoc-msg" (linje 893 i vedlegg A) til å motta en melding. Regelen henter invoc-meldingen fra konfigurasjonen og plasserer den i mottagerobjektets meldingskø. Denne regelen representerer avslutningen av trinn 1 i figur 5.3 på neste side. Vi blir nødt til å lage en ligning som brukes istedenfor denne regelen når invoc-meldingen stammer fra et internt kall. Ligningen matcher tilfellet hvor sender er lik mottager, representert ved variabelen O.

$eq\ < O : Qu|Ev : M >\quad invoc(O, MM, (ODL))$
$=$
$< O : Qu|Ev : M + invoc(O, MM, (ODL)) > .$

Den eksisterende bindingsmekanismen har to regler som genererer en
bindMtd-melding på grunnlag av en mottatt invoc-melding. Begge reglene heter "recieve-call-req" (linje 814 og 821 i A). Forskjellen mellom dem er at den ene brukes i tilfellet statisk kall, den andre i tilfellet dynamisk kall. Vi må definere to ligninger som brukes istedenfor disse. Dette gjør faktisk regelen for statiske kall overflødig, siden alle statiske kall er interne.

$eq\ < O : Ob|Cl : C\#V >\ \ < O : Qu|Ev : M + invoc(O, Q, (O\ DL)) >$
$=$
$< O : Ob|Cl : C\#V >\ \ < O : Qu|Ev : M >\quad bindMtd(O, Q, (O\ DL), C\#V[nil]).$

$eq\ < O : Qu|Ev : M + invoc(O, Q@C, (O\ DL)) >$
$=$
$< O : Qu|Ev : M >\quad bindMtd(O, Q, (O\ DL), C'\#0[nil]).$

Konklusjon

Både bindMtd-løsningen og invoc-løsningen, har blitt testet og virker. Fordelen med bindMtd-løsningen er lite kode. Ulempen er at vi mister den uniforme måten å behandle metodekall på. Selv om løsningen baserer seg på en delmengde av den opprinnelige mekanismen, gjør den spesifikasjonen av Creol sin semantikk mindre intuitiv. Det er kanskje også en fare for at interpreten blir skjørere i betydningen mer utsatt for følgefeil ved eventuelle framtidige endringer.

Fordelen med invoc-løsningen er at vi i større grad beholder den uniforme måten å behandle meldinger på. Ulempen er at løsningen krever mer kode. I tillegg til tre ligninger som også den andre løsningen krever, trenger vi ytterligere tre. Den ene av disse vil riktig nok erstatte en eksisterende regel, men det er fremdeles en relativt stor endring av interpreten.

Jeg har valgt bindMtd-løsningen fordi jeg vektlegger lite kode og at den kompliserer interpreten i mindre grad enn invoc-løsningen fordi den gjenbruker en delmengde av den eksisterende mekanismen.

5.5.4 Forlengelse av gren i ekspansjonstreet

La oss si at vi har lykkes i å finne de kallene som skal bindes og å legge en bindMtd- eller invoc-melding i konfigurasjonen. Den eksisterende bindingsmekanismen sørger for at metodeinstansen havner i PrQ. Vi trenger å hente instansen fra PrQ til Pr for å inkludere metodeinstansen i treet som i avsnitt 5.5.1 på side 58.

I følgende ligning ser vi hvorfor insert-setningen trenger å ta et heltall som parameter. Tallet er metodekallets merkelappverdi, som ble hentet fra Lcnt da kallet ble bundet. Merkelappen i insert-setningen brukes til å finne den riktige prosessen blant alle som måtte ligge i PrQ. Instansen pakkes inn i en element-setning og plasseres fremst i Pr. Deretter blir ligningene for ekspansjon brukt på metodeinstansen hvis mulig.

$$
\begin{aligned}
&eq\ < O : Ob | Pr : awaitingElement(((insert(N'); P), L'')); R, \\
&\quad PrQ : (P', (('caller : O), ('label : int(N')), L')) : W > \\
&= \\
&< O : Ob | Pr : element((P', (('caller : O), ('label : int(N')), L'))); \\
&\quad awaitingElement(((insert(N'); P), L'')); R, PrQ : W > .
\end{aligned}
$$

5.5.5 Folding av grenene i ekspansjonstreet

Vi skal se på reglene som nester termer til å bli én sammensatt setning med ekspanderte kall. Splitt-og-hersk-taktikken har gitt en kompleks spesifikasjon for ekspansjonsfasen. Det er mye lettere å trekke sammen trådene i foldefasen. Vi lar simpelthen koden i en element-setning erstatte insert-setningen inne i awaitingElement-setningen. Koden pakkes inn i en run-setning, slik at ekspansjonsligningene ikke skal kunne bli brukt på det samme setningsleddet på et senere tidspunkt.

$$
\begin{aligned}
&rl[fold - to - element - syncmerge] : \\
&element((P, L)); awaitingElement(((insert(0) \& P'); R, L)) \\
&=> \ element(((run(P) \& P'); R, L)).
\end{aligned}
$$

$$
\begin{aligned}
&rl[fold - to - element - nondet] : \\
&element((P, L)); awaitingElement(((insert(0) [] P'); R, L)) \\
&=> \ element(((run(P) [] P'); R, L)).
\end{aligned}
$$

$$
\begin{aligned}
&rl[fold - to - element - merge] : \\
&element((P, L)); awaitingElement(((insert(0) ||| P'); R, L)) \\
&=> \ element(((run(P) ||| P'); R, L)).
\end{aligned}
$$

Vi ser at den lokale tilstanden L i element-setningen er den samme som i awaitingElement-setningen og den kan derfor kastes.

Vi må ta spesielt hensyn til nesting av metodeinstanser. Her får vi et gjensyn med runMtd-setningen:

$crl[fold-to-element-call]:$
$element(PROC);awaitingElement((insert(N);P,L))$
$=>\ element((runMtd(PROC);P,L))$
$if\ N>0.$

Betingelsen $N > 0$ gjør at vi er sikre på at prosessen i $PROC$ er en metodeinstans. Merk at de lokale variablene i PROC blir beholdt i runMtd-setningen, i motsetning til i reglene ovenfor.

Til slutt trenger vi regler som folder treet helt sammen. Treet får nå muligheten til å ekspandere langs grenen som starter med P', hvis det ikke allerede har blitt gjort.

$rl[fold-to-root-syncmerge]:$
$element((P,L));insert(0)\&P'\ =>\ run(P)\&P'.$

$rl[fold-to-root-nondet]:$
$element((P,L));insert(0)[]P'\ =>\ run(P)[]P'.$

$rl[fold-to-root-merge]:$
$element((P,L));insert(0)|||P'\ =>\ run(P)|||P'.$

5.6 Redefinering av fletting

På bakgrunn av den bindingsmekanismen som har blitt beskrevet hittil, kan vi nå gå ut ifra at vi har en &-setning med ekspanderte metodekall. Det har vært gitt fra begynnelsen at setningene i leddene i &-setningen skal flettes med |||-operatoren. Følgelig er semantikken til synkronisert fletting betinget av semantikken til flettemekanismen. Vi skal nå se nærmere på semantikken til |||-operatoren.

Det hadde vært ønskelig å la definisjonen av fletting være uberørt av implementasjonen av synkronisert fletting, men jeg har redefinert den av to grunner. For det første er binding av kall i kontekst av synkronisert fletting tilsynelatende enklest å implementere hvis |||-operatoren er kommutativ.[5] Den versjonen av interpreten som jeg har brukt som utgangspunkt for denne masteroppgaven har en ikke-kommutativ variant, men i Creol har det visstnok tidligere vært foreslått en kommutativ variant. I kontekst av problemstillingen for denne masteroppgaven er det altså ønskelig å se på mulighetene for en kommutativ variant. For det andre oppdaget jeg en liten feil i de eksisterende reglene som nevnt i avsnitt 5.5.1 på side 59, så de måtte ha blitt endret uansett.

Vi skal først se på råderommet (eng: scope) for vakter i kontekst av &-operatoren. Deretter ser vi på ulike flettepolicyer. Etter dette fortsetter vi med resten av Maude-reglene/-ligningene for synkronisert fletting i avsnitt 5.7.

5.6.1 Tolkninger av synkronisert fletting

Vi må gjøre det helt klart hva som bør være råderommet til en vakt i kontekst av &-operatoren. Hva mener jeg med dette?

Når vi skal se på råderommet til vaktene, er vi interessert i å se på hva tolkningene sier om hvordan leddene kan påvirke hverandres vakter. Vi tar utgangspunkt i følgende setning:

$$(await\ G1;S1;S1')\ \&\ (await\ G2;S2;S2')$$

Hvis vi ikke vet noe om semantikken til |||, kan rekkefølgen på kjøringen av setningen S1, S1', S2 og S2' være helt vilkårlig. I [8] utvikler forfatterne resonneringsregler for synkronisert fletting som brukes til å resonnere omkring invarianter i Creol-programmer. Reglene forutsetter at ingen variabler som inngår i G2 vil bli endret av $S1;S1'$ og at ingen variabler som inngår i G1 vil bli endret av $S2;S2'$. Det ser ut til å være meningen at det skal være programmererens ansvar å oppfylle denne antagelsen. Dette ansvaret kan kanskje overlates til typeanalysen, men det er mer relevant for oss at det er mulig å spesifisere mekanismen for synkronisert fletting slik at antagelsen blir garantert av interpreten. Vi skal se nærmere på mulighetene for dette.

[5] Denne påstanden reflekterer jeg over i 5.6.3.

For å anskueliggjøre de mulige effektene av &-operatoren, kan vi bruke en notasjon som er inspirert av programmeringslogikk [1], som er en variant av Hoare-logikk. I denne notasjonen definerer jeg $\{G\}S$ til å bety at betingelsen G er oppfylt når kjøringen av programsetningen S starter. En setning S uten en slik forbetingelse angitt med krøllparenteser betyr at prosessen kan være i en hvilken som helst tilstand før kjøringen av S.

Vi kan velge én av tre følgende tolkninger av $(await\ G1; S1; S1')\&(await\ G2; S2; S2')$, som forklares ytterligere nedenfor:

1. $\{G1 \wedge G2\}\ (S1; S1')\ |||\ (S2; S2')$
2. $\{G1 \wedge G2\}\ (\{G1\}S1; S1')\ |||\ (\{G2\}S2; S2')$
3. $\{G1 \wedge G2\}\ (\{G1 \wedge G2\}S1; S1')\ |||\ (\{G1 \wedge G2\}S2; S2')$

Alle variantene fastslår at før noen av setningene i den opprinnelige &-setningen blir kjørt, skal konjunksjonen av leddenes vakter være sann.

Det er også mulig å tolke synkronisert fletting slik:

- $\{G1 \wedge G2\}\ (\{G1\}S1; \{G1\}S1')\ |||\ (\{G2\}S2; \{G2\}S2')$
- $\{G1 \wedge G2\}\ (\{G1 \wedge G2\}S1; \{G1 \wedge G2\}S1')\ |||\ (\{G1 \wedge G2\}S2; \{G1 \wedge G2\}S2')$

Disse tolkningene kan kanskje implementeres med et tenkt konstrukt superAwait som fungerer slik:

$$superAwait(G, S; S') \rightsquigarrow await\ G; S; superAwait(G, S')$$

Jeg vurderer disse to tolkningene til å være så langt fra den opprinnelige definisjonen av synkronisert fletting at jeg ikke ser nærmere på dem. De er mer relevante for et annet konstrukt (&&) som er ment å bli inkludert i Creol.

1. tolkning tilsvarer spesifikasjonen i den første versjonen, overfladisk beskrevet i 5.4. Der fjernes de innledende vaktene fra leddene etter at &-setningen har blitt vurdert til å være beredt. Merk at etter at vi har sjekket at betingelsene G1 og G2 er oppfylte samtidig, så gir vi ingen garanti for at de forblir oppfylte. Det betyr at på det tidspunktet hvor S2 skal kjøres, kan S1 eller S1' ha endret tilstanden i objektet slik at S2 sin forutsetning G2 ikke lenger er til stede. Vi garanterer heller ikke at G1 gjelder når vi begynner å kjøre S1'.

Dette er ikke helt heldig for resonneringen over programmer. Når vi i det andre leddet skriver koden $await\ G2; S2 \ldots$, forventer vi at G2 gjelder før S2 slipper til. Vi kan risikere at S2 blir kjørt under betingelser som den ikke ville blitt kjørt under hvis den ikke hadde forekommet i kontekst av &. Det blir vanskelig å bevise at synkroniseringsbetingelsen til en tidligere bevoktet setningssekvens er oppfylt i alle mulige kjøringer/eksekveringshistorier.

En implementasjon av denne tolkningen vil ikke realisere den ovenfornevnte antagelsen.

2. tolkning spesifiserer at det fremste leddet sin vakt er sann når vi begynner kjøringen av det fremste leddet og at det andre leddet sin vakt er sann når vi begynner kjøringen av det andre leddet. Dette tilsvarer den implementasjonen som har blitt gjengitt hittil i kapitlet. Vi gir ingen garantier for tilstanden til objektet før kjøringen av S1' eller S2'.

En implementasjon av denne tolkningen vil realiser den ovenfornevnte antagelsen.

3. tolkning sier at hver innledende vakt i hvert ledd i &-setningen skal forsterkes med alle de andre innledende vaktene. Konjunksjonen av vakter skal være sann når vi begynner på kjøringen av hvert ledd, men ikke nødvendigvis etter at den fremste setningen i leddet er kjørt.

En implementasjon av denne tolkningen vil realisere den ovenfornevnte antagelsen og ytterligere styrke betingelsene for kjøring. Anta at S1 endrer variabler som inngår i G1. Da blir en konsekvens av den 3. tolkningen at $S2; S2'$ blir hindret i å kjøre i en tilstand hvor $S2; S2'$ ville ha blitt kjørt hvis den ikke forekom i kontekst av &-operatoren. Dette vanskeliggjør inkrementell resonnering.

Konklusjonen er at tolkning 2 ser ut til å være den vi ønsker oss. Mens vi initielt spesifiserte en omskriving

$$(await\ G1; S1)\ \&\ (await\ G2; S2) \rightsquigarrow S1\ |||\ S2$$

på betingelsen av at &-setningen er beredt, har vi når forsikret oss om at den valgte løsningen

$$(await\ G1; S1)\ \&\ (await\ G2; S2) \rightsquigarrow (await\ G1; S1)\ |||\ (await\ G2; S2)$$

er den beste. Tilfeldigvis er den også den enkleste å implementere, da vi slipper å fjerne vakter fra CMC-koden.

5.6.2 Policyer for fletting

I forrige avsnitt kom vi fram til hva som skal være råderommet til innledende vakter i ledd i &-setninger. Hvordan må flettemekanismen være definert for at dette målet skal bli oppfylt? Definisjonen må angi hvordan vi skal avgjøre hvilket ledd sin første setning som skal kjøres til enhver tid. Vi trenger å spesifisere det jeg velger å kalle en "flettepolicy".

I likhet med at en skedulerer og en dispatcher i et operativsystem nødvendigvis har hver sin policy, har flettemekanismen i Creol en flettepolicy. Ordet "skedulering" brukes generelt om å administrere en kø av prosesser som samtidig ønsker tilgang til den samme ressursen, for eksempel prosessoren. For at en skeduleringspolicy eller -algoritme skal være korrekt, må den sikre at en prosess som kjøres ikke kompromitterer sin egen eller andre prosessers dataintegritet som følge av skeduleringsrekkefølgen. [23] Det er en viss likhet mellom lettvektsprosesser/tråder og leddene i en |||-setning, i den forstand at vi må stille det samme kravet til flettepolicyen. Vi ser på tre ulike policyer for fletting i Creol.

Policy 1: Tilfeldig fletting

Den første av de tre flettepolicyene har jeg kalt "tilfeldig fletting". Den baserer seg på at |||-operatoren er kommutativ. I motsetning til den opprinnelige flettemekanismen, kan vi da klare oss med bare én regel, som kan skisseres slik:

$$run(SP; P)|||P' \rightsquigarrow SP; (run(P)|||P')\ if\ ready(SP, \ldots)$$

Denne regelen vil bli brukt på |||-setningen såframt det finnes et klart ledd. Det er fordi at selv om regelens venstreside syntaktisk angir en rekkefølge på leddene, så sier kommutativitetsegenskapen til |||-operatoren at rekkefølgen er tilfeldig.

Policyen gir ingen forutsigbarhet når det gjelder rekkefølgen som setningene kjøres i. Dette er uheldig for resonnering. Vi tyr igjen til notasjonen fra programmeringslogikk som ble introdusert tidligere for å forklare hvorfor. Denne gangen skiller vi mellom for- og bakbetingelser, slik at $\{G1\}S\{G2\}$ er et utsagn om at når kjøringen av S begynner, er betingelsen G1 oppfylt, og når S terminerer er G2 oppfylt.

Se på dette eksemplet, hvor G1 er en boolsk variabel:
$(await\ G1; S1)\ |||\ (await\ G2; G1 := false)$ Siden flettingen er helt vilkårlig, vil kjøringen av setningene kunne ta rekkefølgen $await\ G1; await\ G2; G1 := false; S1$. Vi ser at S1 blir kjørt etter at S1 sin betingelse ikke lenger er oppfylt. Selv om vakten G1 er sann etter setningen await G1, er den ikke sann før S1. Vi ønsker

$$\{\ldots\}await\ G1\{G1\}; \{G1\}S1\{\ldots\}$$

men vi kan få

$$\{\ldots\}await\ G1\{G1\}; \{\neg G1\}S1\{\ldots\}$$

Dette forpurrer hele poenget med vakter. Tilfeldig fletting er ikke brukbart.

Policy 2: Delvis ordnet fletting

Tilfeldig fletting tilbyr ingen forutsigbarhet. Delvis ordnet fletting gir ikke forutsigbarhet for rekkefølgen av leddene, men for setningene i et ledd. Hele poenget med denne policyen er å sikre at hvis et setningsledd har kodeskevensen $await\ G; S$ og await-setningen blir kjørt, så gjelder G også når vi starter kjøringen av S. Vi har $\{\ldots\}await\ G\{G\}; \{G\}S\{\ldots\}$). Dette løser problemet som gjorde tilfeldig fletting uakseptabelt.

Jeg beskriver nå min implementasjon av tilfeldig fletting. Min variant av |||-operatoren er kommutativ. Vi deklarerer en ny operator ///, som er lik |||, med det unntaket at /// ikke er kommutativ. Vi definerer en ny hjelpefunksjon toggleMergeOrder().

$$
\begin{array}{ll}
op\ \_///\_\ :\ ProgList\ ProgList & -> Prog\ [ctor\ assoc].\\
op\ toggleMergeOrder\ :\ ProgList & -> ProgList.\\
eq\ toggleMergeOrder(P|||P') & = toggleMergeOrder(P)///\\
 & \quad toggleMergeOrder(P').\\
eq\ toggleMergeOrder(P///P') & = toggleMergeOrder(P)|||\\
 & \quad toggleMergeOrder(P').\\
eq\ toggleMergeOrder(P) & = P[owise].
\end{array}
$$

Hvis vi gir en |||-setning som parameter til toggleMergeOrder(), får vi tilbake en ///-setning med de samme leddene. Disse leddene har da en vilkårlig rekkefølge som konsekvens av kommutativitetsegenskapen til |||-operatoren og som konsekvens av at leddene har forekommet i kontekst av &-operatoren, som også er kommutativ, på et tidligere tidspunkt. Men rekkefølgen er uforanderlig, såframt den ikke endres eksplisitt, på grunn av at ///-operatoren ikke er kommutativ. Hvis vi gir toggleMergeOrder() en ///-setning som parameter, får vi tilbake en |||-setning med de samme leddene. Leddene har da ikke lenger noen ordnet rekkefølge.

Vi får delvis ordnet fletting ved å bruke følgende skisserte regler. Første regel sier at hvis det fremste leddet er klart, skal dets fremste setning kjøres:

$$
\begin{array}{c}
run(SP; P)///P' \rightsquigarrow SP; (run(P)///P')\\
if\ ready(SP, \ldots)\ .
\end{array}
$$

Neste regel sier at hvis det fremste leddet ikke er klart, så opphever vi rekkefølgen på leddene:

$$
\begin{array}{c}
run(SP; P)///P' \rightsquigarrow toggleMergeOrder(run(P)///P')\\
if\ not\ ready(SP, \ldots)\ .
\end{array}
$$

Siste regel sier at hvis det finnes et klart ledd blant en uordnet mengde ledd, så skal dette leddet stå fremst i en liste av de samme leddene:

$$
\begin{array}{c}
run(SP; P)|||P' \rightsquigarrow run(SP; P)///toggleMergeOrder(P')\\
if\ ready(SP, \ldots)\ .
\end{array}
$$

Forøvrig vises bruken av denne hjelpefunksjonen i avsnitt 5.7.3 på side 78.

Denne flettepolicyen er delvis ordnet fordi:

1. Vi har ingen forutsigbarhet i valg av rekkefølgen på leddene. Dette er en konsekvens av at &-operatoren er kommutiativ, noe som gjør at setningsleddene ikke har noen ordnet rekkefølge før flettingen påbegynnes.
2. Derimot, når flettemekanismen først har valgt et ledd, har vi forutsigbarhet for kjøringen av setningene fra det leddet.

Policy 3: Ordnet fletting

Den policyen som jeg velger å kalle "ordnet fletting" er den som er brukt i den opprinnelige definisjonen av |||. Anta at vi løser problemet med å isolere det første leddet ved å bruke den nevnte run-setningen. Vi bruker en ikke-kommutativ variant av |||-operatoren, slik at leddene har en listerekkefølge. Reglene "merge-first" og "merge-second" beskrevet i avsnitt 5.5.1 på side 59 kan da skisseres slik:

$$
\begin{array}{c}
run(SP; P)|||P' \rightsquigarrow_{merge-first} SP; (run(P)|||P')\\
if\ ready(SP, \ldots)
\end{array}
$$

$$
\begin{array}{c}
run(P)|||P' \rightsquigarrow_{merge-second} P'|||run(P)\\
if\ ready(P', \ldots)\ and\ not\ ready(P, \ldots)
\end{array}
$$

Denne flettingen er ordnet på den måten at vi har stor grad av forutsigbarhet hva gjelder rekkefølgen på kjøringen av setningene. Hvis første setning i et ledd blir kjørt, vet vi at andre setning i samme ledd vil bli kjørt, såframt setningen er klar.[6] Hvis det fremste leddet ikke er klart, vet vi at det alltid er det nest fremste leddet som vil bli forsøkt kjørt.

[6] Semantikken til hjelpefunksjonen ready() og begrepet "klar" er gjort rede for i avsnitt 4.2.15 på side 47.

5.6.3 Kritikk av mitt valg av flettepolicy

Vi har et valg blant fire alternativer når det gjelder hvordan en &-setning skal kjøres. Disse policyene har ulik grad av "strenghet" ved seg, det vil si hvor forutsigbar rekkefølgen på kjøringen av setningene i &-setningen blir:

1. Tilfeldig fletting.
2. Delvis ordnet fletting.
3. Ordnet fletting.
4. Ingen fletting. I løpet av kjøringen av en &-setning kan vi bytte ut &-operatoren med ;-operatoren istedenfor |||.

Tilfeldig fletting er uaktuelt fordi det ødelegger den fundamentale virkemåten til prosessorslippunkter. Alternativ 4 faller ut fordi det er et krav at leddene skal flettes. Jeg endte opp med å implementere delvis ordnet fletting med en ad hoc begrunnelse, som vi straks kommer inn på. I ettertid har jeg bestemt meg for at ordnet fletting er best. Jeg forklarer nå hvorfor.

Ingen implementeringsfordel med delvis ordnet fletting

Motivasjonen for å gjøre &-, |||- og []-operatoren kommutative var at det ville være lettere å implementere og kreve færre Maude-regler. Jeg implementerte tilfeldig fletting før jeg oppdaget problemet med det. Jeg implementerte deretter delvis ordnet fletting for å korrigere semantikken. I den policyen er flettemekanismen mer forutsigbar, men grunnen til at policyen ikke er ordnet er at i bindingsfasen av kjøringen av en &-setning, før &-operatoren byttes ut med |||, vil leddene ordnes i en vilkårlig rekkefølge. Dette gjør at selv om definisjonen av ||| implementerer ordnet fletting, vil kommutativiteten til &-operatoren gjøre at vi får delvis ordnet *synkronisert* fletting.

Med en ikke-kommutativ fletteoperator er vi nødt til å lage en egen regel for å stokke om på rekkefølgen av setningsleddene:

$$
\begin{array}{l}
eq\ < O : Ob | Pr : (run(SP; P)///P'); R, Lvar : L, Att : A > \\
< O : Qu | Ev : M > \\
= \\
< O : Ob | Pr : (P'///run(SP; P)); R, Lvar : L, Att : A > \\
< O : Qu | Ev : M > \\
if\ enabled(P', (L, A), M)\ and \\
not\ (enabled(run(SP; P), (L, A), M)\ or\ SP :: RunMtd).
\end{array}
$$

Her ser vi at fremste leddet i ///-setningen plasseres bakerst hvis det ikke er beredt. Da vil ledd nummer to få sjansen til å kjøre. Hvis ikke ledd nummer to er beredt, gjentar vi handlingen, og ledd nummer tre får sjansen, og så videre. I reglene i den gitte interpreten var det et problem at setningsleddene ikke ble korrekt isolerte (se avsnitt 5.5.1 på side 59). Introduksjonen av run-setningen gjør at vi ikke har dette problemet her.

Med en ikke-kommutativ &-operator er vi også nødt til å endre rekkefølgen på leddene eksplisitt for å få ekstrahert hvert ledd i &-setningen. Jeg trodde at jeg var nødt til å lage en ligning som gjør det for hver av de seks ekstraksjonsligningene i 5.5.1 på side 60 og side 62. Det viser seg at det ikke er nødvendig. La oss bruke en av de seks ekspansjonsligningene som eksempel. Vi husker at &-operatoren er kommutativ i denne ligningen:

$$
\begin{array}{l}
ceq\ < O : Ob | Pr : (P \& P'); R, Lvar : L > \\
= \\
< O : Ob | Pr : element((P, L)); (insert(0) \& P'); R, Lvar : L > \\
if\ not\ (P :: Run\ or\ P :: SyncMerge).
\end{array}
$$

&-setningen kan ha vilkårlig mange ledd, men ligningen vil bli brukt på dem alle. Hvordan skal vi oppnå det samme når &-operatoren ikke er kommutativ? Vi husker at et ekstrahert ledd blir "nestet tilbake på plass" av en regel, i dette eksemplet av denne:

$rl[fold-to-root-syncmerge]:$
$element((P,L)); insert(0)\&P' \Rightarrow run(P)\&P'.$

Vi gjør følgende veldig enkle endring:

$rl[fold-to-root-syncmerge]:$
$element((P,L)); insert(0)\&P' \Rightarrow P'\&run(P).$

Da vil ovenforstående ligning — også med en ikke-kommutativ & — bli brukt på alle setningsleddene. I motsetning til hva jeg trodde, er det ikke nevneverdig enklere å implementere synkronisert fletting med kommutative operatorer enn med ikke-kommutativer operatorer.

Semantiske problemer med delvis ordnet fletting

En mye viktigere grunn til å velge ordnet framfor delvis ordnet fletting er at det er alvorlige semantiske problemer med delvis ordnet fletting. Jeg skal nå forklare hvorfor.

Simulering med Maude sin rew-kommando viste at i noen situasjoner er det kritisk at leddene i &-setningen ikke endrer rekkefølge. Vi skal nå se på to konkrete programeksempler som har latt meg oppdage dette. Kjøringen av disse førte til at programmet terminerte i en uventet og uønsket tilstand.

1) Lock-eksemplet Problemet med å la &-operatoren være kommutativ viser seg blant annet i metoden put() i klassen MutexBuffer, som vi møtte i kapittel 3.3:

```
class MutexBuffer(bufferSize: Nat) inherits GBuffer(), Lock()
begin
  op put() == lock@Lock(caller;) & put@GBuffer();
       unlock@Lock(caller;) .
...
end
```

Rekkefølgen på leddene kan bli $put@GBuffer(); lock@Lock(caller;)$. Stikk i strid med hva man skulle tro når man leser koden, vil vi da aksessere bufferen først og be om aksessrettigheten etterpå. I Creol og CMC kjører bare én prosess om gangen, så vi har aldri "race conditions" i forbindelse med kritiske regioner [1]. Likevel kan denne situasjonen føre til en alvorlig feil: Anta at put() inneholder et prosessorslippunkt som gjør at prosessen suspenderes. Det er da fritt fram for andre prosesser å aksessere bufferen.

Er det mulig å sno seg unna dette problemet ved å definere metoden annerledes? Se på

```
  op put() == lock@Lock(caller;);
       put@GBuffer() & unlock@Lock(caller;) .
```

Vi er nå sikre på at bufferen er låst før vi gjør put-operasjonen. Men igjen får vi problemer fordi rekkefølgen på setningsleddene kan bli annerledes enn vi forventer. Ved kjøring kan vi få rekkefølgen

$$lock@Lock(caller;); unlock@Lock(caller;); put@GBuffer()$$

Dette er like ille.

Det er også et annet problem med denne alternative metodedefinisjonen. Anta at metodene blir forsøkt kjørt i den rekkefølgen de er skrevet. Anta at låsen er ledig, slik at metoden lock() terminerer. Anta så at bufferen er full. Da vil put@Gbuffer() bli stoppet av metodevakten sin. Dermed suspenderes prosessen mens den eier låsen. Det betyr at ingen annen prosess får tilgang til å bufferen for å hente elementer ut. Vi får en vranglås som hindrer alle klienter fra å bruke bufferen.

2) gget-eksemplet Problemet gjelder metoden gget() i klassen GBuffer, som vi også husker fra kapittel 3.3:

```
class GBuffer(bufferSize: Nat) inherits 2Buffer()
begin var afterGet: Bool=false .
  op get() == get@Buffer();
       afterGet := true .
  op gget() == (await not afterGet) & get@GBuffer() .
...
end
```

Alt går bra så lenge vi antar at kjøringen av leddene i &-setningen tar den rekkefølgen som de er skrevet i. Problemet oppstår hvis de tar rekkefølgen $get@GBuffer(); (await\ not\ afterGet)$. Da blir først variabelen afterGet satt til sann i metoden get() i klassen GBuffer. Deretter forsøker vi å kjøre $await\ not\ afterGet$. Følgen av dette er at prosessen suspenderes når den ikke burde.

Jeg ser fire løsninger på dette problemet:

1. Vi kan fjerne vaktene/await-setningene fra koden til alle setningsleddene på tidspunktet etter at &-setningen har blitt vurdert til å være beredt og før kjøringen av leddene starter. Denne løsningen tilsvarer tolkning 1 av synkronisert fletting i 5.6.1, som ble forkastet på grunn av resonneringsproblemer.

2. Det er mulig å tenke seg en variant av &-operatoren som fjerner setningsledd som består utelukkende av en await-setning. Bruken av & vil vanligvis bestå av bare å legge til en vakt til en allerede definert metode. Etter at &-setningen har blitt vurdert til å være beredt, er det unødvendig å kjøre den await-setningen som implementerer vakten; den har allerede gjort nytten sin. Dette vil løse problemet i gget-eksemplet, men ikke i Lock-eksemplet. Denne varianten bør likevel implementeres, som en forfining av semantikken og en effektivisering av implementasjonen.

3. Vi kan gjøre &-operatoren ikke-kommutativ. Vi overlater da til programmereren å skrive leddene i en slik rekkefølge at vaktene ikke "går i beina på hverandre", som vi nettopp har sett et eksempel på. Det er dermed forholdsvis enkelt å resonnere seg fram til at denne delen av programmet er korrekt. Denne løsningen kan uten problemer kombineres med den ovenforstående.

4. I akkurat tilfellet med gget-eksemplet er det mulig å unngå suspendering ved å definere metoden annerledes. Vi kan kalle get() i klassen Buffer istedenfor GBuffer. Vi definerer da gget() slik:

   ```
   ((await not afterGet) & get@Buffer()); afterGet := true .
   ```

   Dette er ikke en brukbar generell løsning på problemet, men jeg bruker den i mitt kodeeksempel i seksjon 6.4 på side 90 for å få koden til å kjøre med den semantikken som jeg har implementert.

Konklusjon

Vi må konkludere med at bare (helt) ordnet fletting gir korrekt programadferd og resonneringskontroll. Det betyr at &-operatoren må være ikke-kommutativ.

Min variant av fletting gjør bruk av kombinasjonen av |||- og ///-operatoren på en slik måte at den ordnede rekkefølgen på setningsleddene oppheves hvis det fremste/aktuelle setningsleddet er ikke-beredt. Det er ikke noe problem, fordi vi vet at alle ledd er beredte på tidspunktet umiddelbart etter at &-operatoren har blitt byttet ut med ///-operatoren. Hvis min implementasjon hadde hatt en ikke-kommutativ &-operator, ville implementasjonen ha gitt en korrekt semantikk.

Hvis rekkefølgen på setningsleddene i en &-setning ikke hadde vært kritisk, ville delvis ordnet fletting vært den mest abstrakte løsningen som er ok. Siden det viser seg at vi trenger en større grad kontroll, tror jeg at min løsning — med den korrigeringen at &-operatoren må være kommutativ — er den mest abstrakte løsningen som er ok.

Det har tidligere blitt foreslått en flettemekanisme for Creol som ligner på den sistnevnte løsningen.[7] I dette forslaget brukes også en ikke-kommutativ ///-operator i kombinasjon med en kommutativ

[7] Forslaget finnes i den utvidete og foreløpig upubliserte versjonen av [14].

|||-operator. Forskjellen fra min implementasjon er at rekkefølgen på setningsleddene ikke oppheves fullstendig ved første forekomst av et ikke-beredt setningsledd.

Vi har sett at det er fullt mulig å bruke en ikke-kommutativ fletteoperator. Hvis vi deklarerer |||-operatoren til å være ikke-kommutativ, blir operatoren ///-overflødig og vi sitter igjen med en flettemekanisme som ligner mye på den opprinnelige og som har en litt enklere implementasjon. Enkelheten gjør en slik løsning tiltalende. Løsningen vil være mindre abstrakt, men det er ingen grunn til å tro at det er problematisk.

5.7 Kjøring av &-setning med ekspanderte kall

Vi har nå sett implementasjonen av to av fasene beskrevet i interpreteringen av en &-setning (5.2 på side 52): Vi har en mekanisme som lar oss binde alle relevante kall, og vi vet hvordan vi kan avgjøre hvorvidt setningen er beredt. Vi mangler å definere den operasjonelle semantikken for fasen som handler om å kjøre &-setningen etter at den har blitt vurdert til å være beredt.

La oss anta at vi har en &-setning, at kallene i denne ekspanderes og at forekomstene av &-operatoren byttes ut med ///. På et stadium i utviklingsprosessen valgte jeg en flettepolicy som gir delvis ordnet fletting. Vi har blitt introdusert for operatoren /// og toggleMergeOrder(), som brukes for å implementere denne policyen. Vi skal nå se hvordan de brukes.

5.7.1 Dispatching av setninger fra ledd i |||-setningen

Umiddelbart etter at &-forekomstene har blitt erstattet med ///, vet vi at det fremste leddet er beredt. Da vil en av følgende to regler bli brukt:

$crl[merge - dispatch]$:
$< O : Ob | Pr : (run(SP; P)///P'); R, Lvar : L, Att : A >$
$< O : Qu | Ev : M >$
$=>$
$< O : Ob | Pr : SP; (run(P)///P'); R, Lvar : L, Att : A >$
$< O : Qu | Ev : M >$
$if\ (not\ SP :: RunMtd)\ or\ enabled(SP, (L, A), M).$

Legg merke til regelens betingelse. For det første bruker vi hjelpefunksjonen enabled() istedenfor ready(). Vi kan godt kan tenke oss at vi ønsker å gjøre blokkerende operasjoner i en flettesetning. Siden vi ønsker en høy grad av resonneringskontroll, ønsker vi ikke at kode fra et annet ledd i flette-setningen blir kjørt når vi gjør en operasjon som ville vært blokkerende utenfor konteksten av flettemekanismen. Det fremste eksemplet på dette er ?-setningen (svarsetningen) i synkrone kall. Mens vi venter på metodeinstansen fra kallet (!-setningen), er ?-setningen ikke klar, men den er beredt.

For det andre sier betingelsen at denne regelen ikke brukes hvis den fremste setningen i det fremste leddet er en runMtd-setning. Hvorfor ikke? Dette forstår vi når vi ser på neste regel:

$rl[merge - dispatch - method]$:
$< O : Ob | Pr : (run(runMtd(PROC); (N'?(OUT)); P)///P'); R >$
$=>$
$< O : Ob | Pr : runMtd(PROC); (N'?(OUT)); (run(P)///P'); R >$

Hvis den fremste setningen i det fremste leddet er en runMtd-setning, betyr det at vi har et synkront kall som har forekommet i kontekst av &-operatoren. Ut i fra hvordan vi har implementert ekspansjon av slike kall, vet vi at andre setning i det fremste leddet må være en ?-setning. Begge setningene hentes ut fra elementet i |||-setningen. Dette tilsvarer flettesemantikken til et synkront kall som ikke har forekommet i kontekst av synkronisert fletting og som derfor ikke har blitt ekspandert. Da ville kallet først blitt flyttet ut av flette-setningen og plassert fremst i Pr. Så ville kallet ha blitt redusert til en !-setning etterfulgt av en ?-setning (linje 865 i A). Denne splittingen har vi gjort på forhånd (se 5.5.2 på side 64 og 5.5.5 på side 69). Regelen "local-reentrance" i interpreten (linje 833) tar imot en etterspurt metodeinstans og skedulerer den. Men regelen matcher bare en tilstand hvor ?-setningen står for seg selv som fremste setning i Pr. Hvis vi i vår nye regel "merge-dispatch" lot ?-setningen bli værende inne i |||-setningen etter

å ha tatt ut runMtd, så ville vi aldri få match med "local-reentrance", og metodeinstansen ville aldri blitt skedulert.

Etter hvert som alle setningene i et ledd blir kjørt og antall ledd i ///-setningen blir færre, trenger vi disse ligningene:

$$
\begin{array}{ll}
eq\ run(empty) & = empty. \\
eq\ empty///P & = P. \\
eq\ < O : Ob | Pr : run(P); R > & =< O : Ob | Pr : P; R > .
\end{array}
$$

5.7.2 Kontekstskifter

Vi trenger å spesifisere hvordan en runMtd-setning skal kjøres. Anta at vi ønsker å kjøre setningssekvensen $runMtd((MC, ML)); S1$, hvor (MC,ML) er en metodeinstans representert av en Maude-term av sort Process, hvor MC er programkoden og ML er den lokale tilstanden. Kjøringen av runMtd-setningen må medføre et kontekstbytte. Etter at prosessen som er inneholdt i runMtd-setningen har terminert, må vi bytte kontekst enda en gang for å kunne kjøre S1. Dette lar seg løse med en regel som kan skisseres slik:

$$
\begin{array}{c}
< O : Ob | Pr : runMtd((MC, ML)); S1, Lvar : L > \\
\rightsquigarrow \\
< O : Ob | Pr : MC; runMtd((S1, L)), Lvar : ML >
\end{array}
$$

Her ser vi at de to prosessenes lokale tilstander holdes adskilt. Når runMtd-setningen skal kjøres, plasserer vi den lokale tilstanden dens i Lvar. Den tilstanden som allerede befinner seg i Lvar, "pakkes inn" i en annen runMtd-setning, sammen med resten av koden.

I den endelig versjonen av implementasjonen av synkronisert fletting har jeg laget en variant av denne regelen. Der legger vi S1 sin prosess i PrQ istedenfor å lage en ny runMtd-setning:

$$
\begin{array}{c}
< O : Ob | Pr : runMtd((MC, ML)); S1, PrQ : W, Lvar : L > \\
\rightsquigarrow \\
< O : Ob | Pr : MC; \ldots, PrQ : (S1, L) : W, Lvar : ML >
\end{array}
$$

$rl[run - mtd]$:
$< O : Ob | Pr : runMtd((P, (('caller : O), ('label : int(N')), L')));$
$(N'?(OUT)); R, PrQ : W, Lvar : L >$
$< O : Qu | Ev : Keep : H >$
$=>$
$< O : Ob | Pr : P; continue(N'), PrQ : (((N'?(OUT)); R), L) : W,$
$Lvar : (('caller : O), ('label : int(N')), L') >$
$< O : Qu | Ev : Keep : H; N' > .$

Continue-setningen brukes for å tvinge fram en skedulering av resten av runMtd-setningens prosess. Vi ser i 5.5.2 på side 64 hvordan setningen $(N'?(OUT))$ har blitt generert, slik at vi får en match med regelen "continue" i den eksisterende interpreten (linje 840 i tillegg A på side 111).

Grunnen til at jeg velger denne siste varianten er at modellen av Creol generelt legger opp til at prosesser som ikke har prosessoren skal lagres i PrQ. På denne måten får vi en mer uniform behandling av prosesser og skeduleringen av dem. Ved binding av kall lagret vi prosesser i Pr fordi det var praktisk for bruken av enabled(). Etter beredthetssjekken av &-setningen er det ikke noen grunn til å ha ikke-kjørende prosesser i Pr.

5.7.3 Skedulering av ledd i flettemekanismen

Hva gjør vi hvis det fremste leddet i ///-setningen ikke er beredt? Hvis ingen av leddene i ///-setningen er beredte, suspenderes prosessen. Hvis det finnes et annet ledd i ///-setningen som er beredt, må vi finne det og kjøre første setning fra det.

Vi husker at rekkefølgen på leddene i utgangspunktet er vilkårlig, fordi &-operatoren er kommutativ. Da er det kanskje like fornuftig som noe annet å ikke forsøke å ha noen bestemt skeduleringsrekkefølge på leddene. Vi bruker den kommutative egenskapen til |||-operatoren til å finne et vilkårlig, beredt ledd. Dette er implementasjonen av delvis ordnet fletting, som ble forklart i 5.6.2 på side 71. Følgende ligning bytter ut alle forekomster av /// med ||| ved bruk av toggleMergeOrder():

$ceq\ < O : Ob|Pr : (run(SP; P)///P'); R, Lvar : L, Att : A >$
$< O : Qu|Ev : M >$
$=$
$< O : Ob|Pr : toggleMergeOrder(run(SP; P)///P'); R, Lvar : L, Att : A >$
$< O : Qu|Ev : M >$
$if\ not\ (enabled(run(SP; P), (L, A), M)\ or\ SP :: RunMtd).$

Dette kunne like godt vært implementert med en regel. Betingelsen sier at ligningen brukes når SP verken er beredt eller SP er av sort RunMtd. Det siste er fordi en term av sort RunMtd representerer et metodekall, og alle kall er beredte i kontekst av ||| eller ///.

Følgende regel finner et beredt ledd og gir leddene en listerekkefølge, hvor dette leddet plasseres fremst:

$crl[merge - find - enabled] :$
$< O : Ob|Pr : (run(P)|||P'); R, L, Att : A >$
$< O : Qu|Ev : M >$
$=>$
$< O : Ob|Pr : (run(P)///toggleMergeOrder(P')); R, Lvar : L, Att : A >$
$< O : Qu|Ev : M >$
$if\ enabled(run(P), (L, A), M).$

Dette må være en regel og ikke en ligning, fordi termen i regelens venstre side som representerer objektet gir match med ligninger som brukes i bindingsmekanismen også. Hvis vi er i bindingsfasen av behandlingen av |||-setningen, ønsker vi å hindre at denne omskrivingen blander seg inn.

Som vi ser, er ikke denne implementasjonen av fletting så grasiøs som man kunne ønske seg. Som jeg konkluderte med i 5.6.3, ville jeg har laget en enklere, ikke-kommutativ implementasjon av |||-operatoren hvis jeg skulle gjort et nytt forsøk, og ville ha unngått ///-operatoren og toggleMergeOrder().

Til slutt kommer regelen "syncmerge". Dette er på en måte hovedregelen i implementasjonen. På grunn av at vi har implementert en mekanisme som gjør et grundig forarbeid, blir denne regelen enkel. Den sjekker om &-setningen er beredt og hvis så, bytter ut forekomstene av &-operatoren med ///. Vi kan bruke hjelpefunksjonen toggleMergeOrder() til denne erstattingen. Det gjør vi med å utvide definisjonen av funksjonen på side 72 med følgende ligning:

$$eq\ toggleMergeOrder(P\&P') = toggleMergeOrder(P)/// \\ toggleMergeOrder(P').$$

For ryddighetens skyld burde kanskje dette være to funksjoner; én som bytter ut & med /// og én som bytter fram og tilbake mellom ///- og |||-operatoren. Men siden vi ønsker å holde antallet nye operatorer nede, lar vi begge oppgavene bli utført av samme hjelpefunksjon.

$crl[syncmerge] :$
$< O : Ob|Pr : (P\&P'); R, Lvar : L, Att : A >$
$< O : Qu|Ev : M >$
$=>$
$< O : Ob|Pr : toggleMergeOrder(P\&P'); R, Lvar : L, Att : A >$
$< O : Qu|Ev : M >$
$if\ enabled((P\&P'), (L, A), M).$

Etter at regelen syncmerge har blitt brukt, er resten av kjøringen overlatt til reglene for ///- og |||-setninger.

5.8 Redefinering av ikke-deterministisk valg

Ekspansjon av kall i kontekst av [] I løpet av redegjørelsen for implementeringen av synkronisert fletting, har jeg presentert kode for ekspandering av kall i kontekst av operatoren for ikke-deterministisk valg uten å kommentere den. Jeg har redefinert []-operatoren fordi jeg så et Creol-kodeeksempel i et artikkelutkast hvor metodekall i kontekst av []-operatoren ble ekspanderte på samme måte som i kontekst av &-operatoren. Ekspansjon av metodekall i forbindelse med ikke-deterministisk valg har ikke blitt en

del av Creol siden da, og det er ikke sikkert at det er ønskelig heller. Mekanismen for ekspansjon er uansett helt lik den som brukes i kontekst av synkronisert fletting. Mengden merarbeid med å implementere det har vært liten. Hvis man skulle bestemme seg for ikke å ekspandere kall i kontekst av ikke-deterministisk valg, vil det kreve bare enkle endringer: Betingelsene i ligningene for ekspansjon av []-setninger endres fra å være lik betingelsene for &-setninger til å bli lik betingelsene for |||-setninger, beskrevet i kapittel 5.5.1 på side 60 og utover.

En kommutativ variant av [] I motsetning til & og |||, kan []-operatoren godt være kommutativ. En []-setning skal per definisjon substitueres med et av sine ledd på en ikke-deterministisk måte, og da er det naturlig å implementere en []-operator som ikke ordner leddene. Den varianten som var implementert i interpreten som ble gitt meg er ikke kommutativ. Jeg kan tenke meg en grunn til at det bør være slik: Ikke-kommutativitet gir programmereren mulighet til å angi en prioritet på leddene. For eksempel kan man plassere det leddet fremst som man har grunn til å tro tar kortest tid å kjøre, og slik få en liten effektivitetsfordel. Kanskje kan det tenkes andre situasjoner hvor man helst vil at et ledd skal bli kjørt framfor et annet.

Jeg har gjort [] kommutativ. Fordelen med dette er at vi da bare trenger én regel, mens vi trenger to når [] ikke er kommutativ. Reglene nondet-p1 og nondet-p2 er de eksisterende reglene:

$$
\begin{array}{l}
crl[nondet-p1]: \\
< O : Ob | Cl : Pr : (P [] P'); R, Lvar : L, Att : A > \\
< O : Qu | Ev : M > \\
=> \\
< O : Ob | Pr : P; R, Lvar : L, Att : A > \\
< O : Qu | Ev : M > \\
if\ ready(P, (L, A), M).
\end{array}
$$

$$
\begin{array}{l}
crl[nondet-p2]: \\
< O : Ob | Cl : Pr : (P [] P'); R, Lvar : L, Att : A > \\
< O : Qu | Ev : M > \\
=> \\
< O : Ob | Pr : P'; R, Lvar : L, Att : A > \\
< O : Qu | Ev : M > \\
if ready(P', (L, A), M).
\end{array}
$$

Vi kan utelate "nondet-p2" og klare oss med "nondet-p1" når []-operatoren er kommutativ.

Hvilket ledd i []-setningen skal kjøres? Analyse av programeksempler som inneholder []-setninger har fått meg til å bli bevisst på en egenskap ved []-setninger som kanskje ikke er tilsiktet. Maude-spesifikasjonen av ikke-deterministisk valg er entydig, som konsekvens av at Maude er et formelt spesifikasjonsspråk, men det er på sin plass å se nærmere på hva den faktisk sier.

Spesifikasjonen av [] gir følgende omskrivingssekvens, hvor G1, G2 og G3 er boolske vakter, G1 og G2 har verdien sann og G3 har verdien usann:

$$
\begin{array}{c}
(await\ G1; G2 := false; G3 := true) \& ((await\ G2; S2) [] (await\ G3; S3)) \\
\rightsquigarrow \\
(await\ G1; G2 := false; G3 := true) /// ((await\ G2; S2) [] (await\ G3; S3)) \\
\text{venstre ledd kjøres} \rightsquigarrow \\
((await\ G2; S2) [] (await\ G3; S3)) \\
\rightsquigarrow \\
(await\ G3; S3)
\end{array}
$$

Initielt vurderes &-setningen til å være beredt, fordi begge leddene er beredte. Det høyre leddet er beredt fordi G2 er beredt. Etter at venstre ledd har kjørt er det høyre leddet beredt fordi G3 er beredt. Er det ønskelig at []-setningens høyre ledd blir kjørt når det var det venstre leddet som betinget kjøringen av &-setningen i utgangspunktet? Dette scenariet gjelder for den ikke-kommutative varianten også.

Ønsker vi en garanti for at vi får følgende omskrivingssekvens?:

$$(await\ G1; G2 := false; G3 := true) \& ((await\ G2; S2) [] (await\ G3; S3))$$
$$\rightsquigarrow$$
$$(await\ G1; G2 := false; G3 := true) /// (await\ G2; S2)$$
$$\text{venstre ledd kjøres} \rightsquigarrow$$
$$(await\ G2; S2)$$

Dette kan vi få til ved å lage en tenkt hjelpefunksjon purgeNondet() som går rekursivt gjennom en sammensatt setning, finner et beredt ledd i en []-setning og fjerner alle andre ledd i []-setningen, enten de er beredte eller ikke. Denne funksjonen kan brukes i den nylig introduserte regelen "syncmerge" på denne måten:

$crl[syncmerge]$:
$< O : Ob | Pr : (P \& P'); R, Lvar : L, Att : A >$
$< O : Qu | Ev : M >$
$=>$
$< O : Ob | Pr : purgeNondet(toggleMergeOrder(P \& P')); R, Lvar : L, Att : A >$
$< O : Qu | Ev : M >$
$if\ enabled((P \& P'), (L, A), M).$

5.9 Oppsummering

Tidlig i utviklingsprosessen identifiserte vi de tre hovedfasene i interpreteringen av en &-setning:

1. Binding av eventuelle innledende metodekall i hvert ledd.
2. Testing av om alle leddene er beredte samtidig, ved bruk av enabled().
3. Hvis &-setningen er beredt: Kjøring av alle leddene. Hvis &-setningen ikke er beredt: Suspendering av prosessen.

Vi oppsummerer svarene på utfordringene i hver av disse fasene.

5.9.1 Binding av innledende metodekall

Hvordan kommer vi oss dit hen at vi har en &-setning med ekspanderte kall? Den store utfordringen med dette er at &-setninger kan ha vilkårlig mange ledd og være vilkårlig komplekst sammensatt av |||-, []- og andre &-setninger. Vi bruker splitt-og-hersk-tilnærmingsmåten og lager en algoritme for å isolere hvert eneste setningsledd som inngår i &-setningen. Eventuelle sammensatte setninger i disse leddene blir splittet opp i sine respektive ledd på samme måte. Dette er visualisert med et trediagram. Når vi i denne algoritmen kommer til et ledd som innledes med et internt, synkront metodekall, bindes kallet, og den aktuelle grenen i ekspansjonstreet forlenges med metodeinstansen fra dette kallet. Dermed kan grenen ekspanderes videre, ved at en eventuell, innledende, sammensatt setning i denne metodekroppen splittes på samme måte. Vi innførte element- og awaitingElement-setningene i CMC (ikke en del av Creol) for å til dette.

Når det ikke finnes flere kall som skal bindes, går vi over i foldingsfasen. Setningsleddene og metodeinstansene nestes inn i hverandre. Vi ender opp med den samme &-setningen som vi hadde i utgangspunktet, bortsett fra at de relevante kallene er byttet ut med runMtd-setninger. RunMtd-setningen gjør det mulig å oppbevare metodeinstansens kode og lokale tilstand i én term inne i &-setningen, slik at vi får gjort de nødvendige kontekstskiftene ved kjøring.

$op\ runMtd\ :\ Process -> RunMtd\ [ctor].$

Det viste seg vanskelig å identifisere systemtilstander som lar oss skille mellom ekspansjonsfasen og foldingsfasen i algoritmen vår. Dette er en utfordring, fordi vi må unngå at vi folder før vi er ferdige med ekspansjonen. Vi løser dette ved å basere oss på en hensiktsmessig arbeidsfordeling mellom Maude-regler og -ligninger, hvor ekspansjon foregår med ligninger og folding med regler.

Bruken av ligninger medfører at vi må endre bindingsmekanismen til å fungere på en spesiell måte ved interne, synkrone kall i kontekst av synkronisert fletting. Den opprinnelige bindingsmekanismen er implementert ved bruk av noen regler. Vi lager en bindingsmekanisme som brukes i kontekst av &-operatoren hvor vi slipper å bruke reglene. En ulempe med dette er at binding av metodekall i språket som helhet ikke skjer på en like uniform måte som tidligere. Dette veies langt på vei opp av at min implementasjon benytter seg av ligningene i den eksisterende bindingsmekanismen og kan sies å være en delmengde av den opprinnelige mekanismen.

5.9.2 Hvordan avgjøre om en &-setning er beredt?

Vi starter med å justere eller rettere sagt konkretisere den uformelle definisjonen av synkronisert fletting. Vi gikk fra at $(await\ G1; S1)\&(awaitG2; S2)$ var definert som $await\ G1 \wedge G2; S1|||S2$ til å være definert som $(await\ G1; S1)|||(await\ G2; S2)$ på den betingelsen at &-setningen er beredt. Hvorvidt en setning er beredt, avgjøres med hjelpefunksjonen enabled().

Interne, synkrone kall ekspanderes i kontekst av &-operatoren. Hvordan avgjør vi om en &-setning med ekspanderte kall er beredt? Vi utvider enabled() med følgende ligning:

$$eq\ enabled(P\ \&\ P', L, M) \quad = enabled(P, L, M)\ and\ enabled(P', L, M).$$

Når enabled() skal avgjøre om en metodevakt i en ekspandert metode er beredt, må vakten evalueres i kontekst av metodens lokale variabler. RunMtd-setningen inneholder som sagt en metodeinstans. Det gjør at vi kan bruke enabled() til å vurdere beredtheten til denne metodeinstansen, etter at vi utvider enabled() til å kunne håndtere runMtd-setningen:

$$eq\ enabled(runMtd((P, L')), L, M) \quad = enabled(P, (L, L'), M).$$

Runmtd-setningen er avgjørende for at vi skal kunne lage følgende, relativt enkle hovedregel for kjøring av &-setninger:

$crl[syncmerge]:$
$< O : Ob|Pr : (P\&P'); R, Lvar : L, Att : A >$
$< O : Qu|Ev : M >$
$=>$
$< O : Ob|Pr : toggleMergeOrder(P\&P'); R, Lvar : L, Att : A >$
$< O : Qu|Ev : M >$
$if\ enabled((P\&P'), (L, A), M).$

5.9.3 Kjøring av en ferdig ekspandert &-setning

Hvordan skal vi kjøre en ferdig ekspandert &-setning? En utfordring med dette er kontekstskiftene som kreves for å kjøre de ekspanderte metodene. Runmtd-setningen lar oss lage en regel som i samarbeid med den eksisterende skeduleringsmekanismen implementerer kontekstskiftene:

$rl[run - mtd]:$
$< O : Ob|Pr : runMtd((P, (('caller : O), ('label : int(N')), L')));$
$(N'?(OUT)); R, PrQ : W, Lvar : L >$
$< O : Qu|Ev : Keep : H >$
$=>$
$< O : Ob|Pr : P; continue(N'), PrQ : (((N'?(OUT)); R), L) : W,$
$Lvar : (('caller : O), ('label : int(N')), L') >$
$< O : Qu|Ev : Keep : H; N' > .$

Ved å reflektere grundigere over semantikken til synkronisert fletting, fant vi tre mulige tolkninger. Hver tolkning kan uttrykkes i en notasjon som er inspirert at Hoare-logikk. Vi forsikret oss om at den tolkningen som ligger til grunn for denne oppgaven er den beste:

$$\{G1 \wedge G2\}\ (\{G1\}S1; S1')\ |||\ (\{G2\}S2; S2')$$

Konjunksjonen av leddenes vakter er sann før &-setningen, og den respektive vakt er sann før hvert ledd.

Semantikken til &-setningen er betinget av semantikken til |||-setningen. Ved ettertanke og maskinell analyse identifiserer vi tre ulike flettepolicyer, som avgjør hvordan setninger fra leddene i en |||-setning skal kjøres: Policyene tilfeldig fletting, delvis ordnet fletting og ordnet fletting har fått sine navn etter hvor forutsigbar rekkefølgen blir på kjøringen av setninger fra leddene i |||-setningen. Fordi vi har gjort &- og |||-operatoren kommutativ, har vi endt opp med å implementere delvis ordnet fletting. Vi innførte operatoren /// og hjelpefunksjonen toggleMergeOrder() for å til delvis ordnet fletting.

Det viser seg at det er alvorlige semantiske problemer med delvis ordnet fletting, og vi konkluderer med at ordnet fletting er den policyen som vil føre til korrekt semantikk for synkronisert fletting. Hvis vi skulle implementert en ny versjon av synkronisert fletting på bakgrunn av denne innsikten, ville vi fått en enklere implementasjon med ikke-kommutative operatorer. Vi ville blant annet unngått den noe kompliserte bruken av operatoren /// og hjelpefunksjonen toggleMergeOrder().

Kapittel 6

Analyse av implementasjonen av synkronisert fletting

Dette kapitlet handler om å bruke maskinell analyse til å gjøre en tentativ verifikasjon og validering av implementasjonen av synkronisert fletting. Jeg ønsker 1) å sannsynliggjøre at implementasjonen er feilfri og 2) å vise at Creol sin strategi for inkrementell endring av synkroniseringsbetingelser kan realiseres i programmer som Creol-interpreten kan tolke.

6.1 Samme–lås–eksemplet

Dette eksemplet er ment å vise 1) hvordan vi definerer en initialtilstand som vi bruker som utgangspunkt for en analyse og 2) at et av designvalgene vi har gjort er riktig.

Vi analyserer et programeksempel hvor flere ledd i én &-setning venter på at samme synkroniseringsbetingelse skal bli oppfylt. Hvorfor er dette meningsfullt? Vi husker at vi har valgt å implementere synkronisert fletting slik at vi har

$$\begin{gathered}(await\ G1; S1)\ \&\ (await\ G2; S2) \\ \rightsquigarrow \\ (await\ G1; S1)\ |||\ (await\ G2; S2) \\ if\ enabled((await\ G1; S1)\ \&\ (await\ G2; S2), \ldots)\end{gathered}$$

Dette står i motsetning til ett av mine tidligste forsøk, som hadde følgende semantikk:

$$\begin{gathered}(await\ G1; S1)\ \&\ (await\ G2; S2) \\ \rightsquigarrow \\ (S1)\ |||\ (S2) \\ if\ enabled((await\ G1; S1)\ \&\ (await\ G2; S2), \ldots)\end{gathered}$$

Her fjernes vaktene når &-setningen omskrives til en |||-setning. Hvis vi bruker disse to semantikkene på en situasjon hvor flere elementer venter på samme betingelse, får vi ulik adferd. Vi skal nå se hvorfor.

6.1.1 Creol

Her følger Creol-koden til et program som vil føre til ulike resultater avhengig av hvilken semantikk vi bruker.

```
class SameCondTest
begin
var cond: Bool=true, error: Bool=false, a: Bool=false,
      b: Bool=false, c: Bool=false .
  op run() == (await cond; if cond = false then error := true fi;
      cond := false; await wait; a := true; cond := true) &
```

(await cond; if cond = false then error := true fi; cond := false;
await wait; b := true; cond := true) &
(await cond; if cond = false then error := true fi; cond := false;
await wait; c := true; cond := true) .
end

6.1.2 Forventet resultat

Initielt er den felles betingelsen cond oppfylt, og variablene a, b, og c usanne. &-setningen blir vurdert til å være beredt, og &-operatoren byttet ut med |||. Det første leddet blir kjørt. Det setter cond til false. Deretter kommer en setning som gjør leddet ikke-beredt. Reglene for fletting vil da prøve å finne et annet ledd som er beredt og kjøre det istedenfor. Det er på dette tidspunktet vi finner forskjellen mellom den tidlige og den sene versjonen av implementasjonen. I den sene versjonen vil det ikke finnes noe beredt ledd, fordi ledd 2 og 3 venter på cond, som ledd 1 har satt til false. I den tidlige versjonen er vaktene fjernet. Flettemekanismen vil dermed sjekke om if-setningene er beredte, noe de alltid er. Et av leddene vil bli kjørt, til tross for at betingelsen som var satt ikke lenger er oppfylt.

Den tidlige versjonen ville ha ført til at systemet terminerte med variabelen error satt til true. Den versjonen av synkronisert fletting som jeg har endt opp med vil få systemet til å terminere i en tilstand hvor error er false og alle andre variabler er true.

6.1.3 CMC

Vi oversetter Creol-koden til CMC. Her har jeg definert en egen modul i en egen fil:

```
in v3-e-syncmerge.maude

mod SAMECOND is including SYNCMERGE .

op init : -> Configuration .
eq init = *** initialtilstanden er som følger:

< 'SameCondTest : Cl | Vs: 0, Inh: nil,
Att: ('cond : bool(true)), ('error : bool(false)),
  ('a : bool(false)), ('b : bool(false)), ('c : bool(false)),
Mtds: < 'run : Mtdname | Latt: no, Code:
  ((await 'cond); (if not 'cond th 'error := bool(true) fi);
  ('cond := bool(false)); (await wait); ('a := bool(true));
  ('cond := bool(true))) & ((await 'cond);
  (if not 'cond th 'error := bool(true) fi);
  ('cond := bool(false)); (await wait); ('b := bool(true));
  ('cond := bool(true))) & ((await 'cond);
  (if not 'cond th 'error := bool(true) fi);
  ('cond := bool(false)); (await wait); ('c := bool(true));
  ('cond := bool(true))) > ,
Ocnt: 0 >

new 'SameCondTest (nil)
. *** Punktum. Slutt på definisjonen av initialtilstanden.
endm

search init =>! C:Configuration . *** Vis alle sluttilstander
```

Filen "v3-e-syncmerge.maude" inneholder min utvidelse av interpreten (modulen SYNCMERGE), og interpreten importeres via den. Vi definerer en operator "init", som ved hjelp av en ligning reduseres til en initialtilstand. Etter modulen kommer søkekommandoen. Den sier at vi skal søke etter sluttilstander fra tilstanden som init reduseres til og som matcher variabelen C av sort Configuration. Det siste betyr at vi søker enhver konfigurasjon, uten begrensninger.

6.1.4 Analyse

Resultatet av søket er som forventet: Det finnes bare én sluttilstand, og i den er objektets variabel error usann og a, b og c sanne. Nedenfor vises den relevante delen av utskriften fra Maude. Den originale utskriften inneholder hele den resulterende konfigurasjonen, inkludert klassedefinisjonen. Vi har allerede sett klassedefinisjonen i initialtilstanden, og siden den ikke endres i løpet av kjøringen, erstatter jeg den med "...".

```
search in SAMECOND : init =>! C:Configuration .

Solution 1 (state 42)
states: 43  rewrites: 2790 in 20ms cpu (20ms real) (139500
   rewrites/second)
C:Configuration -->
< ob('SameCondTest0) : Qu | Ev: none,Keep: 1 >
...

< ob('SameCondTest0) : Ob | Cl: 'SameCondTest # 0,Pr: continue(1),
  PrQ: (await 1 ? ; 1 ?(nil)),no,Lvar:
  ('caller : ob('SameCondTest0)),'label : int(1), Att:
  ('this : ob('SameCondTest0)),('cond : bool(true)),
  ('error : bool(false)), ('a : bool(true)),('b : bool(true)),
  'c : bool(true),Lcnt: 2 >

No more solutions.
states: 43  rewrites: 2790 in 20ms cpu (20ms real) (139500
   rewrites/second)
```

6.2 Setninger sammensatt av &, [] og |||

Som vi gjorde klart i 5.4.2 på side 57, kan sammensatte setninger bestå av en vilkårlig kompleks nesting av &-, []- og |||-setninger. Vi trenger et programeksempel som viser at kjøring av slike setninger går bra.

6.2.1 Creol

Vi skriver et Creol-program som inneholder en komplekst sammensatt &-setning:

class Complex
begin
var a: Bool=true, b: Bool=true, c: Bool=true, d: Bool=true, e: Bool=true .
  op run() == a := false & (b := false ||| (c := false [] (d := false & e := false))) .
end

6.2.2 Forventet resultat

Vi lager en initialtilstand der alle variablene har verdien true. På grunn av at []-operatoren gjør et ikke-deterministisk valg, forventer vi to sluttilstander. I den ene vil alle variablene være false unntatt c. I den andre vil alle variablene være false unntatt d og e.

6.2.3 CMC

Oversettelsen fra Creol til CMC gir følgende CMC-kode:

```
in v3-e-syncmerge.maude

mod COMPLEX is including SYNCMERGE .

op init : -> Configuration .
eq init =

< 'Complex : Cl | Vs: 0, Inh: nil,
Att: ('a : bool(true) ), ('b : bool(true)), ('c : bool(true)),
  ('d : bool(true)), ('e : bool(true)),
Mtds: < 'run : Mtdname | Latt: no, Code:
  (('a := bool(false)) & (('b := bool(false)) |||
  (('c := bool(false)) [] (('d := bool(false)) &
  ('e := bool(false))))); end(nil) >
, Ocnt: 1 >

new 'Complex (nil)
.
endm
```

6.2.4 Analyse

Resultatet fra søket er som forventet. Nedenfor vises utskriften fra Maude. Dette indikerer at implementasjonen av synkronisert fletting håndterer komplekst sammensatte setninger.

```
search in COMPLEX : init =>! C:Configuration .

Solution 1 (state 40)
states: 42  rewrites: 1444 in 7ms cpu (7ms real) (180522
   rewrites/second)
C:Configuration -->
< ob('Complex1) : Qu | Ev: none,Keep: empty >
...

< ob('Complex1) : Ob | Cl: 'Complex # 0,Pr: empty,
  PrQ: none,Lvar: no,Att: ('this : ob('Complex1)),
  ('a : bool(false)),('b : bool(false)),('c : bool(false)),
  ('d : bool(true)),'e : bool(true),Lcnt: 2 >


Solution 2 (state 43)
states: 44  rewrites: 1466 in 9ms cpu (9ms real) (146629
   rewrites/second)
C:Configuration -->
< ob('Complex1) : Qu | Ev: none,Keep: empty >
...

< ob('Complex1) : Ob | Cl: 'Complex # 0,Pr: empty,
  PrQ: none,Lvar: no,Att: ('this : ob('Complex1)),
  ('a : bool(false)),('b : bool(false)),('c : bool(true)),
  ('d : bool(false)),'e : bool(false),Lcnt: 2 >

No more solutions.
states: 44  rewrites: 1466 in 10ms cpu (10ms real) (133296
   rewrites/second)
```

6.3 Dyp binding i kontekst av synkronisert fletting

Dyp binding betegner den serien av bindinger av metodekall som kreves for å ekspandere kall som forekommer i koden til ekspanderte kall. Dette programeksemplet har tre poenger ved seg

1. Vi har metoder som kaller metoder.
2. Vi har mange ledd på flere nivåer. Bindingsmekanismen må traversere et tre av setningsledd som er både dypt og bredt.
3. Vi får sjekket at &-operatoren tar hensyn til metodevakter som befinner seg langt nede i aggregatet av vakter når vi kjører.

6.3.1 Creol

Jeg bruker setningen *await true*, som i seg selv ikke gjør noe som helst. I dette eksemplet fungerer den som en vakt som &-operatoren må ta med i betraktningen, og som er med på å vise at synkronisert fletting klarer å håndtere mange vakter.

```
class SM
begin
var v11: Bool=true, v21a: Bool=true, v21c: Bool=true,
        v21d: Bool=true, v22a: Bool=true, v22b: Bool=true, v22c: Bool=true,
        v23: Bool=true, finished: Bool .
  op run() == finished := false;
        m1() & m2() & await true;
        finished := true .
with Any
  op m1() == m11() & await true .
  op m11() == v11 := false .
  op m2() == m21a() & m21b() & m21c() & m21d() .
  op m21a() == v21a := false .
  op m21b() == m22a() & m22b() & m22c() & m22d() .
  op m21c() == v21c := false .
  op m21d() == v21d := false .
  op m22a() == v22a := false .
  op m22b() == v22b := false .
  op m22c() == v22c := false .
  op m22d() == m23() & await true .
  op m23() == await true;
        v23 := false .
end
```

6.3.2 Forventet resultat

Initielt er alle boolske variabler unntatt finished true. Vi bygger et aggregat av vakter, som er beredt. Systemet terminerer i en tilstand hvor alle variablene unntatt finished er false og hvor finished er true. Det interessante med dette eksemplet er å se hva som skjer hvis vi forandrer beredtheten til en av metodevaktene. Vi endrer await-setningen i m23() til å bli *await false*. Da vil systemet terminere i en tilstand hvor prosessen til &-setningen er suspendert.

6.3.3 CMC

Dette er CMC-oversettelsen av Creol-koden ovenfor:

```
mod SYNCMERGE-TEST is
including SYNCMERGE .

op init : -> Configuration .

eq init =

new 'SM(nil)

< 'SM : Cl | Vs: 0, Inh: nil,

Att: ('v11 : bool(true)), ('v21a : bool(true)),
  ('v21c : bool(true)), ('v21d : bool(true)), ('v22a : bool(true)),
  ('v22b : bool(true)), ('v22c : bool(true)), ('v23 : bool(true)),
  ('finished : null),

Mtds: < 'run : Mtdname | Latt: no, Code: ('finished := bool(false));
  (('m1(nil ; nil)) & ('m2(nil ; nil)) & (await bool(true)));
  ('finished := bool(true)); end(nil) >

* < 'm1 : Mtdname | Latt: no, Code: (('m11(nil ; nil)) &
 (await bool(true))); end(nil) >

* < 'm11 : Mtdname | Latt: no, Code: ('v11 := bool(false));
  end(nil) >

* < 'm2 : Mtdname | Latt: no, Code: (('m21a(nil ; nil)) &
  ('m21b(nil ; nil)) & ('m21c(nil ; nil)) & ('m21d(nil ; nil)));
  end(nil) >

* < 'm21a : Mtdname | Latt: no, Code: ('v21a := bool(false));
  end(nil) >

* < 'm21b : Mtdname | Latt: no, Code: (('m22a(nil ; nil)) &
 ('m22b(nil ; nil)) & ('m22c(nil ; nil)) &
 ('m22d(nil ; nil))); end(nil) >

* < 'm21c : Mtdname | Latt: no, Code: ('v21c := bool(false));
  end(nil) >

* < 'm21d : Mtdname | Latt: no, Code: ('v21d := bool(false));
  end(nil) >

* < 'm22a : Mtdname | Latt: no, Code: ('v22a := bool(false));
  end(nil) >

* < 'm22b : Mtdname | Latt: no, Code: ('v22b := bool(false));
  end(nil) >

* < 'm22c : Mtdname | Latt: no, Code: ('v22c := bool(false));
  end(nil) >
```

```
* < 'm22d : Mtdname | Latt: no, Code: (('m23(nil ; nil)) &
 (await bool(true))); end(nil) >

* < 'm23 : Mtdname | Latt: no, Code: (await bool(true));
 ('v23 := bool(false)); end(nil) > *** Test med å veksle mellom true
 og false i await-setningen her.

, Ocnt: 1 >
.
endm
```

6.3.4 Analyse

Søk med *await bool*($true$)

Her ser vi resultatet av søket fra den første initialtilstanden, som er definert ovenfor. I denne initialtilstanden innledes metoden m23() med setningen *await bool*($true$). Resultatet av søket er som forventet.

```
Solution 1 (state 124)
states: 125  rewrites: 13143 in 140ms cpu (140ms real) (93878
  rewrites/second)
C:Configuration -->
< ob('SM1) : Qu | Ev: none,Keep: empty >
...

< ob('SM1) : Ob | Cl: 'SM # 0,Pr: empty,PrQ: none,Lvar: no,
  Att: ('this : ob('SM1)), ('v11 : bool(false)),
  ('v21a : bool(false)),('v21c : bool(false)),
  ('v21d : bool(false)),('v22a : bool(false)),
  ('v22b : bool(false)),('v22c : bool(false)),
  ('v23 : bool(false)),'finished : bool(true),Lcnt: 14 >

No more solutions.
states: 125  rewrites: 13143 in 140ms cpu (140ms real) (93878
  rewrites/second)
```

Søk med *await bool*($false$)

Her ser vi resultatet av søket fra den samme initialtilstanden, men hvor vi har endret verdien på en av vaktene. Metoden m23() innledes nå med setningn *await bool*($false$). Resultatet er som forventet: &-setningen blir ikke kjørt. Vi tar oss ikke bryet med å finlese dette resultatet; det viktige er å se at objektet har terminert i en tilstand hvor prosessen ligger i PrQ.

```
Solution 1 (state 34)
states: 35  rewrites: 2991 in 30ms cpu (30ms real) (99700
  rewrites/second)
C:Configuration -->
< ob('SM1) : Qu | Ev: none,Keep: 1 >
...

< ob('SM1) : Ob | Cl: 'SM # 0,Pr: empty,PrQ: ((await 1 ? ;
  1 ?(nil)),no) : ((run(await bool(true)) &
  run(runMtd(((run(await bool(true)) & run(runMtd((('v11 :=
  bool(false)) ; end(nil)),('caller : ob('SM1)),'label : int(3)) ;
  3 ?(nil))) ; end(nil)),('caller : ob('SM1)),'label : int(2)) ;
  2 ?(nil)) & run(runMtd(((run(runMtd(((run(runMtd(((run(await
```

```
  bool(true)) & run(runMtd((await bool(false) ; ('v23 :=
  bool(false)) ; end(nil)),('caller : ob('SM1)), 'label : int(11));
  11 ?(nil))) ; end(nil)),('caller : ob('SM1)),'label : int(10)) ;
  10 ?(nil)) & run(runMtd((('v22a := bool(false)) ; end(nil)),
  ('caller : ob('SM1)), 'label : int(7)) ; 7 ?(nil)) &
  run(runMtd((('v22b := bool(false)) ; end(nil)), ('caller :
  ob('SM1)),'label : int(8)) ; 8 ?(nil)) & run(runMtd((('v22c :=
  bool(false)) ; end(nil)),('caller : ob('SM1)),'label : int(9));
  9 ?(nil))) ; end(nil)),('caller : ob('SM1)), 'label : int(6)) ;
  6 ?(nil)) & run(runMtd((('v21a := bool(false)) ; end(nil)),
  ('caller : ob('SM1)),'label : int(5)) ; 5 ?(nil)) &
  run(runMtd((('v21c := bool(false)) ; end(nil)),
  ('caller : ob('SM1)), 'label : int(12)) ; 12 ?(nil)) &
  run(runMtd((('v21d := bool(false)) ; end(nil)),
  ('caller : ob('SM1)),'label : int(13)) ; 13 ?(nil))) ; end(nil)),
  ('caller : ob('SM1)),'label : int(4)) ; 4 ?(nil))) ;
  ('finished := bool(true)) ; end(nil) ; continue(1)),
  ('caller : ob('SM1)),'label : int(1),Lvar: no,
  Att: ('this : ob('SM1)),('v11 : bool(true)),('v21a : bool(true)),
  ('v21c : bool(true)),('v21d : bool(true)), ('v22a : bool(true)),
  ('v22b : bool(true)),('v22c : bool(true)),('v23 : bool(true)),
  'finished : bool(false),Lcnt: 14 >

No more solutions.
states: 35  rewrites: 2991 in 30ms cpu (40ms real) (99700
  rewrites/second)
```

Konklusjon

Søkene fra disse to initialtilstandene sammen bidrar til å sannsynliggjøre at implementasjonen av synkronisert fletting oppfyller kravspesifikasjonen med hensyn på ekspansjon av kall.

6.4 Arveanomali

Endelig skal vi se på arveanomalieksemplene som ble presentert tidligere i seksjon 3.3. Her har jeg implementert disse eksemplene med minst mulig forretningskode. Det er to grunner til dette. For det første er forretningskoden unødvendig for vårt formål. For det andre vil den gjøre modellen mer kompleks i betydningen at den vil føre til at antall systemtilstander som er oppnåelige fra en initialtilstand blir høyere enn hvis vi ikke har forretningskode. Dette kan hemme analysen vår, som vi snart skal se et eksempel på.

6.4.1 Creol

Vi definerer En klasse Buffer, 2Buffer, GBuffer og MutexBuffer som arver en klasse Lock, for å framprovosere alle de tre formene for arveanomali. Videre definerer vi en produsentklasse som gjør put-operasjoner på en buffer, og tre ulike konsumentklasser som gjør ulike get-operasjoner på en buffer. Hver konsumentklasse kaller hver sin metode get(), get2() eller gget().

```
class Buffer(bufferSize: Nat)
begin var noElmts: Nat=0 .
with Any
  op put() == await noElmts < bufferSize;
       noElmts := noElmts + 1 .
```

```
  op get() == await noElmts > 0;
       noElmts := noElmts - 1 .
end

class 2Buffer(bufferSize: Nat) inherits Buffer()
begin
with Any
  op get2() === await noElmts >= 2;
       get@Buffer();
       get@Buffer() .
end

class GBuffer(bufferSize: Nat) inherits 2Buffer()
begin var afterGet: Bool=false .
with Any
  op put() == put@Buffer();
       afterGet := false .
  op get() == get@Buffer();
       afterGet := true .
  op get2() == get2@2Buffer();
       afterGet := true .
  op gget() == (await not afterGet) & get@Buffer();
       afterGet := true .
end

class Lock
begin var lock: Oid=nullptr, noHasLock: Nat=0 .
  op waitFree() == await lock = nullptr .
  op waitSync(in origin: Oid) == await lock = origin .
with Any
  op lock(in origin: Oid) == waitFree@Lock();
       lock := origin;
       noHasLock := noHasLock + 1 .
  op unlock(in origin: Oid) == waitSync@Lock(origin;);
       lock := nullptr;
       noHasLock := noHasLock - 1 .
end

class MutexBuffer(bufferSize: Nat) inherits GBuffer(), Lock()
begin
with Any
  op put() == lock@Lock(caller;) & put@GBuffer();
       unlock@Lock(caller;) .
  op get() == lock@Lock(caller;) & get@GBuffer();
       unlock@Lock(caller;) .
  op get2() == lock@Lock(caller;) & get2@GBuffer();
       unlock@Lock(caller;) .
  op gget() == lock@Lock(caller;) & gget@GBuffer();
       unlock@Lock(caller;) .
```

```
end

class Producer(buffer: Oid, willPut: Nat)
begin
  op run() == while willPut > 0 do
       buffer.put();
       willPut := willPut - 1
       od .
end

class ConsumerGet(buffer: Oid, willGet: Nat)
begin
  op run() == while willGet > 0 do
       buffer.get();
       willget := willGet - 1
       od .
end

class ConsumerGet2(buffer: Oid, willGet2: Nat)
begin
  op run() == while willGet2 > 0 do
       buffer.get2();
       willget2 := willGet2 - 1
       od .
end

class ConsumerGget(buffer: Oid, willGget: Nat)
begin
  op run() == while willGget > 0 do
       buffer.gget();
       willGget := willGget - 1
       od .
end
```

6.4.2 Forventet resultat

Anta at produsenten gjør like mange put-operasjoner som konsumentene sammenlagt gjør get-operasjoner. Da forventer vi at enhver sluttilstand er slik at bufferen sin variabel noElmts er 0, produsentens variabel willPut er 0 og alle konsumentenes willGet-variabler er 0.

6.4.3 CMC

Creol-koden oversettes til en initialtilstand som inneholder følgende klassedefinisjoner:

```
< 'Buffer : Cl | Vs: 0, Inh: nil,
Att: ('bufferSize : null), ('noElmts : int(0)),
Mtds: < 'put : Mtdname | Latt: no, Code:
  (await ('noElmts < 'bufferSize));
  ('noElmts := 'noElmts + int(1)); end(nil) >
 * < 'get : Mtdname | Latt: no, Code:
  (await ('noElmts > int(0)));
```

```
  ('noElmts := 'noElmts - int(1)); end(nil) >
, Ocnt: 1 >

< '2Buffer : Cl | Vs: 0, Inh: ('Buffer # 0 [ nil ]),
Att: ('bufferSize : null),
Mtds: < 'get2 : Mtdname | Latt: no, Code:
  (await ('noElmts >= int(2))); ('get @ 'Buffer (nil ; nil));
  ('get @ 'Buffer (nil ; nil)); end(nil) >
, Ocnt: 1 >

< 'GBuffer : Cl | Vs: 0, Inh: ('2Buffer # 0 [ nil ]),
Att: ('bufferSize : null), ('afterGet : bool(false)),
Mtds: < 'put : Mtdname | Latt: no, Code:
  ('put @ 'Buffer(nil ; nil)); ('afterGet := bool(false));
  end(nil) >
 * < 'get : Mtdname | Latt: no, Code:
  ('get @ 'Buffer(nil ; nil)); ('afterGet := bool(true));
  end(nil) >
 * < 'get2 : Mtdname | Latt: no, Code:
  ('get2 @ '2Buffer(nil ; nil)); ('afterGet := bool(true));
  end(nil) >
 * < 'gget : Mtdname | Latt: no, Code: ((await (not 'afterGet)) &
  ('get @ 'Buffer(nil ; nil))); ('afterGet := bool(true));
  end(nil) >
, Ocnt: 1 >

< 'Lock : Cl | Vs: 0, Inh: nil,
Att: ('lock : nullptr), ('noHasLock : int(0)),
Mtds: < 'waitFree : Mtdname | Latt: no, Code:
  (await ('lock = nullptr)); end(nil) >
 * < 'waitSync : Mtdname | Latt: ('origin : null), Code:
  (await ('lock = 'origin)); end(nil) >
 * < 'lock : Mtdname | Latt: ('origin : nullptr), Code:
  ('waitFree @ 'Lock(nil ; nil)); ('lock := 'origin);
  ('noHasLock := 'noHasLock + int(1)); end(nil) >
 * < 'unlock : Mtdname | Latt: ('origin : null), Code:
  ('waitSync @ 'Lock('origin ; nil)); ('lock := nullptr);
  ('noHasLock := 'noHasLock - int(1)); end(nil) >
, Ocnt: 1 >

< 'MutexBuffer : Cl | Vs: 0, Inh: ('GBuffer # 0 [ nil ])
  ('Lock # 0 [ nil ]),
Att: ('bufferSize : null),
Mtds: < 'put : Mtdname | Latt: no, Code:
  (('lock @ 'Lock('caller ; nil)) & ('put @ 'GBuffer(nil ; nil)));
  ('unlock @ 'Lock('caller ; nil)); end(nil) >
 * < 'get : Mtdname | Latt: no, Code:
  (('lock @ 'Lock('caller ; nil)) & ('get @ 'GBuffer(nil ; nil)));
  ('unlock @ 'Lock('caller ; nil)); end(nil) >
 * < 'get2 : Mtdname | Latt: no, Code:
  (('lock @ 'Lock('caller ; nil)) & ('get2 @ 'GBuffer(nil ; nil)));
  ('unlock @ 'Lock('caller ; nil)); end(nil) >
 * < 'gget : Mtdname | Latt: no, Code:
  (('lock @ 'Lock('caller ; nil)) & ('gget @ 'GBuffer(nil ; nil)));
  ('unlock @ 'Lock('caller ; nil)); end(nil) >
```

```
, Ocnt: 1 >

< 'Producer : Cl | Vs: 0, Inh: nil,
Att: ('buffer : null), ('willPut : null),
Mtds: < 'run : Mtdname | Latt: no, Code:
  (while ('willPut > int(0)) do ('buffer . 'put(nil ; nil));
  ('willPut := 'willPut - int(1)) od); end(nil) >
, Ocnt: 1 >

< 'ConsumerGet : Cl | Vs: 0, Inh: nil,
Att: ('buffer : null), ('willGet : null),
Mtds: < 'run : Mtdname | Latt: no, Code:
  (while ('willGet > int(0)) do ('buffer . 'get(nil ; nil));
  ('willGet := 'willGet - int(1)) od); end(nil) >
, Ocnt: 1 >

< 'ConsumerGet2 : Cl | Vs: 0, Inh: nil,
Att: ('buffer : null), ('willGet2 : null),
Mtds: < 'run : Mtdname | Latt: no, Code:
  (while ('willGet2 > int(0)) do ('buffer . 'get2(nil ; nil));
  ('willGet2 := 'willGet2 - int(1)) od); end(nil) >
, Ocnt: 1 >

< 'ConsumerGget : Cl | Vs: 0, Inh: nil,
Att: ('buffer : null), ('willGget : null),
Mtds: < 'run : Mtdname | Latt: no, Code:
  (while ('willGget > int(0)) do ('buffer . 'gget(nil ; nil));
  ('willGget := 'willGget - int(1)) od); end(nil) >
, Ocnt: 1 >
```

6.4.4 Analyse

Initialtilstand 1

I tillegg til klassene som er definert ovenfor, inneholder initialtilstanden følgende setninger for instansiering av objekter:

```
new 'MutexBuffer (int(5))                    *** Bufferens størrelse initieres til 5
new 'Producer (ob('MutexBuffer1) int(12))    *** Vi gjør 12 put-operasjoner
new 'ConsumerGet (ob('MutexBuffer1) int(3))  *** Vi gjør 3 get-operasjoner
new 'ConsumerGet2 (ob('MutexBuffer1) int(3)) *** Vi gjør 3 get2-operasjoner (=6 get)
new 'ConsumerGget (ob('MutexBuffer1) int(3)) *** Vi gjør 3 gget-operasjoner
```

Parametrene i new-setningene blir tilordnet de respektive objektattributtene (variablene i Att-listen i klassedefinisjonene) ved objektinstansiering.

Simulering fra initialtilstand 1

Vi kjører kommandoen `frew init =>! C:Configuration .` Vi bruker frew istedenfor rew fordi da blir reglene forsøkt brukt på en mer variert måte. Simulering med frew representerer en litt mer realistisk eksekvering. Resultatet er en konfigurasjon hvor objektene har følgende tilstand:

```
< ob('ConsumerGet1) : Ob | Cl: 'ConsumerGet # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGet1)),
  ('buffer : ob('MutexBuffer1)), 'willGet : int(0),Lcnt: 5 >

< ob('ConsumerGet21) : Ob | Cl: 'ConsumerGet2 # 0,Pr: empty,
```

```
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGet21)),
  ('buffer : ob('MutexBuffer1)), 'willGet2 : int(0),Lcnt: 5 >

< ob('ConsumerGget1) : Ob | Cl: 'ConsumerGget # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGget1)),
  ('buffer : ob('MutexBuffer1)), 'willGget : int(0),Lcnt: 5 >

< ob('MutexBuffer1) : Ob | Cl: 'MutexBuffer # 0,Pr: empty,
  PrQ: none, Lvar: ('caller : ob('ConsumerGet21)),'label : int(4),
  Att: ('this : ob('MutexBuffer1)),('lock : nullptr),
  ('noHasLock : int(0)), ('bufferSize : int(5)),
  ('noElmts : int(0)),'afterGet : bool(true),Lcnt: 134 >

< ob('Producer1) : Ob | Cl: 'Producer # 0,Pr: empty,
  PrQ: none, Lvar: no, Att: ('this : ob('Producer1)),
  ('buffer : ob('MutexBuffer1)), 'willPut : int(0),Lcnt: 14 >
```

Vi ser at alle objektene har Pr lik empty, og ingen objekter har ventende prosesser i PrQ. Det indikerer at objektene har klart å fullføre alle sine operasjoner. Vi ser at produsentens attributt willPut er 0, hvilket betyr at alle put-operasjoner som skal bli gjort har blitt gjort. Vi ser at konsumentenes attributter willGet, willGet2 og willGget er 0, hvilket betyr at de tre typene get-operasjoner har blitt utført alle de gangene de bør. Vi ser at bufferens attributt noElmts er 0, hvilket betyr at de elementene som har blitt puttet i bufferen har blitt hentet ut igjen. Resultatene av simuleringen bidrar til å sannsynliggjøre at implementasjonen av synkronisert fletting er korrekt.

Søk fra initialtilstand 1

Simulering med frew-kommandoen viser bare resultatet av én eksekveringshistorie, så for å se om de andre kjøringene som systemspesifikasjonene tillater også fører til korrekte sluttilstander, må vi bruke search-kommandoen. Vi gjør et søk med kommandoen
`search init =>! C:Configuration .` Hvis søket terminerer, får vi en opplisting av alle sluttilstander. Hvis alle sluttilstander er korrekte, har vi sannsynliggjort ytterligere at implementasjonen av synkronisert fletting er korrekt.

Det viser seg at søket ikke terminerer. Nærmere bestemt har søket i skrivende stund pågått i seks timer på en dedikert maskin. Hva sier dette oss? I teorien vet vi at hvis et søk aldri terminerer, så er det fordi systemspesifikasjonen er ikke-terminerende. I praksis kan vi selvfølgelig ikke observere et system i uendelig tid. Når vi har et ikke-terminerende søk etter begrenset tid, vet vi at det enten finnes veldig mange løsninger, at vi trenger veldig lang tid på å komme fram til én av løsningene eller at spesifikasjonen er ikke-terminerende på grunn av en feil i implementasjonen. Konklusjonen er at dette søket har hjulpet oss lite med å sannsynliggjøre at mekanismen for synkronisert fletting virker som antatt.

Søk fra andre initialtilstander

Fra erfaring med Maude-søk vet jeg at det ikke er uvanlig at søk ikke terminerer innen rimelig tid på grunn av at antall tilstander som systemet kan komme til fra initialtilstanden er for høyt. Derfor jobber vi videre ut i fra antagelsen om at spesifikasjonen er terminerende. Vi kan redusere antall oppnåelige tilstander ved å endre initialtilstanden. Vi ser hva som skjer når vi fjerner ett av de tre objektene og i tillegg reduserer antallet operasjoner som hvert objekt utfører. Vi kan lett endre initialtilstanden ved å gi andre parametre i new-setningene. Klassedefinisjonene er uendrede.

Vi har tre ulike objekter. Vi anstrenger oss for å være minst mulig forutinntatt i analysen, og holder muligheten åpen for at kombinasjonen av de tre ulike objektene i ett system kan være årsak til problemer. Derfor hadde det vært ønskelig å analysere en initialtilstand hvor alle de tre konsumentobjektene forekommer. For å kompensere for dette, angir vi tre starttilstander som har ulike sammensetninger av konsumentobjekter. På denne måten får vi søkt i systemer hvor hvert enkelt konsumentobjekt forekommer sammen med hvert av de andre objektene. Hvis søkene gir de forventede resultatene, har vi sannsynliggjort at kombinasjonen av de ulike objektene ikke fører til problemer.

Initialtilstand 2:

```
new 'MutexBuffer (int(4))
new 'Producer (ob('MutexBuffer1) int(6))
new 'ConsumerGet (ob('MutexBuffer1) int(2))
new 'ConsumerGet2 (ob('MutexBuffer1) int(2))
```

Initialtilstand 3a:

```
new 'MutexBuffer (int(4))
new 'Producer (ob('MutexBuffer1) int(6))
new 'ConsumerGet (ob('MutexBuffer1) int(3))
new 'ConsumerGget (ob('MutexBuffer1) int(3))
```

Initialtilstand 4a:

```
new 'MutexBuffer (int(4))
new 'Producer (ob('MutexBuffer1) int(6))
new 'ConsumerGet2 (ob('MutexBuffer1) int(2))
new 'ConsumerGget (ob('MutexBuffer1) int(2))
```

Resultat initialtilstand 2

Søket fra initialtilstand 2 terminerer etter omtrent to timer. Vi får to sluttilstander. Jeg viser her bare den delen av konfigurasjonen som består av termene som representer objektene. Det er fordi de er de eneste relevante.

```
Solution 1 (state 1188784)
states: 1188786  rewrites: 126389693 in 4692300ms cpu (5859450ms
  real) (26935 rewrites/second)
CO -->
...

< ob('ConsumerGet1) : Ob | Cl: 'ConsumerGet # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGet1)),
  ('buffer : ob('MutexBuffer1)),'willGet : int(0),Lcnt: 4 >

< ob('ConsumerGet21) : Ob | Cl: 'ConsumerGet2 # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGet21)),
  ('buffer : ob('MutexBuffer1)),'willGet2 : int(0),Lcnt: 4 >

< ob('MutexBuffer1) : Ob | Cl: 'MutexBuffer # 0,Pr: empty,
  PrQ: none,Lvar: ('caller : ob('ConsumerGet21)),'label : int(3),
  Att: ('this : ob('MutexBuffer1)),('lock : nullptr),
  ('noHasLock : int(0)),('bufferSize : int(4)),('noElmts : int(0)),
  'afterGet : bool(true),Lcnt: 66 >
```

```
< ob('Producer1) : Ob | Cl: 'Producer # 0,Pr: empty,
  PrQ: none,Lvar: no,Att: ('this : ob('Producer1)),
  ('buffer : ob('MutexBuffer1)),'willPut : int(0), Lcnt: 8 >


Solution 2 (state 1188785)
states: 1188786  rewrites: 126389705 in 4692300ms cpu (5859790ms
  real) (26935 rewrites/second)
CO -->
...

< ob('ConsumerGet1) : Ob | Cl: 'ConsumerGet # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGet1)),
  ('buffer : ob('MutexBuffer1)),'willGet : int(0),Lcnt: 4 >

< ob('ConsumerGet21) : Ob | Cl: 'ConsumerGet2 # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGet21)),
  ('buffer : ob('MutexBuffer1)),'willGet2 : int(0),Lcnt: 4 >

< ob('MutexBuffer1) : Ob | Cl: 'MutexBuffer # 0,Pr: empty,
  PrQ: none,Lvar: ('caller : ob('ConsumerGet1)),'label : int(3),
  Att: ('this : ob('MutexBuffer1)),('lock : nullptr),
  ('noHasLock : int(0)),('bufferSize : int(4)),('noElmts : int(0)),
  'afterGet : bool(true),Lcnt: 66 >

< ob('Producer1) : Ob | Cl: 'Producer # 0,Pr: empty,
  PrQ: none,Lvar: no,Att: ('this : ob('Producer1)),
  ('buffer : ob('MutexBuffer1)),'willPut : int(0), Lcnt: 8 >


No more solutions.
states: 1188786  rewrites: 126389705 in 4692310ms cpu (5859800ms
  real) (26935 rewrites/second)
```

Det første vi ser på er om de kritiske variablene har riktige verdier: Vi ser at produsentens willPut-variabel er 0, at konsumentenes willGet-variabler er 0 og at bufferens noElmts-variabel er 0. Dette tyder på at alt har gått som det skal, og at alle mulige kjøringer fra den gitte initialtilstanden gir korrekt resultat. Men hvorfor er det flere sluttilstander?

Det kan være vanskelig å se forskjellen på tilstandene manuelt, fordi de består av mange og store termer og utgjør mye tekst på en form som er vanskelig å lese. Jeg fant forskjellen ved å bruke unix-kommandoen "diff", som sammenligner to tekstfiler. Årsaken til at vi har to tilstander er at rekkefølgen på get- og get2-operasjonene kan variere. Variabelen caller i buffer-objektet inneholder objektnavnet til det objektet som sist kalte en metode i bufferen. I den første sluttilstanden ser vi at det er objektet ConsumerGet21 som gjorde det siste kallet, og i den andre sluttilstanden gjorde ConsumerGet1 det siste kallet. Grunnen til at vi har flere sluttilstander er altså ikke at noe er galt.

Resultat initialtilstand 3a

Det tar omtrent seks timer å fullføre søket fra initialtilstand 3, og resultatet består av 51 sluttilstander. Dette er mye mer enn forventet, og det er for mange tilstander til at vi kan sammenligne dem manuelt. Er det mulig å søke på en annen måte, slik at vi får et resultat som er oversiktelig nok til å lære noe om hvordan systemet virker?

Ved å ta et raskt overblikk over resultatet, ser vi at ikke alle gget-operasjonene har blitt utført i mange av tilstandene. Vi kan lese at i de fem første sluttilstandene har vi disse variabelverdiene:

$$\begin{array}{ll} Producer1.willPut & \mapsto 0 \\ MutexBuffer1.noElmts & \mapsto 3 \\ MutexBuffer1.afterGet & \mapsto true \\ ConsumerGet1.willGet & \mapsto 0 \\ ConsumerGget1.willGget & \mapsto 3 \end{array}$$

Forskjellen på disse fem tilstandene ser ut til å være i PrQ-attributtene i bufferen og gget-konsumenten.

Det ser ut som at mange av sluttilstandene skyldes at gget-operasjonene ikke blir utført fordi synkroniseringsbetingelsen ikke er oppfylt. Betingelsen er at variabelen afterGet skal være usann, noe som krever at det gjøres en put-operasjon. Hvis vi lar produsenten gjøre flere put-operasjoner — mer bestemt like mange flere put-operasjoner som antallet gjenstående gget-operasjoner — så vil vi kunne "hjelpe" systemet til å terminere i utelukkende de tilstandene hvor alle operasjoner har blitt utført. Et søk med denne økningen i antall put-operasjoner vil forhåpentligvis resultere i betydelig færre sluttilstander.

Initialtilstand 3b:

```
new 'MutexBuffer (int(4))
new 'Producer (ob('MutexBuffer1) int(9))
new 'ConsumerGet (ob('MutexBuffer1) int(3))
new 'ConsumerGget (ob('MutexBuffer1) int(3))
```

Det viser seg at dette søket ikke klarer å terminere i løpet av omtrent ti timer. Vi gjør et nytt forsøk. Vi reduserer antallet operasjoner som utføres ytterligere, ved å redusere verdiene på parametrene i new-setningen, samtidig som vi sørger for at forholdet mellom antallet put-, get- og gget-operasjoner er slik at alle put-, get- og gget-operasjoner bør bli fullført.

Initialtilstand 3c:

```
new 'MutexBuffer (int(3))
new 'Producer (ob('MutexBuffer1) int(6))
new 'ConsumerGet (ob('MutexBuffer1) int(2))
new 'ConsumerGget (ob('MutexBuffer1) int(2))
```

Søket terminerer på mindre enn to timer, med tre sluttilstander. Alle sluttilstandene er akseptable, som vi ser i følgende tabell over variabelverdier i de ulike sluttilstandene. Variabelen caller forteller hvilket objekt som sist gjorde et kall på en av bufferens metoder.

Sluttilstand nummer	**1**	**2**	**3**
$Producer1.willPut$	0	0	0
$MutexBuffer1.noElmts$	2	2	2
$MutexBuffer1.afterGet$	$false$	$true$	$true$
$MutexBuffer1.caller$	$Producer1$	$ConsumerGet1$	$ConsumerGget1$
$ConsumerGet1.willGet$	0	0	0
$ConsumerGget1.willGget$	0	0	0

Her vises utskriften fra søket:

```
search in INHANOMALY : init =>! CO .

Solution 1 (state 1074649)
states: 1074652  rewrites: 129127561 in 4374550ms cpu (4965510ms
  real) (29517 rewrites/second)
CO -->
...

< ob('ConsumerGet1) : Ob | Cl: 'ConsumerGet # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGet1)),
  ('buffer : ob('MutexBuffer1)),'willGet : int(0),Lcnt: 4 >

< ob('ConsumerGget1) : Ob | Cl: 'ConsumerGget # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGget1)),
  ('buffer : ob('MutexBuffer1)),'willGget : int(0),Lcnt: 4 >

< ob('MutexBuffer1) : Ob | Cl: 'MutexBuffer # 0,Pr: empty,
  PrQ: none,Lvar: ('caller : ob('Producer1)),'label : int(7),
  Att: ('this : ob('MutexBuffer1)), 'lock : nullptr),
  ('noHasLock : int(0)),('bufferSize : int(3)),
  ('noElmts : int(2)),'afterGet : bool(false),Lcnt: 62 >

< ob('Producer1) : Ob | Cl: 'Producer # 0,Pr: empty,
  PrQ: none,Lvar: no,Att: ('this : ob('Producer1)),
  ('buffer : ob('MutexBuffer1)),'willPut : int(0), Lcnt: 8 >


Solution 2 (state 1074650)
states: 1074652  rewrites: 129127573 in 4374550ms cpu (4965580ms
  real) (29517 rewrites/second)
CO -->
...

< ob('ConsumerGet1) : Ob | Cl: 'ConsumerGet # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGet1)),
  ('buffer : ob('MutexBuffer1)),'willGet : int(0),Lcnt: 4 >

< ob('ConsumerGget1) : Ob | Cl: 'ConsumerGget # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGget1)),
  ('buffer : ob('MutexBuffer1)),'willGget : int(0),Lcnt: 4 >

< ob('MutexBuffer1) : Ob | Cl: 'MutexBuffer # 0,Pr: empty,
  PrQ: none,Lvar: ('caller : ob('ConsumerGget1)),'label : int(3),
  Att: ('this : ob('MutexBuffer1)),('lock : nullptr),
  ('noHasLock : int(0)),('bufferSize : int(3)),('noElmts : int(2)),
  'afterGet : bool(true),Lcnt: 62 >

< ob('Producer1) : Ob | Cl: 'Producer # 0,Pr: empty,
  PrQ: none,Lvar: no,Att: ('this : ob('Producer1)),
  ('buffer : ob('MutexBuffer1)),'willPut : int(0), Lcnt: 8 >


Solution 3 (state 1074651)
states: 1074652  rewrites: 129127585 in 4374560ms cpu (4965580ms
  real) (29517 rewrites/second)
```

```
CO -->
...

< ob('ConsumerGet1) : Ob | Cl: 'ConsumerGet # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGet1)),
  ('buffer : ob('MutexBuffer1)),'willGet : int(0),Lcnt: 4 >

< ob('ConsumerGget1) : Ob | Cl: 'ConsumerGget # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGget1)),
  ('buffer : ob('MutexBuffer1)),'willGget : int(0),Lcnt: 4 >

< ob('MutexBuffer1) : Ob | Cl: 'MutexBuffer # 0,Pr: empty,
  PrQ: none,Lvar: ('caller : ob('ConsumerGet1)),'label : int(3),
  Att: ('this : ob('MutexBuffer1)),('lock : nullptr),
  ('noHasLock : int(0)),('bufferSize : int(3)),
  ('noElmts : int(2)),'afterGet : bool(true),Lcnt: 62 >

< ob('Producer1) : Ob | Cl: 'Producer # 0,Pr: empty,
  PrQ: none,Lvar: no,Att: ('this : ob('Producer1)),
  ('buffer : ob('MutexBuffer1)),'willPut : int(0), Lcnt: 8 >


No more solutions.
states: 1074652  rewrites: 129127585 in 4374560ms cpu (4965590ms
   real) (29517 rewrites/second)
```

Resultat initialtilstand 4a

Det tar omtrent 2,5 timer å fullføre søket fra initialtilstand 4, og resultatet består av 29 sluttilstander. Vi har det samme problemet som med initialtilstand 3: Antallet sluttilstander er for høyt, fordi gget-operasjonene ikke får kjøre. Løsningen er den samme: Vi gjør et nytt søk fra en initialtilstand som fører til flere put-operasjoner, for å "hjelpe" gget-operasjonene til å få oppfylt synkroniseringsbetingelsen sin. Vi forventer betydelig færre sluttilstander fra dette søket.

Initialtilstand 4b:

```
new 'MutexBuffer (int(4))
new 'Producer (ob('MutexBuffer1) int(8))
new 'ConsumerGet2 (ob('MutexBuffer1) int(2))
new 'ConsumerGget (ob('MutexBuffer1) int(2))
```

Dette søket terminerer ikke i løpet av omtrent ti timer. Vi prøver å redusere kompleksiteten i systemet ytterligere, samtidig som forholdet mellom antallet operasjonene er slik at alle put- get2- og gget-operasjonene bør bli utført.

Initialtilstand 4c:

```
new 'MutexBuffer (int(3))
new 'Producer (ob('MutexBuffer1) int(6))
new 'ConsumerGet2 (ob('MutexBuffer1) int(1))
new 'ConsumerGget (ob('MutexBuffer1) int(2))
```

Søket terminerer på mindre enn to timer, med to sluttilstander. Alle sluttilstandene er akseptable i forhold til det forventede resultatet, som vi ser i følgende tabell over variabelverdier i de ulike sluttil-

standene. Variabelen caller forteller hvilket objekt som sist gjorde et kall på en av bufferens metoder.

Sluttilstand nummer	**1**	**2**
$Producer1.willPut$	0	0
$MutexBuffer1.noElmts$	2	2
$MutexBuffer1.afterGet$	$false$	$true$
$MutexBuffer1.caller$	$Producer1$	$ConsumerGget1$
$ConsumerGet21.willGet2$	0	0
$ConsumerGget1.willGget$	0	0

Her vises utskriften fra søket:

```
search in INHANOMALY : init =>! CO .

Solution 1 (state 677495)
states: 677497  rewrites: 77795797 in 1928400ms cpu (2472530ms
  real) (40342 rewrites/second)
CO -->
...

< ob('ConsumerGet21) : Ob | Cl: 'ConsumerGet2 # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGet21)),
  ('buffer : ob('MutexBuffer1)),'willGet2 : int(0),Lcnt: 3 >

< ob('ConsumerGget1) : Ob | Cl: 'ConsumerGget # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGget1)),
  ('buffer : ob('MutexBuffer1)),'willGget : int(0),Lcnt: 4 >

< ob('MutexBuffer1) : Ob | Cl: 'MutexBuffer # 0,Pr: empty,
  PrQ: none,Lvar: ('caller : ob('Producer1)),'label : int(7),
  Att: ('this : ob('MutexBuffer1)), ('lock : nullptr),
  ('noHasLock : int(0)),('bufferSize : int(3)),('noElmts : int(2)),
  'afterGet : bool(false),Lcnt: 58 >

< ob('Producer1) : Ob | Cl: 'Producer # 0,Pr: empty,
  PrQ: none,Lvar: no,Att: ('this : ob('Producer1)),
  ('buffer : ob('MutexBuffer1)),'willPut : int(0), Lcnt: 8 >


Solution 2 (state 677496)
states: 677497  rewrites: 77795809 in 1928400ms cpu (2472580ms
  real) (40342 rewrites/second)
CO -->
...

< ob('ConsumerGet21) : Ob | Cl: 'ConsumerGet2 # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGet21)),
  ('buffer : ob('MutexBuffer1)),'willGet2 : int(0),Lcnt: 3 >

< ob('ConsumerGget1) : Ob | Cl: 'ConsumerGget # 0,Pr: empty,
  PrQ: none,Lvar: no, Att: ('this : ob('ConsumerGget1)),
  ('buffer : ob('MutexBuffer1)),'willGget : int(0),Lcnt: 4 >

< ob('MutexBuffer1) : Ob | Cl: 'MutexBuffer # 0,Pr: empty,
  PrQ: none,Lvar: ('caller : ob('ConsumerGget1)),'label : int(3),
  Att: ('this : ob('MutexBuffer1)),('lock : nullptr),
  ('noHasLock : int(0)),('bufferSize : int(3)),
```

```
  ('noElmts : int(2)),'afterGet : bool(true),Lcnt: 58 >

< ob('Producer1) : Ob | Cl: 'Producer # 0,Pr: empty,
  PrQ: none,Lvar: no,Att: ('this : ob('Producer1)),
  ('buffer : ob('MutexBuffer1)),'willPut : int(0), Lcnt: 8 >

No more solutions.
states: 677497  rewrites: 77795809 in 1928410ms cpu (2472590ms
   real) (40341 rewrites/second)
```

6.4.5 Resultatet av analysen av arveanomalieksemplet

Hva har vi klart å vise med analysen av programeksemplet med arveanomali? Vi ønsker å vite om implementasjonen av synkronisert fletting er korrekt. Hvis den er korrekt, kan &-operatoren brukes til å redusere problemet med arveanomali.

Simulering av et system med en buffer, en produsent og tre ulike konsumenter ved bruk av frew-kommandoen, bidrar til å sannsynliggjøre at implementasjonen er korrekt.

Analyse ved hjelp av search-kommandoen viste seg å bli vanskeligere, fordi søkene ikke terminerte eller terminerte med for mange sluttilstander. Ideelt sett skulle vi ha søkt i det samme eksemplet som vi kjørte med frew, men det var ikke mulig. Vi ble nødt til å forenkle programeksemplet, blant annet ved å redusere antallet objekter i systemet. Ved å redusere kompleksiteten, får vi søkene til å terminere. Vi har delt analysen av systemet opp i tre søk fra tre ulike starttilstander, og fått korrekte resultater. Til sammen utgjør de tre søkene en tilnærmet like god analyse som et søk fra en større initialtilstand.

Ulempen med denne framgangsmåten er at ved å søke fra enkle initaltilstander kan vi ha forenklet oss bort fra tilstander som ville ha ført til ikke-determinisme eller en ukorrekt sluttilstand. Vi har tre ulike konsumentobjekter. Ingen av de søkene som vi fikk til å terminere hadde flere enn to konsumentobjekter. Det kan tenkes at kombinasjonen av de tre konsumentene vil gi opphav til systemtilstander som ikke oppstår med noen kombinasjon av to av konsumentene. Vi har ingen spesiell grunn til å tro dette, men en av fordelene med maskinell analyse er nettopp at vi kan håpe på å oppdage feil som vi ikke forventer. Til tross for forenklingen, må vi kunne si at resultatene av analysen bidrar til å sannsynliggjøre at implementasjonen av synkronisert fletting kan brukes til å redusere problemet med arveanomali.

Kapittel 7

Oppsummering og konklusjon

Vi husker fra kapittel 1.2 på side 4 at hovedmålet for oppgaven var å ta utgangspunkt i den foreslåtte skissen av semantikken til synkronisert fletting, videreutvikle og presisere semantikken og vurdere ulike semantiske alternativer. Dette skulle resultere i at jeg utvidet interpreten med mitt forslag til operasjonell semantikk for synkronisert fletting. Jeg oppsummerer utfordringene, oppdagelsene og resultatene i arbeidet med dette. Deretter gjør jeg rede for forsøket på å bruke Maude til maskinell analyse for å gjøre en tentativ validering og verifikasjon av implementasjonen.

7.1 Utvidelse av Creol-språket med synkronisert fletting

Utgangspunktet for oppgaven var følgende uformelle definisjon av synkronisert fletting (hvor G1 og G2 er vakter, S1 og S2 vilkårlige setningslister):

> Synkronisert fletting — (**await** $G1; S1$) &(**await** $G2; S2$) — er definert som **await** $G1 \wedge G2; (S1 \| S2)$. Ikke-bevoktede argumenter til & behandles som om de er bevoktet av en sann vakt. Metodekall som er synkrone og interne ekspanderes.

Operatoren for synkronisert fletting skal la oss bygge et aggregat av frittstående vakter og metodevakter fra metoder i ulike klasser. Det er ekspansjonen av metodekall som gjør at &-operatoren kan brukes til inkrementell endring av synkroniseringsbetingelser ved arv. Ekspansjon av metodekall er også det som gjør implementeringen av synkronisert fletting vanskelig. Det har vært nødvendig å bruke mye ressurser på bindingsmekanismen i forbindelse med ekspansjon av interne, synkrone kall i kontekst av synkronisert fletting. Grunnen til dette er at den eksisterende bindingsmekanismen ikke var designet med tanke på ekspansjon av metodekall.

7.1.1 Utfordringer

Jeg identifiserte tre faser i kjøringen av en &-setning. Videre identifiserte jeg følgende utfordringer i forbindelse med hver fase:

1. Binding av eventuelle innledende metodekall i hvert setningsledd. Hvordan finne de kallene som skal bindes?
   - &-setninger kan være vilkårlig komplekst sammensatt av |||- og []-setninger, som igjen kan inneholde ledd bestående av andre &-setninger.
   - Metoder og ledd i en &-setning kan innledes med metodekall. Det betyr at vi må kunne ekspandere kall som forekommer i koden til ekspanderte kall.
2. Testing av om alle leddene i &-setningen er beredte samtidig.
   - Hvordan avgjøre om en &-setning er beredt? På en eller annen måte må hjelpefunksjonen enabled() brukes. Skal enabled() sjekke selve &-setningen eller skal den sjekke en ekstra await-setning som interpreten automatisk genererer og som inneholder alle relevante vakter?

- Metodevakter som forekommer i ekspanderte metodekall må evalueres i kontekst av de lokale variablene til metodeinstansen sin. Hvordan kan metodeinstansens lokale tilstand gjøres tilgjengelig for hjelpefunksjonen enabled()?

3. Kjøring av &-setningen etter at den har blitt vurdert til å være beredt.

   - Kontekstskifter ved kjøring av en &-setning med ekspanderte kall.
   - Semantikken til synkronisert fletting er betinget av semantikken til fletting. Hvordan må semantikken for fletting være for at vi skal få den ønskede semantikken til synkronisert fletting?

Jeg har innført noen nye setninger i CMC for å få løst de tekniske/praktiske problemene. Den viktigste av disse er runMtd-setningen. Den inneholder en metodeinstans, og gjør at interpreten har både koden og tilstanden lett tilgjengelig når den håndterer en prosess. Vi har bruk for runMtd-setningen både når vi skal sjekke om metodevakten i et ekspandert kall er beredt og ved kjøring av et ekspandert kall, når vi trenger å gjøre kontekstbytter.

Problemet med komplekst sammensatte &-setninger lar seg løse med en splitt-og-hersk-tilnærmingsmåte. Ved å lage en bindingsmekanisme som isolerer hvert setningsledd, kan vi vurdere om leddet inneholder et kall som skal bindes. Mekanismen har blitt illustrert med et trediagram, hvor en node er et setningsledd eller en metodeinstans. Mekanismen virker slik at grenene i treet vekselvis ekspanderer og folder seg sammen, inntil alle kall som skal bindes har blitt bundet.

Problemet med ekspansjon av kall som forekommer i koden til ekspanderte kall lar seg løse med kombinasjonen av splitt-og-hersk-algoritmen og introduksjonen av Element-setningen. Algoritmen gjør at vi klarer å finne kallet, og Element-setningen gjør at vi kan binde det med parametre som er evaluert i den riktige konteksten.

Se på setningen $(await\ G1; S1)\ \&\ m(inParams; outParams)$. I den opprinnelige definisjonen er det ikke antydet overhodet hvordan det ekspanderte kallet på metoden m() skal kunne sjekkes for beredthet. La m() sin metodekropp være $await\ G2; S2$. Kombinasjonen av bindingsmekanismen og introduksjonen av runMtd-setningen gjør at vi kan omforme setningen til å bli

$$(await\ G1; S1)\ \&\ runMtd(await\ G2; S2,\ L2)$$

hvor L2 er den lokale tilstanden til det ekspanderte kallet. Ved i tillegg å utvide enabled() til å takle &-setninger og runMtd-setninger, blir det enkelt å avgjøre om en &-setning med ekspanderte kall er beredt.

Alle de ovenfornevnte løsningene gjør at den uformelle definisjonen av synkronisert fletting kan endres fra

$$(await\ G1; S1)\ \&\ (await\ G2; S2)\ \rightsquigarrow\ await\ G1\ \&\ G2; S1\ |||\ S2$$

til

$$S1\ \&\ S2\ \rightsquigarrow\ S1\ |||\ S2$$

hvis &-setningen er beredt, som avgjøres med $enabled((S1\ \&\ S2,\ L, M)$

hvor S1 og S2 er vilkårlige setningssekvenser som kan starte med await-setninger eller ekspanderte metodekall (og hvor L er prosessens tilstand og M meldingene i prosessens meldingskø).

Introduksjonen av runMtd-setningen gjør i tillegg at vi får implementert kontekstskifter i forbindelse med kjøring av ekspanderte kall på en god måte.

7.1.2 Diskusjonen om semantikk

Utfordringene med bindingsmekanismen og kontekstskifter har ikke vært veldig sentrale for semantikken til synkronisert fletting. De har dreid seg hovedsaklig om tekniske/praktiske programmeringsproblemer. Mer interessant i så måte er drøftingen av ulike policyer for fletting.

Jeg reflekterte over hva som burde være tolkningen av synkronisert fletting med utgangspunkt i hva som bør være råderommet for vakter i en &-setning. Sagt på en annen måte, reflekterte jeg over hvor i koden synkroniseringsbetingelsene skulle være garantert av interpreten. Jeg identifiserte tre rimelige tolkninger av $(await\ G1; S1)\&(await\ G2; S2)$, uttrykt i en notasjon inspirert av Hoare-logikk hvor jeg abstraherer vekk await-setningen:

1. $\{G1 \wedge G2\}\ (S1 \mid\mid\mid S2)$
2. $\{G1 \wedge G2\}\ (\{G1\}S1 \mid\mid\mid \{G2\}S2)$
3. $\{G1 \wedge G2\}\ (\{G1 \wedge G2\}S1 \mid\mid\mid \{G1 \wedge G2\}S2)$

Jeg konkluderte med at tolkning nummer 2 var best.

Videre studerte jeg mekanismen for fletting og dens operator |||, siden semantikken til synkronisert fletting er betinget av semantikken til fletting. Jeg identifiserte tre mulige tolkninger av fletting. Jeg har kalt tolkningene for policyer, inspirert av likheten til operativsystemers policyer for skedulering av prosesser. Vi bruker setningen $(await\ G1; S1) \mid\mid\mid (await\ G2; S2)$ som eksempel:

Tilfeldig fletting Ved å bruke en variant av |||-operatoren som er kommutativ, vil vi aldri ha noen garanti for hvilken rekkefølge setningene i leddene i |||-setningen kjøres i. Hver gang vi skal kjøre en ny setning, kan setningen hentes fra et vilkårlig ledd. Det betyr at selv om $await\ G1$ er plassert syntaktisk umiddelbart foran S1, så kan en hvilken som helst annen setning fra |||-setningen bli kjørt mellom de to. Dermed forpurres hele poenget med vakter.

Delvis ordnet fletting Dette er den policyen som jeg har implementert. Ved å bruke en ikke-kommutativ variant av fletteoperatoren — /// —, får vi en viss garanti for rekkefølgen. Etter å ha kjørt $await\ G1$, vet vi at den neste setningen som vil bli kjørt er fra S1, så framt S1 er beredt. Policyen er bare delvis ordnet, fordi rekkefølgen på leddene i ///-setningen er vilkårlig. Grunnen til dette er at jeg har brukt en kommutativ variant av &-operatoren. Kommutativitetsegenskapen til &-operatoren gjør at før &-operatoren byttes ut med ///-operatoren, kan leddene ta en hvilken som helst rekkefølge.

Jeg har vist to eksempler på at det er alvorlige semantiske problemer med denne policyen. Vi er ikke garantert korrekt programadferd.

Ordnet fletting Ved å bruke ikke-kommutative varianter av både &- og |||-operatoren får vi det jeg har kalt ordnet fletting. Med den har vi for det første en listerekkefølge på setningsleddene. For det andre har vi den samme forutsigbarheten for kjøringen av setninger i ett ledd som delvis ordnet fletting gir. Denne policyen har ingen av problemene som jeg har oppdaget i de to andre.

Det var simulering med Maude sin frew-kommando som først gjorde meg klar over problemene med de to første policyene.

En typisk bruk av &-operatoren vil være å legge til en await-setning til et kall på en bevoktet metode. Etter at en &-setning har blitt vurdert til å være beredt, er setningsledd som bare består av én enkelt await-setning overflødige. Mekanismen for synkronisert fletting kan raffineres og rasjonaliseres ved at slike ledd fjernes før flettingen påbegynnes.

7.2 Maskinell analyse

Denne oppgavens sekundære mål var å bruke noen av Maude sine muligheter for maskinell analyse til å 1) sannsynliggjøre at implementasjonen av synkronisert fletting er feilfri og 2) vise at Creol sin strategi for inkrementell endring av synkroniseringsbetingelser kan realiseres i programmer som Creol-interpreten kan tolke.

7.2.1 Verifikasjon

Maude sine kommandoer rew, frew, search og show path har vært en del av den rutinemessige debuggingen og analysen underveis gjennom hele utviklingsprosessen. Jeg har simulert mange programeksempler som har vært designet ad hoc for å teste enkelte aspekter ved implementasjonen. I kapittel 6 velger jeg å vise analysen av tre programeksempler. De er valgt ut fordi de bidrar til å sannsynliggjøre at implementasjonen har noen sentrale og nødvendige egenskaper.

Samme-lås-eksemplet er designet for å avsløre en feil som ville ha oppstått med en variant av synkronisert fletting som tilsvarer tolkning 2 på side 104. Det andre eksemplet bidrar til å sannsynliggjøre at implementasjonen håndterer komplekst sammensatte &-setninger. Tredje eksempel bidrar til å sannsynliggjøre at implementasjonen håndterer ekspansjon av metodekall i kontekst av &-operatoren.

7.2.2 Validering

Til slutt har vi CMC-implementasjonen av de klassiske arveanomali-eksemplene. Her implementerte vi en CMC-modell av et distribuert system med en buffer som er en tjener for en produsent- og tre konsumentklienter. Produsenten legger uspesifiserte elementer i bufferen, mens konsumentene kaller ulike operasjoner i bufferen for å hente elementer ut. Jeg har designet bufferklassen slik at jeg framprovoserer arveanomali. Det gir meg anledning til å bruke Creol sin tilnærmingsmåte til å håndtere anomalien, som blant annet inkluderer bruk av synkronisert fletting.

Enkel simulering av systemet med frew-kommandoen fra en gitt starttilstand ga forventet resultat. Neste skritt var å søke etter enhver sluttilstand som var mulig å oppnå fra den samme starttilstanden. Det viste seg at dette søket ikke terminerte innen rimelig tid. Jeg antok at årsaken var en nokså vanlig årsak til ikke-terminering av Maude-søk, nemlig at antallet systemtilstander som Maude må søke i blir for høyt. Jeg designet tre initialtilstander som hver for seg var betydelig mindre komplekse, men som til sammen ville bidra til å sannsynliggjøre at alt fungerte som det skulle. Søkene ga forventet resultat.

7.3 Resultater

Et av de viktigste resultatene fra arbeidet med denne oppgaven er følgende: Jeg prøvde først å implementere tilfeldig fletting, og oppdaget at det ikke vil garantere korrekt programadferd. Så implementerte jeg delvis ordnet fletting, og oppdaget at heller ikke dét garanterer korrekt programadferd. *En implementasjon av synkronisert fletting må være basert på en implementasjon av policyen for (helt) ordnet fletting.*

Jeg har gitt en formell definisjon av synkronisert fletting. Definisjonen kan forbedres ytterligere ved først og fremst å implementere ordnet fletting. Tidsrammen som er satt for denne oppgaven hindrer meg i å gjøre dette selv.

Jeg har identifisert de viktigste utfordringene som må møtes av enhver implementasjon av synkronisert fletting som skal baseres på den gitte varianten av interpreten.

Jeg har sannsynliggjort ved maskinell analyse at implementasjonen av synkronisert fletting er feilfri og kan brukes etter hensikten, nemlig til å endre synkroniseringskode inkrementelt i distribuerte objekter ved arv.

Bibliografi

[1] Gregory R. Andrews. *Foundations of Multithreaded, Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201357526.

[2] Marte Arnestad. En abstrakt maskin for Creol i Maude. Hovedfagsoppgave, Institutt for informatikk, Universitetet i Oslo, november 2003. [Online; aksessert 15-Oktober-2005] `http://heim.ifi.uio.no/~creol`.

[3] Eyvind Wærsted Axelsen. A meta-level framework for recording and utilizing communication histories in Maude. Hovedfagsoppgave, Institutt for informatikk, Universitetet i Oslo, august 2004. [Online; aksessert 15-Oktober-2005] `http://www.duo.uio.no/`.

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

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer og Carolyn Talcott. *Maude Manual (Version 2.1.1)*. SRI International, Menlo Park, CA 94025, USA, April 2005. [Online; aksessert 26-September-2005] `http://maude.cs.uiuc.edu/maude2-manual/`.

[6] Jerome Lacoste (CoffeeBreaks). Testdrevet utvikling. Foredrag 24. februar 2005 i regi av Cybernetisk selskap, Institutt for informatikk, Universitetet i Oslo.

[7] George Coulouris, Jean Dollimore og Tim Kindberg. *Distributed Systems. Concepts and Design*. Addison Wesley, tredje utgave, 2001. ISBN 0201-61918-0.

[8] Johan Dovland, Einar Broch Johnsen og Olaf Owe. Reasoning about asynchronous method calls and inheritance. I Chunming Rong, redaktør, *Proc. of the Norwegian Informatics Conference (NIK'04)*, side 213–224. Tapir akademiske forlag, november 2004.

[9] Tzilla Elrad, Mehmet Aksits, Gregor Kiczales, Karl Lieberherr og Harold Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001. ISSN 0001-0782.

[10] Tzilla Elrad, Robert E. Filman og Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001. ISSN 0001-0782.

[11] Szabolcs Ferenczi. Guarded methods vs. inheritance anomaly: Inheritance anomaly solved by nested guarded method calls. *SIGPLAN Notices*, 30(2):49–58, 1995.

[12] Svend Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. I O. Lehrmann Madsen, redaktør, *Proceedings of ECOOP'92, Utrecht, The Netherlands*, side 185–196. Springer-Verlag, Berlin, 1992.

[13] Einar Broch Johnsen, Olaf Owe og Isabelle Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. I M. Steffen og G. Zavattaro, redaktører, *Proc. 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, bind 3535 av *Lecture Notes in Computer Science*, side 15–30. Springer-Verlag, juni 2005.

[14] Einar Broch Johnsen og Olaf Owe. An asynchronous communication model for distributed concurrent objects. I *Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, side 188–197. IEEE Computer Society Press, september 2004. En utvidet versjon er innsendt til publisering i 2005.

[15] Einar Broch Johnsen og Olaf Owe. Inheritance in the presence of asynchronous method calls. I *Proc. 38th Hawaii Intl. Conf. on System Sciences (HICSS 2005)*. IEEE Computer Society Press, januar 2005.

[16] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier og John Irwin. Aspect-oriented programming. I Mehmet Akşit og Satoshi Matsuoka, redaktører, *Proceedings European Conference on Object-Oriented Programming*, bind 1241, side 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[17] Gregor Kiczales og Mira Mezini. Aspect-oriented programming and modular reasoning. I *ICSE '05: Proceedings of the 27th international conference on Software engineering*, side 49–58, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-963-2.

[18] Ramnivas Laddad. I want my AOP! [Online; aksessert 14-September-2005] `http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html`, 2002.

[19] Satoshi Matsuoka og Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. I G. Agha, P. Wegner og A. Yonezawa, redaktører, *Research Directions in Concurrent Object-Oriented Programming*, side 107–150. MIT Press, 1993.

[20] Giuseppe Milicia og Vladimiro Sassone. The inheritance anomaly: Ten years after. I *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, side 1267–1274, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-812-1.

[21] David L. Parnas. *The secret history of information hiding*, side 399–409. Springer-Verlag New York, Inc., New York, NY, USA, 2002. ISBN 3-540-43081-4.

[22] Oxford Reference Online, Oxford University Press. Modular programming — a dictionary of computing. [Online; aksessert 15-September-2005] `http://www.oxfordreference.com/views/ENTRY.html?subview=Main&entry=t11.%e3304`, 2004.

[23] Oxford Reference Online, Oxford University Press. Scheduler — a dictionary of computing. [Online; aksessert 24-August-2005] `http://www.oxfordreference.com/views/ENTRY.html?subview=Main&entry=t11.%e4618`, 2004.

[24] Peter Csaba Ölveczky og Einar Broch Johnsen. [Online; aksessert høsten 2004] http://www.uio.no/studier/emner/matnat/ifi/INF5130/h04/undervisningsplan.xml, 2004. Forelesningsnotater fra kurset inf5130 'Utvalgte emner i omskrivingslogikk' ved Institutt for informatikk, Universitetet i Oslo.

[25] Peter Csaba Ölveczky. Formal modeling and analysis of distributed systems in Maude. [Online; aksessert våren 2003] `http://heim.ifi.uio.no/~peterol/`, 2003. Forelesningsnotater fra kurset inf220 'Formell modellering og analyse av kommuniserende systemer' ved Institutt for informatikk, Universitetet i Oslo.

[26] CREOL: A formal framework for reflective component modelling. [Online; aksessert 14-September-2005] `http://www.ifi.uio.no/~creol/`, 2005.

[27] The Maude system. [Online; aksessert 26-September-2005] `http://maude.cs.uiuc.edu/`.

Tillegg A

Samlet Maude-spesifikasjon av interpreten

Dette er Maude-implementasjonen av interpreten slik den ble overlatt til meg, med noen få unntak av tillegg og utkommenteringer av meg. De viktigste endringene mine finnes på line 274-281, 297-300 og 709-751 og er markert med tekststrengen "ARE".

De første 200 linjene inneholder definisjoner av datatyper i CMC. Vakter og setningstyper defineres i linje 227-139. Konfigurasjonen bestående av klasser, objekter, meldingskøer og meldinger defineres i 324-410. Hjelpefunksjoner i interpreten defineres i linje 414-546.

Reglene som beskriver den operasjonelle semantikken til Creol finnes først og fremst fra linje 641 og utover. Fra linje 913 finnes reglene for oppdatering av klassedefinisjoner under kjøretid, som ikke har blitt nevnt i denne oppgaven.

```
1  fmod DATA is
2  pr QID .
3  pr STRING .
4  pr INT .
5  pr FLOAT .
6
7  sorts Nil Aid Oid AidList Data DataList Call Expr List . *** List= list of Expr
8
9  subsorts Oid < Data < Expr DataList < List .
10 subsorts Qid < Aid < AidList  < List .
11 subsorts Nil < AidList DataList .
12 subsort  Call Aid < Expr .
13
14 op null :           -> Data [ctor] . *** undef. value/none pointer
15 op _[_] : Qid List  -> Call [ctor] . *** call
16 op int  : Int       -> Data [ctor] .
17 op str  : String    -> Data [ctor] .
18 op bool : Bool      -> Data [ctor] .
19 op char : Char      -> Data [ctor] .
20 op float : Float    -> Data [ctor] .
21 op pair : Data Data -> Data [ctor] .
22 op pair : Expr Expr -> Expr [ctor] .
23 op list : DataList  -> Data [ctor] .
24 op list : List      -> Expr [ctor] .
25 op set  : DataList  -> Data [ctor] .
26 op set  : List      -> Expr [ctor] .
27 op ob   : Qid       -> Oid  [ctor] .  *** for giving oid-names
28 op nullptr :        -> Oid  [ctor] . *** ARE
29
```

```
op nil  :                   -> Nil      [ctor] .
op __   : List     List     -> List     [ctor assoc id: nil] .
op __   : DataList DataList -> DataList [ctor ditto] .
op __   : AidList  AidList  -> AidList  [ctor ditto] .
op __   : Nil      Nil      -> Nil      [ctor ditto] .

op _asInt          : Expr -> Int .
op _asBool         : Expr -> Bool .
op _asStr          : Expr -> String .

var N : Nat .
var B : Bool .
var S : String .
eq int(N)  asInt  = N .  *** otherwise error...
eq bool(B) asBool = B .
eq str(S)  asStr  = S .
endfm


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

fmod DATATYPER is
pr DATA .

*** PREDEFINED INFIX NOTATION
ops _+_ _-_ _*_ _/_ _cat_ : Expr Expr -> Expr .
ops _<_ _<=_ _>_ _>=_     : Expr Expr -> Expr .
ops _or_ _and_                 : Expr Expr -> Expr .
ops _/=_ _=_                  : Expr Expr -> Expr .
ops _|-_  _-|_                 : Expr Expr -> Expr .
ops neg_  not_  #_        : Expr -> Expr .
ops _.fst _.scd           : Expr -> Expr .  *** first and second part of pair
ops _bw_ _ew_                  : Expr Expr -> Expr .

*** variables
vars S S'     : String .
vars L L' L'' : List .
vars E E'     : Expr .
vars D D'     : Data .
vars I I'     : Int .
var  B              : Bool .
vars N N'     : Nat .
vars F F'     : Float .


**** translation from infix to prefix notation
eq not E         = 'not[E].
eq neg E         = 'neg[E].
eq # E                = 'length[E].

eq E + E'         = 'plus [E E'].
eq E - E'         = 'minus [E E'].
eq E * E'         = 'times [E E'].
eq E / E'         = 'div [E E'].
eq E <  E'         = 'less[E E'].
```

```
eq E <= E'         = 'lessEq[E E'].
eq E >  E'         = 'less[E' E].
eq E >= E'         = 'lessEq[E' E].
eq (E =  E')         = 'equal[E E'].
eq E /=  E'         = 'not['equal[E E']].

eq E and E'         = 'and[E E'].
eq E or  E'         = 'or[E E'].
eq E cat E'      = 'plus [E E']. *** old string syntax, may now use +
eq E .fst         = 'fst[E]. *** pair-functions
eq E .scd         = 'scd[E].

eq E bw E'         = 'begwith[E E'].
eq E ew E'         = 'endwith[E E'].
eq E |- E'       = 'add[E E'].      *** appendright
eq E -| E'       = 'addFirst[E E']. *** appenleft


*** reduction for sets
eq set(L E L' E L'')=  set(L E L' L'').

eq 'neg[int(I)]        = int(-(I)). ***

eq 'fst[pair(E, E')]   = E .
eq 'scd[pair(E, E')]   = E' .

*** bool-functions
eq 'not[bool(false)]   = bool(true).
eq 'not[bool(true )]   = bool(false).
eq 'or[bool(true)   E] = bool(true).
eq 'or[bool(false)  E] =  E .
eq 'and[bool(false) E] = bool(false).
eq 'and[bool(true)  E] =  E .

*** equal-func
eq 'equal[D D']              = bool(D == D') .

*** less-func
eq 'less  [int(I) int(I')] = bool(I < I') .
eq 'lessEq[int(I) int(I')] = bool(I <= I') .
eq 'less  [str(S) str(S')] = bool(S < S') .
eq 'lessEq[str(S) str(S')] = bool(S <= S') .

*** list-functions
eq 'head[list(nil)]        = null .
eq 'head[list(E L)]        = E .
eq 'last[list(nil)]        = null .
eq 'last[list(L E)]        = E .
eq 'rest[list(nil)]        = null .
eq 'rest[list(L E)]        = list(L).
eq 'tail[list(nil)]        = null .
eq 'tail[list(E L)]        = list(L).

eq 'length[list(nil)]   = int(0) .
eq 'length[set(nil)]    = int(0) . *** ARE
```

```
140
141 eq 'length[list(E L)]   = int(1) + ('length[list(L)]) .
142 eq 'length[set(E L)]    = int(1) + ('length[set(L)]) .
143
144 eq 'isempty[list(nil)]  = bool(true) .
145 eq 'isempty[list(E L)]  = bool(false) .
146
147 eq 'remove[list(E)   E']  = if  E == E' then list(nil) else list(E) fi .
148 eq 'remove[list(nil) E ]  = list(nil) .
149 eq 'remove[list(E L) E']  = if E == E' then 'remove[(list(L)) E']
150                             else 'plus[list(E) ('remove[(list(L)) E'])] fi .
151 eq 'remove[set(E)    E']  = if  E == E' then set(nil) else set(E) fi .
152 eq 'remove[set(nil)  E ]  = set(nil) .
153 eq 'remove[set(E L)  E']  = if E == E' then set(L)
154                             else 'plus[set(E) ('remove[(set(L)) E'])] fi .
155
156 eq 'appendLeft [list(L) E]  = list(E L) .
157 eq 'addFirst   [list(L) E]  = list(E L) .   *** alias
158 eq 'appendRight[list(L) E]  = list(L E) .
159 eq 'add[list(L)    E]       = list(L E) .    *** alias
160 eq 'add[set (L)    E]       = set (L E) .
161
162 eq 'has[list(nil) E ]           = bool(false) .
163 eq 'has[list(E L) E']       = if (E' == E) then bool(true)
164                               else 'has[list(L) E'] fi .
165 eq 'has[set(nil)  E ]           = bool(false) .
166 eq 'has[set(E L)  E']       = if (E' == E) then bool(true)
167                               else 'has[set(L) E'] fi .
168 ***index starts at 1
169 eq 'after[list(nil) int(N)] = null .
170 eq 'after[list(E L) int(0)] = list(E L) .
171 eq 'after[list(E L) int(N)] = 'after[(list(L)) (int(N) - int(1))].
172
173 eq 'index[(list(L))   int(0)] = null .
174 eq 'index[(list(nil)) int(N)] = null .
175 eq 'index[(list(E L)) int(N)] = if (N == 1) then E else
176                                  'index[(list(L)) int(N - 1)] fi .
177
178 eq 'begwith[list(E) E']    = bool('head[(list(E))] == E') .
179 eq 'begwith[list(E L) E']  = bool('head[(list(E L))] == E') .
180 eq 'endwith[list(E) E']    = bool('last[(list(E))] == E') .
181 eq 'endwith[list(E L) E']  = bool('last[(list(L))] == E') .
182
183 eq 'plus  [list(L) list(L')]   = list(L L').
184 eq 'plus  [set(L) set(L')]     = set(L L').
185 eq 'plus  [str(S) str(S')]     = str(S + S').
186 eq 'plus  [int(I) int(I')]     = int(I + I').
187 eq 'plus  [float(F) float(F')] = float(F + F').
188
189 eq 'minus [int(I) int(I')]     = int(I - I').
190 eq 'minus [float(F) float(F')] = float(F - F').
191
192 eq 'times [int(I) int(I')]     = int(I * I').
193 eq 'times [float(F) float(F')] = float(F * F').
194
```

```
eq 'div   [int(I) int(I')]     = int(I quo I').
eq 'div   [float(F) float(F')] = float(F / F').

eq  'set[list(L)] = set(L).
eq  'list[set(L)] = list(L).

endfm

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

*** Bound variables ***
fmod LIST-QID-VAL is
protecting DATATYPER .

sorts    BndVar Subst InitVar InitSubst .   *** Subst: non-repetitive list of
*** BndVar.  InitVar InitSubst has Expr where BndVar Subst has Data.
subsorts BndVar < Subst InitVar < InitSubst .

op _:_ : Aid Data    -> BndVar  [ctor format (! o o o) ] .
op _:_ : Aid Expr    -> InitVar [ctor format (! o o o) ] .
op no  :             -> Subst   [ctor] .
op _,_ : Subst Subst -> Subst   [ctor assoc id: no] .
op _,_ : InitSubst InitSubst -> InitSubst [ctor assoc id: no] .

*** Remove multiple values of same variable in a Susbt
var  A : Aid .
var  L : InitSubst .
vars D D' : Expr .
eq   (A : D), L, (A : D') = if D' == null then (A : D), L else (A : D'), L fi .

endfm

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

*** CREOL guards ***
fmod GUARDS is
protecting LIST-QID-VAL .

sorts     Guard Wait Return ExtGuard .
subsorts  Return Expr Wait < Guard  < ExtGuard .

op wait :   -> Wait [ctor] .           *** suspension
op _? : Qid -> Return [ctor] .         *** reply guard
op _? : Nat -> Return [ctor] .         *** low level reply guard
op _?G(_) : Qid List -> ExtGuard .     *** reply guard with side effect
op nothing : -> Guard [ctor] .
op _&_ : Guard Guard -> Guard [ctor id: nothing assoc comm prec 55] .
op _&_ : Guard ExtGuard -> ExtGuard [ctor ditto] .

*** reduction of guards to normalform: [wait &]? [bool &]? return ***
vars E  E' : Expr .
eq   wait & wait = wait .
eq   E & E'      = E and E' .
endfm
```

```
250 ***********************************************************
251
252 *** CREOL program code ***
253 fmod PROG is
254 protecting GUARDS .
255
256 sorts   Stm Prog ProgList Process MProg . *** Stm is basic statem
257 *** without a leading guard.
258 subsort Stm < Prog < ProgList .
259
260 *** Cid is class identifier, Mid method name
261 sorts   Cid Mid .
262 subsort Qid < Cid Mid .
263 subsort Qid < Aid < Mid .
264
265 op _._ : Expr Qid -> Mid [ctor] . *** remote call *** ARE: Qid var tidligere Cid.
266 *** 2.argument er metodenavn, ikke klasseID.
267
268 op _@_ : Qid  Cid -> Aid [ctor] . *** attribute qualified by class name (local)
269 op empty :                        ->   ProgList [ctor] .
270 op _;_   : ProgList     ProgList ->   ProgList [ctor assoc id: empty] .
271
272
273
274 **** ARE start
275 sorts SyncMerge Merge Nondet .
276 subsorts SyncMerge Merge Nondet < Prog .
277
278 op _&_    : ProgList ProgList     -> SyncMerge [ctor assoc comm] .
279 op _|||_  : ProgList ProgList     -> Merge [ctor assoc comm] .
280 op _[]_   : ProgList ProgList     -> Nondet [ctor assoc comm] .
281 **** ARE slutt
282
283
284
285 *** CREOL program syntax
286 op _:=_   : AidList List -> Stm [ctor]. ***simultaneous assignment, same length
287 op _:= new_(_) : Aid Cid List -> Stm [ctor] . *** object creation
288 op if_th_el_fi : Expr ProgList ProgList -> Stm [ctor] .
289 op if_th_fi    : Expr ProgList          -> Stm .
290 op while_do_od : Expr ProgList          -> Stm [ctor] .
291 op   _(_;_) : Mid List List -> Stm [ctor] .  *** sync. call (with reply)
292 op  !_(_)   : Mid List      -> Stm [ctor] .  *** async. call (without label)
293 op _!_(_)   : Qid Mid List  -> Stm [ctor] .  *** async. call (with label)
294 op _?(_)    : Qid List      -> Stm [ctor] .  *** async. reply statement
295 op end      : List          -> Stm [ctor] .  *** method return
296 op continue : Nat           -> Stm [ctor] .  *** sync. termination
297 ***op _[]_     : ProgList ProgList -> Prog [ctor assoc].
298  *** non-determ. prog. *** ARE har kommentert ut
299 ***op _|||_    : ProgList ProgList -> Prog [ctor assoc].
300  *** interleaved progs. *** ARE har kommentert ut
301 op await_   : Guard             -> Prog [ctor] .
302 op await_   : ExtGuard          -> Prog .    *** reply guards with side effect
303
304 *** Low level CMC mechanism for sync. calls (dummy labels using Nat)
```

```
op _?(_)  : Nat List -> Stm [ctor].            *** sync. reply statement

subsort Process < MProg .    *** Multiset of Processes
op _:_ : MProg MProg -> MProg [ctor assoc comm id: none] .

*** A Process is a pair of Prog and bound variables ***
op none : -> Process [ctor] .
op _,_ : ProgList Subst -> Process [ctor] .

var EL : List . var P : ProgList .
eq (nil := EL)    ; empty = empty .
eq (await nothing); empty = empty .
eq (empty, no)   = (none).Process .
eq (empty ||| P) = P .
endfm

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

*** CREOL classes ***
fmod CLASS is
protecting PROG .

sorts    Class Mtd MMtd ClVs Inh InhList . *** inheritance list
subsorts Nil Inh < InhList .

qop  _#_    : Cid Nat   -> ClVs [ctor] . *** class with version
op  _[_]   : ClVs List -> Inh  [ctor] . ***   and with parameters.
*** use in objects like <O:OB|CL: C#V, ..>
op  __   : InhList InhList -> InhList [ctor assoc id: nil] .

var Ih : Inh . var S : InhList .
eq  Ih S Ih = Ih S .  *** inherit same thing twice, means once

op <_: Mtdname | Latt:_, Code:_> : Qid Subst ProgList -> Mtd [ctor
 format (m! om m m m m m g m m m! on)] .

subsort Mtd < MMtd .    *** Multiset of methods

op none : -> MMtd [ctor] .
op _*_  : MMtd MMtd -> MMtd [ctor assoc comm id: none] .
op _+_  : MMtd MMtd -> MMtd . *** method redefinition/addition

op <_: Cl | Vs:_, Inh:_, Att:_, Mtds:_, Ocnt:_> :
    Cid Nat InhList InitSubst MMtd Nat -> Class [ctor format (nb! b! ob b b
         b o g  b o g  b o g  b o g  b o  b! on )] .

endfm

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

*** CREOL objects ***
fmod OBJECT is
protecting CLASS .

sort Object .
```

```
360
361 op <_: Ob | Cl:_, Pr:_, PrQ:_, Lvar:_, Att:_, Lcnt:_> :
362 Oid ClVs ProgList MProg Subst Subst Nat -> Object [ctor
363   format (nr! r! ob r r  or b g  r r g  r m g  r o g  r o g  r o r! no )] .
364 endfm
365
366 *** CREOL messages and queues ***
367 fmod COMMUNICATION is
368 protecting DATA .
369 pr PROG .
370
371 sort NatS . *** list of nats
372
373 sort Msg MMsg Kid Queue .
374 subsort Msg < MMsg .
375 subsort Nat < NatS .
376
377 op none : -> MMsg [ctor] .
378 op _+_ : MMsg MMsg -> MMsg [ctor assoc comm id: none] .
379
380 *** INVOCATION and REPLY
381 op invoc(_,_,_) : Oid Mid DataList -> Msg [ctor format (rg o o o o o o o no)] .
382 op comp(_) : DataList -> Msg [ctor format (rg o o o no)] .
383 op error : String -> Msg [ctor] .                          *** error
384 op warning : String -> Msg [ctor] .                        *** warning
385
386 *** message queue
387 op empty :         -> NatS  [ctor].
388 op _;_ : NatS NatS -> NatS [ctor assoc id: empty] .
389
390 op <_: Qu | Ev:_, Keep:_> : Oid MMsg NatS -> Queue
391                           [format (nm! m! om m m  m o m  m o m! no)] .
392
393 endfm
394
395 ***********************************************************
396
397 *** STATE CONFIGURATION ***
398 fmod CONFIG is
399 protecting OBJECT .
400 protecting COMMUNICATION .
401
402 sort Configuration .
403
404 subsorts Object MMsg Queue Class < Configuration .
405
406 op none : -> Configuration [ctor] .
407 op __ : Configuration Configuration -> Configuration
408                           [ctor assoc comm id: none] .
409
410 endfm
411
412 ******************** AUXILIARY FUNCTIONS ******************************
413
414 fmod FUNKSJONER is
```

```
pr COMMUNICATION .
pr OBJECT .

*** Queue-function ***
op inqueue  : Nat MMsg -> Bool . *** checks if  Msg is in the queue

*** Class/method functions ***
op get      : Qid MMtd DataList -> Process .  *** fetches pair (code, vars)
op _in_     : Qid MMtd -> Bool .  *** checks if Q is a declared method

*** VarList functions ***
op val      : Aid Subst -> Data . *** fetches value of prog. var.
op _in_     : Aid Subst -> Bool . *** checks occurence in list
op evalList : List Subst -> DataList . *** maps list to values
op eval            : Expr Subst -> Data .     *** evaluate expression
op evalB    : Expr Subst -> Bool .
op evalI    : Expr Subst -> Int .

op enabled  : ProgList Subst MMsg -> Bool . *** eval initial guard
op ready    : ProgList Subst MMsg -> Bool . *** no initial waiting?
op assign   : InitSubst List  -> InitSubst . *** parameter substitution

*** variables
vars A A'     : Aid .
vars Q Q' R   : Qid .
vars O O' O'' : Oid .
vars L L'     : Subst .
vars IL       : InitSubst .
vars D D'     : Data .
vars DL DL'   : DataList .
vars Str Str' : String .
vars G G'     : Guard .
vars SP SP'   : Prog .
vars P P' P'' : ProgList .
var  St       : Stm .
var  MP       : MProg .
vars I J      : List .
var  MM       : MMsg .
vars X X'     : Expr .
vars B B'     : Bool .
vars N N'     : Nat .
vars C C'     : Int .
var  MTD      : Mtd .
vars MMTD MMTD' : MMtd .

eq A in no            = false .
eq A in ((A' : D), L) = if A == A' then true else (A in L) fi .

eq val(A,((A' : D),L))= if A == A' then D else val(A, L) fi .

eq evalI(X,L)         = eval(X,L) asInt .
eq evalB(X,L)         = eval(X,L) asBool .
*** evaluates to Data ***

eq eval(null, L)      = null .
```

```
eq eval(A, L)         = val(A, L) .

eq eval(pair(X,X'),L) = pair(eval(X,L),eval(X',L)) .
eq eval(list(J), L)   = list(evalList(J, L)) .
eq evalList(nil, L)   = nil .
eq evalList(X I, L)   = eval(X, L) evalList(I, L) .
eq eval(Q[I], L)      = Q[evalList(I, L)] .
eq eval(D, L)         = D [owise] .

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

*** test of guard by ENABLED
eq enabled(SP ; SP' ; P,   L, MM) = enabled(SP, L, MM) .
eq enabled(P []  P',       L, MM) = enabled(P, L, MM) or enabled(P', L, MM) .
eq enabled(P ||| P',       L, MM) = enabled(P, L, MM) or enabled(P', L, MM) .
eq enabled(await(wait & G),L, MM) = false . *** Note: no wait in PrQ!
eq enabled(await (X & G),  L, MM) = evalB(X, L) and enabled(await G, L, MM) .
eq enabled(await((Q ?)& G),L, MM) = inqueue(evalI(Q, L), MM) and
                                    enabled(await G, L, MM) .
eq enabled(await((N ?)& G),L, MM) = inqueue(N, MM) and
                                    enabled(await G, L, MM) .
eq enabled(empty,          L, MM) = true .
eq enabled(St,             L, MM) = true .

*** test of guard by READY: no active or passive wait
eq ready(SP ; SP' ; P, L, MM) = ready(SP, L, MM) .
eq ready(Q ?(I),       L, MM) = inqueue(evalI(Q, L), MM) .
eq ready(N ?(I),       L, MM) = inqueue(N, MM) .
eq ready(P []  P',     L, MM) = ready(P, L, MM) or ready(P', L, MM) .
eq ready(P ||| P',     L, MM) = ready(P, L, MM) or ready(P', L, MM) .
eq ready(SP,           L, MM) = enabled(SP, L, MM) [owise].


*** fetches code and variables

eq get(Q, none, DL) = none . *** No method/run means empty method/run.
eq get(Q, < R : Mtdname | Latt: L, Code: P > * MMTD, D D' DL) =
    if Q == R then (P, (('caller : D), ('label : D'), assign(L, DL)))
              else get(Q, MMTD, D D' DL) fi .
eq Q in none = false .
eq Q in (< R : Mtdname | Latt: L, Code: P > * MMTD) =
            if Q == R then true else Q in MMTD fi .
eq none + MMTD =   MMTD .
eq (< R : Mtdname | Latt: L, Code: P > * MMTD) + MMTD' = if R in MMTD'
  then none else < R : Mtdname | Latt: L, Code: P > fi * (MMTD + MMTD').

*** makes substitutions
eq assign(IL, nil) = IL .
eq assign(no, I)   = no . *** not really needed (when type correct)
eq assign(((A : X), IL), X' I) = (A : X'), assign(IL, I) .

*** transform wait guards when active code is suspended. remove all waits.
```

```
525 op clear : ProgList -> ProgList .
526 *** op clear : Prog     -> Prog  .
527 op clear : Guard    -> Guard .
528
529 eq clear(empty)     = empty .
530 eq clear(SP ; SP' ; P) = clear(SP) ; SP' ; P .
531 eq clear(P []  P')  = clear(P) [] clear(P').
532 eq clear(P ||| P')  = clear(P) ||| clear(P').
533 eq clear(await G)   = await( clear(G) ).
534 eq clear(St)        = St .
535
536 eq clear(nothing)   = nothing .
537 eq clear(wait & G)  = clear(G) .
538 eq clear(X & G)     = X & clear(G).
539 eq clear((Q ?) & G) = (Q ?) & clear(G).
540 eq clear((N ?) & G) = (N ?) & clear(G).
541
542 ***    EXPANSION OF LANGUAGE MACROS:
543 eq await (G' & Q ?G(I)) = await (G' & Q ?); (Q ?(I)) .
544 eq if X th P fi     = (if X th P el empty fi) .
545
546 endfm
547
548 ************************************************************
549
550 *** the CREOL INTERPRETER ***
551
552 mod INTERPRET is
553 pr CONFIG .
554 pr FUNKSJONER .
555 pr CONVERSION .
556
557 op new_(_) : Cid List -> Msg . *** initialise program message
558
559 vars O O'  : Oid .
560 vars B B'  : Process .
561 vars C C'  : Cid .
562 var  Q     : Qid .
563 vars X Y   : Aid .
564 var  D D'  : Data .
565 vars W W'  : MProg .
566 vars I J K : List .
567 var  M     : MMsg .
568 var  Ms    : Msg .
569 var  Mt Mt' : MMtd .
570 var  MM    : Mid .
571 vars E E' OE : Expr .   *** OE is oid-expr.
572 var  F F'  : Nat .       *** Ocnt:-values
573 vars N N'  : Nat .
574 vars G G'  : Guard .
575 var  AL    : AidList .
576 vars DL    : DataList .
577 vars IA IA' : InitSubst .  *** list of initialized attribute declaratins
578 vars L L' A A' : Subst .   *** local var and attribute var-lists, respec.
579 vars P P' R R' : ProgList .
```

```
var  SP    : Prog .
var  St    : Stm .
vars H H'  : NatS .
vars S S' S'' : InhList .    *** list of (parameterized) superclasses
vars V V' V'' : Nat .        *** version number

 **********VIRTUAL BINDING of METHODS With MULTIPLE INHERITANCE

op bindMtd : Oid Qid List InhList -> Msg [ctor] . ***Bind method request
*** Given: callee method params (list of classes to look in)
op boundMtd : Oid Process         -> Msg [ctor] . *** binding result
*** CONSIDER the call O.Q(I).
***  bindMtd(O,Q,I,C S) try to find Q in class C or superclasses, then in S
***  boundMtd(O,Mt)  is the result.

ceq bindMtd(O,Q,D D' I,nil) =
boundMtd(O,(end(nil),('caller : D), ('label : D')))
if Q == 'run .

eq bindMtd(O,Q,I,(C # V'[J]) S')
< C : Cl | Vs: V,Inh: S,Att: IA,Mtds: Mt,Ocnt: F >
=
if Q in Mt then boundMtd(O,get(Q,Mt,I))
else bindMtd(O,Q,I,S S') fi
< C : Cl | Vs: V,Inh: S,Att: IA,Mtds: Mt,Ocnt: F > .

**********ATTRIBUTE inheritance with multiple inheritance
*** CMC ensures that all attributes names are (globally) different

op findAttr  : Oid InhList InitSubst -> Msg [ctor] .
  *** collect attributes
op foundAttr : Oid InitSubst         -> Msg [ctor] .
  *** resulting Subst
*** look in InhList, collect attributes in Subst, give result to Oid
eq findAttr(O, nil, IA) = foundAttr(O, evalSS(('this : O),IA),no)) .
  *** collection completed.

eq findAttr(O,(C # V'[I]) S', IA )
                  < C : Cl | Vs: V,Inh: S, Att: IA', Mtds: Mt, Ocnt: F >
 = findAttr(O,(S S'), assign(IA',I), IA )
                  < C : Cl | Vs: V,Inh: S, Att: IA', Mtds: Mt, Ocnt: F > .

op evalSS : InitSubst Subst -> Subst . *** eval Subst Sequentially
eq evalSS( no, IA )        = no .
eq evalSS(((X : E),IA), A) =
   (X : eval(E,A)), evalSS(IA, (A,(X : eval(E,A)))).

*************************** Names of new objects ********************
op newId : Cid Nat -> Oid .
eq newId(C, F)    = ob(qid(string(C) + string(F))) .
***** some code to make string/qid total
op string : Nat    -> String  .
op string : Cid    -> String [ditto] .
op string : Oid    -> String [ditto] .
op qid    : Qid    -> Qid .
```

```
op qid    : String -> Qid [ditto] .
eq string(F)      = string(F,10) .
eq qid(Q)         = Q .
eq qid(string(Q)) = Q .
eq qid(string(C)) = qid(C) .

********************** INTERPRETER RULES ***************************

*** assign ***

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

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

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

***  Object Creation: by new-msg and by new-statements
rl [new-start-req] :  *** Note: simply starting run
(new C (DL))
< C : Cl | Vs: V,Inh: S, Att: IA, Mtds: Mt, Ocnt: F >
=>
< C : Cl | Vs: V,Inh: S, Att: IA, Mtds: Mt, Ocnt: (F + 1) >
< newId(C,F): Ob | Cl: C # V, Pr: 'run (nil ; nil),
     PrQ: none, Lvar: no, Att: no, Lcnt: 1 >
< newId(C, F): Qu | Ev: none, Keep: empty >
findAttr(newId(C,F), C # V[DL], no ).

eq *** rl [new-start] :
foundAttr(O, A)
```

```
690 < O : Ob | Cl: C # V, Pr: P, PrQ: W, Lvar: L, Att: A', Lcnt: N >
691 =
692 < O : Ob | Cl: C # V, Pr: P, PrQ: W, Lvar: L, Att: A, Lcnt: N > . *** A',
693
694 rl [new-req] :
695 < O : Ob | Cl: C # V, Pr: (X := new C'(I)); P, PrQ: W, Lvar: L,
696  Att: A, Lcnt: N >
697 < C' : Cl | Vs: V',Inh: S, Att: IA, Mtds: Mt, Ocnt: F >
698 =>
699 < O : Ob | Cl: C # V, Pr: (X := newId(C', F)); P, PrQ: W,
700  Lvar: L, Att: A, Lcnt: N >
701 < C' : Cl | Vs: V',Inh: S, Att: IA, Mtds: Mt, Ocnt: (F + 1) >
702 < newId(C',F): Ob | Cl: C' # V', Pr: 'run (nil ; nil),
703     PrQ: none, Lvar: no, Att: no, Lcnt: 1 >    *** Att: IA
704 < newId(C',F): Qu | Ev: none, Keep: empty >
705 findAttr(newId(C',F), C' # V [evalList(I,(A,L))], no ).
706
707 *** Non-deterministic choice ***
708
709 *** ARE har kommentert ut nondet-regler.
710
711 *** crl [nondet-p1] :
712 *** < O : Ob | Cl: C # V, Pr: (P [] P'); R, PrQ: W, Lvar: L, Att: A,
713 ***   Lcnt: N >
714 *** < O : Qu | Ev: M, Keep: H >
715 *** =>
716 *** < O : Ob | Cl: C # V, Pr: P ; R, PrQ: W, Lvar: L, Att: A, Lcnt: N >
717 *** < O : Qu | Ev: M, Keep: H >
718 *** if ready(P, (L , A), M) . *** if evalG(P, (L , A), M) .
719
720 *** crl [nondet-p2] :
721 *** < O : Ob | Cl: C # V, Pr: (P [] P'); R, PrQ: W, Lvar: L, Att: A,
722 ***   Lcnt: N >
723 *** < O : Qu | Ev: M, Keep: H >
724 *** =>
725 *** < O : Ob | Cl: C # V, Pr: P' ; R, PrQ: W, Lvar: L, Att: A, Lcnt: N >
726 *** < O : Qu | Ev: M, Keep: H >
727 *** if  ready(P', (L , A), M) . *** if evalG(P', (L , A), M) .
728
729 *** Merge ***
730
731 *** ARE har kommentert ut merge-regler.
732
733 *** crl [merge-first] :
734 *** < O : Ob | Cl: C # V, Pr:((SP ; P)||| P'); R, PrQ: W, Lvar: L,
735 ***   Att: A, Lcnt: N >
736 *** < O : Qu | Ev: M, Keep: H >
737 *** =>
738 *** < O : Qu | Ev: M, Keep: H >
739 *** < O : Ob | Cl: C # V, Pr: SP ; (P ||| P'); R, PrQ: W, Lvar: L,
740 ***   Att: A, Lcnt: N >
741 *** if ready(SP,(L,A), M) .
742
743 *** crl [merge-second] :
744 *** < O : Ob | Cl: C # V, Pr: (P ||| P'); R, PrQ: W, Lvar: L,
```

```
*** Att: A, Lcnt: N >
*** < O : Qu | Ev: M, Keep: H >
*** =>
*** < O : Qu | Ev: M, Keep: H >
*** < O : Ob | Cl: C # V, Pr: (P' ||| P); R, PrQ: W, Lvar: L,
*** Att: A, Lcnt: N >
*** if ready(P',(L,A), M) and not ready(P,(L,A), M).


*** Suspension ***

crl [suspend] : *** all kinds of code P
< O : Ob | Cl: C # V, Pr: P, PrQ: W, Lvar: L, Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
=>
< O : Ob | Cl: C # V, Pr: empty, PrQ: W :(clear(P),L), Lvar: no,
  Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
if not enabled(P, (L , A), M) .

*** Guards ***

rl [boolguard] :
< O : Ob | Cl: C # V, Pr: await E ; P, PrQ: W, Lvar: L, Att: A, Lcnt: N >
=>
if evalB(E, (A , L)) then
< O : Ob | Cl: C # V, Pr: P , PrQ: W, Lvar: L, Att: A, Lcnt: N >
else
< O : Ob | Cl: C # V, Pr: empty, PrQ: W : (await E ; P, L),
 Lvar: no, Att: A, Lcnt: N >
fi .

*** Reduction of label to number in guard
eq < O : Ob | Cl: C # V, Pr: await (Q ? & G); P, PrQ: W, Lvar: L,
  Att: A, Lcnt: N' >
=
  < O : Ob | Cl: C # V, Pr: await (evalI(Q, L)? & G); P , PrQ: W, Lvar: L,
  Att: A, Lcnt: N' > .

rl [replyguard-inQ] : *** removal of reply guard
< O : Ob | Cl: C # V, Pr: await(N ? & G); P, PrQ: W, Lvar: L,
  Att: A, Lcnt: N' >
< O : Qu | Ev: M + comp(O int(N) DL), Keep: H >
=>
< O : Ob | Cl: C # V, Pr: await G ; P, PrQ: W, Lvar: L, Att: A, Lcnt: N' >
< O : Qu | Ev: M + comp(O int(N) DL), Keep: H > .

*** Evaluate guards in suspended processes ***

crl [PrQ-enabled] : ***evaluation
< O : Ob | Cl: C # V, Pr: empty, PrQ: (P', L') : W, Lvar: L, Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
=>
< O : Ob | Cl: C # V, Pr: P', PrQ: W, Lvar: L', Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
```

```
if  enabled(P', (A , L'), M). *** evalG(P', (L' , A), M) .

*** rl [replyguard-susp] :
eq  < O : Ob | Cl: C # V, Pr: P', PrQ: ((await (N ? & G); P), L'): W, Lvar: L,
  Att: A, Lcnt: N' >    < O : Qu | Ev: M + comp(O int(N) DL), Keep: H >
= < O : Ob | Cl: C # V, Pr: P', PrQ: ((await  G ; P), L'): W, Lvar: L,
  Att: A, Lcnt: N' >    < O : Qu | Ev: M + comp(O int(N) DL), Keep: H > .


*** METHOD CALLS ***

eq   ! Q(I) =   ! 'this . Q(I) .

*** receive invocation message ***
rl [receive-call-req] :
< O : Ob | Cl: C # V, Pr: P, PrQ: W, Lvar: L, Att: A, Lcnt: N' >
< O : Qu | Ev: M + invoc(O, Q, DL), Keep: H >
=>
< O : Ob | Cl: C # V, Pr: P, PrQ: W, Lvar: L, Att: A, Lcnt: N' >
< O : Qu | Ev: M, Keep: H >  bindMtd(O, Q, DL, C # V[nil]) .

rl [receive-call-req] :
< O : Qu | Ev: M + invoc(O, Q @ C, DL), Keep: H >
=>
< O : Qu | Ev: M, Keep: H >  bindMtd(O, Q, DL, C # 0[nil]) .
 *** dont use version number here (therefore 0 as default).

eq *** rl [receive-call-bound] :
boundMtd(O, (P', L'))
< O : Ob | Cl: C # V, Pr: P, PrQ: W, Lvar: L, Att: A, Lcnt: N >
=
< O : Ob | Cl: C # V, Pr: P, PrQ:(clear(P'),L'): W, Lvar: L, Att: A, Lcnt: N > .

rl [local-reentrace] :
< O : Ob | Cl: C # V, Pr: (N ? (J)); P,
  PrQ:(P',('caller : O),('label : int(N)),L'): W, Lvar: L, Att: A, Lcnt: N' >
=>
< O : Ob | Cl: C # V, Pr: P' ; continue(N), PrQ:(await N ? ;(N ?(J)); P, L): W,
           Lvar: ('caller : O), ('label : int(N)), L', Att: A, Lcnt: N' > .

rl [continue] :
< O : Ob | Cl: C # V, Pr: continue(N), PrQ: (((N ?(J)); P), L') : W, Lvar: L,
  Att: A, Lcnt: N' >
=>
< O : Ob | Cl: C # V, Pr:((N ?(J)) ; P), PrQ: W, Lvar: L', Att: A, Lcnt: N' > .

rl [local-async-qualified-req] :
< O : Ob | Cl: C # V, Pr: ( ! Q @ C'(I)); P,PrQ: W,Lvar: L,Att: A,Lcnt: N >
=>
< O : Ob | Cl: C # V, Pr: P, PrQ: W, Lvar: L, Att: A, Lcnt: N + 1 >
invoc(O, Q @ C', (O int(N) evalList(I, (A, L))) ) .

*** REMOTE METHOD CALLS ***
eq < O : Ob | Cl: C # V, Pr: (Q ! MM(I)); P, PrQ: W, Lvar: L, Att: A, Lcnt: N >
   < O : Qu | Ev: M, Keep: H >
```

```
=  < O : Ob | Cl: C # V, Pr: (Q := int(N)); (! MM (I)); P, PrQ: W, Lvar: L,
               Att: A, Lcnt: N >    < O : Qu | Ev: M, Keep: H ; N > .

rl [remote-async-reply] :
< O : Ob | Cl: C # V, Pr: ( ! OE . Q(I)); P, PrQ: W, Lvar: L, Att: A, Lcnt: N >
=>
< O : Ob | Cl: C # V, Pr: P, PrQ: W, Lvar: L, Att: A, Lcnt: N + 1 >
invoc(eval(OE, (A, L)), Q, (O int(N) evalList(I, (A, L)))) .

*** Reduce sync. call  to async. call with reply
eq < O : Ob | Cl: C # V, Pr: (MM(I ; J)); P, PrQ: W, Lvar: L, Att: A, Lcnt: N >
   < O : Qu | Ev: M, Keep: H >
  =
   < O : Ob | Cl: C # V, Pr: (! MM(I); (N ?(J)); P), PrQ: W, Lvar: L, Att: A,
       Lcnt: N >  < O : Qu | Ev: M, Keep: H ; N > .

*** emit reply message ***
rl [reply] :
< O : Ob | Cl: C # V, Pr: (end(J)); P, PrQ: W, Lvar: L, Att: A, Lcnt: N >
=>
< O : Ob | Cl: C # V, Pr: P, PrQ: W, Lvar: L, Att: A, Lcnt: N >
comp(evalList('caller 'label J,(A, L))) .

*** reduce label
eq
< O : Ob | Cl: C # V, Pr: (Q ?(J)); P, PrQ: W, Lvar: L, Att: A, Lcnt: N >
=
< O : Ob | Cl: C # V, Pr: (evalI(Q,L)?(J)); P, PrQ: W, Lvar: L, Att: A, Lcnt: N > .

*** blocking reply sentence ***
eq ***rl [block-sync-reply] :
  < O : Ob | Cl: C # V, Pr: (N ? (J)); P, PrQ: W, Lvar: L, Att: A, Lcnt: N' >
  < O : Qu | Ev: M + comp(O int(N) DL), Keep: H >
=
  < O : Ob | Cl: C # V, Pr: (J := DL); P, PrQ: W, Lvar: L, Att: A, Lcnt: N' >
  < O : Qu | Ev: M, Keep: H > .

*** Transport rules: include new message in queue
rl [invoc-msg] :
< O : Qu | Ev: M, Keep: H > invoc(O, MM, DL)
=>
< O : Qu | Ev: M + invoc(O, MM, DL), Keep: H > .

rl [reply-msg] :
< O : Qu | Ev: M, Keep: H ; N ; H' > comp(O int(N) DL)
=>
< O : Qu | Ev: M + comp(O int(N) DL), Keep: H ; H' > .

op _in_ : Nat NatS -> Bool .
eq N in empty = false .
eq N in H ; N' = if N == N' then true else N in H fi .

crl [reply-msg] :
< O : Qu | Ev: M, Keep: H > comp(O int(N) DL)
=>
```

```
910 < O : Qu | Ev: M, Keep: H >
911 if N in H == false .
912
913 ************* dynamic class extension ***************************
914
915 ************* version control
916
917 ceq < C  : Cl | Vs: V, Inh: S (C' # V'[I]) S', Att: IA, Mtds: Mt, Ocnt: F >
918     < C' : Cl | Vs: V'', Inh: S'', Att: IA', Mtds: Mt', Ocnt: F' >
919 =   < C  : Cl | Vs: s V, Inh: S (C' # V''[I]) S', Att: IA, Mtds: Mt, Ocnt: F >
920     < C' : Cl | Vs: V'', Inh: S'', Att: IA', Mtds: Mt', Ocnt: F' >
921 if V'' > V' .
922
923 ************* add attributes
924 ceq < O : Ob | Cl: C # V, Pr: empty, PrQ: W, Lvar: L, Att: A, Lcnt: N >
925    < C  : Cl | Vs: V',Inh: S, Att: IA, Mtds: Mt, Ocnt: F >
926 =  < O : Ob | Cl: C # V', Pr: empty, PrQ: W, Lvar: L, Att: A, Lcnt: N >
927    < C  : Cl | Vs: V',Inh: S, Att: IA, Mtds: Mt, Ocnt: F >
928    findAttr(O, C # V'[nil], A) if V' > V  .
929
930 ceq < O : Ob | Cl: C # V, Pr: continue(N'), PrQ: W, Lvar: L, Att: A, Lcnt: N >
931    < C  : Cl | Vs: V',Inh: S, Att: IA, Mtds: Mt, Ocnt: F >
932 =  < O : Ob | Cl: C # V', Pr: continue(N'), PrQ: W, Lvar: L, Att: A, Lcnt: N >
933    < C  : Cl | Vs: V',Inh: S, Att: IA, Mtds: Mt, Ocnt: F >
934    findAttr(O, C # V'[nil], A) if V' > V  .
935
936 *************** class updates
937 op update : Class -> Msg [ctor] . *** class update
938
939 crl [class-update] :
940              < C : Cl | Vs: V ,Inh: S , Att: IA , Mtds: Mt , Ocnt: F  >
941       update(< C : Cl | Vs: V',Inh: S', Att: IA', Mtds: Mt', Ocnt: F' >)
942 =>
943    < C : Cl | Vs: s V, Inh: S S', Att:(IA,IA'), Mtds: Mt + Mt', Ocnt: F >
944 if F > F' . *** the if-test provides a course notion of time
945 *** update messages do not really need the Vs and Ocnt attributes.
946
947 endm
948 eof
```

Tillegg B

Samlet Maude-spesifikasjon av synkronisert fletting

Dette er Maude-koden til min utvidelse av interpreten. Interpreten inkluderes i denne modulen.

De første linjene inneholder nødvendige deklarasjoner og utvidelser av hjelpefunksjoner. Mekanismen for ekspansjon av metodekall finnes i linje 71-237. Resten av implementasjonen definerer kjøring av setninger med ekspanderte kall og flettepolicyen for delvis ordnet fletting.

```
1  in interpreter
2
3  mod SYNCMERGE is
4  including INTERPRET .
5
6  *** Husk at i interpreter.maude har vi følgende deklarasjoner:
7  *** sorts SyncMerge Merge Nondet .
8  *** subsorts SyncMerge Merge Nondet < Prog .
9  *** op _&_    : ProgList ProgList     -> SyncMerge [ctor assoc comm] .
10 *** op _|||_  : ProgList ProgList     -> Merge [ctor assoc comm] .
11 *** op _[]_   : ProgList ProgList     -> Nondet [ctor assoc comm] .
12
13 sorts Run RunMtd ProgElement .
14 subsorts Run RunMtd ProgElement < Prog .
15
16 op _///_     : ProgList ProgList -> Prog [ctor assoc] .
17 op insert    : Nat               -> Stm [ctor] .
18 op element   : Process           -> ProgElement [ctor] .
19 op awaitingElement  : Process    -> ProgElement [ctor] .
20 op runMtd    : Process           -> RunMtd [ctor] .
21 op run       : ProgList          -> Run [ctor] .
22 op toggleMergeOrder : ProgList   -> ProgList .
23
24 vars O   : Oid .
25 var  OE  : Expr .
26 vars C C' : Cid .
27 var  Q     : Qid .
28 vars W   : MProg .
29 var  M     : MMsg .
30 var  MM    : Mid .
31 vars N N' : Nat .
32 vars G G' : Guard .
33 vars DL    : DataList .
```

```
vars L L' A : Subst .
vars P P' R : ProgList .
var  SP    : Prog .
vars H   : NatS .
vars V  : Nat .

vars IN OUT : List .
var  PROC   : Process .
var  CV     : ClVs .
var  P''    : ProgList .
var  L''    : Subst .
var  SP'    : Prog .
var  N''    : Nat .

eq run(empty) = empty .

eq empty /// P = P .

eq clear(P & P')         = clear(P) & clear(P') .
eq clear(run(P))         = run(clear(P)) .
eq clear(runMtd((P, L))) = runMtd((clear(P), L)) .

eq enabled(runMtd((P, L')), L, M) = enabled(P, (L, L'), M) .
   *** overskriver lokal tilstand, beholder obj.attributter
eq enabled(P & P',           L, M) = enabled(P, L, M) and enabled(P', L, M) .
eq enabled(P /// P',        L, M) = enabled(P, L, M) or enabled(P', L, M) .
eq enabled(run(P),           L, M) = enabled(P, L, M) .

eq ready((P & P'),           L, M) = ready(P, L, M) and ready(P', L, M) .
eq ready(run(P),            L, M) = ready(P, L, M) .
eq ready(runMtd((P, L')),   L, M) = ready(P, (L, L'), M) .

eq toggleMergeOrder(P ||| P') = toggleMergeOrder(P) /// toggleMergeOrder(P') .
eq toggleMergeOrder(P /// P') = toggleMergeOrder(P) ||| toggleMergeOrder(P') .
eq toggleMergeOrder(P & P')   = toggleMergeOrder(P) /// toggleMergeOrder(P') .
eq toggleMergeOrder(P) = P [owise] .


**** ekstraksjon fra rot

ceq
< O : Ob | Cl: CV, Pr: (P & P'); R, PrQ: W, Lvar: L, Att: A, Lcnt: N >
=
< O : Ob | Cl: CV, Pr: element((P, L)); (insert(0) & P'); R, PrQ: W, Lvar: L,
  Att: A, Lcnt: N >
if not (P :: Run or P :: SyncMerge) .

ceq
< O : Ob | Cl: CV, Pr: (P [] P'); R, PrQ: W, Lvar: L, Att: A, Lcnt: N >
=
< O : Ob | Cl: CV, Pr: element((P, L)); (insert(0) [] P'); R, PrQ: W, Lvar: L,
  Att: A, Lcnt: N >
if not (P :: Run or P :: Nondet) .

ceq
```

```
< O : Ob | Cl: CV, Pr: ((SP ; P) ||| P'); R, PrQ: W, Lvar: L, Att: A, Lcnt: N >
=
if SP :: SyncMerge or SP :: Nondet
then < O : Ob | Cl: CV, Pr: element(((SP ; P), L)); (insert(O) ||| P'); R,
  PrQ: W, Lvar: L, Att: A, Lcnt: N >
else < O : Ob | Cl: CV, Pr: (run(SP ; P) ||| P'); R, PrQ: W, Lvar: L, Att: A,
  Lcnt: N >
fi
if not ((SP ; P) :: Run or (SP ; P) :: Merge) .


**** ekstraksjon fra element

ceq element(((P & P'); R, L)) =
  element((P, L)); awaitingElement(((insert(O) & P'); R, L))
if not (P :: SyncMerge or P :: Run) .

ceq element(((P [] P'); R, L)) =
  element((P, L)); awaitingElement(((insert(O) [] P'); R, L))
if not (P :: SyncMerge or P :: Run) .

ceq element((((SP ; P) ||| P'); R, L)) =
if SP :: SyncMerge or SP :: Nondet
then element(((SP ; P), L)); awaitingElement(((insert(O) ||| P'); R, L))
else element(((run(SP ; P) ||| P'); R, L))
fi
if not ((SP ; P) :: Merge or (SP ; P) :: Run) . *** P kan være empty


**** Binder metoder.

*** lokalt virtuelt kall med Oid
ceq < O : Ob | Cl: C # V, Pr: element((((OE . Q(IN ; OUT)); P), L')) ; R,
  PrQ: W, Lvar: L, Att: A, Lcnt: N >
=
< O : Ob | Cl: C # V, Pr: awaitingElement(((insert(N) ; (N ?(OUT)); P), L')) ; R,
  PrQ: W, Lvar: L, Att: A, Lcnt: N + 1 >
bindMtd(O, Q, (O int(N) evalList(IN, (A,L'))), C # V[nil])
if eval(OE, (A,L)) == O .

*** lokalt virtuelt kall uten Oid
eq < O : Ob | Cl: C # V, Pr: element((((Q(IN ; OUT)); P), L')) ; R, PrQ: W,
  Lvar: L, Att: A, Lcnt: N >
=
< O : Ob | Cl: C # V, Pr: awaitingElement(((insert(N) ; (N ?(OUT)); P), L')) ; R,
  PrQ: W, Lvar: L, Att: A, Lcnt: N + 1 >
bindMtd(O, Q, (O int(N) evalList(IN, (A,L'))), C # V[nil]) .

*** lokalt statisk kall (alle statiske kall er lokale)
eq < O : Ob | Cl: CV, Pr: element((((Q @ C (IN ; OUT)); P), L')) ; R, PrQ: W,
  Lvar: L, Att: A, Lcnt: N >
=
< O : Ob | Cl: CV, Pr: awaitingElement(((insert(N) ; (N ?(OUT)); P), L')) ; R,
  PrQ: W, Lvar: L, Att: A, Lcnt: N + 1 >
bindMtd(O, Q, (O int(N) evalList(IN, (A,L'))), C # O[nil]) .
```

```
144  *** Versjonsnr. brukes ikke. 0 er default.
145
146 -----
147
148 **** ALTERNATIV BINDINGSMEKANISME
149
150 *** *** disse tre ligningnene erstatter de tre ligningnene som legger
151 *** *** en bindMtd i konfigurasjonen:
152
153 *** *** lokalt virtuelt kall uten Oid
154 *** eq < O : Ob | Cl: CV, Pr: element((((Q(IN ; OUT)); P), L')) ; R, PrQ: W,
155 ***   Lvar: L, Att: A, Lcnt: N >
156 *** =
157 *** < O : Ob | Cl: CV, Pr: (! Q(IN)); awaitingElement(((insert(N) ;
158 ***   (N ?(OUT)); P), L')) ; R, PrQ: W, Lvar: L, Att: A, Lcnt: N > .
159
160 *** *** lokalt virtuelt kall med Oid
161 *** *** tar seg av this og alt annet som evaluerer til O
162 *** ceq < O : Ob | Cl: CV, Pr: element((((OE . Q(IN ; OUT)); P), L')) ; R,
163 ***   PrQ: W, Lvar: L, Att: A, Lcnt: N >
164 *** =
165 *** < O : Ob | Cl: CV, Pr: (! OE . Q(IN)); awaitingElement(((insert(N) ;
166 ***   (N ?(OUT)); P), L')) ; R, PrQ: W, Lvar: L, Att: A, Lcnt: N >
167 *** if eval(OE, (A,L)) == O .
168
169 *** *** lokalt statisk kall
170 *** eq < O : Ob | Cl: CV, Pr: element((((Q @ C (IN ; OUT)); P), L')) ; R,
171 ***   PrQ: W, Lvar: L, Att: A, Lcnt: N >
172 *** =
173 *** < O : Ob | Cl: CV, Pr: (!(Q @ C)(IN)); awaitingElement(((insert(N) ;
174 ***   (N ?(OUT)); P), L')) ; R, PrQ: W, Lvar: L, Att: A, Lcnt: N > .
175
176
177 *** *** Denne ligningen brukes istedenfor regelen [invoc-msg] når kallet
178 *** *** er lokalt. Dermed brukes ingen regler i bindingen av lokale
179 *** *** kall.
180 *** eq < O : Qu | Ev: M, Keep: H > invoc(O, MM, (O DL)) ***caller=O=mottager
181 *** =
182 *** < O : Qu | Ev: M + invoc(O, MM, (O DL)), Keep: H > .
183
184 *** *** Disse to ligningene brukes istedenfor de to regelene med
185 *** *** samme label rl[recieve-call-req] når kallet er lokalt
186 *** eq < O : Ob | Cl: C # V, Pr: P, PrQ: W, Lvar: L, Att: A, Lcnt: N' >
187 *** < O : Qu | Ev: M + invoc(O, Q, (O DL)), Keep: H >
188 *** =
189 *** < O : Ob | Cl: C # V, Pr: P, PrQ: W, Lvar: L, Att: A, Lcnt: N' >
190 *** < O : Qu | Ev: M, Keep: H >  bindMtd(O, Q, (O DL), C # V[nil]) .
191
192 *** eq < O : Qu | Ev: M + invoc(O, Q @ C, (O DL)), Keep: H >
193 *** =
194 *** < O : Qu | Ev: M, Keep: H >  bindMtd(O, Q, (O DL), C # O[nil]) .
195 *** dont use version number here (therefore 0 as default).
196
197 -----
198
```

```
*** Metoden hentes fra PrQ og legges fremst i Pr. Dette er grunnen til
*** vi trenger at insert tar et heltall som parameter.
eq < O : Ob | Cl: CV, Pr: awaitingElement(((insert(N') ; P), L'')) ; R,
  PrQ: (P',(('caller : O),('label : int(N')),L')) : W, Lvar: L, Att: A, Lcnt: N >
=
< O : Ob | Cl: CV, Pr:  element((P',(('caller : O),('label : int(N')),L')));
  awaitingElement(((insert(N') ; P), L'')) ; R, PrQ: W, Lvar: L, Att: A,
  Lcnt: N > .


*** **** folding

rl [fold-to-element-syncmerge]:
element((P, L)); awaitingElement(((insert(0) & P'); R, L)) =>
element(((run(P) & P'); R, L)) .

rl [fold-to-element-nondet]:
element((P, L)); awaitingElement(((insert(0) [] P'); R, L))  =>
element(((run(P) [] P'); R, L)) .

rl [fold-to-element-merge]:
element((P, L)); awaitingElement(((insert(0) ||| P'); R, L)) =>
element(((run(P) ||| P'); R, L)) .

crl [fold-to-element-call]:
element(PROC); awaitingElement((insert(N); P, L))
=>
element((runMtd(PROC); P, L))
if N > 0 .
*** merk: prosessen beholder sin lokale tilstand i PROC

rl [fold-to-root-syncmerge]:
element((P, L)); insert(0) & P'   => run(P) & P' .

rl [fold-to-root-nondet]:
element((P, L)); insert(0) [] P'  => run(P) [] P' .

rl [fold-to-root-merge]:
element((P, L)); insert(0) ||| P' => run(P) ||| P' .


*** **** Kjøring

crl [syncmerge] :
< O : Ob | Cl: CV, Pr: (P & P'); R, PrQ: W, Lvar: L, Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
=>
< O : Ob | Cl: CV, Pr: toggleMergeOrder(P & P') ; R, PrQ: W, Lvar: L,
  Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
if enabled((P & P'), (L, A), M) .

crl [nondet-choice] :
< O : Ob | Cl: CV, Pr: (P [] P'); R, PrQ: W, Lvar: L, Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
```

```
=>
< O : Ob | Cl: CV, Pr: P ; R, PrQ: W, Lvar: L, Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
if ready(P, (L, A), M) .

eq < O : Ob | Cl: CV, Pr: run(P) ; R, PrQ: W, Lvar: L, Att: A, Lcnt: N >
=
< O : Ob | Cl: CV, Pr: P ; R, PrQ: W, Lvar: L, Att: A, Lcnt: N > .

rl[run-mtd] :
< O : Ob | Cl: CV, Pr: runMtd((P, (('caller : O), ('label : int(N')), L')));
 (N'?(OUT)); R, PrQ: W, Lvar: L, Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
=>
< O : Ob | Cl: CV, Pr: P ; continue(N'), PrQ: (((N'?(OUT));R), L) : W,
  Lvar: (('caller : O), ('label : int(N')), L'), Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H ; N' > .

*** Må være regel for å bli brukt etter bindingsmaskineriet.
*** Merk at ikke spesifiserer hvordan enabled element skal finnes.
crl [merge-find-enabled] :
< O : Ob | Cl: CV, Pr: (run(P) ||| P'); R, PrQ: W, Lvar: L, Att: A, Lcnt: N  >
< O : Qu | Ev: M, Keep: H >
=>
< O : Ob | Cl: CV, Pr: (run(P) /// toggleMergeOrder(P')); R, PrQ: W, Lvar: L,
  Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
if enabled(run(P), (L,A), M) .

*** må være ligning for å bli brukt før rl[suspend]
ceq < O : Ob | Cl: CV, Pr: (run(SP ; P) /// P'); R, PrQ: W, Lvar: L,
  Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
=
< O : Ob | Cl: CV, Pr: toggleMergeOrder(run(SP ; P) /// P') ; R, PrQ: W, Lvar: L,
  Att: A, Lcnt: N >
< O : Qu | Ev: M, Keep: H >
if not (enabled(run(SP ; P), (L,A), M) or SP :: RunMtd) .
*** Alle metodekall er enabled i kontekst av |||. Hvis SP er et
*** metodekall, ønsker vi ikke å bruke denne ligningen. Derfor siste
*** del av if-testen.

crl [merge-dispatch] :
< O : Ob | Cl: CV, Pr: (run(SP ; P) /// P'); R, PrQ: W, Lvar: L, Att: A,
  Lcnt: N >
< O : Qu | Ev: M, Keep: H >
=>
< O : Ob | Cl: CV, Pr: SP ; (run(P) /// P'); R, PrQ: W, Lvar: L, Att: A,
  Lcnt: N >
< O : Qu | Ev: M, Keep: H >
if (not SP :: RunMtd) or enabled(SP, (L,A), M) .

rl [merge-dispatch-method] :
< O : Ob | Cl: CV, Pr: (run(runMtd(PROC); (N' ?(OUT)); P) /// P'); R,
  PrQ: W, Lvar: L, Att: A, Lcnt: N >
```

```
=>
< O : Ob | Cl: CV, Pr: runMtd(PROC); (N' ?(OUT));(run(P) /// P'); R,
  PrQ: W, Lvar: L, Att: A, Lcnt: N > .

endm
eof
```